

```

    using std::cout;
    cout << "*";
    ++k;
}
std::cout << std::endl; // ここではstd::が必要
return 0;
}

```

第3章

たくさんデータを扱う

前2章のプログラムは、文字列を読み込んでそれに少々の飾りをつけて出力する程度のものでした。もちろん、大抵のプログラムが解決しなければならない問題は、もっと複雑です。そのような複雑な問題の代表例に、「同様のデータを多数扱わなければならない」というものがあります。

実は、ここまでに見てきたプログラムも1つの文字列(string)が多数の文字を含んでいるという意味で同じ問題を処理しています。実際、不特定数の文字を1つのstringオブジェクトに格納できるという事実がプログラムを簡単にしているのでした。

この章では、試験の点数と宿題の点数から学生の最終成績を計算するというプログラムを書きながら、たくさんのデータを扱う他の方法を見ていきます。そこで、あらかじめいくつあるかわからない、すべての点数を格納する方法を見ることになります。

3.1 学生の成績をつける

学生の成績をつけるのに、中間試験の点数がその20%、期末試験の点数がその40%になり、宿題の平均が残りの40%になるとしましょう。まず、学生が成績を計算するプログラムを次のように考えてみます。

```

#include <iomanip>
#include <ios>
#include <iostream>
#include <string>
using std::cin;           using std::setprecision;
using std::cout;          using std::string;
using std::endl;          using std::streamsize;

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) : ";
// これまでに読み込んだ宿題の点数の数 (count) とそれらの合計 (sum)
int count = 0;
double sum = 0;
// 読み込みようの変数
double x;
// 不要な表明：
// ここまでcount個読み込み、
// sumはそれらの点数の合計
while (cin >> x) {
    ++count;
    sum += x;
}
// 結果を出力
streamsize prec = cout.precision();
cout << "あなたの最終成績：" << setprecision(3)
      << 0.2 * midterm + 0.4 * final + 0.4 * sum / count
      << setprecision(prec) << endl;
return 0;
}

```

いつものように、プログラムは、使用するライブラリ用の#include ディレクティブと using 宣言ではじまっています。ただし標準ライブラリから、新しく<iomanip>と<iostream>が含まれています。<iostream>ヘッダには、入出力ライブラリが大きさを表すのに使う streamsize という型が定義され、<iomanip>ヘッダには setprecision という操作子（マニピュレータ）が定義されているのです。これは、出力する数字の桁数を指定するときに使うものです。

実は endl も操作子の一種ですが、このときは<iomanip>ヘッダをインクルードする必要はありませんでした。それは endl はとても頻繁に使われる所以<iomanip>ではなく<iostream>に含まれるようしてあるからです。

プログラムは、名前、中間試験の点数、期末試験の点数をユーザに入力させています。その後で、学生の宿題の点数を、end-of-file (ファイルの終わり) シグナルを読み込むまで入力させています。このシグナルをプログラムに送る方法は環境によって異なりますが、入力の終わりを示すのは一般的なものです。たとえば、マイクロソフトの Windows なら [Ctrl]+[D] で、Unix や Linux では [Ctrl]+[D] で、このシグナルが送られます。

宿題の点数を読み込む際には、count という変数に読み込んだ宿題の数を、また sum という変数には、そこまでの点数の合計を記録していきます。そして、全点数を読み込むと、プログラムはその学生の最終成績を出力します。この計算に、count と sum が使われています。

このプログラムの大部分はすでに見てきたものですが、いくつか新しいことも出ています。これについて説明しましょう。

最初の新しいことは、学生の試験の点数を入力する場所にあります。

```

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;

```

1行目は単に学生に対する指示を出力しているだけです。2行目は midterm と final という 2 つの double 型の変数を定義しています。double は倍精度浮動小数点数とよばれる組み込みの型です。単精度浮動小数点数で

3.1 学生の成績をつける

ある float という型もあり、その方が適切に思えるかもしれません、大抵の場合、浮動小数点数の計算では double を使うのがよいのです。

これらの名前はメモリが貴重であった時代の名残です。float は 10 進数でたかだか 6 衔の精度を保証するものです。これでは家の値段すらまともに表せません。double なら最低でも 10 衔の精度が保証されていますし、現実には、私の知る限りの環境では 15 衔の精度が保証されているのです。現代のコンピュータ上では、double は float よりはるかに正確で、しかもあまり遅くはないのです。それどころか早い場合もあります。

これで、midterm と final の定義ができました。次に入力です。出力演算子 (§ 0.7) のように、入力演算子も左側のオペランドを戻します。このおかげで出力演算子のように入力演算子をつなげて書くことができます。これにより、

```
cin >> midterm >> final;
```

は、

```
cin >> midterm;
cin >> final;
```

と同じ働きになるのです。まず、midterm に入力された値が格納され、次に final に入力された値が格納されるのです。

次のステートメントは

```
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) : ";
```

です。このステートメントは文字列を 2 つ出力しているようですが、よく見ると << は 1 つか使われていません。実は、空白のみで分けられている文字列リテラルは、実際には 1 つにまとめられるという規則があるので。したがって、上のステートメントは

```
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) : ";
```

と同じなのです。これを 2 行にしたのは、1 行を短くしてプログラムを読みやすくするためです。

プログラムの次の部分では、入力された値を格納する変数を定義しています。おもしろいのは、

```
int count = 0;
double sum = 0;
```

です。これは count と sum の両方を 0 に初期化しています。しかし、0 という値は int 型なので、sum の初期化ではコンパイラが 0 を自動的に double に変換するのです。sum の値は 0.0 で初期化することで、変換しないで済ませることもできますが、今の場合では実際的な違いはありません。ちゃんとしたコンパイラなら、コンパイル時にこの変換を行うので、実行時のオーバーヘッド（余分な時間）は起らず、まったく同じ結果になるのです。

変換より重要なことは、これらの変数に初期値を与えていたという事実そのものです。変数の初期値を与えないといと、暗に、デフォルトの初期化 (default-initialization) を利用することになるのです。このデフォルトの初期化は変数の型によって異なります。クラスのオブジェクトの場合、クラス自身がどのように初期化するか定めることになっています。たとえば、§ 1.1 で書きましたが、string オブジェクトは明示的に初期化しなければ、

空の文字列になるようになっています。組み込み型のローカル変数は、特別な値に初期化されることはありません。

明示的に初期化しない組み込み型のローカル変数の値は未定義（undefined）になります。これは、その変数が作られるときに、たまたまそのメモリ上にあったもの、つまりでたらめでゴミのようなものが、そのまま値になるという意味です。未定義な値は有効な値で上書きするべきです。それ以外のことはルール違反です。多くのコンパイラはこのルール違反を検出せず、未定義の値にアクセスするようなコードを許します。しかし、そのようなコードはクラッシュか間違いを引き起こします。たまたまメモリ上にあった値が正しい値であるということはほとんどないからです。それどころか、しばしば、その型には無効な値であるのです。

もし、`sum` と `count` を初期化しておかなければ、このプログラムは誤った結果を出してしまうでしょう。`count` はインクリメントの前にまず読まれ、`sum` も読み込んだ宿題の点数を足すために読まれます。このように、変数の未定義の値が使われてしまうからです。しかし、変数 `x` を初期化する必要はありません。これは入力の読み込みに使われるのですが、入力が読み込まれるときには、その前に入っていた値は消されてしまうからです。

`while` 文では、その条件がまだ説明していない形をしています。

```
// 不要な表明:  
// ここまでcount個読み込み、  
// sumはそれらの点数の合計  
while (cin >> x) {  
    ++count;  
    sum += x;  
}
```

`while` 文の説明によれば、条件「`cin >> x`」が成立する限り、`while` の中身が実行されることになります。「`cin >> x`」を条件として扱うということの詳細な意味は § 3.1.1 で説明しますが、ここでは「`cin >> x`」は読み込みに成功すれば条件として `true` になるということだけ述べておきましょう。

`while` の中身では、インクリメントと複合代入演算子を使っていますが、これらは第 2 章で説明しました。`++count` は `count` に 1 を足すという意味で、`sum += x` は `sum` に `x` を足すという意味です。

最後は、このプログラムがどのように出力するかの説明です。

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

最終的には有効数字 3 桁で成績を出すことが目的ですが、これは `setprecision` でできます。`endl` と同様に `setprecision` も操作子（マニピュレータ）で、これは出力ストリームに作用し、次の出力を与えられた有効桁数にするものなのです。`setprecision(3)` とすることで、有効数字を 3 桁にし、成績を基本的に整数部が 2 桁、小数部が 1 桁にできます。

`setprecision` はそれに続く `cout` への出力の有効桁数を決めてしまいます。今のプログラムではこの部分が最後なので、次の出力というものはありません。しかし、それでも、もとの有効桁数に戻しておくほうが賢明でしょう。そのため `precision` という `cout` のメンバ関数（§ 1.2）を使っているのです。この関数は、浮動小数点数の出力に使っている有効桁数を戻すものです。はじめに `setprecision` で有効桁数を 3 にし最終成績を

3.1 学生の成績をつける

出力していますが、最後に再び有効桁数を `precision` で与えられたものに戻しているのです。成績の計算には算術演算子をいくつか使っています。明らかに、*はかけ算、/はわり算、+はたし算です。¹

また、以下のように、メンバ関数の `precision` を有効桁数の設定に使うこともできます。

```
// 下は、有効桁数を3に設定し、前の有効桁数を戻り値として受け取っている  
streamsize prec = cout.precision(3);  
cout << "Your final grade is "  
    << 0.2 * midterm + 0.4 * final + 0.4 * sum / count << endl;  
// 有効桁数をもとに戻している  
cout.precision(prec);
```

しかし、`setprecision` 操作子を使えば、基本的でない有効桁数を使う場所をプログラム中で最小限にできます。この理由で、こちらを使いました。

3.1.1 入力の最後を見る

このプログラムの概念上で本当に新しいことは `while` 文の条件です。その条件には、明示的ではありませんが、`istream` そのものが使われています。

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

このステートメントは実質的に `cin` からの読み込みを意図するものです。読み込みが成功すれば `x` には読み込んだ値が格納され、`while` の中身も実行されます。読み込みが失敗（入力がなくなったり、`x` の型と違うものが入力されると失敗します）すると、`while` の条件は `false` と解釈されます。このとき、`x` に格納されている値を使ってはいけません。

このコードがどのように動作するかは、少し難しいかもしれません。まず、`>>` 演算子は左オペランドを戻すことを思い出しましょう。つまり、「`cin >> x`」の戻り値を求めるということは、実際には `cin >> x` を実行した後に、`cin` の値を求めるということです。たとえば、次のように `x` に値をひとつ読み込み、それが成功したかどうかを確認することもできます。

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

これは、次のような組み合わせと同じ意味を持ちます。

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

つまり、`cin >> x` を条件にするということは、単に、その条件を調べているのではなく、その副作用として `x` に値を読み込んでいるのです。あとは `cin` を `while` の条件に使う意味さえわからぬわけです。

`cin` は標準ライブラリにある `istream` 型ですから、`if (cin)` や `while (cin)` の意味を知るためにには、`istream` の定義を見る必要があります。しかし、その定義はとても複雑なので詳細は § 12.5 で説明します。詳細はそのときに見るにしても、実は、すでに重要なことはわかっているのです。

¹ 註注：なお、このプログラムは、データ数が 0 の場合、0 による除算が行われ実行時エラーになりますので注意してください。

第2章で扱った条件には、bool値を戻す関係演算子を使いました。また、数値を戻すエクスプレッショ
ンも使いました。条件として使われると、数値はbool値に変換されるのです。0以外の値はtrueになり、0は
falseになります。結局問題になるのは、cinを条件として使えるようにする変換がistreamにあるかどうか
です。cinの値そのものの説明は今はしませんが、要するにこれがbool値に変換されるのだとい
うことを知っていればよいのです。これにより、cinを条件の中で使うことができるのです。この値はistream
オブジェクトであるcinの内部状態によって異なります。そして、内部状態には最後の読み込みの結果が残
されているのです。つまり、cinを条件に使うということは、最後の読み込みが成功したかどうかをチェックする
と同じことなのです。

ストリームからの読み込みが失敗するのは以下のような場合です。

- 読み込みが入力用のファイルの最後に到達した場合
 - int を入力させるところで整数でないものを入力してしまうというように、入力されるものが期待されているものではなく、そのように変換もできない場合
 - システムが入力装置のハードウェア的な問題を見つけた場合

どのような場合でも結果は同じになり、`cin`を条件に使うと、これは `false`として解釈されるのです。また、一度読み込みに失敗すると、リセットしない限りそのストリームからは読み込みができないになります。これについては § 4.1.3 で説明します。

3.1.2 ループの不变な表明

§ 2.3.2 で見たようなループの「不变な表明」を今の場合に理解するためには、while の条件の副作用のことを考慮しなくてはなりません。この副作用は不变な表明の成立と不成立に影響を及ぼすからです。cin >> x で読み込みに成功すると、不变な表明の前半部分「ここまで count 個読み込み、」が正しくなります。つまり、条件の副作用の効果も考えなければならないのです。

条件の副作用によるものではありません。条件を評価する前は、不变な表明が成り立っているとしているので、すでに count 個の成績データを読み込んだと考えます。そこで、`cin >> x` が読み込みに成功したなら、読み込んだ成績データの数は `count + 1` になります。したがって、`count` を 1 増やせば、また不变な表明が成立するようになります。しかし、こうすることによって、不变な表明の後半部分「`sum` はそれらの点数の合計」が成り立たなくなります。なぜなら `count` を 1 増やしたため、`sum` は `count` ではなく `count - 1` 個の成績の和になってしまうからです。これは `sum += x;` また成立するようにできます。これらをまとめれば、`while` の中の処理全体で不变な表明は、再び成り立つようになります。

もし、読み込みに失敗すれば、もうデータを入れられなくなるので、不变な表明はやはり成立します。そのため while の終了後に条件の副作用を考える必要はありません。

3.2 平均ではなくメジアンを使う

ここまで作ってきたプログラムにはデザイン上の欠点があります。それは宿題の点数を読み込むとすぐに捨てるということです。平均値を計算するならそれでもよいでしょう。しかし、平均値ではなくメジアン²⁴が

*2 註注：複数個のデータの真ん中の値。

3.2 平均ではなくメジアンを使う

いいときはどうしたらいいでしょうか。

データのメジアンを見つけるストレートな方法は、それらを昇順か降順に並べその真ん中の値を取り出すというものです。また、もしデータが全部で偶数個ある場合は、真ん中の 2 つのデータの平均値がメジアンです。いくつか特にひどい成績があるだけで平均値は下がりますが、メジアンは全体の傾向が反映されるという意味で平均値より役に立ちます。

メジアンを求めるためには、これまでのプログラムを根本的に書き換える必要があることがわかります。総数がわからないデータのメジアンを求めるには、すべてを読み終えるまでどこかにデータを保持しておかなければなりません。一方、平均値を求める場合は、データ数とデータの合計がわかればよいのです。平均値は、データの合計をデータ数で割れば求められるのですから。

3.2.1 データを vector に保持する

メジアンを求めるには、宿題のすべての成績データを読み込んで保持し、それをソート（整列）し、その真ん中の値（もしくは真ん中の2つの値）を取り出す必要があります。これを簡単に効率よく行うために、次のことを考えます。

- 始めに何個あるかわからないデータを、1度に1つずつ読み込みそれを保持（記録）する
 - それらをソートする
 - 真ん中の値を効率よく取り出す

標準ライブラリには、これらの問題を解決するのに便利な `vector` という型があります。`vector` は、決まった型のオブジェクトの列を保持するのですが、データを追加する場合にはサイズが必要なだけ大きくなり、また、個別のデータを取り出すことができます。

まず、宿題の個数とその点数の合計を計算するのではなく、個々の点数を `vector` に入れるようにプログラムを書き直しましょう。とのコードは次のようになっていました。

```
// もとのコード（一部）
int count = 0;
double sum = 0;
double x;
// 不変な表現：
// ここまでcount個読み込み
// sumはそれらの点数の合計
while (cin >> x) {
    ++count;
    sum += x;
}
```

このコードは、読み込んだ点数の数とそれまでの合計を記録しています。読み込みの各ステップでこれらの変数を処理するため、不变な表明は比較的複雑にもなっています。それに比べ、`vector` を使い点数を記録すると極めて簡単になります。

```
// 改訂版  
double x;  
vector<double> homework
```

```
// 不変な表現：ここまで読み込んだ点数はすべてhomeworkに記録されている
while (cin >> x)
    homework.push_back(x);
```

ファイルの終わりまで、データを1度に1つずつxに読み込むというコードの基本構造は変わっていません。違ひは読み込んだデータの扱いです。

まずhomeworkの説明をしましょう。これはvector<double>型の変数です。vectorは複数のデータを保持するコンテナ(container)とよばれるもの一種です。vector内のすべてのデータは同じ型ですが、1つ1つのvectorは違う型のオブジェクトを保持することができます。そのためvectorを定義するときは、その中に入れるオブジェクトの型を指定する必要があります。homeworkの定義は、これがdouble型のオブジェクトを保持するvectorであることを示しています。

vectorの定義には、テンプレートクラス(template class)とよばれる仕組みを使っています。テンプレートクラスの作り方は第11章で見ます。ここで重要なことは、vectorという型とvectorが保持するデータの型は違うものだということです。vectorの保持するデータの型は、角カッコ<>の中で指定します。たとえば、vector<double>はdoubleのデータを多数保持できる入れ物となり、vector<string>はstringデータの入れ物になります。

whileループのすることは、データを標準入力から読み込み、それをvectorに格納しているだけです。以前と同じように、読み込みはファイルの終わりに到達するかdouble以外のデータに出会うまで続けます。ここで新しいのは

```
homework.push_back(x);
```

です。§1.2のgreeting.size()でsizeがstring型のメンバ関数であったのと同じように、push_backはvector型のメンバ関数で、これをhomeworkという名前のオブジェクトのために使っているのです。これは関数でxを引数に取っています。push_backの仕事はvectorの最後にデータを付け加えることです。つまり、push_backは、その引数をvectorの最後に押し込み、その副作用として、vectorの大きさを1つ増やします。

push_backは今の私たちの目的にとてもよくあってるので、これによりループの不变な表現が成り立つことは明らかです。また、そのため、ループ終了後は期待通りにすべての宿題の成績を読み込んだことになっています。

次に考えるべきことは、出力です。

3.2.2 出力を作る

§3.1にあるもとのプログラムでは、学生の最終成績を出力のエクスプレッションの中で計算しました。

```
streamsize prec = cout.precision();
cout << "Your final grade is " << setprecision(3)
     << 0.2 * midterm + 0.4 * final + 0.4 * sum / count
     << setprecision(prec) << endl;
```

ここで、finalとmidtermが試験の点数を表し、sumとcountが宿題の点数の合計と宿題の数を表しています。

§3.2.1で述べましたが、データをソート(整列)し、その真ん中、あるいはデータ数が偶数のときは真ん中の2つのデータの平均がメジアンです。これがメジアンを見つける一番簡単な方法です。今のは、出力と計算

を分けた方がわかりやすいでしょう。

メジアンを見つけるため、vectorであるhomeworkのサイズを2回知る必要があります。1度は、サイズが0でないことを確かめるとき、もう1度は真ん中を見つけるときです。サイズを2度求める为了避免するために、これはローカルな変数に保持することにします。

```
typedef vector<double>::size_type vec_sz;
vec_sz size = homework.size();
```

vector型にはvector<double>::size_typeという型とsizeという関数が付随しています。これらのメンバはstringのメンバと同様に使うことができます。size_typeはvectorのサイズを格納できることが保証されたunsigned型です。また、size()はsize_type型の値であるvectorのサイズを戻す関数です。

vectorのサイズが2度必要になるので、それをローカルな変数に格納します。vectorのサイズは環境によって異なるかもしれませんので、その違いを避けるために別の型を使わず、ライブラリがコンテナのサイズ用に定義しているsize_typeを使うのがよいです。この理由で、上ではsize_type型を使っています。

しかし、この名前は長くて読み書きが面倒です。それでプログラムを短くするために、typedefという機能を使います。typedefは変数の定義に使うのではなく、型名の別名を定義するために使います。つまり、上のコードでは、vector<double>::size_typeという型の別名としてvec_szを定義しています。このtypedefを使って定義した名前のスコープは他の名前と同じです。そのため、現在のスコープが終わるまでvec_szをsize_typeの別名として使ってよいのです。

さて、homework.size()の戻り値を格納する変数の型を決めたので、これをローカル変数に格納するのは簡単です。その変数名をsizeとします。sizeというのはvectorのメンバ関数と同じ名前ですが、混乱することはありません。なぜならvectorオブジェクトのサイズを求めるために、そのメンバ関数を使うときは、そのオブジェクトの右側にドット「.」を置いてその右にsizeと書くからです。言葉を変えると、vectorのメンバ関数のsizeと今定義したsizeでは、スコープが違うのです。これらの名前はスコープが違うので、コンパイラは(そしてプログラマも)どちらのsizeだろうかと混乱することはあります。

次に、データがない場合にメジアンを見つけることは意味が無いので、データがあるかどうかを調べます。

```
if (size == 0) {
    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;
}
```

データがないかどうかはsizeが0かどうかを調べればわかります。もしデータがない場合、とても微妙ですが、警告を表示しプログラムを止めることにします。ここで失敗を示すために1を戻すことになりました。第0章で説明しましたが、システムはmainが0を戻したときに、プログラムが成功して終了したと判断します。0以外の戻り値がどう判断されるかは環境によって違いますが、大抵の場合、0以外は失敗と判断するのです。

データがある場合は、メジアンの計算に進みます。まず、最初に行うことはデータのソートです。これはライブラリの関数を呼び出すことでできます。

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

この sort 関数は<algorithm>ヘッダに定義されているもので、コンテナ内のデータを非降順に並べ替えるのです。「非降順」はほとんど「昇順」と同じですが、同じデータもあり得るためそういう言葉を使います。

sort の引数はソートする範囲を指定しています。このために、vector には 2 つのメンバ関数 begin と end があります。これらについては § 5.2.2 でより詳しく説明しますが、ここでは homework.begin() が homework という vector オブジェクトの先頭を意味し、homework.end() が homework の末尾（最後の 1 つ後）を意味するということだけ述べておきます。

この sort 関数は、新しくソートされた vector オブジェクトを生成するのではなく、指定された vector オブジェクトのデータを並べ替えるのです。

homework をソートした後は、その真ん中の要素を見つけます。

```
vec_sz mid = size/2;
double median;
median = size % 2 == 0 ? (homework[mid] + homework[mid-1]) / 2
                      : homework[mid];
```

上の最初の行では、vector の真ん中を見つけるため、size を 2 で割っています。もし size が偶数なら割り算の結果は正確なものです。もし奇数なら、割り算の結果の小数点以下は捨てられます。

メジアンの計算の仕方は、size が偶数か奇数かで違います。もし size が偶数なら、メジアンは真ん中の 2 つの要素の平均になります。size が奇数のときは、真ん中の要素は 1 つありますので、それがメジアンです。

メジアンを求めるエクスプレッションでは、2 つの新しい演算子を使いました。剩余 (remainder) 演算子%と条件 (conditional) 演算子で、後者はしばしば?:演算子とよばれるものです。

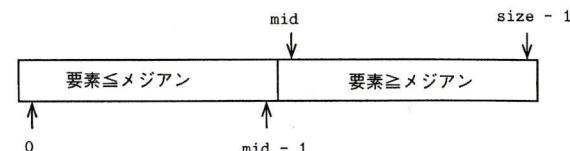
剩余演算子の%は左のオペランドを右のオペランドで割った余りを戻す演算子です。size を 2 で割った余りが 0 なら、プログラムが読み込んだデータ数は偶数ということになります。

条件演算子は if-then-else の短縮形です。上のコードでは、まず size%2==0 を評価します。?:前のエクスプレッションは bool 値を戻す条件にするのです。これが true なら?:の間の値が戻され、false なら?:の後の値が戻されるのです。つまり、読み込んだデータの数が偶数なら、median には homework[mid] と homework[mid-1] の値の平均が入ります。もし、読み込んだデータの数が奇数なら median には homework[mid] が入ります。&&や||と同様に?:演算子も一番左のオペランドを最初に評価します。そして、その結果によって、残りのオペランドの 1 つだけが評価されます。

homework[mid] や homework[mid-1] は、vector オブジェクトの要素のアクセス方法の 1 つを示しています。vector オブジェクトのすべての要素はインデックス (index) とよばれる整数值を付随して持っています。つまり、homework[mid] はインデックスが mid の要素ということになるのです。§ 2.6 から類推できるかもしれません、homework の最初の要素が homework[0] で、最後の要素が homework[size - 1] です。

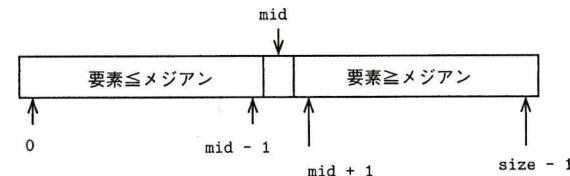
コンテナ内の要素はみな指定された型の（名前のない）オブジェクトです。したがって、homework[mid] などは double のオブジェクトです。そのため、double 型のものにできることは、なんでも homework[mid] にできます。ここではこれらを足して 2 で割って平均を出しています。

homework の要素のアクセス方法がわかったので、メジアンの計算方法をもう 1 度説明します。まず、size が偶数だとしましょう。そのとき mid は size/2 になります。そうすると、homework の真ん中には要素がなく、真ん中の両側に mid 個の要素があることになります。



真ん中の両側に mid 個の要素があるので、真ん中の両側の要素のインデックスは mid と mid-1 であることがわかります。したがって、メジアンはこれらの平均になるわけです。

もし、全要素数が奇数であれば、割り算の結果の小数点以下が捨てられるので、mid は実際には (size - 1) / 2 に等しくなります。これがメジアンのインデックスなのです。



どちらの場合も、vector の要素にインデックスを使ってアクセスできるということを使っています。

メジアンの計算が終われば、次にすることは、最終成績を計算し出力することです。

```
streamsize prec = cout.precision();
cout << "あなたの最終成績：" << setprecision(3)
      << 0.2 * midterm + 0.4 * final + 0.4 * median
      << setprecision(prec) << endl;
```

このプログラムは § 3.1 のものに比べて、ずっと多くの仕事をしますが、それほど複雑ではありません。特に、vector オブジェクトである homework は、適当に自分のサイズを大きくして、学生がした不特定個の宿題の点数を保持するので、プログラム中でそのためのメモリ確保を考える必要はないのです。このような仕事はすべて標準ライブラリが行ってくれるわけです。

ここでプログラム全体をまとめてみましょう。ここまでで説明していない部分は、#include ディレクトリと using 宣言といくつかのコメントだけです。

```
#include <algorithm>
#include <iomanip>
#include <iostream>
#include <ios>
#include <iostream>
#include <string>
#include <vector>
using std::cin;
using std::cout;
using std::endl;
using std::setprecision;
using std::sort;
using std::streamsize;
using std::string;
using std::vector;
```

```

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;
    double x;
    // 不要な表明：ここまで読み込んだ点数はすべてhomeworkに記録されている
    while (cin >> x)
        homework.push_back(x);
    // 宿題の点数が入力されたかをチェック
    typedef vector<double>::size_type vec_sz;
    vec_sz size = homework.size();
    if (size == 0) {
        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;
    }
    // 宿題の点数をソート
    sort(homework.begin(), homework.end());
    // 宿題のメジアンを計算
    vec_sz mid = size/2;
    double median;
    median = size % 2 == 0 ? (homework[mid] + homework[mid+1]) / 2
        : homework[mid];
    // 最終成績を計算して出力
    streamsize prec = cout.precision();
    cout << "あなたの最終成績：" << setprecision(3)
        << 0.2 * midterm + 0.4 * final + 0.4 * median
        << setprecision(prec) << endl;
    return 0;
}

```

3.2.3 追加コメント

前節のプログラムにはいくつか重要なポイントがあります。まず、homework が空の場合に、なぜプログラムを終了すべきなのかをもう少し検討しましょう。論理的には、データがない場合のメジアンは定義できません。つまり、どうすればよいかわからないのです。ですから、プログラムの終了は正しいと思います。しかし、もしプログラムの実行を続行した場合、どうなるかを知っておく価値があります。入力が無くそれをチェックしなかった場合、メジアンの計算のコードは問題を起こします。なぜでしょうか。

3.3 詳細

もし、宿題の成績データが1つも無ければ、homework.size() は0を戻し、size は0になります。同様に mid も0になります。したがって、homework[mid] は homework の最初の要素（インデックスが0）を見ることになります。しかし、homework にはデータが1つも入っていないのです！homework[0] というコードが実行されて何かが得られるということはありません。vector は自分でインデックスの有効性はチェックしないのです。それはユーザ（プログラマ）の責任なのです。

次に重要なことは、標準ライブラリの同様のものはみなそうなのですが、vector<double>::size_type が符号なしの整数型 (unsigned integral type) だということです。このような型はマイナスの数を保持することができません。代わりに 2^n (この n は環境によって変わります) の値を保持できます。そのため、homework.size() < 0 のようなチェックは決して必要にならないのです。この比較は、常に false になるからです。

また、普通の整数型と符号なしの整数型がエクスプレッションの中で混じると、普通の整数は符号なしの整数に変えられてしまいます。そのため、homework.size() - 100 のようなエクスプレッションは符号なしの整数になってしまい、決して0より小さくはありません。これは homework.size() < 100 であってもです。

最後に述べたいことは、必要なメモリを1度に確保するのではなく、入力するたびに vector<double>オブジェクトのサイズを大きくするという方法を取っているにもかかわらず、このプログラムの実行効率はとてもよいということです。

C++の標準に従っているライブラリの効率は保証されているのです。ライブラリ内にあるツールは、仕事をするだけでなく一定の基準以上の効率で仕事を実行するように決められているからです。したがって、ライブラリを安心して使うことができます。C++の標準は、以下のことを守るように決めています。

- vector に要素をたくさん付け加える場合、「時間が要素数に比例する」より遅くてはならない
- ソートされる要素数が n のとき、「sort にかかる時間が $n \log(n)$ に比例する」より遅くてはならない（ただし、これは平均的な入力の場合で、病的な入力の場合は遅くなるかもしれません）

その結果、基準を満たしている標準ライブラリを使えば、プログラム全体の実行速度も、宿題の数 n に対して $n \log(n)$ より遅くならないのです。実際、標準ライブラリは効率にとても気を使って作られているのです。そもそも C++ は実行効率が重要なアプリケーションを開発するために作られた言語であり、そのスピード尊重はライブラリにも受け継がれているのです。

3.3 詳細

ローカル変数： 意識的に初期化しないとデフォルトで初期化される。組み込み型のデフォルトの初期化とは、値がどうなるかわからないということ。保持している値のわからない変数は、代入式の左辺でのみ使うべき。

型の定義：

`typedef type name;` name を type の別名として定義

vector 型： 特定の型のオブジェクトを連続して保持するコンテナの一種で、<vector>ヘッダの中で定義されている。vector のサイズは、実行中に大きくなる。重要な操作等のいくつかをまとめると次のようになる。

```

vector<T>::size_type
    vector に入るオブジェクトの最大数を表すことができる整数の型。
v.begin()
    v の先頭の位置を戻す。
v.end()
    v の末尾の位置（最後の要素の1つ後）を戻す。
vector<T> v;
    T 型のオブジェクトを保持する空の vecotr を生成する。
v.push_back(e)
    e で初期化されたオブジェクトを v に付け加える。
v[i]
    インデックスが i である v の要素を戻す。
v.size()
    v の全要素の数を戻す。

```

その他のライブラリの仕組み：

```

sort(b, e)      [b, e) の範囲の要素を非降順にソートする。<algorithm>に定義されている。
max(e1, e2)    e1 と e2 の大きいほうを戻す。e1 と e2 は同じ型でないといけない。<algorithm>に定義されている。
while (cin >> x) 正しい入力を x に読み込みストリームの状態をチェックする。もし、ストリームがエラー状態になら条件は false となり、そうでない場合は条件が true となり while の中身が実行される。
s.precision(n) これ以後の出力の際のストリームの精度を決める（n を省略すると現在の精度を変更しない）。この関数はそれまでの精度を戻す。
setprecision(n) 出力ストリームに書き込まれると s.precision(n) と同じ効果をもつ。<iomanip>に定義されている。
streamsize     setprecision の引数の型であり precision の戻り値の型。<ios>に定義されている。

```

課題

- 3-0 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。
- 3-1 あるデータの集まりでメジアンを見つけたいとします。現在、いくつかのデータを読み込んでおり、この後にどれくらいのデータが残っているかわからないとします。この場合、すでに読み込んだデータは1つも破棄してはいけないことを示してください。ヒント：値を何か破棄した後で、その後に読み込まれたデータによって、破棄されたデータがメジアンになることがある…と示す方法があります。
- 3-2 整数の集合で、クオータイルを計算し出力するプログラムを書いてください。（ここでクオータイルとは、全データを大きい順に4つのグループに分けたものです。）
- 3-3 入力された単語について、それぞれの単語が何回登場したか調べるプログラムを書いてください。
- 3-4 入力された単語について、string オブジェクトとして、もっとも長いものともっとも短いものを報告するプログラムを書いてください。
- 3-5 複数の学生の成績を処理するプログラムを書いてください。このプログラムには2つの vector を使うよいかかもしれません。1つの vector は学生の名前を保持し、もう1つの vector は入力から最終成績を

計算したものを持続するのです。今のは、宿題の数は最初からわかっていると仮定してかまいません。
§ 4.1.3 では、学生の名前といくつかの成績を混ぜて扱う方法を説明します。