Programming C++
Lecture Note 13
継承と動的結合を使う

Jie Huang

基底と派生

- 基底(base)クラスの定義

```
// Coreクラス、一般的な学生のデータを格納する
class Core {
public:
  Core();
  Core(std::istream&);
  std::string name() const;
  std::istream& read(istream&);
  double grade() const;
private:
  std::istream& read_common(std::istream&);
  std::string n;
  double midterm, final;
  std::vector<double> homework;
}
```

基底と派生

- 派生(derived)クラスの定義
  // Gradクラス、Coreクラスから派生し、卒研生のデータを格納する

```
class Grad: public Core {
public:
  Grad();
  Grad(std::istream&);
  double grade() const;
  std::istream& read(std::istream&);
private:
  double thesis;
};
```

派生における継承

- 派生クラスは基底クラスのpublicとprotected部分を継承する。しかしprivate部分にあるデータは継承できない。
- protected部分は外部からはアクセスできないが、派生クラスには継承される。

```
class Core {
public:
  Core();
  Core(std::istream&);
  std::string name() const;
  std::istream& read(istream&);
  double grade() const;
protected:
  std::istream& read_common(std::istream&);
  double midterm, final;
  std::vector<double> homework;
private:
  std::string n;
};
```

階層構造における関数の定義

- Coreクラスの関数

```
string Core::name() const { return n; }
double Core::grade() const {
  return ::grade(midterm, final, homework);
}
istream& Core::read_common(istream& in){
  in >> n >> midterm >> final;
  return in;
}
istream& Core::read(istream& in) {
  ream_common(in);
  read_hw(in, homework);
  return in;
}
```

階層構造における関数の定義

- Gradクラスの関数

```
istream& Grad::read(istream& in) {
  read_common(in);
  in >> thesis;
  read_hw(in, howmwork);
  return in;
}
double Grad::grade() const {
  return min(Core::grade(),thesis);
}
```

Gradクラスは基底クラスCoreの関数を使うことができるので、自分の固有の部分だけを定義する。

継承とコンストラクタ

- 派生クラスオブジェクトの生成手順
  - 全クラス分のメモリを確保する
  - 基底クラスのコンストラクタを実行し、基底クラス部分を初期化する
  - コンストラクタの初期化子によって、派生クラスのメンバーを初期化する
  - 派生クラスのコンストラクタが実行される

```
class Grad: public Core {
public:
  Grad(): thesis(0) { }
  Grad(std::istream& is) { read(is); }
  // ...
};
```

多態性(polymorphism)と仮想関数

- 基底クラスと派生クラスの共通部分を扱う関数

```
bool compare(const Core& c1, const Core& c2) {
  return c1.name() < c2.name();
}
Core c(cin);
Grad g(cin);
// 派生クラスGradのオブジェクトと
// 基底クラスCoreのオブジェクトを比較する
compare(g, c);
```

この場合、比較は基底クラスと派生クラスの共通の部分について行う。

多態性(polymorphism)と仮想関数

- 基底クラスと派生クラスの異なる部分を扱う関数

```
bool compare_grades(const Core& c1, const Core& c2) {
  return c1.grade() < c2.grade();
}
Core c(cin);
Grad g(cin);
// 派生クラスGradのオブジェクトと
// 基底クラスCoreのオブジェクトを比較する
compare_grades(g, c);
```

この場合、派生クラスと基底クラスの両方でgrade関数を定義しているが、compare_grades関数のパラメータはCoreクラスであるため、派生部分は見えないので、両方とも基底クラスのgrade関数が呼び出される。
->確実に派生クラスで定義した関数を呼び出す機構が必要となる。

多態性(polymorphism)と仮想関数

- 仮想(virtual)関数

```
class Core {
public:
  // 基底クラスの関数にvirtualキーワードを付ける
  virtual double grade() const;
  // ...
};
```

virtualというキーワードを付けることでcompare_gradeを実行する時に、c1とc2のオブジェクトの実際の型を見て、正しいバージョンのgrade関数(派生クラスで定義し直した関数)を実行できる。
この時、基底クラスの関数は、派生クラスの同名の関数によって再定義されるという。

多態性(polymorphism)と仮想関数

- 動的結合、dynamic binding
  - 関数の呼び出しがコンパイル時ではなく、実行時に決められる
  - virtual関数による実行時の選択は、参照かポインタを通して呼ばれる時にだけ行われる。
  - 1つの型がたくさんの型を表すことを多態性、動的結合によって仮想関数を呼び出すことを多態性呼び出しという。
- 静的結合, statically bound
  - virtual関数であっても、値による呼び出しの場合は動的な結合(dynamic binding)は行われない。
  - この場合、Gradクラス固有の部分は切り落とされて、Coreクラスの部分だけが関数に渡されることになる。

```
// 誤った実装の例
bool compare_grade(Core c1, Core c2) {
  return c1.grade() < c2.grade();
}
```

多態性(polymorphism)と仮想関数

```
class Core {
public:
  Core();
  Core(std::istream&);
  std::string name() const;
  // CoreとGradにある多重定義の部分を
  // virtual宣言する
  virtual std::istream& read(istream&);
  virtual double grade() const;
protected:
  std::istream& read_common(std::istream&);
  double midterm, final;
  std::vector<double> homework;
private:
  std::string n;
};
```

多態性(polymorphism)と仮想関数

- 新しい成績処理プログラム

```
int main(){
  vector<Core*> students; //実体ではなくポインタを使う
  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);
  }
  sort(students.begin(), students.end(),
    compare_Core_ptrs);
```

多態性(polymorphism)と仮想関数

- 新しい成績処理プログラム(続く)

```
  for (std::vector<Core*>::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;}
    delete students[i];
  }
  return 0;
}
```

多態性(polymorphism)と仮想関数

- 仮想デストラクタ
  - 上の新しいバージョンのプログラムはほとんどの問題を解決したが、まだ1つの問題を残してある。
    delete students[i];
    を実行した時、Coreクラスのオブジェクトか、Gradクラスのオブジェクトか判断できないので、Gradクラスオブジェクトの固有の部分は解放されない可能性がある。
  - この問題を解決するめたには、以下の様に、デストラクタをvirtual宣言することで解決される。

```
class Core {
public:
  virtual ~Core() { }
  // ...
};
```

ハンドルクラス

- ポインタと仮想関数を使った実装は継承によって生じたクラスの階層構造に動的な結合を実現した。
  - しかし、このようなプログラムは異なるオブジェクトの生成と破棄を管理することが必要で、またポインタの扱いも複雑になり、バグを呼び込みやすくなる欠点がある。
- 新しいハンドルクラスを定義する
  - インターフェース部分：ポインタのカプセル化と異なるクラスへの共通インターフェースを提供する。
  - スマートポインタ部分：ポインタのように振る舞い（動的結合ができる）、メモリの生成と破棄を自動で行う。
    ＊スマートポインタとしてのハンドルクラスは次回で説明する

ハンドルクラスの実装

```
class Student_info {
public: Student_info(): cp(0) { }
  Student_info(std::istream& is): cp(0) { read(is); }
  Student_info(const Student_info&); // コピーコンストラクター
  Student_info& operator=(const Student_info&); // 代入演算子
  ~Student_info() { delete cp; } // デストラクター
  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: Core* cp;
};
```

ここで、Student_infoクラスがインターフェースを提供し、CoreとGradクラスへのポインタ操作を吸収するので、ユーザーからはポインタ操作が見えなくなる。また、同時にメモリの管理も行う。

ハンドルクラスオブジェクトのコピー

- 新しい問題として、ハンドルクラスではデータへのポインタを使うので、コピーする際、データの型が基底クラスか派生クラスかは分からない。
- virtual関数cloneを追加し、クラスStudent_infoをCoreのfriendとする

```
class Core {
  friend class Student_info;
  //ほかは以前と同じ
protected:
  virtual Core* clone() const { return new Core(*this);  }
};
class Grad {
  //ほかは以前と同じ
protected:
  Grad* clone() const { return new Grad(*this);
}
```

ハンドルクラスオブジェクトのコピーと代入

- 一般的にはvirtual宣言する関数は戻り値も含めて同じである必要があるが、戻り値が基底クラスと派生クラスへのポインタの場合は違ってもよい。
- クラスStudent_infoからGradクラスのclone関数へのアクセスはあくまで基底クラスのCoreを通して行われるので、Gradクラスでfriend宣言する必要はない。
- Student_infoクラスのコピーコンストラクタと代入演算子定義

```
Student_info::Student_info(const Student_info& s) : cp(0) {
  if (s.cp) cp = s.cp->clone();
}
Student_info& Student_info::operator=(const Student_info& s) {
  if (&s != this) {delete cp; if (s.cp) cp = s.cp->clone();  else cp = 0;}
  return *this;
}
```

ハンドルクラスにデータを読込む

```
istream& Student_info::read(istream& is){
  // delete previous object, if any
  delete cp;
  char ch;
  // get record type
  is >> ch;
  if (ch == 'U') {
    cp = new Core(is);
  } else {
    cp = new Grad(is);
  }
  return is;
}
```

静的メンバー関数の定義と特徴

- ```
  static bool compare(const Student_info& s1,
      const Student_info& s2) {
      return s1.name() < s2.name();
  }
  ```
- メンバー関数であるため、一般関数のcompareとは区別され、共存することができる。
  ```

ハンドル(兼インターフェース)クラスを使う

```
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;
}
```