

<code>equal(b, e, d)</code>	<code>search(b, e, b2, e2)</code>
<code>find(b, e, t)</code>	<code>find_if(b, e, p)</code>
<code>copy(b, e, d)</code>	<code>remove_copy(b, e, d, t)</code>
<code>remove_copy_if(b, e, d, p)</code>	<code>remove(b, e, t)</code>
<code>transform(b, e, d, f)</code>	<code>partition(b, e, p)</code>
<code>accumulate(b, e, t)</code>	

- 8-3 § 4.1.4 で見たように、コンテナを戻す（渡す）のは、無駄が多いかもしれません。しかし、§ 8.1.1 で書いた median 関数は `vector` を引数に使っています。引数を `vector` ではなく反復子を使うものに書き換えられますか。もしそうすると、効率はどうなると思いますか。
- 8-4 § 8.2.5 で使った swap 関数の定義を書いてください。`*beg` と `*end` の値を交換せずに swap を使うのはなぜでしょう。ヒント：試してみてください。
- 8-5 第 7 章の `gen_sentence` と `xref` 関数で、出力方法を変え反復子を使うように、書き換えてください。これらの新しい関数を試すため、標準出力に対応する出力反復子を使ったり、`list<string>` や `vector<string>` に結果を格納するプログラムを書いてください。
- 8-6 `m` の型が `map<int, string>` とし、`copy(m.begin(), m.end(), back_inserter(x))` という関数呼び出しがあったとします。ここで `x` の型はどうあるべきでしょうか。逆に `copy(x.begin(), x.end(), back_inserter(m))` のときはどうでしょうか。
- 8-7 どうして、`max` 関数は 2 つの引数の型に対応する 2 つの型パラメータをもったテンプレート関数にしないのでしょうか。
- 8-8 § 8.2.6 の `binary_search` 関数で、`begin + (end - begin) / 2` の代わりにもっと簡単な `(begin + end) / 2` を使わないのはなぜでしょう。

第9章

新しい型を定義する

C++には 2 つの型があります。1 つは組み込み型でありもう 1 つはクラスです。組み込み型とは、言語の一部として定義されている型であり、`char`、`int`、`double` などがそうです。一方、`string`、`vector`、`istream` などライブラリで定義されている型はみなクラスです。入出力ライブラリにある、いくつかの低レベルでシステム依存のルーチンを除くと、ライブラリのクラスも一般のプログラマが自分のアプリケーションのために使える仕組みを使って書かれているものです。

組み込み型と同様に簡単に使える型をプログラマが自分で作り出せるということが、C++という言語のデザインにおいては特に重視されています。後で見ますが、ストレートで直感的なインターフェースを持ったクラスを作るためには、そのための感性や判断も必要ですが、言語にそのような機能がないとできません。ここでは第 4 章の成績処理問題を考えながら、クラスを定義するもっとも基本的な仕組みを紹介していきます。第 11 章からは、これらの基礎の上に、ライブラリのクラスのように完璧なクラスをどのように作るかを見ていきます。

9.1 Student_info をもう 1 度

§ 4.2.1 では、`Student_info` という単純なデータ構造といくつかの関数を書きました。これは学生の成績処理プログラムに使うためのものです。しかし、そこで作ったデータ構造と関数は必ずしも他のプログラマにも使い易いものではありません。

私たちが意識するしないに関わらず、私たちの関数を使いたいと考えるプログラマは、いくつかの規則に従う必要があります。たとえば、`Student_info` オブジェクトのユーザは、まずそのデータを読み込んでそのオブジェクトに与えるようにしました。これをしないければ、`vector` である `homework` が空のままになり、`midterm` と `final` は不定な値を持ってしまいます。これらの値をそのまま使うと結果は予想できないものになります。つまり、単純に結果が間違っているか、悪いときにはコンピュータがクラッシュしてしまいます。さらに、ユーザが `Student_info` オブジェクトが有効な値を保持しているかどうかチェックしたい場合は、実際のデータメンバーを調べるしかありません。これをするには、`Student_info` 内部の詳細な知識が必要になります。

関連する問題として、私たちのプログラムのユーザは「学生のデータをファイルから読み込んだらそのデータはその後は変更されない」と思いこむかもしれないということが考えられます。しかし、私たちのコードはそのような保証はしていません。

3 番目の問題は、もとの `Student_info` の「インターフェース」がばらばらになっているということです。たとえば、`read` のような `Student_info` の状態を変える関数を、1 つのヘッダーファイルにまとめることもできます。もしそうするならば、私たちのコードのユーザには便利でしょう。しかし、そのようなグループ作りをしないで

すませてしまうこともあるでしょう。

この章では、`Student_info` を拡張して、これらのすべての問題を解決します。

9.2 クラス型

もっとも基本的なレベルでは、クラスとは、関連するデータを1つにまとめておく機能ということができまます。これにより、複数のデータを、1つのものとして扱えるわけです。たとえば、§ 4.2.1 では `Student_info` を次のように定義しました。¹

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

これにより、`Student_info` のオブジェクトを扱うことができました。このオブジェクトは、4つのデータを内部に持つです。それは、`name` という名前の `std::string`、`homework` という名前の `std::vector<double>`、それから `midterm` と `final` という名前の `double` です。

あきらかに、この `Student_info` を使うプログラマは、ここでのデータを直接扱うことができるし、また、そうしなければなりません。このような直接的な操作ができるのは、データの要素へのアクセスがまったく制限されていませんからです。このようにしたのは、他に `Student_info` を操作するよい方法がなかったからです。

しかし、ユーザのデータアクセスを自由にするより、`Student_info` の内部構造を隠したいと考えるかもしれません。特に、データへのアクセスを関数を通して行うようにするとよいかもしれません。こうするためには、ユーザが `Student_info` を操作するのに便利なものをそろえる必要があるでしょう。これらが、この型（クラス）のインターフェースになるのです。

そのような関数を考える前に、なぜ `std::string` や `std::vector` のような完全修飾形を使い、`using` 宣言で修飾のない名前を使えるようにしないかを考え直してみましょう。`Student_info` を使いたい場所ではこのクラスの定義が必要なので、これをヘッダファイルに入れる考えます。§ 4.3 で指摘したように、他のプログラマに使われることが予想されるファイルでは、宣言の数は必要最小限にしておくべきなのです。あきらかに、`Student_info` はこれから使いたい名前なので定義しておく必要があります。しかし、その内部に `string` や `vector` があるのは、定義の内部仕様です。内部仕様のために、ユーザに `using` 宣言を強要することはできないのです。

プログラムの例として、また、よい習慣付けてしても、ヘッダファイルに入る名前は完全修飾形で使うことにします。ただ、ソースファイルのほうには、`using` 宣言があると仮定します。したがって、ヘッダファイルに使うつもりのないコードでは、修飾をつけない名前を使います。

9.2.1 メンバ関数

`Student_info` のデータへのアクセスをコントロールするために、プログラマが使えるインターフェースを定義する必要があります。まず、データを読み込む `read` 関数と最終成績を計算する `grade` 関数を定義してみま

¹ 訳注：`struct` で定義されるものは構造体とよばれます。構造体も（広い意味での）クラスです。これは 9.3 で説明されます。

しょう。

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

    std::istream& read(std::istream&); // 付け加えた
    double grade() const; // 付け加えた
};
```

このようにしても `Student_info` オブジェクトは4つのデータ要素を持つことに変わりはありません。そこに、さらに2つのメンバ関数を付け加えたのです。これらのメンバ関数は、`Student_info` オブジェクトに対し、入力ストリームからデータを読み込む `read` と最終成績を計算する `grade` です。`grade` の宣言の後についている `const` は `grade` を呼び出しても `Student_info` オブジェクトのデータは変わらないということを意味しています。

この本でメンバ関数の話が最初に出てきたのは、§ 1.2 で `string` クラスの `size` 関数を説明したときです。基本的に、メンバ関数はクラスオブジェクトに対するメンバです。メンバ関数を呼ぶときには、そのメンバ関数が所属するオブジェクトを指示しなければなりません。たとえば、`size` は `string` オブジェクトである `greeting` に対して `greeting.size()` という形で使いました。`s` を `Student_info` のオブジェクトとすると、メンバ関数は `s.read(cin)`、`s.grade()` という形でつかうようになります。ここで `s.read(cin)` は標準入力から適当なデータを読み込み `s` にセットするという意味です。`s.grade()` は `s` のデータから最終成績を計算して戻すという意味です。

最初のメンバ関数の定義は § 4.2.2 のもとの `read` によく似ています。

```
istream& Student_info::read(istream& in)
{
    in >> name >> midterm >> final;
    read_hw(in, homework);
    return in;
}
```

前と同様に、この関数の定義は共通のソースファイルに入れることにします。その名前はコンパイラによって、`Student_info.cpp`、`Student_info.C`、`Student_info.c` などとしました。ただし、これは重要なことなのですが、これらの関数の宣言は、`Student_info` の一部つまりヘッダファイル内にあるのです。これは、すべての `Student_info` クラスのユーザが、これらの関数を使えるようにするためにです。

新しいコードと前のコードの間には3つの重要な相違点があります。

1. 関数名が `read` ではなく `Student_info::read` になりました。
2. この関数は `Student_info` オブジェクトのメンバなので、`Student_info` オブジェクトを引数として渡さなくてよくなりました。そのため新たに `Student_info` オブジェクトを生成する必要もなくなりました。
3. 関数の定義内で、オブジェクトのデータ要素には直接アクセスできます。たとえば § 4.2.2 では `s.midterm` と書きましたが、ここでは単に `midterm` と書けばよいのです。

この違いを順番に説明しましょう。

関数名にある`::`は§ 0.7 から出てきていますが、標準ライブラリ内で定義されている名前を使うためにあるスコープ演算子と同じものです。たとえば、`string`のメンバである `size_type` という名前を `string::type_size` と書きました。同様に、`Student_info::read` で、「`Student_info`のメンバである `name` という名前の関数」ということになります。

このメンバ関数は `istream&` パラメータのみを持ちます。この理由は、前のバージョンで必要だった `Student_info&` は明示的でなくとも、いつもあると考えることができるからです。実は、`string` や `vector` のメンバ関数を呼び出すときには、どの `string` オブジェクトに対してか、どの `vector` オブジェクトに対してかを指定していました。たとえば、`string` オブジェクト `s` に対して `s.size()` を使うには、`s.size()` としました。逆に `string` オブジェクトを指定しないで `string` のメンバ関数を呼び出すことはできないのです。このように、新しい `read` 関数を呼び出すときも、読み込みをしている `Student_info` オブジェクトをはっきり示すことになります。この場合、そのオブジェクトを `read` 関数の中で使うことができます。

`read` の中で他のメンバを使うときには修飾を付けません。というのは、それらが現在操作しているオブジェクトのメンバであるからです。言葉を換えると次のようにも言えます。もし、`Student_info` オブジェクトである `s` に対して `s.read(cin)` という関数呼び出しをすれば、それは `s` の操作であるわけです。したがって、`read` が `midterm`, `final`, `homework` を使うとき、それは `s.midterm`, `s.final`, `s.homework` を意味するのです。

それでは、別のメンバ関数 `grade` 見てみましょう。

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

これは§ 4.2.2 のものに似ています。そしてその違いは `read` のときと同様です。`grade` は `Student_info` のメンバ関数として定義され、そのため、引数という形で `Student_info` オブジェクトを受け取ることなくそのメンバにアクセスでき、しかも他のメンバにアクセスするのに修飾を使わないのです。

ただ、上のコードは `read` に比べて、2 つの重要な違いを含んでいます。まず、`::grade` という関数呼び出しに注意してください。`::` を前に付けるのは、それによって、これはメンバではないことを示しているのです。今の場合、§ 4.1.2 で定義した `double` が 2 つと `vector` が 1 つの引数を取る同名の関数 `grade` を呼び出したいので、この`::` は必要なのです。もし、これを入れないと、コンパイラは `Student_info` 内の `grade` を再び `Student_info::grade` と解釈し、「引数を 3 つも持つことはできない」というエラーになってしまうでしょう。

もう 1 つの重要な違いは、`grade` のすぐあとに `const` を付けているところです。これは新しい関数と前の関数の宣言を見比べるとわかりやすいと思います。

```
double Student_info::grade() const { . . . }      // メンバ関数バージョン
double grade(const Student_info&) { . . . }        // § 4.2.2 のオリジナルバージョン
```

前のバージョンではパラメータリストの `Student_info` の参照に `const` を付けました。こうすることで、`grade` のパラメータが `const Student_info` オブジェクトになり、これを関数内で変更しようとコンパイラがエラーと判断してくれます。

メンバ関数を呼ぶときには、そのメンバ関数の属するオブジェクトを引数に渡すことはありません。そのため、`const` を付けるべきものが、パラメータリストには何もありません。その代わり、関数自身に修飾を付け、

9.3 データ保護

これを `const` メンバ関数（`const member function`）にするのです。`const` 付きのメンバ関数は、自身が処理しているオブジェクトの内部的な状態を変えることができません。こうすることで、たとえば `Student_info` オブジェクト `s` に対して `s.grade()` としても、これは `s` のデータメンバを変えないという保証をすることになるのです。

この関数はオブジェクトの値を変えないで、`const` オブジェクトに対して呼ぶこともできます。逆に、`const` オブジェクトに対して `const` でない関数を呼ぶことはできません。たとえば、`const` な `Student_info` オブジェクトに対して `read` を使うことはできないのです。実際、`read` はオブジェクトの内容を変えてしまうのです。そのような関数を `const` オブジェクトに対して使えば、`const` の宣言と矛盾してしまうわけです。

プログラムが実際に `const` オブジェクトを直接生成することがなくても、関数の引数として `const` オブジェクトへの参照が作られるかもしれないことに注意してください。`const` でないオブジェクトを `const` な参照引数を取る関数に渡した場合、この関数はそのオブジェクトを `const` であるように扱います。そのためコンパイラは、このオブジェクトに対し、`const` なメンバ関数の使用しか許可しないのです。

`const` キーワードはクラス定義の中の関数の宣言と関数の定義の両方に付けたことに注意してください。もちろん、引数のタイプは関数宣言と定義で同じでなければなりません。(§ 4.4)

9.2.2 メンバでない関数

新しいデザインでは、`read` と `grade` をメンバ関数にしました。`compare` はどうしましょうか。これはクラスのメンバ関数にするべきでしょうか。

§ 9.5, § 11.2.4, § 11.3.2, § 12.5, § 13.2.1 で説明しますが、C++ではある種の関数はメンバ関数にするべきです。しかし、`compare` はそうではありません。そのため、メンバ関数にしてもしなくてもよいのです。実は、ある一般的なルールがあります。それは、「もしオブジェクトの状態を変える関数なら、それはメンバ関数であるべき」というものです。しかし、このルールは、オブジェクトの状態を変えない関数については何も言っていません。したがって、自分で考えて決めなければならないのです。

のために、この関数が何をし、ユーザが何を期待するかを考える必要があります。`compare` 関数は `Student_info` である引数の `name` に基づいて、その順番を決めるものです。§ 12.2 で見ますが、このような比較の関数はクラスの外に定義しておいた方が便利なことがあります。そのため、`compare` はグローバル関数とし、その定義は後ですることにしましょう。

9.3 データ保護

`grade` と `read` をメンバにして問題の半分を解決しました。それは、`Student_info` のユーザがそのオブジェクトの内部を直接操作しなくてよいようにする、ということです。しかし、まだ、直接操作しようと思えばできるのです。ここでデータを隠蔽し、ユーザがデータを変更したいときは、メンバ関数を使うしかないようにしたいと思います。

C++では、メンバを `public` と指定することでユーザの使用を可能にし、`private` と指定することで使用を不可にすることができます。これを使うと `Student_info` は次のようになります。

```
class Student_info {
public:
```

9.3 データ保護

```
struct Student_info {
private:
    std::string name;
    // 他のprivateメンバ
public:
    double grade() const;
    // 他のpublicメンバ
};
```

と同じなのです。どの例でも、`Student_info` オブジェクトのメンバ関数はユーザが自由に使え、データメンバは使えないようにしてあります。

つまり、`struct` でも `class` でも本質的な違いはないのです。実際、ユーザはコードの詳細を見なければ、どちらが使われているか区別がつかないはずです。`struct` と `class` を使い分ける意味は、一種の文書化なのです。²一般に、内容を公開するつもりの小さなデータ構造なら `struct` を使います。そのため、第4章のオリジナルの `Student_info` は `struct` としました。ここでは、メンバ関数のアクセスを制限したいので、`Student_info` の定義に `class` を使ったのです。

9.3.1 アクセス関数

ここまでで、データメンバを隠蔽し、`Student_info` のデータをユーザが変更できないようにしました。ユーザは、データを直接操作するのではなく、`read` 関数でデータをセットし `grade` 関数でそのオブジェクトが表す学生の最終成績を取り出します。ここで、もう1つ、付け加えるべき操作があります。それは、ユーザが学生の名前を取り出せるようにしなければならないということです。たとえば、§ 4.5 のようなプログラムを考えてください。そこでは学生の成績を整形して出力しました。このような報告を出力するために、学生の名前を取り出す必要があるのです。ただ、ここで必要なのは学生の名前を読み出すだけで、それを変えたいのではありません。これはストレートにできます。

```
class Student_info {
public:
    double grade() const;
    std::istream& read(std::istream&);          // 定義を変える必要あり
    std::string name() const { return n; }         // 付け加えた
private:
    std::string n;                                // 変更した
    double midterm, final;
    std::vector<double> homework;
};
```

ユーザが直接 `name` にアクセスできるようにするのではなく、`name` という名前の関数を付けました。これは対応するデータメンバを（読み込み専用）取り出す関数です。ただし、関数の名前を `name` にしたので、混乱の無いようデータメンバの名前を `n` にしました。

`name` 関数は、引数を取らず `string` である `n` を戻すだけの `const` なメンバ関数です。`n` の参照ではなくコピーを戻すことで、ユーザは `n` の値を読めてもそれを変更することはできないようにしてあるのです。また、こ

*2 訳注：「コードを読む人に、コードの意図を明示する」という意味です。

```
// インタフェースはここに
double grade() const;
std::istream& read(std::istream&);

private:
    // 実装はここに
    std::string name;
    double midterm, final;
    std::vector<double> homework;
};
```

ここで `Student_info` には少し変更を加えました。まず、`struct` の代わりに `class` という言葉を使い、2つの保護ラベル（protection label）を付けたのです。保護ラベルは、その下のメンバのアクセス制限を指示するものです。これはクラス内でどのような順番で使ってもよいし、何度も使ってもかまいません。`name`、`homework`、`midterm`、`final` を `private` ラベルの下に置くことで、これらに `Student_info` のユーザ（このクラスを使うプログラマ）がアクセスできないようにしたのです。こうしたのでメンバ関数以外の関数は、これらを使うことはできなくなりました。使おうとすると、コンパイラが、これらはプライベートであるので、これらを使うことはできません。一方、`public` の下にあるメンバはどこからでもアクセスできないというエラーメッセージを出すはずです。一方、`public` の下にあるメンバはどこからでもアクセスできます。つまり、`read` と `grade` は誰でも使えるわけです。

struct の代わりに `class` を使ったのはなぜでしょう。実は、新しい型を定義するのに、どちらも使えるのです。`struct` と `class` のたった1つの違いは、ラベルが無い場合の、つまりデフォルトのアクセス制限です。最初の{}から最初の保護ラベルまでの間に置かれたメンバは自動で `private` になります。逆に `struct` `Student_info` と書けば、最初の{}と最初の保護ラベルまでの間に置かれたメンバは `public` になります。たとえば、

```
class Student_info {
public:
    double grade() const;
    // などなど
};

は、

struct Student_info {
    double grade() const; // デフォルトでpublic
    // などなど
};
```

と同じ意味になります。また、

```
class Student_info {
    std::string name;           // デフォルトでprivate
    // 他のprivateメンバ
public:
    double grade() const;
    // 他のpublicメンバ
};
```

は、

の関数は読み込みアクセスだけなので、`const` 宣言をしました。

`grade` と `read` の定義は、クラス定義の外に書きました。`name` のようにメンバ関数をクラス定義の一部として定義すると、コンパイラに「可能ならば関数呼び出しのオーバーヘッドを避けてインライン展開（§ 4.6）するよう」と指示することになります。

`name` のような関数はしばしばアクセス関数（accessor function）と呼ばれます。この関数は実際にはデータ構造の一部にのみアクセスを許可する関数なので、この名前はやや誤解を生むかもしれません。実際、歴史的にはデータへのアクセスを許し今まで作り上げてきた情報のカプセル化を破壊する関数がしばしば使われてきました。しかし、ここでいうアクセス関数は、クラスの抽象インターフェースの一部なのです。しばしありました。

しかし、ここでいうアクセス関数は、クラスの抽象インターフェースの一部なのです。

`Student_info` の場合の抽象とは、学生とその学生の成績の表現を意味します。学生の名前の表現も抽象の一部

であり、それが `name` 関数なのです。一方、`midterm`、`final`、`homework` にはアクセス関数を付けません。これらは、内部の詳細であり、インターフェースの一部ではないのです。

`name` 関数を付け加えたので、`compare` 関数を書くことができます。

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

この関数は § 4.2.2 のものによく似ています。唯一の違いは、学生の名前をどのように取り出しているかだけです。オリジナル版では、データメンバを直接使っていました。ここでは、学生の名前を戻す `name` 関数を使っています。`compare` もインターフェースの一部なので、この関数の宣言も `Student_info` を定義するヘッダファイルで宣言しておかなければなりません。また、その定義は `Student_info` のメンバ関数の定義があるソースファイルに書きます。

9.3.2 空でないか確かめる

メンバを隠蔽し、適切なアクセス関数を定義しましたが、まだ 1 つ問題が残っています。まだ、ユーザがオブジェクトのデータを直接チェックしたいと考える場合があるのです。たとえば、次に示すように `read` を呼ぶ前

のオブジェクトに対して `grade` を使ったら何が起こるでしょう。

```
Student_info s;
cout << s.grade() << endl; // 例外:sにはまだデータがない
```

`s` にデータを入れるために `read` を呼んでいないので、`s` の `homework` は空です。そこに `grade` を呼べば例外を呼び出しが止まっています。もちろん、ユーザが例外を `catch` することもできますが、問題の発生を事前に知り、関数投げてしまいます。

第 4 章の `Student_info` 構造体を使っていれば、ユーザは `grade` を呼んでよいかどうか確認することができます。それは `homework` が空かどうかを直接確認したのです。この方法は有効ですが、これを実行するためにはユーザにデータ構造の詳細を見せることになります。実は、もっと抽象的なチェック関数を作ればよいのです。

```
class Student_info {
public:
```

```
bool valid() const { return !homework.empty(); }
// 以前のものと同じ
};
```

`valid` 関数はオブジェクトに有効な（valid）データがあるかどうかを報告する関数で、宿題が少なくとも 1 つ以上記録されていれば `true` を戻すようになっています。その場合には、成績を計算することができるからです。たとえば、`grade` を呼ぶ前に、ユーザはオブジェクトが有効かどうかを `valid` で調べ、例外の発生を避けることができるわけです。

9.4 Student_info クラス

ここまでで、オリジナルの `Student_info` 構造体の問題の大部分を解決しました。ここすべてをまとめて書き出しておきましょう。

```
class Student_info {
public:
    std::string name() const { return n; }
    bool valid() const { return !homework.empty(); }
    // § 9.2.1 で定義したように nameではなくnに読み込む
    std::istream& read(std::istream&);
    double grade() const; // § 9.2.1 で定義した
private:
    std::string n;
    double midterm, final;
    std::vector<double> homework;
};
```

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

ユーザは `Student_info` オブジェクトの状態を、`read` を使ってのみ変更できます。オブジェクトの中に入り、直接データメンバを変更することはできません。また、ユーザはクラスの内部の詳細を知る必要もなくなりました。詳細を隠す代わりにいくつかの関数を作ったからです。`Student_info` の関数をまとめるのは理にかなっています。

9.5 コンストラクタ

前節のクラスはほぼ完成していて、見ての通り有用ですが、もう 1 つ考えるべきことがあります。オブジェクトが生成されるときにどうするか何も決めていないのです。

ライブラリにあるクラスでは、オブジェクトの生成時に適当な値が入れられることを学びました。たとえば、`string` や `vector` を初期値を与えずに生成すると、空の `string` や `vector` になります。一方、中に入れる文字やサイズを指定したり適当な初期値を持つように `string` や `vector` を生成することもできます。

オブジェクトの初期化を定義するのは、コンストラクタ（constructor）という特別なメンバ関数です。コンストラクタを明示的に呼び出す方法はありません。その代わりに、クラスのオブジェクトが生成されるときに、副作用として適当なコンストラクタが呼ばれるようになっています。

コンストラクタを定義しないと、コンパイラが自動で 1 つ作ってくれます。このコンストラクタについては

§ 11.3.5 で詳しく説明しましょう。ここでは、コンストラクタを定義しないとどうなるかだけ、知っておく必要があるのです。その場合でも、ユーザは `Student_info` オブジェクトを生成できるのですが、他のオブジェクトをコピーする以外では、明示的に初期化することはできないのです。

コンパイラが作ったコンストラクタはオブジェクトの作られ方にしたがってデータメンバを初期化します。オブジェクトをローカル変数として生成するなら、データメンバはデフォルトで初期化されます（§ 3.1）。もし、オブジェクトがコンテナの要素のために作られるなら、つまり、新しい要素を `map` に付け加えるのに使われたオブジェクトがコンテナの要素として使われるなら、値初期化（§ 7.2）されます。これらのルールは少し複雑ですが、本質は下のようにまとめられます。

- 1 つかそれ以上のコンストラクタを持ったクラスなら、そのクラスのオブジェクトの初期化は適当なコンストラクタによって完全にコントロールされる。
- 組み込み型なら、値初期化は 0 に初期化し、デフォルト初期化は不定値に初期化します。
- 上の場合にあてはまらない場合、コンストラクタを持たないクラスということになる。その場合の値初期化、デフォルト初期化は、データメンバそれぞれの値初期化、デフォルト初期化になる。データメンバがコンストラクタを持つクラスのオブジェクトであった場合、上に書いたように初期化される。

あきらかに、`Student_info` は上の 3 番目にあたります。つまり、コンストラクタで初期化を指定していないクラスなのです。そのため、`Student_info` をローカル変数として初期化すると、`n` と `homework` はコンストラクタを持つクラスのオブジェクトなので、そのコンストラクタにしたがって空の `string` と `vector` になります。`midterm` と `final` はデフォルト初期化で不定値を持つようになります。これらの変数が生成されるとき一方、`midterm` と `final` は空にし、`midterm` と `final` は 0 にします。これは次のように書かれます。

今考へているような簡単な例では、このようにしておいても問題はないかもしれません。プログラム中の操作で、`midterm` や `final` を `read` で初期化して適当な値を与える前に使うことはないからです。しかし、普通は、すべてのデータメンバが常に意味のある値を持つようにしておいた方がよいでしょう。たとえば、後で（自分の後任者かもしれません）、データメンバを使うような操作を付け加えるかもしれないからです。コンストラクタでデータメンバを初期化しないと、そのような操作がいずれエラーを引き起こすかもしれないのです。また、`§ 11.3.5` で見ますが、`midterm` や `final` を明示的に使わなくても、複合的な操作で非明示的に使うことがあります。不定値には、上書きする以外のことをしてはいけません（§ 3.1）。そのため、厳密に言えかもしれないのです。不定値には、上書きする以外のことをしてはいけません（§ 3.1）。そのため、厳密に言えば、データメンバは初期化しなければならないのです。

実際には、2 つのコンストラクタを付けたいのです。1 つは引数を取らずに空の `Student_info` オブジェクトを生成するもので、もう 1 つは入力ストリームの参照を引数にし、ストリームから学生のデータを読み込んでオブジェクトを初期化するものです。このようなコンストラクタを作ると、次のようなコードが書けるようになります。

```
Student_info s;           // 空のStudent_infoオブジェクト
Student_info s2(cin);    // cinからデータを読み込んでs2を初期化
```

コンストラクタは他のメンバ関数と違う 2 つの特徴を持っています。名前がクラスの名前と同じだということと、戻り値の型がないということです。逆に、コンストラクタは他のメンバ関数と同じように、引数の数や型の異なるものを複数定義してよいという性質を持っています。この知識をもとに、`Student_info` に 2 つのコンストラクタを付け加えます。

```
class Student_info {
public:
    Student_info();           // 空のStudent_infoオブジェクトを生成
    Student_info(std::istream&); // ストリームからデータを読み込んで生成
    // 前と同じ
};
```

9.5.1 デフォルトコンストラクタ

引数を取らないコンストラクタはデフォルトコンストラクタ（default constructor）とよばれます。その仕事は、普通、オブジェクトのデータを適当に初期化することです。`Student_info` オブジェクトの場合は、まだ学生のデータを読み込んでいないことがわかるように初期化するのがよいでしょう。`vector` である `homework` と `string` である `n` は空にし、`midterm` と `final` は 0 にします。これは次のように書かれます。

```
Student_info::Student_info(): midterm(0), final(0) {}
```

このコンストラクタの定義には新しい記法を使いました。:と { } の間にあるものは、コンストラクタ初期化子（constructor initializer）とよばれるものです。これは、データメンバの値を丸カッコの中のもので初期化するようにと、コンパイラに指示するものです。そのため、今の場合、このコンストラクタのすることは、`midterm` と `final` を 0 にすることです。それ以外に、目に見える仕事はしていません。実際、関数の中身は空です。また、後で見ますが、`homework` と `n` は非明示的に初期化されているのです。

オブジェクトの生成と初期化を理解するためには、コンストラクタ初期化子を理解しなければなりません。クラスオブジェクトを新しく生成するときには、次のようなステップが踏まれるのです。

1. オブジェクトを保持するため、メモリが確保され（割付られる）。
2. コンストラクタ初期化子の指示にしたがって、オブジェクトが初期化される。
3. コンストラクタの中身（{ }の中身）が実行される。

オブジェクトのすべてのデータメンバは、対応するコンストラクタ初期化子があってもなくても、すべて初期化されます。そしてまた、それをコンストラクタの中身ですぐに書き換えることもできます。しかし、初期化子による初期化の方がコンストラクタの中身の実行よりも前に行われることは覚えておいてください。一般には、初期化子で値を初期化する方が、中カッコの中で値を代入するよりよいでしょう。中カッコの中で値を代入すると、初期化と代入で、二度手間になるからです。

コンストラクタを書くことで、生成されたオブジェクトのデータは意味のある状態になると言いました。一般的にいうと、このようなデザインでは、すべてのコンストラクタがすべてのデータを初期化すべきです。これは、データメンバが組み込み型の場合、特に言えることです。コンストラクタが初期化をしないと、ローカルなスコープで定義されたオブジェクトのデータメンバには、ゴミが入っています。もちろん、これはほとんど誤った値です。

ここで、なぜ `Student_info` のデフォルトコンストラクタが、他に目に見える仕事をしないかが説明できます。明示的に初期化したのは `midterm` と `final` だけですが、他のメンバは非明示的に初期化されるのです。特に、`n` は `string` のデフォルトコンストラクタにより、また `homework` は `vector` のデフォルトコンストラクタによって初期化されるのです。

9.5.2 引数を取るコンストラクタ

`Student_info` のもう 1 つのコンストラクタはもっと簡単です。

```
Student_info::Student_info(istream& is) { read(is); }
```

これは、初期化の仕事を `read` 関数に任せているのです。このコンストラクタには初期化子がありませんので、`homework` と `n` はそれぞれ `vector` と `string` のデフォルトコンストラクタで初期化されます。`midterm` と `final` は意図的に初期化しないと、意味のある値を持ちませんが、すぐに `read` でデータを与えるので問題はありません。

9.6 Student_info クラスを使う

完成した `Student_info` クラスは、もともとの第 4 章で定義した `Student_info` 構造体とはかなり異なるものになりました。当然、その使い方も異なっています。データメンバを `private` にすることで、ユーザはオブジェクトのデータを直接変更できなくなりました。その代わり、ユーザにはクラスに付けたインターフェースを使ってプログラムしてほしいのです。たとえば、§ 4.5 の `main` プログラムを次のように書きなおすことができます。これは、この章のクラスを使って、すべての学生の最終成績を整形した形で出力するプログラムです。

```
int main()
{
    vector<Student_info> students;
    Student_info record;
    string::size_type maxlen = 0;
    // データを読み込み、保持する
    while (record.read(cin)) { // 変更した
        maxlen = max(maxlen, record.name().size()); // 変更した
        students.push_back(record);
    }
    // 学生のデータをアルファベット順に並べる
    sort(students.begin(), students.end(), compare);
    // 名前と成績を出力する
    for (vector<Student_info>::size_type i = 0;
         i != students.size(); ++i) {
        cout << students[i].name() // 変更した
            << string(maxlen + 1 - students[i].name.size(), ' ');
        try {
            double final_grade = students[i].grade(); // 変更した
            streamsize prec = cout.precision();
            cout << setprecision(3) << final_grade
                << setprecision(prec) << endl;
        } catch (domain_error e) {
            cout << e.what() << endl;
        }
    }
    return 0;
}
```

9.7 詳細

ここで変更した場所は、`name`、`read`、`grade` 関数の呼び出し部分です。たとえば、今

```
while (record.read(cin)) {
```

となっているループは、もともと、

```
while (read(cin, record)) {
```

となっていたものです。新しいバージョンの `read` はオブジェクト `record` に対して呼ばれているメンバ関数なのです。もとのバージョンの `read` はグローバル関数であり、`record` は引数として渡されていました。もとのバージョンも新しいバージョンも同じ事をしているだけです。これにより `cin` から読み込まれたデータが `record` のデータメンバに代入されるのです。

9.7 詳細

ユーザ定義型： `struct` と `class` で定義できる。これらの違いは、保護ラベルの前に置かれたメンバのデフォルトのアクセス制限だけである。`struct` の場合、そのメンバは `public` として扱われ、`class` の場合は `private` として扱われる。

保護ラベル： クラスメンバのアクセス制限を指示する。`public` メンバは一般にどこからでもアクセスでき、`private` メンバはクラスのメンバからのみアクセスできる。保護ラベルは任意の順番で複数回使うことができる。

メンバ関数： ユーザの定義型では、データ同様、関数をメンバにできる。メンバ関数は、個々のオブジェクトに対して呼ばれる。メンバ関数の中でデータメンバや他のメンバ関数を使おうとすると、それらは同一のオブジェクトのメンバが使われることになる。

メンバ関数の定義は、クラスの定義の内にも外にも書くことができる。クラス定義の中に書く場合は、コンパイラにその関数をオンライン展開し呼び出しのオーバーヘッドを無くすように指示したことになる。クラス定義の外に書く場合は、その関数がクラスのスコープにあることを示すように修飾しなければならない。メンバ関数の名前は、`class-name::member-name` という形で、クラス `class-name` からメンバ `member-name` を参照する形になる。

メンバ関数はパラメータリストの後に `const` を付けることで、`const` 関数にすることができる。このようなメンバ関数は、そのオブジェクトの状態を変えることができない。`const` オブジェクトに対しては `const` 関数しか呼べない。

コンストラクタ： その型のオブジェクトがどのように初期化されるかを定義する特別なメンバ関数。コンストラクタはクラスと同じ名前を持ち、戻り値はない。パラメータリストでパラメータの数や型が違うものなら、複数個のコンストラクタを作ることができる。コンストラクタの呼び出し終了後にはすべてのデータメンバに意味のある値が入っているようにするとよい。

コンストラクタ初期化子リスト： コンストラクタ初期化子は、`member-name(value)` という形をしていて、コンマで分けて並べられる。このようすると、そのメンバ `member-name` がその値 `value` で初期化される。明示的に初期化されないデータメンバは非明示的に初期化される。