

第6章

ライブラリのアルゴリズムを使う

第5章で見たように、多くのコンテナの操作は2つ以上の型に適用されます。たとえば、`vector`、`string`、`list`では、要素の挿入は`insert`関数で、削除は`erase`関数で行います。これらの操作は、それぞれのコンテナに対し同じインターフェースを持つわけです。特に、多くのコンテナに対する操作は`string`にも適用できることに注意してください。

それぞれのコンテナは、`string`も含めて反復子（イテレータ）の型を持っています。この反復子を使ってコンテナ内の要素にアクセスできるのです。また、反復子も、異なるコンテナに対して同じインターフェースで使えるようにライブラリはできています。たとえば、`++`演算子は反復子のタイプにかかわらず反復子を1つ進めることがあります。また、`*`演算子はその反復子の指す要素にアクセスするために使われます。

この章では、ライブラリの標準アルゴリズムが、これらの反復子を通して、共通のインターフェースを持つことを見ていきます。このようなアルゴリズムを使うことで、同じコードを何度も書く（あるいは書き直す）ことを避けることができます。さらに重要なことは、これを用いればプログラムを小さく単純に書くことができるということです。これはときには驚くほどの効果があります。

コンテナと反復子と同様に、アルゴリズムも一貫したインターフェース規約を持っています。そのおかげで、いくつかのアルゴリズムを学ぶと、必要に応じて、それを他のアルゴリズムにも応用できるのです。この章では、`string`の処理と学生の成績に関する問題を解決するため、ライブラリのアルゴリズムをいくつか見てみます。これにより、アルゴリズムのライブラリの中心的な概念を扱うことになります。

なお、特に述べない限り、この章で使うアルゴリズムは`<algorithm>`ヘッダに定義されています。

6.1 string の解析

§5.8.2では、文字の絵をつなぐために、次のようなループを書きました。

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

このループは`bottom`のコピーを`ret`の最後に入れるのと同じことで、そのような操作は次のように直接行うこともできます。

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

実は、これにはもっと一般的な方法もあるのです。ここでは、コンテナの最後に要素を入れるというより、コピーすることを考えます。

```
copy(bottom.begin(), bottom.end(), back_inserter(ret));
```

この copy がこの章で紹介するジェネリックアルゴリズムというものの一例です。back_inserter は反復子アダプタの一種です。

ジェネリックアルゴリズム（汎用アルゴリズム、genetic algorithm）とは、特定のコンテナに付随しているのではなく、引数の型からデータ構造を判断し、多くのコンテナに共通して作用できるアルゴリズムです。標準ライブラリのジェネリックアルゴリズムは、普通は反復子を引数に取り、その反復子によってコンテナを操作します。たとえば、copy アルゴリズムは 3 つの反復子を引数に取ります。これを begin、end、out とすると、このアルゴリズムは [begin, end) の範囲のすべての要素を out の場所に、必要ならコンテナを拡張して、コピーするものです。言い換えれば、

```
copy(begin, end, out);
```

は

```
while (begin != end)
    *out++ = *begin++;
```

と同じ効果を持つのです。ただし、while では中身の反復子の値を変えていますが、copy では変えません。

反復子アダプタの説明の前に、上のループを説明しておきましょう。このループではインクリメント（1つ進める）演算子の後置版（postfix）が使われています。これはこれまで使ってきた前置版のものとは違います。begin++ はもとの begin を戻し、副作用で begin が保持する値をインクリメントするのです。言葉を変えると、

```
it = begin++;
```

は

```
it = begin;
++begin;
```

と同じです。インクリメント演算子は * と同じ優先度を持ち、両方とも右結合なので、*out++ は *(out++) という意味になります。そのため、

```
*out++ = *begin++;
```

は、

```
{ *out = *begin; ++out; ++begin; }
```

と同じ意味になるのです。

それでは、反復子アダプタ (iterator adaptor) の説明をしましょう。これは引数からいろいろ役に立つ反復子を生成する関数です。これは<iterator>ヘッダに定義されています。そして、一番使われるものが back_inserter です。これはコンテナを引数に取り、今の使い方では、コンテナにデータを追加する場所を表す反復子を戻すのです。たとえば、back_inserter(ret) は、この場合は ret にデータを追加するための反復子です。そのため、

```
copy(bottom.begin(), bottom.end(), back_inserter(ret));
```

で、bottom の全要素をコピーして ret の最後に追加することになるのです。この関数が実行されると、ret のサイズは bottom.size() が戻す分だけ大きくなります。

ここで次のような間違いをしないように気をつけてください。

```
// エラー：retは反復子ではない
copy(bottom.begin(), bottom.end(), ret);
```

copy の最後の引数は反復子であってコンテナではありません。また、次も間違いです。

```
// エラー：ret.end()の指す場所に要素はない
copy(bottom.begin(), bottom.end(), ret.end());
```

このようにしても、プログラムはコンパイルできるので、これはタチの悪い間違いです。プログラムがコンパイルできても、そのコンパイルした実行可能ファイルがちゃんと実行できるという保証はありません。実行すると copy は最初 ret.end() の要素を探してそこに値を入れようとするでしょう。しかし、そこには要素はないのです。そこでどうなるかは予想できません。

どうして copy はこのように設計されたのでしょうか。それは、要素のコピーとコンテナの拡張を切り離しておくことで、プログラマの選択範囲を広げるためです。たとえば、すでに要素があるコンテナに対し、コンテナのサイズはそのまままで、別のコンテナの要素を上書きしたいと考えるかもしれません。また、§ 6.2.2 で別の例を見ますが、back_inserter を使って、別のコンテナの単なるコピーではない要素をコンテナに付け足すこともできます。

6.1.1 split の別の書き方

§ 5.6 で考えた split も標準アルゴリズムを使うと、もっと直接的なものに書き換えることができます。この関数の難しいところは、単語（文字データ）を分けるもの（今のは空白）の扱い方でした。まず、反復子を使うことでインデックスを使わないようにできます。そして、仕事の大部分を標準ライブラリのアルゴリズムに任せることができます。

```
// 引数が空白ならtrueそうでなければfalse
bool space(char c)
{
    return isspace(c);
}

// 引数が空白ならfalse、そうでなければtrue
bool not_space(char c)
{
    return !isspace(c);
}

vector<string> split(const string& str)
{
    typedef string::const_iterator iter;
    vector<string> ret;
```

```

iter i = str.begin();
while (i != str.end()) {
    // はじめに空白があればそれらは無視する
    i = find_if(i, str.end(), not_space);
    // 次の単語の終わりを探す
    iter j = find_if(i, str.end(), space);
    // [i, j)の範囲の文字をコピー
    if (i != str.end())
        ret.push_back(string(i, j));
    i = j;
}
return ret;
}

```

このコードでは新しい関数をたくさん使ったので、少々説明が必要です。しかし、キーとなるアイデアは、もとのものと同じであることは押されておいてください。*i* と *j* が *str* の中の単語（入力文字列 *str* 中にある空白で区切られた文字列データ）の始めと終わりを指すことにし、単語を見つけると、この部分を *str* からコピーして *ret* に格納しているのです。

ただし、今度は *i* と *j* はインデックスではなく反復子です。まず、反復子の長い名前を繰り返さないために `typedef` を使い、`string::const_iterator` を *iter* としています。`string` はコンテナとまったく同じではありませんが、反復子を持っています。そのため、`vector` の要素を取り出したように、`string` の要素である文字を標準ライブラリのアルゴリズムで扱うことができるのです。

たとえば、ここでは `find_if` というアルゴリズムを使っています。この最初の引数は、シーケンス（データの並び）を示す反復子で、3番目の引数が判定関数（predicate）です。判定関数とは、引数を調べて `true` か `false` を戻す関数です。`find_if` 関数は、シーケンス内の各要素に対し判定関数を適用していく、これが `true` を戻す要素のときに、その処理を止めるのです。

標準ライブラリには `isspace` という文字が空白かどうかを調べる関数があります。しかし、この関数は、日本語のような `wchar_t` (§ 1.3) を使う言語に対応するためオーバーロード^{*1}されています。このようにオーバーロードされた関数を直接 `find_if` のような関数の引数にすることは簡単ではありません。なぜなら、関数を `find_if` の引数にするときに、その関数自身の引数を指示しないので、コンパイラがオーバーロードされた同名の関数のうちのどれを使うべきか判断できなくなるからです。そのため、ここでは `isspace` をもとに `space` と `not_space` という独自の関数を作りました。

最初に `find_if` を使っているのは、最初の空白でない文字を探すためです。これが単語の始めになるわけです。入力文字列の先頭にも、単語間にも複数の空白が入っているかもしれないことを忘れないでください。これらの空白は出力には含めたくないのです。

この関数を最初に実行すると、*i* は（もしあれば）*str* の最初の空白でない文字を指すようになります。そして、次回にこの関数が呼ばれるときには、範囲 `[i, str.end()]` の内で最初の空白でない文字列を探すことになります。もし、判定関数が `true` を戻す要素がない場合、`find_if` は 2番目の引数を戻すことになっています。そのため *j* は次の単語とそれ以降を分ける空白か、（すでに最後の単語に入っている場合）`str.end()` になります。

*1 訳注：引数の種類や数が異なる同名の関数が定義されているという意味。演算子については第1章 1.2節で見ました。

6.1 string の解析

これで *i* と *j* は *str* 内の単語の始めと終わりを指すことになりました。次にすることは、*str* から *i* と *j* に挟まれた部分をコピーして *ret* に入るだけです。前の `split` では、コピーを作るために `string::substr` を使いました。しかし、今度はインデックスではなく反復子を使っています。反復子に働く `substr` 関数はないのです。その代わり、反復子から直接新しい `string` オブジェクトを作ることができます。それが `string(i, j)` のことです。これは § 1.2 で説明した `spaces` の作り方に少し似ていますね。今の場合、範囲 `[i, j)` の文字のコピーで新しい `string` ができるのです。そして、これを *ret* に格納して終わりです。

新しい `split` ではインデックス *i* と `str.size()` の比較がなくなっていることに注意してください。それに対応する反復子のチェックもしていません。その理由は、ライブラリのアルゴリズムは空の範囲を渡されてもちゃんと処理できるようになっているからです。たとえば、ある時点で、最初の `find_if` が戻す *i* の値と `str.end()` が同じになるわけですが、それをチェックしておく必要はないのです。2番目の `find_if` は `[i, str.end()]` が空の領域だとわかると、検索しているものがなかったと判断して、`str.end()` を戻して終了するからです。

6.1.2 回文（単語）

他の文字操作関連の問題で、ライブラリをうまく使えるものに、単語の回文性を判定するというものがあります。回文とは、前から読んでも後ろから読んでも同じになる単語で、civic（市民の）、eye（目）、level（レベル）、madam（マダム）、rotor（ローター、回転するもの）などがそうです。

これを判定する関数は、ライブラリを使うと、次のように簡単に書けます。

```

bool is_palindrome(const string& s)
{
    return equal(s.begin(), s.end(), s.rbegin());
}

```

この関数は `equal` という関数の戻り値を戻しています。`equal` とその引数の `rbegin` というメンバ関数はここではじめて紹介するものです。

`begin` と同様に `rbegin` も反復子を戻す関数ですが、これはコンテナの最後から前に向かっていく反復子を戻すのです。

`equal` 関数は 2 つのシーケンスを比べて同じ値であるかどうかを判定する関数です。例によって、最初の 2 つの引数は比較するための最初のシーケンスを表す 2 つの反復子です。3 番目の引数は、比較するもう 1 つのシーケンスの先頭を表すものです。`equal` は長さの等しいシーケンスを比較するように作られているので、2 番目のシーケンスの終わりを示す反復子はわざわざ使いません。そして、`s.rbegin()` を 2 番目のシーケンスの先頭を表す反復子として与えているので、`s` をさかさまに見たシーケンスと `s` のものを比較していることになります。つまり、`equal` はまず `s` の最初の文字と最後の文字を比較し、次に 2 番目の文字と最後から 2 番目の文字を比較し… と言うように続いていきます。これは、まさに、回文を見つける関数になっています。

6.1.3 URL を見つける

文字操作の最後の例として、`string` オブジェクトの中から Web アドレス（ユニフォーム・リソース・ロケーター、URL）を見つける関数を書いてみましょう。この関数の引数には、文書がまるまる 1 つ入っている

stringオブジェクトを使うかもしれません。このドキュメントを走査して、文書内のすべてのURLを見つけるというものです。

URLは次のような形をした文字の並びです。

```
protocol-name://resource-name
```

ここでプロトコル名 *protocol-name* は文字だけを含み、リソース名 *resource-name* は文字（アルファベット）と数字とドット（ピリオド）を含みます。今書きたい関数は string オブジェクトを引数に取り、その中に :// を見つけだすものです。そして、:// が見つかるたびに、その前の *protocol-name* とその後の *resource-name* を探すものとします。

入力中に含まれるすべての URL を求めたいので、それを `vector<string>` で戻すことにします。この1つひとつの要素が URL になるのです。関数は反復子 *b* を string 内で動かしながら URL の一部である :// を探すようにします。それが見つかればその前後の *protocol-name* と *resource-name* を探すのです。

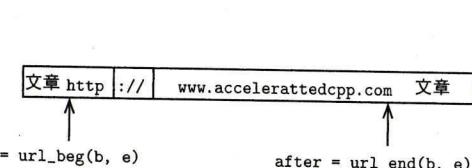
```
vector<string> find_urls(const string& s)
{
    vector<string> ret;
    typedef string::const_iterator iter;
    iter b = s.begin(), e = s.end();
    // 入力全体を調べる
    while (b != e) {
        // 「://」の場所を探す
        b = url_beg(b, e);
        // もしあれば
        if (b != e) {
            // URLの残りを取り出す
            iter after = url_end(b, e);
            // そのURLを記録する
            ret.push_back(string(b, after));
            // bを進め、次のURLを探す
            b = after;
        }
    }
    return ret;
}
```

まず、見つけた URL を格納するため、vector である *ret* を定義します。それから、URL を挟んで取り出すための反復子を定義します。この関数のために、さらに URL の先頭を見つける *url_beg* 関数と URL の末尾を見つける *url_end* 関数を定義しておいたとします。ここで *url_beg* は有効な URL が残っているかをチェックし、あればその *protocol-name* の先頭を指す反復子を戻すように書きます。この関数は、有効な URL がない場合、検索失敗を報告するため、2番目（今は *e*）の引数を戻すことになります。

もし、*url_beg* が URL を見つけたら、今度は *url_end* でその URL の末尾を探すことになります。この関数は、与えられた場所から、入力の終わりか URL の一部とは考えられないものに到達するまで検索をします。そして、URL の最後の文字の1つ後を指す反復子を戻します。

こうして *url_beg* と *url_end* を呼び出した後は、反復子 *b* が URL の先頭を、そして反復子 *after* が URL の末尾（最後の文字の1つ後）を指すようになっているわけです。

6.1 string の解析



この範囲の文字列から string オブジェクトを生成し、それを *ret* に格納しています。

次の仕事は、*b* を進めてまた次の URL を探すだけです。2つの URL が重なることはないので、*b* を先に見つけた URL の末尾（最後の文字の1つ後）にセットし、入力全部を調べ終わるまで while ループを続けます。ループが終了すると、URL のすべてが記録されている *ret* を戻します。

さあ、*url_beg* と *url_end* を考えましょう。*url_end*の方が簡単なので、まずこちらを見てみます。

```
string::const_iterator
url_end(string::const_iterator b, string::const_iterator e)
{
    return find_if(b, e, not_url_char);
}
```

この関数は単に、§ 6.1.1 で紹介したライブラリの *find_if* に仕事をさせているだけです。*find_if* に渡される判定関数の *not_url_char* は、次のように自分で書く関数です。これは URL で使えない文字に対して *true* を戻す関数です。

```
bool not_url_char(char c)
{
    // アルファベット以外でURLに使える文字
    static const string url_ch = "~;/?:@=&$-_+.!*'()";
    // cがURLに使えるかどうかをチェックし、その真偽を逆にしたものに戻す
    return !(isalnum(c) ||
        find(url_ch.begin(), url_ch.end(), c) != url_ch.end());
}
```

この関数のコードは短いですが、新しいものをいくつか使っています。最初は記憶クラスの規定子 (storage class specifier) である *static* です。*static* 宣言されたローカル変数は、関数の複数回の呼び出しに渡って保持されます。つまり、*url_ch* という string オブジェクトは、*not_url_char* の最初の呼び出し時に生成され初期化され、以後の呼び出し時にはそれが使われるだけになります。*url_ch* は *const* であるので、その値を初期値から変えることもありません。

また、*not_url_char* 関数は、<cctype> ヘッダで定義される *isalnum* 関数も使っています。これは、その引数がアルファベットと数字のどちらかである場合 *true* を戻す関数です。

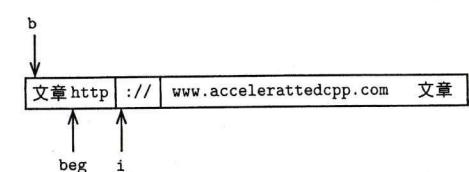
最後には、*find* という新しいアルゴリズムを使っています。これは *find_if* と似ていますが、判定関数ではなく特定の値を3番目の引数に取り、その値を探します。そして、*find_if* と同様に、探している値があった場合、シーケンス内でその値を持っている最初の要素を指す反復子を戻すことになっています。もし検索している値がなければ、*find* は第2引数を戻すのです。

以上の知識から、`not_url_char` 関数を理解することができます。特に戻り値は、最初に求めた値の真偽を逆にしていることに注意してください。このため、`not_url_char` 関数は、`c` がアルファベット、数字、または URL 内の文字であるとき、`false` を戻すのです。そして、`c` がこれ以外の文字のとき、`true` を戻すのです。`url_char` の内文字であるとき、`false` を戻すのです。この関数は、いろいろな入力を処理する必要があり、さて、いよいよ `url_beg`、つまり難しいところです。この関数は、いろいろな入力を処理する必要があり、ときには `://` が URL とは無関係な場所でも使われる所以、とてもごちゃごちゃします。実際には、可能なときには `://` の前には 1 文字以上の文字があり、その後ろには少なくとも 1 文字以上がある場合を、URL と考える所以です。

```
string::const_iterator
url_beg(string::const_iterator b, string::const_iterator e)
{
    static const string sep = ":// ";
    typedef string::const_iterator iter;
    // i は「://」が見つかった場所を指すようにする
    iter i = b;
    while ((i = search(i, e, sep.begin(), sep.end())) != e) {
        // 「://」が最初や最後にないことを確認する
        if (i != b && i + sep.size() != e) {
            // beg はプロトコル名の最初を指すようにする
            iter beg = i;
            while (beg != b && isalpha(beg[-1])) {
                --beg;
            }
            // 「://」の前後に 1 つ以上の有効な文字があることを確認する
            if (beg != i && !not_url_char(i[sep.size()]))
                return beg;
        }
        // 「://」が URL の一部でない場合は、i をその分だけ進める
        i += sep.size();
    }
    return e;
}
```

関数のヘッダ部分（パラメータリストと戻り値）は簡単です。検索範囲を示すため 2 つの反復子が与えられ、もしく URL があるなら、その最初の文字の反復子を戻します。また、URL を特定するための文字列 `://` を保持するローカルな `string` オブジェクトを定義しています。`not_url_char` 関数（§ 6.1.3）の `url_char` と同じく、この `string` は `static` で `const` です。そのため、この `string` オブジェクトは `url_beg` が最初に呼び出されたときに生成され初期化され、以後はそれを変更することはできません。

この関数は `b` と `e` で挟まれる文字列に対して働きます。



反復子 `i` は URL があればその `://` を指すようになります、`beg` はその `protocol-name` の最初を指すようになります。

`url_beg` は `search` というライブリの関数を使って `://` を探します。この `search` は反復子のペアを 2 つ引数に取ります。始めのペアは調べるシーケンスを示し、後のペアは探しているシーケンスを示します。他の同様なライブリ関数と同様に、検索して見つからなかった場合は、2 番目の反復子を戻すことになります。そのため、`search` を呼んだあとは、`i` が入力文字列の末尾（最後の 1 つ後）か、`://` の `:` を指していることになります。

ここで `://` を見つけたなら、次は（もあるなら）`protocol-name` を取り出すという作業です。最初のチェックは、入力文字列が `://` だけでないかのチェックです。もし、`://` ならそれは URL ではありません。もし、これ以外にも文字があるなら、まず `beg` の位置を求めます。内側の `while` ループは、アルファベットでない文字か入力文字列の先頭に到達するまで反復子 `beg` を前に戻していくです。ここで 2 つの新しいことが出てきます。まず、コンテナがインデックスを持つなら、その反復子もインデックスを持つということです。言葉を変えると、`beg[-1]` は `beg` の指す要素の直前の要素を意味するのです。実は `beg[-1]` は `*(beg - 1)` の省略形と考えることもできるのですが、これについては § 8.2.6 で説明しましょう。もう 1 つの新しいものは、`<cctype>` で定義されている `isalpha` 関数です。これは引数がアルファベットかどうかをチェックする関数です。

ここで `beg` を 1 文字でも前に戻せたらそれがプロトコル名だします。ただし、`beg` を戻す前に `://` の後に有効な文字が 1 つでもあるかどうか確認する必要があります。この確認はさらに複雑です。まず、`i + sep.size()` と `e` を比べている `if` 文の中にいるので、文字が少なくとも 1 つはあることが分かっています。そしてその文字を調べるのですが、それは `i[sep.size()]` で表されます。これは先ほどと同様に、`*(i + sep.size())` の省略形です。この文字が URL のリソース名の文字であるかどうかは `not_url_char` を使って調べます。この関数は文字が有効なら `false` を戻す関数でしたが、!を付けることで、文字が有効なら `true` を戻すようにしています。

そして `://` が URL の一部ではないときには、`i` をこの部分だけ進めて、また検索を続けるようにしました。

この関数のコードでは、デクリメント演算子（decrement operator）を使いました。これは § 2.7 で紹介しましたが、使うのははじめてです。これはインクリメント演算子のように働きますが、ただし、1 つ減らす方向に働きます。また、インクリメント演算子と同様に、前置と後置の 2 つのバージョンがあります。今使っている前置バージョンでは、オペランドを 1 つ減らしてから（反復子の場合、1 つ前に進めるということ）、新しい値を戻します。

6.2 成績処理の仕組みを比べる

§ 4.2 では学生の最終成績を宿題の点数のメジアンなどを使って算出することを考えました。悪賢い学生はこの仕組みを悪用して、宿題全部を真剣にしないかもしれません。結局、全宿題中下半分の点数は、最終成績に何の影響もないのです。ある数の宿題をうまくこなしたなら、もうあとは提出しないかもしれません。

私たちの経験では、大抵の学生はこのような抜け道を使いません。しかし、あるところでは、喜んでしかも隠しもせずにこのようなことをする学生もいました。そこで、宿題をあまりしなかった学生の最終成績は、宿題をしもせずにこのようなことをする学生もいました。この問題を考える一方で、成績処理の仕組みを変えたらすべての学生のものより悪いだろうかと思いました。この問題を考えるために、成績処理の仕組みを採用するということです。

- メジアンではなく平均を使う。提出しなかった宿題は0点として計算する。
- 学生が提出した宿題のみのメジアンを考える。

これらのすべての仕組みでの成績処理の結果と、最初のメジアンを使う成績処理の結果を、宿題を全部提出した学生とそうでなかった学生について比べてみたかったのです。ここで2つの異なる問題を解決するプログラムが必要になったわけです。その問題とは次のものです。

1. 全学生的成績データを読み込み、すべての宿題をした学生とそうでない学生を分ける。
2. それぞれのグループについて、それぞれの成績処理の仕組みを使って最終成績を出す。

また、始めに考えた全宿題のメジアンを使う方法も実行する。

6.2.1 学生の成績データを扱う

最初の問題は学生の成績データを読み込み、分類することです。幸い、この問題を途中まで解決してくれるコードは書きました。§ 4.2.1 の `Student_info` と § 4.2.2 の `read` 関数です。まだ足りないところは、学生が宿題全部を提出したかどうかのチェックです。そのための関数を書くことは簡単です。

```
bool did_all_hw(const Student_info& s)
{
    return ((find(s.homework.begin(), s.homework.end(), 0))
            == s.homework.end());
}
```

この関数は `s.homework` を調べ、0が記録されているかどうかをチェックしています。私たちは、提出さえすれば何がしかの点を与えていたので、0点という点は未提出という意味だからです。`find` の戻り値と `s.homework.end()` を比較していますが、`find` は探しているものが見つからなかった場合、2番目の引数を戻すことになっているからです。

`read` とこの関数を使えば、学生の成績データを読み込み、分類するコードはすぐ書けます。学生のデータを1つずつ読み込み、宿題を全部提出したかをチェックし、`did`（「した」という意味）と `didnt`（「しなかった」という意味）の2つの `vector` に振り分けていくのです。この `vector` にはもっとよい名前が思いつかなかったのでこのようにしました。一方、意味のある処理をするために、どちらの `vector` も空でないことを確かめることにしました。

6.2 成績処理の仕組みを比べる

```
vector<Student_info> did, didnt;
Student_info student;
// すべてのデータを読み込み、宿題を全部したかどうかで分類する
while (read(cin, student)) {
    if (did_all_hw(student))
        did.push_back(student);
    else
        didnt.push_back(student);
}
// 両方のvectorが空でないことをチェック
if (did.empty()) {
    cout << "宿題を全部提出した学生はいません！" << endl;
    return 1;
}
if (didnt.empty()) {
    cout << "全部の学生が宿題を全部提出しています！" << endl;
    return 1;
}
```

ここで新しいことは、`empty` というメンバ関数だけです。この関数は、コンテナが空のとき `true` を戻し、そうでなければ `false` を戻します。`size` の戻す値が0かどうかを見るより、この関数を使う方がよいのです。その理由は、コンテナによっては、コンテナが空かどうかだけをチェックする方が、全要素数を正確に数えるより効率的だからです。

6.2.2 成績の解析

これで学生の成績データを読み込み、これらを `vector` の `did` と `didnt` に分類することができるようになりました。次に、これらを解析する方法を考えなければなりません。これは、解析の構造を考えるということです。

成績処理の方法は3つありました。それぞれについて、宿題をすべて提出した学生とそうでない学生について実行する必要があります。すべての成績処理を2つのグループについて行うので、それぞれに関数を作るのがよいでしょう。しかし、共通のフォーマットで報告するようなところでは、操作が同じなので、学生グループを独立に扱うよりペアで処理していく方がよいでしょう。また言うまでもなく、それぞれの学生グループについて結果を書き出すのは1つの関数でいいものです。

工夫が必要な部分は、それぞれの成績処理の結果を書き出す関数を3回呼び出す必要があるというところです。この関数自身は、`did` と `didnt` のそれぞれに1回ずつ成績処理の関数を呼び出すことになるのですが、その関数の内容は成績処理の仕組みごとに異なるわけです。どうすればよいでしょうか。

一番簡単な方法は、3種類の成績処理関数を定義し、これを報告関数に引数として渡すというものです。すでにそのような引数を扱っています。たとえば、§ 4.2.2 で `sort` 関数の引数に `compare` 関数を使っていました。結局、報告の関数は5つの引数を取ることになります。

- 結果を出力するストリーム
- 成績処理方法を表す `string`
- 解析に使う関数
- 成績処理を施したい2つの `vector` オブジェクト

たとえば、最初のメジアンを使う成績処理の方法は median_analysis という関数で実行されるとし、2つの学生グループの成績処理の結果を出力する関数を write_analysis とすると、その実行は次のようになります。

```
write_analysis(cout, "median", median_analysis, did, didnt);
```

この write_analysis を書く前に、median_analysis を書いてみましょう。この関数は学生のデータである vector を引数に取り、始めに決めたメジアンの方法で成績処理を行い、これらの学生の成績のメジアンを戻すことにします。

```
// この関数は実はうまく働かない
double median_analysis(const vector<Student_info>& students)
{
    vector<double> grades;
    transform(students.begin(), students.end(),
              back_inserter(grades), grade);
    return median(grades);
}
```

この関数は一見複雑そうですが、実は、新しいものは transform 関数だけです。この関数は、引数に3つの反復子と1つの関数を取ります。最初の2つの反復子は変換 (transform) する要素の範囲を表します。そして、3番目の反復子はこの関数の実行結果を書き出す場所を表しているのです。

この transform 関数を呼び出すときには、入力シーケンスからの値を書き出す場所が3番目の引数の反復子のところにあるようにしなければなりません。今の場合には問題ありません。back_inserter (§ 6.1) を使ってるので、transform の結果は grades に追加されていくことになります。この場合、grades は必要なだけ拡張されるのです。

transform の4番目の引数は、transform が入力シーケンスに順番に適用し、出力シーケンスを作っていく関数です。そのため、今の例では、transform が呼ばれると students の各要素に grade が適用され、その成績が grades という vector に入れられていくわけです。このようにして全学生の成績が求まつたら、§ 4.1.1 で定義した median を使って、メジアンを計算すればよいわけです。

しかし、1つだけ問題があります。コメントに書いたように、この関数はうまく働かないのです。

その1つの理由は、grade にはいくつものオーバーロードされたバージョンがあるからです。transform の引数としての grade には引数を付けないので、コンパイラが判断できないのです。これは、もちろん § 4.2.2 のバージョンを使いたいのですが、そのようにコンパイラに知らせる必要があるのです。

もう1つの理由は、grade 関数は学生がまったく宿題を提出していない場合に例外を投げる（スローする）のに、transform 関数は例外について何もしないからです。もし例外が起こると、transform 関数は例外の発生時点でストップし、制御を median_analysis に戻します。しかし、median_analysis も例外を処理しないので、例外はさらにその外へ投げられます。その結果、関数は途中で止められ、その関数を呼び出した関数に例外が投げられ、その関数も止められ…が、catch 節に到達するまで続いているのです。もし、適当な catch がなければ、実際今の場合はそうなっていそうですが、プログラムそのものが止められ、メッセージが出力されるでしょう（これは環境によりますが）。

この問題は、grade 関数を try というブロックで使い、例外を処理する補助的な関数を作ることで解決できます。その場合、grade は引数として渡されるのではなく、その関数内で実際に使われる所以、コンパイラはどのバージョンを使うべきか判断できるのです。

```
double grade_aux(const Student_info& s)
{
    try {
        return grade(s);
    } catch (domain_error) {
        return grade(s.midterm, s.final, 0);
    }
}
```

この関数は § 4.2.2 の grade を呼びます。例外が起こると、catch のブロックが実行され、引数に2つの試験結果と宿題の合計3つの double 値を取る § 4.1 で定義した grade が呼ばれます。これにより、宿題を提出していない学生の宿題の点数は0点とし、中間と期末の試験結果のみを使うことになるわけです。

これをもとに成績処理関数は次のように書き換えられます。

```
// 今度はうまくいく
double median_analysis(const vector<Student_info>& students)
{
    vector<double> grades;
    transform(students.begin(), students.end(),
              back_inserter(grades), grade_aux);
    return median(grades);
}
```

成績処理を見たので、次は、結果を書き出す write_analysis を書きましょう。これは、成績処理の関数を2つの学生グループそれぞれに使うものです。

```
void write_analysis(ostream& out, const string& name,
                    double analysis(const vector<Student_info>&),
                    const vector<Student_info>& did,
                    const vector<Student_info>& didnt)
{
    out << name << ": median(did) = " << analysis(did) <<
        ", median(didnt) = " << analysis(didnt) << endl;
}
```

この関数も驚くほど短いのですが、新しいことを2つ使っています。まず、関数を表すパラメータの定義の仕方です。パラメータの analysis の定義は、§ 4.3 の関数の宣言と同じ形をしています。（実際、§ 10.1.2 で説明しますが、もう少し見た目以上のものがあります。しかし、それはここでは問題になりません。）

もう1つの新しいものは、void です。この組み込み型は特別な場合にのみ使われるのですが、その1例が、上のように戻り値の型としての void です。このように関数が void を「戻す」とすると、この関数は何も戻さないということになります。値を戻さない関数の実行を終了するには、値なしで return を使います。つまり、

return;

となるのです。あるいは、上の例がそうであるように、途中に return がなければ、関数は最後まで実行されます。そのようなことは普通はできないのですが、void を戻す関数ならよいのです。

ここで、プログラムの main を書くことができます。

```

int main()
{
    // didは宿題をすべて提出した学生、didntはそれ以外
    vector<Student_info> did, didnt;
    // 学生の成績データを読み込み、2つのグループに分ける
    Student_info student;
    while (read(cin, student)) {
        if (did_all_hw(student))
            did.push_back(student);
        else
            didnt.push_back(student);
    }
    // 解析が無意味でないことを確認する
    if (did.empty()) {
        cout << "宿題を全部提出した学生はいません！" << endl;
        return 1;
    }
    if (didnt.empty()) {
        cout << "全部の学生が宿題を全部提出しています！" << endl;
        return 1;
    }
    // 解析
    write_analysis(cout, "median", median_analysis, did, didnt);
    write_analysis(cout, "average", average_analysis, did, didnt);
    write_analysis(cout, "median of homework turned in",
                  optimistic_median_analysis, did, didnt);
    return 0;
}

```

残っているのは average_analysis (平均を使った処理という意味) と optimistic_median_analysis (楽観的にメジアンを使った処理という意味) を書くだけです。

6.2.3 宿題の平均を使った成績処理

average_analysis は学生の宿題の総合点数を、メジアンではなく平均から出す関数です。そのため、まず、vector の平均を出す関数を書くのが理にかなっているでしょう。これを使って、成績を計算するのです。

```

double average(const vector<double>& v)
{
    return accumulate(v.begin(), v.end(), 0.0) / v.size();
}

```

この関数は accumulate 関数を使っています。これは、今までのライブラリのアルゴリズムと違って、この関数は numeric に宣言されています。このヘッダは名前からわかるように、数値計算 (numeric は「数値の」という意味) のためのツールを用意しています。accumulate 関数は、最初の 2 つの引数で範囲を決め、3 番目の引数にその範囲のものを足していくことになります。

その合計の型は 3 番目の引数と同じものになるので、0 ではなく 0.0 と書いたのには重要な意味があります。もし 0 としておくと、これは int 型と解釈され、小数部分は捨てられてしまうのです。

6.2 成績処理の仕組みを比べる

この accumulate を使うことですべての要素の合計が計算できるので、後はそれを全要素数である v.size() で割って平均を出しています。そして、これを関数の呼び出し側に戻しているわけです。

average 関数を定義したので、平均の方法を使った成績を計算する average_grade を書くことができます。

```

double average_grade(const Student_info& s)
{
    return grade(s.midterm, s.final, average(s.homework));
}

```

この関数は宿題の総合点数を出すのに average を使い、それから最終成績を出すために § 4.1 で定義した grade を使っています。

ここまで下準備で、average_analysis を簡単に書くことができます。

```

double average_analysis(const vector<Student_info>& students)
{
    vector<double> grades;
    transform(students.begin(), students.end(),
              back_inserter(grades), average_grade);
    return median(grades);
}

```

median_analysis と average_analysis の違いは、名前そのものと grade_aux を使うか average_grade を使うかの違いだけです。

6.2.4 提出した宿題のメジアン

最後の方法を表す関数は、optimistic_median (楽観的なメジアン) です。この名前は、提出しなかった宿題の成績は、提出した宿題とほぼ同じだろうという楽観的な考えに従っていることを表しています。この仮定に基づいて、提出した宿題のメジアンをもって宿題の総合点数にするわけです。まず、この楽観的な方法を実現する optimistic_median 関数を書いてみましょう。もちろん、宿題をまったく提出しない学生もいるわけです。その場合は、宿題の成績は 0 点とすることにします。

```

// 要素数が0でない場合はメジアンを戻し、要素がないときは0を戻す
double optimistic_median(const Student_info& s)
{
    vector<double> nonzero;
    remove_copy(s.homework.begin(), s.homework.end(),
                back_inserter(nonzero), 0);
    if (nonzero.empty())
        return grade(s.midterm, s.final, 0);
    else
        return grade(s.midterm, s.final, median(nonzero));
}

```

この関数はまず homework から 0 でない要素を取り出し、それを nonzero という vector に格納します。こうして 0 でない宿題の成績を取り出したら、§ 4.1 の grade を使います。この関数は、試験と宿題の総合点数から最終成績を付けるもので、こうして実際に提出された宿題のメジアンを使って成績を出すことになります。

この関数で新しいところは、`nonzero` に要素を入れるのに `remove_copy` というアルゴリズムを使っている点です。この関数を見る前に、ライブラリはいろいろな種類の「コピー」を提供していることを理解してください。`remove_copy` は `remove` 関数の仕事をしながら、その結果をコピー先にコピーしていくのです。

`remove` 関数は、与えられた値に一致する要素を探し、それをコンテナから「除外」していくものです。² 入力シーケンス内で「除外しなかった」要素だけがコピーされるわけです。この除外の意味はすぐ後で説明します。

`remove_copy` 関数は 3 つの反復子と 1 つの値を引数に取ります。他の多くのアルゴリズムと同様に、最初の 2 つの反復子が入力シーケンスを表しています。3 番目の反復子は、コピー先の頭を指しています。`copy` と同じく、`remove_copy` アルゴリズムは、コピー先に十分なスペースがあるものとして動作します。`nonzero` を必要なだけ拡張するために、ここでは `back_inserter` を使っています。

さあ、これで `remove_copy` の効果を説明できます。これは `s.homework` の 0 でない要素をすべて `nonzero` にコピーするのです。それから、`nonzero` が空かどうかをチェックし、空でなければ `nonzero` の要素のメジアンから最終成績を計算し、空であれば宿題の総合点数は 0 としています。

もちろん、この解析プログラムを完成させるには、`optimistic_median` を呼ぶ成績処理関数を書かなければなりません。それはみなさんの課題ということにしましょう。

6.3 学生の分類 改訂版

第 5 章で不合格者の成績データを `vector` に入れ、もとデータの `vector` からはそれらを取り除くことをしました。これにわかりやすく簡単な方法を使うと、入力データの数が増えるにつれ、効率が非常に悪くなりました。そこで、`vector` の代わりに `list` というものを使うと効率をよくできることを示したのです。しかし、データ構造をこのように変えなくても、アルゴリズムを変えることで似たような効率化が可能であることを後で示すと予告しました。

ここで 2 種類の方法に、ライブラリのアルゴリズムが使えるのです。最初の方法は、ライブラリのアルゴリズムを 2 回使い、すべての要素を 2 回チェックするので、わずかに遅いものです。しかし、より特殊なライブラリアルゴリズムを使うことでチェックを 1 回のみにして効率を上げることができます。

6.3.1 2 パスの方法³

最初の方法は、§ 6.2.4 で 0 でない宿題を探すのに考えたものに似ています。そのときは、`homework` そのものは変わらなかったので、0 点でない宿題の点数を `remove_copy` で別の `vector` にコピーしたのです。今度は、コピーをしながらデータを取り去らなければなりません。

```
vector<Student_info>
extract_fails(vector<Student_info>& students) {
    vector<Student_info> fail;
    remove_copy_if(students.begin(), students.end(),
                  back_inserter(fail), pgrade);
    students.erase(remove_if(students.begin(), students.end(),
                           fgrade), students.end());
```

² 訳注：ここで「除外」は `remove` の訳です。英語の `remove` には「取り除く・破棄する」という意味もありますが、ここでは「破棄」の意味はありません。単に「とばす、無視する」だけです。そのため、「」を付けたのだと、後の方に書かれています。

³ 訳注：入力データを 2 回操作することを 2 パスと言います。

```
fgrade), students.end());
}
return fail;
```

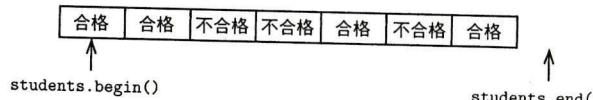
この関数は `vector` を使います。そのためインターフェースは、インデックスではなく反復子を使う § 5.3 のものと同じです。学生データの入った `vector` を受け取り、そこには合格者だけを残し、不合格者は `fail` に入れて戻すわけです。しかし、類似はここで終わりです。§ 5.3 ではコンテナ内を走査するのに `iter` という反復子を使いました。そうして、不合格者の成績を `fail` に入れていました。そこで、`students` からデータを削除するのに `erase` という関数を使いました。今度は、不合格者のデータを `fail` に入れるのに、`remove_copy_if` という関数を使います。この関数は、§ 6.2.4 で紹介した `remove_copy` に似ていますが、これは値ではなく判定関数（述語関数）を使うのです。この関数の引数に使うため、判定関数は `fgrade` の戻り値を逆にするように定義します。

```
bool pgrade(const Student_info& s)
{
    return !fgrade(s);
}
```

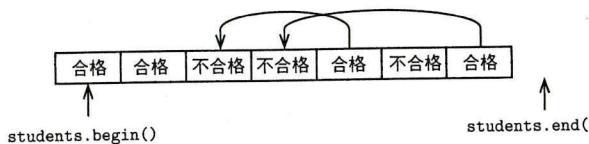
このような判定関数を `remove_copy_if` に渡すと、この判定関数が真を戻す要素は「除外」するようになります。ここで、「除外」とは「コピーしないこと」を意味します。つまり、判定関数を満たさない（偽を戻す）ものだけをコピーすることになるのです。`pgrade` を引数にすることで、`remove_copy_if` は不合格者のデータだけコピーするのです。

次のステートメントは少し複雑です。まず、`remove_if` 関数で不合格者のデータを「除外」しています。ここでも「除外」は実際にデータを破棄するのではなくて「」を付けています。その代わり `remove_if` は、判定関数を満たさないもの（ここでは `fgrade` で偽を戻す要素、つまり合格者）をすべてコピーすることになるのです。

この関数には少し難しいところがあります。それは、`remove_if` はコピー元とコピー先が同じシーケンスになるからです。この関数は、述語関数を満たさないものを、シーケンスの頭の方にコピーしていくのです。たとえば、次のような 7 人の成績データがあったとします。



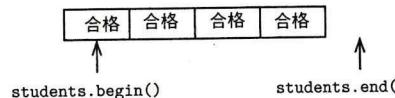
ここで `remove_if` を呼ぶと、最初の 2 要素はそのままで、次の 2 要素は不合格なので「除外」され、別の要素に上書きされてしまう空地 (free space) になります。それから 5 番目の要素がチェックされると、これは合格者なので、最初に「除外」された不合格者の要素のあった場所に上書きされるのです。後は、同様に続けられます。



この結果、合格者 4 人分のデータがシーケンスの前に集められ、後ろの 3 つの要素はそのまま残ります。そして、`remove_if` 関数の戻り値は、「除外」しなかったもの一番最後の 1 つ後ろを指す反復子になります。このため、データのどこまでが有効なのがわかるのです。



次に `erase` を使って、`students` の不要部分を削除しています。`erase` をこのように使うのは始めてです。引数に 2 つの反復子を取り、その間のすべての要素を破棄するのです。`remove_if` の戻り値と `students.end()` の間にすべて削除すると、合格者のデータだけが残るわけです。



6.3.2 1 パスの方法

最初に紹介したアルゴリズムの方法はなかなかよく動作します。しかし、もう少しよくすることもできるのです。§ 6.3.1 の方法では、`students` の各要素を 2 回チェックしています。1 回目は `remove_copy_if` で、もう 2 回目は `remove_if` です。

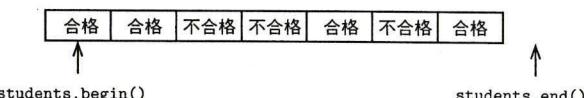
ライブラリのアルゴリズムの中に、単独で都合よく動作するものはありませんが、今考えている問題を別の角度から扱うものがあります。これはシーケンスを受け取り、判定関数を満たすものが満たさないもの前に来るよう並べ替えるものです。

実は、このアルゴリズムには、`partition` と `stable_partition` の 2 つがあります。`stable_partition` は、グループ分け以外の順番は変えないのに対し、`partition` ではもとからある順番が残ることは保証されないのです。たとえば、学生のデータがアルファベット順になっていて、合格・不合格のグループ分け後も、そのグループ内でアルファベット順になっていてほしければ、`partition` ではなく `stable_partition` を使うべきです。

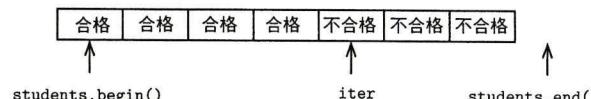
これらのアルゴリズムは後ろのグループの最初の要素を指す反復子を戻します。そのため、不合格者を取り出す関数は次のように書けます。

```
vector<Student_info>
extract_fails(vector<Student_info>& students)
{
    vector<Student_info>::iterator iter =
        stable_partition(students.begin(), students.end(), pgrade);
    vector<Student_info> fail(iter, students.end());
    students.erase(iter, students.end());
    return fail;
}
```

動作を具体的に考えるために、ふたたび次のような 7 人分のデータを考えましょう。



`stable_partition` を呼び出した後は、これが次のようにになります。



そして、`[iter, students.end()]` の範囲にある不合格者のデータをコピーして `fail` を作り、これらを `students` から削除します。

ここで紹介したアルゴリズムを使う方法は、`list` を使う方法とだいたい同じくらいの効率でした。予想どおり、入力が大きくなっていくと、この節のアルゴリズムや `list` を使った方法は、単純な `vector` の方法よりもよくなります。アルゴリズムを利用した 2 つの方法はとてもよいもので、75,000 個のデータを扱う場合、入力部分（のライブラリ）に一番時間をとります。`extract_fails` で考えた 2 つの方法を比べるために、プログラムのこの部分だけの実行時間を計測してみたところ、確かに 1 パス方法は 2 パスの方法の 2 倍の速さであることも確かめられました。

6.4 アルゴリズム、コンテナ、反復子

アルゴリズム、コンテナ、反復子を使うために理解しておくべきことが 1 つあります。それは

アルゴリズムはコンテナの要素に作用するので、コンテナに作用するのではないということです。`sort`、`remove_if`、`partition` は、コンテナ内の要素の位置を変えるのですが、コンテナの性質そのものは変えません。たとえば、`remove_if` は作用したコンテナのサイズを変えません。単に、コンテナ内でコピーをしているだけです。

この区別はアルゴリズムが output 用のコンテナをどう扱うか理解する上で、特に重要です。ここで § 6.3.1 の `remove_if` をもっと詳しく考えてみましょう。これは次のように使われていました。

```
remove_if(students.begin(), students.end(), fgrade)
```

これは `students` のサイズを変えません。`students` の要素が判定関数を満たさない（判定関数を適用すると `false` を戻す）ものは前の方にコピーされますが、他の要素はそのまま残されます。`vector` の要素を削除しそのサイズを小さくしたいなら、自分でそうしなければならないのです。

たとえば、次のようなステートメントを考えました。

```
students.erase(remove_if(students.begin(), students.end(), fgrade),
    students.end());
```

ここで `erase` はその引数にしたがって、シーケンスの要素を削除し、`vector` を変えています。`erase` によって、`students` を短くし、不要なデータを取り去ったのです。ここで `erase` は `vector` のメンバであることに注意してください。これはコンテナの単に要素だけではなくコンテナそのものに作用するのです。

同様に、反復子とアルゴリズム、コンテナの操作の関係も見ておいてください。§ 5.3、§ 5.5.1 で書きましたが、`erase` を使ってコンテナを操作すると、削除された要素の反復子は無効になります。さらに重要なことですが、`vector` や `string` の場合、`erase` や `insert` を使って要素を削除したり挿入したりすると、その要素の後の要素を指す反復子が無効になりました。これらの操作は反復子を無効にできるので、これらの操作をしながら反復子を記録するような場合には注意が必要です。

また、`partition` や `remove_if` は、コンテナの中の要素を動かすので、反復子で指していたものが変わることもあります。これらの関数を実行した後は、反復子が正しい要素を指しているとは限らないのです。

6.5 詳細

記憶寿命の指定：

```
static type variable;
```

ローカルな宣言では、変数 `variable` を `static` なものとする。これは、制御がスコープの外に 1 度出てから戻っても変数の値は保持されていること、また、最初に使われる前に初期化されることが保証されることを意味する。§ 13.4 で別の意味を紹介する。

型： 組み込み型の `void` はいくつか制限された状況で使われる。その 1 つは、関数に戻り値がないことを示すために使われるものである。そのような関数は戻り値を指定しない `return`; で終了するか、関数の最後まで行って終了する。

反復子アダプタ（イテレータアダプタ）： 反復子を戻す関数。よくあるのは、`insert_iterator` を戻すものである。これは付随するコンテナを必要に応じて拡大する反復子である。このような反復子はコピーアルゴリズムでコピー先として安全に使うことができる。これらは`<iterator>` ヘッダに定義されている。

```
back_inserter(c)
```

コンテナ `c` に対し、要素を最後に付け加えるための反復子を戻す。このコンテナは `push_back` を持つていなければならぬ。これは `list`、`vector`、`string` などである。

6.5 詳細

```
front_inserter(c)
```

`back_inserter` と似ているが、コンテナの先頭に要素を付け加えるための反復子。このコンテナは `push_front` を持っていないなければならない。`list` にはこれがあるが、`string` と `vector` はない。

```
inserter(c, it)
```

`back_inserter` と似ているが、要素を `it` の指す要素の直前に挿入する。

アルゴリズム： 特に断らなければ、`<algorithm>` ヘッダに定義されている。

```
accumulate(b, e, t)
```

`t` のコピーを作り、それに $[b, e]$ の範囲の要素を足して行き、その結果を戻す（戻り値が `t` と同じ型になるので `t` の型がとても重要になる）。`<numeric>` に定義されている。

```
find(b, e, t)
```

```
find_if(b, e, p)
```

```
search(b, e, b2, e2)
```

与えられた値を $[b, e]$ のシーケンスから探す。`find` は値 `t` を探し、`find_if` は各要素を判定関数 `p` でテストし、`search` は $[b2, e2]$ で示されるシーケンスを探す。

```
copy(b, e, d)
```

```
remove_copy(b, e, d, t)
```

```
remove_copy_if(b, e, d, p)
```

$[b, e]$ の範囲のシーケンスを `d` で示されるコピー先にコピーする。`copy` はシーケンス全体をコピーする。`remove_copy` は `t` に等しくない要素をコピーする。`remove_copy_if` は判定関数 `p` が `false` を戻す要素のみをコピーする。

```
remove_if(b, e, p)
```

$[b, e]$ の範囲の要素に作用する。判定関数 `p` が `false` を戻す要素は前方に、そうでないものは後方に並べ替える。戻り値は、「除外されなかった」要素のグループの最後の 1 つ後を指す反復子。

```
remove(b, e, t)
```

`remove_if` と似ているが、これは各要素が `t` に等しいかどうかをチェックする。

```
transform(b, e, d, f)
```

$[b, e]$ の範囲の各要素に `f` を適用し、その戻り値を `d` で示される場所に記録する。

```
partition(b, e, p)
```

```
stable_partition(b, e, p)
```

$[b, e]$ の範囲の要素を判定関数 `p` に基づいてグループ分けする。その際、判定関数 `p` が `true` を戻す要素が前に来る。戻り値は、`p` が `false` を戻す最初の要素を指す反復子か、すべての要素について `p` が `true` を戻すなら `e` になる。`stable_partition` では、分類されたグループ内の順番が、もとからあつた順番通りになる。