第13章「継承と動的結合を使う」

主なトピック

- クラスの継承（親クラス，子クラス）
- 多態性（ポリモルフィズム）と仮想関数(virtual)

2種類の学生と3つのクラス

- 学部生
  - 名前，中間試験，期末試験，演習
  - Coreクラス
- 大学院生
  - 学部生のデータに付け加え，論文
  - Gradクラス
- ハンドルクラス
  - 上の2つを使いやすくするためのクラス
  - Student_infoクラス

Coreクラス

```
class Core{
public:
    Core( );
    Core( std::istream& );
    std::string name( ) const ;
    virtual double grade( ) const;
    std::istream& read( std::istream& );
    //      次の実装もあり得る（詳細は後で説明）
    //      virtual std::istream& read( std::istream& );
protected:
    std::istream& read_common( std::istream& );
    double midterm, final;
    std::vector<double> homework;
private:
    std::string n;
};
```

Coreクラスの解説

- public: すべてから利用可能
- protected: 派生クラスから利用可能
  - midterm, final, homework, read_common()はGradクラスでも利用可能
- private: 派生クラスからでも利用不可能
  - nはGradクラスからは直接は利用できない
  - ただし，name()という関数は利用可能

Gradクラス

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

Gradクラスで特有の部分のみを定義するだけで良い

Gradクラスの解説

- class Grad : public Core
  - Gradクラスは，Coreクラスをpublic継承する
    - Coreクラスのpublicは，Gradクラスのpublicに
    - Coreクラスのprotectedは，Gradクラスのprotectedに
    - Coreクラスのprivateは，Gradクラスのprivateに
  - Gradクラスは，以下のCoreクラス
    - name(), read()というメンバ関数
    - n, midterm, final, homeworkというメンバ変数
  - を利用可能
  - さらに
    - thesisというメンバ変数を追加
    - grade(), read()というメンバ関数はGradクラスで再定義

参考までに

- class Grad : protected Coreとすると
  - Gradクラスは，Coreクラスをprotected継承する
    - Coreクラスのpublicは，Gradクラスのprotectedに
    - Coreクラスのprotectedは，Gradクラスのprotectedに
    - Coreクラスのprivateは，Gradクラスのprivateに
- class Grad : private Coreとすると
  - Gradクラスは，Coreクラスをprivate継承する
    - Coreクラスのpublicは，Gradクラスのprivateに
    - Coreクラスのprotectedは，Gradクラスのprivateに
    - Coreクラスのprivateは，Gradクラスのprivateに

関数の定義 Coreクラス

```
std::string Core::name( ) const { return n; }

double Core::grade( ) const
{
    return ::grade( midterm, final, homework );
}

std::istream& Core::read_common( std::istream& in )
{
    in >> n >> midterm >> final;
    return in;
}
```

関数の定義 Coreクラス

```
std::istream& Core::read ( std::istream& in )
{
    read_common( in );
    read_hw( in, homework );
    return in;
}
```

関数の定義 Gradクラス

```
std::istream& Grad::read ( std::istream& in )
{
    read_common( in );
    in >> thesis;
    read_hw( in, homework );
    return in;
}

double Grad::grade() const
{
    return min( Core::grade(), thesis );
}
```

継承とコンストラクタ

- 派生クラスのオブジェクトの生成
  - 全オブジェクト分のメモリを確保
  - 基底クラスのコンストラクタを実行，基底クラスの部分を初期化
  - 派生クラスのメンバを初期化（: 以降の部分）
  - 派生クラスのコンストラクタの中身（{}の中の部分）を実行

継承とコンストラクタ

```
Core::Core( ) : midterm(0), final(0) { }

Core::Core( std::istream& is ) { read(is); }

//  下の2つともCore()で基底クラス部分を初期化
Grad::Grad ( ) : theisis (0) { }

Grad::Grad( std::istream& is ) { read(is); }
```

親クラスのコンストラクタが動いた後に、子クラスのコンストラクタがうごく。ky

多態性（ポリモルフィズム）と仮想関数

```
bool compare(const Core&c1, const Core& c2)
{
      return c1.name() < c2.name();
}
```

- Coreオブジェクトでも Gradオブジェクトでも実行可能
  - Core c1(cin), c2(cin); Grad g1(cin), g2(cin);
  - compare(c1, c2);
  - compare(g1, g2);
  - compare(c1, g2);
  - CoreクラスもGradクラスもCore::name()を利用するため

多態性（ポリモルフィズム）と仮想関数

```
bool compare_grades( const Core&c1, const Core& c2 )
{
      return c1.grade( ) < c2.grade( );
}
```

- 上の関数はCoreクラスのCore::grade()関数を実行
- しかし，Gradクラスでは Grad::grade()を実行すべき
- このままでは，正しく動作しない

virtual（仮想）関数

```
class Core  {
public:
      virtual double grade() const;
};
```

- Core::grade()（基底クラス，親クラス）にvirtualを付けると，compare_grades(c1, c2)を実行時にc1とc2のオブジェクトの型を見て，該当するオブジェクトのクラスのgrade()関数を実行
- Grad::grade()にvirtualキーワードは付けなくても良いし（継承されるため），付けても良い
- この多態性を使うときは必ず基底クラス（親クラス）の関数にvirtualを付けること

overrideとfinal

```
class Grad  {
public:
      virtual double grade() const override;
};
```

- Grad::grade()（派生クラス，子クラス）側のvirtual関数にoverrideキーワードを付け，明示的に処理が書き換えられている（オーバーライドされている）ことを示してもよい．（コードを見たときの分かりやすさ）．

```
class Grad  {
public:
      virtual double grade() const final;
};
```

- Grad::grade()（派生クラス，子クラス）側のvirtual関数にfinalキーワードを付け，明示的にこれ以上継承させない，あるいはオーバーライドさせないことを宣言することができる．

動的結合と静的結合

- 動的結合: プログラムの実行時にオブジェクトの型が決められる
  - 仮想関数で，参照かポインタを通して呼ばれる時
- 静的結合: プログラムのコンパイル時にオブジェクトの型が決められる
  - 仮想関数でも，オブジェクトを通して呼ばれる時は静的結合
  - 静的結合の例

```
bool compare_grades( Core c1, Core c2 )
{  return c1.grade( ) < c2.grade( );  }
```

動的結合と静的結合

- Core c;
- Grad g;
- Core *p = &c; p = &g;
- Core& r = g; Core& r = c;
- c.grade(); // Core::grade()に静的結合
- g.grade(); // Grad::grade()に静的結合
- p->grade(); // 動的結合 pの指す型に依存
- r.grade(); // 動的結合 rの指す型に依存
- 動的結合見分け方は，ポインタで仮想関数が呼ばれているか

ハンドルクラス

- Coreクラスは学部生，Gradクラスは大学院生
- 今のままでは，両者が混在した場合，それぞれのオブジェクト準備しなければならない
  - 学部生ならデータの先頭がu，大学院生ならg
  - 名前，中間試験，期末試験，大学院生なら論文（学部生はなし），演習の成績
  - u name 100 100 50 50 50
  - g name 100 100 80 50 50 50
- 両方を同時に扱うためのハンドルクラスを定義

virtualな型のコンテナ

- vector<Core> students;
- Core record;
  - 上の2つはCore型のオブジェクトのみ
  - Grad型のオブジェクトは扱えない
  - 静的結合だから

virtualな型のコンテナ

- vector<Core*> students;
- Core* record;
  - これだと動的結合になるので，Core型とGrad型のオブジェクトが利用可能
- 上はポインタだけで実体（メモリ）が確保されていないため実行不可能
- ハンドルクラスにはメモリ管理機能が必要

Student_infoクラス

```
class Student_info{
private:
    Core *cp;
public:
    Student_info( ) : cp( 0 ) { }
    Student_info( std::istream& is ) : cp( 0 ) { read(is); }
    Student_info( const Student_info& is );
    ~Student_info( ) { delete cp; }
    std::istream& read( std::istream& );
    std::string name( ) const              {
        if(cp) return cp->name();
        else throw std::runtime_error("uninitialized");
    }
    static bool compare(const Student_info& s1, const Student_info& s2){
        return s1.name() < s2.name();
    }
};
```

staticなメンバ関数

- 特定のオブジェクトではなく，クラスに付随した関数
- クラスのスコープ内で定義
  - 常にStudent_info::compareでアクセスされる
  - 通常の関数だとs.compareでアクセス
  - sort関数を呼び出す時に，他のcompare関数と区別するために利用

ハンドルを読む

```
std::istream& Student_info::read( std::istream& is)
{
    delete cp;
    char ch;
    is >> ch;
    if(ch == 'u')     {
            cp = new Core( is );
    }
    else  {
            cp = new Grad( is );
    }
    return is;
}
```

ハンドルオブジェクトのコピー

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

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

ハンドルオブジェクトのコピー

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

- GradではStudent_infoをfriendにする必要なし
  - Student_info はGrad::cloneを使わないから
  - friendは継承されない

ハンドルオブジェクトのコピー

```
Student_info::Student_info( const Student_info& s) : cp( 0 )
{
    if( s.cp ) cp = s.cp -> clone( );
}
```

ハンドルオブジェクトのコピー

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

ハンドルクラスを使う

```
int main() {
    vector<Student_info> students;
    Student_info record;
    //      データの読み込み
    while( record.read( cin ) )  {
            students.push_back( record );
    }
    //      学生をアルファベット順に並び替える
    sort( students.begin(), students.end(), Student_info::compare );
    for(vector<Student_info>::size_type i = 0; i != students.size(); ++i)   {
            cout << students[ i ].name() << “ ”;
            //  Student_infoクラスにvalid()という関数を追加
            if( students[ i ].valid() ) {
                cout << students[ i ].grade() << endl;
            }
            else {
                cout << “No homework” << endl;
            }
    }
}
```