

第 12 章

値のように振る舞うクラスオブジェクト

組み込み型のオブジェクトは、一般に値のように振舞います。たとえば、組み込み型のオブジェクトをコピーすると、オリジナルとコピーは同じ値を持ちますが、独立したものになります。片方のオブジェクトを変えても、もう片方が影響を受けることはありません。このような型のオブジェクトは、生成したり、関数に渡したり、関数から受け取ったり、また別のオブジェクトにコピーしたり、代入したりできます。

組み込み型の大部分には、たくさんの操作が可能で、論理的に近いもの同士の自動的な変換もあります。言語がそのように作られているからです。たとえば、int に double を足すと、コンパイラは自動的に int を double に変換するのです。

自分でクラスを定義する場合は、そのオブジェクトが値のように振る舞うように、クラスの定義をコントロールすることもできます。第 9 章と第 11 章で見たように、クラスの制作者は、そのオブジェクトの生成、コピー、代入、破棄の仕方を決めることができます。特に、コピーと代入を適当に定義することで、クラスの制作者はそのクラスのオブジェクトを値のようにできます。つまり、オブジェクト同士が独立しているようにできるわけです。Vec と Student_info クラスは値のようなオブジェクトを持つクラスの例です。

この章では、クラスの制作者がクラスオブジェクトの変換と関連する関数を決め、これらをさらに組み込み型のオブジェクトのようにできることを見ます。これは、標準ライブラリの string がよい例になっています。string には多くの関数と自動的な型変換があるからです。したがって、この章では、string の簡単化したもの Str を考えます。これは第 11 章で vector を簡単化したものを考えたのに似ています。この章では、string を使ったエクスプレッションに含まれる操作や変換に着目しましょう。ただし、効率は考えません。その代わり、第 14 章で Str を考え直し、そのオブジェクトの使うメモリを効率よく管理するテクニックを紹介します。

実は Str の実装について悩む必要はありません。その大部分は Vec クラスの実装で、すでに行っています。この章の大部分は、適切なインターフェースをクラスに定義する方法を議論することになります。

12.1 単純な文字列のクラス

まず、だいたい期待通りにオブジェクトを生成できる Str クラスを考えてみましょう。

```
class Str {
public:
    typedef Vec<char>::size_type size_type;
    // 空のStrオブジェクトを生成するデフォルトコンストラクタ
    Str() {}
```

```

// n個のcのコピーを含むオブジェクトを生成する
Str(size_type n, char c): data(n, c) {}
// ヌル文字で終わる文字の配列からオブジェクトを生成する
Str(const char* cp) {
    std::copy(cp, cp + std::strlen(cp), std::back_inserter(data));
}
// 反復子bとeで囲まれた範囲をコピーしてオブジェクトを生成する
template<class In> Str(In b, In e) {
    std::copy(b, e, std::back_inserter(data));
}
private:
    Vec<char> data;
};

このクラスはそのデータの管理を第11章で書いたVecクラスに任せています。Strの実装には、Vecでほぼ十分です。ただし、Vecにclearではなく、その追加は第11章では課題にしておきました。
Strには4つのコンストラクタがありますが、それぞれがVecオブジェクトの内部データであるdataを適当に処理し、Vecオブジェクトを初期化しています。
Strのデフォルトコンストラクタは、非明示的にVecのデフォルトコンストラクタを呼び出し、空のStrオブジェクトを生成しています。このデフォルトコンストラクタの仕事は、コンパイラが自動で生成してくれるデフォルトコンストラクタと同じですが、他のコンストラクタを定義しているので、ここには明示的に書かなければならなかったのです。他の3つのコンストラクタは、dataを生成したり初期化したりするために必要な値を受け取るコンストラクタです。
サイズと文字を受け取るコンストラクタは、Vecの対応するコンストラクタを使ってdataを生成しています。それ以上にすることはないので、定義の中身は空です。
最後の2つのコンストラクタは互いに似ています。これらのコンストラクタには初期化子がありません。これはdataが非明示的に空のVecとして生成されるということです。それぞれのコンストラクタはcopyを使い、与えられた文字列をdataの中にコピーしています。たとえば、const char*を引数に取るコンストラクタは、strlen関数で文字列の長さを決めています。この長さと入力文字列の最初を指す反復子から、範囲の末尾を示す反復子を計算し、その範囲の文字列をcopyで空のdataに追加しています。ここでdataへの追加にはback_inserterを使っています。これにより、cpで表される文字配列の文字がすべてdataに格納されることになるのです。
最後のコンストラクタが一番おもしろいものです。引数に2つの反復子を取り、それらの間にある文字列を、新しく生成したStrオブジェクトにコピーしているのです。前のコンストラクタと同様にcopyとback_inserterを使い、[b, e)の値をdata入れています。このコンストラクタでおもしろいところは、これ自身がテンプレート関数であるということです。これがテンプレートであるため、いろいろなコンストラクタを効率よくまとめて定義できるのです。たとえば、このコンストラクタは文字配列からも、あるいはVec<char>からもStrオブジェクトを生成できるのです。
このクラスにはコピー構造体、代入演算子、デストラクタがないことに注意してください。なぜだかわかりますか。
```

それは、コンストラクタが生成してくれるもので十分だからです。Strクラス自身はメモリの確保をしていません。コンパイラが自動で生成してくれる関数が、実際にはdataの対応する関数を呼び出すので、それに任せ

てよいのです。たとえば、これがどのように働くのか、デストラクタを例に考えてみてください。もし、デストラクタを付けてもする仕事はないはずです。一般に、デストラクタが必要でないクラスでは、コピー構造体と代入演算子もわざわざ書く必要はないのです(§11.3.6)。

12.2 自動の変換

ここまででコンストラクタを定義し、コピー、代入、破棄もどうするか決めました。これらの関数のおかげでStrオブジェクトは値のようになります。つまり、Strオブジェクトをコピーすると、オリジナルとコピーが独立したものになるわけです。すると、次の問題は型の変換です。組み込み型では、しばしば自動で型変換が行われます。たとえば、int型の値でdouble型の変数を初期化することができます。また、int型の値をdouble型の変数に代入することもできます。

```

double d = 10;      // 10をdoubleに変換し、それでdを初期化
double d2;
d2 = 10;           // 10をdoubleに変換し、それをdに代入

```

今考えているStrクラスでは、const char*からStrを生成するには、

```
Str s("hello"); // sの生成
```

と書くことができます。ここではconst char*を引数に取るコンストラクタを明示的に使ってsを生成しています。また、次のように書くこともできます。

```

Str t = "hello"; // tを初期化
s = "hello";     // 新しい値をsに代入

```

§11.3.3で説明しましたが、=という記号には上のように異なる2つの意味があることを思い出してください。最初のステートメントはtの定義であり、=は初期化を意味します。この形の初期化は常にconst Str&を引数に取るコピー構造体を使います。2番目のステートメントは定義ではありません。ここでは=は代入演算子を意味します。その代入演算子はコンパイラが生成するものです。この引数もconst Str&です。つまり、上の2つの例は、どちらもconst Str&の来るべきところに、文字列リテラルを使っているのです。

Strに新たにconst char*を引数に取る代入演算子を定義し、コピー構造体のオーバーロードも考えるべきだと思うかもしれません。しかし、幸い、何もする必要はないのです。それは、Strにはすでにconst char*を引数に取るコンストラクタが定義されているからです。このコンストラクタがユーザ定義の型変換(user defined conversion)になるのです。ユーザ定義の型変換とは、クラスオブジェクトをどのように型変換するかを示すものです。組み込み型のときと同様に、コンパイラは必要なときに値の型をユーザ定義の型変換に基づいて変換するのです。

クラスの型変換には2方向があります。1つは他の型からその型への変換であり、もう1つはその型から他の型への変換です。後者の変換は§12.5で説明しましょう。より一般的なのは、今考えている他の型からの変換です。これは引数が1つだけのコンストラクタを定義することで定義されるのです。

Strにはすでにそのようなコンストラクタがあったのです。それが、const char*を引数に取るコンストラクタです。Strが必要な場所でconst char*が使われていると、コンパイラがこのコンストラクタを使うのです。const char*のStrへの代入ではまさにこれが行われています。s = "hello";とあるとき、実際にはコンバ

イラが `Str(const char*)` を使い無名のローカルな `Str` オブジェクトを文字列リテラルから生成しているのです。それから、(コンパイラが生成した) `Str` の代入演算子を使って、その無名オブジェクトを `s` に代入するわけです。

12.3 Str の演算子

ここで `string` で書いたコードを考えてみましょう。

```
cin >> s    // 入力演算子を使いstringに読み込み
cout << s    // 出力演算子を使いstringを出力
s[i]        // インデックス演算子を使い文字にアクセス
s1 + s2    // +演算子を使い2つのstringをつなげる
```

これらはすべて2項演算子です。そのため、これらを関数として定義するなら、これらは2つの引数を取るようになります。さらにこれらをメンバ関数とするなら、引数の1つは非明示的になります。§ 11.2.4で見たように、演算子をオーバーロードする場合は、`operator`というキーワードにその演算子を付けて、関数名とします。たとえば、`operator>>`が入力演算子をオーバーロードする関数の名前になり、`operator[]`がインデックス演算子の関数の名前になり、というわけです。

ここではインデックス演算子から始めましょう。これは§ 11.2.4で一度実装を見ているからです。この関数はクラスのメンバ関数でなければなりません。

```
class Str {
public:
    // コンストラクタは前と同様
    char& operator[](size_type i) { return data[i]; }
    const char& operator[](size_type i) const { return data[i]; }
private:
    Vec<char> data;
};
```

このインデックス演算子は、対応する `Vec` の演算子を呼び出しているだけです。`Vec` のときと同様に、インデックス演算子には2つのバージョンがあることに注意してください。1つは `const` オブジェクトに対して使えますが、もう1つの方は使えません。文字への参照を戻すことで、`const` でないバージョンでは戻す文字を書き換えることができます。`const` バージョンは、`const char`への参照を戻すので、ユーザが参照している文字を書き換えることはできません。ここで `char` ではなく `const char&` を戻しているのは、標準の `string` クラスに合わせるためにです。

他の演算子はどうしましょう。このとき一番面白いのは、これらを `Str` クラスのメンバ関数とするかどうかを判断することです。この間に答えようとすると、個々の演算子の特性を考えることになります。そこでまず、入出力演算子を考え、§ 12.3.3で文字列をつなぐ演算子を考えることにします。

12.3.1 入出力演算子

§ 9.2.2で `compare` を `Student_info` のメンバにするかどうかを決めました。1つの決め方は、その演算子がオブジェクトの状態を変えるかどうかによると書きました。入力演算子は確かにオブジェクトの状態を変えま

12.3 Str の演算子

す。入力演算子は、すでにあるオブジェクトに新しい値を読み込むために使うものだからです。したがって、入力演算子は `Str` クラスのメンバ関数がよいと考えるかもしれません。しかし、そうすると期待通りの動作をしなくなるのです。

この理由を考えるのに、エクスプレッションのオペランドがオーバーロードされた演算子関数にどのように結び付けられるかを思い出してみましょう (§ 11.2.4)。2項演算子の場合、それがメンバ関数でなければ、左オペランドは常に始めの引数と解釈され、右オペランドは2番目の引数とされます。演算子がメンバ関数の場合、最初のパラメータ (左オペランド) はいつもメンバ関数に非明示的に渡されるものです。そのため、

```
cin >> s;
```

とあれば、これは

```
cin.operator>>(s);
```

のよう `cin` のオーバーロードされた演算子 `>>` を使うのと同じ意味になります。これから、`>>` 演算子は `istream` クラスのメンバでなければならないとわかります。

もちろん、`istream` の定義を変えることはできませんので、この演算子を付け加えることもできません。もし、`operator>>` を `Str` のメンバ関数にすれば、次のようにそれが使えます。

```
s.operator>>(cin);
```

あるいは、これは

```
s >> cin;
```

と同じです。これは標準ライブラリの使用方式と違ってしまいます。そのため、入力、そして同様に出力演算子もメンバ関数にできないと結論できるのです。

そこで入出力演算子の宣言を加えて `Str` クラスの定義があるファイル、`Str.h`などを更新しましょう。

```
std::istream& operator>>(std::istream&, Str&);           // 追加した
std::ostream& operator<<(std::ostream&, const Str&); // 追加した
```

出力演算子の定義は簡単に書けます。これは `Str` のデータである文字を1つずつ書き出すだけです。

```
ostream& operator<<(ostream& os, const Str& s)
{
    for (Str::size_type i = 0; i != s.size(); ++i)
        os << s[i];
    return os;
}
```

ただ、上のように演算子を定義するには、`Str` に新しくサイズを戻す関数 `size` を付け加える必要があります。

```
class Str {
public:
    size_type size() const { return data.size(); }
    // 以前と同様
};
```

出力演算子の定義は簡単ですが、すべてをきちんと理解する必要があります。ループの各ステップでは、`Str::operator[]`を使って出力する文字を取り出しています。この演算子は、内部データである `Vec` から `Vec::operator[]` を使って文字を取り出しているのです。同様に、ループの各ステップで `Str` のサイズを知るために `s.size()` を呼び出しています。これはさらに `Vec` のメンバ関数 `size` を呼び出し、そのサイズを報告させていることになります。

12.3.2 フレンド

入力演算子の定義も、出力演算子に比べてとても難しいということはありません。これは入力ストリームから文字を読み込み保持するものです。入力演算子を使うと、最初にある空白は破棄してから読み込みを始め、空白かファイルの終わりに到達するまで読み込みを続ける必要があります。ここでは入力演算子を少し簡単化して、この本の範囲を超えた入出力演算子の微妙な機能は考えないことにします。今必要な機能だけを付けるということです。

```
// これはまだコンパイルできません
istream& operator>>(istream& is, Str& s)
{
    // 現在のデータを破棄
    s.data.clear();
    // 空白を読んで破棄
    char c;
    while (is.get(c) && isspace(c))
        ; // 条件のテスト以外何もしない
    // 読み込むべきものがあれば次の空白まで読み込む
    if (is) {
        do s.data.push_back(c); // コンパイルエラー！dataはprivate
        while (is.get(c) && !isspace(c));
        // 空白を読み込んだならストリームに戻しておく
        if (is)
            is.unget();
    }
    return is;
}
```

この関数の説明をしてから、なぜコンパイルできないかを説明します。

この関数は、まず、`data` が持っている前のデータを破棄するところから始めます。それは `Str` に何かを読み込めば前のデータは破棄されるべきだからです。それから、意味のある文字の前にあるかもしれない空白を破棄するため、1 文字ずつ与えられたストリームから、空白でない文字が現われるまで読み込んでいきます。ここで読み込んだ文字が空白かどうかを見る必要があるので、入力には `get` 関数を使います。オーバーロードした `>>` 関数は空白を無視しますが、`get` 関数は空白文字であっても、1 文字ずつ順にストリームから読み込んでいくのです。そのため、`while` ループは、空白でない文字を見つけるか、入力がなくなるまで読み込みを続けることになります。ここでは読み込んだ文字が空白文字であった場合、何もせずに読み込みを続けます。そのため、`while` ループの中には何も書いていないのです。

`if` の条件は `while` が終了したのが、空白でない文字に到達したからなのか、ファイルの終わりに到達したからなのかをチェックしています。もし、前者なら、今度は次に空白文字に到達するまで文字を読み込み、`data`

に付け加えていきたいのです。これは次の `do while` ループ（§ 7.4.4）で行っています。前のループで読み込んだ最後の文字も `data` に付け加え、それからファイルの終わりか空白が出るまで文字を読み続けます。空白でない文字を読み込むたびに、`push_back` 関数を使って `data` に文字を付け加えています。

その後、`is` から読み込みができなくなるか、空白を読み込むと `do while` ループは終了します。後者の場合、すでに 1 文字（空白文字）を読み込みすぎたことになるので、これを `is.unget()` でストリームに戻します。`unget` 関数は `get` で最後に読み込んだ文字を入力ストリームに戻してしまう関数です。`unget` を呼び出した後は、ストリームはその前の `get` がまったく実行されなかったのと同じ状態に戻ります。

実は、コメントしてあるように、このコードはコンパイルできません。問題は `operator>>` が `Str` クラスのメンバでないため、`s` のデータメンバである `data` にアクセスできないのです。§ 9.3.1 でも、`comapare` 関数が `Student_info` オブジェクトのメンバである `name` にアクセスする必要があるという、同じような問題に直面しました。そのときはアクセス関数を付け加えることで問題を解決しました。しかし、今は `data` に読み込み専用のアクセス関数を付け加えるだけでは足りません。入力演算子は `data` を読み込むのではなく、`data` にデータを書き込むからです。入力演算子は `Str` という抽象化の一部であるので、`data` に書き込みアクセスさせることは問題ありません。しかし、すべてのユーザが `data` に書き込みできるようにはしたくありません。そのため、`operator>>` に（そして他のユーザに）利用可能な、`data` にデータを入れる `public` メンバを付け加えることはできないのです。

そこで（`public` な）アクセス関数を付け加えるのではなく、入力演算子を `Str` クラスの `friend`（フレンド）というものにすればよいのです。`friend` に指定されたものは、そのクラスのメンバと同様に他のメンバにアクセスできるようになります。つまり、入力演算子を `friend` に宣言し、メンバ関数のように `Str` クラスの `private` メンバにアクセス可能にすることができるのです。

```
class Str {
    friend std::istream& operator>>(std::istream& is, Str& s);
    // 以前と同様
};
```

これは `Str` クラスの定義に `friend` 宣言を付け加えたものです。この宣言は、`istream&` と `Str&` を引数に取るバージョンの `operator>>` は、`Str` の `private` メンバにもアクセスできるという意味です。この宣言を `Str` に付け加えることで、入力演算子のコンパイルが可能になります。

`friend` 宣言はクラスの定義のどこに書いてもかまいません。これが `private` や `public` のラベルの後にあっても何も変わりません。`friend` 関数は特別なアクセスができるので、クラスのインターフェース部分と考えることができます。そこで、`friend` 宣言は、クラス定義の始めの方で、`public` インタフェースのそばにまとめて置くのがよいでしょう。

12.3.3 他の 2 項演算子

`Str` クラスでまだ残っているのは+演算子の実装です。しかし、その前に、いくつか決めるべきことがあります。この演算子はメンバであるべきでしょうか。そのオペランドの型は何でしょうか。戻り値は何にしましょうか。後で説明しますが、これらの間には微妙な意味があるのです。

これらの間の答を推量してみます。やりたいことは、2 つの `Str` オブジェクトをつなげられるようにすることです。また、つなげるときに両方のオペランドの内容は変えません。これらから、この演算子をメンバにする特

別な理由は見当たりません。さらに、いくつかの文字列を続けてつなげていけるようにしたいのです。これは `s1`、`s2`、`s3` を `Str` オブジェクトとすると、次のようなエクスプレッションを可能にしたいということです。

```
s1 + s2 + s3
```

このように使うためには、この演算子の戻しものは `Str` であるとわかります。

以上の考察から、この演算子は次のようなメンバでない関数として実装することになります。

```
Str operator+(const Str&, const Str&);
```

この定義の前に少し考えてみると、`operator+`を定義するなら `operator+=`も定義したほうがよいと思うかもしれません。たとえば、`Str` クラスのユーザが、`s` と `s1` をつなぎだものをあらためて `s` にしたいと考えるかもしれません。これは次のような形にしたいものです。

```
s = s + s1;
s += s1;
```

実は、後で、`operator+`の定義を書くには、`operator+=`を先に書いたほうが簡単だとわかります。単純な+演算子と違って、複合型演算子である`+=`は左辺のオペランドを変えてしまいます。そのため、これは `Str` クラスのメンバ関数にしましょう。文字列をつなぐ新しい関数の定義を付け加えて、`Str` の最終版は次のようにになります。

```
class Str {
    // §12.3.2で定義した入力演算子
    friend std::istream& operator>>(std::istream&, Str&);
public:
    Str& operator+=(const Str& s) {
        std::copy(s.data.begin(), s.data.end(),
                  std::back_inserter(data));
        return *this;
    }

    // 前と同じ
    typedef Vec<char>::size_type size_type;
    Str() {}
    Str(size_type n, char c): data(n, c) {}
    Str(const char* cp) {
        std::copy(cp, cp + std::strlen(cp), std::back_inserter(data));
    }
    template<class In> Str(In i, In j) {
        std::copy(i, j, std::back_inserter(data));
    }
    char& operator[](size_type i) { return data[i]; }
    const char& operator[](size_type i) const { return data[i]; }
    size_type size() const { return data.size(); }
private:
    Vec<char> data;
};

// §12.3.2で定義したもの
std::ostream& operator<<(std::ostream&, const Str&);
Str operator+(const Str&, const Str&);
```

12.3 Str の演算子

内部のデータ保持に `Vec` を使っているので、`+=`の定義は簡単です。`copy` 関数を使って右オペランドのコピーを左オペランドである `Vec` オブジェクトに付け加えているのです。また、普通の代入演算子と同様に、左オペランドのオブジェクトを戻すことになりました。

すると `operator+`の定義は `operator+=`を使って簡単に書けます。

```
Str operator+(const Str& s, const Str& t)
{
    Str r = s;
    r += t;
    return r;
}
```

`Str` をつなぐのはメンバでない関数であり、それは新しい `Str` を生成します。新しい `Str` は、`r` というローカルな変数で `s` をコピーして生成します。この初期化では、`Str` のコピーコンストラクタが使われます。次に、`+=` 演算子を使って `r` に `t` をつなぎ、`r` を（また非明示的ですが、コピーコンストラクタを呼んで）戻しています。

12.3.4 複合エクスプレッション

const `Str&` 型のオペランドを取る文字列結合演算子を定義しました。エクスプレッションに文字へのポインターが含まれる場合はどうなるでしょうか。たとえば、§ 1.2 のプログラムで `Str` を使ったらどうなるでしょう。プログラムには次のようなコードがありました。

```
const std::string greeting = "Hello, " + name + "!";
```

ここで `name` は `string` オブジェクトです。同様に次のようなコードを書きたいと思います。

```
const Str greeting = "Hello, " + name + "!";
```

ここで `name` は `Str` とします。`+` 演算子は左結合なので、このエクスプレッションは、まず、

```
"Hello, " + name
```

を評価し、その結果と「`!`」に`+`演算子を適用するのと同じです。これは次のようにも表現できます。

```
("Hello, " + name) + "!"
```

エクスプレッションをこのように分解することで、2つの異なる`+`があることがわかります。1つは文字列リテラルが左オペランドになり `Str` オブジェクトが右オペランドになるというものです。もう1つは、文字列を結合した結果である `Str` オブジェクトが左オペランドになって、文字列リテラルが右オペランドになるものです。左右が逆になんでも、どちらも `const char*` と `Str` に対して、`+` 演算子を適用していると言えます。

§ 12.3.3 では引数の型が `Str` のものを定義しましたが、`const char*` のものは定義しませんでした。しかし、§ 12.2 で見たように、`const char*` 型の引数を取るコンストラクタを定義することで、`const char*` から `Str` への変換が定義されたことになります。あきらかに、`Str` クラスはこのようなエクスプレッションを処理できるのです。`+` 演算子のどちらの場合でも、コンパイラが `const char*` 型の引数を `Str` に変換し、それから`+` 演算子を呼び出すことになります。

このような型変換の意味を理解することは重要です。たとえば、

```
Str greeting = "Hello, " + name + "!";
```

は、次のように書いた場合の `greetings` と同じ値を持つことになるのです。

```
Str temp1("Hello, ");
Str temp2 = temp1 + name;           // operator+(const Str&, const Str&)
Str temp3("!")
Str greeting = temp2 + temp3;      // operator+(const Str&, const Str&)
```

このようにたくさんの一時オブジェクトを使うところを見ると、この方法は効率が悪いと考えるかもしれません。実際、このような一時オブジェクトを生成するコストを考えて、商用の開発環境で使われる `string` は、しばしば、コンストラクタによる自動の型変換を利用せず、個別の型に対するバージョンを定義するという手間のかかる方法をとっています。

12.3.5 2項演算子をデザインする

2項演算子の定義における型変換の役割は重要です。もしクラスが型変換を定義しているなら、普通は2項演算子をメンバ関数でないように定義するのがよいでしょう。そうすることで、オペランドの型を左右入れ替えても使えるからです。

もし演算子がクラスのメンバなら、左オペランドには、自動の型変換の結果で生成されるオブジェクトを使うことはできません。その理由は、プログラマが書いた `x + y` のようなエクスプレッションに対し、コンパイラは `operator+` をメンバに持つなんらかの型に `x` を変換できるかどうかのチェックをしないからです。コンパイラ（とプログラマ）は非メンバである `operator+` 関数と `x` のクラスのメンバである `operator+` 関数のみをチェックするのです。

メンバでない演算子の左オペランドと、いずれの演算子でもその右オペランドは、普通の関数の引数と同じルールに従います。これらのオペランドは、パラメータの型に変換できる型ならなんでもよいのです。もし2項演算子をメンバ関数として定義すると、その左右オペランドに許される型が異なってしまうのです。つまり右オペランドは型変換でパラメータの型になればよいのですが、左オペランドはそうはできないわけです。これは`+=` のように最初から非対称な演算子では問題になりません。しかし、対称性が予想される演算子に対しては、混乱とエラーを招いてしまいます。そのような演算子は、たいていの場合、左右オペランドに対して対称に定義することが望ましいのです。そして、それはその演算子をメンバでない関数として定義することによって実現できるのです。

ただし、2項演算子の中でも代入演算子については、左オペランドをそのクラスのオブジェクトに限定すべきでしょう。そうでなければ何が起こるでしょうか。左オペランドに型変換を許すと、オペランドをクラスオブジェクトに変換し、新しい値をその一時オブジェクトに代入することになります。しかし、それは一時オブジェクトですから、代入が完了すると、今代入したばかりのオブジェクトにアクセスする方法がなくなるのです。したがって、代入演算子自身を含めて、`+=`などの複合代入演算子はクラスのメンバであるべきなのです。

12.4 変換が危険な場合

§ 11.2.2 でサイズを引数にとるコンストラクタに `explicit` 宣言をしたことを思い出してください。今、引数を1つしか取らないコンストラクタは型変換を表すということを学んだので、コンストラクタを `explicit` に

することの意味を理解できるようになりました。`explicit` にすることで、コンパイラに「このコンストラクタは引数を明示的に受け取る場合のみ働く」と宣言したわけです。コンパイラは、エクスプレッションや関数において、`explicit` なコンストラクタを、非明示的な型変換に使わないのです。

この `explicit` コンストラクタの重要性を見るために、`Vec` のコンストラクタが `explicit` でないと仮定してみましょう。するとサイズから非明示的に `Vec` オブジェクトが生成できてしまいます。このような変換が、§ 5.8.1 の `frame` のような関数の呼び出しで起こってしまうのです。この関数は、引数を1つしか取らずその型は `const vector<string>&` で、入力の `vector` 内の文字にフレームを付けたものを戻す関数です。今は `Vec<string>&` を引数に取る同様の `frame` を考えたとします。もし、整数を引数に取る `Vec` のコンストラクタが `explicit` でなければ、

```
Vec<string> p = frame(42);
```

ということもできてしまいます。これは何でしょうか。何が起るべきでしょうか。更に重要なことは、ユーザはこれで何を予想するでしょうか。

この場合、42列の空行からなる絵にフレームが付いたものが生成されるのです。これはユーザの期待した結果でしょうか。ユーザは「42」という数字の周りにフレームを付けたかったのではないでしょうか。このような関数呼び出しが、間違いの可能性があるので、`Vec` クラスの整数を引数に取るコンストラクタは `explicit` になっているのです。

一般に、オブジェクトの構造を決めるコンストラクタを `explicit` にすると有効です。逆に、引数がオブジェクトの一部になるコンストラクタは、普通、`explicit` にすべきではありません。

たとえば、`string` と `Str` の `const char*` を引数に取るコンストラクタは、`explicit` ではありません。それらのコンストラクタは `const char*` の引数からオブジェクトを初期化しているのです。引数がオブジェクトの持つ値を決めるので、エクスプレッションや関数呼び出しで `const char*` からの自動の型変換を許すのは合理的でしょう。

一方、`vector` と `Vec` の `size_type` 型の引数を取るコンストラクタは `explicit` です。これらは、引数の値からどれだけの要素分のメモリを確保するか決めているのです。つまり、引数はオブジェクトの構造を決めるだけで、値は決めていないのです。

12.5 型変換演算子

§ 12.2 ではコンストラクタが変換を定義することを見ました。クラスの制作者は明示的に型変換演算子（conversion operator）を定義することもできます。型変換演算子はクラスのメンバでなければなりません。また型変換演算子の関数の名前は、`operator` の後に変換後の型名を続けたものになります。たとえば、あるクラスが `operator double` というメンバを持っていたなら、このクラスオブジェクトから `double` 型の値を生成できるという意味になります。たとえば、

```
class Student_info {
public:
    operator double();
    // ...
};
```

とあれば、`Student_info` オブジェクトから `double` を生成するということです。この型変換が実際に何をするかは、その定義によって決まります。たとえば、オブジェクトを対応する最終成績に変換するものなどが考えられるでしょう。`double` が必要な場所に `Student_info` が使われている場合には、コンパイラがこの型変換演算子を呼び出すことになります。たとえば、`vs` を `vector<Student_info>` とすると、学生の成績の平均値を次のように計算することができます。

```
vector<Student_info> vs;
// vsにデータを入れる
double d = 0;
for (int i = 0; i != vs.size(); ++i)
    d += vs[i]; // vs[i]は自動でdoubleに型変換される
cout << "成績の平均：" << d / vs.size() << endl;
```

型変換演算子は、大抵、クラス型から組み込み型に変換するものが有用です。しかし、自分で書いたのではないクラス型への変換も有用になることがあります。どちらの場合も、変換先の型に新たにコンストラクタを付け加えることはできません。自分のクラスのメンバとして型変換演算子を定義するのです。

実際、`istream` の値をチェックするループでは、このような型変換演算子を使っているのです。§ 3.1.1 で議論したように、`istream` オブジェクトは条件に使えます。

```
if (cin >> x) { /* ... */ }
```

これは、

```
cin >> x;
if (cin) { /* ... */ }
```

と同じです。このエクスプレッションで何が行われるかもう理解できるのです。

前に説明したように `if` は条件を調べます。条件とは、真偽値を戻すエクスプレッションのことです。真偽値の正確な型は `bool` といいます。条件に数値やポインタを使うと、それらは自動的に `bool` 値に変換されるのです。そのため、これらの値も条件のエクスプレッションとして使えるのです。もちろん、`iostream` はポインタでも数値でもありません。しかし、標準ライブラリは `istream` から `void` 型へのポインタである `void*` への型変換を定義しているのです。これは `istream::operator void*` で定義されているのですが、`istream` のいろいろな状態フラグ（状態を表す変数）をチェックして `istream` が有効かどうかを調べ、0 か（ストリームの状態を表す）コンパイラが定義した 0 でない `void*` 値を戻すのです。

`void` という型は始めて出てきました。`void` という特別な型は、§ 6.2.2 で紹介しました。今はそのポインタを扱っているわけです。`void` へのポインタは、しばしばユニバーサルポインタとよばれ、どのような型でも指せるポインタなのです。もちろん、これでは具体的な型がわからないので、ポインタの指すオブジェクトの値をそのまま取り出すことはできません。しかし、`void*` は `bool` 値に変換できるのです。そして、今はこの性質を正在しているのです。

`istream` が `operator bool` ではなく `operator void*` を持っている理由は、コンパイラに次のような間違いを検出させるためです。

```
int x;
cin << x; // cin >> x;と書くべきです
```

もし `istream` で `operator bool` を定義していると、上のエクスプレッションで、`istream::operator bool` が使われ、`cin` が `bool` に変換され、それが続いて `int` に変換され、`x` の値だけビットシフトされ、最後にその結果が捨てられてしまうのです。しかし実際には数値ではなく、`void*` への変換をしているおかげで、`istream` を条件として使うことができ、しかも数値として誤った使われ方をされないのです。

12.6 型変換とメモリ管理

多くの C++ プログラムは C やアセンブリ言語で書かれたシステムとインターフェースを持って動作します。そのような言語では文字列データを保持するのにヌル文字を最後に付けた文字の配列を使います。§ 10.5.2 で見たように、入力・出力ファイルの名前を得るときには、C++ の標準ライブラリもこの方式を使っています。そのため、`Str` クラスも `Str` オブジェクトからヌル文字が最後にある文字配列への変換が必要だと考えるかもしれません。そうすると、ユーザは（自動的に）`Str` オブジェクトをヌル文字が最後にある配列で動作する関数に渡すこともできるようになります。しかし、残念ながら、そのようなことをするとメモリ管理の落とし穴に落ちてしまうのです。

`Str` から `char*` への型変換が欲しかったとしましょう。すると、おそらく `const` 版と非 `const` 版の 2 つが欲しくなるでしょう。

```
class Str {
public:
    // もっともらしいが、問題のある型変換演算子
    operator char*(); // 付け加えた
    operator const char*() const; // 付け加えた
    // 前と同様
private:
    Vec<char> data;
};
```

`Str` をこのように書き換えると、ユーザは次のようなコードが書けるようになります。

```
Str s;
// ...
ifstream in(s); // 希望:sを変換しsが保持する名前のストリームを開く
```

しかし、このような方法を実装することはほとんど不可能なのです。まず、この型変換で `data` をそのまま戻すことはできません。その理由は、`data` の型が `Vec<char>` ので、型が正しくないのです。ここで必要なのは `char` の配列です。さらに微妙ですが、もし型が合っていたとしても、`data` を戻せば `Str` クラスのカプセル化を壊してしまうのです。`data` のポインタを得たユーザが、`string` の値を変えてしまうこともできるのです。さらに悪いことに、`Str` オブジェクトが破棄されたときのことを考えてください。`Str` オブジェクトが破棄された後でも、ユーザがそのポインタを使おうとするかもしれません。このとき、ポインタの指す場所はシステムに戻されたメモリ領域であり、もはや有効ではないのです。

型変換演算子を `const char*` だけにすることでカプセル化の問題は解決しますが、それでもユーザが `Str` を壊し、そのポインタを使ってしまう可能性が残ります。この問題を解決する方法として、新しくメモリを確保しそこに `data` の内容をコピー、そのアドレスを戻すというものがあるかもしれません。しかし、この場合は、ユーザがそのメモリを管理し、不要になったら解放しなければなりません。

しかし、考えてみると、このデザインではうまくいきません。型変換は明示的でなくとも起こりうるわけです。そのような場合には、そのポインタがどこにも保持されないので使えず、したがってユーザはオブジェクトが破棄できないのです。たとえば、以下をみてください。

```
Str s;
ifstream is(s); // 非明示的な型変換。どうやってメモリを解放しましょう。
```

もし Str が上に書いたような型変換を持っていれば、そのオブジェクト s を ifstream のコンストラクタに渡せます。これにより、Str は非明示的にコンストラクタが予測する型である const char* に変換されます。この変換が、新しいメモリを確保してそこに s の内容をコピーし、そのアドレスを戻すのなら、このメモリを指すポインタは明示的には存在せず、そのメモリを解放することができなくなるのです。あきらかに、メモリリークを起こすデザインは正しいものではありません。

クラスをデザインするときには、無害に見えて実はユーザに問題を引き起こすようなコードは避けるべきです。C++ の標準が文字列クラスを定義する前に、たくさんの企業がいろいろな文字列クラスを提案しました。そのうちいくつかは、文字配列への非明示的な型変換を持っていました。しかし、これはユーザに先に指摘した間違いを起こさせるものです。

標準の string は別の方針を取りました。ユーザに string の内容をコピーした文字配列を渡すのですが、明示的にのみ渡すようにしたのです。標準の string クラスは、string オブジェクトから char の配列を得る 3 つのメンバ関数を持っています。その最初のものが c_str() で、これは string の内容をヌル文字が最後にある char の配列にコピーするものです。string クラスがこの配列を持ち、ユーザはそのポインタを delete しないと考えます。この配列のデータは短命で、他のメンバ関数が string の内容を変えるまでしか有効ではありません。ユーザはその配列へのポインタをすぐに使うか、自分で管理できる場所にコピーするかしなければならないのです。次に、data() という関数があります。これは c_str() と同様ですが、最後にヌル文字の付いていない文字配列へのポインタを戻します。最後は copy 関数です。これは、char* と整数を引数に取り、その整数分だけ char* で示された場所に文字をコピーするのです。この場合、char* で示される場所のメモリは、ユーザが自分で確保・解放しなければなりません。Str にこのような関数を付け加えるのは、みなさんの課題に残しておきます。

c_str と data には、非明示的な const char*への型変換と同じ落とし穴があることに注意してください。これらの型変換では明示的に関数を使うしかないので、ユーザも自分の使っている関数に注意できるでしょう。この場合、よく言われてきたことですが、ポインタをコピーすることの危険性についても知っておかなければなりません。標準ライブラリが非明示的な型変換を許したなら、より簡単にユーザが問題を起すだろうと思います。そのとき、彼らは型変換をしていることすら気づかず、何が悪いのかを理解できない可能性が高いのです。

12.7 詳細

型変換: explicit でない引数を 1 つだけとするコンストラクタ、または、operator type-name() という形の型変換演算子で定義される。ここで type-name はクラスが変換される型の名前である。型変換演算子はメンバでなければならない。2 つのクラスがお互いへの型変換を定義すると結果は不明になる。

12.7 詳細

friend: この宣言はクラス定義のどこにあってもよい。friend と宣言されたものは、private のメンバにも自由にアクセスできるようになる。

```
template<class T>
class Thing {
    friend std::istream& operator>>(std::istream&, Thing&);
    // ...
};
```

§ 13.4.2 で説明するように、クラスも friend ができる。

メンバであるテンプレート関数: クラスはテンプレート関数をメンバ関数にできる。その場合、クラスそのものはテンプレートであってもよい。テンプレートメンバ関数を持つクラスは、同じ名前の関数群を効率的に持つことになる。テンプレートメンバ関数の宣言・定義は、他のテンプレート関数と同じ。

string の関数:

| | |
|--------------|---|
| s.c_str() | ヌル文字が最後についた文字配列を指す const char*を戻す。配列内のデータは、次の string の関数が s を変更するまで有効。ユーザは delete を実行してはいけない。また、そのポインタの指すオブジェクトは短命なのでポインタも保持しておかないこと。 |
| s.data() | s.c_str() に似ているが、戻す文字配列の最後にヌル文字はない。 |
| s.copy(p, n) | s の保持する文字を、p が指すメモリ上へ n 文字コピーする。ユーザは、自分で、p が n 文字を保持できるメモリ領域を指すようにしなければならない。 |

課題

- 12-0 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。
- 12-1 メモリを自分で管理するように Str の定義を直してください。たとえば、文字の配列とそのサイズをデータメンバにしてみてください。このようなデザインの変更で、コピー管理はどうなるでしょうか。また、Vec を使うときのコスト（たとえば、メモリ管理のオーバーヘッドなど）も考えてください。
- 12-2 c_str、data、copy 関数を書いてください。
- 12-3 Str に比較演算子を定義してください。そのとき <cstring> にある strcmp 関数が役に立つでしょう。これは 2 つの文字ポインタを引数に取る関数で、前のポインタが指すヌル文字で終わる文字配列が、後のポインタが指すものより小さければ負の値、同じなら 0、大きければ正の値を戻す関数です。
- 12-4 Str に == 演算子と != 演算子を定義してください。
- 12-5 Str に const char* からの型変換を使わない文字列結合演算子を定義してください。
- 12-6 Str が非明示的に条件に使えるように、型変換演算子を定義してください。これは Str が空のとき偽となりそうでないときは真となるものにします。
- 12-7 標準の string クラスには、オブジェクトの文字を操作するために、ランダムアクセス反復子があります。反復子を定義し、先頭と末尾を指す反復子を戻す関数を Str に定義してください。
- 12-8 Str クラスに getline 関数を付け加えてください。