

A	90 - 100
B	80 - 89.99 ...
C	70 - 79.99 ...
D	60 - 69.99 ...
F	< 60

そして、出力は A が何人、B が何人... となるようにしてください。

7-3 § 7.3 の対照表プログラムでは、1 つの単語が同じ行に複数回現われたなら、プログラムはその行番号をそのまま出力します。これを、同じ単語が同じ行に複数回現われても、その行は 1 回しか出力しないようなプログラムに書き換えてください。

7-4 対照表のプログラムは入力ファイルが大きくなると、その出力は見苦しくなります。出力が長すぎる場合は、改行が入るようにプログラムを書き直してください。

7-5 文法による文の構成プログラムで、文を構成するデータ構造を list にして書き直してください。1 つは、構成した文を

7-6 文法による文の構成プログラムで、vector を 2 個使うように書き直してください。1 つは「ルール」を保持しスタックとして使うものです。これにより再帰をなくすの

です。

7-7 対照表のプログラムの main を書き換え、文が 1 つのときは line を出力し、文が複数あるときは lines を出力するようにしてください。

7-8 対照表のプログラムを、入力ファイル中のすべての URL とそれが現われた行を書き出すプログラムに変えてください。

7-9 (難) § 7.4.4 の nrand は引数が RAND_MAX を超えると動作しません。一般には、RAND_MAX は可能な整数の最大値と一致しているので、これは問題になりません。しかし、RAND_MAX が可能な整数の最大値よりずっと小さいような環境もあります。たとえば、RAND_MAX が $32767(2^{15} - 1)$ で、整数の最大値が $2147483647(2^{31} - 1)$ 。n がどんな値でもちゃんと動作するように nrand を書き換えてください。

第8章

ジェネリック関数を書く

ここまで、主に、C++言語の基本的な仕組みと標準ライブラリの提供する抽象^{*1}を使って、具体的な問題を解くプログラムを書いてきました。この章からは、どのようにして自分自身で「抽象」を書いていかを考えることにします。

これらの抽象にはいくつかの形式があります。この章では、ジェネリック関数というものを考えます。これは、関数の実行時までそのパラメータの型を正確に決めなくてもよいというものです。第 9 章から 12 章までは抽象データ型の実装について説明します。第 13 章からになりますが、最後に、オブジェクト指向プログラミング (Object-Oriented Programming, OOP) について、説明しましょう。

8.1 ジェネリック関数とは何か

ここまで書いてきた関数は、そのパラメータと戻り値の型がはっきりわかっていました。これらの型は関数の定義に不可欠な部分に見えたかもしれません。しかし、実は、パラメータの型と戻り値の型がはっきり決まっていない関数をすでに（書いてはいませんが）使っているのです。

たとえば、§ 6.1.3 では、ライブラリの関数 find を使いました。これは 2 つの反復子と 1 つの値を引数に取りました。どのコンテナに対しても適当な型の値を探すのに、この find 関数が使えるのです。これはつまり、find の引数や戻り値の型を使うときに決めているということです。このような関数がジェネリック関数 (generic function) と呼ばれるのです。ジェネリック関数を作り出し使うことができるということは、C++の重要な特徴です。

ジェネリック関数を作り出す C++ の仕組みを理解することは難しくありません。難しいのは、「find は適当な型ならなんでも引数に使える」という意味を正確に理解することです。たとえば、find を使いたいと思う人に、特別な型の値について find の振る舞いをうまく説明するにはどうしたらよいでしょうか。この質問の答えのある部分は C++ 言語の中にあり、他の部分はその外にあります。

言語の中にある部分の答えとは、このような関数が引数をどのように使うかには制限があるということです。たとえば、パラメータに x と y を持っていて $x+y$ を計算する関数があったとします。このような関数が動作するための最低限の条件は、x と y が $x+y$ という演算を許す型であるということです。このような関数を呼び出したときに、与えられた引数で関数内の処理を行ってよいのかどうかは言語が決めているのです。

C++ 言語の外にある部分は、このような関数の引数について、標準ライブラリ内の仕組みが制限を置いているということです。すでに紹介した反復子は、そのようなライブラリ内の仕組みの 1 例です。あるものは反復子で

*1 訳注：ここでは「内部の詳細を気にせず使える便利なツール」という意味です。

あり、そうでないものもあります。そして、`find`という関数の前の2つの引数は、反復子でないといけないのです。

ある型が反復子だという場合、実際には、その型がサポートする操作のことを問題にしているのです。つまり、ある型が特定の操作をサポートするとき、そしてそのときのみ、その型は反復子であると言えるのです。もし、`find`の定義を自分で書くことになれば、すべての反復子がサポートする操作だけを使って書かなければならぬのです。第11章で実際に見ますが、自分でコンテナを書くことになれば、すべての適当な操作をサポートする反復子も作らなければならないのです。

反復子そのものはC++言語の一部ではありません。しかし、これは標準ライブラリの基本的な一部なのです。そして、これがあるからこそ、ジェネリック関数は有用なのです。この章では、ジェネリック関数が標準ライブラリでどのように実装されているかを例として見ます。それとともに、反復子とは何かを説明します。実は、反復子には5つの種類があります。

この章はこれまでに比べてかなり抽象的になります。それはジェネリック関数はもともと抽象的だからです。関数を特別な仕事のために書けば、それはもうジェネリックではないのです。この章で扱う関数の大部分は、これまでの例の中で使ってきたので、むしろなじみがあると感じるはずです。また、なじみのないものでも、その使い方を想像するのは難しくないはずでしょう。

8.1.1 知らない型のメジアン

ジェネリック関数を書くためのC++の仕組みはテンプレート関数(template function)といわれるものです。テンプレートは多くの似たような関数や型を何度も書かずに一度で済ませてしまうのです。ここで言う似たような関数とは、同じように振る舞うけれど扱う型だけが違うのです。この型の違いはテンプレートパラメータ(template parameter)というもので吸収します。この章ではテンプレート関数を扱い、第11章ではテンプレートクラスを扱います。

テンプレートを使う基本的な理由は、異なる型のオブジェクトであっても同じように振る舞うものがあるということです。テンプレートパラメータを使うと、そのパラメータに入る型を實際にはまだ知らないでも、共通した振る舞いからコードを書くことができるのです。もちろん、テンプレートを使うときにはそこで使う型はわかるので、プログラムのコンパイル・リンクができます。つまり、ジェネリックパラメータについて、実行中にいろいろな型を使うことになると、コンパイラが悩むことはないのです。コンパイラはコンパイルの時に仕事をするだけです。

テンプレートは標準ライブラリの重要な要素ですが、自分で使うこともできます。たとえば、§ 4.5では`vector<double>`のメジアンを計算するプログラムを書きました。この関数の内部では、`vector`を整列させる`sort`関数を使い、それから適切なインデックスの要素を取り出しました。このように作った関数を任意の値のシーケンスに対して使うことはできません。しかし、この関数を`vector<double>`にだけ適用する理由もないのです。他の型を要素に持つ`vector`に対してもメジアンを見つける関数が書けるのです。それがテンプレート関数です。これは次のように書きます。

```
template<class T>
T median(vector<T> v)
{
    typedef typename vector<T>::size_type vec_sz;
    vec_sz size = v.size();
```

8.1 ジェネリック関数とは何か

```
if (size == 0)
    throw domain_error("median of an empty vector");
sort(v.begin(), v.end());
vec_sz mid = size/2;
return size % 2 == 0 ? (v[mid] + v[mid-1]) / 2 : v[mid];
}
```

ここで見覚えがないのはテンプレートヘッダとよばれるものでしょう。

```
template<class T>
```

それと、このTをパラメータリストと戻り値の型に使っていることです。テンプレートヘッダは、コンパイラにこれからテンプレート関数を定義することを示しているのです。そして、こうすると、この関数は型パラメータ(type parameter)というものを持つようになります。型パラメータは関数におけるパラメータに似たもので、この関数のスコープ内で使われる名前を定義するのです。ただし、型パラメータは変数ではなく型を示します。この関数の中でTという文字を使うと、コンパイラはこのTが何らかの型を表すと判断するのです。`median`関数の中では、vという名前の`vector`の保持する値の型、また、関数の戻り値の型として型パラメータTを使っています。

この`median`関数を使うと、コンパイラがコンパイル時にTを特定の型として扱うことになります。たとえば、`vector<int>`であるvのメジアンを考えたいなら、この関数は`median(v)`という形で使われます。使われる形から、コンパイラはTをintと判断するのです。そして、関数内にTという文字があると、コンパイラはこれをintとしてバイナリを生成するのです。つまり、コンパイラは、`median`のコードを`vector<int>`に對してint型の値を戻すように、具体化(instantiates)するのです。この具体化については少し先で説明を加えます。

次に見覚えがないのは、`vec_sz`の定義の中にある`typename`というものでしょう。これは、コンパイラに`vector<T>::size_type`は型の名前であることを示しているのです。これは、この時点でのコンパイラがTを具体的に知らないでもかまわぬようにするためにです。たとえば、`vector<T>`と書くと、この定義はTの定義によって変わってきます。そして、`size_type`のようなメンバを使いたいときには、この名前の前に`typename`を書く必要があるのです。これを書くことで、コンパイラはこれを型の名前として扱うことになるのです。実は、標準ライブラリではTが何であれ、`vector<T>::size_type`は特定の型名のですが、コンパイラは標準ライブラリとは独立していて、これが型名だと判断することはできないのです。

型パラメータは、たとえ他の型との依存関係が明らかでなくても、関数の定義内で使われます。`median`関数では、型パラメータは戻り値とパラメータリスト、`vec_sz`の定義ではっきり使っています。しかし、vは`vector<T>`という型の変数であり、vに関わる操作はすべて、Tを非明示的に使っているのです。たとえば、

```
(v[mid] + v[mid-1]) / 2
```

では、v[mid]、v[mid-1]の型、つまりvの要素の型を知らなければなりません。これらの型が決まって、+や-/の意味も定ります。そこで、もし、`median`を`vector<int>`型の引数で使えば、+や-/はintの演算と解釈され、戻り値もint型になります。`median`を`vector<double>`に使えば、同様の計算がdoubleに対して実行されます。しかし、`vector<string>`に対して`median`を使うことはできません。`median`は割り算/を使っているのに、stringにはそのような演算がないからです。これは、私たちの予想通りではないでしょうか。「`vector<string>`のメジアン」などというものに意味はないのですから。

8.1.2 テンプレートの具体化

`median` を `vector<int>` 型引数を使った場合、コンパイラはこの関数の `T` を `int` に置き換えて具体化したものの（インスタンス、instance）を生成すると書きました。もし、`median` を `vector<double>` 型引数を使った場合は、コンパイラは同様に、この型から `T` を `double` と判断し、`T` を `double` に置き換えた別バージョンの `median` を生成するのです。

C++の標準は、テンプレートの具体化をコンパイラがどのように行うかを規定していません。したがって、
の詳細はコンパイラ次第なのです。みなさんのコンパイラがこれをどのように扱うかをここで述べることはでき
ませんが、2つの重要なポイントは覚えておいてください。それは、「プログラムの入力ーコンパイルーリンク」
という伝統的なスタイルに従うコンパイラでは、しばしば具体化がコンパイル時ではなくリンク時に行われ
ます。テンプレートが具体化される前に、コンパイラは特定の型でテンプレートが使用可能か判断するのです。
そのため、コンパイル時のエラーと思えるものがリンク時に現われることもあるのです。

ため、コンパイラが
テンプレートを書くときのもう1つのポイントは、現在の多くのコンパイラは、テンプレートを具体化するに
ために、単にその宣言ではなく、その定義があることを要求していることです。これは、一般的にいうと、テンプ
レートのヘッダファイルだけでなく、ソースファイルがアクセス可能でないといけないということです。この
ソースファイルをどこに置くかはコンパイラによって異なります。しかし、多くのコンパイラはヘッダファイル
ソースファイルを直接(#includeという形ではなく)含まれていることを要求しています。この問題に関しては、
みなさんのコンパイラのマニュアルを見てみるのが一番確かです。

8.1.3 ジェネリック関数と型

§ 8.1 でテンプレートの設計と使用の難しいところは、テンプレートとテンプレートで使用できる「適当な空」の関係を理解するところだと書きました。`median` のテンプレート版の定義では、明らかな型依存性を見ました。それは `median` に引数として渡される `vector` に保持される要素は、足し算と割り算ができるなければならないということです。幸い、多くの割り算を持つ型は算術的な型で、このような型依存性が実際に問題を起こすことはあまりありません。

もっと微妙な問題は、テンプレートと型変換の間で起こります。たとえば、`find` 関数を使って、学生が相槌を全部提出したかどうかを調べたいとします。ここで以下のように書いたとしましょう。

```
find(s.homework.begin(), s.homework.end(), 0)
```

今、homework は `vector<double>` ですが、`find` では `int` 型を調べています。このような型の小釣合いは、実際にには何の問題も起こしません。`int` 型と `double` 型は、情報を失うことなく比較できるからです。

しかし accumulate 関数は次のように使いました。

```
    auto vbegin() { return v.end(); } v.end() = 0; }
```

ここで最後の引数を `int` ではなく `double` の `0.0` にしたことは、正しい答えを得るために重要でした。というのは、`accumulate` 関数は、最後の引数の型を使って合計の計算をするからです。もし、`double` 型の値の合計を求めたいのに、ここに `int` 型の値を入れれば、合計は小数部を切り捨てて整数部分だけで行われるでしょう。コンパイラは `int` で計算するわけですが、その合計は不正確なものになってしまうわけです。

8.2 データ構造非依存性

最後に、`max` を呼ぶ場合を思い出してください。

```
string::size_type maxlen = 0;  
maxlen = max(maxlen, name.size());
```

ここで maxlen の型が name.size() の型と正確に一致していることが重要だと言いました。もしこの型が一致していないければ、これはコンパイルされないので。テンプレートパラメータの実際の型は与えられた引数の型で決められます。このことを考えれば理由もわかります。max 関数の定義は次のようなものでしょう。

```
template<class T>
T max(const T& left, const T& right)
{
    return left > right ? left : right
}
```

ここで、int と double の型を引数に使ったとすると、コンパイラはどちらの型をどちらの型に変換するべきか判断できないのです。つまり、double の変数の型を int に変換し両方の型を int に統一して処理を進めるべきか、int の変数の型を double に変換し両方の型を double に統一して処理を進めるべきかわからないのです。そのため、コンパイラはこれをエラーとするのです。

8.2 データ構造非依存性

この章で考えている median 関数は、vector の要素で許される型をいろいろなものにするために、テンプレートを使って書きました。これにより、算術的であればどの型の値を保持していても、その vector のメアンを求められるようになりました。

さらに一般的に、値が `list`、`vector`、`string` など、どのようなデータ構造に対しても使える関数にしたいと思います。この場合、コンテナ全体を操作するのではなく、コンテナの一部のデータだけを処理しようと考えます。

たとえば、標準ライブラリの関数 `find` では、反復子で指定して、コンテナの連続した一部分のみを調べることができます。`c` をコンテナとし、`val` をこのコンテナが格納できる型の値とすると、`find` を使ったエクスプレッションは次のように書けます。

```
find(c.begin(), c.end(), val)
```

なぜ、`c` を 2 回も書かなければならぬのでしょうか。なぜ、ライブラリは、`c.size()` のように

• 16(1)

いふ形で使う関数を用意してくれなかつたのでしよう。あるいは、なぜ、

6146

として、引数で `find` にコンテナを指示できないのでしょうか。これらの形が標準ライブラリにない理由は同じです。反復子を使うことで、`c` を 2 回書かなければなりませんが、これにより、どのようなコンテナの連続部分に対しても適用可能な関数をたった 1 つ作ることができたのです。上に書いたような形式を使ってしまうと、このようにはできません。

まず、`c.find(val)` を考えてみましょう。もし、ライブラリがこのような形を用意していたとすると、`find` は `c` のメンバ関数ということになります。ということは、コンテナ `c` の型を定義する人が `find` をメンバ関数として定義しなければならなくなります。また、ライブラリが `c.find(val)` という形式をアルゴリズムに使うと、この関数は組み込みの配列には使えなくなるのです。これについては、第10章で詳しく見てみましょう。

それでは、なぜ、ライブラリの `find` は `c` そのものではなく、`c.begin()` と `c.end()` を引数に必要とするのでしょうか。それは、こうすることで、コンテナ全体ではなく範囲を指定し、そこだけを調べることもできるからです。たとえば、`find_if` がコンテナ全体にしか使えないとした場合、§ 6.1.1 の `split` をどう書くかを考えてみてください。

ジェネリック関数がコンテナ自身ではなく、反復子を引数に使うのには、さらに微妙な理由もあります。それは、反復子を使うと、普通の意味ではコンテナに入っていない要素にもアクセスできるからです。たとえば、`§ 6.1.2` で `rbegin` という関数を使いました。この関数はコンテナの要素を後からアクセスさせてくれる反復子です。そのような反復子を `find` や `find_if` の引数に使うことで、コンテナの要素を後から探索（検索）できます。もし、これらの関数がコンテナ自身を引数に取るなら、このようなことはできないはずです。

もちろん、ライブラリの関数をオーバーロードして、コンテナを引数に取るバージョンと、反復子のペアを引数に取るバージョンの2つを作ることはできます。しかし、ライブラリをそのように複雑にするほどのメリットはないのです。

8.2.1 アルゴリズムと反復子

テンプレートを使うことで、どのようにしてデータ構造に依存しないプログラムコードが書けるかを理解するには、もっとも一般的な標準ライブラリの関数の実装を調べてみるのが一番です。そのような関数は引数に反復子を取り、それによりその関数が実際に作用するコンテナを判断しているのです。すべての標準ライブラリのコ子を取り、コンテナと `string` のような他のいくつかのものは、コンテナの要素にアクセスするための反復子を持っているでした。

しかし、コンテナによって、可能な操作が異なります。これは、反復子のサポートする操作の違いになります。たとえば、`vector` の要素にはインデックスで直接アクセスすることができますが、`list` の要素にはできません。したがって、`vector` では反復子に整数（インデックスの差）を足すことで別の要素を指す反復子にすることができますが、`list` で同じような操作はできません。

異なる反復子は異なる操作をサポートするのです。そのため、いろいろなアルゴリズムがどのような操作のできる反復子を使うか、どのような反復子にどのような操作が可能かを理解することはとても重要なのです。そこで、同じ意味の操作は、どの反復子に対しても同じ名前であります。たとえば、どの反復子に対しても、コンテナの次の要素を指すのには、`++` が使われるのです。

すべてのアルゴリズムが反復子のすべての操作を必要とするわけではありません。たとえば、`find` のようなアルゴリズムは、反復子のわずかな操作しか使いません。そのため、今まで見てきたどの反復子でも `find` の引数に使うことができます。逆に、`sort` のようなアルゴリズムは、算術的なものも含めて一番多くの操作を反復子に行います。そのため、ここまで見てきた中では、`vector` と `string` の反復子しか `sort` には使えないのです。`(string)` のソートは、個々の文字を非降順に並べ替えます。

それぞれの反復子は、コンテナの要素にどのようにアクセスするかで分類できます。また、反復子の分類がア

8.2 データ構造非依存性

クセスの仕方に対応するということは、反復子は特定のアルゴリズムに対応するとも言えます。たとえば、ある種のアルゴリズムは入力を1回走査するだけなので、複数回走査する能力のある反復子は必要ないのです。また、ある種のアルゴリズムは任意のインデックス要素に効率よくアクセスする必要があります。そのような場合は、整数を足すことができる反復子が必要になります。

これから、おのののアクセスの仕方を説明し、それを使うアルゴリズムを紹介し、対応する反復子の分類をまとめます。

8.2.2 順次読み込み専用アクセス

シーケンシャル（連続的、順次的）な要素を読み込むストレートな方法は、シーケンシャルにアクセスするというものです。標準ライブラリの中で `find` はそのような関数です。これは次のように実装できます。

```
template <class In, class X> In find(In begin, In end, const X& x)
{
    while (begin != end && *begin != x)
        ++begin;
    return begin;
}
```

この関数を `find(begin, end, x)` のように使うと、戻り値は `[begin, end]` の中に最初に `*iter == x` となるような反復子 `iter` か、そのような反復子が存在しない場合は `end` になります。

この関数は内部で `++` しか使わず、したがって `[begin, end]` の範囲にシーケンシャルにアクセスしていきます。また、この関数内では `begin` に `++` を適用しているのと同時に、`begin` と `end` を比較するのに `!=` を使い、`begin` の指す要素にアクセスするのに `*` を使っています。これらは、反復子の指す範囲の要素に順次アクセスするのに十分な操作です。

ただ、上の例ではこれだけで十分ですが、実際にはもっと多くの操作が必要かもしれません。たとえば、`find` が次のように書かれているかもしれません。

```
template <class In, class X> In find(In begin, In end, const X& x)
{
    if (begin == end || *begin == x)
        return begin;
    begin++;
    return find(begin, end, x);
}
```

大抵の C++ プログラムはこのような再帰関数を使いませんが、Lisp や ML に慣れたプログラマには別段奇異ではないかもしれません*2。このバージョンでは、`find` は `++begin` ではなく `begin++` を使い、`!=` の代わりに `==` を使っています。この例から、コンテナの要素に順次読み込み専用アクセスするための反復子は、`++`（前置と後置の両方）、`==`、`!=`、`*` をサポートするべきだとわかります。

*2 訳注：Lisp 言語は LIST Processing（リスト処理）、ML 言語は Meta Language（メタ言語）の略です。似たような言語には、Scheme や Haskell があります。関数型プログラミングに興味を持った読者は、萩谷 昌己著『関数プログラミング』（日本評論社）やサスマン／エイブルソン著『計算機プログラムの構造と解釈』（ピアソン・エデュケーション）等をご覧ください。

それにもう1つサポートすべき操作があります。それは `iter->member` と `(*iter).member` が同じ意味になるということです。つまり、§ 7.2 で見たように、`it->first` は `(*it).first` と同じなのですが、これは一般的に成り立つべき性質と考えられます。

これらのすべての操作ができる反復子を入力反復子 (input iterator) とよびます。実はこれまで見てきたところでは、これらの操作が可能です。したがって、みな入力反復子とよぶことができます。もちろん、コンテナの反復子ではこれらの操作が可能です。しかし、他の操作が可能であれ不可能であれ、これ反復子によっては、これ以上の操作が可能かもしれません。しかし、他の操作が可能であれ不可能であれ、これらは入力反復子とよべるわけです。

「`find` の第1と第2引数には入力反復子が使われる」と言えるわけですが、これは入力反復子に必要なすべての操作が可能な反復子という意味であって、他の機能を持った反復子も使えることに注意してください。

8.2.3 順次書き込み専用アクセス

入力反復子はシーケンシャルな要素の読み込みに使えます。しかし、あきらかに、反復子を使って書き込みをしたい場合もあります。たとえば、`copy` 関数を考えましょう。

```
template<class In, class Out>
Out copy(In begin, In end, Out dest)
{
    while (begin != end)
        *dest++ = *begin++;
    return dest;
}
```

この関数は3つの反復子を引数に取っています。最初の2つはコピー元の要素の範囲を示し、最後の反復子はこの関数は3つの反復子を引数に取っています。最初の2つはコピー元の要素の範囲を示し、最後の反復子はコピー先のシーケンスの先頭を指すものです。これと同じ `while` ループは § 6.1 で見ました。この関数はコンテナ内を `begin` で走査していく、それぞれの要素を `end` に達するまで `dest` の中にコピーしていくのです。

ここで `begin` と `end` の型である `In` と書いたクラスはその名の通り入力反復子です。これらの反復子は `find` のときと同様に読み込みにしか使いません。それでは、`dest` の型である `Out` はどういうものでしょうか。この関数内での `dest` の操作は、`*dest = value` と `dest++`だけです。`find` のときもそうでしたが、`dest++`ができるなら `++dest` もできると考えた方が論理的で整合性があります。

また、あまり明確ではありませんが、もう1つ操作に対して要求されることがあります。ここで `it` を出力専用の反復子とします。すると次のような操作が考えられるでしょう。

```
*it = x;
++it;
++it;
*it = y;
```

`*it` で値を代入する間に、`it` を2回インクリメントしたのです。すると、出力シーケンスの中にギャップができるわけです。もし、反復子を出力だけに使いたいなら、`++it` を `*it` の代入の間に2回以上入れないようにしなければなりません。逆に、`it` をインクリメントする前に `*it` への代入も2回以上してはいけません。

もし、関数がこれらの要求を満たす型を使うならば、この型を出力反復子 (output iterator) とよぶのです。

もし、関数がこれらの要求を満たす型を使うならば、この型を出力反復子 (output iterator) とよぶのです。ここで「書すべての標準コンテナは、`back_inserter` のように、この条件を満たす反復子を持っています。ここで「書き込みは1度だけ」という性質は反復子の性質ではなく、反復子を使うコードが満たすべき性質です。つまり

8.2 データ構造非依存性

り、出力専用の反復子は、コードのこのような性質をサポートする反復子ということになります。たとえば、`back_inserter` で生成される反復子は出力反復子ですが、プログラムは「書き込みは1度だけ」という性質を満たさなければならないのです。しかし、コンテナの反復子はみなそれ以外の操作をサポートするので、それらを使うコードがこの制約を受けるというわけではありません。

8.2.4 順次読み書きアクセス

シーケンスの要素の読み書きをシーケンシャルにだけ行いたいとします。これは、ある要素を1度処理したら、もうその要素に戻れないということです。たとえば、`<algorithm>` ヘッダに定義されているライブラリの `replace` 関数がその例です。

```
template<class For, class X>
void replace(For beg, For end, const X& x, const X& y)
{
    while (beg != end) {
        if (*beg == x)
            *beg = y;
        ++beg;
    }
}
```

この関数は `[beg, end)` の範囲の要素を調べ、`x` に等しいものがあればそれを `y` に置き換える関数です。つまり `For` は入力反復子と出力反復子の両方のすべての操作を持っていることはあきらかです。さらに、「代入は1度だけ」という出力反復子の使用時の制限はありません。なぜならば、要素の値を代入の後に読み込んで、またそれを変えるということも考えられるからです。このような反復子を前方向反復子 (forward iterator) と言います。前方向反復子とは、正確には、次のような操作を持っているものです。

```
*it (読み込みと書き出しの両方で使う)
++it と it++ (ただし、--it と it--はなくてよい)
it == j と it != j (j は itと同じ反復子とする)
it->member ((*it).member の代わりに)
```

すべての標準ライブラリのコンテナは前方向反復子を持つ条件を満たします。

8.2.5 可逆アクセス

コンテナ内の要素に後からアクセスする関数もあります。もっともストレートな例は、`<algorithm>` ヘッダに定義されている `reverse` のような関数です。

```
template<class Bi> void reverse(Bi begin, Bi end)
{
    while (begin != end) {
        --end;
        if (begin != end)
            swap(*begin++, *end);
    }
}
```

}

このアルゴリズムでは、`end` をコンテナの終わりから前方へ進め、`beg` ははじめから後方へ進め、それぞれが指す要素を入れ替えていっているのです。

この関数は `begin` と `end` を前方向反復子のように扱っていますが、さらに`--`という演算子も使ってています。これはあきらかに、反復子を後から前に動かすのに必要なものです。このように、前方向反復子のすべての条件を満たし、(前置と後置両方の) `--`操作を持つものを、双向方向反復子 (bidirectional iterator) といいます。

すべての標準ライブラリのコンテナは双向方向反復子を持っています。

8.2.6 ランダムアクセス

ある種の関数は、コンテナ内をとびまわってその要素にアクセスします。その1つのよい例は、古典的な2分探索のアルゴリズムです。標準ライブラリではこのアルゴリズムがいろいろな形で使われていますが、もっともストレートなものは、`binary_search` です。標準ライブラリの実際のコードでは（この本の範囲をはるかに超えた）賢い方法が使われ、前方向反復子の定義されたシーケンスで2分探索ができるようになっています。しかし、以下で見るよう、単純な方法ではランダムアクセス反復子が必要になるのです。

```
template<class Ran, class X>
bool binary_search(Ran begin, Ran end, const X& x)
{
    while (begin < end) {
        // 探索範囲の中央を見つける
        Ran mid = begin + (end - begin) / 2;
        // xがどちらの範囲にあるかを調べ、次にその範囲を調べるようにする
        if (x < *mid)
            end = mid;
        else if (*mid < x)
            begin = mid + 1;
        // ここに到達すれば、*mid == xがあったということ
        else return true;
    }
    return false;
}
```

この関数では、これまで見てきた反復子の性質に加えて、反復子の計算を行っています。たとえば、反復子から反復子を引いたものは整数になるという性質、反復子に整数を足すと、また反復子になるという性質を使っています。このような性質を論理的かつ整合的にまとめると、`p` と `q` を反復子とし、`n` を整数とすると、双向方向反復子の満たすべき条件に加えて、次のようなものになるということです。

```
p + n, p - n, n + p
p - q
p[n] (*p + n)と同じ
p < q, p > q, p <= q, p >= q
```

反復子同士の引き算の答えは、それぞれの間の「距離」になります。この型は整数ですが詳細は § 10.1.4 で説明します。ここで`==`と`!=`を書きませんでしたが、ランダムアクセス反復子は双向方向反復子の性質を持っており、それはすでに含まれているからです。

ここまで見てきたアルゴリズムで実際にランダムアクセス反復子を使っているのは `sort` 関数です。`vector` と `string` の反復子はランダムアクセス反復子なのです。しかし、`list` の反復子は違います。`list` には双向方向反復子しかありません。どうしてでしょうか。

`list` は要素のすばやい挿入と削除を最適化するように作られているからです。そのため、`list` の任意の要素にすばやくアクセスすることはできません。任意の要素にアクセスするには、シーケンスの各要素を最初からたどっていかなければならぬのです。

8.2.7 反復子の範囲と最後の1つ後の値

これまで見てきたように、アルゴリズムで範囲を指定するのに2つの引数を使う方法は、標準ライブラリで非常によく見られるものです。ここで前の引数は範囲の先頭（最初の要素）を指し、後の引数は範囲の末尾（最後の要素の1つ後）を指します。どうして、最後の要素の1つ後を考えるのでしょうか。そもそもそれは有効なのでしょうか。

実は § 2.6 で、与えられた範囲の上限として最後の要素の1つ後が重要であることを見ています。つまり、最後の要素の値によって範囲の終わりを示すようにしておくと、最後の要素は特別なものになるのです。最後の値が特別扱いされると、最後に到達する前に誤って終了するループを書きがちになります。ループに関する限り、範囲の最後を示すのに直接最後の要素を指す反復子ではなく、その1つ後を指す反復子を使う理由は、さらに3つあります。

最初の理由は範囲に要素がまったくない場合です。その場合、最後の要素もないので、そのような場合、最後の要素を指す反復子が、最初の要素の1つ前を指すようにすることになるでしょう。しかし、そうすると、範囲に何も要素がない場合だけ特別な処理をするようにしなければなりません。これではプログラムは、わかりにくく信頼できないものになります。§ 6.1.1 で見たように、範囲に要素がなくても要素がある場合と同じように扱えるということで、プログラムが簡単になるのです。

2番目の理由は、範囲の最後を示すのに最後の要素の1つ後を使うと、引数の反復子同士が等しいか等しくないかを一度だけ調べればよくなるということです。ポイントは、引数の2つの反復子が一致していれば、指定された範囲に要素がないことがすぐわかるということです。そして、指定された範囲に要素がないのはこの場合に限られています。逆に2つの反復子が異なれば、前の反復子は実際の要素を指し、それを処理してループの次のステップに進み、処理すべき範囲を要素1つ分小さくできるわけです。より具体的にいうと、最初の要素を指す反復子と最後の要素の1つ後を指す要素で範囲を指定すると、次のような形のループが書けるということです。

```
// 不変な表明：[begin, end)の範囲を処理しなければならない
while (begin != end) {
    // beginの指す要素を処理する
    ++begin;
}
```

各ステップで、反復子同士が等しくないか（または等しいか）どうかを1度調べているだけです。

3番目の理由は、範囲を示すのに最初の要素の位置と最後の要素の位置の1つ後を使うことで、「範囲の外」を自然に示すことができるということです。自分で書くアルゴリズムもそうですが、多くの標準ライブラリのアルゴリズムは2番目の範囲外の反復子を戻すことでなんらかの失敗を表せます。たとえば、§ 6.1.3 で URL を探

し出す関数 url_beg を書きましたが、この関数は URLがないときには 2番目の反復子を戻すようにしました。簡単な要素の 1つ後を指す反復子というものは少々奇妙でも、これを使うことでプログラムをより簡単に信頼性の高いものにできるということです。このため、すべてのコンテナには、このような反復子がどのように作られています。そして、コンテナの end というメンバ関数がその反復子を戻すのです。また、他 c.size() 分だけインクリメントすると、その結果は c.end() に等しくなります。ただし、最後の要素の 1つ後を指す反復子が、実際に指す要素は定義されていません。これについては、最初の要素の 1つ前も、最後の要素のいくつか後も同じです。

8.3 入力反復子と出力反復子

標準ライブラリのコンテナでは、入力反復子・出力反復子と前方向反復子の区別がないのに、どうして、そんな違いを考えるのでしょうか。1つの理由は、必ずしもすべての反復子が直接コンテナに付随しているのではなく、たとえば、c が push_back を持つコンテナとするときに、back_inserter(c) は他の反復子の条件を満たさない出力反復子になるのです。

また、別の例をあげると、標準ライブラリには、入力ストリームと出力ストリームに結びついた反復子があります。驚くことではありませんが、istream を表す反復子は入力反復子の条件を満たし、ostream を表す反復子は出力反復子の条件を満たしています。このようなストリームに対応する反復子を使うことで、istream や ostream への操作を、普通の演算子で行えるのです。たとえば、++はストリームに対応する反復子を 1つ先に進めることになります。入力ストリームに対して *は入力の現在位置の値を戻し、出力ストリームに対して *は対応する ostream への書き込みを可能にします。このようなストリーム反復子は <iterator> ヘッダで定義されています。

入力ストリーム反復子は入力反復子の一種で istream_iterator といいます。

```
vector<int> v;
// 整数を標準入力から読み込みvに格納していく
copy(istream_iterator<int>(cin), istream_iterator<int>(),
     back_inserter(v));
```

いつものように、copy のはじめの 2つの引数はコピーする範囲を指定するものです。最初のものは cin について新しい istream_iterator を生成し引数にしているのですが、これは int 型の値を読み込むことを示しています。C++の入力と出力では、処理される値の型が決まっています。つまり、ストリームから読み込むときは、読み込まれる値の型が、たとえ非明示的でも、あらかじめ決めるのです。たとえば、以下のようにになります。

```
getline(cin, s); // 読み込まれるデータはstringに格納される
cin >> s.name >> s.midterm >> s.final; // stringと2つのdoubleを読み込む
```

同様に、ストリーム反復子を定義するときは、どういう型の値を読み込むのか、あるいは書き込むのかを指定する必要があるのです。そのため、ストリーム反復子はテンプレートになっています。

copy の 2番目の引数はデフォルトの（空の）istream_iterator<int>オブジェクトを生成しています。これはどのファイルにも対応しないオブジェクトです。istream_iterator には特別に定義されたデフォルトの値があり、反復子がファイルの終わりに到達するか、入力エラーが起こると、反復子自身がこのデフォルト値になるようになっています。そのため、このデフォルト値を、便宜上「最後の要素の 1つ後を指す反復子」として copy 関数に渡すことができるのです。

こうして、上のような copy の使い方で、標準入力から「ファイルの終わり」か int として無効な値に到達するまで、値を読み込めるわけです。

しかし、書き込みには ostream_iterator は使えません。書き込みをしたい場合は、出力用の反復子である ostream_iterator のオブジェクトを使わなければならないのです。

```
// vの要素を標準出力に空白で区切りながら出力
copy(v.begin(), v.end(), ostream_iterator<int>(cout, " "));
```

これで vector の要素をすべて標準出力に書き込むことができます。ここで 3番目の引数は、cout に int を書き込むための反復子です。

ostream_iterator<int> を生成するために使われている 2つの引数のうち後の方は、それぞれの要素の後に書き込むものを示しているのです。ここで使われる典型的なものは文字列リテラルです。もし、ここに何も指定しなければ、ostream_iterator は各要素のあとに何も書き込みず、続けて出力することになります。つまり、これを省略すると、copy はすべての要素を連続して出力するので、ごちゃごちゃして読めなくなってしまうでしょう。

```
// 各要素の間に何も書かない!
copy(v.begin(), v.end(), ostream_iterator<int>(cout));
```

8.4 柔軟性のための反復子

§ 6.1.1 で書いた split 関数を少しだけ改良することができます。そこでは split は vector<string>を戻す関数でしたが、それでは柔軟性がありません。この関数のユーザは、vector ではなく list<string>の、別のコンテナを使いたいと思うかもしれません。また、split のアルゴリズムは決して vector のみに適用可能なではありません。

ここで split を、値を戻すのではなく、出力反復子を引数に取るものにすれば、より柔軟性のあるものになります。このバージョンでは、見つけた単語を書き込む場所を示すのに反復子を使うことにします。この関数のユーザは、出力場所を確保してその反復子を引数に使うのです。

```
template <class Out> // 変更した
void split(const string& str, Out os) { // 変更した
    typedef string::const_iterator iter;
    iter i = str.begin();
    while (i != str.end()) {
        // 頭にある空白を無視する
        i = find_if(i, str.end(), not_space);
        // 次の単語の終わりを見つける
        iter j = find_if(i, str.end(), space);
```

```
// [i, j)の範囲の文字をコピーする
if (i != str.end())
    *os++ = string(i, j); // 変更した
i = j;
}
```

§ 6.2.2で書いた`write_analysis`関数のように、この新しいバージョンの`split`は何も戻しません。そのため、この関数の戻り値は`void`にしました。そして、この関数をテンプレート関数にし、そのパラメータを`Out`としました。出力反復子を使いたいので、このような名前にしたのです。前方向、双向向、ランダムアクセス反復子は、すべて出力反復子として使えることに注意してください。そのため、新しい`split`は、`istream_iterator`のような純粋な入力反復子以外には使えるのです。

関数の引数である`os`の型は`Out`です。見つけた単語を書き込むのに、これを使いますが、これは関数の終わりの方で行われています。

```
*os++ = string(i, j); // 変更した
```

これで見つけた単語の書き込みになります。`*os`は`os`に付随したコンテナの現在の書き込み場所を意味します。そこに`string(i, j)`を書き込んでいるのです。それから出力反復子の規則に従って`os`をインクリメントしています。これで次のステップではコンテナの次の場所に新しい単語を書き込むことになるわけです。

この新しい`split`を使うときはプログラムを変更しなければなりません。しかし、このようにしたおかげでほとんどすべてのコンテナに対して`split`が使えるようになったのです。たとえば、`s`を`string`とし、その中の単語を`word_list`という名前の`list`に入れたいとします。すると、`split`は次のように使えるでしょう。

```
split(s, back_inserter(word_list));
```

同様に、`split`をチェックする簡単なプログラムを次のように書くことができます。

```
int main()
{
    string s;
    while (getline(cin, s))
        split(s, ostream_iterator<string>(cout, "\n"));
    return 0;
}
```

§ 5.7で書いたチェックプログラムと同様に、ここでは`split`を使って入力行を単語に分解しています。そして、単語は標準出力に出力されるようにしました。`split`の引数にした`ostream_iterator<string>`は`cout`に対応するようにしました。`split`内部で`*os`に文字列が割り当てられるときに、実際には`cout`に書き込まれるわけです。

8.5 詳細

テンプレート関数： 単純な型の値を戻すものは次のような形をしている

8.5 詳細

```
template<class type-parameter [, class type-parameter] ... >
ret-type function-name( parameter-list)
```

型パラメータ`type-parameter`は関数定義の中で使われるだろう型の名前。一般に、型を示すにはパラメータリストが使われる。しかしパラメタリスト`parameter-list`がなければ、この関数のユーザがその型をはっきり示す必要がある。たとえば、

```
template<class T> T zero() { return 0; }
```

は、`zero`が型パラメータを1つ持つテンプレート関数であることを示しているが、その型は戻り値の型として使われている。この関数を使う場合は、戻り値を、たとえば次のように明示しなければならない。

```
double x = zero<double>();
```

`typename`は、テンプレートの型パラメータを使って宣言されるものに使われる。たとえば、

```
typename T::size_type name;
```

は、`name`の型が`T`の内部の型である`size_type`であることを示している。

コンパイラは関数に使われている型を見て、テンプレート関数を個別に具体化する。

反復子： C++の標準ライブラリの基本的な考え方方は、アルゴリズムとコンテナを反復子でつなぐことで、データ構造非依存性が得られるというものである。さらに、アルゴリズムを、それが使う反復子の操作の種類によって分類することで、アルゴリズムが使えるコンテナをはっきりさせている。

反復子は次の5つに分類される。基本的に下に挙げたものは、上の反復子の持つ操作を持っている。

入力反復子	一方向へのシーケンシャルなアクセスに使う。入力のみに使われる。
出力反復子	一方向へのシーケンシャルなアクセスに使う。出力のみに使われる。
前方向反復子	一方向へのシーケンシャルなアクセスに使う。入出力に使える。
双向向反復子	前・後方の両方向へのシーケンシャルなアクセスに使う。入出力に使える。
ランダムアクセス反復子	どの要素にも効率的にアクセスできる。入出力に使える。

課題

8-0 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。

8-1 § 6.2で書いた`analysis`にはいろいろなバージョンがありました。これらには共通するものがありました。これらは最終成績を計算するために内部で使う関数によって異なるのです。これを、「成績計算の関数の型」をパラメータとするテンプレート関数に書き直し、成績処理を比較するのに使ってください。

8-2 第6章で使い、§ 6.5で説明した以下のライブラリの関数の定義を書いてみてください。どのような反復子を使うか考えてください。その際、それぞれの関数で必要な反復子の条件をなるべく小さいものにしてください。§ B.3を見て、自分の考えたものと比較してください。