12-9 ostream_iterator を使って Str の出力演算子を定義し直してください。ここで、「istream_iterator を使って Str の入力演算子を定義し直してください」と質問しないのは、なぜだかわかりますか。

12-10 § 12.1 で反復子 2 つを引数に取る Str のコンストラクタの定義を考えました。このようなコンストラクタは Vec クラスで有用だと思います。このようなコンストラクタを Vec に付け加え、copy の代わりにその Vec のコンストラクタを使って Str のコンストラクタを書き直してください。

12-11 この課題にある関数を付け加えれば Str クラスは、本書で文字列を扱っているすべてのプログラムで使えるようになります。第 5 章の文字絵の関数の定義や § 5.6 と § 6.1.1 の split 関数の定義を書き換えてみてください。

12-12 2 つの反復子を引数に取る insert 関数を Vec と Str に付け加えてください。

12-13 配列のデータを Vec のオブジェクトに代入できるような代入関数を定義してください。

12-14 string オブジェクトで Vec を初期化するコードを書いてください。

12-15 § 4.1.3 の read_hw 関数は読み込みに使っているストリームを調べ、ファイルの最後に到達したのか無効な入力に遭遇したのかを判断しています。しかし、Str の入力演算子はそのようなチェックをしません。なぜでしょうか。それはストリームを無効な状態にして、そのまま放置してしまうでしょうか。

第 13 章

継承と動的結合を使う

ここまでの数章では、自分のデータ型をどのように作るかという話をしてきました。これはオブジェクト指向プログラミング（OOP）の基本の 1 つです。この章では、OOP の他の重要な要素である、継承と動的結合を紹介します。

第 9 章で、第 4 章の成績処理の問題を解決するために、小さなカプセル化されたクラスを定義しました。この章ではこれを再び考えてみます。今度は、同じコースを学部生*1と大学院生の両方の学生が混じって出席できるように、大学の仕組みが変更されたとします。ただし、大学院生として単位を取る場合は、学部生よりやることが増えるとします。学部生も大学院生も共通して宿題をし、試験を受けるほかに、大学院生は論文も書かなければならないとしましょう。後でも見るように、このような変更には、オブジェクト指向の解決策があります。この解決策を見ながら、C++がどのように OOP をサポートしているか説明していきましょう。

この章では、このような新しい道具立てを使って新しいクラスを書いていきます。また、§ 9.6 で考えた以前のコードも破棄せず残しておきたいわけです。成績データを読み込み、最終成績を計算し、オリジナルのコードを使って、整形されたレポートを出力するようにします。

13.1 継承

成績処理の問題では、学部生と大学院生の単位取得の条件は、大学院生が関連論文を書くということを除けば同じでした。このよう文脈で、**継承**（インヘリタンス、inheritance）は自然なものです。継承とは、OOP の重要な要素の 1 つです。私たちには、あるクラスに対してほとんど同じだが少しだけ拡張したものが必要になるということがよくあります。今の場合、すべての学生は宿題とテストをしなければならないのに対し、そのうちの一部の学生（大学院生）は論文も書かなければならないということです。ここで 2 つのクラスを定義したいと考えます。1 つは学部生を表すクラス（実は、学部生・大学院生の共通部分のデータを持つクラス）で、もう 1 つは大学院生を表すクラスです。

最初のクラスをどう書くかは大体わかっています。これは前に書いた Student_info クラスに近いものでしょう。ただ名前は Core（コア）とします。名前を変える理由は § 13.4 で説明します。ここで重要なことは、Core はすべての学生を表すのではなく、学部生・大学院生の共通部分のデータだけを持つ学生、つまり学部生を表すということです。Student_info はすべての学生を表すクラスの名前に使うために取っておくのです。名前の変更に加えて、Core クラスには、学部生大学院生に共通な部分の点数を読み出す private なユーティリティ関数を付け加えておきます。

*1 訳注：いわゆる「大学生」を大学院生と区別する言葉です。

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

Grad クラスは大学院生を表すクラスで、論文の点数も含むものにします。

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

この定義により Core から派生した (derived from)、あるいは Core を継承した (inherits from) Grad クラスを定義しているのです。あるいは、Core は Grad の基底 (基本) (base) クラスとも言います。Grad は Core を継承しているので、コンストラクタ、代入演算子、デストラクタを除くすべての Core のメンバは Grad のメンバです。そして Grad はさらにデータである thesis (論文の点数) とコンストラクタをメンバに持っています。また、grade や read 関数で行うように基底クラスであるものを定義しなおすこともできます。しかし、基底クラスのメンバを削除することはできません。

継承における pubic Core の public は、Grad は Core の実装ではなくインタフェースを継承するという意味です。つまり、Grad は Core の public なインタフェースを継承し、それを Grad の public なインタフェースにするということです。Core の public なメンバは、そのまま Grad の public なメンバになるのです。たとえば、Grad オブジェクトがあれば、Grad が独自に name 関数を定義していなくても、学生の名前を得るために name メンバ関数を呼び出せるのです。

Grad クラスは論文の点数を保持し、最終成績を計算するのに異なるアルゴリズムを使うという意味で Core クラスとは違うものです。実際、Grad は 5 つのデータメンバを持ちます。4 つは Core から継承したもので、5 番目のものは double である thesis です。また、2 つのコンストラクタと 4 つのメンバ関数を持ちます。そのうち、2 つは Core の対応する関数を再定義したものです。また、name 関数、read_common 関数は Core から継承したものです。

13.1.1 もう一度、データ保護

Core の 4 つのデータメンバと read_common 関数は、Grad のメンバ関数からアクセスすることはできません。Core のメンバを private にしたからです。private なメンバにアクセスできるのは、自分自身とそれに friend なクラスだけです。残念ならが、Grad の grade 関数と read 関数を書くためには、private なメンバにアクセスする必要があるのです。この問題を解決するためには、これまでに説明しなかった保護ラベルを使って、Core クラスを次のように書き換えればよいのです。

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

ここでもまだ n は private です。しかし、read_common 関数とデータメンバである midterm、final、homework は **protected** というものにしました。protected ラベルがあると、他のクラスからはアクセスできないのですが、Grad のような派生クラスからはそのメンバにアクセスできるのです。

n は Core の private メンバなので、Core のメンバか Core の friend しか、アクセスできません。Grad は n に直接アクセスできないのです。できるのは、Core の public なメンバ関数を通してのアクセスのみです。

read 関数、name 関数、grade 関数は Core の public なメンバ関数なので、Core のすべてのユーザが使用可能です。それは Core の派生クラスでも使用可能だということです。

13.1.2 関数の定義

クラスの定義を完成するために、コンストラクタの定義を書かなければなりません。デフォルトコンストラクタと istream を引数に取るコンストラクタで、これらは両方のクラスにあります。さらに 6 つの関数、つまり Core の name と read_common、また両方のクラスの read と grade も定義する必要があります。コンストラクタについては § 13.1.3 で書くことにします。

コードを書く前に、学生の成績データをどのような形式で書くかを少し考える必要があります。前に見たように、宿題の数は決まっていないので、これらの点数は学生データ 1 件 1 件の最後に置くことにします。そこでまず、学生データ 1 件は、名前、中間試験の点数、期末試験の点数の順になっているとします。そして、もしこの学生が学部生ならその後に宿題の点数が続くとし、大学院生ならその後にまず論文の点数があって、さらにその後に宿題の点数が続くとします。

このように考えると、Core の関数は次のように書くことができます。

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

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

istream& Core::read_common(istream& in)
{
    // 学生の名前と試験の成績を読み込む
    in >> n >> midterm >> final;
    return in;
}

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

Grad::read 関数は似ていますが、read_hw を呼び出す前に thesis を読み込みます。

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

Grad::read の定義では、基底クラスのメンバをそのまま使っていることに注意してください。これらは Grad クラスのメンバと考えることもできるのです。もし、Core から継承したメンバをはっきり示したいなら、スコープ演算子を使って次のように書くこともできます。

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

もちろん、thesis は Grad 自身のメンバであって Core のメンバではないので何も修飾が付きません。これは Grad::thesis と書くこともできますが、Core::thesis ではないのです。

grade 関数は論文の点数（thesis）があるかどうかで定義が変わります。ここでは、論文の点数と「宿題・試験から計算したもとの最終成績の点数」を比べて、低い方の点数を大学院学生の最終成績にすることにします。

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



```
}
```

ここで、thesis と比べる点数を計算するために基底クラスの grade 関数を呼び出しました。この場合、スコープ演算子の使用は非常に重要です。もし、

```
return min(grade(), thesis);
```

などと書いてしまえば、Grad の grade の中で同じ grade を（再帰的に）呼び出すことになってしまい、結果は悲惨なことになります。

また、戻すべき点数を求めるために、<algorithm>の min 関数を使いました。min は max と同じように動作する関数ですが、これは 2 つのオペランドのうち小さいほうを戻す関数です。max と同じで 2 つのオペランドの型は正確に同じでなければなりません。

13.1.3 継承とコンストラクタ

Core と Grad のコンストラクタを書く前に、派生クラスがどのように生成されるかを理解しておく必要があります。他のクラスと同様に、まずオブジェクトのためにメモリが確保されるところからはじまります。次に、適当なコンストラクタを実行し、オブジェクトを初期化します。ここで派生クラスのオブジェクトの生成に関しては、基底クラスの部分を生成するため、生成手順が 1 ステップ多くなるのです。派生クラスのオブジェクトは次のように生成されます。

- （派生クラスのメンバも基底クラスのメンバも同様に）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し、オブジェクトの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したがって派生クラスのメンバを初期化する
- 派生クラスのコンストラクタの中身（あれば）を実行する

ここで実行する基底クラスのコンストラクタをどのように選ぶかということが問題になります。実は、派生クラスの初期化子を使って実行する基底クラスのコンストラクタを選ぶことができるのです。派生クラスのコンストラクタでは（空かもしれませんが）パラメータリストの後に、基底クラスのコンストラクタを初期化子として置くのです。これは、驚くことではないと思います。もし、初期化子で起動するコンストラクタを指定しないと、基底クラス部分の生成には基底クラスのデフォルトコンストラクタが使われることになります。

```
class Core {
public:
    // Coreのデフォルトコンストラクタ
    Core(): midterm(0), final(0) { }
    // istreamからCoreを生成する
    Core(std::istream& is) { read(is); }
    // . . .
};

class Grad: public Core {
public:
    // 下のコンストラクタは両者ともCore::Core()で基底クラス部分を初期化
    Grad(): thesis(0) { }
    Grad(std::istream& is) { read(is); }
```

```
    // . . .
};
```

Core のコンストラクタは § 9.5.1、§ 9.5.2 のものと同じです。Core を「無」または istream から生成するのです。Grad のコンストラクタは Grad を Core と同様に、引数無しかストリームから生成するものです。ただし、派生クラスのコンストラクタは基底クラスのコンストラクタと同じ引数をとらなければならない、というような決まりはありません。

Grad のデフォルトコンストラクタは Grad を外からのデータ無しで生成できることを示しています。これでは Core の部分を生成し、thesis メンバを 0 にセットします。あきらかに、この動作の大部分は、明示的ではありません。コンストラクタの初期化子はなく、そのためデータメンバの midterm、final、homework、name を初期化するのに、非明示的に Core のデフォルトコンストラクタが呼ばれます。同様に、Core のコンストラクタは非明示的にそれぞれのコンストラクタを使って name と homework を初期化します。ここで明示的に初期化されるのは midterm と final だけです。Grad のコンストラクタが明示的にしていることは、データメンバである thesis の初期化です。これが済めばすることはないので、コンストラクタの定義の中身は空です。

istream から Grad クラスを生成する方法は、istream から Core を生成する方法と同じで、メンバ関数の read を呼び出しています。しかし、そうする前に、(非明示的に) 基底クラスのデフォルトコンストラクタを呼び出し、オブジェクトの基底クラス部分は初期化してあるのです。また、このコンストラクタは Grad のものなので、ここで使われている read は Grad::read です。thesis はわざわざ初期化しませんが、それは read 関数が is からデータを thesis に読み込むからです。

派生クラスのオブジェクトの生成のされ方を理解しておくことは重要です。たとえば、

```
Grad g;
```

とすると、Grad の 5 つのデータメンバを保持するのに十分なメモリが確保され、g のうち Core 部分のデータメンバを初期化するために、Core のデフォルトコンストラクタが呼ばれ、それから Grad のデフォルトコンストラクタが呼び出されるわけです。同様に、

```
Grad g(cin);
```

の場合、適当なメモリが確保された後、Core のデフォルトコンストラクタが呼ばれ、次に Grad::Grad(istream&) コンストラクタが、name、midterm、final、thesis、homework メンバにデータを読み込むのです。

13.2　多態性（ポリモルフィズム）と仮想関数

まだ元の Student_info の抽象化を完全に再定義していません。前の定義では、そのインタフェースの一部をサポートするためにメンバでない関数を使っていました。2 つの学生のデータを比べるのに compare という関数を使ったのです。この関数は学生データをアルファベット順に並べるのに必要だったのです。

型の名前が変わったことを除くと、ここで書く新しい比較の関数は § 9.3.1 で書いたものと同じです。

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



```
}
```

2 つの学生データを比べるのに名前を使いました。compare の仕事を、string ライブラリの<演算子に任せているのです。このコードで注目すべきは、これを Core オブジェクト同士の比較にも、Grad オブジェクト同士の比較にも、また Core オブジェクトと Grad オブジェクトの比較にも使えるということです。

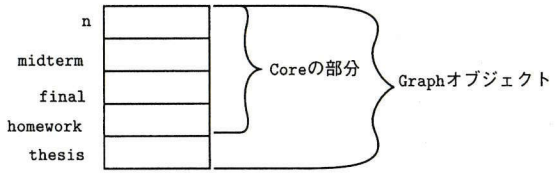
```
Grad g(cin);        // Gradオブジェクトにデータを読み込む
Grad g2(cin);       // Gradオブジェクトにデータを読み込む

Core c(cin);        // Coreオブジェクトにデータを読み込む
Core c2(cin);       // Coreオブジェクトにデータを読み込む

compare(g, g2);     // 2つのGradオブジェクトを比較する
compare(c, c2);     // 2つのCoreオブジェクトを比較する
compare(g, c);      // GradオブジェクトとCoreオブジェクトを比較する
```

compare のどの呼び出しでも、Core クラスのメンバである name が実行され、compare の戻り値が決められるのです。あきらかに、これは Core にとって正しいメンバですが、Grad についてはどうでしょう。Grad を定義したときに、これは Core を継承するとし、その上で name 関数は再定義していません。そのため、たとえば Grad のオブジェクト g で g.name() を実行すると、これは Core から継承した name の呼び出しになるのです。この関数は Core のときと同様に Grad でも動作します。オブジェクトの Core 部分の n を取り出すのです。

Core&を引数に取る関数に Grad を渡してよいのは、Grad が Core を継承しているからです。すべての Grad オブジェクトは Core の部分を持っているのです。



すべての Grad オブジェクトが Core 部分を持っているので、compare の Core への参照パラメータに Core オブジェクトを渡せるのとまったく同様に、そのパラメータに Grad オブジェクトを与えることができるのです。また、compare 関数の引数を、Core へのポインタか（参照ではなく）Core オブジェクトにしても、この関数に Grad オブジェクトを渡すことができます。引数がポインタの場合、Grad へのポインタを渡せばよいのです。すると、コンパイラは Grad*を Core*に変換し、それを Grad オブジェクトの Core の部分を指すポインタにするのです。もし関数の引数が単に Core オブジェクトであった場合、渡されるものは Core 部分だけです。ここで引数がオブジェクトであるか参照かポインタであるかでは、動作は驚くほど変わってきます。これは次に見ることにしましょう。

13.2.1 オブジェクトの型を知らずにオブジェクトを受け取る

compare 関数は、引数に Grad オブジェクトを渡されても、name 関数は Grad と Core の両方にあると考えられるので問題は起こりません。しかし、学生を名前ではなく最終成績で比較したい場合はどうでしょう。たとえば、学生を名前順に並べてその最終成績を出力するのではなく、最終成績順に学生を並べなおす必要も出てくるでしょう。

そこで、compare に似た関数を書いてこの問題を解決してみましょう。

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

compare との違いは、内部で呼び出しているのが name ではなく grade 関数というだけです。しかし、このわずかな違いが大きな違いになるのです。

違いは Grad クラスが grade 関数を再定義していることから来ます。私たちは、Grad と Core の同名の関数を区別するようなことはしませんでした。compare_grades 関数を実行すると、compare が Core::name を実行するように、compare_grades は Core::grade を実行します。そこで、もし Grad オブジェクトについてこの関数を実行すると、Core の grade が実行され、間違った成績で比較が行われてしまうのです。grade 関数は Core と Grad で違うからです。Grad オブジェクトについては thesis まで計算に入れた Grad::grade を実行しなければならないのです。

ここで、compare_grades では、渡されたオブジェクトの実際の型の grade を実行するようにしたいと考えます。つまり、c1、c2 が Grad オブジェクトの参照なら、compare_grades の中では Grad の grade を実行したいはずです。そして、もちろん、オブジェクトが Core なら、実行されるべきなのは Core の grade です。このような判断を実行時に行いたいということです。言い換えれば、関数に渡されるオブジェクトの実際の型に基づいて正しい関数が実行されるようにしたいのです。ただし、渡されるオブジェクトの型は実行時までわからないことに注意してください。

このような実行時の選択をサポートするために、C++には **virtual な関数**（仮想関数）というものがあります。

```
class Core {
public:
    virtual double grade() const;   // virtualというキーワードを付けた
    // . . .
};
```

上のようにすることで grade を virtual 関数にできるのです。このようにして compare_grades を実行すると、参照 c1、c2 の指すオブジェクトの実際の型を見て、正しいバージョンの grade が呼ばれるようになるのです。これは引数に渡されたオブジェクトを調べて、実行すべき関数を決めているということです。もし引数が Grad オブジェクトを指すなら、Grad::grade 関数が実行され、Core なら Core::grade 関数が実行されます。

この virtual というキーワードは、クラス定義の中でだけ使われます。関数の定義がその宣言とは別にクラス定義の外にある場合、定義のところで virtual を繰り返しません。そのため、Core::grade の定義は変わらないのです。また、同じように、virtual という性質は継承されるので、Grad クラスの grade の宣言に

virtual を付ける必要もありません。ただし、新しい Core クラスの定義のコードはコンパイルし直す必要があります。そうすると、基底クラスの関数で virtual と指定したことになり、コードは望んだ振る舞いをするようになります。

13.2.2 動的結合

virtual な関数（仮想関数）のこのような実行時の選択は、参照かポインタを通して関数が呼ばれるときだけ行われます。仮想関数であっても、（参照やポインタではなく）オブジェクトに対して実行されるなら、その型はコンパイル時に決まっているのであり、実行時に変わることはありません。一方、参照やポインタは基底クラスでも派生クラスでも指すことができ、したがって、それらの指すオブジェクトの型は実行時に決められることになります。virtual の仕組みが異なる結果を出すのはこのような場合なのです。

たとえば、compare_grades を次のように書いたとします。

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

パラメータはオブジェクトであって参照ではありません。この場合、c1 と c2 の型はいつも Core とわかっているのです。このような定義でも引数に Grad オブジェクトを使うことができます。しかし、このとき、Grad の型は使われません。その場合、オブジェクトの Core 部分だけが渡されるのです。つまり、Grad オブジェクトの Core にない部分は、**切り落**（cut down）とされてしまい、Core の部分だけがコピーされて compare_grades に渡されることになります。パラメータが Core オブジェクトであるため、この関数内の grade 呼び出しは、（コンパイル時に）Core::grade になります。このようにコンパイル時に呼び出される関数が決められていることを、**静的結合**（statically bound）と言います。

参照やポインタを使う関数呼び出しの場合は**動的結合**（dynamic binding）とよばれますが、これと静的結合の違いは、C++がどのように OOP をサポートするかを理解する上で、非常に重要です。静的結合という言葉が「呼び出す関数がコンパイル時に決められる」を意味するのに対し、動的結合という言葉は「関数の呼び出しが実行時に決められる」という意味です。仮想関数であっても、オブジェクトに対して呼び出すなら、その呼び出しは静的結合です。そのオブジェクトがコンパイル時と実行時に異なる型になることはないからです。一方、ポインタや参照に対しては、仮想関数は、実行時に内容が決められる、動的結合になります。仮想関数は、参照やポインタが実行時に指す型によって、異なるバージョンのものが使われるのです。

```
Core c;
Grad g;
Core* p;
Core& r = g;

c.grade();  // Core::grade()に静的に結合
g.grade();  // Gore::grade()に静的に結合
p->grade(); // 動的な結合：pの指すオブジェクトの型によって異なる
r.grade();  // 動的な結合：rの指すオブジェクトの型によって異なる
```

はじめの 2 つの関数呼び出しは静的結合です。c が Core オブジェクトであることは実行時でも変わらないからです。そのため、grade が仮想関数であっても、コンパイラはこの関数呼び出しをコンパイル時に決定してしまうのです。しかし、3、4 番目の関数呼び出しでは、p や r の指すオブジェクトの型が実行時までわかりません。それらは Core かもしれないし Grad かもしれないのです。そのため、実際にどのバージョンの grad を使うかは、実行時まで決められないのです。実行時の判断は、p や r の指すオブジェクトの型によってなされます。

基底クラスのポインタや参照が派生クラスのオブジェクトを指すことができるわけですが、これは多態性（ポリモルフィズム）という OOP の重要な概念の例になっています。この単語はギリシャ語の *πολύμορφος* から来たものですが、「たくさんの形」を意味し、19 世紀の中ごろには英語の単語になっています。プログラミングの世界では、1 つの型がたくさんの型を表せることを意味します。C++は、仮想関数の動的結合機能を通して、多態性をサポートしているのです。ポインタや参照を通して仮想関数を呼び出すことを、多態性呼び出しということもできます。参照（やポインタ）そのものの型は決まっていますが、それが指すオブジェクトの型は、参照（やポインタ）が最初に意図した型でも、その派生クラスの型でもよいのです。これにより、1 つの型を通して、たくさんの関数を呼び出せるわけです。

仮想関数について最後にコメントがあります。この関数は、実際にプログラムで使うかどうかに関わらず定義が必要であるということです。仮想でない関数は、プログラム中で使わないなら、宣言だけで定義をしないということも可能でした。しかし、仮想関数については、宣言だけで定義を書かないとコンパイラは不思議なエラーメッセージを出すことでしょう。もし、コンパイラが、理解できないエラーメッセージを出したら、そしてそのメッセージが「何かを定義していない」という意味のものなら、すべての仮想関数の定義を書いたかチェックしてみるとよいでしょう。もし書き忘れがあれば、それを直すだけでエラーがなくなるはずです。

13.2.3 まとめると

先に進む前に、ここまでをまとめておきます。さらに、少しだけ書き換えもします。read 関数も virtual にしておくのです。read 関数も、実行時の型によって異なるバージョンが呼び出されるようにしておきたいからです。この書き換えをして、まとめると、クラスは次のようになります。

```
class Core {
public:
    Core(): midterm(0), final(0) { }
    Core(std::istream& is) { read(is); }
    std::string name() const;
    // §13.1.2で定義したもの
    virtual std::istream& read(std::istream&);
    virtual double grade() const;
protected:
    // 派生クラスからアクセスできる
    std::istream& read_common(std::istream&);
    double midterm, final;
    std::vector<double> homework;
private:
    // Coreのみからアクセスできる
    std::string n;
};
```



```
class Grad: public Core {
public:
    Grad(): thesis(0) { }
    Grad(std::istream& is) { read(is); }
    // §13.1.2で定義したもの；gradeとreadは継承により仮想であることに注意
    double grade() const;
    std::istream& read(std::istream&);
private:
    double thesis;
};

bool compare(const Core&, const Core&);
```

2 種類の学生データをカプセル化するために 2 つのクラスを定義しました。1 つは Core で、この授業に参加する学生の共通データを表していると言えます。もう 1 つは Core を継承したクラスで、論文の点数を付け加えたものです。Core や Grad オブジェクトを生成するには 2 通りの方法があります。デフォルトコンストラクタはオブジェクトのデータを適当に設定し、istream&を引数に取るコンストラクタは特定のストリームからデータを読み込んでオブジェクトを生成するのでした。関数には、データを読み込む関数、名前を取り出す関数、最終成績を取り出す関数があります。この節から、grade と read の両方を virtual にしました。また、今考えているインタフェースには、2 つのオブジェクトを名前で比較する compare 関数があります。これはメンバではないグローバルな関数です。

13.3 問題解決に継承を使う

異なる学生を表すクラスを作ったので、これを使って § 9.6 以来の成績処理プログラムを考え直しましょう。プログラムは、学生の成績データの入っているファイルを読み込み、各学生について最終成績を計算し、学生を名前順にならべて成績を整形して出力するというものです。ここでは同じことを、ただし 2 種類の学生のデータが混在しているファイルについてしたいのです。

問題のすべてを解決する前に、2 つのより簡単な問題を考えましょう。まず、学生が学部生か大学院生のどちらかだけである場合のプログラムを書いてみます。このプログラムは、型の宣言を除くともとのプログラムによく似ています。

```
int main()
{
    vector<Core> students;   // 学生データがCoreの場合
    Core record;
    string::size_type maxlen = 0;
    // データを読み込み、保持する
    while (record.read(cin)) {
        maxlen = max(maxlen, record.name().size());
        students.push_back(record);
    }
    // 学生データを名前順に整列させる
    sort(students.begin(), students.end(), compare);
    // 名前と成績を出力
    for (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();   // Core::grade
            streamsize prec = cout.precision();
            cout << setprecision(3) << final_grade
                 << setprecision(prec) << endl;
        } catch (domain_error e) {
            cout << e.what() << endl;
        }
    }
    return 0;
}
```

Grad に対しても、型の定義を変えれば同じプログラムが使えます。

```
int main()
{
    vector<Grad> students;          // vectorの要素の型が異なる
    Grad record;                    // 異なる型のオブジェクトへの読み込み
    string::size_type maxlen = 0;
    // データを読み込み、保持する
    while (record.read(cin)) {   // Coreではなく Gradから読み込む
        maxlen = max(maxlen, record.name().size());
        students.push_back(record);
    }
    // 学生データを名前順に整列させる
    sort(students.begin(), students.end(), compare);
    // nameとgradesを出力する
    for (vector<Grad>::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();   // Grad::grade
            streamsize prec = cout.precision();
            cout << setprecision(3) << final_grade
                 << setprecision(prec) << endl;
        } catch (domain_error e) {
            cout << e.what() << endl;
        }
    }
    return 0;
}
```

もちろん、それぞれのプログラムで実行される関数は、record と vector の要素の型によって決まります。たとえば、あるエクスプレッションで Core::read 関数と Grad::read 関数のどちらを呼び出すかは、record の型によって決まるわけです。この関数呼び出しは静的結合であることに注意してください。record の型で決まるということに、ここでは read 関数の virtual な性質は何も関係ないのです。関数をオブジェクトに対して呼び出しているのであって、オブジェクトを指すポインタや参照に対して呼び出しているのではないからです。そのため、record は Core であるか Grad であるか最初から固定していて、record.read(cin) でどのバージョンの read が呼ばれるかはコンパイル時に決まっているのです。同様に出力するときに呼ばれる grade 関数の


```
students[i].grade()
```

も、前のプログラムでは Core に静的結合し、後のプログラムでは Grad に静的結合しているのです。

どちらのプログラムでも、name() は（仮想関数でない）Core のものが使われています。この関数は Grad にも継承されているので、Grad オブジェクトに対してこの関数を実行しても、Core で定義されたものが実行されるのです。sort に渡す compare 関数のパラメータは Core への参照ですが、Grad オブジェクトを整列するときは、内部で使われる name は仮想関数ではないので、Core の部分のみを比較して整列することになります。

あきらかに、2 つのバージョンのプログラムを別々に書くのは面倒です。ここでしたいことは、1 つのプログラムを書いて、それで Core オブジェクトでも Grad オブジェクトでも処理できるようにするということです。

Core のファイルでも Grad のファイルでも読み込める関数を書くには、コードを詳細に検討し、record の型が問題になるところを見つけておく必要があります。Core でも Grad でも処理できるプログラムを書くためには、以下のような型依存性をなくす必要があるのです。

- 読み込むデータを保持する vector の定義で、要素の型を示している
- 読み込みに使うローカルな一時変数
- read 関数
- grade 関数

残りのコードは型によらない（vector の中身を整列するコードや、それを取り出すコード）か、Grad と Core が同じもの（name や compare 関数）を使うのです。なお、grade と read は仮想関数にしたので、上に挙げた問題のうち後の 2 つは解決済みと言えます。

前の 2 つの問題、つまりローカルな一時変数変数の型とコンテナに格納する型の問題は、同じ方法で解決できます。この問題解決には、実は 2 つの異なるアプローチがあるのです。1 つ目のアプローチはストレートなのですが、これはすぐ次で見ることにします。2 つ目のアプローチは C++プログラムではよく見られるもので、イディオム（定石）とも言えるものです。これは § 13.4 で考えます。

13.3.1　（virtual な）知らない型のコンテナ

ここで考えなければならない問題は、次のようなコードの型依存性を緩めるということです。

```
vector<Core> students;   // Coreオブジェクトのみを保持、多態型ではない
Core record;             // Coreオブジェクトであって、その派生クラスではない
```

上のコードの型依存性は明らかです。record を定義するときまさにこの型は Core だと言っているのです。同様に、students を定義するときにも、これは Core 型のオブジェクトを保持する vector だと言っています。このようなコンテナに関しては § 13.6.1 でさらに詳しく書きますが、ここでは vector<Core>と書けば、保持する要素の型は Core であってその派生クラスではないということだけ確認しておきましょう。

2 つのプログラムの型依存性を考えたときに、問題の半分は read と grade を仮想関数にしたことで解決されていることを見ました。残りの問題は、上で説明したように、これらの関数呼び出しが静的結合であるということなのです。これを実行時の結合、つまり動的結合にするためには、read と grade を Core のポインタか参照

を使って呼び出せばよいのです。ポインタか参照が実際に指すオブジェクトの型は、実行時に決まるからです。ここから4つのすべての問題を解決する方法がわかります。オブジェクトではなくポインタを使うようにプログラムを書き換えればよいのです。vector は vector<Core*>とし、record もポインタとして定義するのです。こうして、vector とローカル変数の型依存性は無くなり、プログラムは実行時に動作が決められるようになったのです。ただし、残念ながら、この方法はあとで見るように、面倒な処理をユーザに押し付けることになります。たとえば、次のようなコードは動作しません。

```
int main()
{
    vector<Core*> students;
    Core* record;
    while (record->read(cin)) {   // クラッシュ！
        // . . .
    }
}
```

これは record がオブジェクトを指していないからです。このプログラムの結果はおそろしいものです。

この問題は解決できますが、そのためには、ファイルからの読み込みで使うオブジェクトのメモリをユーザが管理しなければならないのです。ユーザは、また、読み込んでいるデータの種類を判別するコードを書かなければなりません。そこで、ファイルの学生データには、学部生か大学院生かを判別する記号が付いていることにします。つまり、大学院生のデータの先頭には G という文字があり、学部生のデータの先頭には U があるとするのです。

ポインタを使うようにプログラムを書き直す前に、もう1つ解決すべき問題があります。それは、どうやってポインタを要素に持つ vector を整列させるかです。簡単な答えは、Core オブジェクトを指すポインタを引数に取る新しい比較の関数を作るというものでしょう。ここで少し困るのは、新しい関数の名前を compare とはできないことです。§8.1.3 で説明したように、テンプレート引数に正しい型の値を与えるのには、微妙な問題がありました。同じような理由で、オーバーロードした関数をテンプレートの引数に使えないのです。もし使えば、コンパイラはどのバージョンの関数を使えばよいか判断できなくなるからです。あきらかに、オーバーロードではない別の名前を持った比較関数を sort に渡さなければなりません。それは次のようなものです。

```
bool compare_Core_ptrs(const Core* cp1, const Core* cp2)
{
    return compare(*cp1, *cp2);
}
```

比較関数を書いたので、プログラム全体を書き直すことができます。

```
// 以下のコードはまだ完成ではありません。§13.3.2を参照
int main()
{
    vector<Core*> students;   // オブジェクトではなくポインタを保持
    Core* record;             // 一時変数もポインタ
    char ch;
    string::size_type maxlen = 0;
    // データを読み込み、格納する
    while (cin >> ch) {
        if (ch == 'U')
            record = new Core; // Coreオブジェクトを生成
        else
            record = new Grad; // Gradオブジェクトを生成
        record->read(cin);     // 仮想関数の呼び出し
        maxlen = max(maxlen, record->name().size());   // データの取り出し
        students.push_back(record);
    }
    // ポインタを引数に取る比較関数を渡す
    sort(students.begin(), students.end(), compare_Core_ptrs);
    // 名前と最終成績を出力する
    for (vector<Core*>::size_type i = 0;
          i != students.size(); ++i) {
        // students[i]はポインタで、それから関数を使う
        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]; // 読み込み時に生成したオブジェクトを破棄
    }
    return 0;
}
```

このコードと元のコードとの違いはコメントで示しています。これらの違いはすべて、オブジェクトではなくポインタを扱うことから来ています。

while ループではまず入力から最初の文字を読み込み、その文字からその学生が学部生か大学院生か判断するようにしました。学生の種類がわかると、適当なクラスのオブジェクトを生成し、そのオブジェクトで read を実行しデータを標準入力から読み込みます。read 関数は仮想関数なので、record が Grad オブジェクトを指しているか Core オブジェクトを指しているかに従って、正しいバージョンの read 関数が呼び出されます。どちらの場合も、read 関数で次の入力をオブジェクトに読み込みます。ここでポインタである record について read を呼び出すには->演算子を使うことに注意してください。一番長い名前の長さを計算するときにもこの形式が使われています。しかし、この部分は、それ以外あまり変更されていません。

出力の while ループでは、students[i] がポインタを表すことに注意してください。students[i] の値はポインタなので、やはり->演算子を使わなければなりません。read 関数と同様に grade 関数も仮想関数であり、最終成績の計算では、オブジェクトが Grad かどうかによって、自動的に正しいバージョンの grade が呼び出されるのです。最後の変更箇所は、オブジェクトの使用していたメモリをシステムに返すということです。students[i] が保持するポインタに delete を適用して、そのメモリを解放しています。

13.3.2 仮想デストラクタ

プログラムはほとんど完成しました。最後の問題は、出力のループ内でしているように、delete でオブジェクトを破棄するときに起こります。これらのオブジェクトは Core クラスか Grad クラスのオブジェクトとして生成されました。しかし、どちらもそれを指すポインタの型は Core*でした。Grad オブジェクトであっても Core*型のポインタに結び付けられているのです。そのため、このポインタに delete を適用すると、Grad オブジェクトに対しても、Core のポインタへの適用になるのです。幸い、これは簡単に解決できる問題です。

ポインタに delete を適用すると、2 つのことが起こります。まずデストラクタがオブジェクトに対して働き、続いてそのオブジェクトが使っていたメモリを解放するのです students[i] が保持するポインタに delete を適用するとき、そのポインタは実際には Core オブジェクトか Grad オブジェクトのどちらかを指しているわけです。Core でも Grad でも明示的にデストラクタを書いていないので、delete が実行されるときには、コンパイラが生成したデストラクタが実行され、オブジェクトの使っていたメモリは解放されます。このコンパイラが生成したデストラクタは、クラスのデータメンバ 1 つ 1 つのデストラクタを実行します。しかし、delete が実行されるとき、システムはどちらのデストラクタを実行するのでしょうか。デストラクタは Core のメンバを破棄するのでしょうか、Grad のメンバを破棄するのでしょうか。そして解放されるメモリは、どのくらいでしょう。Core の分でしょうか、Grad の分でしょうか。

これらの問は、virtual の仕組みが解決してくれそうではありませんか。実際、そうなのです。デストラクタを仮想関数にするには、virtual を付けた仮想関数を定義する必要があります。

```
class Core {
public:
    virtual ~Core() { }
    // 以前と同じ
};
```

このようにしておけば、delete students[i] が実行されると、実行されるデストラクタは、students[i] が実際に指しているオブジェクトの型によるようになるのです。また、解放されるメモリも students[i] の指すオブジェクト分です。

デストラクタの定義の中には何も書きませんでした。Core の破棄で必要なことは、データメンバの破棄だけであって、これはシステムが自動で行ってくれるのです。空の仮想デストラクタはめずらしいものではありません。派生クラスのオブジェクトが基底クラスのポインタを通して破棄される場合に、仮想デストラクタは必要なのです。それだけの理由で仮想デストラクタを定義する場合、基底クラスのデストラクタは特別にすることはないのです。

ここで Grad クラスにデストラクタを付け加える必要はありません。すべての仮想関数がそうであるように、仮想デストラクタも継承されるからです。派生クラスでもオブジェクトの破棄で特別な処理が必要ないので、そこでデストラクタを再定義する必要もないのです。派生クラスでは基底クラスの virtual の性質を継承するので、するべきことはコンパイルのし直しだけなのです。

13.4 簡単なハンドルクラス

前の節で考えたアプローチはストレートですが、問題もありました。プログラムがポインタの管理に関して複雑になり、バグを呼び込みやすい落とし穴ができてしまったということです。クラスのユーザ（プログラマ）はデータを読み込むたびにオブジェクトを生成する必要があり、そのオブジェクトが不用になったら破棄しなければならないのです。また、プログラム中いたるところで、ポインタからオブジェクトを取り出しています。それにもかかわらず、2 種類の学生データが混在したファイルからデータを読み込み処理できるプログラムが書けたということも事実です。

ここでしたいことは、Core オブジェクトや Grad オブジェクトを扱っていた簡単なプログラムの良さを残しつつ、前節のプログラムの問題を取り除いて、両方のデータを処理できる新しいプログラムを書きたいのです。これは、よく知られたハンドルクラス（handle class）というテクニックを使えばよいことがわかります。ここではこれを紹介しましょう。

実行時まで正確な型がわからないオブジェクトを扱おうとして、コードが少々見づらくなってきました。そのオブジェクトの型は、Core クラスかその派生クラスであることがわかっています。前節ではポインタを使うことにしました。Core のポインタを用意して、それで Core と Grad の両方のオブジェクトを指し示すことができたからです。問題は、Core や Grad クラスのユーザに、間違いやすいポインタの管理を任せることになるということです。この管理をなくすことはできませんが、Core へのポインタをカプセル化するクラスを書くことで、ユーザから隠すことはできるのです。

```
class Student_info {
public:
    // コンストラクタとコピー管理
    Student_info(): cp(0) { }
    Student_info(std::istream& is): cp(0) { read(is); }
    Student_info(const Student_info&);
    Student_info& operator=(const Student_info&);
    ~Student_info() { delete cp; }
    // 関数
    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;
};
```

ここで重要なことは Student_info オブジェクトが Core と Grad の両方を表せるということです。その意味では、これはポインタのようなものです。しかし、Student_info のユーザは、これが表している Core か Grad オブジェクトのメモリ確保について悩む必要はないのです。このクラスが面倒でエラーを起こしやすい部分を処理してくれるからです。

Student_info オブジェクトは cp というポインタを内部に持っています。このポインタは Core でもその派生クラスでも指すことができます。§ 13.4.1 で見ますが、read 関数が cp の指すオブジェクトを生成するようにします。まず、どのコンストラクタでも最初に cp を 0 に初期化します。これは Student_info がまだどのオブジェクトも表していないということです。それから istream を引数に取るコンストラクタでは、Student_info::read 関数を呼びます。この関数は適当な型のオブジェクトを新しく生成し、istream から読み込んだ値をそのオブジェクトに代入します。

3 のルール (§ 11.3.6) から、ポインタを処理するため、コピーコンストラクタ、代入演算子、デストラクタが必要だということが言えます。デストラクタの仕事は簡単です。これは生成したオブジェクトを破棄するだけだからです。§ 13.3.2 で Core の仮想デストラクタを定義したので、Student_info のデストラクタは正しく動作し、オブジェクトが実際には Grad でも Core でも、それを正しく破棄します。コピーコンストラクタと代入演算子は、§ 13.4.2 で定義します。

ユーザは Student_info オブジェクトを Core オブジェクトや Grad オブジェクトの代わりに使うので、Student_info は Core と同じインタフェースを持たなければなりません。name と grade に関しては何も特別なことはありません。これらの関数は、cp の指す Core か Grad のオブジェクトに仕事を任せるだけです。

ただし、cp は 0 にもなり得ます。たとえば、Student_info のデフォルトコンストラクタが使われてオブジェクトが生成され、まだ read が実行されていない場合がそうです。cp が 0 のときは、仕事をオブジェクトに回すことができません。したがって、そのときは、問題が起こったことを示すため、runtime_error を投げることにしました。

ここで Core::grade が仮想関数であることに注意してください。したがって、cp に対してこの関数を呼ぶと、実行時にどのバージョンの関数が呼ばれるかは、cp の指すオブジェクト次第になります。たとえば、cp が Grad オブジェクトを指す場合、grade は Grad::grade という意味になります。

残りのインタフェースは compare 関数ですが、この関数には 2、3 の面白い点があります。まず Core クラスの定義を思い出してください。もともと compare はグローバル関数であってメンバ関数ではありませんでしたが、ここでは **static なメンバ関数** (static member function) として定義しました。static なメンバ関数は、個々のオブジェクトに作用しないという点で、他のメンバ関数とは違うものです。他のメンバ関数と違い、特定のオブジェクトではなくクラスそのものに付随した関数なのです。そのため、この関数はクラスの static でないデータメンバにアクセスすることはできません。この関数に付随するオブジェクトがないわけですから、そのオブジェクトのメンバも使えないわけです。

今の場合、static メンバ関数は重要な利点を持っています。それは、このような関数の名前はクラスのスコープ内で定義されているということです。そのため、compare がメンバ関数であるという場合、この関数の名前は Student_info::compare という意味です。関数の名前がこのように修飾されるため、これは最初に定義した Core を比較するための compare 関数のオーバーロードにはなりません。そのため、コンパイラが 2 つの関数を区別できるので、ユーザは sort に Student_info::compare を渡すことができるのです。

もう 1 つの面白い点は、この関数の定義です。この関数は Student_info::name 関数を使って、学生データの名前を取り出しています。ここで何が起こっているか考えておくことは重要です。cp が有効なオブジェクトを指す場合、Student_info::name の呼び出しは Core::name の呼び出しになります。しかし、cp が 0 なら name は例外を投げますが、これは compare を呼び出しているコードまで到達します。compare が Student_info のパブリックなインタフェースを使っているので、cp を直接チェックする必要はないのです。ユーザレベルのコードと同様に、チェックは Student_info クラスに任せているのです。

13.4.1 ハンドルを読む

read 関数には 3 つの責任があります。まず、このハンドルに結びついているオブジェクトがすでにあれば、これを破棄する必要があります。それから、読み込むべき学生の種類を決めなければなりません。そして、正しいオブジェクトを生成し、istream から読み込んだデータでこのオブジェクトを初期化しなければなりません。

```
istream& Student_info::read(istream& is)
{
    delete cp;    // もしあれば前のオブジェクトを破棄
    char ch;
    is >> ch;     // 学生データのタイプを読み込む
    if (ch == 'U') {
        cp = new Core(is);
    } else {
        cp = new Grad(is);
    }
    return is;
}
```

read 関数は、(もしあれば) このハンドルに結び付けられた前のオブジェクトを破棄するところからはじめます。delete を使う前に、cp が 0 かどうかをチェックする必要はありません。C++では、0 という特別な値を持ったポインタに delete を適用しても問題がないようになっているからです。前のオブジェクトを破棄すれば、新しいデータを読み込むことができます。まず、行頭の 1 文字を読み込み、チェックします。そして、この文字にしたがって、正しいオブジェクトを生成するのです。この際、istream を引数に取る適当なコンストラクタを使ってデータを初期化します。これらのコンストラクタは、自分のクラスの read 関数を呼び出し、入力ストリームから新しいオブジェクトにデータを読み込むのです。オブジェクトが生成された後は、そのアドレスを cp に格納します。そして最後に、読み込み用に与えられた入力ストリームを戻します。

13.4.2 ハンドルオブジェクトのコピー

Core のポインタを扱うために、コピーコンストラクタと代入演算子が必要です。コンストラクタは、オブジェクトを生成しそのアドレスをこのポインタに渡し、副作用として read を呼びます。Student_info のコピーを作るときには、新しいオブジェクトを生成し、コピー元のオブジェクトのデータをそのオブジェクトに与えます。しかし、思わぬ問題があります。それは、どの型のオブジェクトをコピーしているかです。コピーしようとしている Student_info が保持しているオブジェクトが Core オブジェクトか Grad のような Core の派生クラスのオブジェクトかを判断する方法はないのです。

この問題を解決するには、Core とその派生クラスに新しい仮想関数 clone を付け加えます。この関数は新しいオブジェクトを生成し、オリジナルの値をそのオブジェクトにコピーする関数です。

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

clone 関数は、驚くほど簡潔ですが、上述の機能を持った関数です。新しい Core オブジェクトを生成し、Core のコピーコンストラクタを使ってその値を与えています。Core クラスでは明示的にコピーコンストラクタを定義していませんでした。しかし、§ 11.3.5 で説明したように、存在するのです。つまり、コンパイラが、Core オブジェクトのメンバを新しいオブジェクトにコピーするコンストラクタを、自動で生成してくれるのです。

clone 関数は実装のツールとして後から付け加えたので、この関数は Core のパブリックなインタフェースに付け加えませんでした。clone が protected ということは、Student_info を Core の friend にしなければなりません。これにより Student_info オブジェクトが clone 関数を使えるようになるのです。クラスの friend 宣言は、§ 12.3.2 で見た関数の場合と同様です。そこでは、friend にした関数は、クラスの private と protected のメンバにアクセスできると書きました。クラスを friend にするのは、そのクラスの全メンバを friend にするのと同じ効果があります。つまり、

```
friend class Student_info;
```

を Core の定義に付け加えることで、Student_info のすべてのメンバ関数が Core クラスの private や protected のメンバにアクセスできるようになった、ということです。

基底クラスに仮想関数である clone を付け加えましたが、派生クラスでは、clone がその派生クラスのオブジェクトを生成するように、定義し直す必要があります。つまり、Grad クラスでは、clone が Grad オブジェクトを生成するようにするのです。

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

Core::clone と同様に、新しいオブジェクトを*this のコピーとして生成していますが、戻り値は Core*ではなく Grad*になっています。一般に、基底クラスの関数を派生クラスで定義しなおすなら、これは正確にパラメータリストと戻り値の型も一致するようにしなければなりません。しかし、基底クラスでそのクラスへのポインタ（参照でも同じ）を戻している場合は、派生クラスは派生クラスへのポインタ（参照でも同じ）を戻してよいのです。

ここで Student_info を Grad の friend にする必要はありません。friend は継承されないにもかかわらずです。これは、Student_info が Grad::clone を直接使わないからです。それはあくまで Core::clone の仮想関数呼び出しで使うだけだからです。Core で Student_info を friend にしたので、これは可能なのです。

以上の変更のもとに、コピーと代入を定義することができます。

```
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;
}
```

コピーコンストラクタでは、まず cp を 0 に初期化し、コピーするべきオブジェクトがある場合に限り、clone を呼び出しています。そうでない場合は、このハンドルは何も指していないという意味で cp は 0 のままになります。代入演算子でも、同様に clone を呼んでいます。もちろん、代入演算子ではその前にするべきことがありました。まず、自己代入は避けなければなりません。そのため、2 つのオペランドのアドレスが等しくないかチェックしているのです。ここで自分と異なるオブジェクトのコピーであることがわかったら、現在 cp が指しているオブジェクトを破棄してから、cp に新しいオブジェクトのアドレスを代入します。

コピーコンストラクタも代入演算子も cp が 0 のときは特別なことはしません。何も指していないハンドルをコピーしたり代入しても問題ないのです。

13.5 ハンドルクラスを使う

ハンドルクラスの定義が完成しました。§ 9.6 で考えたようなプログラムで、これを使うことができます。ここで変更は 1 箇所だけです。

```
int main()
{
    vector<Student_info> students;
    Student_info record;
    string::size_type maxlen = 0;
    // データを読み込み格納する
    while (record.read(cin)) {
        maxlen = max(maxlen, record.name().size());
        students.push_back(record);
    }
    // 学生のデータをアルファベット順に並べ替える
    sort(students.begin(), students.end(), Student_info::compare);
    // 名前と最終成績を出力する
    for (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;
}
```

このプログラムの入力のループは 2 種類のデータを読み込んでいます。それはこのコースを履修する最低条件のみを満たしている学部生のデータと、さらに論文を書いている大学院生のデータです。Student_info の read 関数はどちらのデータも読み込めるのです。これは最初の文字を読み込み、現在読み込み中のデータが学部生のものか大学院生のものかを判断し、それに基づいて istream で初期化した適当なオブジェクトを生成しているのです。これにより、Core か Grad のオブジェクトができあがるわけですが、そのアドレスは record 内の cp に格納されます。そして、この Student_info オブジェクトは vector である students に追加されますが、このときは Student_info のコピーコンストラクタが呼び出されてコピーされるのです。

次に sort を使って、データを整列させています。sort には Student_info::compare 関数を渡しています。この関数は基底クラスの name を呼び出し、オブジェクトの名前を比較するのでした。

出力のループは変更されていません。ループの各ステップで students[i] は第 i 番の Student_info オブジェクトを示していますが、オブジェクト内のポインタは Core か Grad のどちらかのオブジェクトを指しています。Student_info オブジェクト（ハンドル）に対して grade 関数を呼び出すと、そのハンドルが実際に指している Core か Grad オブジェクトの（仮想関数である）grade がポインタを通じて呼び出されるのです。ハンドルが指している実行時のオブジェクトの型によって、呼び出される関数のバージョンが決められるわけです。

最後に、Student_info オブジェクトの read で生成された内部のオブジェクトは、main 関数の終了時に自動で破棄されます。というのは、main 終了時に vector が破棄されるからです。vector のデストラクタが students の各要素を破棄し、そのためそれぞれの要素についてデストラクタが呼び出されるのです。デストラクタが呼び出されると、read が生成したオブジェクトが delete で破棄されます。

13.6　微妙な点

継承と動的結合の考え方は強力ですが、少なくとも最初は、奇妙に見えるかもしれません。ここまでこの考え方を使った例を見ましたが、関連して、しばしば問題を起こす微妙な点を見てみましょう。

13.6.1　継承とコンテナ

§ 13.3.1 で、Core オブジェクトをコンテナに入れたい場合、コンテナは正確に Core オブジェクトを保持するのだということを言いました。これには少し驚くかもしれません。コンテナには Core とその派生クラスのオブジェクトを入れたいと思うかもしれないからです。しかし、第 11 章の Vec の定義を思い出してください。Vec では格納するオブジェクトのメモリを確保していました。オブジェクトのためにメモリを確保する場合、そのオブジェクトの型が正確にわかっていないとだめなのです。オブジェクトの型判定とそのためのメモリ確保に仮想関数のような仕組みはないのです。

もし、vector<Core>オブジェクトを作って、そこに Core オブジェクトと Grad オブジェクトを格納しようとすると、さらに驚くかもしれません。それはできるのです。しかし、その結果が驚くべきものです。たとえば、

```
vector<Core> students;
Grad g(cin);              // Gradを読み込む
students.push_back(g);    // gのCore部分（！）をstudentsに格納
```

Core オブジェクトの参照が予想される場所で Grad オブジェクトを使うことができるので、students に Grad オブジェクトを格納することができます。push_back 関数は、vector の要素の型の参照を引数に取るので、g を push_back に渡すことができるのです。しかし、このオブジェクトを students に格納しようとすると、その Core の部分だけがコピーされてしまうのです。最初に見たときは驚くかもしれませんが、§ 13.2.2 で見たように、この振る舞いはもともと予想されるものです。students に格納された要素は Core オブジェクトとして振る舞うでしょう。vector は g の Core 部分だけをコピーして Core オブジェクトを生成し、それを students に格納するのです。このとき、Grad の他の部分は捨てられてしまいます。

13.6.2　どの関数を使いたいのか

基底クラスと派生クラスが同じ名前のメンバを持っていながら、それらのパラメータの型や数が異なると、それらは完全に無関係の関数として振る舞うことに注意してください。たとえば、学生の最終成績を変更できるアクセス関数を付け加えたとしましょう。Core ではこの関数は単に final を設定するだけの関数とし、Grad では 2 つの引数を持つ関数とします。そして、その第 2 引数は thesis の設定値とします。

```
void Core::regrade(double d) { final = d; }
void Grad::regrade(double d1, double d2) { final = d1; thesis = d2; }
```

ここで r を Core の参照とすると、次のようになります。

```
r.regrade(100);          // OK。Core::regradeの呼び出し
r.regrade(100, 100);     // コンパイルエラー。Core::regradeは引数を1つしか取らない
```

2 行目の関数呼び出しは、r が実際には Grad オブジェクトを指していてもエラーになります。r の型は Core の参照なので Core の regrade が呼ばれるのですが、これは double の引数を 1 つ取るだけだからです。

もっと驚くことは、r が Grad の参照の場合でしょう。

```
r.regrade(100);          // コンパイルエラー。Grad::regradeには引数が2つ必要
r.regrade(100, 100);     // OK。Grad::regradeの呼び出しになる
```

今度は Grad オブジェクトを指す r について実際に呼び出す関数を探すことになるのですが、Grad オブジェクトに対しては、regrade 関数は 2 つの引数を取る関数です。引数が 1 つの同名の関数が基底クラスにありますが、これは派生クラスの regrade によって隠されてしまうのです。もし基底クラスの関数を呼び出したいなら、次のように関数名を明示しなければならないのです。

```
r.Core::regrade(100);    // OK。Core::regradeの呼び出し
```

regrade を仮想関数にしたいなら、基底クラスでも派生クラスでも同じインタフェースにする必要があります。それには、デフォルト値付きで、実際には使われないパラメータを持った関数を Core に定義しておけばよいのです。

```
virtual void Core::regrade(double d, double = 0) { final = d; }
```

13.7 詳細

継承： わずかな違いを除いては似ているクラスを制作できる。

```
class base {
public:
    // 共通のインタフェース
protected:
    // 派生クラスからアクセスできるメンバ
private:
    // このクラスからしかアクセスできないメンバ
};

// 基底クラスのpublicなインタフェースは派生クラスのインタフェースの一部になる

class derived: public base { ... };
```

基底クラスから派生したクラスの関数は定義しなおすことができる。また、独自のメンバを付け加えることもできる。クラス全体を private に継承することもできる。

```
class priv_derived: private base { ... };
```

これはあまり使われず、通常、実装の際に便利な方法として使われるのみ。

(public に) 派生したクラスのオブジェクトは、基底クラスのオブジェクトとしても使え、また基底クラスの参照やポインタで表すこともできる。

派生クラスの派生クラスも作れる。

```
class derived2: public derived { ... };
```

derived2 オブジェクトは derived の部分を持ち、derived オブジェクトは base 部分を持つ。したがって、derived2 オブジェクトは derived と base の性質を持っている。

派生クラスオブジェクトの生成では、まずオブジェクト全体に必要なメモリが確保され、基底クラスの部分が生成され、それから派生部分が生成される。派生クラスオブジェクトの生成でどのコンストラクタが呼ばれるかは、そのオブジェクトを生成するときの引数によって決まる。派生クラスのコンストラクタは、初期化子リストによって、基底クラスのコンストラクタに引数を渡し、基底クラス部分のオブジェクトの生成をコントロールできる。初期化子で明示的に基底クラス部分を初期化しなければ、基底クラスのデフォルトコンストラクタが使われる。

動的結合： 関数を使っている実際の型に基づいて、実行時にその型の関数が選ばれる仕組み。動的結合は、実質的には、ポインタか参照を通して仮想関数を呼び出すときに起こる。仮想関数の仮想という性質は継承されるので、派生クラスで virtual を書く必要はない。

派生クラスで仮想関数を再定義する必要はない。もし、仮想関数を派生クラスで再定義しなければ、もっとも近い基底クラスのものが継承される。しかし、クラス定義内で宣言した仮想関数は、その定義を書かなければならない。仮想関数を宣言して定義を書かないと、しばしば不可思議なコンパイル時エラーになる。

オーバーライド: 派生クラスで、基底クラスのものと、同じ名前、同じ型で同じ数のパラメータ、const のありなしも同じ関数を定義すると基底クラスの関数が上書きされたことになる。これをオーバーライドという。この場合、一般には戻り値も一致していなければならない。しかし、§ 13.4.2 で見たように、基底クラスの関数がクラスのポインタ (または参照) を戻す場合、派生クラスは派生クラスのポインタ (または参照) を戻すことができる。引数の型や数が一致しなければ、基底クラスと派生クラスの関数は、同名であっても無関係と考えられる。

仮想デストラクタ： 基底クラスのポインタに delete が使われてオブジェクトが破棄される場合、そのオブジェクトは派生クラスのものかもしれない。そのような場合、仮想デストラクタが必要である。もし、仮想デストラクタで特別な処理が必要なくても、仮想デストラクタそのものは定義すべきで、その中身を空にしておけばよい。

```
class base {
public:
    virtual ~base() { }
};
```

他の関数と同様に、仮想デストラクタの仮想という性質は継承されるので、派生クラスで仮想デストラクタを再定義しなおす必要はない。

コンストラクタと仮想関数： オブジェクトが生成されつつあるとき、それが派生クラスの一部としてであっても、その型は生成されつつあるクラスの型である。そのため、コンストラクタの中で仮想関数を使うと、その生成されつつあるオブジェクトの型に静的に結合したバージョンの関数呼び出しになる。

クラスの friend 宣言： クラスは他のクラスを friend (フレンド) に宣言できる。これにより、そのクラスの全メンバ関数を friend にしたことになる。この性質は継承されたり伝播されたりはしない。つまり、friend の friend があっても、それはもとのクラスの friend ではない。

static メンバ： クラスの個々のオブジェクトのメンバとしてではなく、クラスに付随するメンバとして存在する。static なメンバ関数は static なデータメンバにしかアクセスできない。static なデータメンバの実体は、クラスに 1 つしか存在しない。これは、通常は、クラスのメンバ関数を定義するソースファイルで初期化しなければならない。クラス定義の外で初期化するので、その際には名前は完全修飾で書く。つまり

value-type class-name::*static-member-name* = *value*;

とすれば、これは、型が *value-type*、クラス名が *class-name* である static なメンバ *static-member-name* を *value* で初期化していることになる。