

## 課題

- 6-0 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。
- 6-1 § 5.8.1 の `frame` と § 5.8.3 の `hcat` を反復子を使って書き直してください。
- 6-2 `find_urls` 関数の動作を試すプログラムを書いてください。
- 6-3 下のコードは何をするものでしょう。

```
vector<int> u(10, 100);
vector<int> v;
copy(u.begin(), u.end(), v.begin());
```

このコードを含むプログラムを書き、コンパイルし実行してください。

- 6-4 上のプログラムで `u` から `v` へのコピーのところを直してください。少なくとも 2 つの直し方があります。
- その両方を書き、両者を比較したときの長所短所を書いてください。
- 6-5 `optimistic_median` 関数を使う成績処理関数を書いてください。
- 6-6 上の問題の関数や § 6.2.2 の関数、§ 6.2.3 の関数は同じ仕事をする関数です。これらを 1 つの関数にしてください。
- 6-7 § 6.2.1 のプログラムの「学生の成績データを読み込み、宿題を全部したかどうかでデータを分類し、その成績を処理する部分」は、`extract_fails` の処理に似ています。この部分を担当する関数を書いてください。
- 6-8 独自の基準で学生を分類する関数を書いてください。そして、この関数を `extract_fails` の中に使いプログラムに組み込み、学生の成績を解析してください。
- 6-9 ライブライアリのアルゴリズムを使って `vector<string>` の全要素をつなげる方法を考えてください。

## 連想コンテナを使う

ここまで使ってきたコンテナはみなシーケンシャル（列になっている）コンテナとよばれるものです。それは、要素はそのコンテナのシーケンス（列）の中にあるということです。`push_back` や `insert` を使ってコンテナに要素を付け加えると、そのコンテナ内の要素の順番を変える操作をしない限り、付け加えられた場所に居ることになっています。

シーケンシャルコンテナだけで効率的に書くことが難しいプログラムもあります。たとえば、整数のコンテナがあったとします。コンテナ内に 42 という整数値があるかどうかを調べるプログラムを書きたいとすると、決して理想的ではない、しかしもっともらしい 2 つの方法が考えられます。1 つの方法は、コンテナの要素を 1 つひとつ調べていって、42 があるかどうかを調べるというものです。この方法は、ストレートですが、コンテナの要素が多いときには、遅くなります。もう 1 つの方法は、コンテナ内の要素を適当に並べておいて、探していく要素を効率よく見つけるというものです。この方法は、速い探索（検索）かもしれませんのが、そのようなアルゴリズムを考えるのは簡単ではありません。言葉をえれば、遅いプログラムでがまんするか、洗練されたアルゴリズム自分で考え出すか、ということです。しかし幸い、この章で紹介するように、ライブライアリを使う第 3 の方法があります。

### 7.1 効率的な探索ができるコンテナ

データを保持するのに、シーケンシャルなコンテナの代わりに、連想コンテナ<sup>1</sup>（associative container）というものを使うことができます。この種のコンテナは要素を挿入すると、その値に基づいてコンテナ内の要素が並び替えられます。それだけでなく、この順番を使って、シーケンシャルコンテナより速い探索方法を提供します。ここでプログラマがコンテナ内の順番を考える必要はないのです。

連想コンテナは、特定の値が内部にあるかを探索するのに効率的で、しかも、さらに情報を付け加えることもできます。この効率のよい探索に使う要素の部分をキー（key）といいます。たとえば、学生のデータなら、学生の名前をキーに使うことが考えられます。それにより、学生を名前で効率よく探せるのです。

シーケンシャルコンテナでキーに近いものをあげると、`vector` の全要素に付随している整数インデックスがキーに近いと言えるかもしれません。しかし、本当にキーというわけでもないのです。`vector` の要素を挿入したり削除したりすると、挿入や削除した後の要素のインデックスは変わってしまうからです。

<sup>1</sup> 訳注：この訳は広く使われているので本書でもそれに従います。しかし、ここでいう associative は「連想」というより、「対応」「結合」「随伴」などがぴったりくる日本語だと思います。以下で見るよう、これは「あるものを別のものに対応させるコンテナ」だからです。

連想型のデータ構造でもっともよくあるものは「キーと値のペア」です。これはキーに値を対応させたもので、キーに基づいて要素を効率的に挿入したり取り出したりするのに使われます。キーと値のペアのデータ構造で、値にキーを対応させた場合、この対応はペアが破棄されるまで変えられません。そのようなデータ構造を連想配列（associative array）とよぶのです。AWK や Perl、Snobolといった多くの言語には、連想配列が組み込まれています。C++では、連想配列はライブラリに入っています。そして C++のもっともよく使われる連想配列は map とよばれるものです。これは、他のコンテナのように、<map> ヘッダに定義されています。

map はいろいろな点で vector に似ています。基本的な違いは、map のインデックスが整数とは限らず、string など順番を決める事でできるものなら他の型の値よいのです。

連想コンテナとシーケンシャルコンテナの他の重要な違いに、連想コンテナは自分で順序を作るので、その要素の順番をプログラムから変更することはできないことがあります。そのため、コンテナ内の要素の順番を変えるアルゴリズムの多くは連想コンテナには使えません。この制限の代償として、連想コンテナはシーケンシャルコンテナの内部構造では効率的に実現できない多くの有用な操作を可能にしているのです。

この章では、探索が大きなウエイトを占めるプログラムを map を使って効率的に書く例をお見せします。

## 7.2 単語を数える

簡単な例として、入力文字列中に同じ単語が何回出てくるかを数える方法を考えましょう。連想コンテナを使うと、それはほとんど自明になります。

```
int main()
{
    string s;
    map<string, int> counters; // 単語とその登場回数を記録する
    // 入力を読み込み、個々の単語とその登場回数を記録していく
    while (cin >> s)
        ++counters[s];
    // 単語とその登場回数を書き出す
    for (map<string, int>::const_iterator it = counters.begin();
         it != counters.end(); ++it) {
        cout << it->first << "\t" << it->second << endl;
    }
    return 0;
}
```

他のコンテナと同様に、まず map の保持するオブジェクトの型を決めなければなりません。map はキーと値のペアを保持するので、値だけでなくキーの型も示す必要があります。それで次のように書くのです。

```
map<string, int> counters;
```

これは counters という map を、キーが string で値が int であるように定義するということです。このようなものを、しばしば「string から int への map」と言います。<sup>\*2</sup>これにより、キーとして string オブジェクトが与えられたときに、map によって対応する int データを取り出すことができるからです。

<sup>\*2</sup> 訳注：map には「対応付け」という意味があります。

この counters をこのように定義したのは、読み込まれた各単語とそれがこれまで何回登場してきたかを対応させたいからです。入力のループは、まず標準入力から単語を 1 つずつ読み込み、それを s に入れます。そして

```
++counters[s];
```

というおもしろい命令を実行するのです。これは、キーとして読み込まれた単語 s を counters の中で探し、counters[s] はその s に対応する整数を表します。この整数をインクリメントするために++を使いますが、これで読み込んだ回数を 1 増やすことになるわけです。

もし、読み込まれた単語が始めて登場するときはどうなるのでしょうか。その場合、counters はそのキーに対応する値を持っていません。実は、map で存在しないキーをインデックスに使った場合、map は自動的にそのキーの要素を新しく生成することになっています。そして、その要素は、値初期化（value-initialized）されているのです。これは int のような単純な型の場合、0 にセットするということです。そこで、始めての単語が読み込まれ ++counters[s] が実行されるときには、counters[s] は 0 になるのです。それをインクリメントするので、これはちゃんと 1 になるわけです。

入力が終われば、counters 内の単語を対応する数、つまり登場回数と一緒に出力しなければなりません。この大部分は、list や vector の出力の場合と同じです。コンテナの中の要素は反復子を使って for ループで出力します。ここで反復子は map で定義されている適当な型のものです。実際、シーケンシャルの場合と本当に違うのは、for の中身だけです。

```
cout << it->first << "\t" << it->second << endl;
```

連想配列はキーと値のペアを保持するものです。[] を使って map の要素にアクセスすると、[] の中にキーを入れて値を取り出しますので、この事実が見えにくくなります。たとえば、counters[s] は int の値でした。しかし、map の全要素について何かを反復して繰り返すときには、キーとそれに対応する値の両方を取り出す必要が出てきます。map では、このため、同様にライブラリにある型の pair を使います。

pair は、first と second という名の 2 つの要素を持ったシンプルなデータ構造です。map の各要素は実際に pair で、その first 要素がキーであり second 要素が対応する値なのです。そこで map の要素を反復子を使って取り出しますと、それは pair オブジェクトなのです。

pair はいろいろな種類の型の値を保持できるので、pair オブジェクトを生成するときには、その first と second の型を指定する必要があります。map に対しては、キーの型 K と値の型 V を指定しますが、これで map の pair は pair<const K, V>という型になるのです。

ここで map に使われる pair のキーの型が const であることに注意してください。pair のキーが const であるため、キーを変えることができないのです。もし、キーが const でなければ、map の要素の位置を見えないところで変わってしまうかもしれません。しかし、キーはいつも const で、map<string, int> の要素を反復子から取り出しますと、pair<const string, int> オブジェクトを得ることができます。そして it -> first が現在の要素のキーになり、it -> second が対応する値になります。it が反復子であるため、\*it は左辺値であり、そのため it -> first と it -> second も左辺値です。しかし、it -> first は const であるため、変えることができないのです。

以上で、出力のステートメントが、キー（これは入力された個別の単語）、それからタブ、そしてその単語の登場回数を書き出すことがわかります。

### 7.3 対照表を作る

入力中の単語の数を数えることができるようになったので、次に、どこにその単語が現われたかを示す対照表（相互参照表）を作るプログラムを書いてみましょう。このように拡張するために、基本プログラムは何箇所か変更することになります。

まず、単語を1つずつ読み込むのではなく、入力を1行単位にしなければなりません。行番号と単語を対応付けるためです。そうすると、読み込んだ1行を単語に分解する必要が出てきます。幸い、そのような関数はもう書いてあります。§ 6.1.1 の `split` です。この関数を使い、入力された行から単語を取り出し `vector<string>` にすることができます。

しかし、`split` は直接使うより、対照表を作る対照表関数 `xref` を作り、そのパラメータにすることにします。そうすることで、行から単語を取り出す方法を後で変更できるわけです。たとえば、§ 6.1.3 の `find_urls` 関数と対照表関数を使うことで、入力文字列中のどこに URL が現われたかを知ることができます。

前節と同様に、`map` を入力中の個別の単語がキーになるように定義します。しかし、対応する値はもう少し複雑にします。単語の登場回数ではなく、その単語が登場した行の番号をすべて知りたいのです。どの単語も何度も現われる可能性があるので、複数の行番号を記録しておく必要があります。

そこで、既出の単語が新しい行で現われた場合、その行の番号をすでに記録してある行番号に付け加えることになるのです。そのためのコンテナはシーケンシャルなもので十分でしょうから、行番号を記録するには `vector` を使います。つまり、`string` を `vector<int>` に対応付ける `map` が必要なわけです。

これらの考察をもとに、次のコードを考えてみましょう。

```
// 入力中の各単語が現われる行を見つける
map<string, vector<int> >
xref(istream& in,
     vector<string> find_words(const string&) = split)
{
    string line;
    int line_number = 0;
    map<string, vector<int> > ret;
    // 次の行を読む
    while (getline(in, line)) {
        ++line_number;
        // 入力された行を単語に分解する
        vector<string> words = find_words(line);
        // 現在の行の単語をすべて記録する
        for (vector<string>::const_iterator it = words.begin();
             it != words.end(); ++it)
            ret[*it].push_back(line_number);
    }
    return ret;
}
```

戻り値とパラメータの両方に注意してください。この戻り値の宣言とローカル変数の `ret` には、`>>` ではなく、気をつけて `>>` を使っているのです。コンパイラは2つの`>`の間にスペースがないと、`>` が2度使われているのではなく、`>>` という演算子と判断してしまうからです。

### 7.3 対照表を作る

パラメータには、`find_words` という関数パラメータが使われています。これは `xref` 関数に、単語を分解する関数を引数として与えたいからです。おもしろいことに、このパラメータの後には `= split` というものが付いています。これはこのパラメータがデフォルト引数（default argument）を持つということです。一般に、パラメータにデフォルト引数を与えると、この関数を使うときに引数を省略できるようになります。関数を使う（呼び出す）ときに、引数を与えればそれが使われますが、もし引数を省略すれば、コンパイラがデフォルトのものを使うのです。つまり、ユーザはこの関数を次の2つ形で使えるのです。

```
xref(cin); // 入力ストリームから単語を取り出すのにsplitを使う
xref(cin, find_urls); // 単語を取り出すのにfind_urlsという関数を使う
```

関数の定義は、`string` オブジェクト `line` を定義することからはじまっています。これは入力文字列を格納するためのものです。つぎに、現在処理している行の番号を保持するために、`line_number` という `int` 整数を定義しています。入力ループは `getline` (§ 5.7) を使って、1度に1行ずつ読み込んでいきます。そして、入力がある限り、`line_number` をインクリメントし、その行の単語を処理していくのです。

処理の始めに `words` というローカル変数を定義し、`find_words` で初期化して、`line` のすべての単語を格納するようにします。いずれにせよ、`find_words` は `line` を単語に分解する `split` (§ 6.1.1) か別の `string` オブジェクトを引数に取り `vector<string>` を戻す関数です。`for` 文では `words` のすべての要素について、`map` である `ret` を更新していきます。

`for` のヘッダはもうおなじみになっているはずです。これは反復子を定義し、`words` の全要素について順番に走査します。`for` の中身は、始めての人には難しく見えるかもしれません。

```
ret[*it].push_back(line_number);
```

そのため少し分解して説明しましょう。反復子 `it` は `words` の要素を指します。つまり、`*it` は入力行の単語の1つを意味します。これを私たちの `map` である `ret` のキーにします。`ret[*it]` というエクスプレッションは、`*it` に対応する値を戻します。これは、この単語がこれまで現われた行番号を保持する `vector<int>` です。そこで、この `vector` のメンバ関数 `push_back` を使い、現在の行番号を追加しているのです。

§ 7.2 で見たように、この単語が登場するのが始めての場合、対応する `vector<int>` は値初期化されます。クラス型オブジェクトの値初期化は少し込み入っているので、詳しくは § 9.5 で見ますが、ここでは `vector` の場合、`vector` が初期値を与えられずに生成されたものと同じになる、ということだけ述べておきましょう。今の場合も初期値なしで `vector` が生成されるのです。そのため、新しい `string` オブジェクトを `map` のキーにした場合、空の `vector<int>` が対応付けられるのです。したがって、この場合、`push_back` は空の `vector` に現在の行番号を入れることになります。

こうして `xref` 関数ができたので、これを使って対照表を作ることができます。

```
int main()
{
    // デフォルトのsplitを使ってxrefを呼び出す
    map<string, vector<int> > ret = xref(cin);
    // 結果を書き出す
    for (map<string, vector<int> >::const_iterator it = ret.begin();
         it != ret.end(); ++it) {
        cout << it->first << "は以下の行で出てきました：" ;
        for (int i : it->second)
            cout << i << " ";
    }
}
```

```
// その単語が登場した行の番号を書く
vector<int>::const_iterator line_it = it->second.begin();
cout << *line_it; // 最初の行番号を書く
++line_it;
// もしあれば、残りの行番号を書く
while (line_it != it->second.end()) {
    cout << ", " << *line_it;
    ++line_it;
}
// 次の単語と分けるため改行する
cout << endl;
}
return 0;
}
```

このコードは map の更新の説明のときのコードのように、見慣れないものかもしれません。それにもかかわらず、これまでに見てきたものしか使ってはいないのです。

まず、`xref` を使って、各単語とそれが登場する行番号を記録したデータ構造を生成しています。この関数の関数パラメータにはデフォルト引数を使っています。つまり、今の場合、`xref` は入力行を単語に分解するのに `split` を使うわけです。プログラムの残りの部分は、生成したデータ構造の中身を出力しているだけです。

実際、このプログラムの大部分は `for` 文です。この形は § 7.2 で説明しました。これは `ret` の最初の要素から始めて、順次すべての要素を見ているのです。

`for` ループの処理部分を考えるときには、反復子を使って取り出す `map` の要素は `pair` であることを思い出してください。`pair` の `first` 要素は (`const` の) キーです。そして `second` はそのキーに対応する値でした。

`for` ループの中身では、まず、処理している単語とメッセージを書き出します。

```
cout << it->first << "は以下の行で出てきました:";
```

単語は、反復子 `it` で示される `map` の要素のキーです。そのキーは `pair` から `first` 要素を取り出すことで得られるのです。

その単語が何回登場しているかわからないので、このように最初にメッセージを書いてしまうことにしました。少なくとも 1 つの行がキーとなる単語を含んでいることは確かですが、それ以外に何行あるかはわかりません。

`it -> first` と同様に、`it -> second` は対応する値を取り出すことになります。今の場合、それは `vector<int>` で、キーの単語が登場する行の番号を保持しています。ここで、`it -> second` の要素にアクセスするために反復子 `line_it` を定義します。

これらの行番号の間にはコンマを入れたいのですが、最後の行番号の後には入れたくありません。そのためには、最初か最後の行番号を特別扱いする必要があります。そこで、最初の行番号を特別扱いして、別個に出力することにしました。`ret` の全要素について、必ずキーを含む行番号が 1 つはあるので、これは問題ありません。そして、最初の要素を出力してから、処理を進めるため反復子をインクリメントします。それから、`while` ループで `vector<int>` の次の要素を (もしあれば) 出力していくのです。それぞれの要素について、まずコンマを出力し、要素の値である行番号を出力しています。

## 7.4 文を作る

本章の最後に、少し複雑な例をお見せしましょう。それは文の構造に関する記述、つまり文法を基に、文法に従った文を作り出すプログラムです。これに `map` を使うのです。たとえば、英語の文は「名詞+動詞」や「名詞+動詞+目的語」などという構造を持っています。

もし、複雑なルールを扱うことができれば、もっとおもしろい文を作ることもできるでしょう。たとえば、単に「名詞+動詞」ではなく、「名詞句+動詞」とするのです。ここで名詞句とは、名詞そのものが「形容詞付きの名詞」のことです。具体的な例として、次のものを始めに考えます。<sup>\*3</sup> 今考えているプログラムは、たとえば次のような文を生成するものです。

```
the table jumps wherever it wants
(テーブルは、好きなところで跳ぶ)
```

このプログラムは常に、文を作るルールを見つけることからはじめます。上に示した中では、文を作るルールは 1 つしかありません。それは表の最後のもの、

```
<sentence>      the <noun-phrase> <verb> <location>
```

です。このルールは、「文は、`the +名詞句+動詞+場所`」で作られるとしています。次にプログラムは、名詞句のルールを探します。ここで名詞句の作り方は 2 つありますが、どちらかをランダムに選ばせます。上の例では、

| 分類            | ルール                                 |
|---------------|-------------------------------------|
| <noun>        | cat                                 |
| <noun>        | dog                                 |
| <noun>        | table                               |
| <noun-phrase> | <noun>                              |
| <noun-phrase> | <adjective> <noun-phrase>           |
| <adjective>   | large                               |
| <adjective>   | brown                               |
| <adjective>   | absurd                              |
| <verb>        | jumps                               |
| <verb>        | sits                                |
| <location>    | on the stairs                       |
| <location>    | under the sky                       |
| <location>    | wherever it wants                   |
| <sentence>    | the <noun-phrase> <verb> <location> |

<sup>\*3</sup> 訳注：左にある「分類」を具体的にどのように構成するかを示しているのが右にある「ルール」です。また、`<noun>` は名詞、`<noun-phrase>` は名詞句、`<adjective>` は形容詞、`<verb>` は動詞、`<location>` は場所、`<sentence>` は文を意味します。

```
<noun-phrase>    <noun>
```

のルールが選ばれました。つまり、名詞句として単なる名詞を使うことにしたのです。次に名詞を作るルールをランダムに選ぶわけですが、ここでは、

```
<noun>          table
```

が選ばれました。それから動詞を選ばなければなりませんが、ここでは、

```
<verb>          jumps
```

が選ばれています。同様に、場所については、

```
<location>      wherever it wants
```

が選ばれたわけです。この最後の選択は、文を完成させるため、複数の単語で場所を表していることに注意してください。

#### 7.4.1 ルールを表す方法

文法の表は2種類のものを含んでいます。1つは角カッコで囲まれた「分類」というもので、もう1つは普通の単語です。「分類」には1つかそれ以上の「ルール」が対応し、単語はそれだけで独立しています。プログラムが角カッコに囲まれた文字列を見つけると、それは「分類」と判断できます。そこでプログラムは「分類」を見つけ、それを右側にある「ルール」にしたがって展開するようにすればよいのです。もし、角カッコつきの「分類」ではなく、単語に行き当たれば、その場所には単語を書き込んで文ができるはずです。

このプログラムは、文の構成方法を読み込み、それからその方法にしたがってランダムに文を生成するわけです。そこで最初の問題は、この構成方法をどのように保持するか、です。文を生成するときは、まず「分類」に対応する「ルール」を見つけ、「ルール」にしたがって「分類」を展開するわけです。たとえば、まず<sentence>があり、それを展開する「ルール」が必要になります。この「ルール」は、上述のように<noun-phrase>、<verb>、<location>などからできているので、これらを展開する「ルール」が必要になります。そして、このような処理が続けられるわけです。あきらかに、「分類」と「ルール」を対応させるmapが役に立ちそうです。

しかし、どのようなmapがよいでしょう。「分類」は簡単です。これはstringとして保持できるので、mapのキーはstringでよいはずです。

値の型はもっと複雑です。表を見ると、「ルール」はstringの集まりであることがわかります。たとえば、<sentence>の「ルール」はtheと他の3つのstringで表せる「分類」、つまり4つの部分から成り立っています。このような値を表すにはどうすればよいか、実はもう知っています。つまり「ルール」を保持するにはvector<string>を使えばよいのです。問題は、それぞれの「分類」が表に何度も現われることです。たとえば、<noun>には3つの「ルール」が対応しています。<adjective>、<location>も同様です。これらの「分類」は3回現われ、それぞれ3つの「ルール」を持っているのです。

このように同じキーに複数の値が対応する場合のもっとも簡単な対処法は、それぞれの値（「ルール」）の集まりを、それぞれvectorにしてしまうというものです。つまり、今考えているプログラムのmapは、「分類」であるstringから「ルール」の集まりであるvectorへの対応をつけ、この「ルール」がさらにvector<string>というものです。

この型名は長いものです。そこで、中間の型名に別名を付けることで、プログラムは簡単になるでしょう。「ルール」がvector<string>であり、mapはstringから「ルール」のvectorへの対応を付けると書きました。結局、「ルール」(Rule)、「ルール」の集まり(Rule\_collection)、map(Grammar)の3つの型が必要だということです。

```
typedef vector<string> Rule;
typedef vector<Rule> Rule_collection;
typedef map<string, Rule_collection> Grammar;
```

#### 7.4.2 文法を読み込む

文法（文の構成方法）の表現の仕方がわかったので、読み込み関数を書いてみましょう。

```
// 与えられた入力ストリームから「文法」を読み込む
Grammar read_grammar(istream& in)
{
    Grammar ret;
    string line;
    // 入力を読む
    while (getline(in, line)) {
        // splitで入力行を単語に分解する
        vector<string> entry = split(line);
        if (!entry.empty())
            // 「分類」と対応する「ルール」を記録する
            ret[entry[0]].push_back(
                Rule(entry.begin() + 1, entry.end()));
    }
    return ret;
}
```

この関数は入力ストリームからデータを読み込んで、Grammarを生成し戻す関数です。whileループは今まで見えてきたものと同じ形をしています。まず、inから1行ずつ読み込んでそれをlineに格納しています。このwhileループは入力がなくなるか不適当なデータにぶつかると終了します。

whileの中身は驚くほど簡潔です。まず、§ 6.1.1のsplitを使い入力行を単語に分解し、それをentryという名前のvectorに格納します。そして、もしentryが空なら、空白の行なので、何もしません。そうでなければ、entryの最初の単語が「分類」であるわけです。

この要素をretのキーに使います。そして、ret[entry[0]]はentry[0]に対応するRule\_collectionオブジェクトを戻します。ここでRule\_collectionは要素がRule（これはvector<string>）であるvectorです。そのため、ret[entry[0]]はvectorであり、その最後尾に今読み込んだ「ルール」を付け加えるのです。ここで「ルール」は読み込んだ単語の最初が「分類」なので、2番目以降の単語です。ここでentryの一部（最初の要素以外）をコピーして新しく名前なしのRuleを作り、そのRuleをRule\_collectionであるret[entry[0]]に付け加えているのです。

### 7.4.3 文を作る

すべての入力を読み込んだら、ランダムに文を作らなければなりません。入力は「文法」でしたが、出力は文です。ここでは複数の文を作って `vector<string>` に格納し、それを出力しましょう。

この部分は簡単です。もっともおもしろいところは、この関数がどのように働くかです。まず、`<sentence>` に対応する「ルール」を見つけなければなりません。さらに、「ルール」や「ルール」の一部から文をばらばらに作らなければならないのです。

原理的には、文の断片をまとめて文にすることもできます。しかし、`vector` に、要素をまとめる機能は組み込まれていません。そこで空の `vector` を作り、`push_back` で順番に文の断片を付け加えていくことにします。

最初に `<sentence>` を使い、空の `vector` に対して `push_back` を使うことから、文の生成には、次のように補助関数が必要になってきます。

```
vector<string> gen_sentence(const Grammar& g)
{
    vector<string> ret;
    gen_aux(g, "<sentence>", ret);
    return ret;
}
```

補助関数である `gen_aux` は、「文法」 `g` から `<sentence>` の「ルール」にしたがい生成された文を `ret` に付け加える関数です。

結局、残りはこの `gen_aux` の定義です。それをする前に、読み込んだ単語が「分類」かどうかをチェックする判定関数を書きましょう。これは角カッコで囲まれているかどうかを判断するのです。

```
bool bracketed(const string& s)
{
    return s.size() > 1 && s[0] == '<' && s[s.size() - 1] == '>';
}
```

`gen_aux` は 2 番目の引数として与えられた入力 `string` を 1 番目の引数である「文法」に基づいて展開し、3 番目の引数に格納するのです。ここで「展開」とは、§ 7.4 で説明した処理のことです。もし、`string` に角カッコが付いていれば、それは「分類」なので、対応する「ルール」を見つけ、それにしたがって展開を進めます。もし、`string` に角カッコがなければ、それは単語なので、それ以上処理をせずにそのまま出力用の `vector` に付け加えます。

```
void
gen_aux(const Grammar& g, const string& word, vector<string>& ret)
{
    if (!bracketed(word)) {
        ret.push_back(word);
    } else {
        // word に対応する「ルール」を見つける
        Grammar::const_iterator it = g.find(word);
        if (it == g.end())
            throw logic_error("empty rule");
    }
}
```

### 7.4 文を作る

```
// 可能なルールをすべて取り出す
const Rule_collection& c = it->second;
// ここからルールを1つ選ぶ
const Rule& r = c[nrand(c.size())];
// 再帰的に「ルール」を展開する
for (Rule::const_iterator i = r.begin(); i != r.end(); ++i)
    gen_aux(g, *i, ret);
}
```

最初は簡単です。もし、2 番目の引数の `word` に角かっこがついていないければ、それ自身を処理する必要がないので、それを `ret` に付け加えて終わりです。そうでない場合が、おもしろいところです。まず、`g` の中で `word` の「分類」を探します。ここで単純に `g[word]` としてよいと思うかもしれません、これは誤った結論を導くかもしれません。§ 7.2 で説明した、存在しないキーを使ったときのことを思い出してください。その場合は、自動的にそのキーに対応する新しい要素が生成されてしまうのです。入力した「文法」に空の「ルール」を付け加えたくないでの、これは決して許されません。また、`g` は `const` な `map` ので、そのような「ルール」を付け加えたくてもできないのです。実際、`const` な `map` には `[]` が定義すらされていません。

明らかに別の方法が必要です。ここでメンバ関数の `find` は `map` の要素をキーで探し、もしそのキーの要素があればそれを指す反復子を戻します。また、`g` の中にどのような要素がなければ、`find` のアルゴリズムは `g.end()` を戻すことになります。そこで `it` と `g.end()` を比較することで、探している「ルール」があったかどうかを判定できるのです。もし、「ルール」が存在しなければ、入力された「文法」がおかしいということになります。つまり、対応する「ルール」がないのに、角カッコで囲んだ文字列があるからです。この場合は、例外を投げることにしました。

この時点で `it` は `map` である `g` の要素を指す反復子です。この反復子を使うと `pair` が得られ、`second` メンバが `map` に格納されたその要素の値であるわけです。つまり、`it -> second` がこの「分類」の「ルール」の集まりになります。また、コードを簡略化するために、このオブジェクトに対して、`c` という参照を定義します。これにより `c` がこのオブジェクトの別名になります。

次に、この「ルール」の集まりから、どれか 1 つをランダムに選ばなければなりません。これを `r` の初期化を行います。

```
const Rule& r = c[nrand(c.size())];
```

というコードは見慣れないものですが、よく見ておく価値があります。まず、`c` を `vector` である `Rule_collection` オブジェクトの参照にしたことに注意してください。`nrand` は § 7.4.4 で定義する関数で、`vector` の要素をランダムに選ぶために使っています。この `nrand` は引数に `n` を与えると、`[0, n)` の範囲のランダムな整数を戻す関数として定義する予定です。これにより、`r` はランダムに選ばれた「ルール」の別名になるわけです。

そして、`gen_aux` の最後の仕事は `r` の各要素を調べて処理するということです。もし、その要素に角カッコが付いていれば、またこれを展開しなければなりません。角カッコがなければ、それを `ret` に付け加えます。これを最初に読むときには魔法のように思えるかもしれません、今述べた処理はまさに `gen_aux` がする仕事そのものです。そのため、ここで `gen_aux` を呼べるのです！

このような関数の呼び出しを再帰 (recursive) といいます。これは何度か自分で実行してみないとわかりにく

いテクニックの1つです。この関数がきちんと動作するのを見るためには、まず `word` が角カッコなしの文字列である場合を考えてください。

次に、`word` が角カッコありの文字列、つまり「分類」である場合を考えます。ただし、その「ルール」の中にはもう角カッコ付きの文字列はないとします。この場合も、プログラムの動作を考えるのは難しくないと思います。というのは、`gen_aux` が再帰的に呼び出された場合にも、次の「ルール」には「分類」がないのすぐ終了するからです。この場合は、単純に `ret` に単語を追加して関数は終了するだけです。

さらに、`word` がもう少し複雑である場合を考えてみてください。右辺に角カッコ付きのものがあっても、それら自身の「ルール」には角カッコがないような場合です。この場合、最初の再帰的な呼び出しで何が起るか詳細に考えるのはやめましょう。そうではなく、この場合はうまくいくことを思い出すのです。なぜなら、そこで「分類」があっても、さらにそれ以上展開を続ける必要がないことは確かめたからです。再帰を繰り返すと、どんどん引数の `word` を単純なものにしていくので、結局、どんな場合でもうまくいくことがわかるかと思います。

私たちは、再帰を確実に説明する方法を知りません。今までの経験では、たいていの初心者は、再帰の動作を理解できず、再帰的なプログラムをただ見ているだけになります。それから、突然ある日、理解できてしまうのです。そして、そのときには、今までなぜこれが難しかったのだろうといぶかしがるものでした。再帰を理解する鍵は、ただ再帰を理解する所にあります。後は簡単です。

`gen_sentence`、`read_grammar` と補助的な関数を書いたので、それらを使ってプログラムが書けます。

```
int main()
{
    // 文を構成する
    vector<string> sentence = gen_sentence(read_grammar(cin));
    // もしあれば、最初の単語を書き出す
    vector<string>::const_iterator it = sentence.begin();
    if (!sentence.empty()) {
        cout << *it;
        ++it;
    }
    // 残りの単語を空白と交互に出力する
    while (it != sentence.end()) {
        cout << " " << *it;
        ++it;
    }
    cout << endl;
    return 0;
}
```

まず、「文法」を読み込み、それから文を生成し、1語ずつ出力して文にします。あまり重要な箇所ではありませんが、少し面倒なのは、2番目以降の単語を出力する前に空白を入れているところです。

#### 7.4.4 ランダムに要素を選ぶ

では `nrand` を書きましょう。まず、標準ライブラリには `rand (<cstdlib>に定義されている)` という関数があることから説明しましょう。この関数は、引数を取らず、`[0, RAND_MAX]` の範囲のランダムな整数を戻します。ここで `RAND_MAX` は `<cstdlib>` に定義されている大きな整数です。ここすべきことは、ランダムな整数の

範囲を `[0, RAND_MAX]` から、`[0, n]` にすることです。最初の範囲は 0 と `RAND_MAX` を含み、求めている範囲では 0 を含み `n` は含まないことに注意してください。また、`n <= RAND_MAX` とします。

ここで `rand() % n` で十分だと思うかもしれません。これは `rand()` を `n` で割った余りという意味です。実際に当たっては、この方法は2つの問題を持っています。

もっとも大きな問題は、実際的なものです。`rand()` は本当にランダムな数ではなく、ランダムに近い数（擬似乱数、pseudo-random-number）を戻すだけだということです。多くの C++コンパイラでは、割る数を小さくすると、余りはランダムでなくなります。たとえば、`rand()` が偶数と奇数を交互に戻すようになることもあります。そのような場合、`n` を 2 にすると、`rand() % n` は 0 と 1 を交互に戻すことになります。

さらに、`rand() % n` にはもう1つ微妙な問題があります。もし、`n` が大きいと `RAND_MAX` は `n` でうまく割れないのです。たとえば、`RAND_MAX` が 32767（規格を満たすコンパイラの `RAND_MAX` に許された最小の数）とし、`n` が 20000 とします。この場合、`rand() % n` を 10000 にする可能性が2つあります（`rand()` が 10000 か 30000 を戻す場合）。しかし、`rand() % n` が 15000 になる可能性は1つしかありません（`rand()` が 15000 の場合）。そのため、`rand() % n` が 10000 を戻す確率が 15000 を戻す確率の倍になってしまいます。

このような問題を避けるため、ここでは別の方法を採用します。それは、範囲を割って、同じ大きさのバケツの集まりにしてしまう方法です。それからランダムな整数を求め、バケツに対応する整数を戻すのです。バケツの大きさは同じで、どのバケツにも入らないランダムな整数もあります。その場合は、バケツに入る数が出るまでランダムな数を試し続けるのです。

この関数は説明をするより書いてしまった方が簡単です。

```
// [0, n)の範囲のランダムな整数を戻す
int nrand(int n)
{
    if (n <= 0 || n > RAND_MAX)
        throw domain_error("Argument to nrand is out of range");
    const int bucket_size = RAND_MAX / n;
    int r;
    do r = rand() / bucket_size;
    while (r >= n);
    return r;
}
```

整数の割り算は余り部分を切り捨てる所以、`bucket_size` の定義はその影響を受けます。これは `RAND_MAX/n` が割り算の正確な答えに等しいかそれより小さいということです。結果として、`bucket_size` は `n * bucket_size <= RAND_MAX` の範囲の最大の整数ということになります。

次は `do while`ステートメントです。これは `while` ステートメントに似ていますが、常にその中身の処理を最低1回は実行し、それから最後に条件をテストすることが違います。この条件が `true` ならループを繰り返し、条件が `false` になるとまで処理を実行します。今の場合、ループの中身は `r` をバケツの番号にセットするだけです。バケツ0は `rand()` の戻すランダムな数が `[0, bucket_size)` の範囲を意味し、バケツ1はランダムな数が `[bucket_size, bucket_size * 2)` の範囲を意味し…以下同様です。もし、`rand()` の戻す値が大きすぎて `r >= n` になった場合は、バケツのどれかに入るまでループを繰り返します。そして、適当な値が定まれば、この関数はその `r` を戻して終了します。

たとえば、`RAND_MAX` を 32767 とし、`n` を 20000 とします。そうすると、`bucket_size` は 1 になり、`nrand()`

は `rand()` が戻す 20000 以上の数は破棄し、20000 未満の数を戻すことになります。また、`n` を 3 にすると、`bucket_size` は 10922 になり、`rand()` の戻す値が [0, 10922) の範囲にある場合に 0 を戻し、[10922, 21844) の範囲にあるときは 1 を戻し、[21844, 32766) の範囲にあるときは 2 を戻します。32766 から 32767 の範囲のものは単に破棄されるだけです。

## 7.5 効率に関する注意

他の言語で連想配列を使ったことがあるなら、その配列の実装はハッシュ表 (hash table) とよばれるデータ構造であったかもしれません。ハッシュ表はとても速いのですが、その代償も持っています。

- キーのタイプによって誰かがハッシュ関数を書かなければならぬ。ハッシュ関数とはキーの値から適当な整数を計算する関数である。
- ハッシュ表の効率はハッシュ関数の詳細で大きく変わってしまう。
- ハッシュ表から便利な順番でデータを取り出す簡単な方法が普通はない。

C++の連想コンテナは以下の性質を持ち、ハッシュ表によって実装することは難しいのです。

- キーの型は<演算子か同等の比較関数を持つだけでよい。
- 与えられたキーの要素にアクセスする時間は、キーの値によらずコンテナ内の全要素数の対数に比例する。
- 連想コンテナの要素はいつもキーにしたがって整列させられている。

言い換えると、「C++の連想コンテナは、典型的な場合にはハッシュ表を使ったデータ構造よりやや遅いが、素朴なデータ構造よりはずっと速い。また、そのためにユーザがよいハッシュ関数を設計する必要はなく、自動で整列しているためハッシュ表より便利に使える」ということになります。もし、連想データ構造になじんでいたら、C++の連想コンテナの実装を知りたいと思うかもしれません。それは一般には自己調整型の平衡ツリー (balanced self-adjusting tree) 構造です。

いろいろな C++処理系がハッシュ表を提供していますが、標準ライブラリの規格には入っていません。そのため、これはこの本の範囲を超ってしまいます。一方、標準的なものがすべての目的にかなうわけではありませんが、標準の連想コンテナは大抵のアプリケーションには十分以上の働きをします。

## 7.6 詳細

`do while` ステートメント: `while` ステートメント (§ 2.3.1) に似ているが、条件のテストは最後に行う。一般的な形式は次のようになる。

```
do statement
  while (condition);
```

始めにステートメント `statement` が実行され、以後は条件 `condition` のテストとステートメントが、条件が `false` になるまで交互に実行される。

## 7.6 詳細

値初期化: 存在しない `map` の要素にアクセスすると値が `V()` で初期化された要素が生成される。ここで `V` は `map` に格納される値とする。このようなエクスプレッションは値初期化とよばれる。この詳細は § 9.5 で説明する。もっとも重要なことは、組み込み型は 0 に初期化されるということである。

`rand()`: [0, RAND\_MAX] の範囲のランダムな整数を戻す関数。`rand` も `RAND_MAX` も <cstdlib> で定義されている。

`pair<K, V>`: 2 つの値を保持する単純なデータ構造。この要素にアクセスするには、その名前 `first` と `second` を使う。

`map<K, V>`: キーの型が `K`、値の型が `V` の連想コンテナ。`map` の要素はキーと値のペアになっている。また、キーによる効率的なアクセスを可能にするため、要素はキーにしたがって並べられている。`map` の反復子は双方指向型 (§ 8.2.5)。反復子を使って取り出す要素の型は `pair<const K, V>` である。`map` の操作には次のようなものがある。

```
map<K, V> m;
  キーの型が const K で値の型が V である空の map を生成する。
map<K, V> m(cmp);
  キーの型が const K で値の型が V である空の map を生成する。ただし、キーの順番は判定関数の cmp で決められる。
m[k]
  型 K のキー k を使って map にアクセスする。これは型が V の左辺値を戻す。もし、そのキーに対応する要素がなければ、値初期化を使って新しい要素が生成され map に挿入される。[] を使うと新しい要素を生成するかもしれない const map では使用できない。
m.begin()
m.end() map の要素にアクセスするための反復子を戻す。反復子を使って要素を取り出すと、それは値だけではなくキーと値のペアになることに注意。
m.find(k)
  キーが k である要素を指す反復子を戻す。ただし、そのような要素がない場合は m.end() を戻す。
map<K, V> とそれに伴う反復子 p に対し、次のようなことができる。
p->first  const K 型の左辺値である、p が指す要素のキーを戻す。
p->second  V 型の左辺値である、p が指す要素の値を戻す。
```

## 課題

7-0 この章のプログラムをコンパイルし、実行し、試してみてください。

7-1 § 7.2 のプログラムを改良して、出力がカウントの少ない順になるようにしてください。つまり、まず、1 回だけ現われた単語をすべて出力し、次に 2 回現われた単語をすべて出力し、としていくものです。

7-2 § 4.2.3 のプログラムを、学生の点数にしたがって、A、B、C、D、E、F を付けるものに改良してください。これらの対応は次のようにします。