

課題

- 13-0 この章のプログラムをコンパイルし、実行し、試してください。
- 13-1 Core と Grad のコンストラクタで、これらが実行されるとその名前と引数リストを出力するように書き直してみてください。たとえば、Grad のコンストラクタが `istream&` 型の引数を取るとき、それを表示したいなら、次のようなステートメントを付け加えればよいわけです。

```
cerr << "Grad::Grad(istream&)" << endl;
// それから、このようなコンストラクタを使う小さいプログラムを書いてみてください。そのとき、出力がどうなるかを事前に考えておき、その予想と実際の出力が一致するまでプログラムか予想を直してください。
// 13-2 この章の Core と Grad クラスについて、以下の関数呼び出しでは、どのバージョンの関数が呼び出され
// るか考えてください。
Core* p1 = new Core;
Core* p2 = new Grad;
Core s1;
Grad s2;

p1->grade();
p1->name();
p2->grade();
p2->name();

s1.grade();
s1.name();
s2.name();
s2.grade();
```

`name` 関数と `grade` 関数に、その関数が呼び出されたことを示す出力のステートメントを書き込み、自分の予想が正しいかどうかチェックしてください。

- 13-3 第9章で書いたクラスには、有効な記録が入っているかどうかをチェックできる関数を付けました。この機能を、継承を使ったクラスに付けてください。

機能を、継承を使って ABC 評価に変換する関数を付け加えてください。

- 13-4 点数である最終成績を、§ 10.3 の方針を使って ABC 評価に変換する関数を付け加えてください。

- 13-5 個々の学生が単位取得に必要な条件をすべて満たしているかどうかをチェックする判定関数を書いてください。つまり、すべての宿題をしたか、また、大学院生ならさらに論文を提出したかをチェックする関数です。

- 13-6 成績として合否のみが判定される学生を表すクラスを書き足してください。そのような学生は宿題をしなくてもよいし、してもよいとします。もし宿題をするなら通常の計算式にしたがって最終成績を付けます。

もし宿題をしていないなら、最終成績は中間試験と期末試験の平均になります。60 点以上が合格です。

- 13-7 コースの聴講生を表すクラスを書き足してください。

- 13-8 上記 2 間の学生を加えた 4 種類の学生の成績を処理し出力するプログラムを書いてください。

- 13-9 § 13.4.2 の代入演算子が自己代入のチェックをしなかった場合、何が起こるでしょうか。

第14章

メモリを（ほとんど）自動で管理する

第13章で作ったハンドルクラス `Student_info` は、2つの抽象的なクラスを結びつけるものでした。このクラスは、学生のデータ操作のインターフェースになっているだけでなく、内部にデータを保持しているオブジェクトへのポインタを持ち、それを処理するのです。しかし、2つの独立した抽象クラスであるべきものを1つのクラスにするということは、しばしばデザインの弱さを示しているものです。

ここでは `Student_info` に似ているけれど、厳密にインターフェースを表すクラスを定義したいと思います。そのようなインターフェースクラスは C++ プログラムではよく見られるもので、特に、継承によって結びついている複数のクラスのインターフェースによく使われます。このインターフェースクラスとは別に、ポインタのようでありしかもメモリ管理など実質的な仕事をしてくれるクラスを作ります。このようにインターフェースクラスとポインタのような実装クラスを分けることで、この実装クラスを多数のインターフェースクラスが使えるようになるのです。

頻繁に起こるメモリ管理のパフォーマンスをよくすることもできます。それには、ポインタのようなクラスのオブジェクトで実際のデータを保持するオブジェクトを指すようにし、そのオブジェクトの不要なコピーを減らすのです。

この章の大部分は 1 つの質問に対する答を考えることにあてられるとも言えます。その質問は、オブジェクトのコピーとは何か、です。この質問の答はあきらかであると思えるかもしれません。オブジェクトのコピーとは、オリジナルの全性質を持った別のオブジェクトを作るということです。しかし、あるオブジェクトが他のオブジェクトと関係するようになるとたんに難しくなります。もし、オブジェクト `x` がオブジェクト `y` を参照していたら、`x` をコピーするときには `y` もコピーするのでしょうか。

場合によっては、この問にも簡単に答えられます。もし `y` が `x` のメンバなら、答は「はい」です。またもし、`x` がオブジェクト `y` を指すポインタなら、答は「いいえ」です。この章では、ポインタのようなクラスの 3 つのバージョンを考えます。これらのバージョンの違いは、コピーをどのように定義するかによるものです。

コピーに関する問題とポインタのようなクラスは、とても抽象的なものです。このような抽象的なクラスの定義を考えるので、当然、この章の内容はこの本の中でも群を抜いて抽象的になります。その結果、注意して勉強する必要があります。しかしながら、もちろん、それだけの価値があるのです。

14.1 オブジェクトをコピーするハンドル

第13章で考えた成績処理の問題をもう一度考え直してみましょう。この問題を解くのに、異なるタイプの学生を表すオブジェクトをまとめて保持し処理する必要がありました。これらのオブジェクトの型は、継承によっ

て結びついていて、後でさらに種類が増えることが予想されるのです。最初の方法は§13.3.1で紹介しましたが、異なる型のオブジェクトをまとめておくために、ポインタを使いました。ポインタはCoreオブジェクトかその派生クラスのオブジェクトを指すようにするのです。この場合、クラスのユーザは、これらのオブジェクトを動的に生成し、最後に破棄する責任を持たれます。プログラムはポインタの管理でごちゃごちゃし、一般に間違えやすくなります。

問題は、ポインタが原始的で低レベルなデータ構造だということです。ポインタを使ったプログラムはエラーを出しやすく悪名が高いものです。ポインタの多くの問題は、ポインタがそれの指すオブジェクトとは独立しているというところから来ています。これが落とし穴になるのです。

- 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。うっかり同じオブジェクトを指す2つのポインタを作ってしまうと、後で驚くようなことになる。
- ポインタの破棄がオブジェクトの破棄にはならない。これはメモリリークを起こしてしまう。
- オブジェクトを破棄してポインタだけ残すと、これはどこも指さないポインタになってしまう。このポインタを使うと動作がどうなるかわからない。
- ポインタを初期化しないで生成すると、そのポインタはオブジェクトを指していないことになる。このポインタを使うと動作がどうなるかわからない。

§13.5で、もう一度成績処理プログラムを考えてみました。このときもStudent_infoハンドルクラスを使いましたが、このクラスがポインタを処理するようにしたので、ユーザはポインタではなくStudent_infoオブジェクトを扱うだけよかったのです。しかし、一方で、Student_infoクラスはCoreとその派生クラスに強く結びついています。実際、このクラスはCoreとその派生クラスのpublicなインターフェースに対応するメンバ関数を持っています。

ここでしたいことは、これらの抽象クラスを分離するということです。インターフェースとしては依然としてStudent_infoを使いますが、「ハンドル」を管理するには別のクラスを使います。つまりこのクラスが実際のデータを持つオブジェクトへのポインタを管理するのです。この新しい型の振る舞いは、ハンドルが結び付けられるオブジェクトの型とは独立にでき、実際、独立にするのです。

14.1.1 一般的なハンドルクラス

これから作るクラスは、それが扱っているオブジェクトの型とは独立したものにするので、このクラスはテンプレートでなければなりません。このクラスはハンドルの振る舞いをカプセル化したものなので、名前はHandleにします。このクラスのサポートする性質は次のようなものです。

- Handleはオブジェクトを指す。¹
- Handle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表れる。つまりこのクラスを通じて

¹ 訳注：「ハンドル」は実際のデータを持っているオブジェクトを扱うためのものです。したがって、「ハンドル」はデータオブジェクトに「結び付けられる」ものです。この様子を、「ハンドルがオブジェクトを指す」「ハンドルがオブジェクトを表す」「ハンドルがオブジェクトに結びついている」と言います。これらはみな同義です。

14.1 オブジェクトをコピーするハンドル

て仮想関数を呼び出す場合、正しいバージョンの関数が動的に選択され実行される。これは本当のポインタを通じて関数を呼び出すのと同様になる。

Handleの使い方は限定されたものになります。Handleをオブジェクトに結びつけると、Handleがそのオブジェクトのメモリ管理をするようになります。ユーザはHandleを任意のオブジェクトに結びつけることができますが、1つのオブジェクトに結びつけるHandleは1つだけにしなければなりません。また、その後は、ポインタなどを使って直接オブジェクトにアクセスしてはいけません。すべてのアクセスはHandleを通して行うのです。このようにインターフェースを制限することで、組み込み型のポインタが持つ問題を受け継がなくて済んでしまうのです。Handleオブジェクトをコピーするときに、結びつけられたオブジェクトも新しくコピーし、新しいHandleがそのオブジェクトを指すようにします。Handleを破棄するときは、それが指しているオブジェクトも破棄しますが、これがそのオブジェクトを破棄する唯一の直接的方法になります。どのオブジェクトにも結びついていないHandleオブジェクトを生成することは可能ですが、そのHandleが指すオブジェクト（実際には、Handleはどのオブジェクトも指していない）を使おうとすると、例外を投げることにしました。また、例外を避けたい場合に、Handleが何かを指しているかを調べられますようにします。

これらの性質はStudent_infoクラスのものに似ています。Student_infoクラスのコピー・コンストラクタと代入演算子は、そのオブジェクトのclone関数を呼び出してコピーを作ります。Student_infoクラスのデストラクタはCoreオブジェクトも破棄します。オブジェクトを使う場合、その前にStudent_infoが実際にオブジェクトに結びついているかをチェックします。今考えているクラスは、このような振る舞いをカプセル化したものです。しかも、どのような型にも使えるものです。

```
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& s);
    ~Handle() { delete p; }
    Handle(T* t): p(t) { }
    operator bool() const { return p; }
    T& operator*() const;
    T* operator->() const;
private:
    T* p;
};
```

Handleクラスはテンプレートであるため、どのような型に対してもHandleが使えるのです。このHandle<T>はオブジェクトを指すポインタを持ち、いろいろな操作はこのポインタを通じて行います。変数名の変更は別にして、最初の4つの関数はStudent_infoのものと同じです。デフォルトコンストラクタはポインタを0に設定し、このHandleが何も指していないことを示します。コピー・コンストラクタは（コピー元にオブジェクトがある場合）オブジェクトのclone関数を呼び出して新しいコピー・オブジェクトを作っています。Handleのデストラクタはオブジェクトを破棄しています。代入演算子は、コピー・コンストラクタと同様に、（コピー元にオブジェクトがある場合）オブジェクトのclone関数を呼び出して新しいコピー・オブジェクトを作ることにします。

```
template<class T>
Handle<T>& Handle<T>::operator=(const Handle<T> rhs)
```

```

{
    if (&rhs != this) {
        delete p;
        p = rhs.p ? rhs.p->clone() : 0;
    }
    return *this;
}

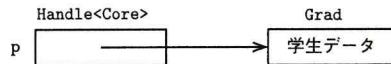
```

また、代入演算子は、いつものように、最初に自己代入をチェックします。そして自己代入の場合は何もしません。自己代入でない場合は、これまで保持していたオブジェクトを破棄し、右辺のオブジェクトのコピーを生成します。コピーのステートメントでは、条件演算子（§ 3.2.2）を使い、`clone` の呼び出しが安全かどうかをチェックしています。`rhs.p` が 0 でないなら、`rhs.p->clone` を呼び出し、その結果を `p` に代入します。そうでなければ、`p` を 0 にしています。

`Handle` はポインタを真似ているので、ポインタを実際のオブジェクトに結びつける方法が必要です。これは `T*` を引数に取るコンストラクタがすることになります。コンストラクタは与えられたポインタを記録し、これで `Handle` オブジェクトがそのポインタが指すオブジェクトに結びついたことになります。たとえば、

```
Handle<Core> student(new Grad);
```

すると、`Handle` オブジェクト `student` が生成され、その内部の `Core*` ポインタが、ここで生成されている `Grad` オブジェクトを指すように初期化されるのです。ただし、`Handle<Core>` を使う場合には、`Handle<Core>` が `Core` の `clone` にアクセスできるように書き直す必要があります。そのためには、`Handle<Core>` を `Core` の `friend` にするか、`clone` を `Core` の `public` メンバにするとよいでしょう。



最後に、3つの演算子関数の定義があります。まず `operator bool()` ですが、これは `Handle` が有効なポインタを持っているかどうかを戻す関数です。もし `Handle` がオブジェクトを指していれば `true` を戻し、そうでなければ `false` を戻します。他の2つの定義は、`operator*` と `operator->` のものですが、これらは `Handle` に結びついたオブジェクトにアクセスするためのものです。

```

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

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

```

14.1 オブジェクトをコピーするハンドル

}

組み込みの演算子`*`をポインタに適用すると、そのポインタの指すオブジェクトが得られます。ここでは独自の`*`を定義しているわけですが、これを `Handle` オブジェクトに適用すると、それが保持しているポインタに組み込みの`*`を適用したものに戻すようにしてあります。`student` オブジェクトに対して`*student` は、`student.p` に`*`を適用したものになるのです（ここでは `p` にアクセスできるとして説明しています）。つまり上の例では、`*student` は `student` の初期化で生成した `Grad` オブジェクトへの参照になるわけです。

`->` 演算子はもう少し複雑です。表面的には`->` は2項演算子のように見えますが、実際には普通の2項演算子とは違います。スコープ演算子やドット演算子`(.)` のように、`->` は左オペランドのメンバである右オペランドへのアクセスに使われます。そしてオペランドに使われる名前はエクスプレッションではないのです。ユーザが要求するような名前に直接アクセスすることはできません。代わりに、C++の言語仕様では、`->` はポインタとして扱えるものを戻すことになっているのです。`operator->` を定義すると、それを持つオブジェクト `x` に対して、

`x->y`

は、

`(x.operator->())->y`

と同じ意味になるのです。今考えているコードでは、`operator->` は、内部のポインタを戻します。そのため、`student` に対して、

`student->y`

は

`(student.operator->())->y`

と同じ意味になり、これは `operator->` の定義により、

`student.p->y`

と同じ意味になるのです（`student.p` には直接アクセスできませんが、そのような意味になるということです）。そこで`->` 演算子を使うと、`Handle` オブジェクトに対する関数呼び出しの要求を、その `Handle` が持つポインタに渡すことになるのです。

組み込みのポインタに対して働く多態性の仕組みが `Handle` にも受け継がれることが1つの目標でした。`operator*` と `operator->` の定義を見ると、すでにそのようになっていることがわかります。これらは参照かポインタを戻すので、それにより動的結合が成立するのです。たとえば、`student->grade()` を実行すると、これは `student` 内部の `p` に対して `grade` を呼び出すことになります。この場合、どのバージョンの `grade` が呼び出されるかは、`p` が指している実際のオブジェクトの型によって決まるのです。`student` が上の例のように `Grad` オブジェクトを指しているなら、これは `Grad::grade` の呼び出しになります。同様に、`operator*` は参照を戻すので、`(*student).grade()` を実行すると、`grade` は参照を通して呼び出されることになり、これも、どのバージョンの `grade` が呼び出されるか実行時に決められることになります。

14.1.2 一般的なハンドルを使う

§ 13.3.1 のポインタを使ったプログラムを Handle を使ったプログラムに書き直すことができます。

```
int main()
{
    vector< Handle<Core> > students; // 型を変更
    Handle<Core> record; // 型を変更

    char ch;
    string::size_type maxlen = 0;
    // データを読み込んで格納
    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); // Handle<T>::->からreadの仮想関数呼び出し
        maxlen = max(maxlen, record->name().size()); // Handle<T>::->
        students.push_back(record);
    }
    // compareはconst Handle<Core>&で働く関数に直す必要がある
    sort(students.begin(), students.end(), compare_Core_handles);
    // 名前と最終成績を出力
    for (vector< Handle<Core> >::size_type i = 0;
         i != students.size(); ++i) {
        // students[i]がHandleオブジェクト。それに対して関数呼び出しをする
        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;
        }
        // もはやdeleteステートメントは不要
    }
    return 0;
}
```

このプログラムは Handle<Core> オブジェクトを Core* オブジェクトの代わりに扱います。§ 13.3.1 で見たように、const Handle<Core>& へ働くオーバーロードではない比較関数を定義し、sort に渡さなければなりません。この関数は compare_Core_handles という名前としますが、その定義はみなさんの課題にしましょう。

オリジナルのプログラムと違っているところは、たった一箇所、出力のループだけです。students の Handle オブジェクトからデータを取り出すのに、Handle<T>::operator-> を使って Core* を取り出し、そのポインタに対しても name や grade を使っています。たとえば、students[i]->grade() の呼び出しでは、オーバーロードした->演算子の働きで、実質的には students[i].p->grade() を実行することになります。grade は仮想

14.1 オブジェクトをコピーするハンドル

関数なので、students[i].p が指すオブジェクトの実際の型に基づいたバージョンの grade が呼び出されます。また、Handle がメモリ管理をするので、students の各要素が指すオブジェクトに delete を適用する必要はなくなりました。

さらに重要なことなのですが、ポインタの扱いを Handle に任せることで Student_info を定義し直し、これを純粋なインターフェースクラスにすることができます。

```
class Student_info {
public:
    Student_info() { }
    Student_info(std::istream& is) { read(is); }
    // コピー、代入、デストラクタはない：もう必要ないから
    std::istream& read(std::istream& is) {
        std::string name() const {
            if (cp) return cp->name();
            else throw runtime_error("uninitialized Student");
        }
        double grade() const {
            if (cp) return cp->grade();
            else throw runtime_error("uninitialized Student");
        }
        static bool compare(const Student_info& s1,
                            const Student_info& s2) {
            return s1.name() < s2.name();
        }
private:
    Handle<Core> cp;
};
```

このバージョンの Student_info では、cp は Handle<Core> であって Core* ではありません。そのため、コピー管理関数を定義する必要はなくなりました。Handle がオブジェクトのメモリを管理するからです。他のコンストラクタの働きも前のものと同様です。name と grade 関数は前と同じに見えますが、cp のチェックをしていて、これを bool 値に変換したものによって実行が変わります。また、ハンドルが指しているオブジェクトの持つ関数が実行されるようにオーバーロードされた->演算子が使われています。

再定義を完成させるために、read 関数の定義を書く必要があります。

```
istream& Student_info::read(istream& is)
{
    char ch;
    is >> ch; // 学生の種類を読み込む
    // 正しい型のオブジェクトを生成する
    // ポインタからHandle<Core>オブジェクトを生成するのにHandle<T>(T*)を使う
    // Handle<Core>を左辺に代入するのにHandle<T>::operator=を使う
    if (ch == 'U')
        cp = new Core(is);
    else
        cp = new Grad(is);
    return is;
}
```

このコードは前の `Student_info::read` に似ていますが、実行はまったく違います。一番あきらかのは、`delete` ステートメントがなくなっていることです。これは `cp` の代入によって、不要なオブジェクトの破棄が自動で行われるからです。このコードを理解するには、注意深く読んでいかなければなりません。たとえば、`new Core(is)` では `Core*` オブジェクトが得られますが、ここで `Handle(T*)` コンストラクタが使われ、非明示的に `Handle<Core>` に変換されています。それからこの `Handle` オブジェクトは `cp` に代入されます。この際、`cp` がすでにオブジェクトを指しているなら、それは自動的に破棄されます。なお、この代入は、不要に `Core` オブジェクトのコピーを生成したり破棄したりしています。次にこのような不要なコピーを避ける方法を考えましょう。

14.2 参照カウント付きハンドル

ここまでで、ポインタの処理とインターフェースの分離に成功しました。その過程で作った `Handle` クラスは、いろいろなインターフェースクラスの実装に使えます。そうすることで、インターフェースクラスはメモリ管理を考えずに済むようになったのです。しかし、`Handle` クラスはまだデータを持つオブジェクトのコピーと代入に問題があります。それは必要のない場合でもコピーをしてしまうということです。`Handle` をコピーするときは、いつでも `Handle` の指すオブジェクトもコピーするように書いたからです。

一般には、このようなコピーをするべきかどうか選べるようにしたいものです。たとえば、コピーしたオブジェクト同士が、同じ内部データを共有するようにしたいこともあるでしょう。このようなクラスは値のような振る舞いをする必要がないか、あるいは望ましくもないのです。また、1度生成したオブジェクトの値を変えることができないようなクラスも考えられます。このようなクラスでは、実際にデータを持つオブジェクトをコピーする理由がありません。そのようなクラスで、コピーをするのは実行時間とメモリ空間の無駄使いです。このようなクラスをサポートするために、ハンドルクラスのオブジェクトをコピーしても、内部でデータを保持しているオブジェクトはコピーされないようにハンドルクラスを作ります。同じオブジェクトに結びついた複数のハンドルというものを許したときは、正しいタイミングでそのオブジェクトを破棄することも必要です。オブジェクトを破棄すべき明らかなタイミングは、そのオブジェクトに結びついている最後のハンドルが消滅するときです。

この目的のため、参照カウンタ（カウント）（reference count）というものを使います。これはあるオブジェクトがいくつのハンドルオブジェクトに結びついているかを保持するものです。個々のデータオブジェクトには、それに付随する参照カウンタがあるようにします。そのオブジェクトに結びついたハンドルを生成するたびに参照カウンタを1増やすことにし、そのオブジェクトに結びついたハンドルが消滅するたびに参照カウンタを1減らすことにします。すると、そのオブジェクトに結びついた最後のハンドルが消滅するときに、参照カウンタは0になります。このときこそ、安全にデータオブジェクトを破棄できるわけです。

このテクニックを使うと、たくさんの不要なメモリ管理やデータのコピーを防ぐことができます。ここでは `Ref_handle` という新しいクラスを作ってみましょう。これは、`Handle` クラスに参照カウンタを付けたものと考えられます。続く2つの節では、値のように振舞いながらデータを共有するクラスを作るのに、参照カウンタがどのように使われるかを見ます。

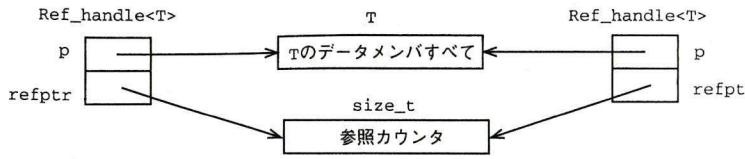
クラスに参照カウンタを付けるには、カウンタを作り、生成、コピー、破棄の関数を書き換え、カウンタを適当に変化させるようにします。`Ref_handle` が結びついているデータオブジェクトは、それに付随する参照カウンタを持ちます。唯一の問題は、このカウンタをどこに置くかです。一般には、`Ref_handle` オブジェクトを結

びつけるクラスのコードを私たちが持っているとは限りません。そのため、データを保持するクラスに単純にカウンタを付けることはできないのです。その代わり、`Ref_handle` に参照カウンタへのポインタを持たせることにします。`Ref_handle` に結び付けられたデータオブジェクトには、どれだけ `Ref_handle` オブジェクトがコピーされているかを記録する参照カウンタを付随させます。

```
template <class T> class Ref_handle {
public:
    // ポインタと同様、参照カウンタも管理する
    Ref_handle(): refptr(new size_t(1)), p(0) { }
    Ref_handle(T* t): refptr(new size_t(1)), p(t) { }
    Ref_handle(const Ref_handle& h): refptr(h.refptr), p(h.p)
    {
        +++refptr;
    }
    Ref_handle& operator=(const Ref_handle&);
    ~Ref_handle();
    // 以前と同様
    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;
    size_t* refptr; // 付け加えた
};
```

`Ref_handle` クラスに2番目のデータメンバを付け加えました。そして、コンストラクタでこのメンバも初期化するようになりました。デフォルトコンストラクタと `T*` を引数に取るコンストラクタは新しい `Ref_handle` オブジェクトを生成します。そのときに、(型が `size_t` である) 参照カウンタをメモリ上に生成し、その値を1にしています。コピーコンストラクタは新しいデータオブジェクトを生成しません。その代わりに引数の `Ref_handle<T>` オブジェクトのポインタをコピーし、参照カウンタを1つ増やします。これは、新しい `Ref_handle` オブジェクト内のポインタが、右辺のオブジェクト内のポインタが指しているオブジェクトを指すようになり、結局そのオブジェクトを指す `Ref_handle` の数が1増えることになるからです。こうして、新しい `Ref_handle<T>` オブジェクトは、コピー元と同じ `T` オブジェクトと同じ参照カウントを指すようになります。たとえば、`X` を `Ref_handle<T>` オブジェクトとし、`Y` を `X` のコピーとして生成したとすると、その構造は

下図のようになります。



代入演算子も、データを表す T オブジェクトのコピーはせず、参照カウンタを操作するだけです。

```

template<class T>
Ref_handle<T>& Ref_handle<T>::operator=(const Ref_handle& rhs)
{
    ++rhs.refptr;
    // 条件がそろえれば、左辺のオブジェクトのポインタが指すオブジェクトを破棄する
    if (--*refptr == 0) {
        delete refptr;
        delete p;
    }
    // 右辺から値をコピー
    refptr = rhs.refptr;
    p = rhs.p;
    return *this;
}

```

いつものように、自己代入に伴うトラブルを避けることは重要です。これは左辺のオブジェクトの参照カウンタを 1 減らす前に、右辺の参照カウンタを 1 増やすことで行っています。もし、同じオブジェクトを指していたとしても、結果としては参照カウンタを変えないことになり、したがって、意図しないうちに参照カウンタが 0 になることは避けられるのです。

参照カウンタが減らされて 0 になると、左オペランドがそのデータオブジェクトに結びついている最後の Ref_handle オブジェクトということになります。そして、そのオブジェクトに別のデータを入れようとしているから、それは元のものを破棄するのです。そこで refptr の指す参照カウンタと p の指すデータオブジェクトを delete で破棄します。p の指すオブジェクトも refptr の指すオブジェクトも動的に生成されたものなので、メモリリークを防ぐためには delete で破棄しなければならないのです。

これらを破棄したあとに、右辺のデータメンバの値を左辺のデータメンバにコピーします。そして、左オペランドへの参照を戻すのです。

デストラクタは、代入演算子と同様に、破棄される Ref_handle がデータオブジェクトを指す最後のオブジェクトかどうかをチェックします。そして、その場合は、データメンバであるポインタの指すオブジェクトを delete で破棄します。

```

template<class T> Ref_handle<T>::~Ref_handle()
{
    if (--*refptr == 0) {
        delete refptr;
        delete p;
    }
}

```

}

このバージョンの Ref_handle は、コピーしたオブジェクト同士が同じデータオブジェクトを持てるような場合にはうまく働きます。しかし、値のような振る舞いが望まれる Student_info ではどうでしょうか。たとえば、Student_info の実装に Ref_handle を使い、次のようなコードを書くとどうなるでしょう。

```

Student_info s1(cin); // s1を標準入力から得たデータで生成
Student_info s2 = s1; // その値をs2に「コピー」

```

こうすると、s2 が s1 のコピーに見えても、s1 と s2 が同じデータオブジェクトを指すことになります。どちらかのデータを書き換えると、もう片方のデータも変わってしまうのです。

§ 14.1.1 で定義した元の Handle クラスは、コピーの時には clone で新しいオブジェクトを生成するので、値のように振る舞います。しかし、あきらかに Ref_handle クラスは clone 関数を使っていません。clone 関数を使わないから、このハンドルに結びついているオブジェクトをコピーで生成することはないのです。一方、Ref_handle は不要なデータコピーを避けるという長所も持っています。問題は、不要かどうかにかかわらず、コピーによる生成はすべて避けているということです。どうすればよいのでしょうか。

14.3 データを共有するかどうかを決められるハンドル

ここまで 2 種類の一般的なハンドルの定義を見てきました。最初のものは、コピーのたびにデータオブジェクトもコピーするもので、次のものは決してデータオブジェクトはコピーしないというものです。もっと有用なハンドルは、これを使うプログラムがデータオブジェクトをコピーするかどうか決められるものです。そのようなハンドルクラスは Ref_handle のパフォーマンスを保ちながら、Handle の振る舞い、つまり値のような振る舞いをするクラスになります。これは組み込みのポインタの有用さを持ちながら、たくさんの落とし穴を回避するものとも言えます。そこで、この最終版のハンドルの名前は、ポインタ (pointer) の代わりになるものという意味で、Ptr とします。Ptr クラスでは、データの内容を変更したいときにそのデータオブジェクトを指している別の Ptr オブジェクトがあれば、そのオブジェクトをコピーしてから変更することにします。参照カウンタがあるので、データオブジェクトを指す Ptr クラスの数がわかるのです。

Ptr クラスの基本構造は、§ 14.2 で書いた Ref_handle クラスの基本構造と同じです。ここですることは、ユーザにメモリ管理の指示をさせるメンバ関数を 1 つ増やすことだけです。

```

template<class T> class Ptr {
public:
    // 必要な場合、条件によってオブジェクトのコピーをするメンバ関数
    void make_unique() {
        if (*refptr != 1) {
            --*refptr;
            refptr = new size_t(1);
            p = p? p->clone(): 0;
        }
    }
    // 残りの部分は名前以外はRef_handleクラスと同じ
    Ptr(): refptr(new size_t(1)), p(0) { }
    Ptr(T* t): refptr(new size_t(1)), p(t) { }
}

```

```

Ptr(const Ptr& h): refptr(h.refptr), p(h.p) { ***refptr; }
Ptr& operator=(const Ptr&); // 定義は§14.2と同様
~Ptr(); // 定義は§14.2と同様
operator bool() const { return p; }
T& operator*() const; // 定義は§14.2と同様
T* operator->() const; // 定義は§14.2と同様

private:
    T* p;
    size_t* refptr;
};

新しいメンバの make_unique は、先に書いた仕事をする関数です。つまり、参照カウンタが 1 なら何もせずに、そうでなければハンドルの指すオブジェクトの clone 関数を使ってそのコピーを作り、p がそれを指すようにするのです。もし参照カウンタが 1 でなければ、そのデータオブジェクトを指す Ptr オブジェクトが自分以外にもあるということになります。その場合は、そのオブジェクトの参照カウンタを 1 減らし（それにより参照カウンタは 1 になるかもしれませんのが 0 にはなりません）します。それからこのハンドルに対して新しい参照カウンタを生成します。この参照カウンタは、将来このハンドルがコピーされるときに使われるのですが、最初はもちろん（このデータオブジェクトを指すハンドルは自分 1 つだけなので）、1 に初期化します。ただし、clone 関数を呼び出す前に、コピーを作ろうとしているオブジェクトが実際に存在しているかをチェックしています。もしそうであれば、clone 関数でそのオブジェクトをコピーするわけです。その後は、p がそのデータオブジェクトを指している唯一の Ptr オブジェクトになります。このデータオブジェクトは（参照カウンタが 1 だった場合）元のものと同じかもしれませんし、（参照カウンタが 1 より大きかった場合）元のもののコピーかもしれませんわけです。

```

このハンドルの最終バージョンである Ptr を、§14.1.2 で書いたハンドルを使う Student_info で使うことができます。この場合、どの関数もデータオブジェクトの値を途中で変えるものではないので、名前を変える以外に変更の必要がないことがわかります。Student_info クラスでデータを変える関数は read だけですが、この関数は常に新しい Ptr メンバを生成するのです。そのときに、Ptr の代入が起こりますが、Ptr の代入演算子は、古いデータオブジェクトを指す Ptr オブジェクトが複数あるかどうかで、それを破棄するか残すか決めます。しかし、いずれにしても、Student_info オブジェクトは、読み込み用に新しい Ptr オブジェクトを生成し、当然その参照カウンタは 1 のはずなのです。ユーザが次のようなコードを書いたとしましょう。

```

Student_info s1;
read(cin, s1); // s1に値を入れる
Student_info s2 = s1; // s2にs1の値をコピー
read(cin, s2); // s2に読み込み、s1はそのままs2のみが変更される

read の呼び出しの後で、s2 の値は変更されますが、s1 の値は変更されません。
一方、§13.6.2 で見たような、仮想関数バージョンの regrade 関数を Core とその派生クラスに定義し、さらに Student_info クラスにもそのインターフェースを定義していたなら、その関数は make_unique を呼び出すように変更しなければなりません。

void Student_info::regrade(double final, double thesis)
{
    // データオブジェクトを変更する前にコピーしておく
    cp.make_unique();
}

```

```

if (cp)
    cp->regrade(final, thesis);
else throw run_time_error("regrade of unknown student");
}


```

14.4 コピーがコントロール可能なハンドルの改良

コピーをコントロールできる Ptr ハンドルは便利なものです。すべての望ましい機能があるわけではありません。たとえば、第12章で考えた Str クラスを Ptr を使って定義し直してみましょう。§12.3.4 では、2つの Str オブジェクトをつなげるときには、たくさんの文字のコピーを非明示的にしていることを見ました。参照カウンタ付きの Str クラスを使うことで、いくつかのコピーは避けられると考えるかもしれません。

```

// これはうまくいくでしょうか？
class Str {
public:
    friend std::istream& operator>>(std::istream&, Str&);
    Str& operator+=(const Str& s) {
        data.make_unique();
        std::copy(s.data->begin(), s.data->end(),
                  std::back_inserter(*data));
        return *this;
    }
    // インターフェースは以前と同じ
    typedef Vec<char>::size_type size_type;
    // コンストラクタの定義は変えてPtrを生成するようにした
    Str(): data(new Vec<char>) {}
    Str(const char* cp): data(new Vec<char>) {
        std::copy(cp, cp + std::strlen(cp),
                  std::back_inserter(*data));
    }
    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));
    }
    // 必要なときにはmake_uniqueを使う
    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(); }
private:
    // vectorを指すPtrオブジェクト
    Ptr<Vec<char> > data;
}; // §12.3.2, §12.3.3 にあるように定義
std::ostream& operator<<(std::ostream&, const Str&);
Str operator+(const Str&, const Str&);


```

`Str` のインターフェースは変えていませんが、実装は根本的に変えました。`Vec` オブジェクトを直接保持する代わりに、`Vec` オブジェクトを指す `Ptr` オブジェクトを保持するようにしました。これより、複数の `Str` オブジェクトが同じデータオブジェクトを共有できるようになりました。コンストラクタは `Vec` オブジェクトを適当に初期化して生成し、`Ptr` オブジェクトをこれに結び付けます。読み込み専用でデータを変えない関数の定義は前と同じです。もちろん、これらの関数は `Ptr` に対して実行されるので、`Ptr` 内のポインタを通じて文字データを取り出さなければなりません。おもしろいのは、入力演算子、結合演算子、`const` でない `operator[]` 演算子といった、`Str` を変更する演算子関数です。

たとえば、`Str::operator+=`を見てください。これはすでにある `Vec` オブジェクトにデータを追加する演算子なので、`data.make_unique()` を呼び出しているのです。この関数呼び出しの後なら、`Str` オブジェクトの `Ptr` は、自由に変更してよいオブジェクトのコピーを指すようになるわけです。

14.4.1 コードを変更できない型のコピー

しかし残念ながら、`make_unique` の定義には重大な問題があります。

```
template<class T>
void Ptr<T>::make_unique()
{
    if (*refptr != 1) {
        --*refptr;
        refptr = new size_t(1);
        p = p? p->clone(): 0; // ここに問題あり
    }
}
```

`p->clone` の呼び出しを見てください。今は `Ptr<Vec<char>>` を使っているので、これは `Vec<char>` のメンバ関数 `clone` を呼び出すことになります。しかし、`Vec` にそんなメンバ関数はないのです。

ここまで仕掛けでは、`clone` は `Ptr` が指すオブジェクトのメンバ関数でなければなりません。そうすることで仮想関数の仕掛けが使えるからです。言い換れば、`Ptr` がすべての派生クラスに対しきちんと働くためには、`clone` という関数がそのクラスに定義されていることが非常に重要なのです。しかし、また、`Vec` の定義を変えることはできません。`Vec` は `vector` の機能を部分的に実現するクラスとして書いたものです。`vector` には `clone` などという関数はありません。これを `Vec` に付け加えると、`vector` の機能の一部を実現するという前提も壊れてしまいます。どうすればよいのでしょうか。

このような困った問題への解答は、しばしば、冗談めかして私たちがいう、ソフトウェアエンジニアリングの基本定理にあります。それは、すべての問題は間接度の度合いを上げれば解決できる、というものです。問題は、存在しないメンバ関数を呼び出そうとしていること、そしてそのメンバ関数を新たに書き加えることができないということでした。解答は、直接メンバ関数を呼び出すのではなく、オブジェクト生成のために呼び出せるグローバル関数を定義し、それを間に使うというものです。この関数の名前も `clone` とします。

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

そして、`make_unique` メンバの定義を変えます。

```
template<class T>
void Ptr<T>::make_unique()
{
    if (*refptr != 1) {
        --*refptr;
        refptr = new size_t(1);
        p = p? p->clone(): 0; // メンバではなくグローバルなcloneの呼び出し
    }
}
```

このような関数を定義し、`make_unique` の定義に挟み込んで、その動作が変わらないことはあきらかだと思います。グローバルな `clone` の呼び出しは、コピーされるオブジェクトのメンバ関数である `clone` の呼び出しになるからです。`make_unique` は、メンバでない `clone` を呼び出し、その `clone` が `p` の指すオブジェクトについてそのメンバ関数を呼び出す、というように、間接度の度合いを上げたのです。`Student_info` のように `clone` をメンバに持っているクラスについては、これは何の得にもなりません。しかし、`Str` で見たように、`clone` をメンバに持たないオブジェクトを `Ptr` が指すような場合には、これが問題解決につながるのです。ここで次のような関数を定義します。

```
// Ptr< Vec<char> >を有効にするための関数
template<class T>
Vec<char>* clone(const Vec<char>* vp)
{
    return new Vec<char>(*vp);
}
```

`template<class T> T* clone(const T* tp)` を最初に書くことで、この関数はテンプレート特化 (template specialization) であることを示します。このようにすることで、特定の引数型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のです。これで、`Vec<char>` のポインタを引数にしたときは、他のポインタを引数にしたときと異なる動作をするように定義できることになります。`clone` に `Vec<char>*` 型引数を渡すと、コンパイラは上記の特別バージョンの `clone` を呼び出すのです。他のポインタを引数に渡すと、コンパイラはテンプレートの `clone` から、「引数のポインタに対して `clone` メンバ関数を呼び出す」関数を具体化します。特化したバージョンでは、引数から `Vec<char>` のコピー構造関数を使って、新しい `Vec<char>` オブジェクトを生成します。このような `clone` の特化では仮想関数の仕組みが使えませんが、`Vec` の派生関数は考えないのでこれは必要ありません。

ここにしたこととは、メンバ関数 `clone` への依存を弱めたということです。それはこの関数がメンバ関数として存在しない可能性もあるからです。間接度を上げることで、特定のクラスのコピーの仕方を `clone` の定義で指示できるようにしました。メンバ関数、コピー構造関数、あるいはまったく違う方法を使うように決められるのです。テンプレート特化がなければ、`Ptr` クラスはデータオブジェクトの `clone` メンバ関数を使います。しかし、それは `make_unique` が使われるときだけです。これは次のようにまとめることができます。

- `Ptr<T>` は使っても `Ptr<T>::make_unique` を使わないなら、`T::clone` は定義されていなくてもかまわない。
- `Ptr<T>::make_unique` を使い、しかも `T::clone` が定義されているなら、`make_unique` は `T::clone` を使う。

- `Ptr<t>::make_unique`を使い、しかも（たぶん、それがないなどの理由で）`T::clone`を使いたくないなら、`clone<T>`を特化して好きなように定義すればよい。

間接度の度合いを上げて `Ptr` の振る舞いを細かくコントロールすることができるようになりました。最後に残るのはもっと難しい部分です。それは、そもそも何をしたいかはっきり決めるということです。

14.4.2 コピーが必要なのはいつ？

この例の最後の部分は詳細に調べ直す価値があります。`operator[]` の2つのバージョンを見直してみましょう。片方は `data.make_unique` を呼び出し、片方は呼び出しません。この違いは何でしょう。

この違いは、その関数が `const` メンバかどうかに関連しています。あとの方の `operator[]` は `const` なメンバ関数です。これはオブジェクトの内容を変えないという約束です。そして、`const char&`を呼び出し側に戻すことでのこの約束を守っています。そのため、データオブジェクトである `Vec<char>` オブジェクトを複数の `Str` で共有していても問題が無いのです。いずれにしても、ここで得た値を使ってもとの `Str` を書き換えることはできないからです。

一方、もう1つの `operator[]` は `char&` を戻します。これを使えば、ユーザは `Str` の内容を変更することもできるわけです。その場合、たまたま同じデータオブジェクトを共有している `Str` にまで変更の影響が行かないようにしなければなりません。そのため、`make_unique` を呼び出してデータオブジェクトを今の `Str` オブジェクト専用のものとし、これを変更しても他の `Str` オブジェクトに影響が無いようにしているのです。そして、それからその `Vec` 内の文字への参照を戻しているわけです。

14.5 詳細

テンプレート特化： 特別なテンプレートの定義のように見えるが、1つ以上の型パラメータを省略し、変わりに特定の型を使う。いろいろなテンプレート特化の使い方はこの本の範囲を超えてしまう。しかし、そのようなものがあるということ、またコンパイル時の型の決定に有用であることは知っておくべきである。

課題

- 14-0 この章のプログラムをコンパイルし、実行し、試してみてください。
- 14-1 `Ptr<Core>` の比較関数の定義を書いてください。
- 14-2 `Ptr<Core>` オブジェクトを使って学生の成績処理プログラムを書き、試してみてください。
- 14-3 最終バージョンの `Ptr` を使って `Student_info` クラスの定義を書き直し、それを使って § 13.5 の成績処理プログラムを書き直してください。
- 14-4 最終バージョンの `Ptr` を使って `Str` クラスの定義を書き直してください。
- 14-5 上の間の `Str` クラスに関して、`Vec<Str>` を使った `split` や文字の絵の関数を考え、それらを使ったプログラムをコンパイルし、試してください。
- 14-6 `Ptr` クラスは実際に2つの問題を解決しています。それは参照カウンタの保持とオブジェクトの生成・破棄の問題です。参照カウンタだけを考え、他には何もしないクラスを定義してください。それからそのクラスを使って `Str` の定義を書き換えてください。