

第5章

シーケンシャルコンテナと string の解析

ここまで、C++言語そのものを学ぶのよいスタートだったと思います。また、string と vector も学びました。これらを使うだけで多くの問題を解決することができます。

この章では、興味の中心をその先に向け、ライブラリの使用についてもう少し深く考えていくことにします。ライブラリのツールを使うと、これまで以上に面倒な問題を解決できることがわかるでしょう。

標準ライブラリは有用なデータ構造や関数を提供するだけでなく、一貫したアーキテクチャー（構造）を持っています。あるコンテナの振る舞いを理解したなら、別のコンテナを理解することも容易になるのです。

たとえば、この章の後半で見ますが、string をあたかも vector のように扱うこともできるのです。このようにライブラリのある型の多くの有用な操作が、論理的には他の型の操作と同じになるのです。ライブラリは、異なる型に対しても、相当する操作は同様にできるよう設計されているからです。

5.1 学生をカテゴリに分類する

学生に成績をつける問題（§ 4.2）をもう一度考えてみましょう。ただし、今度は最終成績をつけるだけでなく、どの学生が不合格になったかを判定するようにしたいと思います。まず、Student_info の vector に学生のデータがあると仮定し、そこから不合格の学生を抜き出し、別の vector に記録するとします。また、不合格の学生はとの vector から削除し、この vector は合格者だけの記録にします。

次に示す、学生が不合格かどうかを判定する簡単な関数から始めます。

```
// 学生が不合格かどうかを判定する関数
bool fgrade(const Student_info& s)
{
    return grade(s) < 60;
}
```

ここで成績を計算するに § 4.2.2 の grade 関数を使い、60 点未満は不合格としました。

望みのコードを書くのに、もっともストレートなやり方は、合格者用の vector と不合格者用の vector を作っておいて、学生 1 人ひとりのデータをチェックしながら、それをどちらかに振り分けていくというものです。

```
// 最初の案：合格者 (pass) と不合格者 (fail) を分ける
vector<Student_info> extract_fails(vector<Student_info>& students)
{
    vector<Student_info> pass, fail;
    for (vector<Student_info>::size_type i = 0;
```

```

    i != students.size(); ++i)
    if (fgrade(students[i]))
        fail.push_back(students[i]);
    else
        pass.push_back(students[i]);
    students = pass;
    return fail;
}

```

もちろん、コンパイルするためには、関数内で使う名前のための #include ディレクティブと using ユニオンを書く必要があります。今後は、そのようなコードは明示的に書かないことにします。ただし、新しいヘッダを使う必要があります。

第4章の read_hw や read 関数のように、この関数は実質的に2つの出力をしています。1つは戻り値で、もう1つは、これは不合格の学生が記録されています。もう1つの出力は、 extract_fails 関数の副作用です。この関数のパラメータは参照なので、これを変えると引数も変わってしまいます。

この関数では、合格者と不合格者を記録するために、2つの vector を作ります。これら2つの vector である。関数の終了後は、引数に使われた vector には合格者だけが記録されています。

この関数では、合格者と不合格者を記録するために、2つの vector を作ります。これら2つの vector である。関数の終了後は、引数に使われた vector には合格者だけが記録されています。

5.1.1 要素を削除する

上の extract_fails 関数は、期待通りの仕事をし、そこそこに効率的ですが、小さな問題を持っています。それは、学生の記録を2つ保持するために十分なメモリを要求するということです。理由は次のようなものです。まず、pass と fail を作ったときに、との vector もあります。そして、関数内の for 文が実行され、結果がコピーされ戻される前には、との学生の記録が2箇所に存在することになるのです。

同じデータのコピーを複数持つことはできる限り避けたいものです。このために考えられる1つの方法は、 pass をまったく使わないことです。つまり、2つの vector を生成するのではなく、戻り値用に1つだけ fail という名前の vector を作ります。そして、students 内の学生1人ひとりのデータに対し、成績を計算し、もしその学生が合格なら何もせず、不合格なら fail に付け加えて students から取り除くのです。

このためには、vector からデータを削除する方法が必要です。幸いそのような仕組みはあります。ただし、残念ながら、vector からデータを削除するのには時間がかかるので、入力データが多い場合もこの方法でよいと考える必要があります。もし、データが非常に多い場合は、効率が驚くほど悪くなるのです。

たとえば、すべての学生が不合格の場合、この関数の実行時間は生徒数の2乗に比例して大きくなるでしょう。これは100人の生徒なら1人の場合より10000倍の処理時間が必要になるということです。問題は、生徒のデータを vector で保持しているという点にあります。vector は各要素へのランダムなアクセスが効率的に作られています。このため、vector の最後ではなく、途中でデータを挿入したり削除する効率が悪くなっているのです。

この効率の問題を解決するのに、あとで2つの方法を見ます。1つは、このようなアルゴリズムに向いている別のデータ構造を使うというものです。もう1つは、このような余分な時間を使わないで、よりよいアルゴリズムを考えるというものです。§ 5.5.2まではもっとよいデータ構造を使う方法を考え、別のアルゴリズムは

5.1 学生をカテゴリに分類する

§ 6.3 で考えることにします。

しかし、改良案がなぜよいか知るために、まず今的方法で関数を完成させておく必要があります。したがって、まず遅いけれどストレートな方法を見ていきましょう。

```

// 2番目の案：正しいけれど遅い
vector<Student_info> extract_fails(vector<Student_info>& students)
{
    vector<Student_info> fail;
    vector<Student_info>::size_type i = 0;
    // 不変な表明：[0, i)の要素は合格した学生の成績のみ
    while (i != students.size()) {
        if (fgrade(students[i])) {
            fail.push_back(students[i]);
            students.erase(students.begin() + i);
        } else
            ++i;
    }
    return fail;
}

```

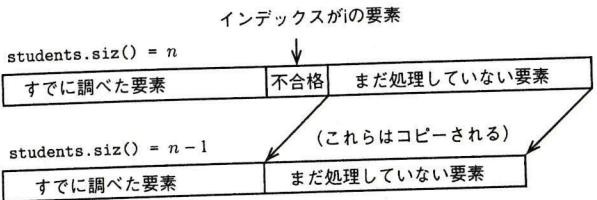
まず、不合格の学生のデータを保持するために fail という vector を生成しています。それから i という変数を用意し、students のインデックスに使用します。students の要素を最後まで順番に処理していきます。

students の要素1つひとつについて合否を調べ、不合格ならそのデータを fail に入れ、students からは削除していきます。push_back はすでに説明しましたが、students[i] のデータのコピーを fail に追加しているのです。students から要素を削除している部分が新しいところです。

```
students.erase(students.begin() + i);
```

vector 型は erase というメンバ関数を持っているのですが、これは vector の要素を削除する関数です。erase の引数がどの要素を削除するかを表しているのですが、これはインデックスではありません。erase 関数には、インデックスによって削除する要素を指定する方法はないのです。詳しくは § 5.5 で説明しますが、ライブラリはすべてのコンテナに同様に働く erase を提供しています。すべてのコンテナにインデックスというものはないので、インデックスを使わないと便利なのです。その代わりに、§ 5.2.1 で詳しく説明する型の引数を取ります。ここでは、あるインデックスの要素を削除したい場合には、そのインデックスに students.begin() の戻り値を足したものを持てばよい、ということだけ理解してください。前に説明ましたが、students.begin() の戻り値は vector の最初の要素（インデックスが0のもの）を表します。そこに i を足せば、それはインデックスが i の要素を表すことになるのです。結局、上のコードの erase は students の第 i 番の要素を削除しているわけです。

```
vector から要素を1つ削除すると、vector の全要素数は1つ減ります。
```



`erase` は `vector` の要素数を 1 つ減らすだけでなく、削除した要素の次の要素をインデックス `i` にし、それ以後の要素を前にずらします。そのため、`i` を 1 つ増やさなくても、`erase` の後はこれが削除した要素の次の要素を指すようになっています。そのため、次のループのステップのために、`i` を増やしてはいけないので。

もし、調べている学生のデータが不合格でなければ、これはそのまま `students` に残したいわけです。その場合は、`while` での次のステップにいくために、`i` を 1 つ増やさなければなりません。

`students` の全データを調べたかどうかは、`i` と `students.size()` を比べて判定しています。`vector` の要素を削除すると、その全要素数が 1 つ減ります。そのため、毎回の判定に `students.size()` を使うことは重要です。たとえば、次のように `size` という変数を作り、そこにサイズを記録したとします。

```
// 誤解に基づく「改良」のため、以下のコードは間違います。
vector<Student_info>::size_type size = students.size();
while (i != size) {
    if (fgrade(students[i])) {
        fail.push_back(students[i]);
        students.erase(students.begin() + i);
    } else
        ++i;
}
```

これは間違います。なぜなら、`erase` を呼び出すたびに `students` の全データ数が変わってしまうからです。このように `size` に最初のデータ数を記録し、1 つでもデータを削除すると、`students` に対して多すぎる処理をしてしまうのです。その結果、`students[i]` は存在しないデータを意味することになります。幸い、`size()` 関数の呼び出しには普通あまり時間がかかりませんので、この呼び出しのオーバーヘッドは無視できるものです。

5.1.2 シーケンシャルアクセス対ランダムアクセス

ここまで作った 2 つの `extract_fails` 関数は、コンテナを使う多くのプログラムに共通する性質を持っていました。それはコードを見るだけではあまり明らかではありませんが、これらの関数はコンテナの要素にシーケンシャルに（順番に）アクセスしているだけということです。つまり、どちらの関数も、学生のデータを 1 つひとつ最初から順番に調べていき、何をするか判断しているということです。

これがあまり明らかでないのは、`students` の要素にアクセスするのに、インデックス `i` を使っているからです。整数 `i` の値はいろいろ変わられるのですが、上のコードでは、コンテナのデータにシーケンシャルにアクセスするように、注意して `i` の値を変えてています。しかし、`students[i]` というアクセス方法は、暗に、シーケンシャルとは限らずに、いろいろな順番でデータにアクセスする可能性があることを示唆しています。

コンテナの要素にシーケンシャルにアクセスするかどうかを気にする理由は、コンテナによっては異なる効率性と操作性を持っているからです。もし、特定のコンテナで効率的な操作を自分のプログラムで使うのがわかっているのなら、そのコンテナを使うことでより効率的なプログラムが書けます。

言葉を換えれば、私たちの関数はシーケンシャルなアクセスしかしていないので、インデックスを使う必要はないということです。インデックスはいろいろな要素にランダムに（自由な順番で）アクセスするためにあるのです。そこで、これからこの関数を、シーケンシャルなアクセスしか許さない方法を使って書き換えてみます。そのために、C++には反復子（イテレータ）と呼ばれる型があります。これを使うことで、データへのアクセスの仕方が、ライブラリに管理できるようになります。この管理のおかげで、ライブラリは効率がよくなっています。

5.2 反復子

話を具体的にするために、`extract_fails` で実際に使っているコンテナについて考えてみましょう。

最初の操作は、インデックス `i` を使って、`Student_info` オブジェクトのデータを取り出すというものです。たとえば、`fgrade(students[i])` とあれば、これは `students` という `vector` オブジェクトの第 `i` 番の要素を `fgrade` という関数の引数にしているのです。しかし、`vector` の要素にはシーケンシャルにしかアクセスしていません。というのは、これらの要素にはインデックス `i` を使ってアクセスし、`i` に対して行っている操作は、`vector` のサイズとの比較と、インクリメント（1 増やすこと）だけだからです。

```
while (i != students.size()) {
    // ここで作業があるが i の値は変えない
    ++i;
}
```

このような使い方から、コンテナをシーケンシャルにしか使っていないことは明らかです。

しかし、残念ながら、私たちにそれがわかつても、ライブラリにはわかりません。それで、インデックスではなく反復子（イテレータ、iterator）を使うことで、ライブラリに知らせることができます。反復子とは次のようなものです。

- コンテナとコンテナ内の 1 つの要素を指す
- その要素の値を取り出しができる
- そのコンテナ内を移動し別の要素を指すことができる
- そのコンテナが効率的に処理できるように操作を限定できる

反復子はインデックスのように振る舞うので、しばしば、インデックスではなく反復子を使ってプログラムに書き換えることも可能です。たとえば、学生データを保持する `students` は `vector<Student_info>`ですが、学生成名を `cout` に出力するコードの書き換えを見てみましょう。まず、インデックスを使うと次のように書けるでしょう。

```
for (vector<Student_info>::size_type i = 0;
     i != students.size(); ++i)
    cout << students[i].name << endl;
```

80

これを反復子を使って書き直すと次のようになります。

```
for (vector<Student_info>::const_iterator iter = students.begin();
      iter != students.end(); ++iter) {
    cout << (*iter).name << endl;
}
```

この書き換えをきちんと説明するには、たくさんすべきことがあります。それらは、以下で見ていくことにします。

5.2.1 反復子のタイプ

`vector` のような標準ライブラリのコンテナには、次の 2 つの型の反復子があります。

コンテナの型::const_iterator
コンテナの型::iterator

ここで「コンテナの型」は、特定の型の要素を複数保持する `vector<Student_info>` のようなコンテナの型を表します。そして、コンテナ内の要素の値を変えたい場合は `iterator` タイプを使い、読み込みアクセスだけのときは `const_iterator` タイプを使います。

抽象化とは選択的な無視のことです。特定の反復子のタイプの詳細は複雑かもしれません、それらをすべて理解する必要はないのです。理解する必要があるのは、各要素を反復子でどのように表すかと、その反復子でどのような操作ができるか、だけです。型の実装^{*1}については何も知らないよいのです。例えば、`iter` を次のように定義したとします。

```
vector<Student_info>::const_iterator iter = students.begin();
```

これは、`iter` が `vector<Student_info>::const_iterator` 型だということです。そして、実際に `iter` が何を指しているかは知らないし、知る必要もないのです。知る必要のあることは、「`vector<Student_info>` が、それを使えば読み込み専用として `vector` の要素にアクセスすることができる `const_iterator` という型を定義している」ということです。

もう 1 つ知っておくべきことは、`iterator` 型から `const_iterator` 型への自動の変換があるということです。後で説明しますが、`students.begin()` の戻す型は `iterator` です。しかし、上では `iter` を `const_iterator` にしました。`iter` を `students.begin()` で初期化するために、コンパイラが `iter` を `const_iterator` に変換するのです。この変換は一方通行で、`iterator` を `const_iterator` に変換することができますが、その逆変換はできません。

5.2.2 反復子の操作

`iter` を定義して、その値を `students.begin()` に初期化しました。実は、この `begin` と `end` はすでに使っている、どんな関数かは説明してあります。つまり、コンテナの先頭と末尾（最後の要素の 1 つ後）を戻す関数です。何度も繰り返した「(最後の要素の 1 つ後)」の説明は、§ 8.2.7 でします。ここで注意してほしい

*1 訳注：本書では「実際の定義」という意味で実装（implementation）という言葉を使います。

ことは、`begin` と `end` の戻す値は、コンテナの反復子の型だということです。そのため、`begin` の戻す値は、`vector<Student_info>::iterator` 型で、このコンテナの最初の位置を示すものなのです。つまり、`students` の最初の要素を指すわけです。

for 文の条件は

```
iter != students.end()
```

でした。これは `iter` がコンテナの最後に達したかどうかをチェックしているのです。ここで `end` はコンテナの末尾（最後の要素の 1 つ後）を指すことを思い出してください。`begin` と同じく `end` の戻す値の型も `<Student_info>::iterator` です。反復子は `const` 付きでもそうでなくとも、等しくないか（あるいは、等しいか）どうかをチェックすることができます。そして、`iter` が `students.end()` に等しい場合は、`for` 文の処理が最後まで行ったということです。

`for` ヘッダの最後には `++iter` というエクスプレッションがあります。これは `for` 文の次の処理のため、`iter` を 1 つ先に進ませて `students` の次の要素を指すようにするのです。ここで `++` というインクリメント演算子を使っていますが、これは反復子用にオーバーロードされているのです。つまり、インクリメント演算子はコンテナ内で反復子の指す要素を 1 つ進めるものです。この演算子が具体的にどのように働くか知りませんし、知る必要もないのです。私たちが知るべきことは、これで反復子がコンテナの次の要素を指すようになる、ということだけです。

`for` 文の中身（処理）では、`iter` は出力したい `students` の要素の位置を表します。その要素自身はデリファレンス（dereference）演算子とよばれる * を使って表しています。反復子に * を作用させると、それはその反復子の指す要素（左辺値）を戻すのです。したがって、

```
cout << (*iter).name
```

すると、`iter` が現在指している要素の `name` を標準出力に書くことになるのです。

ここでコードが正しく作用するように、丸カッコを使いました。これにより演算子の実行順序を指定することができるのです。`*iter` は `iter` の指す値を表し（戻し）ます。ピリオド「.」の優先度は * より高いので、 * . の前に作用させるためには、`*iter` を `(*iter)` のように丸カッコでくくらなければなりません。逆に、`iter.name` と書けば、`*(iter.name)` として扱われます。これは、`iter` オブジェクトのメンバ `name` を取り出し、そのオブジェクトをデリファレンスするという意味になります。しかし、`iter` に `name` というメンバはないので、コンパイラはエラーと判断するでしょう。`(*iter).name` と書くことで、`name` は `*iter` オブジェクトのメンバであると解釈されるのです。

5.2.3 簡単な書き方

今見たコードでは、反復子の指すオブジェクトを取り出し、そのメンバを使いました。このような操作はよく行われるので、短縮形があります。たとえば、

```
(*iter).name
```

```
iter->name
```

と書けます。この書き方を使うと、§ 5.2 の最後の例は次のようになります。

```
for (vector<Student_info>::const_iterator iter = students.begin();
     iter != students.end(); ++iter) {
    cout << iter->name << endl;
}
```

5.2.4 students.erase(students.begin() + i) の意味

もう少し反復子について学んでみましょう。反復子の重要なポイントは、§ 5.1.1 示したプログラムの次のエクスプレッションにあります。

```
students.erase(students.begin() + i);
```

これまでのなかで、`students.begin()` は `students` の最初の要素を表す反復子を戻し、`students.begin() + i` は `students` の第 *i* 番の要素を表すのを見ました。ここでこのエクスプレッションが意味を持つのは、`students.begin()` と *i* の間に+という演算子が定義されているからです。逆に、反復子とインデックス型から、+の意味が決められるとも言えます。

`students` がランダムアクセス用のインデックスを使えないコンテナである場合には、`students.begin()` は+演算ができるず、コンパイラはエラーとするでしょう。結果的に、そのようなコンテナはランダムアクセスを禁止していても、反復子によるシーケンシャルアクセスは許すのです。

5.3 インデックスの代わりに反復子を使う

これまで見てきた反復子の性質ともう 1 つ新しいことを使って、インデックスなしで `extract_fail` を書き直すことができます。

```
// バージョン3: 反復子を使いインデックスは使わない。まだ効率は悪い
vector<Student_info> extract_fails(vector<Student_info>& students)
{
    vector<Student_info> fail;
    vector<Student_info>::iterator iter = students.begin();
    while (iter != students.end()) {
        if (fgrade(*iter)) {
            fail.push_back(*iter);
            iter = students.erase(iter);
        } else
            ++iter;
    }
    return fail;
}
```

前の関数と同様に、まず `fail` を定義しています。それから、インデックスの代わりに `students` の要素を指示するために反復子 `iter` を定義しています。ここで反復子のタイプが `const_iterator` ではなく `iterator` であることに注意してください。

5.3 インデックスの代わりに反復子を使う

```
vector<Student_info>::iterator iter = students.begin();
```

これは `students` を変える目的で使うからです。実際、これから `erase` を使いデータを削除していきます。そして `iter` の値は `students` の最初の要素を指すように初期化しています。

次に `while` 文があります。これで `students` の各要素を調べていきます。`iter` はコンテナの要素を指す反復子ですから、その要素自身は`*iter` で表されます。学生が合格か不合格かを決めるに、そのデータを `fgrade` に渡しました。同様に、不合格者のデータを `fail` に追加するには、

```
fail.push_back(*iter); // 反復子からオブジェクトを得るのにデリファレンスを使った  
としました。これは、
```

```
fail.push_back(students[i]); // オブジェクトを得るのにインデックスを使った  
を書き換えたものです。反復子を直接使っているから erase は簡単になります。
```

```
iter = students.erase(iter);
```

前のように `students.begin()` にインデックス *i* を足すような計算をしないでよいのです。

ここで登場した新しいことは、簡単に見落としてしまいそうですが、非常に重要なことです。それは、`erase` の戻り値を `iter` に格納している点です。これはなぜ重要なのでしょうか？

少し考えてみると、`iter` の指す要素を削除した後の `iter` は無効 (invalid) であることがわかると思います。`students.erase(iter)` を呼んだ後は、`iter` の指すものは削除されているからです。事実、`vector` で `erase` を使うと、その削除した要素以降の要素を指すすべての反復子が無効になります。§ 5.1.1 の図を見てください。そこで不合格と書いた要素を削除すると、その後の要素も動かされていることがわかります。要素が動いてしまうので、それらを指す反復子は無効になってしまいます。

しかし、`erase` は削除した要素の次の要素を指す反復子を戻すようにできているのです。そのため、

```
iter = students.erase(iter);
```

すると、`iter` は削除した要素の次の要素を指すようになります。これはまさに `while` 内で必要なことでした。調べている学生が合格であった場合でも、`while` ループのステップを進めるため、反復子 `iter` の指す要素を一つ進める必要があります。そのため、`else` のブロックでは、`iter` を 1 つ進めています。

ところで、§ 5.1.1 と同様に、`students.end()` の値を保持し `while` の条件のところで毎回 `end` を呼び出すのを止めて、効率をよくしようと考えるかもしれません。つまり、

```
while (iter != students.end())
```

を次のように変えたいかもしれないということです。

```
// このコードは誤った効率化による間違い
vector<Student_info>::iterator iter = students.begin(),
                                end_iter = students.end();
while (iter != end_iter) {
    // ...
}
```

このコードはたぶん実行時エラーを出します。どうしてでしょう。

それは、`students.erase` を実行するたびに、削除した要素の後の要素を指す反復子を無効にしているからです。このため `end_iter` も無効になります！§ 5.1.1 のコードでは `students.size` をループの処理ごとに呼び出す必要があったように、今回は、`students.end` を呼び出す必要があるのです。

5.4 効率をよくするためにデータ構造を再検討する

入力データの数が少ない場合は、これまでのコードで問題はありません。しかし、§ 5.1.1 でも述べましたが、データの数が大きくなると効率はとても悪くなります。

ここで `erase` が `vector` から要素を 1 つ削除する方法について考え直してみましょう。ライブラリの制作者は、ランダムアクセスが最大限に効率よくなるように、`vector` の構造を決めています。また、§ 3.2.3 で見たように、`vector` に要素を 1 つずつ付け加えていくのも、最後の要素として付け加えている限り効率はよいものです。

しかし、要素を途中で挿入したり、削除するのは別の話です。効率のよいランダムアクセスを維持するため、挿入や削除のたびに、後ろの要素を再配置する必要があるからです。ここまで見てきたコードで要素を削除するのにかかる時間は、`vector` の要素数の 2 乗に比例します。これはデータ数が少ない場合には問題になりませんが、データ数が増えれば、その 2 乗に比例して実行時間がかかるようになるということです。1 つの組の学生ではなく学校全体の学生を扱うようになると、高速なコンピュータでもプログラムの実行に時間を取るようになります。

これを改良するためには、コンテナのどの場所での挿入・削除も、効率よく実行されるデータ構造が必要になるのです。しかし、そのようなコンテナでは、インデックスを使ったランダムアクセスはできないでしょう。もし、仮にできたとしても、データの挿入と削除をするなら、整数のインデックスはあまり有用ではなくなるはずです。私たちは、反復子の使い方を覚ましたので、このようなインデックスのないデータ構造でも扱えるようになりました。

5.5 list 型

コードを反復子を使って書き直し、インデックスを除きました。次に、別のデータ構造、つまり要素の削除が効率的なコンテナを使ってプログラムを書き直すことにしましょう。

データ構造にデータ（要素）を挿入したり削除するということは、よくあることです。そのため、当然、ライブラリもそのためのものを用意しているのです。そのような操作を効率的に行うデータ構造は `list` と呼ばれ、`<list>` ヘッダに定義されています。

`vector` はランダムアクセスを効率的に行うコンテナですが、`list` はデータの挿入と削除を効率的に行うコンテナなのです。`list` は複雑な構造をしているので、シーケンシャル（逐次的）にデータにアクセスするにしても、`vector` より効率は悪くなります。つまり、データの追加や削除が末尾だけかほとんど末尾だけで起こるなら、`vector` は `list` より効率がよいのです。しかし、データが多く、今考えているプログラムのようにたくさんデータをコンテナの中ほどから削除するなら、`list` の方が効率的になるでしょう。これはデータ数が多いほど差が開いてきます。

`vector` と同じように、`list` もほとんどすべての型のオブジェクトを保持できるコンテナです。そして、以下

5.5 list 型

で見ますが、`list` と `vector` では多くの操作が共通しています。そのため、しばしば `vector` を使ったプログラムを `list` を使ったプログラムに変更したり、その逆をすることができます。そして、大抵の場合、変えるべきところは変数の型だけなのです。

`vector` にあって `list` にないもっと重要なものの 1 つがインデックスです。今見たように、不合格の学生データを抜き出す `extract_fails` 関数は、`vector` を使いながらインデックスの代わりに反復子を使っています。このため、型を適切に直すだけで、`vector` ではなく `list` を使うように `extract_fails` を書き換えることができるのです。

```
// バージョン4: vectorではなくlistを使う
list<Student_info> extract_fails(list<Student_info>& students)
{
    list<Student_info> fail;
    list<Student_info>::iterator iter = students.begin();
    while (iter != students.end()) {
        if (fgrade(*iter)) {
            fail.push_back(*iter);
            iter = students.erase(iter);
        } else
            ++iter;
    }
    return fail;
}
```

このコードを § 5.3 のものと比べてみると、その違いのすべては、最初の行の `vector` を `list` に書き換えるためだけだということがわかると思います。たとえば、戻り値の型とパラメータの型が `list<Student_info>` であるため、不合格者のデータを格納する関数内のコンテナ `fail` も `list<Student_info>` にしてあります。また、同様に、反復子も `list` に対するものに変えました。つまり、`iter` は `list<Student_info>` の `iterator` のことです。`list` はテンプレートであるため、どの型のオブジェクトを格納するのかを角カッコで示さなければならぬことは `vector` のときと同じです。

プログラムの論理には何の変更もありません。もちろん、この関数を使うには、引数に `list` オブジェクトを渡さなければなりませんし、戻り値も `list` になるという変更はあります。さらに、`vector` から `list` を使うように変えたため、ライブラリ内の実際のコードはまったく異なるものになっています。しかし、`++iter` という操作をすると、それは何にせよ、`list` 内で `iter` の指す要素を 1 つ進めるという意味になります。同様に、

```
iter = students.erase(iter);
```

は、`list` の `erase` を呼び出し、`list` の反復子を戻し `iter` に代入しています。もちろん、インクリメントと `erase` の中身（実装）は `vector` のときとまったく違うものであるはずです。

5.5.1 いくつかの重要な違い

`list` を使った方法が `vector` と違うことで重要なことの 1 つは、いくつかの反復子に対する操作の効果が異なるということです。たとえば、`vector` に対して `erase` を使った場合、削除された要素およびその後の要素を指す反復子はすべて無効にされます。`push_back` を使ってデータを最後尾に追加したときも全反復子が無効になります。これは、要素の削除によりその後ろのオブジェクトが前にずらされるから、また、要素の追加の

際には新しい要素を付け加えるために全要素を別の場所に移すかもしれないからです。したがって、このような操作をするループでは、無効になるかもしれない反復子のコピーを利用しないように、とても気をつけなければならぬのです。特に `studetns.end()` の値をコピーして使うことは多くのバグのもとになります。

一方、`list` に関しては、`erase` や `push_back` は他の要素を指す反復子を無効にしません。無効になる反復子は、実際に削除される要素を指す反復子だけです。

ここまで `list` クラス反復子はランダムアクセスの機能をサポートしないという話をしました。§ 8.2.1 では、反復子についてさらに詳細な話をします。ここでは、この性質のため、`list` の要素のソート（整列）のために標準ライブラリの `sort` 関数を使うことはできないということだけ覚えてください。このため、`list` には `sort` というメンバ関数があります。これは `list` のデータをソートするのに最適なアルゴリズムを使っています。そのため、`list` のデータをソートするには、次のようにするのです。

```
list<Student_info> students;
students.sort(compare);
```

これは次のような `vector` の場合と違います。

```
vector<Student_info> students;
sort(students.begin(), students.end(), compare);
```

なお、`compare` 関数は `Student_info` オブジェクトに作用する関数なので、`vector` だけでなく `list` であってもその内部の `Student_info` オブジェクトをソートするのに使えるのです。

5.5.2 なぜ、気にするのか？

不合格者のデータを取り出す関数は、データ構造の選択が効率にどう影響するかを見るよい例です。このコードは要素にシーケンシャル（順番に）にアクセスするので、それだけ見れば、`vector` がもっとよい選択です。しかし、要素をコンテナから削除する部分があるので、これは `list` がよいということになります。

効率の問題ではいつもそうなのですが、効率をどのように取り扱うかがデータ構造の選択に影響します。効率の問題は難しいので、一般的にはこの本の範囲外になりますが、データ構造の選択がプログラムの効率を非常に変えることがある、ということは覚えておくとよいでしょう。入力データ数が少ない場合は、`list` を使った方が `vector` を使うより遅くなります。逆に、データ数が多いと、`vector` を不適当に使ったものは `list` を使うものよりも遅くなります。データ数の増加に伴って効率がどう変化するかは驚くほどのものがあります。

実際にプログラムの効率を見るため、学生のデータを記録した 3 つのファイルを使って実験してみました。最初のファイルには 735 個のデータがあり、次のファイルにはその 10 倍のデータが、また最後のファイルにはさらにその 10 倍のデータ、つまり 73,500 のデータがあるものにしました。次の表は、それぞれのファイルを処理するのにかかった時間を秒の単位で示したものです。

ファイルサイズ	list	vector
735	0.1	0.1
7,350	0.8	6.7
73,500	8.8	597.1

データ数が 73,500 個の場合、`list` を使ったプログラムでは 9 秒もかかるのに、`vector` を使ったものは 10 分近くかかっています。不合格の学生数が増えると、この差はもっと大きくなっていくでしょう。

5.6 string オブジェクトを分ける

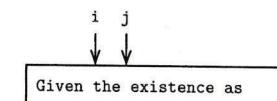
ここまでにコンテナで何ができるかを見てきました。ここでまた `string` を考えることにしましょう。`string` の操作はそれほど紹介していません。`string` オブジェクト（文字列）の生成、それらの連結、書き換え、そのサイズの見方、を扱っただけです。どの操作も、`string` オブジェクトを 1 つのデータとして扱ってきました。このような抽象的な扱いはしばしば望ましいものです。つまり、大抵は文字列の詳細を知る必要はないのです。しかし、文字列の中の特定の文字に着目したいときもあります。

実は、`string` は特別なタイプのコンテナであることがわかります。これは文字だけを格納し、普通のコンテナの全部ではありませんがいくつかの操作が可能なものです。このコンテナとしてできる操作には、インデックスを使うことも含まれます。また、`vector` もそうであるように反復子を使うこともできます。つまり、`vector` にできる多くのことが `string` にもできるのです。

たとえば、1 行の文字列を空白（スペース、タブ、バックスペース、行の終わり）で区切られたデータ^{*2}に分けたいと考えるかもしれません。このような単語を直接受け取るなら、分割された単語を取り出すのは簡単です。それは、自動で `string` への読み込みで行われるからです。つまり、`string` への入力演算では、空白のところまでの単語が読み込まれるのであります。しかし、まず空白も含めた 1 行を読み込んで、その 1 行を空白で分けられた単語に分割したいときもあるでしょう。そのような例は § 7.3 と § 7.4.2 で見ます。

そのような操作はいろいろ便利だと思われる所以関数にしましょう。この関数は、`string` オブジェクトを引数に取り、`vector<string>` を戻すのです。この戻り値には、引数の文字列の中で空白で分けられる単語の 1 つひとつが格納されることになります。この関数を理解するために、`string` には `vector` とよく似たインデックスがあることを説明しましょう。たとえば、`s` が 1 文字以上の文字を含む `string` オブジェクトであるとき、`s` の最初の文字は `s[0]` で表されます。また、最後の文字は `s[s.size() - 1]` です。

今考えている関数では、2 つのインデックス `i` と `j` を使います。これは単語（空白で区切られたデータのことでした）の 1 つひとつを順に示していくためのものです。つまり、1 つの単語が `i` と `j` で `[i, j)` のように表されるようにしたいのです。たとえば、



このようなインデックスがあれば、これらに区切られる単語を含む `string` オブジェクトを生成し、`vector` オブジェクトにそのコピーを格納します。そして最後にこの `vector` オブジェクトを戻せばよいわけです。

```
vector<string> split(const string& s)
{
    vector<string> ret;
```

*2 訳注：文字列の中で空白で分けられるデータなので、本書では以後これを「単語」とよんでいます。

```

typedef string::size_type string_size;
string_size i = 0;
// 不変な表明：[もとの i, 現在の i)までを処理
while (i != s.size()) {
    // 始めにある空白を無視する
    // 不変な表明：[もとの i, 現在の i)はすべて空白
    while (i != s.size() && isspace(s[i])) {
        ++i;
    }
    // 次のデータ（単語）の終わりを見つける
    string_size j = i;
    // 不変な表明：[もとの j, 現在の j)はすべて空白ではない
    while (j != s.size() && !isspace(s[j])) {
        ++j;
    }
    // もし空白でない文字を見つけたら
    if (i != j) {
        // iからj - iまでのコピーを格納
        ret.push_back(s.substr(i, j - i));
        i = j;
    }
}
return ret;
}

```

これまでに必要だった標準ヘッダに加えて、このコードは `isspace` を定義している `<cctype>` ヘッダが必要です。一般的にいうと、このヘッダには個々の文字を処理する有用な関数が定義されています。C では `ctype` とす。最初の `c` が、これはもとは C のヘッダだったことを示す。そのオブジェクトの不要なコピーを避けることができるのです。

さて、`split` 関数にはパラメータが 1 つしかありません。これは `const string` への参照で、`s` という名前で、`split` 関数には `const` の参照を使っています。これは引数の `string` オブジェクトから切り出した単語を格納するためになります。単に、`s` から単語をコピーするだけなので、`split` 関数は `string` オブジェクトを変更しません。

まず、`ret` の定義から始めています。これは引数の `string` オブジェクトから切り出した単語を格納するためになります。この `while` の中では、2 つのインデックスの位置を決めます。まず、`s` の中で空白でない最初の文字を見つけて、そのインデックスを `i` にします。入力される引数の文字列には空白文字が複数並んでいることもあるので、`i` を空白でない文字が現われるまで進めるのです。これは次のステートメントで行われますが、説明する

たくさんあります。

```

while (i != s.size() && isspace(s[i]))
    ++i;

```

まず `isspace` は `char` を引数に取り、それが空白かどうかを判定し、その結果を戻す関数です。`&&` 演算子は、左側のオペランドが共に `true` かどうかをテストし、それらの少なくとも一方が `false` なら `false` を戻す演算子です。この `while` の条件のエクスプレッションでは、`i` が `s` のサイズに等しくなく（つまり、まだ文字列の最後に到達していないこと）、`s[i]` が空白であるとき `true` になります。その場合は、`i` を進めてまた同じチェックを繰り返すのです。

5.6 string オブジェクトを分ける

§ 2.4.2.2 で見たように、論理演算子である `&&` は、オペランドの評価に関して短絡型です。そして、前の例とは違って、今度は `&&` のこの性質を使っているのです。短絡型とは、2 項論理演算子 (`&&` と `||`) で、まず左側のオペランドを評価するものです。そしてここで結果が確定すれば、右側のオペランドは評価されないのです。`&&` の場合では、左側のオペランドが `true` のときのみ右側のオペランドが評価されるわけです。具体的には、この条件はまず `i != s.size()` を調べ、この条件が正しいときに限り、`s` の第 `i` 番の文字を調べるということです。もし、`i` が `s.size()` に等しければ、第 `i` 番の文字ありませんが、そのときにはループは終了してしまうわけです。

この `while` が終わると、`i` は空白でない文字を指すか、入力された文字列にそのような文字がもう残っていないかのどちらかです。

ここで、文字列の終わりにはまだ到達せず、`i` は有効なインデックスとすると、次の `while` は `s` 内の現在見ている単語の直後の空白を探すものになります。これを示すインデックスは `j` としますが、まず `j` を `i` に等しくしてループを始めています。その `while` ループは次のようなものです。

```

while (j != s.size() && !isspace(s[j]))
    ++j;

```

これは前の `while` と似ていますが、今度は空白を見つけたら止まるようにしたのです。まず、`j` がまだ文字列の終わりに到達していないことを確かめてから、第 `j` 番の文字が空白かどうか `isspace` でチェックしています。ここでは `isspace` 関数の戻り値を、論理的否定演算子 (logical negation) `!` で、真偽の逆転をさせています。`!` の働きにより、`isspace(s[j])` が `false` のときのみ、`!isspace(s[j])` は `true` になります。

この 2 つの内部 `while` ループを終了すると、次の単語を見つけたか、文字列の最後に到達したかのいずれかになります。文字列の最後に到達している場合は、`i` も `j` も `s.size()` に等しいはずです。逆にそうでない場合は、`ret` に格納すべき単語（空白で区切られた文字列）を見つけたことになります。

```

// 空白でない文字を見つけたら
if (i != j) {
    // iからj - iまでのコピーを格納
    ret.push_back(s.substr(i, j - i));
    i = j;
}

```

`push_back` では、`string` のメンバ関数である `substr` 関数を使っています。これはここで始めて紹介する関数ですが、引数にインデックスと文字列の長さを受け取り、もとの文字列のそのインデックスからその長さ分だけ文字をコピーして、部分文字列を作る関数です。先ほど見つけたインデックス `i` を使い、第 `i` 番からはじまる部分文字列の生成です。この関数を使い `[i, j)` の（半分開いた）範囲の文字列を `ret` に格納しています。§ 2.6 で説明しましたが、半分開いた範囲の要素数は、境界のインデックスの差になります。つまり、コピーする文字数は正確に `j - i` に等しいわけです。

5.7 split 関数をテストする

関数を書いたので、テストをしてみましょう。もっとも簡単な方法は、1行の文字列を読み込み、それを `split` 関数で区切ることであります。そのようなテストプログラムは出力を調べるのが簡単で、期待通りの単語の集まりを生成しているかどうかで、`split` が正しく働いているかを見ることができます。

さらに1行ずつではなく空白で区切られたデータ（単語）ごとに読み込みをし、それを出力するプログラムも書いておくと便利です。これは前のプログラムと同じ結果を出力するはずだからです。実際、そのようなプログラムを書き、実行し、同じ入力に対していつも同じ出力をすることも確かめられます。そこまでは正しいと思えるようになります。

まず、`split` のテストプログラムを書いてみましょう。

```
int main()
{
    string s;
    // 1行ずつ読み込みそれを単語に分解する
    while (getline(cin, s)) {
        vector<string> v = split(s);
        // v内の単語をすべて書き出す
        for (vector<string>::size_type i = 0; i != v.size(); ++i)
            cout << v[i] << endl;
    }
    return 0;
}
```

このプログラムは入力を1行ずつ読んでいかなければなりません。幸い、`string` ライブラリには `getline` という便利な関数があります。これは1行全部を読み込む関数です。この関数は2つの引数を取りますが、最初のものは読み込みの `istream` で、もう1つは読み込んだ文字列を格納する `string` 変数の参照を表します。そして、この関数は他の入力と同様に、読み込みをしたストリーム `istream` への参照を戻すので、戻り値をチェックして `istream` の状態を知ることができます。もし、ファイルの終わりや不適正な入力に出会うと、`getline` は `false` に相当するものを戻すので、`while` から抜けることができます。

正しい入力が続く限り、それを `s` に格納し、`split` に渡し、その戻り値を `v` に格納します。次に、その1行1行に対し、`vector` である `v` 内の全単語をループを使って出力します。

プログラムに、`split` の定義を書くヘッダも含めて、適当なヘッダを `#include` でインクルードすれば、このプログラムは実行でき、`split` が予想通り動作することを確かめることができます。しかし、次のようにライブラリに仕事をさせるプログラムを書き、実行し、その結果と上のプログラムの結果を比べる方がよいでしょう。

```
int main()
{
    string s;
    while (cin >> s)
        cout << s << endl;
    return 0;
}
```

これら2つのプログラムは同じ入力に対しては、同じ結果を出力するはずです。`string` に対する入力演算子

は、1行に書いた文字列を単語（空白で区切られたデータ）に分解して入力していくからです。同じ、しかし、複雑な入力を使って、`split` 関数が正しく動作していることを確信することができるでしょう。

5.8 string オブジェクトをまとめる

§ 1.2 と § 2.5.4 で、*の四角の中央に人の名前を書くというプログラムを考えました。しかし、そこではプログラムの出力を保持する `string` オブジェクトは作ませんでした。その代わり、出力を部分に分け、それらを個別に出力して、出力画面上で1つの画像にしました。

ここでは、この問題をもう1度考え直し、フレーム（枠）付きの文字列全体をあらわすデータ構造を構成してみましょう。これは「文字で絵を描く」という私たちのお気に入りの例を簡単化したもので、文字を四角く並べて表示するものです。ビットマップ・グラフィックスを基礎にする実際のアプリケーションで行われることを簡単化したものとも言えます。実際のアプリケーションではビットを扱うのに対しここでは文字を使い、グラフィカルハードウエアの代わりに、文字を表示するディスプレイを使うということです。この問題は、もともと Stroustrup の The C++ Programming Language (Addison-Wesley, 1986) の第1版の課題として挙げられ、私たちが Ruminations on C++ (Addison-Wesley, 1997)^{*3} で解答を披露したものです。

5.8.1 文字絵にフレームを付ける

この節で考えたい「文字で絵を描く」プログラムは、`vector<string>` に格納されている単語をフレーム（枠）付きの文字列として出力するというものです。文字列はフレームの中で左詰めにし、フレームの*と出力する文字列の間に空白が1つ入るようにします。

ここで `p` が `vector<string>` とし、「this is an」「example」「to」「illustrate」「framing」を含んでいるとします。そのとき、`frame` という名前の関数で、`frame(p)` として、次のようなデータの入った `vector<string>` を作りたいのです。

```
*****
* this is an *
* example   *
* to        *
* illustrate *
* framing   *
*****
```

ここで、文字列の長さはいろいろなのに、フレームはどこかしていない四角であることに注意してください。これから、`vector` 内の最長の文字列の長さを見つける関数が必要であることがわかります。まず、これから考えましょう。

```
string::size_type width(const vector<string>& v)
{
    string::size_type maxlen = 0;
    for (vector<string>::size_type i = 0; i != v.size(); ++i)
        maxlen = max(maxlen, v[i].size());
```

^{*3} 訳注：『C++再考』（ピアソン・エデュケーション／星雲社発行）

```
return maxlen;
}
```

関数の中では `vector` の要素に対して `for` 文が処理を行い、`maxlen` にそこまでの最長の文字列の長さをセットすることにしました。ループが終了すれば、`maxlen` には `v` の中の全文字列中最長のものの長さが保持されることになります。

`frame` 関数の難しいところは、インターフェースだけです。引数には `vector<string>` を取ると決めましたが、戻り値はどうしましょうか。与えられた絵（引数の `vector`）を変更するのではなく、新たに絵（`vector`）を生成する関数の方が便利でしょう。

```
vector<string> frame(const vector<string>& v)
{
    vector<string> ret;
    string::size_type maxlen = width(v);
    string border(maxlen + 4, '*');
    // フレームの1番上を出力
    ret.push_back(border);
    // 内部の行を左右に*と空白を入れて出力
    for (vector<string>::size_type i = 0; i != v.size(); ++i) {
        ret.push_back("* " + v[i] +
                      string(maxlen - v[i].size(), ' ') + " *");
    }
    // フレームの1番下を出力
    ret.push_back(border);
    return ret;
}
```

この関数は与えられた文字絵（つまり引数の `vector`）を変えないので、パラメータは `const` の参照にしました。そして、新しい絵は `vector<string>` である `ret` の中に構成することにしました。まず、最初に最長の文を出します。それから、フレームの1番上と下に使うため*のみの文字列を作っています。

フレームの横幅は最長の文字列より4文字分長くします。左右それぞれにフレームの*とフレームと文字列を分ける空白があるからです。§1.2で見た `string` オブジェクト `spaces` の構成方法を思い出してください。これと同様に、`border` という `maxlen + 4` 個の*を保持する `string` オブジェクトを作っています。そして、そのコピーを `push_back` で `ret` に格納します。

次に、引数の文字列をコピーします。`for` 文でインデックス `i` を定義し、`v` の要素1つひとつを最後までコピーします。ただし、§1.2で見た方法、つまり+を使って `v` に文字列をつなぎながら、それを `push_back` の引数にしています。

出力行を作るために、`v[i]` に保持されている文字列の左右にフレームを付けなければなりません。文字列をつなぐ中で3番目の `string(maxlen - v[i].size(), ' ')` は、必要な分だけ空白を含む名前なしの一時オブジェクトです。これは `border` の初期化と同じ形をしています。また、空白の数は `maxlen` から現在の文字列の長さを引いて求めています。

これらから、`push_back` の引数は、*、空白を1つ、出力する文字列、それからその文字列の長さが `maxlen` になるだけの数の空白、さらに空白を1つ、*からできていることがわかると思います。

最後はフレームの1番下を付け加えて、`ret` を戻せばおしまいです。

5.8.2 文字絵を縦につなぐ

文字絵はおもしろい例で、1度作るといろいろなことができます。今見たのはフレーム付けという1つの操作でした。他に、縦あるいは横に絵をつなぐ操作も考えられます。ここでは縦方向につなげる操作を考え、横方向につなげる操作は次の節で考えることにします。

文字絵は基本的に文字列の行からできます。そのため、この絵を `vector<string>` で表したのです。したがって、2つの絵を縦につなげるのは簡単です。つまり、`vector` をつなげればよいのです。こうすると左の線がそろうことになりますが、それは縦につなぐ場合には自然でしょう。

問題は、`string` には「つなぐ」という操作があるのにに対し、`vector` にはそのような操作がないことです。結局、それは自分でコードを書かなければなりません。

```
vector<string> vcat(const vector<string>& top,
                     const vector<string>& bottom)
{
    // 上の絵をコピー
    vector<string> ret = top;
    // 下の絵を全部をコピー
    for (vector<string>::const_iterator it = bottom.begin();
         it != bottom.end(); ++it)
        ret.push_back(*it);
    return ret;
}
```

この関数はこれまで紹介してきたものだけを使って書きました。まず `ret` を `top` のコピーとし、それに `bottom` の文字列を1つずつ追加していく、最後に `ret` を戻すのです。

しかし、この関数で行っているように、あるコンテナの中身を別のコンテナの中に挿入するということは、よくあることです。この場合は、特に中ほどへの挿入ではなく、最後への付け足しだけですが、それは挿入の特別な例として見ることができます。

実は、このような操作はよくあることなので、ループを書かずに処理する方法がライブラリにあります。つまり、

```
for (vector<string>::const_iterator it = bottom.begin();
     it != bottom.end(); ++it)
    ret.push_back(*it);
```

は、次のように書くことができるのです。

```
ret.insert(ret.end(), bottom.begin(), bottom.end());
```

これらは、同じ結果になります。

5.8.3 横方向につなぐ

ここで「横方向につなぐ」とは、2つの絵をもとに、1つを左にもう1つを右に配置して新しい絵を作るという意味です。始める前に、絵のサイズが異なる場合にどうするか考えておかなければなりません。ここでは、上

の線をそろえることにしましょう。したがって、出来上がる絵のすべての行は、2つの絵の行をつなげたものになります。

左の絵の右側には適当な空白を入れなければなりませんが、さらに、2つの絵の行数が違う場合も考える必要があります。たとえば、pがある文字の絵を意味し、pとpにフレームを付けたものを横方向につなぐという操作も考えられます。これは hcat(p, frame(p)) と書くことにしますが、その結果は次のようにになります。

```
this is an ****
example * this is an *
to   * example   *
illustrate * to   *
framing   * illustrate *
           * framing   *
*****
```

ここで左の絵の行数の方が右の絵の行数より少ないと注意してください。このため、左の絵の下には足りない分だけ、空白行を挿入しなければならないです。しかし、逆に、左の絵の行数のほうが大きいときは、左の絵の行を新しい絵の行にするだけで、右の絵の下にわざわざ空白行を入れたりはしないことにします。

これで横方向につなぐ関数を次のように書けるでしょう。

```
vector<string>
hcat(const vector<string>& left, const vector<string>& right)
{
    vector<string> ret;
    // 2つの絵の間に空白を1つ入れる
    string::size_type width1 = width(left) + 1;
    // leftとrightから1行ずつ読み込むためのインデックス
    vector<string>::size_type i = 0, j = 0;
    // 両方の絵のすべての行を処理するまでループを続ける
    while (i != left.size() || j != right.size()) {
        // 両方の絵の行をつないで保持するためのstringオブジェクト
        string s;
        // 左の絵に行が残っていれば、1行をコピー
        if (i != left.size())
            s = left[i++];
        // 適当な数の空白を入れる
        s += string(width1 - s.size(), ' ');
        // 右の絵に行が残っていれば、1行をコピー
        if (j != right.size())
            s += right[j++];
        // sを新しい絵に付け加える
        ret.push_back(s);
    }
    return ret;
}
```

frameやvcatのときと同様に、後で戻す絵(vector)を定義するところから始めています。それから左の絵の足りないところに詰める空白の数を決めるため、幅を計算しています。ただし、width1は、2つの絵の間に空白を1つ入れるため、左の絵の幅より1つ大きくなっています。それから左の絵から1行コピーし、必要なら空白を詰め、右の絵から1行コピーするのを繰り返します。

気を付けるところは1ヶ所だけで、それは、片方の絵の方ももう1つの絵より先にコピーが終わってしまう場合の処理です。上の関数は繰り返しを、入力した絵(vector)のすべてをコピーするまで続けています。つまり while ループは、両方の絵の最後に到達するまで繰り返しを続けるのです。

まだ left の絵をすべてコピーしないちは、その1行を s にコピーしますが、left にコピーすべき行が残っていてもいなくても、必要な数の空白だけ string オブジェクトを作り、s に+=でつなぎます。この演算子は、string オブジェクトに対して、たぶん普通の人が予想するように、右のオペランドを左のオペランドにつなぎ、その結果を左のオペランドに格納するのです。つまり、ここで「足す」ということは、「つなぐ」ことなのです。

詰め込む空白の数は width1 から s.size() 引いて出しています。s.size() は左の絵からコピーした行の長さか、0になるはずです。ここで0になるのは、左の絵がコピーされつくされ、もう何もコピーされなかつた場合です。width1は2つの絵の間に空白を1つ入れる目的で左の絵の最長の行の長さに1を足したものになっています。そのため、s.size() は、s に左の絵をコピーした場合でも、width1より大きくなることはありません。この場合は、1つ以上の空白が左の絵の行の右に詰め込まれることになります。一方、s.size() が0なら、長さ width1 の空白を s にコピーすることになります。

左の絵に空白を詰め込んだものを s にコピーし、さらに右の絵が残っているなら、次は右の絵の行をコピーするだけです。ただし、もう右の絵が残っていないなら、s には何も付け加えません。いずれにしても、この s を出力する vector に追加します。こうして、左右の入力の絵のすべてをコピーし終わったら、新しい絵である ret を戻して関数は終了です。

ここで、s が while ループ内で定義されていることに注意してください。s はループ内で繰り返しのたびに空の文字列として生成され、使用後は破棄されているのです。そのようになっていなければ、このコードは正しく動かないのです。

コンテナと反復子（イテレータ）： 標準ライブラリでは、異なるコンテナに対する似たような操作は、インターフェースと構文が同じになるように工夫されている。ここまで扱ってきたコンテナはみなシーケンシャル(sequential)である。第7章では、標準ライブラリの連想コンテナを紹介する。すべてのシーケンシャルなコンテナと string には次のような操作がある。

```
container<T>::iterator
container<T>::const_iterator
    コンテナ container<T>の反復子の型名。
container<T>::size_type
    コンテナ container<T>の最大サイズを扱える適当な型の名前。
c.begin()
c.end()    コンテナの先頭と末尾（最後の要素の1つ後）の要素を指す反復子。
c.rbegin()
c.rend()    コンテナを逆順に扱うための、最後と最初（の1つ前）の要素を指す反復子。
```

```

container<T> c;
container<T> c(c2);
    空のコンテナ c の生成、コンテナ c2 をコピーしたコンテナ c の生成。

container<T> c(n);
    T の初期化方法に基づいて初期化された T オブジェクトを n 要素持つコンテナ c の生成 (§ 7.2)。
    もし、T がクラスなら、初期化の方法をコントロールできる。T が組み込み型 (C++にもとからある型) の数値なら、0 に初期化される。

container<T> c(n, t);
    t をコピーした n 要素を持つコンテナ c の生成。

container<T> c(b, e);
    反復子 b と e に対し、範囲 [b, e) のコピーを持つコンテナ c の生成。

c = c2    c を c2 のコピーと置き換える。

c.size()  コンテナ c の要素数を戻す。その型は size_type。

c.empty()  コンテナに要素がないかどうかを戻す。

c.insert(d, b, e)
    反復子 b と e に対し、範囲 [b, e) のコピーを反復子 d で表される c の要素の直前に挿入する。

c.erase(it)
c.erase(b, e)
    反復子 it で表される要素の削除、反復子 b と e に対し、範囲 [b, e) の削除。この操作は list では速いが、vector と string では遅いかもしれない。vector と string では、削除了した要素の後ろの要素をすべて前に移動しなければならないからである。list では、削除了した要素を指していた反復子は無効になる。vector と string では、削除了した要素とそれ以後の要素を指していた反復子が無効になる。削除了した要素の次の要素を指す反復子が戻り値になる。

c.push_back(t)
    c の最後に値 t を追加する。

ランダムアクセスをサポートするコンテナと string 型の場合は次の通り。

c[n]  コンテナ c の第 n 要素をとどくことができる

反復子操作：

*it    反復子 it の指す要素を表す。これによりその要素の値を取り出すことができる。その要素がクラスオブジェクトである場合、しばしばそのメンバを使うことがある。メンバを x とすると、デリファレンス * は優先順位が低いので、x を使うには、(*it).x と書く必要がある。この優先順位は ++や--と同じである。

it->x  (*it).x と同じ。反復子 it の指す要素のメンバ x を表す (戻す)。演算子 . と同じ優先順位である。

++it
it++    反復子がコンテナの現在の要素の 1 つ後を指すようにする。

```

5.9 詳細

b == e
b != e 反復子が等しいか等しくないかを比較。

string 型： 反復子を持ち、それにより vector 同じ操作ができる。詳しくは第 8 章で紹介するが、特に、string ではランダムアクセスができる。

s.substr(i, j) s 内でインデックスが [i, i + j) である範囲の文字をコピーした string オブジェクトを生成する。

getline(is, s) 入力から 1 行読み込み、それを s に格納する。

s += s2 s に s2 をつないだものを新しい s にする。

vector 型： ライブラリのコンテナ中もっとも強力なランダムアクセス反復子とよばれる反復子を使える。その詳細は第 8 章で紹介する。

これまでに書いた関数ではすべて、vector のメモリ確保を動的に行っているが、最初にメモリを確保する方法もある。また、確保したメモリをすぐに使わず、必要になったときに使い、メモリ確保のオーバーヘッド (余分にかかる時間) を減らすことができる。

v.reserve(n) n 要素分のメモリを確保するが、初期化はしない。この操作でコンテナのサイズは変わらない。これは push_back が呼ばれて vector がメモリ確保をする頻度に影響するのみである。

v.resize(n) v のサイズを n にする。もし、n が現在のサイズより小さいときは、それより先の要素は破棄される。もし、n が現在のサイズより大きければ、新しい要素が付け足される。これらは v の中に初期化される。

list 型： コンテナの任意の位置での要素の挿入と削除をもっとも効率よくするようにできている。list とその反復子の操作は § 5.9 で紹介した。さらに、次のようなものがある。

l.sort()
l.sort(cmp) 演算子 < を使って l の要素をソートする、あるいは bool 値を戻す判定関数 cmp を使って l をソートする。

<cctype>ヘッダ： 文字データを操作するのに有用な関数を提供する。

isspace(c)	c が空白のとき true。
isalpha(c)	c がアルファベットのとき true。
isdigit(c)	c が数字のとき true。
isalnum(c)	c がアルファベットか数字のとき true。
ispunct(c)	c がピリオドのとき true。
isupper(c)	c が大文字のとき true。
islower(c)	c が小文字のとき true。
toupper(c)	c を大文字にする。
tolower(c)	c を小文字にする。