

## 第4章

### プログラムとデータの構成

§ 3.2.2 のプログラムは少々大きいと感じるかもしれません、`vector`、`string`、`sort` というものがなければもっと大きいものになっていたでしょう。このようなライブラリのツールは、他のものと同様に、いくつかの性質を持っています。それは、

- 特定の問題を解決するためにある
- 他の大部分のものと独立している
- 名前がある

です。私たちのプログラムも最初の 1 つは同じですが、後の 2 つは持っていません。小さなプログラムではそれでかまいませんが、大きな問題を解くときには、プログラムを独立した名前をもつ部分に分解していかなければ、とても管理できなくなります。

他のプログラミング言語と同様に、C++は大きなプログラムを構成するのに 2 つの基本的な仕組みを用意しています。それは関数（言語によってはサブルーチンとよばれます）とデータ構造です。さらに、C++では関数とデータ構造をまとめてクラスというものにすることができます。これは第 9 章以降に見ていくことにします。

プログラムの構成のために、関数とデータ構造の使い方を学べば、プログラムを複数のファイルに分割し、それぞれを別々にコンパイルし、その後でつなげることもできます。この章の最後では、C++における分割コンパイルの方法を紹介します。

#### 4.1 計算方法を構成する

学生の中間試験と期末試験と宿題の総合点数から最終成績を計算する関数を書いてみましょう。今は、宿題の総合点数が、個々の宿題の点数から平均なりメジアンなりで計算されているとします。また、この仮定とは別に、宿題と期末試験は最終成績の 40% になり、中間試験は 20% になるという方針は今まで通り維持することにします。

同じ計算を数箇所でする場合、あるいはそう予想される場合、それらをまとめて関数にすることを検討すべきです。そうすれば、関数を使うことで計算式を何度も繰り返す必要がなくなるからです。しかし、関数のメリットはこのようにプログラムを簡単にしてくれるだけではありません。関数を使うことで、後で計算を変えることも簡単になるのです。たとえば、最終成績の計算方針を変えたくなったとしましょう。成績処理の部分をプログラムの隅から隅まで探しわって、そこを書き直さなければならないとしたら、すぐに嫌になってしまふでしょう。

さらに、関数を使うことで、微妙なメリットを得ることもあります。それは関数には名前があるということです。「計算」に名前を付ければ、それをより抽象的に考えることができるのであります。つまり、その「計算」が何をするかをよく考えることができ、それがどのように仕事をするかは考えないようになるのです。問題の重要な部分を特定でき、それらに対応するプログラムの部分に名前を付けることができれば、そのプログラムを理解し問題を解決することは容易になるのです。

次に、私たちの方針にしたがった成績計算関数を示します。

```
// midterm (中間試験) と final (期末試験) と homework (宿題) から最終成績を計算する
double grade(double midterm, double final, double homework)
{
    return 0.2 * midterm + 0.4 * final + 0.4 * homework;
}
```

ここまででは、自分で定義した関数は mainだけでした。このように他の関数もほとんど同じ形で定義することができます。まず、戻り値の型を指定し、関数の名前を書き、その後の丸カッコ()の中にパラメータリスト(仮引数、parameter list)を書き、最後に中カッコ{}で囲まれた関数の定義を書くといふものです。他の関数を表すものを戻り値にする関数のように、もっと複雑なものもありますが、それらは§ A.1.2にまとめてあります。

今の場合、パラメータは midterm、final、homeworkで、これらはみな double型です。これらは関数内のローカルな変数のように振る舞います。つまり、関数の呼び出し時に生成され、関数が終了するときに破棄されるのです。

他の変数と同様に、パラメータは使う前に定義しないといけません。しかし、他の変数と違って、関数を定義したときにそのような変数が生成されるのではなくて、関数を使うときに始めて生成されるのです。そのため、関数を使うときには、かならず対応する引数(argument)を与える必要があります。<sup>\*1</sup>これは関数が実行されるときにパラメータを初期化するために使われます。たとえば、§ 3.1で最終成績を次のように計算しました。

```
cout << "あなたの最終成績：" << setprecision(3)
      << 0.2 * midterm + 0.4 * final + 0.4 * sum / count
      << setprecision(prec) << endl;
```

もし、grade関数を使うなら、これは次のように書き換えられます。

```
cout << "あなたの最終成績：" << setprecision(3)
      << grade(midterm, final, sum / count)
      << setprecision(prec) << endl;
```

引数は、関数の呼び出し時にただ与えるだけでなく、定義したときのパラメータと同じ順番で与えなければなりません。つまり、grade関数の呼び出しで、最初の引数は中間試験の点数、2番目の引数は期末試験の点数、最後の引数は宿題の総合点数でなければならないのです。

また、引数は sum / countのように、単なる変数でなくエクスプレッションでもかまいません。一般に、それぞれの引数が対応するパラメータを初期化し、これらのパラメータは関数の中でローカル変数のように振る舞うわけです。たとえば、grade(midterm, final, sum / count)の場合、gradeの関数のパラメータが引数

<sup>\*1</sup> 訳注：パラメータを仮引数とよんだり、もっと略して単に引数とよぶ本もあります。もちろん、仮引数と引数は厳密には区別すべきものです。本文中にあるように、引数とは、関数を使うときに関数に与えるものです。

値をコピーして初期化されて使われます。この際、もとの引数そのものを直接操作することはありません。このような仕組みは、パラメータが引数の値のコピーを使うので、しばしば値渡し(call by value)とよばれます。

#### 4.1.1 メジアンを探す

§ 3.2.2で考えたもう1つの問題で、他の状況でも必要になりそうなものは、vectorのメジアンを見つけるといふものです。§ 8.1.1では、vectorの保持するオブジェクトの型がどんなものでもそのメジアンを見つけられる関数を作りますが、ここではvector<double>に限定して考えましょう。

関数の定義は、まず、§ 3.2.2でメジアンを計算したプログラムの一部を少し変形するところからはじめましょう。

```
// vector<double>のメジアンを計算する
// この関数の呼び出しはvector全体をコピーすることに注意してください
double median(vector<double> vec)
{
    typedef vector<double>::size_type vec_sz;
    vec_sz size = vec.size();
    if (size == 0)
        throw domain_error("空のvectorでmedianを探した");
    sort(vec.begin(), vec.end());
    vec_sz mid = size/2;
    return size % 2 == 0 ? (vec[mid] + vec[mid-1]) / 2 : vec[mid];
}
```

変更の1つは、vectorオブジェクトの名前を homework(宿題)から vecに変えたことです。この関数は宿題に限らずどんなものでもメジアンを計算するのですから。また、medianという変数も、ここでは使わないのではなくしました。メジアンは計算したらすぐに戻り(return)してしまうからです。sizeとmidという変数は使いますが、これらはこの関数のローカル変数で、他の場所からアクセスすることはできません(無効です)。median関数を呼び出すとこれらの変数が内部で作られ、関数が終了するとこれらは破棄されるのです。vec\_szもローカルに定義しました。というのは、この名前を別の場所で別の目的に使いたい場合に、名前が衝突しないようにしたいからです。

もっとも大きな変更は、vectorが空のときの処理です。§ 3.2.2では、プログラムを起動しているものに対して「失敗の報告」を出しました。その場合、プログラムを使う人が誰か、どのような報告が有効かがわかつっていました。しかし、今の場合、この関数のユーザが誰なのか、その目的が何なのかわかりません。したがって、失敗の報告はもっと一般的にしなければならないのです。もっとも一般的な方法は、vectorオブジェクトが空の場合、例外を投げる(スローする、throw an exception)といふものです。

プログラムで例外が投げられると、その実行は throwの場所で止まり、関数の呼び出し側が使える情報を持った例外オブジェクト(exception object)をプログラムの別の部分に渡すことになるのです。

渡される情報の中でもっと重要なのは、大抵、例外的な事態が起こったという事実そのものです。つまり、この例外オブジェクトの型と共に、普通は、例外が起こったという事実だけで、関数の呼び出し側は次に何をするべきかわかるのです。今の例で、投げる例外は domain\_errorといふものです。これは標準ライブラリが<stdexcept>ヘッダに定義しているもので、関数の引数が許される範囲外にあることを知らせるためのものです。この domain\_errorを作るときは、問題を示す文字列(stringオブジェクト)を与えることができます。

プログラムは、この例外を受け取り、その文字列を診断メッセージに使うこともできます。これは§4.2.3で紹介します。

この関数の振る舞いについては、もう1つ理解すべき重要な点があります。この関数を呼ぶと、パラメータは引数によって初期化されるローカル変数と考えることができます。これは、引数がパラメータにコピーされるということです。特に、median関数を呼ぶ場合、引数のvectorオブジェクトはvecにコピーされるわけです。

median関数の場合、引数のコピーに時間がかかったとしても、コピーは必要だったのです。というのは、median関数は内部でsortを使いパラメータの中身を書き換えるからです。引数をコピーすることで、関数の呼び出し側にsortの影響を及ぼさないで済むのです。これはそうあるべきことです。メジアンを見つけることによって、vectorオブジェクト自身が変更されるべきではないからです。

#### 4.1.2 成績評価方針をコードにする

§4.1のgrade関数は、学生の宿題の個々の点数ではなく、総合点数がわかっていると仮定して作りました。宿題の総合点数をどうつけるかは、成績評価方針の一部です。§3.1では平均を使い、§3.2.2ではメジアンを使いました。そのため、§4.1のときと同様に、この成績評価の方針を関数にしたいと考えることもあるでしょう。

```
// 中間試験、期末試験、宿題の点数から学生の最終成績を計算する
// medianは引数をコピーするのが、この関数は引数をコピーしない
double grade(double midterm, double final, const vector<double>& hw)
{
    if (hw.size() == 0)
        throw domain_error("この学生は宿題を1つもしていない");
    return grade(midterm, final, median(hw));
}
```

この関数には3つの面白いポイントがあります。

まず、1つ目は、3番目の引数の型、const vector<double>&です。この型はしばしば、「const vector<double>への参照（リファレンス）」とか「const vector<double>参照（リファレンス）」と呼ばれます。ある名前が参照またはリファレンス（reference）であるというのは、それがあるオブジェクトの別名だという意味です。たとえば、

```
vector<double> homework;
vector<double>& hw = homework; // hwはhomeworkの別名
```

とあれば、hwをhomeworkの別名として定義したということです。したがって、hwにすることは、みな同じことがhomeworkにされ、逆にhomeworkにすることはhwにされることになります。ここで次のようにconstを付けると、homeworkがchwの別名であることには変わりありませんが、chwの値を変えないという約束になるのです。

```
// chwは読み込み専用のhomeworkの別名
const vector<double>& chw = homework;
```

参照はhomeworkの別名なので、参照の参照はありません。参照の参照を定義すると、それはもとのオブジェクトの別名になります。たとえば、次のように書いたとします。

#### 4.1 計算方法を構成する

```
// hw1とchw1はhomeworkの別名。chw1は読み込み専用
vector<double>& hw1 = hw;
const vector<double>& chw1 = chw;
```

hw1はhwと同じくhomeworkの別名になります。そしてchw1もchwと同じくhomeworkの別名になります。ただし、chw1は書き込み用には使えません。

constでない参照を定義した場合、これは書き込みも可能になりますが、これをconstの参照に対応させることはできません。それはconstの書き込み不可という性質と矛盾してしまうからです。そのため、次のようなことはできないのです。

```
vector<double>& hw2 = chw; // chwを書き込み可能にしようとしているのでエラー
```

これはchwを変えないという約束に違反するからです。

同様に、パラメータがconst vector<double>&の型を持つということは、コピーせずに引数に直接アクセスすることと、このパラメータの値を決して変えないことを約束するということなのです。（パラメータの値を変えると当然引数の方も変わってしまうからです。）このパラメータはconstの参照なので、この関数の引数にはconstであるvectorオブジェクトでもconstでないvectorオブジェクトでも使えます。また、参照を使うことで引数のコピーに時間を取られることもあります。

gradeの2番目の面白い点は、§4.1の関数と同じ名前だということです。さらに、この関数はそのgrade関数を内部で呼んでいます。同じ名前の関数をいくつも定義することをオーバーロード（overloading）と言います。これは多くのC++プログラムで見られるものです。同じ名前の関数が2つあっても、gradeを呼び出すときに引数のリストを与えるので、コンパイラは引数からプログラマがどのgradeが呼びたいのかを判断できるのです。

最後のポイントは、median関数が同様のチェックをするのにもかかわらず、homework.size()が0かどうかをチェックしているということです。medianの中でvectorオブジェクトが空だとわかった場合、それは「空のvectorでmedianを探した」というメッセージを持った例外のスローになります。学生の成績を計算している人に、このメッセージは直接役立たないので、この場合独自の例外を考えたのです。こうすることで、何が悪いかをユーザが見つけやすくしたいのです。

#### 4.1.3 宿題の点数を読み込む

宿題のvectorへの読み込みも、他の場面で必要となる問題です。

このような関数をデザインするときに考えるべきことが1つあります。それは、2つの値を同時に戻したことです。当然1つは、読み込んだ宿題の点数です。もう1つは読み込みが成功したかどうかを示す報告です。

しかし、複数の値を戻す直接的な方法はありません。1つの間接的な方法は、パラメータをオブジェクトの参照にするということです。このオブジェクトの値を置き換えることで、結果を関数の外に出すわけです。この方法は読み込みの関数にはよく使われるものなので、私たちも使うことにしましょう。これにより関数は次のようになります。

```
// 入力ストリームからvector<double>に宿題の点数を読み込む
istream& read_hw(istream& in, vector<double>& hw) {
```

```
// ここに定義を書く
return in;
}
```

§ 4.1.2 では、`const vector<double>&`というパラメータを見ました。ここでは `const` を落としています。一般に、参照パラメータに `const` が付いていない場合は、引数のオブジェクトを変更するつもりであることを示しています。たとえば、次のようなコードがあったとします。

```
vector<double> homework;
read_hw(cin, homework);
```

この場合、`read_hw` の 2 番目のパラメータが参照なので、`read_hw` は `homework` の内容を変更していると考えられます。

この関数は引数を変えるので、引数にどんなエクスプレッションでも入れられるというわけではありません。参照パラメータには左辺値 (lvalue) というものを渡さなければならないからです。左辺値とは一時的なオブジェクトではないという意味です。変数や参照、参照を戻す関数の戻り値は左辺値です。一方、`sum / count` のように算術的な値になるものは左辺値ではありません。

`read_hw` の 2 つのパラメータは両方とも参照ですが、それはどちらの状態も関数によって変えられると考えてよいからです。まだ、`cin` の詳細は説明していませんが、これはライブラリが定義したデータ構造で、入力ファイルに関して必要なデータをすべて保持するものなのです。標準入力ファイルからの入力はそのファイルの状態を変えるので、`cin` の状態も変わると考えられるわけです。

ここで `read_hw` は `in` を戻していることに注意してください。さらにこれは参照です。これは、入力ストリームのオブジェクトのコピーではなくそのものを受け取り、そのコピーではなくそのものを戻しているということです。入力ストリームを戻しているので、次のようなことも可能になります。

```
if (read_hw(cin, homework)) { /* ... */ }
```

これは

```
read_hw(cin, homework);
if (cin) { /* ... */ }
```

の短縮形です。

これから宿題の点数の読み込み方を考えます。もちろん、宿題の数だけその点数を読み込みたいので、次のようなものを考えるかもしれません。

```
// 最初の試み：正しくない
double x;
while (in >> x)
    hw.push_back(x);
```

この方法は次の 2 つの理由で完全にはうまくいきません。

1 つ目の理由は、`hw` は関数の呼び出し側が定義したもので、関数が定義するものではないということです。自分で定義したものではないので、中にどんなデータが入っているかわからないのです。この関数の呼び出し側は、この関数を使って複数の学生の宿題の点数を処理しているかもしれません。その場合は、前の学生のデータ

が入っているとも考えられます。この問題は読み込みの前に、すべてのデータを破棄する `hw.clear()` を実行することで解決できます。

2 つ目の理由は、読み込みの中止をきちんとしておかないと失敗するということです。上のコードでも点数を読めるだけ読み込みますが、その後で問題が起こります。読み込みが失敗する理由は 2 つ考えられます。ファイルの終わりに到達したか、点数ではないものを読み込みそうになったかです。

最初のケースでは、関数の呼び出し側もそう考えるでしょう。これは正しいのですが、誤解を生みやすくもあります。ファイルの終わりはすべてのデータを正しく読み込んだ後にあるのですが、これ自身は通常は読み込みの失敗と解釈されるからです。

もう 1 つのケースでは、点数でないものにぶつかった場合ですが、この場合、ライブラリはストリームを失敗状態 (failure state) という状態にします。この状態になるとファイルの終わりに到達したときと同様に、次の読み込みはできなくなります。そのため、関数の呼び出し側は、点数の読み込み中に点数以外のものを読み込みそうになると、入力で何か問題が起ったことがわかるのです。

どちらの場合も、それ以後の点数を読み込まなければよいだけです。このような処理は簡単です。ファイルの終わりに達していれば実際に読み込むものは無いし、途中で点数で無いものにぶつかったなら、ライブラリがそれを読み込みまいかからです。結局、ファイルの終わりであっても点数以外のものであっても、読み込みを中止 (失敗) させるものに出会ったなら、そこで読み込みを中止し、ストリームの状態をもとに戻すだけです。これは `in.clear()` で行えます。これによりライブラリに失敗状態からもとに戻すように指示できます。<sup>2</sup>

もう 1 つ詳細に考えるべきことがあります。それは入力をはじめる前にすでに入力ストリームが、ファイルの終わりや間違ったデータなどで失敗状態になっている可能性があるということです。そのような場合、ストリームには手を加えずに戻すべきです。なぜなら、ストリームの状態を変えてしまうと、関数の呼び出し側を惑わし、存在しないデータを読み込むうとさせるかもしれないからです。

以下に `read_hw` の完全版を書きます。

```
// 入力ストリームからvector<double>に宿題の点数を読み込む
istream& read_hw(istream& in, vector<double>& hw)
{
    if (in) {
        // 以前のデータを破棄する
        hw.clear();
        // 宿題の点数を読み込む
        double x;
        while (in >> x)
            hw.push_back(x);
        // 次の読み込みのためストリームをリセットする
        in.clear();
    }
    return in;
}
```

ここで同じ `clear` が `istream` と `vector` に対し、まったく違う働きをしていることに注意してください。`istream` に対しては、失敗状態をリセットし読み込みを続けるために使いました。`vector` に対しては、`vector` オブジェクト内に残っているかもしれない全データを破棄し、オブジェクトを空にするために使いました。

<sup>2</sup> 訳注：これは次の読み込みのためにすることです。

#### 4.1.4 3種類の関数パラメータ

ここで少し立ち止まって、いくつか重要なポイントを見てみましょう。ここまで宿題の点数を保持する `vector` オブジェクトを扱う 3 つの関数、`median`、`grade`、`read_hw` を定義しました。これらの関数のパラメータはみな理由があつて違うものでした。

`median` 関数（§ 4.1.1）のパラメータの型は `vector<double>` でした。したがって、この関数が呼び出されるところのオブジェクトがたとえ巨大な `vector` であっても引数はコピーされるのです。これは非効率的に見えても `median` には正しい選択なのです。`median` がもとの `vector` を変えないことを保証するからです。`median` 関数の中では `sort` を使ってパラメータをソートしています。もし、これが引数のコピーでなければ `median` を呼び出した後には引数にしたオブジェクトが変わってしまう（整列させられる）のです。

`grade` 関数（§ 4.1.2）のパラメータの型は `const vector<double>&` です。ここで `&` は、パラメータを引数のコピーにするのではなくそのものにするようにコンパイラに指示しており、また `const` は引数のオブジェクトを変更しないと約束しています。このテクニックはプログラムを効率的にするときに重要です。特に、パラメータが `vector` や `string` のようにコピーに時間がかかるもので、しかもその値を変えることが無い場合には有効です。しかし、`int` や `double` のような組み込み型に `const` 参照を使っても普通は効率的にはなりません。そのような小さなオブジェクトのコピーは大抵とても速く、余計な時間はほとんどかかりません。

`read_hw` 関数のパラメータの型は `const` なしの `vector<double>&` です。`grade` のときと同様に、`&` はコピーを避け、パラメータを引数のものにするという意味です。この場合にコピーを避ける理由は、引数の値を意図的に変更するためでした。

`const` でない参照パラメータは左辺値でなければなりません。つまり、それは一時的なオブジェクトではないということです。値で渡される引数や `const` の参照なら、引数に条件はありません。たとえば、空の `vector` を戻す次のような関数があったとします。

```
vector<double> emptyvec()
{
    vector<double> v; // 要素は無い
    return v;
}
```

この関数を呼んでその戻り値を § 4.1.2 で考えた 2 番目の `grade` 関数の引数に使うこともできるのです。

```
grade(midterm, final, emptyvec());
```

実行すると、`grade` 関数は引数の `vector` が空なのですぐに例外を投げるでしょう。しかし、上のコードは文法的には間違いではないのです。

しかし、`read_hw` のパラメータは `const` でない参照なので、引数は両方とも左辺値でないといけません。もし、左辺値でない `vector` を次のように渡すと、

```
read_hw(cin, emptyvec()); // エラー: emptyvec()は左辺値でない
```

コンパイラはエラーだと言っています。`emptyvec` 関数が戻す名前の無い `vector` は `read_hw` が終了するときに破棄されてしまうからです。もし、これを許してしまうと、アクセスできないオブジェクトに入力を格納することになるでしょう。

#### 4.1 計算方法を構成する

#### 4.1.5 学生の成績計算に関数を使う

これまで関数を書いてきたのは、与えられた問題を解決するためでした。たとえば、§ 3.2.2 の成績処理プログラムを次のように書き換えることもできます。

```
// ライブラリを使うためのincludeディレクティブ、using宣言
// § 4.1.1にあるmedian関数のコード
// § 4.1にあるgrade(double, double, double)関数のコード
// § 4.1.2にあるgrade(double, double, const vector<double>&)関数のコード
// § 4.1.3にあるread_hw(istream&, vector<double>&)関数のコード
int main()
{
    // 学生の名前を入力させる
    cout << "姓を入力してください: ";
    string name;
    cin >> name;
    cout << "こんにちは、" << name << "さん！" << endl;
    // 中間試験と期末試験の点数を入力させる
    cout << "中間試験と期末試験の点数を入力してください: ";
    double midterm, final;
    cin >> midterm >> final;
    // 宿題の点数を入力させる
    cout << "宿題の点数を全部に入力してください"
        " (ただし、最後はend-of-file) : ";
    vector<double> homework;
    // 宿題の点数を読み込む
    read_hw(cin, homework);
    // 可能なら、最終成績を計算し出力する
    try {
        double final_grade = grade(midterm, final, homework);
        streamszie prec = cout.precision();
        cout << "あなたの最終成績：" << setprecision(3)
            << final_grade << setprecision(prec) << endl;
    } catch (domain_error) {
        cout << endl << "あなたの宿題の点数が必要です。"
            "やりなおしてください。" << endl;
        return 1;
    }
    return 0;
}
```

前のプログラムとの違いは、宿題の点数の読み込みと最終成績の計算・出力の部分です。

ユーザに宿題の点数を入力するように求めた後、データを読み込むために `read_hw` 関数を呼んでいます。`read_hw` の中の `while` ループが、ファイルの終わりか `double` でないデータに達するまで宿題の点数を読み込みます。

この例で登場するもっとも重要なものは `try`ステートメントです。これはまず `try` の後の中カッコ{}の中のステートメントを実行しようとしています。そして、もし `domain_error`例外がその中のどこかで発生した場合、その実行を中止し次の中カッコ{}の中身が実行されるというものです。後者のステートメントは `catch`節 (`catch`)

clause) というもので、これは catch というキーワードと例外の型を示す部分に続きます。

もし try と catch の間のステートメントが完全に実行され例外が投げられなければ、catch 節はスキップされ、次のステートメントが実行されるように決められています。この例で次のステートメントとは return 0; です。

try ステートメントを書くときは、いつも、その副作用がいつどのような効果を持つかを慎重に考える必要があります。try と catch の間のものは何でも例外を投げるかもしれないと考えるべきなのです。そして、その場合は、例外が起こらなかった場合に実行される計算をみなスキップしなければなりません。また、例外後の処理は必ずしもすべてプログラム中で行われるものでもないことを理解してください。

たとえば、出力部分が次のような簡明なものであったとします。

```
// この例はうまくいきません
try {
    streamsize prec = cout.precision();
    cout << "あなたの最終成績：" << setprecision(3)
        << grade(midterm, final, homework) << setprecision(prec);
} ...
```

<<演算子は順番に実行されるのですが、オペランドの評価の順番は決まっていないことが問題です。たとえば、grade は「あなたの最終成績：」と出力した後に呼ばれるかもしれません。もし grade が例外を投げれば、メッセージが混ざって出力されるかもしれません。さらに、前の setprecision の呼び出しが出力ストリームの精度を 3 にしたのに、後の setprecision が実行されず、精度はリセットされないかもしれません。したがって、grade は出力の前に呼び出されなければならないのですが、いつ実行されるかはコンパイラ次第なのです。

ここで正しいコードでは、出力部分が 2 つの部分（ステートメント）に分けられている理由がわかると思います。出力の前に最初に grade を実行しているのです。

1 つのステートメントには 2 つ以上の副作用を持たせない、というのがよいルール（経験則）です。そして、例外を投げるということは副作用の一種です。したがって、例外を投げるかもしれないステートメントを他の副作用、とくに入出力の副作用を持つものと混ぜない方がよいのです。

もちろん、これだけでは main を実行できません。ライブラリの用意したものを使うためには、include ディレクトリと using 声言が必要です。また、read\_hw の定義と 3 番目のパラメータの型が const vector<double>& である grade の定義も必要です。しかし、これらの関数の定義には、さらに median と 3 番目のパラメータの型が double である grade の定義も必要になります。

つまり、このプログラムを実行するためには、これらの関数が（正しい順番で） main 関数の前に定義されなければならないのです。しかし、そうするとプログラムは扱いづらい長さになってしまいます。ここでそれを書くことは止めましょう。§ 4.3 でこのようなプログラムを簡潔にファイルにわける方法を紹介します。その前に、データの構造をよりよくする方法を考えます。

## 4.2 データの構成

学生 1 人の成績を計算するのはその学生には便利ですが、このくらいの計算なら電卓でもすんでしまいそうです。一方、私たちが教員なら、受け持ち学生全員の成績処理をしたいと思うでしょう。そこで、プログラムを教員用のものに書き換えて見ましょう。

### 4.2 データの構成

1 人の学生の成績をインタラクティブな操作の後に報告するのではなく、多くの学生の名前と点数が書いてあるファイルを持っていると仮定し、それを使うことを考えます。今、そのファイルには名前に続いて中間試験の点数と期末試験の点数と 1 つ以上の宿題の点数が書かれているとします。それはたとえば、次のようになっているでしょう。

```
Smith 93 91 47 90 92 73 100 87
Carpenter 75 90 87 92 93 60 0 98
...
```

私たちのプログラムは、このファイルからそれぞれの学生の最終成績を計算するものです。ここで、宿題の総合点数はメジアンとし、これと期末試験の点数がそれぞれ全体の 40%、中間試験が全体の 20% になるようにします。

出力は次のようになるでしょう。

Carpenter	86.8
Smith	90.4
...	

出力は学生名をアルファベット順にし、最終成績を見やすいように縦に並べることにします。アルファベット順に並べ替えるためには、全学生の成績をどこかに記録する必要があります。また、名前と最終成績の間にどれだけ空白を入れるかを決めるため、もっとも長い名前の長さも調べなければなりません。

ここで学生 1 人のデータを入れる変数を作れるとすれば、あとは vector で全学生のデータが保持できることに注意してください。そして、全学生のデータを vector に入れておけば、sort が使え、最終成績の計算も出力もできるはずです。まず、学生のデータを保持する入れ物を考え、次にこれらのデータを読み込み処理する補助的な関数を考えます。これらの抽象化の後に、これらを使って始めて述べたような動作をするプログラムを作ります。

#### 4.2.1 すべての学生のデータをまとめる

個々の学生のデータを読み込み、それを名前順に並べなおす必要があります。そのときに、学生の名前と成績がばらばらにならないようにしなければなりません。そのため、学生 1 人の全情報を一箇所にまとめて保持するものが必要です。それは学生の名前、中間試験の点数、期末試験の点数、すべての宿題の点数を保持できるものです。

C++ では、そのようなデータ構造を次のように定義します。

```
struct Student_info {
    string name;
    double midterm, final;
    vector<double> homework;
}; // セミコロンがあることに注意
```

ここで struct は、4 つのデータメンバを持つ Student\_info という型を定義しています。この Student\_info は型（このように定義した型を構造体という）なので、これを定義した以後は、この型のオブジェクトを定義できます。そしてそれぞれのオブジェクトが 4 つのデータメンバを持つことになるのです。

最初のメンバは `name` で、その型は `string` です。2番目のメンバと3番目のメンバは `midterm` と `final` で、その型は `double` です。そして最後のメンバは `double` の `vector` である `homework` です。`Student_info` オブジェクト1つひとつが学生1人ひとりの情報を表すわけです。そして、`Student_info` は型なので、複数の学生の情報を保持するためには `vector<Student_info>` が使えるのです。これは複数の宿題の点数を保持するのに `vector<double>` を使ったのと同じです。

#### 4.2.2 学生の成績データを管理する

プログラムを分解して管理できるように考えると、結局3つのステップに分解でき、それぞれを独立した関数で表すことができます。まず、`Student_info` にデータを読み込むステップが必要です。それからその `Student_info` のデータに対応する最終成績を計算するステップが必要で、最後に `Student_info` の `vector` オブジェクトをソートするステップが必要です。

まず、学生1人分のデータを読み込む関数は§4.1.3で書いた `read_hw` によく似ています。実際、宿題の点数を読み込むのに、この関数を内部で使えます。さらに学生の名前と試験の点数を読み込む部分もつけなければなりません。

```
istream& read(istream& is, Student_info& s)
{
    // 学生の名前、中間試験の成績、期末試験の成績を読み込む
    is >> s.name >> s.midterm >> s.final;
    read_hw(is, s.homework); // 宿題の全成績を読み込む
    return is;
}
```

この関数の2番目のパラメータの型が何を読み込むか示しているので、単に `read` という名前にもその内容があいまいということにはなりません。オーバーロードの機能のおかげで、他に `read` という名前の関数があっても、それは別のデータを読み込むものとして、この `read` とは区別されるはずなのです。この関数は `read_hw` と同じように、読み込み先を表す `istream` と、読み込んだものを格納するオブジェクトへの参照がパラメータになっています。もちろん、パラメータ `s` を関数内部で使っているということは、引数として渡される `Student_info` オブジェクトのデータを変えるということです。

この関数は、まずオブジェクト `s` のメンバである `name`、`midterm`、`final` にデータを読み込んでいます。それから `read_hw` を使って宿題の点数を読み込んでいます。読み込みの過程で、ファイルの終わりや不適当なデータに到達することがあり得るわけですが、その後は読み込みは行われず、`is` はエラー状態のまま関数から戻されることになります。これが可能なのは、`read_hw` 関数 (§4.1.3) は、よく考えて、エラー状態の入力ストリームを引数に受け取った場合、何もせずに戻すようにしたからです。

もう1つ書かなければならない関数は、`Student_info` オブジェクトから最終成績を計算する関数です。これは§4.1.2の `grade` ではほぼ解決されています。ここではこの関数をオーバーロードし、`Student_info` オブジェクトに対して最終成績を計算するような関数を定義します。

```
double grade(const Student_info& s)
{
    return grade(s.midterm, s.final, s.homework);
}
```

#### 4.2 データの構成

この関数は `Student_info` オブジェクトを引数に取り、`double` である最終成績を戻します。パラメータの型が `Student_info` ではなく `const Student_info&` なのは、この関数を呼び出すときに引数の `Student_info` オブジェクトのコピーで余分な時間を使わないためです。

この関数では内部で呼び出している `grade` の例外処理を乗っ取るようなことはしていません。なぜなら、今の場合、内部で呼び出している `grade` の例外処理以上のこととはできないからです。今定義した `grade` は例外を `catch` しないので、投げられた例外はこの関数の呼び出し側に届くはずです。この関数の呼び出し側は、宿題をしていない学生をどう扱うべきか知っているはずでしょう。

プログラム全体を書く前に残った仕事は、`Student_info` をどのようにソートするか決めることです。`median` 関数 (§4.1.1) では、`vec` という名前の `vector<double>` パラメータを `sort` という関数でソートしました。

```
sort(vec.begin(), vec.end());
```

しかし、`vector<Student_info>` のオブジェクトを `students` とした場合、単純に以下のようにソートすることはできません。

```
sort(students.begin(), students.end()); // 完全には正しくない
```

どうしてでしょうか。`sort` や他のライブラリのアルゴリズムについて、詳しくは第6章で説明しますが、ここで少しだけ、`sort` がどのように働くか考えてみるのもよいでしょう。特に、`sort` がどのように `vector` 内のデータに順番を付けるのでしょうか。

この `sort` 関数が `vector` の中のデータを並べ替えるのに、データ同士を比べる必要があります。この比較は、ソートをしたい `vector` 内のデータの型に対して、<演算子を使うことで行われます。たとえば、`vector<double>` をソートできるのは、`double` のデータ同士には<が使え、正しい結果が得られるからです。それでは、`sort` が2つの `Student_info` オブジェクトを比較しようとした場合、何が起こるでしょうか。<演算子は、`Student_info` オブジェクトには意味を持たないので、実際、`sort` を使いこれらのオブジェクトをくで比較しようとすると、コンパイラがエラーメッセージを出します。

幸い、`sort` 関数は3番目の引数に判定関数 (述語関数、predicate) というものを持たせることもできます。この判定関数というものは、真偽の値、特に `bool` 値を戻す関数のことです。この3番目の引数が与えられると、`sort` 関数はくの代わりに、この関数を要素の比較に使うことになります。したがって、2つの `Student_info` オブジェクトを引数に取り、前のオブジェクトが後のオブジェクトより小さいかどうかを真偽値で戻す関数を定義すればよいのです。今は、学生を名前のアルファベット順にソートしたいので、比較の関数は `name` のみを比較することにします。

```
bool compare(const Student_info& x, const Student_info& y)
{
    return x.name < y.name;
}
```

この関数は単純に `Student_info` 同士の比較を `string` 同士の比較にしているだけです。これは `string` に比較の演算子<が定義されているので可能なのです。`string` オブジェクト同士の<による比較は普通の辞書式の順序に従います。つまり、アルファベット順で左のオペランドが右のオペランドよりも前にあるなら、左のオペランドのほうが右のオペランドよりも小さいと考えられるのです。これはまさに望ましいものです。

`compare` 関数を定義したので、ライブラリの `sort` 関数の 3 番目の引数に `compare` を渡すことで `vector` のソートができます。

```
sort(students.begin(), students.end(), compare);
```

こうすると、`sort` が要素を比較する場合、`compare` 関数が呼ばれて比較が行われるのです。

### 4.2.3 レポートの作成

学生の成績処理に必要な関数をすべて書いたので、レポートを作成することができます。

```
int main()
{
    vector<Student_info> students;
    Student_info record;
    string::size_type maxlen = 0;
    // すべてのデータを読み込み、最長の名前の長さを求める
    while (read(cin, record)) {
        maxlen = max(maxlen, record.name.size());
        students.push_back(record);
    }
    // アルファベット順に並べる
    sort(students.begin(), students.end(), compare);
    for (vector<Student_info>::size_type i = 0;
         i != students.size(); ++i) {
        // 名前を出し、全体が maxlen + 1 文字になるように名前の右側に空白を入れる
        cout << students[i].name
            << string(maxlen + 1 - students[i].name.size(), ' ');
        // 最終成績を計算し出力する
        try {
            double final_grade = grade(students[i]);
            streamsize prec = cout.precision();
            cout << setprecision(3) << final_grade
                << setprecision(prec);
        } catch (domain_error e) {
            cout << e.what();
        }
        cout << endl;
    }
    return 0;
}
```

このコードの大部分はすでに見ていますが、いくつか新しいこともあります。

まず、最初にライブラリの関数で `<algorithm>` ヘッダに定義されている `max` という関数を使っています。一見して、`max` の振る舞いは明らかでしょう。しかし、1 つだけ明らかでないことがあります。それは § 8.1.3 で説明する複雑な理由によって、引数は正確に同じ型のものでないといけないということです。この事情により、`maxlen` という変数の型を単純に `int` にすることはできず、`string::size_type` としたのです。  
また、

```
string(maxlen + 1 - students[i].name.size(), ' ')
```

### 4.3 みんなまとめて

が新しいものです。これは § 1.1 で説明したような名前なしの `string` オブジェクトで、半角空白文字が `maxlen + 1 - students[i].name.size()` 個並んだ文字列です。これは § 1.2 節で空白の表示を使ったものと似ていますが、こちらはオブジェクトに名前を付けていないところが違います。名前を付けないので、定義をエクスプレッションとして扱えるのです。

このエクスプレッションを出力することで、名前の横に正しい数の空白を入れることができるわけです。

それから、`students` の要素の 1 つひとつを表すのにインデックス `i` を使っています。`students` にインデックスを付けることで `Student_info` のオブジェクトを取り出せるのでした。このオブジェクトの `name` と空白文字列を組み合わせて出力しているのです。

この後では、学生 1 人ひとりの最終成績を出力しています。もし、学生がひとつも宿題をやっていないなら、`grade` の計算は例外を投げます。今のは、この例外を受け取り（キャッチし）、数値の成績を出力するかわりに例外オブジェクトの持つメッセージを出力するようにしました。`domain_error` のような標準ライブラリの例外は、オプションで引数を持つのですが、この引数が例外を引き起こした原因を記述しているのです。この引数は例外のメンバ関数である `what` によってコピーを作り利用することができるのです。今の場合、メッセージは「この学生は宿題を 1 つもしていない」となります。例外が起らぬ場合、`setprecision` という操作子を使って有効数字を 3 術にし、`grade` の結果を出力しています。

### 4.3 みんなまとめて

ここまでにいくつかの抽象化（関数とデータ構造）を行いました。これらは問題の解決に役立つものです。そして、これら抽象化の定義をすべて 1 つのファイルに書き込み、それをコンパイルする方法をとってきました。明らかに、この方法はプログラムが大きくなるとすぐに複雑になってしまいます。このような複雑さを軽減するために、C++ のような言語には分割コンパイル（separate compilation）の機能があります。これはプログラムを複数のファイルに分け、それぞれを別々にコンパイルする方法です。

まず始めに `median` 関数を独立させ、他のプログラムでも使えるようにする方法を示します。それには `median` 関数の定義を 1 つのファイルに書き込み、それを独立したコンパイル可能なファイルにするのです。そのためには、`median` 関数が使うすべての名前の宣言をファイルに入れなければなりません。`median` はライブラリの `vector` 型、`sort` 関数、`domain_error` 例外を使うので、これらのための適当なヘッダをインクルードする必要があります。

```
// median関数のソースファイル
#include <algorithm> // sortの宣言がある
#include <stdexcept> // domain_errorの宣言がある
#include <vector> // vectorの宣言がある
using std::domain_error; using std::sort; using std::vector;
// vector<double>オブジェクトのメジアンを計算する
double median(vector<double> vec)
{
    // § 4.1.1 にあるこの関数の定義
}
```

どのファイルでもそうですが、このファイルにも名前を付けなければなりません。C++ 標準規格にはソースファイルの名前の付け方についての規定はありません。しかし、一般に、ソースファイルの名前は中身がわかるもの

がよいでしょう。さらに、大抵のコンパイラでは、ファイル名の最後に特定の文字列（拡張子）を付けるようにしています。コンパイラはこの拡張子でソースファイルを判断するのです。大抵の場合、C++のソースファイルの拡張子は.cpp、.C、.cなどですから、median関数のソースファイルはmedian.cpp、median.C、median.cとなります。これらはコンパイラによって異なることに注意してください。

次のステップは、このmedian関数を他のユーザにも使えるようにすることです。標準ライブラリではヘッダの中に名前の定義が書いていましたが、同様に、自分の定義した名前（関数などの名前）を使うためのヘッダファイル（header file）を定義することができます。たとえば、median.hファイルを作れば、他のプログラムでもmedian関数を使えるようにできるのです。そうしておけば、ユーザが次のような形でmedian関数を使えます。

```
// medianを使うためのとてもよい方法
#include "median.h"
#include <vector>
int main() { /* ... */ }
```

#include ディレクティブで角カッコ<>でなくダブルクオーテーション""を使った場合は、コンパイラに、#include ディレクティブのあるその場所に、その名前のヘッダファイル全体をコピーするように要求することになるのです。これによりプログラムの中にヘッダファイルが書き込まれることになります。そのヘッダファイルの置き場所や""内の表記と実際のファイル名との関係はコンパイラによって違います。したがって、「median.h」というヘッダファイル」というのは、正しくは「median.h」という名前に対応するヘッダファイルとコンパイラが判断するファイル」の省略形だと思ってください。

ここで私たちの作るヘッダはヘッダファイルとよぶのに対し、コンパイラが提供する標準はヘッダであってヘッダファイルとはよばないことに注意してください。というのは、私たちが作るヘッダはどのC++コンパイラに対しても実際のファイルですが、標準ヘッダは特別なものでファイルとは限らないからです。#include ディレクティブはヘッダファイルにも標準ヘッダにも使えますが、それらが同じように実装されているとは限らないのです。

さて、ヘッダファイルを作らなければなりません。それはどのようなものであるべきでしょうか。簡単にいうと、それはmedian関数の宣言（declaration）でなければなりません。これは関数の定義部分をセミコロン；に置き換えるだけです。

```
double median(vector<double>);
```

もちろん、median.hヘッダはこの宣言だけではなく、宣言に使われる名前も含ませる必要があります。median関数の宣言にはvectorを使ってるので、medianの宣言の前に、この名前をコンパイラが認識するようにならなければなりません。

```
// median.h
#include <vector>
double median(std::vector<double>);
```

vectorヘッダをインクルードして、medianの宣言の中でstd::vectorが使えるようにしたのです。using宣言を使わず、std::vectorのようにstd::を省略しなかった理由は微妙です。

### 4.3 みんなまとめて

一般に、ヘッダファイルは必要な名前のみを宣言すべきです。必要な名前に限定しておくことで、ユーザの自由度を最大にしておけるからです。たとえば、median関数のユーザがvectorをどう表現するかわからないので、修飾つきのstd::vectorにしたのです。median関数のあるユーザはvectorにusing宣言を使いたくないかもしれません。しかし、ヘッダファイルでusing std::vectorを宣言していると、このファイルをインクルードしたプログラムでは、使いたくなくてもこの宣言が使われてしまうのです。要するに、ヘッダファイルはusing宣言ではなく、修飾された名前を使うべきなのです。

あと1ついうべきことがあります。それはヘッダファイルは1つのプログラムで2度以上インクルードされても問題が無いようにしておかなければならぬということです。実は今のmedian.hは、宣言しか含んでいないので、2度以上インクルードされても問題ありません。しかし、必要のない場合でも、複数回のインクルードに対処する仕組みを入れておくことをお勧めします。これはプリプロセッサというものをファイルに付け加えることができます。

```
#ifndef GUARD_median_h
#define GUARD_median_h
// median.h: 最終バージョン
#include <vector>
double median(std::vector<double>);
#endif
```

1行目にある#ifndefディレクティブ（directive）はプリプロセッサ変数（preprocessor variable）<sup>\*3</sup>とよばれるものが定義されているかどうかをチェックするものです。今の場合、それはGUARD\_median\_hで、これが定義されているかどうかをチェックしているのです。このプリプロセッサ変数はプログラムのコンパイルの仕方を制御するのに使われる変数です。しかし、プリプロセッサの詳細は本書の範囲を越えてしまうのでここでは説明しません。

今の場合、#ifndefディレクティブは、もし与えられた名前（プリプロセッサ変数の名前）が存在しなければ次に現われる#endifまでの間のものを処理するように、プリプロセッサに要求しているのです。この名前はプログラム全体の中で同じものがないようにします。そのため、GUARD\_のような他では使われそうにない文字列を一部に使ってこの名前を定義しているのです。

最初のmedian.hがインクルードされるときには、GUARD\_median\_hは未定義なので、プリプロセッサはこのファイルの中身をすべて処理します。そして最初にすることは、GUARD\_median\_hの定義です。そのため、2回目以降にこのファイルがインクルードされるときには、GUARD\_median\_hが定義されているので、処理は何もされないです。

最後に付け加えることは、#ifndefはコメントよりも前、ファイルの1行目にしてよいということです。

```
#ifndef variable
...
#endif
```

それはC++のコンパイラは、この構造のファイルを見つけ、そのプリプロセッサ変数variableが定義されていることがわかると、もう読み込みをしないからです。

<sup>\*3</sup> 訳注：ここではGUARD\_median\_hという名前の変数を定義して使っていますが、この名前はプログラマが自由に決められるものです。

#### 4.4 成績処理プログラムの分割

これまでに median 関数を分離してコンパイルする方法を見ました。次は、Student\_info と付随する関数を分離することです。

```
#ifndef GUARD_Student_info
#define GUARD_Student_info
// ヘッダーファイル Student_info.h
#include <iostream>
#include <string>
#include <vector>

struct Student_info {
    std::string name;
    double midterm, final;
    std::vector<double> homework;
};

bool compare(const Student_info&, const Student_info&);
std::istream& read(std::istream&, Student_info&);
std::istream& read_hw(std::istream&, std::vector<double>&);
#endif
```

この Student\_info.h ファイルでは、using 宣言を使わず標準ライブラリのものは明示的に std:: の修飾付きの名前を使っていること、Student\_info 構造体に関連している compare、read、read\_hw の宣言もしていることに注意してください。これらの関数を使うときには Student\_info を使うので、関数と構造体の定義をまとめておくことには意味があります。

関数の定義はソースファイルに記述されますが、それは次のようなものになります。

```
// Student_infoと関連する関数のソースファイル
#include "Student_info.h"
using std::istream; using std::vector;
bool compare(const Student_info& x, const Student_info& y)
{
    return x.name < y.name;
}

istream& read(istream& is, Student_info& s)
{
    // §4.2.2にあるこの関数の定義
}

istream& read_hw(istream& in, vector<double>& hw)
{
    // §4.1.3にあるこの関数の定義
}
```

最初に Student\_info.h をインクルードしているので、このファイルには関数の宣言と定義の両方が書かれています。この重複に害はありません。それどころかよいことなのです。コンパイラに宣言と定義が一貫している

#### 4.4 成績処理プログラムの分割

かチェックされることになるからです。ただし、このようなチェックは本来プログラム全体に行われるべきものなので、ヘッダーファイルをインクルードしたソースファイルをチェックしておくことはとても有益ですが、完全ではありません。

チェックが有益である理由とそれが完全ではない理由には共通の原因があります。C++ 言語では、関数の宣言と定義は戻り値とパラメータリストまで含めて一致していかなければならないのです。これはコンパイラのチェックが完全であることを意味していますが、それがどうしてチェックの不完全さにつながるかわかりますか。それは、もし宣言と定義が一致していないければ、コンパイラはこれらがオーバーロードにより異なる形をしていると判断し、見つからない定義はどこか別のところにあると仮定するからです。たとえば、§ 4.1.1 で定義した median に対し、誤って次のように不完全な宣言をしたとします。

```
int median(std::vector<double>); // 戻り値がdoubleでない
```

これをコンパイラは、コンパイル時にエラーとします。というのは、median(vector<double>) は同時に double と int の 2 種類の値を戻せないからです。しかし、次のように宣言してしまうと

```
double median(double); // 引数の型がvector<double>でない
```

コンパイラは（これ自身は）エラーとは判断しません。それは median(double) という関数がどこかに定義されているかもしれないからです。この関数を使うときに初めて、その定義が探され、定義がなければその時点でのみエラーになるのです。

ソースファイルで using 宣言をすることに問題はありません。ヘッダーファイルと違い、ソースファイルはこれらの関数を使うプログラムに何の影響も与えないからです。したがって、using 宣言を使うかどうかは、純粋にこのファイルだけの問題になるのです。

最後に残ったのは、いくつかのオーバーロードバージョンのある grade 関数のヘッダーファイルを書くということです。

```
#ifndef GUARD_grade_h
#define GUARD_grade_h
// grade.h
#include <vector>
#include "Student_info.h"

double grade(double, double, double);
double grade(double, double, const std::vector<double>&);
double grade(const Student_info&);

#endif
```

このようにオーバーロードされた関数の宣言を一箇所にまとめておくと、それぞれの違いが見やすいことにも注意してください。これらは相互に関連しているので、その定義は 1 つのファイルにまとめることにしましょう。ファイルの名前は使用するコンパイラにあわせて、grade.cpp、grade.C、grade.c などにします。

```
#include <stdexcept>
#include <vector>
#include "grade.h"
#include "median.h"
```

```
#include "Student_info.h"
using std::domain_error; using std::vector;
// §4.1. §4.1.2, §4.2.2にあるgrade関数の定義
```

## 4.5 成績処理プログラム改訂版

これで、ようやくプログラムを書くことができます。

```
#include <algorithm>
#include <iomanip>
#include <iostream>
#include <iostream>
#include <stdexcept>
#include <string>
#include <vector>
#include "grade.h"
#include "Student_info.h"
using std::cin;           using std::setprecision;
using std::cout;          using std::sort;
using std::domain_error; using std::streamsize;
using std::endl;          using std::string;
using std::max;           using std::vector;
int main()
{
    vector<Student_info> students;
    Student_info record;
    string::size_type maxlen = 0; // 最長の名前の長さ
    // 全学生のデータを読み込む
    // 不要な表明：studentsが読み込んだ全学生のデータを保持し
    // maxlenがstudents内の最長の名前の長さになる
    while (read(cin, record)) {
        // 最長の名前の長さを見つける
        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する
    sort(students.begin(), students.end(), 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) {
        // 名前を出力しその右側にmaxlen + 1個の空白を入れる
        cout << students[i].name
            << string(maxlen + 1 - students[i].name.size(), ' ');
        // 最終成績を計算し出力する
        try {
            double final_grade = grade(students[i]);
            streamsize prec = cout.precision();
            cout << setprecision(3) << final_grade
                << setprecision(prec);
        } catch (domain_error e) {
            cout << e.what();
        }
    }
}
```

```
}
cout << endl;
}
return 0;
}
```

このプログラムはストレートな構造をしています。いつものように `include` と `using` 宣言から始めています。もちろん、これらはソースファイルで使うもののヘッダと宣言だけです。このプログラムではライブラリのヘッダに加えて、自分で書いたヘッダファイルをインクルードしています。これらのヘッダのインクルードで、`Student_info` 型や最終成績の計算のために `Student_info` オブジェクトを処理する関数の宣言が読み込まれるので、`main` 関数自身は、§ 4.2.3 のものと同じです。

## 4.6 詳細

プログラム構造：

```
#include < ライブラリヘッダ>
角カッコ(<>)でライブラリ(システム)ヘッダを囲む。ライブラリヘッダはファイルかもしれないしそうでないかもしれない。
#include "ユーザ定義のヘッダファイル名"
ダブルクォーテーションマーク"でユーザ定義のヘッダファイルをインクルードできる。特に、ユーザ定義のヘッダには拡張子.hを付ける。
```

ヘッダファイルは「`#ifndef GUARD_ヘッダの名前`」のようなディレクティブを付けることで重複インクルードを避ける。ヘッダでは使わない名前の宣言はするべきでない。特に `using` 宣言をしないこと。標準ライブラリのものを使う場合は、明示的に `std::` を付けて使う。

型：

<code>T&amp;</code>	型 T の参照(リファレンス)を表す。関数内部で引数を変更する予定のときにパラメータを参照にするという使い方がもっとも多い。このような場合、引数は左辺値(lvalue)である必要がある。
<code>const T&amp;</code>	変更する予定のない型 T のオブジェクトへの参照を表す。普通は、関数のパラメータのコピーを省くために使われる。

構造体： 構造体は 0 個以上のメンバを持つ型である。構造体のオブジェクトはそのメンバのインスタンス(オブジェクト)を内部に持つ。

構造体は次のように定義される：

```
struct type-name {
    type-specifier member-name;
    ...
}; // セミコロンがある
```

他の定義と同様に、構造体の定義は1つのソースファイルに重複して書くことはできない。そのため、通常、重複インクルードを避けるようにヘッダを書く必要がある。

**関数：** 関数を使うファイル内には、その関数の宣言が必要。また、定義は重複してはならない。宣言と定義は同じ形である。

```
ret-type function-name ( param-dcls );           // 関数の宣言
/ inline / ret-type function-name (param-decls) { // 関数の定義
    // 関数の定義はここに書く
}
```

ここで戻り値の型 *ret-type* は関数が戻す値の型を意味し、パラメータの宣言 *param-dcls* はコンマで区切られた関数のパラメータの型のリストである。関数は先に宣言されていなければ、呼び出すことはできない。引数の型は対応するパラメータの型と同じかそれに変換されるものでなければならない。戻り値の型がより複雑な場合、関数の宣言や定義の形は違ってくる。これについては§ A.1.2に書いたので参照のこと。

関数名はオーバーロードできる。つまり、同じ名前でも異なる個数や型のパラメータを持つ関数を定義できる。コンパイラは同じ型の *const* 付きの参照と *const* なしの参照を異なる型として区別する。

オプションとして関数の定義に *inline* を付けることができる。これは関数呼び出しのオーバーヘッドを避けるため、可能な場合は、関数を呼び出している場所に関数の定義コードを展開するという意味である。ただし、必要なら展開されるコードは定義コードを少し変形したものになる。これをするためにコンパイラは関数の定義を知っている必要がある。そのため *inline* 関数の定義はソースファイルではなくヘッダファイルに書き込む。

#### 例外処理：

```
try { /* コード 例外を投げる（スローする）かもしれないブロックの始めに書く。 */
} catch(t) { /* コード */ }
    try のブロックを終了し型 t に一致する例外の処理をする。catch の後の中カッコの何に
    は型 t の例外を処理するためのコードを書く。
throw e;           現在の関数を終了し、例外の値 e を呼び出し側に投げる。
```

**例外クラス：** ライブラリはいくつかの例外のクラス（型）を定義している。それぞれがどのような例外を報告するものは名前から推測することができる。

logic_error	domain_error	invalid_argument
length_error	out_of_range	runtime_error
range_error	overflow_error	underflow_error
e.what()	エラーを起こした原因についての報告を戻す	

ライブラリにある仕組み：

s1 < s2 文字列 s1 と s2 を辞書的順序で比較する。

#### 4.6 詳細

s.width(n)	ストリーム s の次の出力の幅を n にする。（もし n を省略すると幅は変えられない。）出力の右側には空白が入れられる。前の幅を戻す。標準の出力演算子は既存の幅を使い、幅をリセットするため width(0) を呼ぶことに注意。
setw(n)	出力ストリーム s に対して s.width(n) を呼ぶのと同じ効果を持ち、streamsize (§ 3.1) の型の値を戻す。

#### 課題

- 4-0 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。  
 4-1 § 4.2.3 で、max の引数の型は正確に一致していないと書きました。下のコードはうまくいくでしょうか。もし、問題があるとすると、直せますか？

```
int maxlen;
Student_info s;
max(s.name.size(), maxlen);
```

- 4-2 100までの整数の2乗を計算し書き出すプログラムを書いてください。ただし、このプログラムの出力は2列で、1列目には1から100までの数字、2列目にはその2乗が書かれるものにしてください。setwなどを使って出力がそろいうようにしてください。  
 4-3 上のプログラムで考える数を1000未満とし、setwを使わないなどのような問題が起るでしょうか。それから、考える数の最大値を i とし、setwの引数を毎回変えなくても、任意の i で綺麗に出力できるようにプログラムを書いてください。  
 4-4 上のプログラムで考える数を int ではなく double にしてください。setwなどを使って出力がそろいうようにしてください。  
 4-5 単語を入力ストリームから読み込み vector に入れる関数を書いてください。この関数を使って、入力された単語数を数えるプログラムとそれぞれの単語が何回現われたかを数えるプログラムを書いてください。  
 4-6 Student\_info 構造体を書き換え、読み込み時に計算し最終成績だけを保持するように直してください（read や grade も書き換える必要があります）。

- 4-7 vector<double>オブジェクト内の数値の平均を計算するプログラムを書いてください。  
 4-8 次のようなコードが正しいとすると、f の戻り値は何だと推測されるでしょう。

```
double d = f()[n];
```