第３章　C++を探検しよう　：　抽象化のメカニズム

あわてるな！
―ダグラス・アダムス

・はじめに
・クラス
　具象型／抽象型／仮想関数／クラス階層
・コピーとムーブ
　コンテナのコピー／コンテナのムーブ／資源管理／演算子の抑制
・テンプレート
　パラメータ化された型／関数テンプレート／関数オブジェクト／可変個引数テンプレート／別名
・アドバイス

3.1 はじめに

本章の目的は、C++での抽象化と資源管理の概念を示すことである。詳細には踏み込まずに、新しい型＝**ユーザ定義型**(user-define type)を定義して利用する方法を大まかに示す。特に、基本的な性質と実装技法、さらに、**具象クラス**(concrete class)と**抽象クラス**(abstract class)と**クラス階層**(class hierarchy)に関する言語機能を解説する。また、(他の)型とアルゴリズムとでパラメータ化される、型とアルゴリズムのメカニズムを実現するためのテンプレートも取り上げる。ユーザ定義型や組込み型に対する処理は、関数として表現するものであるが、**テンプレート関数**(template function)や**関数オブジェクト**(function object)として一般化することもある。本章で紹介するのは、**オブジェクト指向プログラミング**(object-oriented programming)や**ジェネリックプログラミング**(generic programming)のプログラミングスタイルに必要な言語機能だ。標準ライブラリの機能や利用方法の例題はこの後の二つの章で示す。

なお、ここではプログラミング経験のある読者を想定している。もし経験がなければ、本章を読み進める前に、まずは、Programming:Principles and Practice Using C++ [Stroustrup. 2008]などのテキストを先に読むことが好ましい。プログラミング経験があっても、その言語や使用したアプリケーションが C++のプログラミングスタイルとかけ離れていれば、やはり同様である。もし、この“短い時間の探検”で混乱するようであれば、もっと系統立てて解説を行っている第 6 章から読み始めるのもよいだろう。

第 2 章でも述べたが、この探検では、C++を一皮ずつじっくりめくるのではなく、全体像を丸ごと示す。そのため、C 言語、C++98 の一部、C++11 の新機能を明確には区別しておらず、その歴史的経緯は§1.4 と第 44 章に別途まとめる。

3.2　クラス

C++の言語機能の中核は、**クラス**(class)である、そのクラスは、プログラムのコードがもつ概念を表すユーザ定義型だ。何らかの有用な概念・アイディア・実体などがプログラムにあれば、どんなものでもクラスとして記述するとよい。そすると、頭の中や、設計ドキュメントや、コメントなどに留まらない形態で、その概念がプログラムとなる。組込み型のみを利用するプログラムよりも、きちんと厳選されてたクラスを用いたプログラムのほうが、はるかに理解しやすく健全になる。

そもそも、基礎的な型・演算子・文以外のすべての言語機能は、よりよいクラスを定義して、効率的で、エレガントで、使いやすくて、読みやすくて、理解しやすい、とういうことだ。ほとんどのプログラミング技法は、ある種のクラスの設計と実装の上に成り立つ。プログラマの要求や嗜好は多彩なので、クラスに対するサポートも広範囲にわたる。ここでは、クラスをサポートする 3 種類の基本機能を考えていこう：

- 具象クラス(§3.2.1)
- 抽象クラス(§3.2.2)
- クラス階層内のクラス(§3.2.4)

この 3 種類だけで、数多くの有用なクラスが作り出せる。もちろん、これら 3 種類を、少し変えたり、組合せたりするだけで、さらに多くのクラスが作れる。

3.2.1　具象型

具象クラス(concrete class)の基本的な考えは、“まるで組込み型のように”振る舞うことだ。たとえば、複素数型や無限精度整数は、独自の意味や演算子をもつこと以外は、組込みの int とほとんど同じだ、同様に、vector や string も豊富な機能をもつことを除くと、組込み配列と似ている(§4.2, §4.3.2, §4.4.1)。

具象型の性質を定義することは、その内部データ表現を定義の一部にするということだ。Vector のような例であれば、自身のデータとしては、どこか別の場所に保持されるデータを指すポインタを 1 個あるいは複数個だけもつ。具象クラスの個々のオブジェクトクラスが自身のデータをもつので、時間と空間の効率を最適化した実装が可能になる。特に、以下の点だ：

- 具象型のオブジェクトは、スタック上にも置けるし、静的なメモリにも置けるし、他のオブジェクト内にも置ける(§6.4.2)
- オブジェクトを(ポインタや参照を経由せずに)そのまま利用できる。
- オブジェクトを即座に完全に初期化できる(コンストラクタなどの手段によって：§2.3.2)。
- オブジェクトをコピーできる(§3.3)。

§2.3.2 で vector の例を示したように、内部表現は非公開にできるし、メンバ関数からのみアクセスできるようにすることも可能だ。しかし、データが内部に存在している以上、内部表現を変更すれば、その変更規模にもかかわらず、ユーザは再コンパイルする必要がある。この点は、具象型を

組込み型とまったく同じように扱うために避けなければいけないコストだ。それほど頻繁に変更しない型や、透明性と効率性を重視する局所変数であれば、このコストは現実的であって納得できるものだろう。
具象型の内部表現の大部分を空き領域(ダイナミックメモリ、ヒープ)に置いた上で、クラスオブジェクト内に置かれた部分を経由してアクセスするようにすれば、柔軟性が向上する。Vector や String は、そのように実装されている。このような具象型は、インターフェースを注意深く設計した資源ハンドルともみなせる。

3.2.1.1 算術型

“古典的なユーザ定義の算術型”の一つが complex だ。

```
class complex {
      double re, im;   // 内部データ表現：  2 個の double 変数
public:
      complex(double r, double i) :re{r}, im{i} {}       // 2 個のスカラから complex を構築
      complex(double r) :re{r}, im{0} {}                    // 1 個のスカラから complex を構築
      complex() :re{0}, im{0} {}                              // デフォルトの complex は{0, 0}

      double real() const { return re; }
      void real(double d) { re = d; }
      double imag() const { return im; }
      void imag(double d) { im = d; }

      complex operator+=( complex z )                          // re と im を加えてその結果を返却
                              { re+=z.re, im+=z.im; return *this; }
      complex operator-=( complex z )
                              { re-=z.re, im-=z.im; return *this; }
      complex operator*=( complex );                          // クラスの外のどこかで定義される
      complex operator/=( complex );                         // クラスの外のどこかで定義される
};
```

これは、標準ライブラリ complex(§40.4)を単純化したものだ。このクラス定義は、内部データ表現にアクセスする演算だけを含んでいる。内部データ表現は単純なので、説明は不要だろう。
現実的な理由によって、50 年も前に実装された Fortran との互換性を維持するとともに、一般的な演算をもたせている。complex には、理論的な要求だけでなく、ちゃんと使えるようにするための効率性の要求もある。効率を重視すると、単純な演算はインライン化することになる。すなわち、コンストラクタ、+=、imag()などの単純な演算は、関数呼び出しの機械語が生成されないように実装しなければならない。クラス内で実装する関数は、デフォルトでインライン化される、標準ライブラリのように実用的な耐久性をもつ complex では、インライン化が適切に行われるように、注意深く実装されている。
引数を与えずに呼び出せるコンストラクタは、デフォルトコンストラクタ(default constrauctor)である。すなわち、complex()は、complex のデフォルトコンストラクタである。デフォルトコンストラクタを定義すると、その型の変数が初期化されない、という事態が避けられる。

複素数の実部や虚部を返す関数に付けられた const 指定子は、その関数がオブジェクトを変更しないことの指定だ。

Complex の内部データ表現に直接アクセスする必要がない多くの演算は、クラス定義とは分離して記述できる：

```
complex operator+(complex a, complex b) { return a += b; }
complex operator-(complex a, complex b) { return a -= b; }
complex operator-(complex a) { return { -a.real(), -aimag() }; }     // 単項マイナス
complex operator*(complex a, complex b) { return a *= b; }
complex operator/(complex a, complex b) { return a /= b; }
```

値渡しによって渡される引数はコピーにすぎないので、呼出し側に影響を与えることなく、引数の値は変更できる。また、変更した値は、返却値として利用できる。

==演算子と!=演算子の定義は、単刀直入なものだ：

```
bool operator == (complex a, complex b)          // 等しい
{
        return a.real() == b.real() && a.imag() == b.imag();
}

bool operator != (complex a, complex b)          // 等しくない
{
        return !(a == b);
}

complex aqrt(complex);

// ...
```

ここで定義した complex クラスの利用例を示そう：

```
void f(complex x)
{
        complex a {2.3};    // 2.3 から{2.3, 0.0}を構築
        complex b {1/a};
        complex c {a + z * complex(1, 2.3)};
        // ...
        if ( c != b )
                c = -(b / a) + 2 * b;
}
```

コンパイラは、complex に適応された演算子を、それに対応する関数呼出しに変換する。たとえば、c!=b は operator!=(c,b)となり、1/a は operator/(complex{1},a)となる。

ユーザ定義演算子(“多重定義された演算子”)は、慣例にしたがった上で注意深く利用すべきものだ。文法は言語が決定するので、単項演算子の/は定義できない。また、組込み型の演算子の意味は変更できない。

3.2.1.2 コンテナ

コンテナ(container)は、要素の集合を保持するオブジェクトだ。Vector は、コンテナとして動作するオブジェクト型なので、コンテナである。§2.3.2 で定義した Vector は、立派な double のコンテナだ。容易に理解できる上に、有意な不変条件を確立し、(§2.4.3.2)、アクセス時には範囲をチェックし(§2.4.3.1)、全要素へアクセスするための size()も提供する。しかし、致命的な問題点もある。New でメモリを割り当てているのに、それを解放していないことだ。C++ではガーベジコレクタのインターフェースを提供する(§34.5)ものの、使われなくなったメモリが新しいオブジェクト用に割り当てられることは保証しない。また、ガーベジコレクタを使用できない環境も存在するので、この Vector はよくない実装である。さらに論理を明確にするためや、性能上の理由から、

オブジェクトの解体(§13.6.4)を、より正確に制御する必要に迫られることがある。コンストラクタが割り当てたメモリは、確実に解放されるという保証が必要だ。そのための手段が、**デストラクタ**(destructor)である：

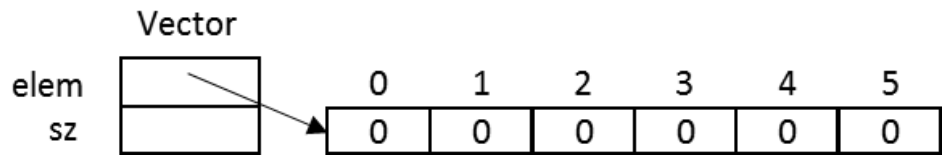
```
class Vector {
private:
        double* elem;           // elem は sz 個の double 配列へのポインタ
        int sz;
public:
        Vector(int s) :elem{new double[s]}, sz{s} // コンストラクタ：資源を確保
        {
                for (int i = 0; i != s; ++i) elem[i] = 0; // 要素を初期化
        }

        Vector() { delete[] elem; }                     // デストラクタ：資源を解放

        double& operator[](int i);
        int size() const;
};
```

デストラクタは、クラス名の直前に補数演算子~付加した名前をもつ。すなわち、コンストラクタを補うものである。Vector のコンストラクタは、new 演算子で空き領域(**ヒープ**や**ダイナミックメモリ**とも呼ばれる)からメモリを割り当てる。そのメモリを delete 演算子で解放するのが、デストラクタだ。この処理に Vector のユーザが介入することはない。ユーザは、組込み型の変数と同じように、単に Vector を構築して利用するだけだ。たとえば：

```
void int(int n)
{
        Vector v(n);

        // ... v を利用 ...

        {
                Vector v2(2*n);
                // ...v と v2 を利用 ...
        } // v2 はここで解体される

        // ...v を利用 ...

} // v はここで解体される
```

命名法、スコープ、メモリ割当て、生存期間などについて、Vector は、int や char などの組込み型と同じ規則にしたがう。オブジェクトの生存期間の制御方法の詳細については、§6.4 を参照しよう。この Vector では、記述を簡潔にするために、エラー処理を省いている。エラー処理については§2.4.3 を参照しよう。

コンストラクタとデストラクタの組合せは、多くのエレガントな技法の基礎となっている。特に、C++でのほとんどの汎用資源管理技法(§5.2, §13.3)の基本だ。Vector を図示してみよう：



コンストラクタは、要素のためのメモリを確保して Vector のメンバを適切に初期化する。デストラクタは、確保したメモリを解放する。この**データハンドルモデル**(handle-to-data model)は、オブジェクトの生存期間中に大きさが変化するデータを管理する手法として、極めて広く使われている。コンストラクタで資源を獲得して、デストラクタで解放する技法は、資源獲得時初期化＝RAII(Resource Acquisition Is Initialization)と呼ばれ、“裸の new 演算”を削減する効果がある。すなわち、メモリ割当て処理は、一般のコードでは行わずに、うまく設計された抽象化の実装に閉じ込める。同様に、“裸の delete 演算”も除去すべきだ。裸の new と裸の delete を除去することで、エラーの確立を大きく下げられるし、資源リークも防ぎやすくなる(§5.2)。

3.2.1.3 コンテナの初期化

コンテナは内部にデータをもつので、コンテナに対してデータを入れるための簡便な方法が必要だ。適切な要素数をもつ Vector を作成して、その後で要素を代入することも可能だが、通常はもっとエレガントな別の方法を使う。ここでは、次の二点だけを説明しよう：

- **初期化子並びコンストラクタ**：要素の並びで初期化する。
- push_back()：既存データの末尾に新しいデータを追加する。

これらは、以下のように宣言できる：

```
class Vector {
public:
        Vector(std::initializer_list<double>);    // 並びで初期化
        // ...
        void push_back(double);                   // 末尾に追加して要素が 1 個増える
        // ...
};
```

Push_back()は、入力される不定個数の数値を Vector の要素として追加する際に有用だ：

```
Vector read(istream& is)
{
        Vector v;
        for (double d; is>>d;)    // 浮動小数点値を d に読み込む
                v.push_back(d);           // v に d 加える
        return v;
}
```

入力のためのループは、ファイル終端(end-of-file)に到達するか、または入力形式エラーが発生するまで処理を繰り返す。ループが終了するまで読み取った数値は Vector に追加される。ループ終了時の v の要素数は、読み取った個数となる。ここでは d のスコープをループに一致させるために、より一般的な while 分の代わりに for 文を使っている。Push_back()の実装については、§13.6.4.3 で改めて解説する。また、read()が膨大なデータ個数を返した場合にも耐える、低コストのムーブコンストラクタを Vector に実装する方法は、§3.3.2 で解説する。

初期化子並びコンストラクタを実装する際に利用する std::initializer_list は、標準ライブラリが実装する型として、コンパイラに知らされる。{1,2,3,4}のような{}並びを与えると、コンパイラは initializer_list 型オブジェクトを作る。そのため、以下のようなコード記述可能だ：

```
Vector v1 = {1,2,3,4,5};                    // v1 の要素は 5 個
Vector v2 = {1.23, 3.45, 6.7, 8};           // v2 の要素は 4 個
```

Vector の初期化子並びコンストラクタは、以下のようにも定義できる：

```
Vector::Vector(std::initializer_list<double> lst) // 並びによる初期化
        :elem{new double[lst.size()]}, sz{static_cast<int>(lst.size())}
{
        copy(lst.begin(), lst.end(), elem);             // lst から elem にコピー
}
```

3.2.2 抽象型

Complex や Vector などの型は、その内部データ表現が定義の一部となっているので、**具象型**(concrete type)と呼ばれる。これは、組込み型と共通する点だ。一方、**抽象型**(abstract type)は、その実装の詳細からユーザを完全に隔離したものだ。その実現には、内部データ表現をインターフェースから切り離して、局所変酢を捨て去る必要がある。抽象型の内部については何も(その大きさすら)知らされていない以上、オブジェクトは常に空き領域から割り当てた上で(§3.2.1.2, §11.2)、必ず参照かポインタを用いてアクセスしなければならない。(§2.2.5, §7.2, §7.7)。

まず、Container というクラスのインターフェースを定義する。これは、先ほどの Vector よりも、抽象度を高めたものだ：

```
class Container {
public:
        virtual double& operator[](int) = 0;            // 純粋仮想関数
        virtual int size() const = 0;                   // const メンバ関数(§3.2.1.1)
        virtual ~Container() {}                         // デストラクタ(§3.2.1.2)
};
```

このクラスは、この後で定義されることになるコンテナの純粋なインタフェースである。virtual は“このクラスから派生したクラスで再定義される”ことを意味する。当然のことだが、virtual と宣言した関数は**仮想関数**(virtual function)と呼ばれる。Container のインタフェースを実装するのは、Container から派生したクラスだ。妙に目立っている=0 という表記は、この関数が**純粋仮想**(pure virtual)関数であることを表す。すなわち、Container から派生したクラスは、この関数を必ず実装しなければならない。そのため、Container 型そのもののオブジェクトを構築することは不可能だ。Container は、operator[]()と size()関数を実装するクラスのインタフェースだけを提供する。純粋仮想関数をもつクラスは、**抽象クラス**(abstract class)と呼ばれる。

Container は次のように利用できる：

```
void use(Container& c)
{
        const int sz = c.size();

        for (int i=0; i!=sz; ++i)
                cout << c[i] << '¥n';
}
```

use()内で、実装の詳細に一切関知せずに、Container インタフェースを利用していることに注目しよう。実装を提供する型を正確に把握しないまま、size()と[]を利用しているのだ。他のさ

まざまなクラスにインタフェースを提供する、Container のようなクラスは、一般に**多相型**(polymorphic type)と呼ばれる(§20.3.2)。

抽象クラスに共通することだが、Container もコンストラクタをもたない。初期化すべきデータを一切もたないからだ。しかし、その一方で Container は、デストラクタをもっている。そして、そのデストラクタは virtual である。これも抽象クラスの共通点だ。抽象クラスのオブジェクトは、参照やポインタを経由して操作されるし、ポインタ経由で Container を解体する側は実装がどんな資源を保持しているかが分からないからである。§3.2.4 も参照しよう。

抽象クラス Container が定義するインタフェースを実装するコンテナは、その内部で具象クラス Vector も利用できる：

```
class Vector_container : public Container {        // Vector_Container は Container を実装
        Vectorv;
public:
        Vector_container(int s) : v(s) { }         // s 個の要素の Vector
        ~Vector_container() {}

        double& operator[](int i) { return v[i]; }
        int size() const { return v.size(); }
};
```

“: public”は、“〜から派生する(is derived from)”とか“〜の部分型である(is a subtype Of)”と読める。Vector_container クラスは Container クラスから派生した(derived)といい、Container クラスを、Vector_container クラスの基底(base)と呼ぶ。なお、その他にも、Vector_container をサブクラス(subclass)と呼び、Container をスーパクラス(superclass)と呼ぶこともある。派生クラスは、基底クラスからメンバを継承したともいい、一般に基底クラスと派生クラスの関係は、継承(inheritance)と呼ばれる。

メンバ operator[]()と size()は、基底クラス Container のメンバをオーバライド(override)していると表現する(§20.3.2)。ここでは、デストラクタ~Vector_container()は、基底クラスのデストラクタ~Container()をオーバライドしている。メンバのデストラクタ~Vector()が、クラスのデストラクタ~Vector_container()によって暗黙裏に呼び出されることに注意しよう。

さて、実装の詳細に一切関知せずに Container を利用する use(container&)のような関数を使うためには、その操作が可能なオブジェクトを、別の関数を作る必要がある。たとえば：

```
void g()
{
        Vector_container vc(10);
        use(vc);
}
```

use()は Vector_container のことは何も知らないが、Container のインタフェースだけは知っているので、Container を実装する別のクラスに対してもまったく同じように動作する。たとえば：

```
class List_container : public Container { // List_container は Container を実装
        std::list<double> ld;             // 標準ライブラリ：double のリスト(§4.4.2)
public:
        List_container() { }      // 空のリスト
        List_container(initializer_list<double> il) : ld {il} { }
```

```
        ~List_container() {}
        double& operator[](int i);
        int size() const { return ld.size(); }
};

double& List_container::operator[](int i)
{
        for (auto& x : ld) {
                if (i==0) return x;
                --i;
        }
        throw out_of_range("List container");
}
```

ここで、クラスの内部データ表現は、標準ライブラリ List<double>だ。通常ならば、私は添字演算をもつ List のコンテナを実装するようなことは行わない。List の添字演算のパフォーマンスが、Vector よりも極めて低いからだ。ここでは、通常とは大幅に異なる実装を示したわけである。

さて、List_container オブジェクトを作って、use()を利用してみる：

```
void h()
{
        List_container lc = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
        use(lc);
}
```

ここで重要なことは、use(Container&)が、受け取る引数が Vector_container なのか、List_container なのか、あるいは、それ以外のコンテナなのかをまったく知らない、というよりも、知る必要すらない点だ。use(Container&)は、あらゆる種類の Container に利用できるものであり、Container が定義するインタフェースのみを知っている。その結果、List_container の実装が変更されても、Container から派生したまったく新しいクラス定義が追加されても、use(Container&)を再コンパイルする必要はない。

この柔軟性の代償は、オブジェクトの操作を、ポインタや参照経由で行わなければならないことだ(§3.3, §20.4)。

3.2.3 仮想関数

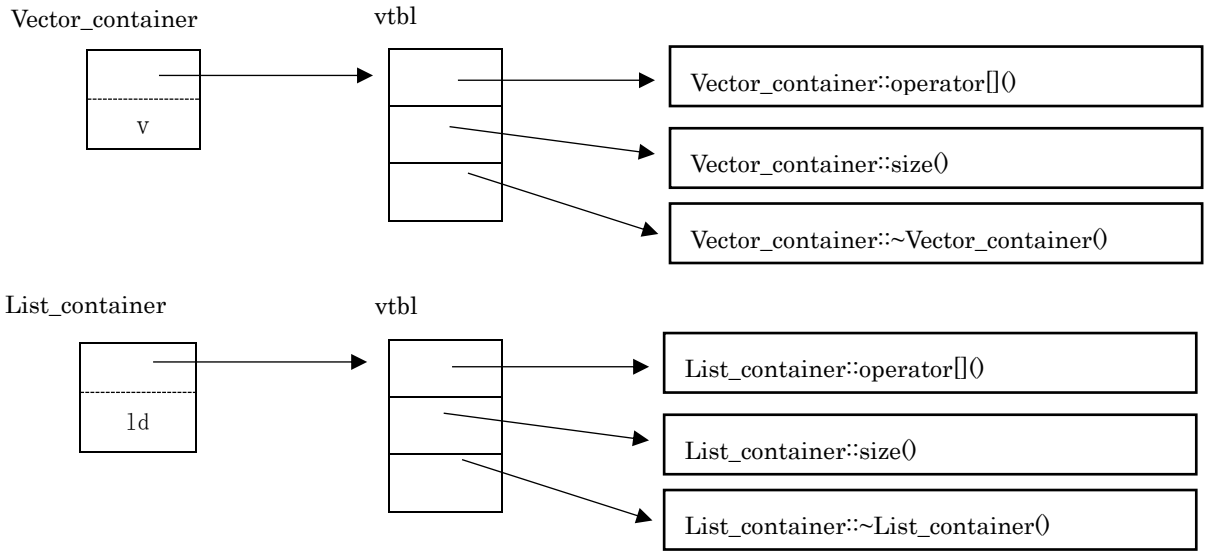
Container の利用例を、もう一度考えよう：

```
void use(Container& c)
{
        const int sz = c.size();
        for (int i=0; i!=sz; ++i)
                cout << c[i] << '¥n';
}
```

use()内の c[i]演算は、どのようにして正しく operator[]()へと解決されるのだろう？ h()が use()を呼び出した場合は、List_container の operator[]()が実行されなければならない。

また、g()が use()を呼び出したい場合は Vector_container の operator[]()が実行されなければならない。この解決には、正しい関数を実行時に選択して呼び出すための情報が、Container オブジェクト内に必要となる。通常は、コンパイラが、仮想関数名を、関数ポインタのテーブル内

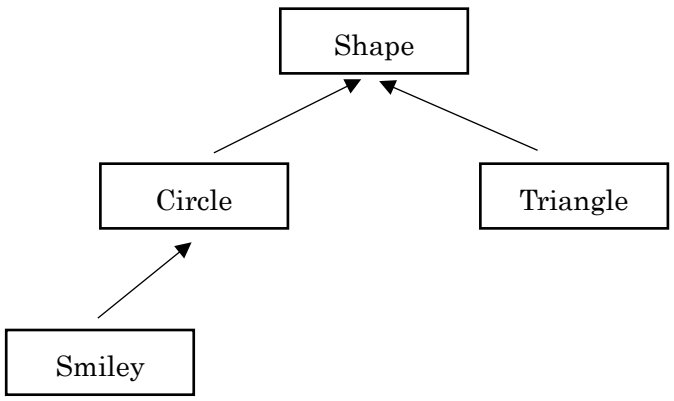
のインデックスへと変換するように実装される。このテーブルは、一般に仮想関数テーブル(virtual Function table)あるいは単に vtbl と呼ばれる。仮想関数をもつ各クラスが専用の vtbl をもち、仮想関数を特定する。図にすると、次のようになる：



関数が vtbl 中に置かれることによって、呼出し側は、オブジェクトの大きさやデータレイアウトなを知ることなく、オブジェクトを適切に利用できるのだ。呼出し側の実装は、Container 内の Vtbl を指すポインタの位置と各仮想関数の vtbl 内でのインデックスだけを知っていればよい。この仮想呼出しの効率性は、“通常の関数呼出し”と比べても遜色ない(その差は 25%以内だ)。メモリ空間のオーバヘッドは、仮想関数をもつクラスのオブジェクト 1 個ごとに 1 個のポインタ、そして、仮想関数をもつクラスごとに 1 個の vtbl である。

3.2.4 クラス階層

Container の例は、極めて単純なクラス階層の一例だ。クラス階層(class hierarchy)は、(public などによる)派生によって束ねられるクラス群のことである。“消防車は貨物自動車の一種であり、貨物自動車は乗り物の一種である”や“ニコちゃんマーク(smiley face)は円の一種であり、円は図形の一種である”などの概念を表現するために、階層構造をもつクラスを利用する。たとえば、数百ものクラスからなる巨大な階層であれば、深さも幅も増えるのが、一般的だ。少しだけ現実的かつ古典的な例として、いくつかのスクリーン上の図形を考えてみよう：



矢印は、継承関係を表す。たとえば、Circle クラスは Shape クラスから派生している。この単純

なダイアグラムをコードで表現しよう。最初に定義するのは、すべて図形に共通する一般的な性質をもつクラスである：

```
class Shape {
public:
        virtual Point center() const = 0;          // 純粋仮想
        virtual void move(Point to) = 0;

        virtual void draw() const = 0;             // 現在の"Canvas"に描画
        virtual void rotate(int angle) = 0;

        virtual ~Shape() {}                        // デストラクタ
};
```

当然、この例のインタフェースは抽象クラスである。内部データに注目すると、すべての Shape に共通するものは、(vtbl を指すポインタを除くと)何もない。この定義が与えられると、図形のポインタのベクタを操作する汎用関数が記述できる：

```
void rotate_all(vector<Shape*>&, v, int angle)          // v の全要素を angle 度だけ回転
{
        for (auto p : v)
                p->rotate(angle);
}
```

特定の図形を定義するには、それが一種の Shape クラスであることと、(仮想関数を含めた)特有の性質を明確にする必要がある：

```
class Circle : public Shape {
public:
        Circle(Point p, int rr); // コンストラクタ

        Point center() const { return x; }
        void move(Point to) { x = to; }

        void draw() const;
        void rotate(int) {}              // 単純で優れたアルゴリズム
private:
        Point x;// 中心
        int r;          // 半径
}
```

さて、この Shape と Circle は、先ほどの Container と Vector_container の例と比較して、特に新しいものではない。定義を続けよう：

```
class Smiley : public Circle { // ニコちゃんマークのために Circle を基底として利用
public:
        Smiley(Point p, int r) : Circle{p, r}, mouth{nullptr} { }

        ~Smiley()
        {
                delete mouth;
                for (auto p : eyes) delete p;
        }

        void move(Point to);

        void draw() const;
        void rotate(int);
```

```
        void add_eye(Shape* s) { eyes.push_back(s); }
        void set_mouth(Shape* s);
        virtual void wink(int i);        // i 番のウインクした目

        // ...

private:
        vector<Shape*> eyes;                        // 一般的に目は 2 個
        Shape* mouth;
};
```

メンバ関数 puch_back()は、受け取った引数を vector(ここでは eyes)に追加して、要素数を 1 だけインクリメントする。

Smiley::draw()の定義は、Smiley の基底の draw()と、メンバの draw()とを呼び出すことによって実現できる：

```
void Smiley::draw()
{
        Circle::draw();
        for (auto p : eyes)
                p->draw();
        mouth->draw();
}
```

Smiley が、自身の目を標準ライブラリの vector に保存していることと、そのデストラクタによって解体していることに注目しよう。Shape のデストラクタは virtual であり、Smiley のデストラクタが、それをオーバライドしている。Virtual なデストラクタは、抽象クラスでは重要な意味をもつ。というのも、派生クラスのオブジェクトが、基底である抽象クラスのインタフェース経由で操作されることがよくあるからだ。しかも、基底クラスへのポインタを介して破棄される可能性もある。その際、仮想関数呼出しの仕組みによって、正しいデストラクタが実行される。そして、呼び出しされたデストラクタは、暗黙裏にメンバのデストラクタと基底クラスのデストラクタとを呼び出すのだ。

ここに示す、簡単化した例において、顔を表している円に対して、目と口を適切に配置するのは、プログラマの仕事である。

派生によって新しいクラスを定義する際は、データメンバーや演算子を追加できる。それによって、柔軟性が飛躍的に向上する一方で、混乱を招いたり貧弱な設計を生むこともある。第 21 章を参照しよう。クラス階層には、以下に示す二つの利点がある：

- **インタフェース継承**(interface inheritance)：派生クラスのオブジェクトは、基底クラスのオブジェクトが要求されるあらゆる箇所で利用できる。すなわち、基底クラスは、派生クラスのインタフェースとして振る舞う。Container クラスと Shape クラスが、その例だ。このようなクラスは、抽象クラスと呼ばれる。
- **実装継承**(interface inheritance)：基底クラスは、派生クラスの実装を単純化する関数とデータを提供する。Smiley における、Circle のコンストラクタと Circle::draw()の利用が、その例である。通常、このような基底クラスは、データメンバとコンストラクタをもつ。

具象クラス、特に内部データが小規模な具象クラスは、組込み型と極めて似ている。似ているの

は、局所変数として定義できる、名前を通じてアクセスできる、コピーができる、などの点である。しかし、クラス階層内のクラスは組込み型とは似ていない。異なる点は、new によって空き領域から確保したり、ポインタや参照によりアクセスしたりするのが一般的であることだ。図形を記述したデータを入力ストリームから読み取って、それに対応する Shape オブジェクトを構築する関数を考えよう：

```
enum class Kind { circle, triangle, smiley };

Shape* rad_shape(istream& is) // 入力ストリーム is から shape の記述を読み込み
{
        // ... shape の先頭部を is から読み込んで Kind k を判断...
        switch (k) {
        case Kind::circle:
                // circle のデータ{Point, int}を p と r に読み込む
                return new Circle{p, r};
        case Kind::triangle:
                // triangle のデータ{Point, Point, Point}を p1 と p2 と p3 に読み込む
                return new Triangle{p1, p2, p3};
        case Kind::smiley:
                // smiley のデータ{Point, int, Shape, Shape, Shape}を p と r と e1 と e2 と m に読み込む
                Smiley* ps = new Smiley{p, r};
                ps->add_eye(e1);
                ps->add_eye(e2);
                ps->set_mouth(m);
                return ps;
        }
}
```

この図形読取り関数を利用する例は、以下のようになる：

```
void user()
{
        std::vector<Shape*> v;
        while (cin)
                v.push_back(read_shape(cin));
        draw_all(v);            // 全要素に対して draw()を呼び出す
        rotate_all(v);          // 全要素に対して rotate(45)を呼び出す
        for (auto p : v) delete p;      // 要素の delete を忘れないようにする
}
```

当然、この例は、単純化したものだ。特に、例外処理については単純化している。とはいえ、User()が、処理対象のオブジェクトの実際の種類についてまったく知らないままに、自身の処理を行えることが、はっきりと分かる例になっている。いったん user()のコードをコンパイルしておけば、その後でプログラムに新しい Shape が追加されても、再コンパイルせずに利用する。なお、Shape を指すポインタは、user()の外に存在しないので、そのオブジェクトを破棄するのは user()の責任となることに注意しよう。ここでは、delete 演算子によって破棄している。その動作は、Shape の仮想デストラクタに強く依存している。そのデストラクタは virtual なので、delete が最派生クラスのデストラクタを呼び出すのだ。これは、本当に重要なことだ。というのも、派生クラスは、いろいろな種類の解放すべき資源(ファイルハンドル、ロック、出力ストリームなど)をもっている可能性があるからだ。なお、この例の場合、Smiley は、自身がもっている eyes と mouth のオブジェクトの破棄を行う。

経験豊富なプログラマは、以下の二点の誤りの可能性が残っていることにきづくだろう：

・ユーザは、read_shape()が返却したポインタの delete に失敗するかもしれない。

・Shape ポインタのコンテナ所有者は、ポインタが指すオブジェクトを delete しないかもしれない。

この意味では、空き領域に確保したオブジェクトのポインタを関数が返却することは、危険なことだといえる。

これら二点の問題の解決策が、“裸のポインタ”の代わりに、標準ライブラリの unique_ptr (§5.2.1)を返して、コンテナに unique_ptr をもたせる方法だ：

```
unique_ptr<Shape>read_shape(istream& is)  // 入力ストリーム is から shape の記述を読み込む
{
        // ... shape の先頭部を is から読み込んで Kind k を判断 ...
        switch (k) {
        case Kind::circle:
                // circle のデータ{Point, int}を p と r に読み込む
                return unique_ptr<Shape>{new Circle{p, r}};                // §5.2.1
        // ...
}

void user()
{
        vector<unique_ptr<Shape>> v;
        while (cin)
                v.push_back(read_shape(cin));
        draw_all(v);              // 全要素に対して draw()を呼び出す
        rotate_all(v, 45);        // 全要素に対して rotate(45)を呼び出す
}       // すべての Shape は暗黙裏に解体される
```

これで、オブジェクトの所有者が unique_ptr になる。しかも、unique_ptr は、スコープから抜け出るときに、不要になったオブジェクトを delete する。

Unique_ptr バージョンの user()を動作させるには、vector<unique_ptr<Shape>>を引数に受け取る draw_all()と rotate_all()が必要だ。このような_all()関数を、たくさん記述するのは面倒なので、代替策を§3.4.3 で示すことにする。

3.3 コピーとムーブ

オブジェクトは、デフォルトでコピー可能である。この点は、ユーザ定義型でも組込み型でも同じだ。コピー演算のデフォルトの意味は、メンバ単位のコピーである。すなわち、すべてのメンバがコピーされる。ここで、§3.2.1.1 の complex を例に考えよう：

```
void test(complex z1)
{
        complex z2 {z1}; // コピー初期化
        complex z3;
        z3 = z2;                  // コピー代入
        // ...
}
```

これで、z1、z2、z3 はすべて同じ値になる。というのも、代入と初期化の両方が、メンバをコピーするからだ。

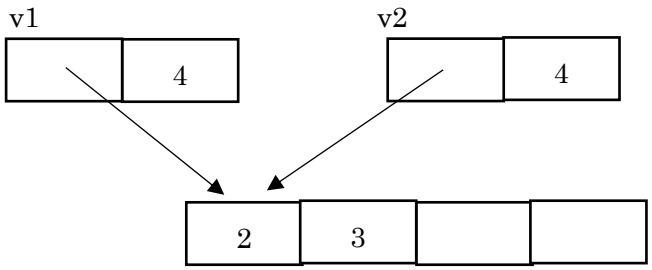
クラスを設計する際は、オブジェクトがコピーされる可能性とコピーの方法を必ず検討しなければならない。単純な具象型であれば、メンバあー単位のコピーが正しいセマンティクスとなることが多い。しかし、Vector のような高度な具象型では、メンバ単位のコピーは、正しいセマンティクスとはならない。また、抽象型では、メンバ単位のコピーは、まず、ありえない。

3.3.1 コンテナのコピー

あるクラスが、**資源ハンドル**(resource handle)、すなわち、ポインタ経由でオブジェクトにアクセスしなければならないクラスであれば、デフォルトのメンバ単位のコピーは、災難となる場合がほとんどだ。メンバー単位のコピー動作は、資源ハンドルの不変条件(§2.4.3.2)に違反する。たとえば、デフォルトのコピーを行うと、コピーされた Vector は、元のオブジェクトと同じ要素を参照することになってしまう：

```
void bad_copy(Vector v1)
{
        Vector v2 = v1;          // v1 の内部データ表現を v2 にコピー
        v1[0] = 2;                       // v2[0]も 2 となる！
        v2[1] = 3;                       // v1[1]も 3 となる！
}
```

V1 が 4 個の要素をもつとしよう。実行結果を図で表すと、以下のようになる：



幸いなことに、Vector がデストラクタをもっているという事実が、デフォルトの(メンバ単位の)コピー動作のセマンティクスが誤りであるという強いヒントを与えてくれる。このような場合、コンパイラは、少なくとも警告を発すべきだ(§17.6)。より適切なコピー動作の定義が必要である。

クラスオブジェクトのコピー動作は、二つのメンバとして定義される。**コピーコンストラクタ**(copy constructor)と、**コピー代入**(copy assignment)である：

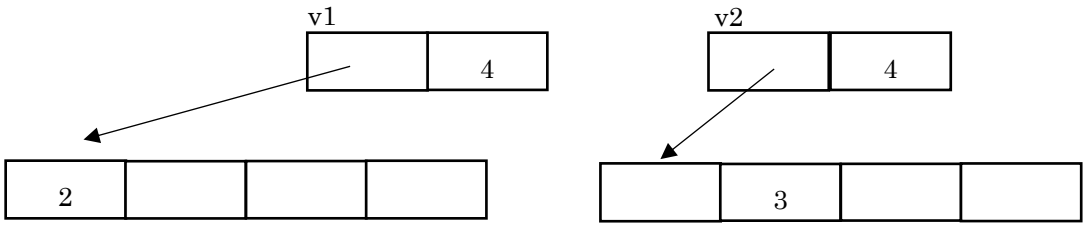
```
class Vector {
private:
        double* elem;    // elem は sz 個の double 配列へのポインタ
        int sz;
public:
        Vector(int s);                    // コンストラクタ：不変条件を確立して資源を獲得
        ~Vector() { delete[] elem; }      // デストラクタ：資源を解放

        Vector(const Vector& a);                  // コピーコンストラクタ
        Vector& operator=(const Vector& a);       // コピー代入

        double& operator[](int i);
        const double& operator[](int i) const;
```

```
        int size() const;
};
```

Vector のコピーコンストラクタが行うべきことは、必要な要素用のメモリを確保して、そこに要素をコピーすることである。そうすると、Vector のコピー構築後に、コピー元オブジェクトとは別に、同じ値の要素をもてるようになる：

```
Vector::Vector(coonst Vector& a)          // コピーコンストラクタ
        :elem { new double[a.sz] },             // 要素用の領域を確保
         sz {a.sz}
{
        for (int i = 0; i != sz; ++i)           // 要素をコピー
                elem[i] = a.elem[i];
}
```

今回は、v2=v1 を実行した様子は、以下のようになる：



もちろん、コピーコンストラクタだけでなく、コピー代入も必要だ：

```
Vector& Vector::oprator=(const Vector& a) // コピー代入
{
        double* p = new double[a.sz];
        for (int i = 0; i != a.sz; ++i)
                p[i] = a.elem[i];
        delete[] elem;                    // 古い要素を delete
        elem = p;
        sz = a.sz;
        return *this;
}
```

this は、メンバ関数内で定義ずみのポインタであって、そのメンバ関数を起動したオブジェクトを指す。

クラス X のコピーコンストラクタとコピー代入演算は、通常 constX&型の引数を受け取る。

3.2.2 コンテナのムーブ

コピー演算の制御は、コピーコンストラクタとコピー代入を定義することによって行える。しかし、大規模なコンテナのコピーは高コストになり得る。次の例を考えてみよう：

```
Vector operator+(const Vector a, const Vector& b)
{
        if (a.size() != b.size())
                throw Vector_size_mismatch{};

        Vector res(a.size());
        for (int i = 0; i != a.size(); ++i)
                res[i] = a[i] + b[i];
        return res;
}
```

+演算関数からのリターン時は、局所変数 res のコピーが作られたうえで、この関数の呼び出し側がアクセス可能な場所にそのコピーが置かれることになる。たとえば、+演算子は以下のように利用される：

```
void f(const Vector& x, const Vector& y, const Vector& z)
{
        Vector r;
        // ...
        r = x + y + z;
        // ...
}
```

このコードは、Vector を少なくとも 2 回コピーする(+演算子を使用するたびに 1 回コピーされるからだ)。もし Vector が大規模で、たとえば 10,000 個もの double 型をもつのであれば、ちょっと厄介だ。もっとも困るのは、operator+()内の res がコピー後に利用されないことである。ここで行いたいのは、コピーではなく、単に関数から演算結果を取り出すことである。すなわち、Vector を**ムーブ**(move)したいのであって、**コピー**(copy)したいのではない。幸いなことに、ここで行いたいことは、以下のように記述できる：

```
class Vector {
        // ...

        Vector(const Vector& a);                    // コピーコンストラクタ
        Vector& operator=(const Vector& a);         // コピー代入

        Vector(Vector&& a);                                     // ムーブコンストラクタ
        Vector operator=(Vector&& a);               // ムーブ代入
};
```

このように定義すると、関数の外に演算結果を転送する際に、コンパイラは**ムーブコンストラクタ**(move constructor)を使うようになる。そのため、r=x+y+z では、Vector は、一切コピーされることなく、ムーブだけが行われる。

例によって、Vector のムーブコンストラクタの定義は、ちょっとしたものだ：

```
Vector::Vector(Vector&& c)
        :elem{a.elem},              // a から“要素を盗む”
        sz{a.sz}
{
        a.elem = nullptr;                   // もはや a には要素はまったくない
        a.sz = 0;
}
```

&&は“右辺値参照”を意味し、右辺値をバインドできる参照である(§6.4.1)。ここで、“右辺値”とは“左辺値”の対になる言葉である。大雑把にいうと、左辺値は“代入演算の左側に記述できるもの”である。最初の説明として、右辺値を少々不正確に説明すると、関数が返す整数などのような、代入できない値のことである。そのため、右辺値参照は、他の誰も代入を行えない何かを参照するものであり、安全に値を“盗む”ことができるものだ。Vector クラスの operator+()内の局所変数 res は、その例である。

ムーブコンストラクタは、const の引数を受け取らない。ムーブコンストラクタが、引数から値を削除するからである。なお、**ムーブ代入**(move assignment)も同じように定義できる。

右辺値参照が、初期化子として、あるいは代入演算の右オペランドとして利用された場合は、ムーブ演算となる。

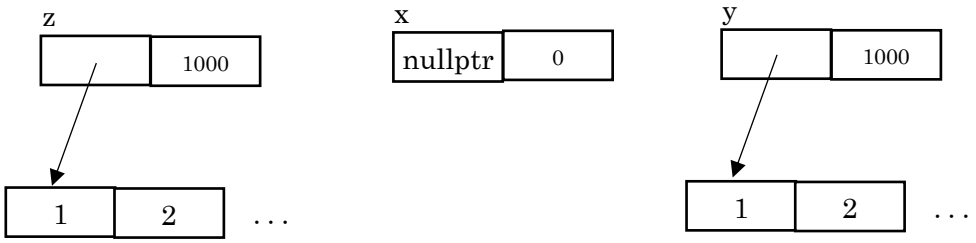
ムーブ後に、ムーブ元オブジェクトは、デストラクタが実行できる状態へと遷移する。通常、ムーブ元オブジェクトに対する代入は行えるようになっている(§17.5、§17.6.2)。

ある値がもう利用されることがないことをプログラマが知っていたとしても、残念ながら、そのことをコンパイラが検出できるわけではない。そのため、プログラマがコンパイラに明示的に教えることができる：

```
Vector f()
{
        Vector x(1000);
        Vector y(1000);
        Vector z(1000);
        // ...
        z = x;                  // コピーを手に入れる
        y = std::move(x);       // ムーブを手に入れる
        // ...
        return z;
}
```

標準ライブラリの move()は、引数の右辺値参照を返す。

Return 直前の状態は、以下のようになっている。



Z が(return によって)ムーブされて解体される際は、x と同様に、その中身は空(要素が 1 個もない状態)となる。

3.3.3 資源管理

コンストラクタ、コピー演算、ムーブ演算、デストラクタを定義することで、プログラマは、(コンテナ内の要素などの)他のオブジェクトに所有されるオブジェクトの生存期間を完全に制御できるようになる。さらに、ムーブコンストラクタによって、スコープをまたぐオブジェクトのムーブも簡潔かつ低コストに実現できる。スコープの外へと、コピーできない、あるいは、コピーしたくない、というオブジェクトを、スコープ外へと簡潔かつ低コストにムーブできる。平行処理を実現する標準ライブラリの thread(§5.3.1)と、百万個もの double をもつ Vector を考えてみよう。前者はコピー不可能なものであり、後者はコピーを行いたくないものだ：

```
std::vector<thread> my_thread;

Vector init(int n)
{
```

```
        thread t {heartbeat};              // heartbeat を並行的に実行(自身の thread 上で)
        my_mythread.push_back(move(t));    // t を my_threads にムーブ
        // ... その他の初期化 ...
        Vector vec(n);
        for ( int i = 0; i < vec.size(); ++i) vec[i] = 777;
        return vec;                        // heartbeat を開始して v を初期化
}

auto v = init(10000);
```

Vector や thread のような資源は、多くの場合で、ポインタに変わる優れた代替案となることがある。実際、標準ライブラリの unique_ptr などの“スマートポインタ”も、その実態は資源バンドルである(§5.2.1)。

ここでは、thread を保持するために、標準ライブラリの Vector を利用している。解説の都合上、§3.4.1 までは、Vector を要素の型でパラメータ化を避けているからだ。

さて、new と delete をアプリケーションから除去するのとほぼ同じ方法で、資源ハンドルを指すポインタも除去できる。いずれの場合も、オーバヘッドを伴いわない、簡潔で保守しやすいコードとなる。特に、強い資源安全(strong resource sefety)、すなわち、資源の概念の一般化によってリークを排除できることは、重要だ。Vector が保持するメモリ、thread が保持するシステムスレッド、Fstream が保持するファイルハンドルなどがその例である。

3.3.4 演算子の抑制

階層内のクラスに対して、デフォルトのコピーやムーブを利用すると、ほとんどの場合は惨事につながる。というのも、基底を指すポインタだけを与えられても、派生クラスのメンバについては何もわからないからだ(§3.2.2)。もちろん、コピーの方法も分からない。最善の策は、デフォルトのコピー演算、ムーブ演算を削除(delete)することだ。すなわちデフォルト定義の除去である：

```
class Shape {
public:
        Shape(const Shape&) = delete;              // コピー演算はなくなる
        Shape& operator=(const Shape&) = delete;

        Shape(Shape&&) = delete;                   // ムーブ演算はなくなる
        Shape& operator=(Shape&&) = delete;
        ~Shape();
        // ...
};
```

こうしておけば、Shape をコピーしようとすると、コンパイラが検出することになる。もしクラス階層内のオブジェクトをコピーする必要があれば、一種のクローン関数を定義する(§22.2.4)。

なお、この場合は、コピー演算とムーブ演算を delete し忘れても、実害は発生しない。ユーザが明示的にデストラクタを宣言したクラスに対しては、ムーブ演算が暗黙裏には生成されないからだ。さらにいうと、この場合のコピー演算の生成は、非推奨とされている。(§44.2.3)。この点こそが、たとえコンパイラがデストラクタを暗黙裏に生成するような場合であっても、デストラクタを明示的に定義する理由の一つである(§17.2.3)。

クラス階層内の基底クラスは、コピー演算の対象とはしたくないオブジェクトのよい例だ。一派に、資源ハンドルは、単にメンバをコピーするだけでは、複製がおこないからだ(§5.2、§17.2.2)。

=delete は汎用的なものである。そのため、あらゆる演算に対して適用できる(§17.6.4)。

3.4 テンプレート

ベクタを使用するユーザが、必ずしも double のベクタを使いたいわけではない。ベクタは一般的な概念であって、浮動小数点の詳細には存在しない。当然、ベクタ内の要素型は、同区立して表現した方が望ましい。テンプレート(template)は、一連の型や値をパラメータ化した、クラスもしくは関数であり、極めて汎用的な概念を表現する。テンプレートに対して、要素型である Double などを引数として指定すると、その型に対応した関数が生成される。

3.4.1 パラメータ化された型

先ほどの double 用のベクタ汎用化して、任意の方用のベクタを作ることにしよう。そのためには、ベクタを templete 化するとともに、double 方をパラメータ化することになる。たとえば:

```
templete<typename T>
class Vector {
private:
        T* elem;         // elem は T 型の sz 個の配列へのポインタ
        int sz;
public:
        Vector(int s);                    // コンストラクタ:不変条件を確立して資源を獲得
        ~Vector() { delete[] elem; }      // デストラクタ:資源を解放

        // ...コピー演算とムーブ演算 ...

        T& operator[](int i);
        const T& operator[](int i) const;
        int size() const { return sz; }
};
```

ここで、templete<typename T>は、それに続く宣言で使われる T をパラメータ化する。これは数学の“すべての T に対して”を C++で表現したものであり、より正確にいえば“すべての型 T に対して”という意味だ。

メンバ関数の定義も同様である:

```
templete<typename T>
Vector<T>::Vector(int s)
{
        if (s < 0) throw Negative_size {};
        elem  new T[s];
        sz = s;
}

templete<typename T>
const T& Vector<T>::operator[](int i) const
{
        if (i < 0 || size() <= i)
                throw out_of_range {"Vector::operator[]"};
        reuturn elem[i];
}
```

以上の定義が与えられると、次のように Vector を定義できる：

```
Vector<char> vc(2000);              // 200 個の文字のベクタ
Vector<string> vs(17);              // 17 個の文字列のベクタ
Vector<List<int>> vli(45);          // 45 個の整数のリストのベクタ
```

Vector<List<int>>にある>>は入れ子となっているテンプレートの引数を閉じるものだ。入力演算子を誤って記述したわけではない。なお、2 個の>のあいだに空白文字を置く必要はない(ただし C++98 では空白文字が必要であった)。

さて、Vector は、以下のように利用できる：

```
void write(const Vector<string>& vs)       // string のベクタ
{
        for (int i = 0; i != vs.size(); ++i)
                cout << vs[i] << '¥n';
}
```

制御 for ループをサポートするには、適切な begin()関数と end()関数の定義が必要だ：

```
templete<typename T>
T* begin(Vector<T>& x)
{
        return x.size() ? &X[0] : nullptr;          // 先頭要素へのポインタもしくは nullptr
}

templete<typename T>
T* end(Vector<T>& x)
{
        return begin(x)+x.size();                   // 末尾要素の 1 個後方へのポインタ
}
```

これらが与えられると、以下のように記述できるようになる：

```
void f2(Vector<string>& vs)        // string のベクタ
{
        for (auto& s : vs)
                cout << s << '¥n';
}
```

同様に、リスト、ベクタ、マップ(すなわち連想配列)なども、テンプレートとして定義もできる(§4.4、§23.2、第 31 章)。

テンプレートは、コンパイル時のメカニズムなので、手作りのコードに比べて、実行時オーバヘッドが増えることはない(§23.2.2)。

3.4.2 関数テンプレート

テンプレートには、要素型のパラメータ化だけでなく、数多くの用途がある。たとえば、標準ライブラリでも、型とアルゴリズムの両方のパラメータ化で広く利用されている(§4.4.5、§4.5.5)。たとえば、任意のコンテナの要素の合計を算出する関数は、次のように記述できる：

```
template<typename Container, typename Value>
Value sum(const Container& c, Value v)
{
```

```
        for (auto x : c)
                v += x;
        return v;
}
```

テンプレート引数 Value と関数の引数 v とが宣言されているので、呼出し側は、型と、関数に渡す初期値(合計を加算する変数)とを指定できるようになる：

```
void user(Vector<int>& vi, std::list<double>& ld, std::vector<complex<double>>& vc)
{
        int x = sum(vi, 0);                     // int のベクタの合計(int を加算)
        double d = sum(vi, 0.0);                // int のベクタの計算(double を加算)
        double dd = sum(ld, 0.0);               // double のリストの合計
        auto z = sum(vc, complex<double>{});    // complex<double>のベクタの計算
}                                               // 初期値は{0.0, 0.0}
```

Double 型の値の合計を int として求める際は、int の最大値よりも大きい値に対する注意が必要である。Sum<T,V>の引数型が、関数の引数から省略されていることに注目しよう。幸いにも、引数型は、明示的な指定の必要がない。

ここで示した sum()は、標準ライブラリの accumulate()(§40.6)を単純化したものである。

3.4.3 関数オブジェクト

テンプレートの用途で特に有用なものの一つが、**関数オブジェクト**(function object)(**ファンクタ**(functor)とも呼ばれる)である()の実装である。これは、あたかも関数のように呼び出せるオブジェクトを定義するときに使われる。たとえば：

```
tmplate<typename T>
class Less_than {
        const T  val;    // 比較する値
public:
        Less_than(const T& v) :val(v) { }
        bool operator()(const T& x) const { return x<val; }        // 呼出し演算子
};
```

Operator()という名前の関数は、“関数呼出し”演算子(“呼出し”演算子や“アプリケーション”演算子とも呼ばされる)である()の実装である。

何らかの型を引数として Less_than 型を与えると、名前付き変数が定義できる：

```
Less_than<int> lti {42};                // lti(i)は i を<によって 42 と比較(i<42)
Less_than<string> lts {"Backus"};       // lts(s)は s を<によって"Backus"と比較(s<"Backus")
```

このようなオブジェクトは、通常の関数とまったく同じように呼び出せる：

```
void fct(int n, const string & s)
{
        bool b1 = lti(n);       // n<42 であれば真
        bool b2 = lts(s);       // s<"Backus"であれば真
        // ...
}
```

関数オブジェクトは、アルゴリズムに対して与える引数として広く利用されている。以下に示すのは、関数オブジェクトである述語が true を返した回数をカウントする例だ：

```
template<typename C, typename P>
```

```
int count(const C& c, P pred)
{


        int cnt = 0;
        for (const auto& x : c)
                if (pred(x))
                        ++cnt;
        return cnt;
}
```

ここで、**述語**(predicate)とは、true あるいは false を返す関数のことだ。たとえば：

```
void f(const Vector<int>& vec, const list<string>& lst, int x, const string& s)
{
        cout << "number of values less than " << x
                 << ": " << count(vec, Less_than<int>{x})
                 << '¥n';
        cout << "number of values less than " << s
                 << ": " << count(lst, Less_than<string>{s})
                 << '¥n';
}
```

ここで、Less_than<int>{x}は、呼出し演算子が x という名前の int の比較を行うオブジェクトを構築する。また、Less_than<string>{s}は、s という名前の string との比較を行うオブジェクトを構築する。これらの関数が美しいのは、比較する値を保持して運んでくれるからだ。それぞれの値(と、それぞれの型)に応じた関数を別々に定義する必要はないし、比較する値を保持するために、汚らわしい広域変数を使う必要もない。また、Less_than のような単純な関数オブジェクトは、インライン化が十分に可能なので、Less_than は、間接的な関数呼出しよりも効果的となる。保持するデータを運べることに加えて、その効率性から、関数オブジェクトはアルゴリズムに対して与えられる引数として特に有用なのだ。

汎用アルゴリズムの中核となる演算を指定するための、(Less_than に対する count()のような)関数オブジェクトは、**ポリシーオブジェクト**(policy object)と呼ばれる。

Less_than の定義は、利用箇所とは別の所で行う必要がある。このことは不便に感じられるだろう。そのため、関数オブジェクトを暗黙裏に生成する記法が用意されている：

```
void f(const Vector<int>& vec, const list<string>& lst, int x, const string& s)
{
        cout << "number of values less than " << x
                 << ": " << count (vec, [&](int a){ return a<x; })
                 << '¥n';
        cout << "number ofvalues less than " << s
                 << ": " << count(lst, [&](const string& a){ return a<s; })
                 << '¥n';
}
```

[&](int a){ return a<x; }という表記は、ラムダ式(lambda expression)(§11.4)と呼ばれ、Less_than<int>{x}と全く同じ関数オブジェクトを生成する。ここでの[&]は、使っている局所名(x など)を、参照を介してアクセスすることを表す**キャプチャ並び**(capture list)だ。もし x だけを“キャプチャ”したければ、**[&x]**と記述する。また、x のコピーとしてオブジェクトを生成したい場合は、**[=x]**と記述する。何もキャプチャしない場合は**[ ]**である。すべての局所名を参照としてキャプチャする場合は**[&]**として、値としてキャプチャする場合は**[=]**とする。

ラムダ式は簡潔で便利だが不明瞭な面もある。処理内容が単純でない場合(たとえば単一

の式以上の記述が必要となる場合)は、私であれば、その目的を表す明白な名前を処理に与えた上で、プログラムの他の場所から利用できるようにする。

§3.2.4 の draw_all()や rotate_all()の例では、Vector の要素がポインタや unique_ptr であれば、数多くの関数の記述が必要となる煩わしさを経験した。関数オブジェクト(特にラムダ式)を使うと、コンテナの走査を、各要素に対して行うべき処理の指示から分離できるようになる。

その場合、まず、ポインタのコンテナ内の要素が指すオブジェクトに対して適用する処理を記述した関数が必要になる：

```
template<typename C, typename oper>
void for_all(C& c, Oper op)                // C がポインタのコンテナと仮定
{
        for (auto& x : c)
                op(*x);                    // 各要素が指す参照を op()に渡す
}
```

これを使うと、§3.2.4 の user()は、_all の関数群を定義することなく作れるようになる：

```
void user()
{
        vector<unique_ptr<Shape>> v;
        while (cin)
                v.push_back(read_shape(cin));]
                for_all(v, [](Shape& s) { s.draw(); });            // draw_all()
                for_all(v, [](Shape& s) { s.rotate(45); });        // rotate_all(45)
}
```

ここでは、Shape への参照をラムダ式に渡しているので、ラムダ式は、コンテナがオブジェクトをどのように保持しているかを正確に把握する必要がなくなる。ここでの for_all()は、仮に v を vector<Shape*>に変更しても、そのまま動作する点で優れている。

3.4.4 可変個引数テンプレート

テンプレートは、任意の型と任意の個数の引数を受け取れるようにも定義できる。このようなテンプレートは、**可変個引数テンプレート**(variadic template)と呼ばれる。たとえば：

```
void f() { }               // 何も行わない

template<typename T, typename ... Tail>
void f(T head, Tail .... tail)
{
        g(head);           // head に対して何らかの処理を行う
        f(tail...);                // tail に対しても行う
}
```

可変個引数テンプレートの実装で重要なのは、引数の並びを渡した場合に、先頭引数が、その後続の引数と分離できるようになっていることだ。先頭引数(head)に対して何らかの処理を行って、

それから、それ以降の引数(tail)に対して f()を再帰的に呼び出す。省略記号...は、並びの“残り”を表す。もちろん、最終的には tail は空になるので、それを処理する関数が別途必要となる。

以下に示すのが、f()の利用例だ：

```
int main()
{
        cout << "first: ";
        f(1, 2.2, "hello");

        cout << "¥nsecond: ";
        f(0.2, 'c', "yuck!", 0, 1, 2);
        cout << "¥n";
}
```

ここで、f(1, 2.2, “hello”)は、f(2.2, “hello”)を呼び出す。呼び出された f(2.2, ”hello”)は、f(“hello”)を呼び出す。呼び出された f(“hello”)は、最終的に f()を呼び出す。それでは、g(head)は何を実行するのだろうか？ いうまでもなく、与えられた要素に対して、姥久手もの処理を実行する。たとえば、引数(ここでは head)を出力するように定義できる：

```
template<typename T>
void g(T x)
{
        cout << x << " ";
}
```

この定義が加わると、先ほどのコードの実行結果は、次のようになる：

```
first: 1 2.2 hello
second: 0.2 c yuck! 0 1 2
```

f()は、あらゆる並びや値を出力する printf()を単純化した変種にも見える。その実装は、たった 3 行のコードと、それを囲む宣言のみである。

可変個数テンプレート(単に**可変個数引数**(variadic)と呼ばれることもある)が強力なのは、あらゆる引数を受け取れることだ。弱点は、インターフェースの型チェックが、テンプレートプログラミングを精巧なものとする可能性があることだ。詳細は§28.6 を、具体例は§34.2.4.2(N個のタプル)と第 29 章(N次元行列)を参照しよう。

3.4.5 別名

よく不思議がられるのだが、型やテンプレートに同義語を与えるのは、有用なことだ(§6.5)。たとえば、標準ヘッダ<catddef>は、size_t という別名を定義している。以下に示すのが、その定義の一例である：

```
using size_t = unsigned int;
```

size_t 型の実際の型は処理系依存だ。そのため、別の処理系では、size_t は、unsigned long かもしれない。Size_t という別名のおかげで、プログラマは可能性の高いコードが記述できる。

パラメータ化された型が、テンプレート引数に関連する型に対する別名を提供するのは、極めて

一般的なことだ。たとえば：

```
templete<typename T>
class Vector {
        public using value_type = T;
        // ...
}
```

実際、標準ライブラリのすべてのコンテナは、自身が保持する値の型の名前として value_type を提供する(§31.3.1)。その結果、この規約にしたがう、あらゆるコンテナに対して動作するコードが記述できるようになる。たとえば：

```
template<typename C>
using Value_type = typename C::value_type;          // C の要素型

template<typename Container>
void alog(Container& c)
{
        Vector<Value_type>Container>> vec;
        // ... vec を利用 ...
}
```

別名は、テンプレートの引数の一部またはすべてをバインドして、新しいテンプレートを定義する際にも利用できる。たとえば：

```
template<typename Key, typename Value>
class Map {
        // ...
}


template<typename Value>
usinge String_map = Map<string, Value>;


String_map<int> m;        // m は Map<string, int>
```

3.5 アドバイス

[1] アイディアは、そのままコード化しよう。 §3.2。
[2] アプリケーションのコンセプトを、そのままコード化したクラスを定義しよう。 §3.2。
[3] 単純なコンセプトの表現と厳しい性能とが求められる部品には、具象クラスを利用しよう。§3.2.1。
[4] “裸の”new 演算子と delete 演算子は使わないように。 §3.2.1.2.
[5] 資源管理には、資源ハンドラと RAII を利用しよう。§3.2.1.2.
[6] インタフェースと実装の完全分離が必要であれば、インタフェースとして抽象クラスを利用しよう。 §3.2.2.
[7] 階層構造をもつ概念を表現するには、クラス階層を利用しよう。 §3.2.4
[8] クラス階層を設計する際は、実装継承とインタフェース継承を使い分けよう。 §3.2.4

[9] オブジェクトの構築、コピー、ムーブ、解体を制御しよう。 §3.3.

[10] コンテナは値で返却しよう(ムーブを活用できるので効率的だ)。 §3.3.2。

[11] 強い資源安全を提供しよう。すなわち、資源とみなせるものをリークさせてはいけない。§3.3.3

[12] 同じ型の値を複数持つ場合は、資源ハンドラのテンプレートとして定義されたコンテナを利用しよう。§3.4.1

[13] 汎用的なアルゴリズムは関数テンプレートとして実現しよう。§3.4.2

[14] ポリシーと実行は、ラムダ式を使うと関数オブジェクトとして実現しよう。§3.4.3

[15] 違いが少ない型や実装には、型とテンプレートに別名を与えることで、統一化された記法を実現しよう。§3.4.5

第 4 章 C++を探検しよう ： コンテナとアルゴリズム

なぜ知識のために時間を浪費するんだ？
無知なら一瞬で手に入るのに。
ーホップス

・ライブラリ
　標準ライブラリ概要／標準ライブラリヘッダと名前空間
・文字列
・ストリーム出力
　出力／入力／ユーザ定義型の入出力
・コンテナ
　Vector／list／map／unordered_map／コンテナのまとめ
・アルゴリズム
　反復子の利用／反復子の型／ストリーム反復子／述語／アルゴリズムのまとめ／コンテナアルゴリズム
・アドバイス

4.1 ライブラリ

それなりの意味を持つプログラムが、素のプログラミング言語だけで記述されることはない。まず、一連のライブラリが開発される。それが、その後の開発基盤となる。ほとんどのプログラムは、良質なライブラリを利用すると、単純に記述できるので、素のプログラミング言語だけで記述するのは、退屈な作業となってしまう。

第 2 章と第 3 章に続いて、本章と続く第 5 章では、短時間の探検で、標準ライブラリ機能を紹介する。なお、ここではプログラミング経験のある読者を想定している。もし経験がなければ、本章を読み進める前に、まずは Programming:Principles and Practice Using C++ 〔Stroustrup, 2008〕などのテキストを先に読むことが好ましい。プログラミング経験があっても、その言語を使用したアプリケーションが C++のプログラミングスタイルとかけ離れていれば、やはり同様である。もし、この“短い時間の探検”で混乱するようであれば、もっと系統立てて解説を行っている第 6 章から読み始めるのもよいだろう。標準ライブラリの系統立てた解説は第 30 章から開始する。

まず、(本章では)string、ostream、vector、map、(第 5 章では)unique_ptr、thread、regex、complex などの標準ライブラリの有用な型を、そのもっとも一般的な使い方とともに解説する。以降の章で、よりよ具体例を提示するので、解説は簡単なものだ。第 2 章と第 3 章でも述べたように、本質に集中することが重要であって、詳細の理解が不足していることに惑わされる必要はない。本章の目的は、以降の章で述べる内容のさわりを示すことであるとともに、もっとも有用なライブラリ機能をおおまかに理解することである。

ISO C++標準では、その3分の2の分量を標準ライブラリの仕様に割いている。その内容は探検すべきものであり、自家製コードよりも優先すべきものだ。標準ライブラリの設計は熟慮されたものだし、実装は安定している。今後も、捕手と拡張に多くの労力が注がれるだろう。

本書が取り上げる標準ライブラリ機能は、C++処理系全体から見ると、ごく一部にすぎない。ほとんどのC++処理系は、標準ライブラリ以外にも、“グラフィカルユーザインタフェース(GUI)”、ウェブインタフェース、データベースインタフェースなどを提供する。同様に、アプリケーション開発環境でも、業務用の“基盤ライブラリ”や“標準”開発・実行環境を提供する。このようなシステムやライブラリは、本書では取り上げない。特に断らない限り、標準仕様として定義されたC++を対象としているし、サンプルの移植性を維持している。読者の皆さんは、大部分のシステムで有効な拡張機能を、ご自身で調べるとよいだろう。

4.1.1 標準ライブラリ概要

標準ライブラリが提供する機能は、以下のように分類できる：

- 実行時の言語支援(メモリ確保や実行時型情報など)。§30.3を参照しよう。
- 標準Cライブラリ(を、型システムの違反を最小限に抑えるために少しだけ改変したもの)。第43章を参照しよう。
- 文字列と入出力ストリーム(国際化文字セットやロケールがサポートされている)。第36章と第38章と第39章を参照しよう。入出力ストリームは、バッファリング方式、文字セットなどをユーザ独自に拡張できるフレームワークである。
- コンテナ(vector、mapなど)とアルゴリズム(find()、sort()、marge()など)のフレームワーク。§4.4と§4.5と第31章～第33章を参照しよう。このフレームワークは、一般にSTL〔Stpanov, 1994〕と呼ばれ、独自のコンテナとアルゴリズムをユーザが追加できる拡張性をもつ。
- 数値演算(標準数学関数、複素数、ベクタに対する算術演算、乱数生成器など)の支援。§3.2.1.1と第40章を参照しよう。
- 正規表現の支援。§5.5と第37章を参照しよう。
- threadやlockなどによる並行プログラミングの支援。§5.3と第41章を参照しよう。並行プログラミングのサポートは基礎的なものであり、ユーザは新しい並行モデルをライブラリとして追加できる。
- 各種おプログラミングを支援するユーティリティ。たとえば、テンプレートメタプログラミングに対しては、型特性(§5.4.2、§28.2.4、§35.4)など、STLスタイルのジェネリックプログラミングに対しては、pair(§5.4.3、§34.2.4.1)など、一般的なプログラミングに対しては、Clock(§5.4.1、§35.2)など。
- 資源管理のための“スマートポインタ”(unique_ptrやshared_ptrなど：§5.2.1、§34.3)とガベージコレクタインタフェース(§34.5)。
- array(§34.2.1)、bitset(§34.2.2)、tuple(§34.2.4.2)などの特殊用途のコンテナ。

クラスをライブラリ化する際の主な基準は、次のとおりだ：

・そのクラスが、ほぼすべての(初心者と熟練者の両方の)C++プログラマにとって、有用であること。
・同じ機能ための簡潔なバージョンと比べて、特別なオーバーヘッドを必要としない、一般的な形態で提供できること。
・クラスの単純な利用方法が(その処理内容と比較して)容易に学習できること。

本質的に、C++標準ライブラリは、もっとも広く使われる基礎的なデータ構造を、それに適用する基礎的なアルゴリズムとともに提供する。

4.1.2 標準ライブラリヘッダと名前空間

すべての標準ライブラリ機能は、標準ヘッダを通じて提供される。たとえば：

```
#include<string>
#include<list>
```

これで、標準な string と list が利用できるようになる。
標準ライブラリ機能は、std という名前の単一の名前空間の中で定義されている(§2.4.2、§14.3.1)。そのため、標準ライブラリ機能を使うときは、名前の前に std::を付加する：

```
std::string s {"Four legs Good; tow legs Baaad!"};
std::list<std::string> slogans {"War is Peace", "Freedom is Slavery",
                                                    "Ignorance is Strength"};
```

簡単化するために、本書のサンプルでは、std::を明示的に記述することはほとんどない。また、必要なヘッダを常に明示的に#include するわけではない。本書のコードを、コンパイルして実行するためには、読者自身が適切な(§4.4.5、§4.5.5、§30.2 に示している)ヘッダを#include して、宣言されている名前が利用できるようにしなければならない。たとえば：

```
#inlucde<string>                // 標準の string 機能をアクセスできるようにする
using namespace std;            // std の名前を std::付けずに利用できるようにする

string s {"C++ is a general-purpose programming languge"}; // OK : string は std::string
```

ある名前空間内のすべての名前を、広域名前空間に持ち込むのは、一般的にお粗末だ。しかし、本書では、ほぼ標準ライブラリだけを使っており、標準ライブラリがどの機能を提供しているかが分かりやすい。そのため、本書では、標準ライブラリ内の名前には std::を付加しない。同様に、すべてのサンプルで、ヘッダの#include は行わない。この点については、今後は繰り返さない。
さて、std 名前空間内の宣言をもつ、標準ライブラリのヘッダを抜粋したものを示す：

標準ライブラリヘッダ(抜粋)			
<algorithm>	copy(),find(),sort()	§32.2	§iso.25
<array>	array	§34.2.1	§iso.23.3.2
<chrono>	duration, time_point	§35.2	§iso.20.11.2

標準ライブラリヘッダ(抜粋)			
<cmath>	sqrt(), pow()	§40.3	§iso.26.8
<complex>	complex, sqrt(), pow()	§40.4	§iso.26.8
<fstream>	fstream, ifstream, ofstream	§38.2.1	§iso.27.9.1
<future>	future, promise	§5.3.5	§iso.30.6
<iostream>	istream, ostream, cin, cout	§38.1	§iso.27.4
<map>	map, multimap	§31.4.3	§iso.23.4.4
<memory>	unique_ptr, shared_ptr, allocator	§5.2.1	§iso.20.6
<random>	default_ramdaom_engine, normal_distribution	§	§
<regex>	regex, smatch	第37章	§iso.28.8
<string>	string, basic_string	第36章	§iso.21.3
<set>	set, multiset	§31.4.3	§iso.23.4.6
<sstream>	istrstream, ostrstream	§38.2.2	§iso.23.4.6
<thread>	thred	§5.3.1	§iso.30.3
<unordered_map>	unordered_map, unordered_multimap	§31.4.3.2	§iso.23.5.4
<utility>	move(), swap(), pair	§35.5	§iso.20.1
<vector>	vector	§31.4	§iso.23.3.6

この表は、すべてのヘッダを網羅しているわけではない。詳細は§30.2を参照しよう。

4.2 文字列

標準ライブラリでは、文字列リテラルを補完するためのstring型を提供している。そのstring型は、連結などの有用な文字列処理を豊富に表現している。たとえば：

```
string compose(const string& name, const string& domain)
{
        return name + '@' + domain;
}
auto addr = compose("dmr", "bell-labs.com");
```

ここで、addrは、dmr@bell-labs.comという文字の並びで初期化される。文字列の“加算”は、連結処理のことだ。stringに対して連結できるのは、string、文字列リテラル、単一文字列、C言語スタイルの文字列である。なお、標準のstringは、ムーブコンストラクタを実装しているので、長いstringを値で返す処理は、効率よく行われる(§3.3.2)。

stringの末尾に対して追加を行う連結処理は、多くのアプリケーションで頻繁に利用される。この処理を直接サポートするのが、stringの+=演算である。たとえば：

```
void m2(string& s1, string& s2)
{
        s1 = s1 + '¥n';          // 改行を追加
        s2 += '¥n';                      // 改行を追加
}
```

ここに示している、2種類のstring末尾への文字の追加は、意味的には等価である。しかし、私は後者を好む。というのも、後者の方が、処理内容がより明確で、簡潔であるとともに、効率的になる可能性があるからだ。

stringは、中身を書き換えれるようになっている。=、=+演算だけでなく、([]による)添字演算や部分文字列の処理も提供される。標準ライブラリのstringについては、第36章で解説する。多くの機能の中から、ここでは部分文字列操作の例を示すことにしよう。たとえば：

```
string name = "Nils Stroustrup";

void m3()
{
        string s = name.substr(6, 10);          // s = "Strousstrup"
        name.replace(0, 5, "nicholas");         // nameは"nicholas Stroustrup"になる
        name[0] = toupper(name[0]);                     // nameは"Nicholas Stroustrup"になる
}
```

substr()処理は、引数で指定された部分文字列のコピーのstringを返す。先頭引数にはstring内でのインデックス(位置)を指定して、2番目の引数には目的の部分文字列の長さを指定する。インデックスは0から始まるので、sの値はStroustrupとなる。

replace()処理が行うのは、部分文字列を、指定された値に置換することだ。ここでは、0から始まる長さが5の部分文字列はNielsなので、それをnicholasに置換する。最後に、先頭文字をそれに対応する大文字に置換する。最終的に、nameの値はNicholas Stroustrupとなる。置換する文字列は、置換される部分文字列の長さと一致していなくてともよいことに注意しよう。

なお、stringどうしの比較や、文字列リテラルとの比較も行える：

```
string incantation;

void respond(const string& answer)
{
        if (answer == incantation ) {
                // 魔法をかける
        } else if (answer == "yes" ) {
                // ...
        }
        // ...
}
```

stringライブラリについては第36章で解説するが、stringの実装でもっともよく使われているテクニックは、stringの例題で示す(§19.3)

4.3 ストリーム入出力

標準ライブラリは、iostreamライブラリで、書式付き入出力を提供する。入力処理は型付けされ、ユーザ定義型に対しても拡張できるようになっている。本節では、iostreamの利用方法を極めて簡略化して解説する。第38章では、iostreamライブラリ機能をおおむね完全に網羅して解説する。

グラフィカル入出力などのような、他の形態のユーザ入出力は、ISO標準の範囲外のライブラリが取り扱うものなので、ここでは取り上げない。

4.3.1 出力

入出力ストリームライブラリは、すべての組込み型の出力処理を定義している。しかも、ユーザ定義型(§4.3.3)の出力処理の定義も簡単だ。ostream 型のオブジェクトに対する出力処理には、<<演算子(“〜へ出力”(put to))を使う。なお、cout は標準出力ストリームを表し、cerr が標準エラー出力ストリームを表す。デフォルトでは、cout に出力する値は、文字列の並びへと変換される。たとえば、10 進数の 10 の出力は、以下のように記述できる：

```
void f()
{
        cout << 10;
}
```

このコードは、文字の 1 と、文字の 0 を、標準出力ストリームに連続して出力する。
同じ出力は、以下のように行うこともできる：

```
void g()
{
        int i {10};
        cout << i;
}
```

型が異なるものも、そのまま自由に組み合わせることができる。たとえば、

```
void h (int i)
{
        cout << "the value of i is ";
        cout << i;
        cout << '¥n';
}
```

h(10)と呼び出すと、次のように出力される：

```
the value of i is 10
```

しかし、関連性の高い一連のデータ出力の際に、出力ストリーム名を毎回記述するのは、退屈だ。幸いにも出力する式の結果は、そのまま次の出力に再利用できる。たとえば：

```
void h2(int i)
{
        cout << "the value of i is " << i << '¥n';
}
```

h2()を実行すると、h()と同じ出力を行う。
単一の文字を単一引用符記号で囲んだものは、文字定数である。文字は、文字コードの数値ではなくて、文字そのものとして出力されることに注意しよう。たとえば：

```
void k()
{
        int b = 'b';              // 注意:char は暗黙裏に int へ変換される
        char c = 'c';
        cout << 'a' << b << c;
}
```

(私が使っている C++処理系は ASCII コードを利用しているので)文字’b’の整数値は 98 であり、実行すると、a98c と表示される。

4.3.2 入力

標準ライブラリは、入力のために istream を提供する。ostream と同様に、istream は組込み型を表す文字の並びが処理できるし、ユーザ定義型を処理するための拡張が簡単に行える。

入力処理には>>演算子(“～から入力”(get from))を使う。なお、cin は標準入力ストリームだ。受け付ける入力と、その対象は、>>の右オペランドの型によって決定される。たとえば：

```
void f()
{
        int i;
        cin >> i;               // 整数を i に読み込む

        double d;
        cin >> d;               // 倍精度浮動小数点値を d を読み込む
}
```

このコードは、1234 のような数値を、標準入力から整数の変数 i に読み取って、それから、12.34e5 のような浮動小数点数を倍精度浮動小数点数の変数 d に読み取る。

文字の並びの読取りが必要となることは多い。string 型の変数へ読み取ると、都合がよい：

```
void hello()
{
        cout << "Please enter your name¥n";
        string str;
        cin >> str;
        cout << "Hello, " << str << "!¥n";
}
```

実行して Eric と入力すると、次のような出力が得られる：

```
Hello, Eric!
```

デフォルトでは、スペースなどの空白類文字(§7.3.2)は読取りを終了させる。そのため、読者が悲運のヨーク王のふりをして Eric Bloodaxe と入力しても、先ほどと同じ出力となる：

```
Hello, Eric!
```

末尾の改行文字までの行全体を読み取る場合は、getline()関数を使う。たとえば：

```
void hello_line()
{
        cout << "Please enter your name¥n";
        string str;
        getline(cin, str);
        cout << "Hello, " << str << "!¥n";
}
```

このプログラムに対して Eric Bloodaxe と入力すると、期待どおりの出力が得られる：

```
Hello, Eric Bloodaxe!
```

行末尾の改行文字は読み捨てられるので、cin は次の行を読み取る準備ができるようになる。

標準の string は、与えられた文字列を保持する大きさとなるよう自動的に拡張するので、最大長を事前に求める必要はない。そのため、このコードを実行して、数メガバイトものセミコロンを入力すると、何ページにもわたるセミコロンが出力されることになる。

4.3.3 ユーザ定義型の入出力

iostream ライブラリは、組込み型と標準 string の入出力に加えて、独自の型に対する入出力をプログラマが定義できるようにしている。たとえば、電話帳内の 1 件のデータを表す単純な Entry 型を考えてみよう：

```
struct Entry {
        string name;
        int number;
};
```

Entry を初期化する際のコードによく似た形式{“name”,number}で出力を行うための、単純な出力演算子は、次のように定義できる：

```
ostream& operator<<(ostream& os, const Entry& e)
{
        return os << "{"" << e.name << "¥", " << e.number << "}";
}
```

ユーザ定義の出力演算子は、先頭引数に出力ストリームを(参照渡しで)受け取って、その出力ストリームを返却する。詳細は§38.4.2 で解説する。

これと対になる入力演算は、書式の確認やエラー処理が必要なので、少し複雑になる：

```
istream operator>>(istrream& is, Entry& e)
        // {"name", number}を読み取る。 注意：括弧{}と二重引用符""とコンマ,がつく形式
{
        char c, c2;
        if (is >> c && c =='{' && is>>c2 && c2=='"') { // {"で始まる
                string name;                        // string のデフォルト値は空文字列すなわち""
                while (is.get(c) && c != '"')       // "より前は name の一部
                        name+=c;

                if ( is >> c && c == ',' ) {
                        int number = 0;
                        if ( is >> number >> c && c == '}' ) {    // number と}読み込む
                                e = {name, number};               // エントリに代入
                                return is;
                        }
                }
        }
        is.setstate(ios_base::failbit);           // 失敗したことをストリームに記録
        return is;
}
```

入力演算子は istream への参照を返すので、その返却値を調べると、処理が成功したかどうかが判断できる。たとえば、条件式に is>>c を使用すると、“is から c への読取りは成功したか？”という意味になる。

is>>c は、デフォルトでは空白文字を読み飛ばすが、is.get(c)は読み飛ばさない。そのため、Entry の入力処理では、名前文字列外の空白文字を無視する(読み飛ばす)が、名前文字列内では読み取る。たとえば：

```
{ "Johe Marwood Cleese", 123456          }
{"Michael Edward Palin", 987654          }
```

このようなデータを Entry への入力として読み取るには、次のように行う：

```
for ( Entry ee; cin >> ee; ) // cin から ee に読み込む
        cout << ee << '¥n';  // cout に ee を書き出す
```

実行すると、以下のように出力される：

```
{"John Marwodd Cleese", 12345}
{"Michael Edwart Palin", 987654}
```

ユーザ定義型の入力演算子の技術的詳細や技法については、§38.4.1 を参照しよう。また、文字の並びをパターン化する系統立てた技法(正規表現)については、§5.5 と第 37 章を参照しよう。

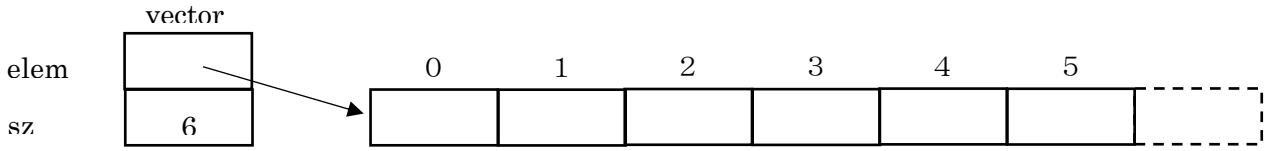
4.4 コンテナ

多くのプログラムで、値の集合を作って、その操作を行う。たとえば、string に複数の文字を読み取る、あるいは、出力するといったことは、その単純な例の一つだ。オブジェクトを内部に保持することを主目的としたクラスは、**コンテナ**(container)と呼ばれる。適切なコンテナを準備して、それを基礎的で有用な演算子によって支援することは、どんなプログラム開発でも重要なステップだ。

標準ライブラリのコンテナを示す例として、名前と電話番号をもつ単純なプログラムを考えていくことにしよう。というのも、この類のプログラムは、これまでの経験と異なるプログラマにとって、異なる各種の手法が“単純明快”となるからだ。単純な電話帳内の 1 件のデータ表現には、§4.3.3 の Entry クラスを使うものとする。なお、電話番号の多くは単純な 32 ビット int では表現できないなど、現実の世界はもっと複雑なのだが、ここでは、あえて無視する。

4.4 vector

標準ライブラリのコンテナの中で、最も有用なのは vector だ。vector は、指定された型の要素が並んだシーケンスである。それらの要素は、メモリ内で連続的に配置される：



§3.2.2 と §3.4 の vector のサンプルからも、vector の実装が推測できるだろう。また、§13.6 と §31.4 でも徹底的に解説する。

さて、vector は、要素型をもった一連の値を与えることで初期化できる：

```
vector<Entry> phone_book = {
        {"David Hume", 1234567},
        {"Karl Popper", 234567},
        {"Bertrand Arthur William Russell", 345678}
};
```

各要素は、添え字演算によってアクセスできる：

```
void print_book(const vector<Entry>& book)
{
        for (int i = 0; i != book.size; ++i)
                cout << book[i] << '¥n';
}
```

例によって、添え字は0から始まるので、ここでのbook[0]は、David Humeの Entryである。メンバ関数size()は、vector内の要素数を返却する。

vector内の要素が範囲を形成するので、範囲forループ(§2.2.5)が利用できる：

```
void print_book(const vector<Entry>& book)
{
        for (const auto& x : book)        // autoについては§2.2.2を参照
                cout << x << '¥n';
}
```

vectorの変数の定数にあたっては、当初の要素数(要素数の初期値)も指定できる：

```
vector<int> v1 = {1, 2, 3, 4};            // 要素数は4
vector<string> v2;                        // 要素数は0
vector<Shape*> v(23);                     // 要素数は23：要素の初期値はnullptr
vector<double> v4(32, 9.9);               // 要素数は32：要素の初期値は9.9
```

要素数を明示的に指定するときは、(23)のように、通常の丸括弧を用いる。その場合、デフォルトでは、各要素が、要素型のデフォルト値(たとえば、ポインタであればnullptr、数値であれば0)で初期化される。デフォルト値による初期化を望まないのであれば、要素の値を第2引数に指定する(ここでの9.9は、v4の32個の要素すべてに適用される)。

要素数は、初期の状態から変更可能である。vectorがもつ演算子で最も有用なものはpuch_back()であり、これは要素の末尾に新しい要素を追加して、要素数を1だけ増加させるものだ。たとえば：

```
void input()
{
        for (Entry e; cin >> e;)
                phone_book.push_back(e);
}
```

このコードは、ファイル終端に到達するか、読み取り書式エラーが発生するまで、標準入力からphone_bookへとEntryを読み取る処理を繰り返す。標準ライブラリのvectorは、push_back()の繰り返しによる要素数増加も効率的に行われるようになっている。

vectorは、代入時と初期化時にコピーが可能だ。たとえば：

```
vector<Entry> book2 = phone_book;
```

vectorのコピーとムーブは、§3.3で解説したように、コンストラクタと代入演算によって実装されている。代入時は、各要素をコピーする。すなわち、ここでの初期化が完了したときに、book2とphone_bookは、それぞれが個別に電話帳のすべてのEntryをもつ。そのため、vectorが数多くの要素を保持しているときに、代入や初期化を何気なく行うと、高コストになってしまう。コピーが望ましくないときは、参照やポインタ(§7.2、§7.7)、あるいは、ムーブ演算(§3.3.2、§17.5.2)を使うべきだ。

4.4.1.1 要素

標準ライブラリのすべてのコンテナと共通することだが、vectorは、何らかの型Tの要素のコンテナであり、それがvector<T>となる。要素の型は任意だ。たとえば、組込みの数値型(char、

int、double など)、ユーザ定義型(string、Entry、list<int>、Matrix<double,2>など)、ポインタ(const char*、Shape*、double*)などである。新しい要素を追加する際は、その値がコンテナ内にコピーされる。たとえば、整数値 7 をコンテナに入れると、内部に保持される要素は、その数値 7 そのものだ。すなわち、要素は、7 という値を内部に保持するオブジェクトへのポインタや参照ではない。そのため、高速にアクセスできる。コンパクトで良質なコンテナが実現できる。この点は、メモリサイズや実行速度を重視する場合、極めて重要だ。

4.4.1.2 範囲チェック

標準ライブラリの vector は、範囲チェックを保証しない。(§31.2.2)。たとえば:

```
void silly(vector<Entry>& book)
{
        int i = book[book.size()].number; // book.size()は範囲外
        // ...
}
```

この初期化は、エラーとなるのではなくて、i の値が不定値となってしまう。これは、期待とは異なる動作であるし、範囲外エラーは、良く起こる問題の一つとなっている。そのため、私は、vector に対して、簡単な範囲チェック用のアダプタを使うことがある:

```
template<typename T>
class Vec : public std::vector<T> {
public:
        using vector<T>::vector;        // vector のコンストラクタ群を利用(名前 Vec のもとで):
                                                                // §20.3.5.1 を参照
        T& operator[](int i)            // 範囲チェック
                { return vector<T>::at(i); }
        const T& operator[](int i) const // const オブジェクトの範囲チェック:§3.2.1.1
                {return vector<T>::at(i); }
};
```

Vec クラスは、vector から添字演算以外のすべてを継承するとともに、範囲チェックを行う処理を再定義している。at()は vector の添字演算であり、引数が要素の範囲を越えている場合に out_of_range 型の例外を送出する(§2.4.3.1、31.2.2)。

Vec を使うと、範囲外へのアクセスに対して例外が送出されるので、ユーザはそれを捕捉できるようになる。たとえば:

```
void checked(Vec<Entry>& book)
{
        try {
                book[book.size()] = {"Joe", "999999"};  // 例外を送出すことになる
                // ...
        }
        catch (out_of_range) {
                cout << "range error¥n";
        }
}
```

このコードでは、例外が送出されて捕捉される(§2.4.3.1、第 13 章)。ユーザが捕捉しなければ、プログラムは、的確に定義された動作で終了する。すなわち、処理を続行したり、定義されな

いないような動作をするのではない。捕捉されなかった例がに驚かされる機会を最小限に抑えるには、main()を try ブロックとして定義するとよい。たとえば：

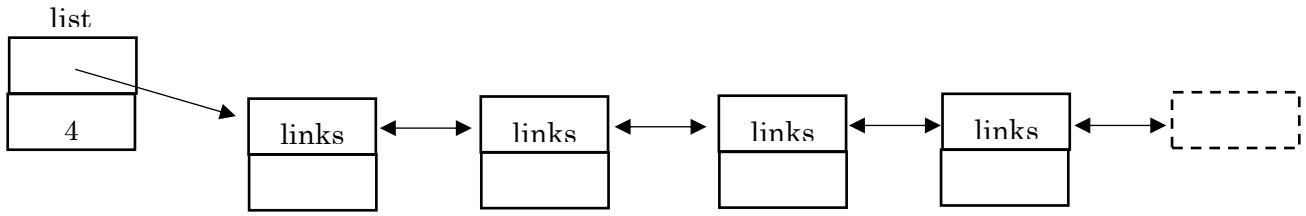
```
int main()
{
        try {
                // 何らかのコード
        }
        catch (out_of_range) {
                cerr << "range error¥n";
        }
        catch (...) {
                cerr << "unknown exception thrown¥n";
        }
}
```

ここでは、デフォルトの例外ハンドラを定義しているので、何らかの例外の捕捉に失敗していたとしても、エラーメッセージが標準エラー出力ストリーム cerr に出力される(§38.1)。

処理系によっては、(コンパイラオプションなどによって)範囲チェック機能をもった vector を提供するので、ここに示した Vec クラス(あるいは同等なクラス)を定義する手間を省ける場合がある。

4.4.2 list

標準ライブラリは、list という名前の双方高結合リストを提供する。



list クラスは、要素を移動することなく、要素の挿入や削除を行う必要があるシーケンスに対して、利用するものだ。電話帳に対しては、挿入や削除がしばしば行われる。そのため、単純な電話帳を表現するには、list が適切だろう。たとえば：

```
list<Entry> phone_back = {
        {"David Hume", 123456};
        {"Karl Popper", 234567};
        {"Bertrand Arthur William Russell", 345678}
};
```

ベクタとは異なり、結合リストでは、要素へのアクセスには添字演算は使わない。そうでなく、目的の値をもつ要素を探索するのだ。その動作を実現するには、§4.5 に示す list の内部構造をうまく利用する：

```
int get_number(const string& s)
{
        for (const auto& x : phone_book)
                if ( x.name == s )
                        return x.number;
        return 0;                       // 0 は“値が見つからなかったこと”を表す
}
```

s の探索は、リストの先頭から開始して、その s が見つかるか、あるいは、phone_book 内の末尾に到達するまで行われる。

list ない要素の特定(識別)が必要になることもある。たとえば、ある要素の削除や、ある要素の直前への新しい要素の挿入をおこないたいときだ。そのために利用するのが**反復子**(iterator)である。list の反復子は list 内の要素を識別できるし、list 全体をとおした(その名のとおり)反復した操作が行える。標準ライブラリの全コンテナは、先頭要素を返す begin()と、末尾要素の直後を返す end()を提供している。(§4.5、§33.1.1)。反復子を明示的に用いると、少々エレガントさに欠けるものの、get_number()という関数を以下のように定義できる：

```
int get_number(const string& s)
{
        for (auto p = phone_book.begin(); p != phone_book.end(); ++p)
                if (p->name == g)
                        return p->nmber;
        return 0;                           // 0 は“値が見つからないこと”を表す
}
```

大雑把にいうと、このコードは、コンパイラが生成する。そっけなくてエラーにつながりにくい、範囲 for ループと同等だ。反復子 p が与えられると、*p は反復子が指す要素となる、また、++p は次の要素を示すように p を進める。なお、p がメンバ m をもつクラスを指している場合、p->m は(*p).m と等価である。

list への要素の追加と削除は容易である：

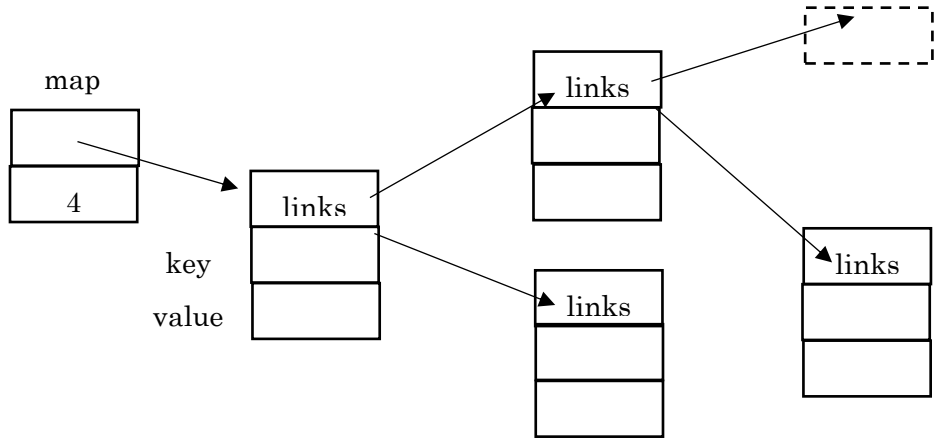
```
void f(const Entry& ee, list<Entry>::iterator p, list<Entry>::iterator q)
{
        phone_book.insert(p, ee);          // p が指す要素の直前に ee 挿入
        phone_book.erase(q);                        // q が指す要素を削除
}
```

insert()と erase()については、§31.3.7 で詳細に解説する。

ここで示した list の例題は、vector を使っても同じ内容が記述できるし、(マシンアーキテクチャを理解していなければ驚くかもしれないが) 要素数が少ない場合は、list よりも vector のほうが性能がよい。単にデータのシーケンスを使いたければ、vector と list の両方が選択肢となり得る。しかし、特に理由がなければ、vector を使ったほうがよい。走査(find()、count()など)や、ソートと探索(sort()、binary_search()など)では、vector のほうが性能が高い。

4.4.3 map

ペア形式である(name、number)のリストから、名前を探索する処理は、面倒なものだ。しかも、線形探索は、ごく短いリストでなければ効率が悪い。標準ライブラリでは、map という探索木(赤黒木)を提供する：



map は、連想配列や辞書などと呼ばれることもあり、平衡二分木として実装される。

標準ライブラリの map(§31.4.3)は、値のペアをもつコンテナであり、探索に特化されている。初期化子は、vector や list と同じ形式のもの(§4.4.1、§4.4.2)が利用可能だ。

```
map<string, list> phone_book {
        {"David Hume", 123456},
        {"karl Poper", 234567},
        {"Bertrand Arthur William Russell", 345678}
};
```

(キー (key)と 8 呼ばれる)第 1 型の値が添字として与えられると、(**値** (value)あるいは**マッピングされた型**(mapped type)と呼ばれる)第 2 型の値を返す。たとえば:

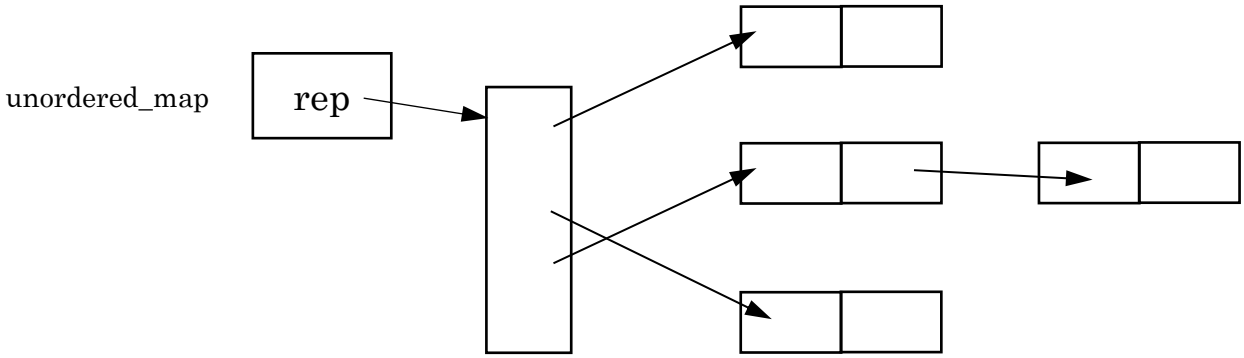
```
int get_number(const string& s)
{
        return phone_book[s];
}
```

換言すると、map の添字演算は、基本的に get_number()で行っている探索のことである。もし key が見つからなければ、value のデフォルト値をもつ要素が、自動的に map に追加される。なお、数値型のデフォルト値は 0 だ。この値は、電話帳の電話番号としては無効な数値である。

電話帳に無効な値が加えられないようにするには、[]ではなくて、find()や insert()を使うとよ(§32.4.3.1)。

4.4.4 unordered_map

map の探索コストは、要素数が n であれば、O(log(n))である。これは、満足できるものだ。たとえば、1,000,000 要素の map を探索する場合でも、要素への比較と関節参照は、たったの約 20 回程度となる。しかし、多くの場合、< 演算子のような順序判定関数を用いた比較演算よりも、ハッシングによる探索のほうが、されらに効率が向上する。標準ライブラリではハッシングによるコンテナ群を、“非順序(unordered)”コンテナと呼んでいる。というのも、それらが順序判定関数を必要としないからだ:



ヘッダファイル<unordered_map>が定義する unordered_map を使うと、電話帳は、以下のように利用できる：

```
unordered_map<string, int> phone_book {
        {"David Hume", 123456},
        {"Karl Poper", 234567},
        {"Bertrand Arthur William Russell", 345678}
};
```

map と同様に、unordered_map でも添字演算が行える：

```
int get_number(const string& s)
{
        return phone_book[s];
}
```

4.4.5 コンテナのまとめ

標準ライブラリは、極めて一般的で有用なコンテナ群を実装しているので、アプリケーションにとって最適なコンテナを、プログラマが選択できる：

標準コンテナの概要	
vector<T>	要素数可変のベクタ(§31.4)
list<T>	双方向結合リスト(§31.4.2)
forward_list<T>	単方向結合リスト(§31.4.2)
deque<T>	両端キュー(§31.2)
set<T>	集合(§31.4.3)
multiset<T>	要素の重複を許す集合(§31.4.3)
map<K,V>	連想配列(§31.4.3)
multimap<K,V>	キーの重複を許す連想配列(§31.4.3)
unordered_map<K,V>	ハッシングによる探索機能をもつ連想配列(§31.4.3.2)
unordered_multimap<K,V>	ハッシングによる探索機能をもち、キーの重複を許す連想配列(§31.4.3.2)
unordered_set<T>	ハッシングによる探索機能をもつ集合(§31.4.3.2)
unordered_multiset<T>	ハッシングによる探索機能をもち、キーの重複を許す集合(§31.4.3.2)

非順序コンテナは、キー(多くの場合は文字列)による探索用に最適化されている。すなわち、

内部でハッシュ表を利用している。

標準コンテナについては、§31.4 で解説する。各コンテナは、<vector>、<list>、<map>などのヘッダファイルで、std 名前空間の中で定義されている(§4.1.2, §30.2)。なお、標準ライブラリは、queue<T>(§31.5.2)、stack<T>(§31.5.1)、priority-queue<T>(§31.5.3)というコンテナアダプタを提供する。さらに、コンテナに似た、より特殊化された型として、要素数を固定した配列 arrray<T,N>(§34.2.1)と bitset<N>(§34.2.2)も提供する。

標準コンテナと、その基本的な処理は、一般的な記法と似たものとなるように設計されている。さらに、コンテナが異なっても処理の意味は変わらない。基本的な演算は、すべての種類のコンテナにおいて、適切な動作を効率的に行うように実装されている。たとえば:

・begin()と end()は、それぞれ、先頭要素への反復子と、末尾要素の直後への反復子を返す。
・push_back()は vector や list などのコンテナ内の末尾に要素を(効率的に)追加する。
・size()は、要素数を返す。

記法と意味に画一性があるので、新しいコンテナが必要になった場合、標準コンテナと同じように使えるものをプログラマが定義できる。先ほどの範囲チェック機能付きの vector(§2.3.2, §2.4.3.1)は、その一例である。コンテナインタフェースの画一性のおかげで、アルゴリズムの指定は、コンテナの方とは無関係に行える。ところが、各コンテナには、向き不向きがある。たとえば、vector の添字演算と走査処理は、低コストで簡単だ。その一方で、vector 内の要素は、挿入や削除によって位置が変化する。ところが、list ではこの点がまったく逆になる。また、少数の小規模の要素をもつ場合、一般に list よりも vector のほうが、(lnsert()や erase()の場合ですら)より効率的であることを覚えておこう。特別な理由がない限り、要素のシーケンスを表す必要があるときは、デフォルトでは標準ライブラリ vector を使うべきだ。

4.5 アルゴリズム

リストやベクタなどのデータ構造は、単独では、それほど有用ではない。コンテナを使う際は、要素の追加や削除などの、(list や vector が提供するような)基本的なアクセス処理が必要だ。また、コンテナを、単なるデータ格納庫として使うことはまれである。コンテナに対しては、ソート、出力、一部要素の抽出、要素の削除、オブジェクトの探索など処理を行う。そのため、標準ライブラリは、極めて一般的なコンテナの型を提供するだけではなく、極めて一般的なコンテナ用アルゴリズムも提供する。たとえば、以下の例は、vector をソートして、一意な要素をリストにコピーする:

```
bool operator(const Entry& x, const Entry& y)      //未満(より小さい)
{
return x.name < y.name;                            //Entry を名前で順序付ける
}

void f(vector<Entry>& vec, list<Entry>& lst)
{
sort(vec.begin(), vec.end());                      //<で順序付け
unique_copy(vec.begin(), vec.end(), lst.begin()); //隣接する等しい要素はコピーしない
}
```

標準アルゴリズムについては、第 32 章で解説する。それらのアルゴリズムは、要素のシーケンスを処理するものだ。なお、**シーケンス**(sequence)は、先頭要素を指す反復子と、末尾要素の直後を指す反復子とで表現される。

図. 4.5 アルゴリズム.jpg

このコードでは、vec.bigen()と vec.end()の反復子が示す範囲のシーケンス、すなわち、vector 内の全要素を sort()によってソートする。書込み(出力)先に対して指定する必要があるのは、書き込み先の先頭要素のみだ。複数の要素を出力する場合は、先頭以降の要素は上書きされる。そのため、エラー発生を避けるためには、lst の要素数は、vec 内に含まれる一意な値の個数以上でなければならない。

新しいコンテナに対して、一意な要素を追加するのであれば、次のような記述することになる：

```
list<Entry> f(vector<Entry>& vec)
{
        list<Entry> res;
        sort(vec.bigen(), vec.end());
        unique_copy(vec.begin(), vec.end(), back_inserter(res));        //res に追加
        return res;
}
```

back_inserter()は、コンテナの末尾に要素を追加するものであり、追加時はコンテナ用に領域を拡張する(§33.2.2)。すなわち、標準コンテナと back_inserter()を使うと、realloc()のような、エラーにつながりやすい C 言語方式のメモリ管理(§31.5.1, §17.5.2)をもっているので、(たとえ list に数千もの要素があっても)res の値の返却は、効率よく行われる。

srot(vec.begin(), vec.end())のような、2 個の反復子を使う形式のコードの記述が面倒であれば、アルゴリズムをコンテナ化して、sort(vec)を定義するとよい(§4.5.6)。

4.5.1 反復子の利用

読者がコンテナに初めて触れたときは、重要な要素を指すための、ごく少数の反復子を使うことになる。begin()と end()が、その典型的な例だ。それだけではなく、多くのアルゴリズムが反復子を返却する。たとえば、標準アルゴリズムの find は、シーケンスから値を探索して、見つけた要素への反復子を返却する：

```
bool has_c(const string& s, char c)
{
        aout p = find(s.begin(), s.end(), c);
        if (p != s.end())
                return true;
        else
                return false;
}
```

標準ライブラリの探索ルゴリズムの多くがそうだが、find は“見つからなかった”ことを伝えるために end()を返却する。そのため、先ほどの has_c()は、もっと簡潔に記述できる：

```
bool has_c(const string& s, char c)        //s は文字 c を含むか？
{
        return find(s.begin(), s.end()) != s.end();
}
```

もっと興味深い例を考えよう。文字列内に存在する、ある特定文字のすべての出現箇所を探索するというものだ。すべての出現箇所は、string の反復子を要素とする vector として表現可能だ。vector はムーブセマンティクス(§3.3.1)を実装しているので、効率的に返却できる。見つかった位置の文字列を変更するかもしれないので、非 const の文字列を与えることにしよう：

```
vector<string::iterator> find_all(string& s, char c)       //s 中のすべての c を探す
{
        vector<string::iterator> res;
        for (auto p = s.begin(); p != s.end(); ++p)
                if (*p == c)
                        res.puch_back(p);
        return res;
}
```

ここまで、通常のループを使って文字列を処理している。反復子 p++で 1 要素ずつ先に進めて、その値に関節参照演算子*でアクセスしている。find_all()は、次のように利用できる：

```
void test()
{
        string m{"Mary had a little lamb"};
        for (auto p : find_all(m, 'a'))
                if (*p != 'a')
                        err << "a bug!¥n";
}
```

find_all()の呼出しを図にすると、以下のようになる：

反復子と標準アルゴリズムは、それらが妥当な意味をもつ、あらゆる標準コンテナで利用可能だ。そのため、先ほどの find_all()は、次のように一般化できる：

```
template<typename C, typename V>
vector<typename C::iterator> find_all(C& c, V v)   //c 中のすべての v を探す
{
        vector<typename C::iterator> res;
        for (auto p = c.begin(); p != c.end(); ++p)
                if (*p == v)
                        res.push_back(p);
        return res;
}
```

typenameは、Cのiteratorが、型であって、整数の7のような値ではないことを、コンパイラに通知する。この実装の詳細は、Iteratorという型別名を導入すると隠蔽できる(§3.4.5):

```
template<typename T>
using Iterator = typename T::iterator;    // Tの反復子

template<typename C, typeame V>
vector<Iterator<C>> find_all(C& c, V v)   // c中のすべてvを探す
{
        vector<Iterator<>> res;
        for (auto p = c.begin(); p != c.end(); ++p)
                if (*p == v)
                        res.push_back(p);
        return res;
}
```

これで、以下のようなコードが記述できるようになる:

```
void test()
{
        string m {"Mary had a little lamb"};
        for (auto p : find_all(m, 'a'))                    // pはstring::iterator
                if (*p != 'a')
                        cerr << "string bug!¥n";

        list<double> ld {1.1, 2.2, 3.3, 1.1};
        for (auto p : find_all(ld, 1.1))
                if (*p != 1.1)
                        cerr << "list bug!¥n";

        vector<string> vs{ "red", "blue", "green", "orange", "green" };
        for (auto p : find_all(vs, "red"))
                if (*p != "red")
                        cerr << "vector bug!¥n";

        for (auto p : find_all(vs, "green")
                *p = "vert";
}
```

反復子を使うと、アルゴリズムとコンテナが分離できる。アルゴリズムは反復子を介してデータを処理するが、そのデータが格納されているコンテナについては何も知らない。その一方で、コンテナは、自身がもつデータに適用されるアルゴリズムについては何も知らない。ただ、要求に応える形で(begin()やend()などの)反復子を提供するだけだ。このような、データとアルゴリズムを分離するモデルによって、汎用的で柔軟性の高いソフトウェアが実現できる。

第3章 4.5.2 反復子の型

反復子は、実際には何だろう? 個々の反復子は、何らかの型のオブジェクトだ。しかし、反復子は、コンテナの型に応じた処理を行うための情報が必要なので、反復子の型には、数多くのものがある。コンテナの種類には加えて、反復子が必要とする個々の要求に対して、反復子の型が異なる可能性がある。たとえば、vectorの反復子は、通常のポインタとなることがある。というのも、vectorの要素を参照するには、ポインタが最適だからだ:

この他にも、vector を指すポインタと添え字とを組み合わせて実装することも可能だ：

このような反復子を使っていれば、範囲チェックが行えるようになる。

なお、list の反復子は、単純なポインタよりも複雑なものとなるはずだ。というのも、一般に list の要素自体は、次の要素がどこに配置されているを知らないからだ。そのため、list の反復子は、以下に示すように、link へのポインタとなるだろう：

すべての反復子に共通するのが、処理に対するセマンティクスと命名規則である。たとえば、反復子に対して、++を適用すると、次の要素を指すようになる。*を適用すると、反復子が指している要素の値が得られる。実際、この単純な規則にしたがってさえいれば、あらゆるオブジェクトが反復子となる(§33.1.4)。反復子の型をユーザが意識しなければならない場面は、ほとんどない。コンテナは、自身の反復子の型を“知っている”し、iterator や const_iterator という一般的な名前で利用できるようになっている。たとえば、list<Entry>::iterator は、list<Entry>の一般的な反復子型である。反復子がどのように定義されているか、その詳細を考える必要は、ほとんどない。

4.5.3 ストリーム反復子

反復子は、コンテナ内の要素のシーケンスを処理する上で、汎用的な有用な概念である。しかし、要素のシーケンスは、コンテナ内だけに見られるものではない。たとえば、入力ストリームは、値のシーケンスを生成するし、出力ストリームに対しては値のシーケンスを書き出せる。そのため、反復子の概念は、出力にも適用可能なのだ。

ostream_iteerator を作りためには、利用するストリームと、出力するオブジェクトの型を指定する必要がある。たとえば：

```
ostream_iterator<string> oo {cout};   //cout に string を書き出す
```

*oo に対して代入を行うと、それが cout へ出力される。たとえば：

```
int main()
{
        *oo = "Hello, ";          // cout<<"Hello, "という意味
        ++oo;
        *oo = world!¥n";          // cout<<"World!¥n"という意味
}
```

これは、単純なメッセージを標準出力へ出力するための、通常とは異なる方法だ。++oo は、ポインタ経由の配列への書込みを模倣したものである。

同様に、istream_iterator は、入力ストリームを、読取り専用コンテナとして扱えるようにするものである。この場合も、ストリームと、期待する値の型の指定が必要だ：

```
istream_iterator<string> ii {cin};
```

入力反復子では、データのシーケンスの表現のために 2 個の反復子が必要なので、入力の終端を表す istream_iterator を提供しなければならない。その入力終端のデフォルト値は、以下の反復子だ：

```
istream_iterator<string> eos{ };
```

ほとんどの場合、istream_iterator と ostream_iterator を直接利用することはない。通常は、アルゴリズムの引数として与える。たとえば、ファイルを読み取って、単語をソートして、重複するものを除去して、その結果を別のファイルへと書き出すプログラムは、以下のようになる：

```
int main()
{
        string from, to;
        cin >> from >> to;                          // 入力元と出力先のファイル名を読み込む

        ifstream is {from};                         // ファイル"from"の入力ストリーム
        isstream_iterator<string> ii {is};          // stream への入力反復子
        isstream_iterator<string> eos { };          // 入力終端のための番兵

        ofstream os {to};                           // ファイル"to"の出力ストリーム
        ostream_iterator<string> oo {os, "¥n"};     // stream への出力反復子

        vector<string> b {ii, eos};                         // b は入力で初期化される vector
        sort(b.begin(), b.end());                           // バッファをソート

        unique_copy(b.begin(), b.end(), oo);        // 重複なく出力にバッファをコピー

        retrun !is.eof() || !os;                    // エラーの状態を返却(§2.2.1, §38.3)
}
```

ifstream は、ファイルとの結び付きが可能な istream であり、ofstream は、ファイルとの結び付きが可能な ostream である。ostream_iterator の第 2 引数は、出力する値の区切りだ。

このプログラムは、実際のものより長くなっている。文字列を vector に読み取って、それを sort()して、重複を排除して出力している。そもそも重複する文字列を保持しなければ、もっとエレガントな解になる。それは、string を set 中に保持することで実現できる。set は、重複する要

素を保持することなく、すべての要素を順序付きで保持する(§31.4.3)。set を使うと、vector を利用してる 2 行を 1 行に置換できるだけでなく、unique_copy()を、もっと単純な copy()に置換できる。

```
set<srtring> b{ii, eos};                    // 入力からの文字列を集める。
copy(b.begin(), b.end(), oo};    // 出力にバッファをコピー
```

ここでは ii、eos、oo という名前を 1 回ずつしか使っていない。そのため、プログラムは、もっと短くできる：

```
int main()
{
        string from, to;
        cin >> from >> to;                          // 入力元と出力元のファイル名を読み込む

        ifstream is {from};                         // ファイル"from"の入力ストリーム
        ofstream os {to};                           // ファイル"to"の出力ストリーム

        // 入力から読み取る
        set<string> b {istream_iterator<string>{is}, istream_iterator<string>{}};
        copy(b.begin(), b.end(), ostream_iterator<string>{os, "¥n"}); // 出力にコピー

        return !is.eof() || !os; // エラーの状態を返却(§2.2.1、§38.3)
}
```

このコードの簡素化によって読みやすくなったかどうかは、嗜好と経験によって変わるだろう。

4.5.4 述語

先ほど示した例題では、アルゴリズムは、シーケンス内の各要素に実行する処理として、単純に“組み込まれた”ものであった。しかし、処理をアルゴリズムのパラメータとしたい場面も多くある。たとえば、find アルゴリズム(§32.4)は、ある特定の値を探索するための簡便な方法を提供する。指定した要件に一致する要素を探索する方法を、さらに一般化したものの一つが、**述語**(predicate)(§3.4.3)だ。ここで、map 内で、42 を超える最初の値を探索する例を考えよう。map の要素は(key, value)なので、map<string, int>のシーケンスから、int の値が 42 を超える pair<const string, int>を探索することになる：

```
void f(map<string, int>& m)
{
        auto p = find_if(m.begin(), m.end(), Greater_than{42});
        // ...
}
```

ここで、Greater_than は比較対象の値(42)を保持する関数オブジェクト(§3.4.3)である：

```
struct Greater_than {
        int val;
        Greater_than(int v) : val {v} { }
        bool operator()(const pair<string, int>& r) { retrun r.second > val; }
};
```

なお、ラムダ式(§3.4.3)を使った別解もある：

```
auto p = find_if(m.begin(), m.end(),
                                    [](const pair<string, int>& r){retrun r.second>42;});
```

第3章 4.5.5 アルゴリズムのまとめ

アルゴリズムの一般的な定義は、“特定の問題を解くための一連の演算を提供する有限個の規則であり、[しかも]五つの重要な機能である、有限性、確定性、入力、出力、効率性をもっているもの”〔Knuth 1986, §1.1〕である。C++標準ライブラリでのアルゴリズムの定義は、要素のシーケンスを処理するための関数テンプレートである。

標準ライブラリは、数十ものアルゴリズムを提供する。いずれも std 名前空間に存在するものであり、ヘッダファイル<algorithm>で定義されている。標準ライブラリのアルゴリズムは、入力としてシーケンスを受け取る(§4.5)。その範囲は、b から e の半開区間[b:e)だ。私が特に有用と考える、いくつかのアルゴリズムを示す：

標準アルゴリズム（抜粋）	
p=find(b, e, x)	[b:e)内で*p==x である最初の p を返す
p=find_if(b, e, x)	[b:e)内で f(*p)==true である最初の p を返す
n=count(b, e, x)	[b:e)内で*q==x である*q の個数を返す
n=count_if(b, e, x)	[b:e)内で f(*p)が真となる*q の個数を返す
replace(b, e, v, v2)	[b:e)内で*q==v である*q を v2 で置換する
replace_if(b, e, f, v2)	[b:e)内で f(*q)が真となる*q を v2 で置換する
p=copy(b, e, out)	[b:e)内の要素を[out:p)にコピーする
p=copy_if(b, e, out, f)	[b:e)内で f(*p)が真となる*q を[out:p)にコピーする
p=unique_copy(b, e, out)	隣接する重複を除き、[b:e)内の要素を[out:p)にコピーする
sort(b, e)	[b:e)内の要素を<基づいてソートする
sort(b, e, f)	[b:e)内の要素を f に基づいてソートする
(p1, p2)=equal_range(b, e, v)	ソートずみの[b:e)内で値 v の出現範囲を返す。基本的に v の 2 文探索。
p=merge(b, e, b2, e2, out)	ソートずみ[b:e)と[b2:e2)のマージ結果を[out:p)に置く

これらを含めた、数多くのアルゴリズム(第 32 章)が、コンテナ内の要素、string、組込み型の配列に適用できる。

第3章 4.5.6 コンテナアルゴリズム

シーケンスは、2 個の反復子[begin:end)で定義される。これは汎用的な上に柔軟だ。とはいえ、アルゴリズムは、コンテナの全体のシーケンスに対して適用することが多い。たとえば：

```
sort(v.begin(), v.end());
```

どうして、単に sort(v)と記述できないのでだろう。ところが、このような短縮版は、以下のように実装することで、簡単に提供できる：

```
namespace Estd {
        using namespace std;

        template<typename C>
        void sort(C& c)
        {
                sort(c.begin(), c.end());
        }

        template<typename C, typename Pred>
        void sort(C& c, Pred p)
        {
                sort(c.begin(), c.end(), p);
        }

        // ...
}
```

私は、sort()(および他のアルゴリズム)のコンテナバージョンを、独自の名前空間 Estd(“拡張 std(extended std)”)に入れている。他のプログラマが、std 名前区間を利用する際の妨げとならないからだ。

4.6 アドバイス

[1] 車輪の再発明をしないように、ライブラリを利用しよう。§4.1。
[2] 選択できるならば、他のライブラリよりも標準ライブラリを優先しよう。§3.1。
[3] 標準ライブラリが万能であると考えないように。§4.1。
[4] 利用する機能のヘッダファイルの**#include** を忘れないように。§4.1.2。
[5] 標準ライブラリ機能は、std 名前空間に定義されていることを覚えておこう。§4.1.2。
[6] C 言語スタイルの文字列(char*：§2.2.5)よりも string を優先しよう。§4.2, §4.3.2。
[7] iostream は型を識別するし、型安全であって拡張性にも優れる。§4.3
[8] T[]よりも、vector<T>、map<K,T>、unordered_map<K,T>を優先しよう。§4.4。
[9] 標準コンテナの長所と短所を把握しよう。§4.4。
[10] デフォルトのコンテナとして、vector を利用しよう。§4.4.1。
[11] コンパクトにまとめられたデータ構造を優先しよう。§4.4.1.1。
[12] 確信がもてない場合は、Vec のような範囲チェック付きベクタを選択しよう。§4.4.1.2。
[13] コンテナへの要素追加には、puch_back()や back_inserter()を使おう。§4.4.1, §4.5。
[14] 配列を realloc()するのではなく、vector に push_back()しよう。§4.5。
[15] main()では、あらゆる例外を捕捉しよう。§4.4.1.2。
[16] 標準アルゴリズムを理解して、手作りのループよりも優先しよう。§4.5.5。
[17] 反復子の利用が面倒であれば、コンテナアルゴリズムとして定義しよう。§4.5.6。
[18] 完全なコンテナには、範囲 for ループが利用できる。

第５章　C++を探検しよう：並列処理とユーティリティ

・はじめに
・資源管理
　unique_ptr と shared_ptr
・並列処理
　タスクと thread／引数の受渡し／結果の返却／データの共有／タスク通信
・小規模ユーティリティ
　時間／型関数／pair と tuple
・正規表現
・数学ライブラリ
　数学関数とアルゴリズム／複素数／乱数／ベクタの算術演算／数値の限界値
・アドバイス

5.1　はじめに

エンドユーザの立場から見れば、理想的な標準ライブラリは、あらゆる基本的なニーズを直接サポートするコンポーネントを提供するものだ。特定分野のアプリケーション用の巨大な商用ライブラリであれば、この理想に近づけるだろう。しかし、それはC++標準ライブラリが目指すものではない。管理しやすくて、普遍的に利用できて、あらゆるプログラムのすべての要求に応えられるライブラリなどは、あり得ない、C++標準ライブラリが目指すのは、大多数のアプリケーション分野で大多数のユーザにとって有益なコンポーネントを提供することだ。すなわち、ユーザ要求の和集合ではなくて、咳集合を満たすことが目的である。さらに、算術演算やテキスト処理などの、少々幅広い重要なアプリケーションのサポートも少しずつ追加している。

5.2　資源管理

ある程度の規模のプログラムであれば、資源を管理することが、処理の要の一つになる。資源は、利用するために獲得して、利用後に(暗黙的あるいは明示的に)解放するものである。その一例が、メモリ、ロック、ソケット、スレッドハンドル、ファイルハンドルなどだ。長時間動作するプログラムでは、適切なタイミングで資源を解放しないと、ある種の“リーク(leak)”が、深刻な性能劣化を招いて、悲惨なクラッシュへとつながる可能性がある。たとえ短いプログラムであっても、リークは障害となる。資源が枯渇すると、実行時間が指数関数的に増えることがある。

標準ライブラリのコンポーネントは、資源リークを発生させないように設計されている。その設計は、資源管理を支援する言語の基本機能の上に成り立っている。コンストラクタとデストラクタを組み合わせることで、オブジェクトが消滅した際に、資源だけが残らないことが保証される。コンストラクタとデストラクタの組合せによって、要素の生存期間を管理する Vector は、その一例であり(§3.2.1.2)、標準ライブラリのすべてのコンテナが同様の手法で実装されている。重要なのは、この手法が、例がを用いたエラー処理と強調できることだ。その手法は、標準ライブラリのロック関連クラスで利用されている：

```
mutex m;          // 共有データへのアクセスの保護のために利用
// ...
void f()
{
        unique_lock<mutex> lck {m};              // mutex m を獲得
        // ... 共有データを操作 ...
}
```

ここでは lck のコンストラクタが、mutex である m を獲得するまでは、実行スレッドの処理は進まない(§5.3.4)。獲得した資源を解放するのは、lck のデストラクタである。すなわち、この例では、スレッドの制御が f()を離れるとき(すなわち、“関数の終端を越えた”とき、あるいは、例外の送出によって戻るとき)に、unique_lock のデストラクタが、mutex を解放する。

これは“資源獲得時初期化”の技法(RAII：§3.2.1.2, §13.3)の一例である。この技法は、C++での資源管理の基盤をなすものだ。(vector や map などの)コンテナや、string や、iostream でも、同様の手法で(ファイルハンドルやバッファなどの)資源を管理している。

5.2.1 unique_ptr と shared_ptr

ここまでの例は、スコープ内で定義されたオブジェクトが、スコープを抜け出るときに、自身が獲得した資源を解放するものであった。それでは、空き領域から割り当てたオブジェクトは、どうすればよいだろう？　標準ライブラリは<memory>で、空き領域にあるオブジェクトの管理を支援するための、2 種類の“スマートポインタ(smart pointer)”を提供ている：

[1] unique_ptr：独占的な所有権を表す(§34.3.1)

[2] shared_ptr：共有された所有権を表す(§34.3.2)

これらの“スマートポインタ”のもっとも基本的な用途は、不注意なプログラミングによって発生するメモリリークの防止である。

```
void f(int i, int j)                        // X* 対 unique_ptr<X>
{
        X* p = new X;                       // 新しい X を確保
        unique_ptr<X> sp {new X};           // 新しい X を確保して、そのポ int 名を unique_ptr に与える
        // ...
        if (i < 99) throw Z{}               // 例外送出の可能性
        if (j < 77) return;                 // "途中で"戻る可能性
        p->do_something;                    // 例外送出の可能性
        sp->do_something;           // 例外送出の可能性
        // ...
        delete p;
}
```

ここまでは、i<99 や j<77 が成立した際の、delete p の実行を“忘れてしまっている”。ところが、unique_ptr のおかげで、f()を終了した理由(例外を送出した、return を実行した、“関数の終端を越えた”)とは関係なく、オブジェクトの適切な解体が、確実に行われる。皮肉なことだが、次に示すように、new を使わず、ポインタも使わなければ、この問題は解決する：

```
void f(int i, int j)  // 局所変数を利用
{
        X x;
        // ...
}
```

残念ながら、new(とポインタと参照)の過度な利用が、問題を悪化させているようだ。

とはいえ、ポインタセマンティクスが本当に必要な場面では、unique_ptr は、極めて軽量だし、組込みのポインタを適切に使うのと比べて空間と時間のオーバーヘッドも発生しない。さらに、空き領域に割り当てたオブジェクトを、関数に渡して、戻してもらうこともできる：

```
unique_ptr<X> make_X(int i)
        // X を作って、すぐに unique_ptr に与える
{
        // ... i のチェックなど ...
        return unique_ptr<X>{new X{i}};
}
```

unique_ptr は、個々のオブジェクト(または配列)、を指すハンドルだ。そのため、オブジェクトのシーケンスのハンドルである vector と似ている。いずれも、別のオブジェクトの生存期間を(RAH で)管理するし、ムーブマンティクスによって、return を簡潔で効率よいものとしている。

shared_ptr は unique_ptr と似ているのだが、自身をムーブではなくコピーする点が異なる。あるオブジェクトに対する所有権を複数の shared_ptr が持っている場合、最後の shared_ptr が解体された時点で、そのオブジェクトも解体される。たとえば：

```
void f(shared_ptr<fstream>);
void g(shared_ptr<fstream>);

void user(const string& name, ios_base::openmode mode)
{
        shared_ptr<fstream> fp {new fstream(name, mode)};
        if (!*fp) throw No_file {};                    // ファイルが正しくオープンされたかどうかを確認

        f(fp);
        g(fp);
        // ...
}
```

ここで、fp のコンストラクタがオープンしたファイルをクローズするのは、fp の最後のコピーを(明示的あるいは暗黙的に)解体する関数である。注意すべき点は、f()や g()が、fp のコピーを保持したまま他のタスクを起動するかもしれないことと、何らかの方法で user()外に存在しているコピーを保持するかもしれないことである。すなわち、shared_ptr は、デストラクタによる資源管理を保持しながら、ある種のガージコレクションの役割を果たす、共有オブジェクトの生存期間を予測することは、コストがかからないわけでも、逆に極端に高いわけでもないが、困難だ。本当に所有権を共有する必要があれば、shared_ptr を用いるべきである。

unique_ptr と shared_ptr が提供されるので、多くのプログラムで、“裸の new の排除”というポリシー(§3.2.1.2)が実現できる。ところが、これら“スマートポインタ”は、現在でも概念的ポインタであるため、私にとっては、資源管理の次善の策となっている。コンテナや他の型が、自身のもつ資源をより高次元の概念レベルで管理するほうが優先だ。特に shared_ptr は、共有オブジェクトに対する所有者の読み書きに足する規則を自身で定めていない。資源管理に関する問題を削減するだけでは、データ競合(§41.2.4)や、そのほかの混乱の原因の解決は容易ではない。

資源に特化して設計された(vector や thread などの)資源ハンドルではなくて、(unique_ptr などの)“スマートポインタ”を使う必要があるのは、どんなときだろう？ べうに驚くことでもないが、その答えは、“ポインタセマンティクスが必要になったとき”である。

- オブジェクトを共有する場合、そのオブジェクトを指すポインタ(あるいは参照)が必要だ。当然、(所有者が一人であることが明らかな場合以外は)shared_ptr を使うことになる。
- 多目的オブジェクトを参照する場合、参照先のオブジェクトの正確な型が(その大きさですら)分からないので、ポインタ(あるいは参照)が必要だ。そのため、当然 unique_ptr を使うことになる。
- 多目的オブジェクトを共有するときは、ほとんどの場合 shared_ptr を使うことになる。

関数からオブジェクトの集合を返す際に、必ずしもポインタを使う必要はない。資源ハンドルであるコンテナを使えば、簡潔かつ効率的に行える。(§3.2.2)。

5.3 変更処理

並行処理は、複数のタスクを同時に実行するものであり、(一つの計算を複数のプロセッサで分担することになる)スループットの向上と、(プログラムの一部分が応答を待っているあいだに他の部分を実行することによる)応答性の向上のために広く使われている。最近のすべてのプログラミング言語は並行処理をサポートしている。C++では、20 年以上利用され、移植性や型安全があるライブラリを進化させた。C++標準ライブラリによってサポートしている。これは、現代のほとんどのハードウェアでサポートされている。標準ライブラリの基本的な目標は、システムレベルの並行処理のサポートであって、洗礼された高レベルの並行モデルを直接提供することではない。高レベルの並行モデルは、標準ライブラリ機能を使って実装されたライブラリとして提供できるものだ。

標準ライブラリでは、単一アドレス空間における複数スレッドの並行実行を直接サポートする。その実現のために、C++は適切なメモリモデル(§41.2)と、一連のアトミック処理(§41.3)を提供する。とはいえ、ほとんどのユーザが目にする並行処理は、標準ライブラリや、標準ライブラリをもとに開発されたライブラリによるものだろう。本節では、並行処理に関する標準ライブラリの主だった機能を、サンプルとともに簡単に解説する。具体的には、thread、mutex、lock()、packaged_task、future である。これらすべての機能は、オペレーティングシステムが提供する機能に直接基づいて作られたものであって、オペレーティングシステムよりも性能的に劣ることはない。

5.3.1 タスクと thread

他の処理と並行的に実行される可能性がある処理は、タスク(task)と呼ばれる。なお、一つのプログラムにおけるシステムレベルのタスクが、スレッド(thread)である。あるタスクの、別のタスクとの並行的な実行は、そのタスクを引数として与えて、<thread>が定期議している std::thread を構築することによって行える。なお、個々のタスクは、関数あるいは関数オブジェクトである：

```
void f();                               // 関数

struct F {                              // 関数オブジェクト
        void operator()();              // F の呼出し演算(§3.4.3)
};

void user()
{
        thread t1 {f};                  // f()は別のスレッドで実行
        thread t2 {F()};        // F()()は別のスレッドで実行

        t1.join();                      // t1 を待つ
        t2.join();                      // t2 を待つ
}
```

join()を呼び出すことで、二つのスレッドの処理が完了するまで user()が終了しないことが保証される。“join する”というのは、“スレッドの終了を待つ”という意味だ。

プログラム内に存在するスレッド群は、同一のアドレス空間を共有する。この点は、通常はデータを直接共有しないプロセスとの違いだ。スレッド群はアドレス空間を共有するので、共有オブジェクトを介することで、スレッド間通信が行える。(§5.3.4)。スレッド間通信は、データ競合(同一変数に対する制御されていない同時アクセス)を防ぐために、ロックやその他の機構によって制御されるのが一般的である。

並行タスクのプログラミングは、極めてトリッキーになる可能性がある。タスク f(関数)と、タスク F(関数オブジェクト)とが、以下に示すものだったら、どうなるだろう：

```
void f() { count << "Hello ";}

struct F {
        void operator()() { cout << "parallel World!¥n"; }
};
```

これは、ひどいエラーの例であある。ここでは、f と F()の両方が、何の同期も行わないままに、同じオブジェクト cout を使っている。その出力結果は予測不可能であるし、他のプログラムの実行状況に左右される。というのも、二つのタスクの処理の実行順序が定義されていないからだ。そのため、以下のような、“奇妙な”出力を行うかもしれない：

PaHerallllel o World!

並行プログラムでタスクを定義する目的は、単純で明白な方法による通信を除いて、タスクを完全に分離することである。もっとも単純な並行タスクは、呼出し側と並行に実行できる関数の実行のことである、と考えよう。その実現に必要なのは、引数を与えること、返却値を得ること、その間に共有データを一切使用しない(データ競合を起こさない)ようにすることだけだ。

5.3.2 引数の受渡し

通常、タスクは、処理の対象となるデータが必要だ。データ(あるいは、データへのポインタや参照)を引数として与えるのは容易である。次の例を見てみよう：

```
void f(vector<double>& vv);                 // v に対して何らかの処理を行う関数

struct F {                                  // 関数オブジェクト：v に対して何らかの処理を行う
        vector<double>& v;
        F(vector<double>& vv) :v{vv} { }
        void operator()();
};

int main()
{
        vector<double> some_vec {1, 2, 3, 4, 5, 6, 7, 8, 9};
        vector<double> vec2 {10, 11, 12, 13, 14};

        thread t1 {f, ref(some_vec)};    // f(some_vec)を別スレッドで実行
        thread t2 {F{vec2}};             // F(vec2)()を別スレッドで実行

        t1.join();
        t2.join();
}
```

見て分かるように、F{vec2}は、受け取った引数であるベクタへの参照を F 内に保持する。F の実行中に、他のタスクが vec2 を使用すると、困ったことになる。なお、vec2 のやりとりが値渡しであれば、このリスクは排除できる。

{f, ref(some_vec)}による初期化では、任意の引数の並びを受け取る、thread 可変個引数テンプレートコンストラクタ(§28.6)を利用している。<functional>が定義する型関数 ref()は、可変個引数テンプレートが some_vec をオブジェクトではなく参照として扱えるようにするために、不本意ながら必要となるものである(§33.5.1)。コンパイラは、第 2 引数以降のすべての引数を、第 1 引数に対する引数として与えた上で、その第 1 引数が実行できるかどうかをチェックして、スレッドに与えるべき関数オブジェクトを生成する。そのため、f()と F::operator()()が同じアルゴリズムを実行するのであれば、これら二つのタスクの扱いは大まかに等価であり、いずれの場合も、実行すべき thread 用の関数オブジェクトが構築される。

5.3.3 結果の返却

§5.3.2 に示した例では、引数が非 const 参照であった。そのようなことを私が行うのは、引数の参照先の値をタスクに変更してほしい時のみである(§7.7)。やや姑息なのだが、結果の返却手段として使われることは、珍しくない。なお、入力引数を const 参照とした上で、結果を置く場所を別の引数として与える方法だと、もっと分かりにくくなる：

```
void f(const vector<double>& v, double* res);          // v から取り出して結果を*res に書く

class F {
public:
        F(const vector<double>& vv, double* p) :v{vv}, res{p} { }
        void operator()();                              // 結果を*res に置く
private:
```

```
        const vector<double>& v;        // 入力元
        double* res;                                    // 出力先
};

int main()
{
        vector<double> some_vec;
        vector<double> vec2;
        // ...
        double res1;
        double res2;

        thread t1 {f, ctrf(some_vec), &res1};   // f(some_vec)を別スレッドで実行
        thread t2 {F{vec2}};                            // F(vec2)()を別スレッドで実行

        t1.join();
        t2.join();

        const << res1 << ' ' << res2 << '¥n';
}
```

このコードは期待どおり動作するし、広く使用されている技法だ。しかし、引数経由で結果を返す方法が特にエレガントであるとは私は思わない。この点については§5.3.5.1 で再考することにしよう。

5.3.4 データの共有

複数のタスクがデータを共有しなければならないことがある。その場合、データへのアクセスを同期させなければならないので、同時にアクセスするタスクを高々1 個に制裁する必要がある。経験豊富なプログラマは、簡単化して考えるかもしれない(たとえば、値が不変なデータを複数タスクが同時にアクセスしても問題ないと考えるだろう)。しかし、実際に複数のオブジェクトに対して同時にアクセスするタスクを高々1 個に制限するにはどうすればよいのだろう？

その解決の基盤となるものが、mutex、すなわち、“相互排他オブジェクト(mutual exclusion object)”である。1 個の thread は、lock()処理によって mutex を獲得する：

```
mutex m;        // mutex を制御
int sh;         // 共有データ

void f()
{
        unique_lock<mutex> lck {m};     // mutex を獲得
        sh += 7;                        // 共有データを操作
}       // mutex を暗黙裏に解放
```

unique_lock のコンストラクタは、(m.lock()の呼び出しによって)mutex を獲得する。他のスレッドが、その mutex をすでに獲得していれば、他のスレッドが処理を完了するまで待機する(“ブロックする”)。あるスレッドが、共有データへのアクスセスを完了したら、unique_lock は(m.unlock()を呼び出すことで)mutex を解放する。このような相互排他とロック機能は、<mutex>で定義されている。

共有データと mutex の対応は、約束ごとである。そのためプログラマは、どの mutex が、どのデータと対応するかを知っていさえすればよい。当然、この方式はエラーにつながりやすいものであり、その対応は、言語のさまざまな手段を通じてはっきりさせるべき点である。たとえば：

```
class Record {
public:
        mutex rm;
        // ...
};
```

Record 型の rec があれば、rec.rm が、rec 中の他のデータをアクセスするより前に獲得すべき mutex であることは、誰にでもわかる。コメントを加えたり、もっとよい名前を付けると、さらに分かりやすくなるだろう。

何らかの処理を実行するために、複数の資源に対して同時にアクセスする必要があることは、珍しいことではない。その場合、デッドロックの発生の可能性が生まれる。たとえば、thread1 が mutex1 をすでに獲得した上で mutex2 を獲得しようとする際に、一方で thread2 が mutex2 をすでに獲得したした上で mutex1 を獲得しようとしているかもしれない。こうなると、両方のタスクの処理が進まなくなる。そのため、標準ライブラリでは、複数のロックを同時に獲得できるようにするための処理を提供している：

```
void f()
{
        // ...
        unique_lock<mutex> lck1 {m1, defer_lock};          // defer_lock: まだ mutex の獲得は試みない
        unique_lock<mutex> lck2 {m2, defer_lock};
        unique_lock<mutex> lck3 {m3, defer_lock};
        // ...
        lock(lck1, lck2, lck3);                            // 3 個すべての lock を獲得
        // ... 共有データを操作 ...
}       //すべての mutex を暗黙裏に解放
```

ここでの lock()によって、すべての引数の mutex を獲得した場合にのみ、処理が進められることになるし、mutex を保持しているあいだはブロックする(“スリープ状態に移行する”)こともない。また、個々の unique_lock のデストラクタのおかげで、thread がスコープを外れる際は、mutex が解放されることになる。

共有データによる通信は、極めて低レベルなものだ。特に問題なのが、複数のタスクによる、処理のどれが完了したか、あるいは、どれが完了していないかを判断する方法を、プログラマが考え出さなくてはならないことだ。この点に関して、データ共有は、引数のコピーとリターンによるものよりも劣っているといえる。しかし、その一方で、データ共有の方が引数のコピーとリターンよりも必ず効率よくなると確信する人々もいる。確かに、膨大な量のデータを処理する場合はそうかもしれないが、ロックとアンロックは、どちらかというと高コストな処理である。さらに最近のマシンは、データコピーをうまく処理する。特に、vector の要素などのように、コンパクトにまとまったデータの場合は極めてうまく処理される。すなわち、通信手段として、データ共有を選択すべきではない。“効率”は、熟慮と計測なしには語れないものだからだ。

5.3.4.1 イベント待ち

thread は、たとえば、他の thread の処理完了や、一定時間の経過などのような、何らかの外部イベントを待つ必要が生じることがある。もっとも単純な“イベント”は、単なる時間の経過だ。

そこで、次の例を考えることにしよう：

```
using namespace std::chrono;                    // §35.2 を参照

auto t0 = high_resolution_clock::now();
this.thread::sleep_for(milliseconds{20});
auto t1 = hight_resolution_clock::now();
cout << duration_cast<nanoseconds>(t1-t0).count() << " nanoseconds passed¥n ";
```

ここで、thread の起動すら行う必要がないことに注意しよう。this_thread は、デフォルトでは、唯一のスレッドを示す(§42.2.6)。

duration_cast を使っているのは、ナノ秒単位の時間を取り出すためだ。時間に関して、より複雑な処理を行いたいのであれば、まずは§5.4.1 と§35.2 を参照しよう。時間関連の機能は、<chrono>で定義されている。

外部イベントによる通信機能は、<condition_variable>が定義する condition_variable で提供される(§42.3.4)。condition_variable のメカニズムによって、thread は、別のスレッド処理完了まで待機される。たとえば、他スレッドの処理結果によって何らかの条件(condtion)(一般にイベント(event)と呼ばれる)が成立するまえ待機させる、といったことが可能だ。

2 個のスレッドが、キューを経由してメッセージをやり取りするという、古典的な(生産者－消費者の)例を考えることにしよう。単純化のために、キュー用の queue と、その queue で競合状態を防ぐためのメカニズムの両方を広域的に宣言して、生産者と消費者がアクセスできるようにする：

```
class Message {                         // 通信すべきオブジェクト
        // ...
};

queue<Massage> mqueue;                  // メッセージキュー
condition_variable mcond;               // 変化する通信イベント
mutex mmutex;                           // ロックのメカニズム
```

queue、condition_variable、mutex は、いずれも標準ライブラリが提供する型である。
消費者 consumer()は、Message を読み取って処理を行う：

```
void consumer()
{
        while(true) {
                unique_lock<mutex> lck {mutex};         // mutex を獲得
                mcond.wait(lck);                        // lck を解放して待機
                                                        // ウェイクアップしたら lck を再獲得
                auto m = mqueue.font();                 // メッセージを取得
                mqueue.pop();
                lck.unlock();                           // lck を解放
                // ... m を処理 ...
        }
}
```

ここでは、queue に対する処理と condition_variable に対する処理を、mutex に対する unique_lock によって明示的に保護している。condition_variable に対する大気は、待機が解除されるまで、ロックの引数を解放する(そのため、キューは空にならない)。そして、待機が終わった後に、もう一度獲得を行う。

これと対応する生産者 producer()は、以下のようになる：

```
void producer()
{
        while(true) {
                Message m;
                // ... メッセージを埋める ...
                unique_lock<mutex> lck {mutex};        // 処理を保護
                mqueue.push(m);
                mcond.notify_one();                    // 通知
        }                                              // (スコープの終端で)lock を解放
}
```

condition_variable を使うことで、エレガントで効率的な共有が達成できる。しかし、少々トリッキーになる可能性がある(§42.3.4)。

5.3.5 タスク間通信

標準ライブラリは、プログラマが低いレベルのスレッドやロックを直接操作しなくてすむように、概念レベルのタスク処理(並行実行される可能性がある処理)を実現できるようにするための、いくつかの機能を提供する：

[1] future と promise：別スレッド上で起動されたタスクから値を返却する。

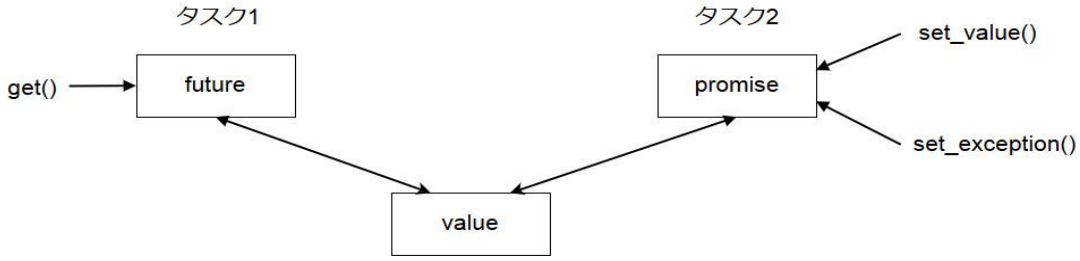
[2] packaged_task：タスク起動を支援して、結果を返すメカニズムとの連携を行う。

[3] async()：関数呼出しと極めた似た方式でタスクを起動する。

これらの機能は、<future>で定義されている。

5.3.5.1 future と promise

future と promise の重要な点は、ロックを見次的に使わずに、タスク間で値を転送できるようにすることだ。転送を効率的に行うのは“システム”である。基本的な考え方は単純だ。あるタスクが、別のタスクに値を転送するときは、その値を promise の中に入れる。そうすると、処理系がその値を対応する future へと置くので、そこから、その値を(通常は、タスクの起動元が)読み取れるようになる。図にすると、以下のような感じだ：



fx という名前の future<x>があれば、そこから X 型の値を get()できる：

```
X v = fx.get();              // 必要であれば、計算された値が得られるまで待機
```

値がまだ置かれていなければ、そのスレッドは値が到着するまでブロックされる。もし値が計算不能であれば、get()は例外を送出する可能性がある(送出元は、システムの場合もあるし、値を get()しようとしたタスクから転送されてくる場合もある)。

promise の主な目的は、future の get()と対になる。単純な“置く(put)”処理(すなわち、set_value()と set_exception())を提供することであある。名前の“future”と“promise”は、過去の経緯によるものだ。どうか、私を非難しないで信じてほしい。なお、これらの語句は、駄洒落のネタである。

さて、promise を所有していて、X 型の結果を future へと送る必要が生じた場合に、行える処理は、値を送るか、例外を送出するかのどちらかだ。たとえば：

```
void f(promise<X>& px)                  // あるタスク：結果を px に置く
{
        // ...
        try {
                X res;
                // ... res の値を計算 ...
                px.set_value(res);
        }
        catch (...) {                   // おっと：res が計算できない
                // future のスレッドに例外を渡す
                px.set_exception(current_exception());
        }
}
```

current_exception()が返却するのは、捕捉した例外への参照だ(§30.4.1.2)。

future を経由して送られた例外を処理するためには、get()の呼出し側は、その例外を捕捉するようにどこかで準備しておく必要がある。たとえば：

```
void g(future<X>& fx)                   // あるタスク：fx から結果を取得
{
        // ...
        try {
                X v = fx.get();         // 必要であれば、計算された値が取得できるまで待機
                // ... v を利用 ...
        }
        catch(...) {                    // おっと：誰かが v を計算できなかった
                // ... エラーの処理 ...
        }
}
```

5.3.5.2 packaged_task

結果を必要とするタスクに future をもたせるには、どうするばよいのだろう？　結果を生成するスレッドに対応する promise をもたせるには、どうすればよいのだろう？　複数の thread 上で動作して、複数の future promise と連携する複数のタスクの準備を支援するのが、pakcaged_task 型である。packaged_task は、タスクからの返却値や例外を promise に置くためのラッパーコード(§5.3.5.1 に示したコードに似たコード)を提供する。その指示を、get_future を呼び出すことによって行うと、packged_task が、その promise に対応する future を返してくれる。標準ライブラリの accumulate()(§3.4.2、§40.6)を使って vector<double>の要素の半分を合計するための、2 個のタスクを実行する例を考えよう：

```
double accum(double* beg, double* end, double init)
        // [beg:end)の計画を初期値 init で開始して計算する
{
        return accumulate(beg, end, init);
}

double comp2(vector<double>& v)
{
        using Task_type = double(double*, double*, doulbe);        // タスクの型

        packaged_task<Task_type> pt0 {accum};                      // タスク(すなわち accum)をパッケージ
        packaged_task<Task_type> pt1 {accum};

        future<double> f0 {pt0.get_future()};              // pt0 の future を入手
        future<double> f1 {pt1.get_future()};              // pt1 の future を入手

        double* first = &v[0];
        thread t1 {move(pt0), first, first+v.size/2.0};            // pt0 用スレッドを開始
        thread t2 {move(pt1), first+v.size()/2, first+v.size(),0};  // pt1 用スレッドを開始

        // ...
        return f0.get() + f1.get();                        // 二つの結果を取得
}
```

packaged_task テンプレートは、テンプレート引数として、タスクの型(例では double(double*, double*, double)の別名である Task_type)を受け取り、そのコンストラクタは、引数としてタスク(この例では accum)を受け取る。なお、move()処理が必要となっているのは、packaged_task がコピーできないからだ。

注目すべき点は、このコードが明示的なロックを行っていないことだ。そのおかげで、通信の管理に気をとられずに、実行すべきタアスクに集中できる。2 個のタスクは、別々のスレッドとして実行するので、並列的に実行される可能性がある。

5.3.5.3 async()

本章では、もっとも単純であると同時に、もっとも説得力があると私が考えている方法で解説を行ってきた。タスクは、他のタスクとへ交に実行できる関数として扱った。これは、C++標準ライブラリが提供する唯一のモデルというわけではないが、幅広い要求に応えられるものである。必要であれば、共有メモリを用いたプログラミングスタイルなど、もっと微妙なトリッキーなモデルも使用できる。

非同期に実行される可能性がるタスク起動には、async()が利用できる：

```
double comp3(vector<double>& v)
{
        if (v.size() < 10000) return accum(v.begin(), v.end(), 0.0);

        auto v0 = &v[0];
        auto sz = v.size();

        auto f0 = async(accum, v0, v0+sz/4, 0.0);                  // 第 1 クォータ
        auto f1 = async(accum, v0+sz/4, v0+sz/2, 0.0);             // 第 2 クォータ
        auto f2 = async(accum, v0+sz/2, v0+sz*3/4, 0.0);   // 第 3 クォータ
        auto f3 = async(accum, v0+sz*3/4, v0+sz, 0.0);             // 第 4 クォータ

        return f0.get()+f1.get()+f2.get()+f3.get();                // 結果を集めて組み合わせる
}
```

基本的に、async()は、関数呼出しの“呼出し部分”を“結果を得る部分”から分離した上で、自身のタスクらから、それらを独立させる。async()を使うと、スレッドやロックの考慮が不要となる。すなわち、非同期に行われる可能性がある演算結果を求めるタスクの考慮だけが必要だ。ここで、明白な制限が一つある。ロックが必要な資源を共有するタスクに対しては、async()を使ってはいけないことだ。async()は、thread 数を決定する前に、利用できるアイドルなコア(プロセッサ)数を確認するかもしれない。

async()が、性能を向上させる並列計算のために特化したものでないことに注意しよう。たとえば、“メインプログラム”をアクティブにしたまま、ユーザからの情報を取り出すタスクの起動などの用途でも利用できる(§42.4.6)。

5.4 小規模ユーティリティ

標準ライブラリのコンポーネントは、“コンテナ”や“入出力”などのような、分かりやあすい名前をもつものだけではない。本節では、小規模だが幅広く有用な、いくつかのコンポーネントの例を紹介する：

・clock と duration：時間の計測

・iterator_traits や is_arithmetic などの型関数：型情報の取得。

・pair と tuple：要素の型がことなってもよい、少数の値の集合。

ここで重要なのは、そもそも関数や型は、複雑なものでなくてよいし、他の多数の関数や型と密接な関係をもたなくてもよいことだ。このようなコンポーネントのほとんどが、標準ライブラリの他のコンポーネントを含めた、より強力なライブラリ機能の構築要素として働く。

5.4.1 時間

標準ライブラリは、時間を扱う機能を提供する。たとえば、何らかの処理時間を測定する基本的な方法は、以下のようになる：

```
using namespace    std::chrono;                          // §35.2 を参照
auto t0 = high_resolution_clock::now();
do_work();
auto t1 = high_resolution_clock::new();
cout << duration_cast<milliseconds>(t1 - t0).count() << "msec¥n";
```

時間のクロックは、time_point(時間の流れのある一点)で表される。2 個の time_point を減算すると、duration(期間)が得られる。時間を表現する単位は、クロックによってさまざまである(私が使っているマシンのクロックは nonoseconds 単位だ)。そのため、duration は、目的の単位へと変化するとよい。それを行うのが、duration_cast である。

時間を処理する標準ライブラリ機能は<chrono>で、std::chrono 部分名前空間の中で定義されている(§35.2)。

時間を実測しない限り、コードの“効率性”について語るべきではない。性能に対する単なる推測は、まったく信用できないものである。

5.4.2 型関数

型関数(type function)は、引数あるい返却値として、ある型が与えられて、コンパイル時に評価される関数である。ライブラリ開発者と、言語・標準ライブラリ・一般的なコードの観点からコーディングするプログラマを支援するために、標準ライブラリは豊富な型関数を提供する。

たとえば、数値型に対しては、<limits>が提供する numeric_limits が、有用な情報を豊富に提供している(§5.6.5)。たとえば：

```
constexpr float min = numeric_limits<float>::min();  // float の正の最小値(§40.2)
```

同様に、オブジェクトの大きさは、組込みの sizeof 演算子で得られる(§2.2.2)。たとえば：

```
constexpr int szi = sizeof(int);      // int のバイト数(char がバイトにフィットすると仮定)
```

これらの型関数は、より厳密な型チェックと、より高い性能を可能な限り現実するための、C++のコンパイル時算出メカニズムの一部となっている。このような機能は、メタプログラミング(metaprogramming)と呼ばれる。また、(テンプレートが併用されていれば)テンプレートメタプログラミング(templete metaprograamming)とも呼ばれる。(第 28 章)。ここでは、標準ライブラリから二つの機能を紹介する。iterator_traites(§5.4.2.1)と型述語(§5.4.2.2)である。

5.4.2.1 iterator_traits

標準ライブラリの sort()は、引数として、シーケンスを定義するための 2 個の反復子を受け取る(§4.5)。それらの反復子は、シーケンス内をランダムにアクセスできるものでなければならない。すなわち、ランダムアクセス反復子(raudom-access iterator)でなければならない。ところが、forward_list などの一部のコンテナは、その反復子をもっていない。たとえば、forward_list は単方向結合リストなので、添付演算は高コストであるし、直前の要素を取り出すための理にかなった方法すら存在しない。しかし、多くのコンテナがそうであるように、forward_list は前進反復子(forward iterator)を実装しているので、各種アルゴリズムや for 文が、シーケンスを走査できるようになっている。(§33.1.1)。

標準ライブラリは、iteraor_traits を提供するので、どの種類の反復子が実装されているかの確認が行えるようになる。これを使えば、vector や forward_list の受取りが可能となるように§4.5.6 に示した sort()を改良できる。たとえば：

```
void test(vector<string>& v; forward_list<int>& lst)
{
        sort(v);   // ベクタをソート
        sort(lst); // 単方向結合リストをソート
}
```

このコードを実行できるようにするため必要なテクニックは、一般的にも有効なものだ。まずは、2 個のヘルパ関数を作る。いずれもランダムアクセス反復子と前進反復子のどちらを

使うべきかを表すための引数が追加されたものだ。ランダムアクセス反復子を使うバージョンは、ちょっとしたものだ：

```
// ランダムアクセス反復子用。[beg:end)内では添字演算が利用できる：
templete<typename Run>
void sort_helper(Ran beg, Ran end, random_access_iterator_tag)
{
        sort(beg, end)                  // ソートするだけ
}
```

前進反復子を使用するバージョンは、リストをいったん vector にコピーして、ソート後にコピーを戻すだけだ：

```
// 前進反復子用。[beg:end)は 1 要素ずつ順に走査できる：
template<typename For>
void sort_helper(For beg, For end, forward_iterator_tag)
{
        vector<Value_type<For>> v {beg, end};  // [beg:end)から vector を初期化
        sot(v.begin(), v.end());
        copy(v.begin(), v.end(), beg);                  // 要素をコピーして戻す
}
```

Value_type<For>は、For の要素の型であり、値型(value type)と呼ばれる。標準ライブラリのすべての反復子は value_type メンバをもっている。Value_type<For>という表記を行うには、次の示す型別名(§3.4.5)が必要だ：

```
templete<typename C>
        using Value_type = typename C::value_type;      // C の値型
```

そのため、X が入力シーケンスの要素の型であれば、v は vector<X>となる。本当の“型の魔法”は、ペルパ関数の選択にある：

```
templete<typename C>
void sort(C& c)
{
        using Iter = Iterator_type<C>;
        sort_heler(c.begin(), c.end(), Iterator_category<Iter>{});
}
```

このコードでは、2 個の型関数を使っている。Iterator_type<C>は、C の反復子型(すなわち C::iterator)を返すものだ。もう一つの Iterator_category<Iter>{}は、反復子の種類を表す“タグ”の値を構築する：

- std::random_access_iterator_tag：C の反復子が、ランダムアクセス処理をサポートする。
- std::forward_iterator_tag：C の反復子が、前進反復処理をサポートする。

この情報が得られると、2 種類のソートアルゴリズムからの選択を、コンパイル時に行えるのだ。このテクニックは、タグ指名(tag dispatch)と呼ばれるものであり、柔軟性と性能を向上させるために、標準ライブラリの中はもちろん、それ以外のコードでも利用されている。

標準ライブラリは、タグ指名のような、反復子を利用するテクニックのサポートのための単純なクラステンプレート iterator_traits を、<iterator>で定義している(§33.1.3)。そのため、sort()で使われているような型関数は、容易に定義できる：

```
templete<typename C>                    // C の反復子型
        using Iterator_type = typename C::iterator;

templete<typename Iter>                 // Iter のカテゴリ
        using Iterator_category = typename std::iterator<Iter>::iterator_category;
```

標準ライブラリの機能を実現するために、どのような“コンパイル時の型の魔法”が使われているのかを知りたくないのであれば、iterator_traits などの機能を無視しても構わない。しかし、そうすると、自身のコードを改良するテクニックが使えなくなってしまう。

5.4.2.2 型述語

標準ライブラリの型述語は、型に関する基本的な情報を返却するだけの、単純な型関数だ。次の例を考えよう：

```
bool b1 = Is.arithmetic<int>();         // そう、int は算術型
bool b2 = Is.arithmetic<string>();      // いや、std::string は算術型ではない
```

このように述語は、§35.4.1 で解説する<type_traits>で定義されている。たとえば、is、class、is_pod、is_literal_type、has_virtual_destructor、is_base_of などがある。いずれも、テンプレートを作成する際に、極めて有用となる。たとえば：

```
templete<typename Scalar>
class complex {
        Scalar re, im;
public:
        static_assert(Is_arithmetic<Scalar>(),
                "Sorry, I only support complex of arithmetic types");
        // ...
};
```

私が独自の型関数を定義した上で利用しているのは、標準ライブラリをそのまま利用するよりも、コードが読みやすくなるからだ：

```
templete<typename T>
constexpr bool Is_arithmetic()
{
        return std::is_arithmetic<T>::value;
}
```

古いプログラムでは、( )ではなくて、::value の記述が必要だった。しかし、これだと実装の詳細が丸見えになるので、見苦しいコードだと私は考えている。

5.4.3 pair と tuple

単なるデータが必要となるのは、よくあることだ。具体的には、的確に定義されたセマンティクスと不要条件をもつクラスオブジェクトではなく(§2.4.3.2、§13.4)、いくつかの値の集合だ。そのようなものは、適切に名前を与えたいくつかのメンバで構成される、単純な struct として定義できる。しかし、その定義は、標準ライブラリのアルゴリズム equal_range(§32.6.1)は、任意の述語が成立するシーケンスを表す反復子の pair を返却する

```
template<typename Forward_iterator T, typename Compare>
        pair<Fowrad_iterator, Fowrad_iterator>
        equal_range(Forward_terator first, Foward_iterator last, const T& val,
                Compare cmp);
```

この equal_range( )は、ソートずみシーケンス「first：last」が与えられると、述語 cmp が成立するシーケンスを表す pair を返却する。これを使うと、ソートずみの Record に格納されているソートずみシーケンスからの探索が行える：

```
auto rec_lt = [](const Record& r1, const Recorad& r2)
                { return r1.name<r2.name; };          // 名前を比較

void f(const vector<Record>& v)                       // v が"name"フィールドに格納されていると仮定
{
        auto er = equal_range(v.begin(), v.end(), Record{"Reg"}, rec_lt};
        for (auto p = er.first; p != er.second; ++p)  // すべての一致レコードを表示
                cout << *p;                           // Record 用に<<が定義されていると仮定
}
```

pair の第 1 メンバ名は first で、第 2 メンバ名は second である。この命令は、創意的ではないので、最初は多少の違和感を抱くかもしれない。しかし、はんようてきなコードを書く際には、整合的な名前の方がありがたいものだ。

<utility>で定義されてる標準ライブラリ pair は、標準ライブラリに限らず、あらゆる箇所で極めて頻繁に利用されている。要素が=、==、<などの演算子を実装していれば、pair もそれらを提供する。make_pair( ) 関数を使うと、明示的な型の記述が不要となるため、pair の作成が容易になる(§32.2.4.1)。たとえば：

```
void f(vector<string>& v)
{
        auto pp = make_pair(v.begin(), 2);            // pp は pair<vecotr<string>::iterator, int>
        // ...
}
```

もし要素数が 2 個以上であれば(2 個未満であっても)、<utility>で定義される tuple が使用できる(§34.2.4.2)。tuple は、異種要素のシーケンスである。たとえば；

```
tuple<string, int, double> t2 {"Sild", 123, 3.14};        // 型が明示的に指定されている

auto t = make_tuple(string{"Herring"}, 10, 1.23);         // 型が省略されている

string s = get<0>(t);                 // 先頭要素を取得："Herring"
int x = get<1>(t);                    // 10
double d = get<2><t>;                 // 1.23
```

pair では各要素に(first と second という)名前があったが、tuple の要素には(0 から始まる)番号が与えられている。コンパイル時に要素を取り出すときに、get(t, 1)や t[1]とはできない。心底から不本意なことなのだが、get<1>(t)という見苦しい記述が必要だ(§28.5.2)。

pair と同様に、tuple でも、要素が実装していれば、代入や比較を行える。

pair はインターフェースの中で良く使われている。というのも、たとえば、結果と、結果の性質を示す値のように、複数の値を返却することがあるからだ。3 個以上の値を返すことは、それほど多くないので、tuple は汎用アルゴリズムの実装でよく使われる。

5.5 正規表現

正規表現は、テキスト処理のための強力な道具だ。(たとえば TX 77845 のような米国郵便番号や 2009-06-07 のような ISO 型式の日付などの)テキスト内のパターンを単純かつ簡潔に表現する手段を提供するし、そのようなパターンの効率よい探索の手段を提供する。標準ライブラリは<regex>で、std::regex クラスと、それを補助する関数とで正規表現のサポートを提供する。
regex ライブラリのスタイルの感じをつかむために、一つのパターンを定義して表示してみよう：

```
regex pat (R"(¥w{2}¥s*¥d{5}(-d{4})?");          // 郵便番号のパターン：XXddddd-dddd その変種
```

言語にかかわらず、正規表現の経験があれば、¥w{2}¥s*¥d{5}(-d{4})?には馴染みがあるだろう。これが表すのは、2 個の文字で始まって(¥w{2})、ゼロ個以上の繰り返しが続く(-¥d{4})というパターンである。正規表現に不慣れであれば、それを学ぶちょうどよい機会だ(〔Stroustrup, 2008〕、〔Maddock, 2009〕、〔Friedl, 1997〕)。
正規表現については、§37.1.1 でまとめている。
本書では、パターンの表現に言文字列リテラル(raw string literal)(§7.3.2.1)を利用する。これは、R"(で始まって、)"で終わる文字列であり、逆斜線や引用符を直接文字列内に記述できるものだ。
パターンを使う利用例としてもっとも簡単なものが、ストリームからの探索だ：

```
int lineno = 0;
for (string line; getline(cin, line); ) {                    // 行バッファに読み込む
        ++lineno;
        samtch matches;                                      // 一致した文字列が入る
        if (regex_search(line, matches, pat))                // line 中の pat 探索
                count << lineno << ": " << matchhes[0] << '¥n';
}
```

regex_serch(line, matches, pat)は、正規表現 pat に一致するものを line から探索して、一致したものを matches に格納する。まったく一致しなければ、regex_search(line, matches, pat)の返却値は false になる。

5.6 数学ライブラリ

C++は本来、数値演算用に設計されたものではなかった。ところが今や、数値演算でも広く使用されているし、それが標準ライブラリにも反映されている。

5.6.1 数学関数とアルゴリズム

<cmath>では、"通常の数学関数"が定義されている(§40.3)。たとえば、float 型、double 型、long double 型を引数に受け取る sqrt()、log()、sin()などだ。そして、これらの関数の複素数バージョンが、<comlex>で定義されている(§40.4)。
<numeric>では、accumulate()などの汎用数値アルゴリズムが定義されている。たとえば：

```
void f()
```

```
{
        list<double> lst {1, 2, 3, 4, 5, 9999.99999};
        auto s = accumulate(lst.begin(), lst.end(), 0.0);          // 合計を求める
        cout << s << '¥n';                                         // 10014.9999 と表示
}
```

汎用数値型アルゴリズムは、標準ライブラリが提供するすべてのシーケンスに対して適用可能なものであり、演算を引数としてあたえることもできる。(§40.6)。

5.6.2 複素数

標準ライブラリでは、§2.3 で解説した complex を始めとする、複素数ファミリを提供する。要素となるスカラとして、単精度浮動小数点(float)や、倍精度浮動小数点数(double)などをサポートするために、標準ライブラリ complex はテンプレートとなっている：

```
template<typename Scalar>
class complex {
public:
        coomplex(const Scalar& re = {}, const Scalar& im = {});
        // ...
};
```

複素数に対する一般的な算術演算と、よく使われる数学関数も提供されている。たとえば：

```
void f(complex<float> fl, complex<double> db)
{
        complex<long double> ld {fl+sqrt(db)};
        db += fl * 3;
        fl = pow(1 / fl, 2);
        // ...
}
```

sqrt()と pow() (べき乗)は、<complex>で定義されている数多くの数学関数の一つである。詳細は§40.4 で解説する。

5.6.3 乱数

乱数は、多くの文脈で有用である。たとえば、テスト、ゲーム、シミュレーション、セキュリティなどだ。アプリケーション分野が幅広いため、標準ライブラリの<ramdom>では、多様な乱数生成関数が提供されている。乱数生成関数は、以下の二つの要素で構成されている：

[1] エンジン(engine)：乱数または疑似乱数を生成する。

[2] 分布(distribution)：生成した値を一定範囲の数学的分布へとマップする。

分布としては、uniform_int_distribution(すべての整数の出現率がほぼ同じ一様分布)、normal_distrabution(“釣鐘型”に分布する正規分布)、exponential_distribution(べき乗的に増加する指数分布)などあり、いずれも値の範囲の指定を行える。たとえば：

```
using my_engine = default_random_engine;                   // エンジンの型
```

```
using my_distribution = uniform_int_distribution<>;              // 分布の型

my_engine re {};                          // デフォルトのエンジン
my_distribution one_to_six{1, 6};         // int の 1〜6 にマップする分布
auto die = bind(one_to_six, re);          // 生成器を作る

int x = die();                            // さいころを振る：x は[1:6]中の値となる
```

標準ライブラリの bind()は、第 2 引数(ここでは re)を受け取って、第 1 引数(ここでは one_to_six)を実行する関数オブジェクトを作成する(§33.5.1)。そのため、die()の呼び出しは one_to_six(re)の呼出しと等価である。

汎用性と性能に関して妥協を許さない配慮のおかげで、熟練者は、乱数生成ライブラリを、“誰もが必要となるように、ライブラリは成長するべきもの”と考えるかもしれない。しかし、“初心者にも使いやすいようなもの”と受け取るには無理がある。ここでは using を用いたので、少しは分かりやすくなっているが、次のようにも記述できる：

```
auto die = bind(uniform_int_distribution<>{1, 6}, default_random_engine{});
```

どちらのコードが読みやすいかは、利用する文脈と読者に完全に依存する。

初心者にとっては(これまでどんな経験をもっていようが)、乱数生成ライブラリのインターフェースが完全に汎用化されていることが、大きな障害となり得る。そのため、単純な一様乱数の生成から始めるとよいだろう。たとえば：

```
Rand_int rnd {1, 10};                     // [1:10]用の乱数生成器を作る
int x = rnd();                            // x は[1:10]中の値となる
```

このコードが動くようにするには、どうするればよいだろう？　rand_int クラスの中に die()のようなものが必要だ：

```
class Rand_int {
public:
        Rand_int(int low, int high) :dist {low, high} { }
        int operator()() { return dist(re); }
private:
        default_random_engine re;
        uniform_int_distribution<> dist;
};
```

この定義は、依然として“専門家レベル”だ。しかし、Rand_int()を使うだけであれば、C++入門コースの第一週目でも行えるだろう：たとえば：

```
int main()
{
        Rand_int rnd {0, 4};                              // 一様乱数生成器を作る

        vector<int> histogram(5);               // 要素数 5 の vector を作る
        for (int i = 0; i != 200; ++i)
                ++histogram[rnd()];                               // histogram[0:4]の出現回数で埋める

        for (int i = 0; i != histogram.size(); ++i) {       // 棒グラフを表示
                        cout << i << '¥t';
```

```
                for (int j = 0; j != histogram[i]; ++j) cout << '*';
                cout << endl;
        }
}
```

実行結果は、(面白くもなんともない)一様分布そのものである(ただし、理にかなった範囲での統計的変動は含まれている) :

```
0 ****************************************
1 ****************************************
2 *************************
3 ****************************************
4 ************************************
```

C++には、標準グラフィックライブラリは存在しないので、ここでは、“ASII § 文字列” を使っている。ご存知のとおり、オープンソースや商用の C++用グラフィックライブラリや、GUI ライブラリが存在する。とはいえ、本書では ISO 標準の機能だけを使うようにしている。

乱数についての詳細は、§ 40.7 を参照しよう。

5. 6. 4 ベクタの算術演算

§ 4.4.1 で解説した vector の設計は、複数の値を保持するための汎用的なメカニズムを提供するものであり、柔軟であり、コンテナと反復子とアルゴリズムのアーキテクチャに沿うものだ。ところが、ベクタの算術演算がサポートされていない。vector に算術演算子を追加するのは容易だが、その汎用性と柔軟性が、数学的な分野で極めて重要な要素である最適化を妨げてしまうことになる。そのため、標準ライブラリでは<valarray>で、vector に似たテンプレート valarray を提供している。これは汎用性を低めることによって、数値演算を最適化しやすくするものだ :

```
templete<typename T>
class valarray {
        // ...
};
```

valarray では、通常の算術演算と、利用頻度の高い数学関数が提供されている。たとえば :

```
void f(valarray<double>& a1, valarray<double>& a2)
{
        valarray<double> a = a1 * 3.14 + a2 / a1;               // *, +, /, =などの数学的配列演算
        a2 += a * 3.14;
        a = abs(a);
        double d = a2[7];
        // ....
}
```

詳細については § 40.5 を参照しよう。注目すべき点は、valarray が、多次元配列の演算を支援するためのストライドアクセスを提供していることだ。

5. 6. 5 数値の限界値

標準ライブラリは<limits>で、たとえば float 型の指数の最大値や int 型のバイト数などのような、組込み型の性質を表すクラスを提供する。詳細は § 40.2 を参照しよう。たとえば、char 型が符号付き型であれば警告を

発するコードは以下のようになる：

```
static_assert(numeric_limits<char>::is_signed, "unsigned characters!");
static_assert(1000000<numeric_llimits<int>::max(), "samll ints!");
```

2 番目の static_assert()は、numeric_limits<int>::max()が constexpr 関数の場合に(のみ)動作することに注意しよう(§2.2.3, §10.4)

5.7 アドバイス

[1] 資源管理には、資源管理ハンドルを使おう(RAII)。§5.2。
[2] 多相型のオブジェクトを使う場合は、unique_ptr を使おう。§5.2.1。
[3] 共有オブジェクトを使う場合は、shared_ptr を使おう。§5.2.1。
[4] 並列処理を使うときは、型安全なメカニズムを使おう。§5.3。
[5] 共有データは、最小限に抑えよう。§5.3.4。
[6] 熟考なしの“効率”や計測に基づかない予測から、通信手段として共有データを選択しないように。§5.3.4。
[7] スレッドではなく、並列タスクとして考えよう。§5.3.5。
[8] ライブラリが有用となるためには、巨大になる必要もないし、複雑になる必要もない。§5.4。
[9] 効率を議論する際は、プログラムの実行時間を事前に実測しておこう。§5.4.1。
[10] 型の性質に明示的に依存するプログラムを記述することができる。§5.4.2。
[11] 単純なパターンマッチングには、正規表現を使おう。§5.5
[12] 言語機能だけで重要な数値演算を行おうとしないように。ライブラリを使おう。§5.6。
[13] 数値型の性質は、numeric_limits から得られる。§5.6.5。

第Ⅱ部　基本機能

第Ⅱ部では、C++の組込み型と、C++を用いたプログラム開発の基本を解説する。古典的なプログラミングスタイルを支援する C++の追加機能とともに、C++のサブセットとしての C 言語も解説する。さらに、論理的あるいは物理的なパーツでプログラムを構成するための基本機能についても解説する。

「私は長い間、すべての問題に対する哲学者たちの決定を疑念に抱いてきた。だから彼らと結論に同意するよりも、反論したい気持ちを大きくもつようになった。彼らがほとんど例外なく犯しやすい一つの間違いがある。すなわち、彼らはあまりにも原理の範囲に閉じこもり、自然がそのすべての働きにおいて大いに愛好してきたあの大きな多様性をなんら説明していないということである。ある哲学者がおそらく多くの自然的な結果を説明するお気に入りの原理をひとたび手に入れると、彼らはその同じ原理を創造物全体にまで広げ、最も乱暴で馬鹿げた推論によって、あらゆる現象をその同一の原理に帰してしまう。・・・・・・」

— デイヴィッド・ヒューム

第 6 章　型と宣言

完璧とは、崩壊寸前に達成されるものだ。
― C・N・パーキンソン

・ISO　C++標準
　処理系／基本ソース文字セット
・型
　基本型／論理型／文字型／整数型／浮動小数点／接頭語と接尾語／void／大きさ／アライメント
・宣言
　宣言の構造／複数の名前の宣言／名前／スコープ／初期化／型の導出：auto と decltype()
・オブジェクトと値
　左辺値と右辺値／オブジェクトの生存期間
・型別名
・アドバイス

6.1 ISO C++標準

C++言語と標準ライブラリの仕様は、ISO 標準：ISO/IEC 14882:2011 で定義されている。本書で ISO 標準を参照する場合は、§iso.23.3.6.1 という形式で表記する。本書の内容が不正確とか、不完全とか、誤っているかもしれないと感じた場合は、ISO 標準を参照しよう。ただし、ISO 標準がチュートリアルであるとか、初心者にも分かりやすいなどと期待しないように。

C++言語とライブラリの標準仕様に厳密にしたがうだけで、よいコードになるという保証はないし、可搬性が確保されるわけでもない。標準は、コードのよし悪しについては何も言及していない。プログラマが、処理系に対して何を求めることができて、何を求めることができないかを述べているだけだ。標準で可搬性が保証されていない機能に依存している。というのも、C++では直接に表現できないとき、あるいは、特定の処理系細部に依存しなければ、システムインターフェースやハードウェアの機能にアクセスできないときは、そうせざるを得ないからだ。

標準では、多くの重要事項が処理系定義(implementation-defined)となっている。すなわち、処理系が、動作を明確化するとともに、その動作を文章に明記しなければならない、ということだ。まず、次のコードを考えよう：

```
unsigend char c1 = 64;          // 的確な定義：char は少なくとも 8 ビットだから 64 は必ず表現可能
unsigned char c2 = 126;         // 処理系定義：char が 8 ビットであれば切捨てが行われる
```

char の大きさは 8 ビット以上でなければならないため、ここでの c1 の初期化は、的確に定義された(well defined)動作である。しかし、char が実際に何ビットなのかは処理系定義なので、c2 の初期化は、処理系定義となる。char の大きさが 8 ビットしかなければ、1256 という値は 232 に切り詰められる(§10.5.2.1)。処理系定義の機能は、プログラムを実行するハードウェアの違いと関連するものが多い。

もう一つ、指定されてない。(unspecified)という動作もある。許容する動作に幅をもたせたものだが、実装者が必ずその中から選択しなければならないわけではない。何かが指定されない、という場合、さまざまな根本的原因によって厳密な動作を正確には予測できないことが多い。たとえば、new が返す正確な値は指定されない。同様に、データ競合を防ぐための何らかの同期を用いることなく、同じ変数に 2 個のスレッドが代入を行うと、代入後の値は指定されない(§ 41.2)。

現実世界のプログラムを開発する際に、処理系定義の動作が必要となるのは、一般的なことだ。極めて幅広いシステムを効率的に操作できるようにするために、必要な犠牲である。もし仮に、すべての文字が 8 ビットであり、すべてのポインタが 32 ビットだったならば、C++はずいぶんと簡潔になっていたはずだ。しかし、16 ビットや 32 ビットの文字セットは珍しくないし、16 ビットや 64 ビットのポインタをもつマシンも広く利用されている。

可能性を最大限確保するには、利用しなければならない処理系定義の機能を明確にし、プログラム内にその旨を分かりやすく記述して、コード分割しておくとよい。ハードウェアによって決定される大きさに依存するものすべてを定数としておいた上で、型の定義とともにヘッダに記述するなどの対策がよく使われる。この技法を支援するために、標準ライブラリは、numeric_limits(§ 40.2)を提供する。また、処理系定義の機能を想定する場合、静的アサーションによってチェックできることが多い(§ 2.4.3.3)。たとえば :

```
static_assert(4<=sizeof(int), "sizeof(int) too small");
```

定義されない動作は、不愉快だ。定義されない(undefined)とは、標準が、理にかなった動作を処理系に対して要求しない、ということである。実装上のテクニックが、定義されない動作の原因となって、プログラムがひどい動作となることが多い。

```
const int size = 4 * 1024;
char page[size];

void f()
{
        page[size + size] = 7;                  // 定義されない
}
```

このコードは、無関係のデータを上書きして、ハードウェアエラーや例外を発せさせるというのが、理屈どおりの結果だ。必ず理屈どおりの結果を生み出すことは、処理系には求められていない。オプティマイザが強力であれば、定義されない動作は、完全に予測不能となるからだ。その機能に理屈どおりで容易に実装できる別の方法が複数あれば、その機能は、定義されないのではなくて、指定されない、あるいは、処理定義となる。

標準が指定しない、あるいは、定義しない、とするものを使わないように努力することは、そのための時間と労力に見合うはずだ。なお、その作業を支援するツールも数多く存在する。

6.1.1 処理系

C++の処理系は、依存処理系(hosted implementation)あるいは自立処理系(freestanding implementation)のいずれかである(§ iso.17.6.1.3)。依存処理系は、標準と本書で解説する標準ライブラリ機能をすべてもつ(§ 30.2)。一方、自立処理系は、すべてをもつとは限らないが、以下にあげる機能は必ず提供される :

自立処理系のヘッダ		
型	<cstddef>	§10.3.1
処理系の性質	<cfloat> <limits> <climits>	§40.2
整数型	<cstdint>	§43.7
実施開始と終了	<cstdlib>	§43.7
動的メモリ管理	<new>	§11.2.3
型識別	<typeinfo>	§22.5
例外処理	<exception>	§30.4.1.1
初期化子並び	<initializer_list>	§30.3.1
その他の実行時支援	<cstdalign> <cstdarg> <cstdbool>	§12.2.4, §44.3.4
型特性	<type-traits>	§35.4.1
アトミック	<atomic>	§41.3

自立処理系は、オペレーティングしてステムの最小限の支援だけでコードが動作することをいとしたものである。極めて小規模でハードウェアよりのプログラムのために、例外を使わないようにする(非標準)オプションを提供する処理系も多い。

6.1.2 基本ソース文字セット

C++標準と本書のサンプルコードは、基本ソース文字セット(basic source character set)を使って記述されている。基本ソース文字セットは、国際 7 ビット文字セット ISO 646-1983 の米国版である ASII (ANSI3.4-1968)と呼ばれるものであって、文字、数字、図形文字、空白類文字で構成される。そのため、文字セットが異なる環境で C++を利用すると、次のような問題が発生する可能性がある：

- ASII は、句読文字や演算子に利用する(]、{、！などの)記号を含むが、それを利用できない文字セットがある。
- 通常の表現法では表せない文字を記述する必要がある(たとえば、改行文字や、“値が 17 である文字”など)。
- ASII は、英語以外で利用する文字をもっていない(たとえば、ñ、ｐ、Æ など)。

ソースコード内で拡張文字セットを利用する場合、プログラミング環境は、たとえば国際文字名を使うなどの何らかのの方法によって、拡張文字セットを基本ソース文字セットへマッピングすることになる(§6.2.3.2)。

6.2 型

次の例を考えてみよう：

```
x = y + f(2);
```

これを C++プログラムとして意味あるものとするには、x、y、f の名前を適切に宣言しなければならない。すなわち、プログラマは、x、ｙ、f という名前の実態が存在する事、および、それらの型が= (代入)、+ (加算)、( ) (関数呼出し)の演算の対象となり得る型であることを、はっきりと指定する必要がある。

C++プログラム内のあらゆる名前(識別子)は、対応する型をもっている。型は、その名前(その名前が表す実体)に対して適用できる演算と、その演算の解釈法を決める。たとえば：

```
float x;          // x は浮動小数点変数
int y = 7;        // y は初期値 7 をもつ整数変数
float f(int);     //      f は int 型引数を受け取って浮動小数点を返却する関数
```

この宣言によって、最初に示したコードに対して初めて意味が与えられる。y が int と宣言されているので、代入可能であることや、+が適用可能であることが分かる。また、f も int の引数が受け取る関数と宣言されているので、整数 2 を与えた呼び出しが行える。

本章では、基本型(§6.2.1)と、宣言(§6.3)とを解説する。ここにあげるサンプルコードは、言語機能を示すためのものであって、何か有用な処理をおこなうわけではない。より本格的で現実的なサンプルコードは、後の章で示す。C++プログラムを構成する基本的要素の大部分を示すのが、本書の目的だ。C++で現実の作業をこなして、さらに他人が書いたコードを読むためには、構成の後の章は読み進められる。そのため、本章の主要な概念を眺めて大まかに理解しておき、詳細が必要になったら本書に戻ってくる。という進め方でも構わない。

6.2.1 基本型

C++には、一連の基本型(fundamental type)がある。これは、コンピュータの記憶単位としてもっとも一般的なものに対応するとともに、主としてデータの保持のために利用されるものだ：

§6.2.2　論理型（bool）
§6.2.3　文字型（char や wchar_t など）
§6.2.4　整数型（int や long long など）
§6.2.5　浮動小数点数型（double や long double など）
§6.2.7　void 型。情報をもたないことを表す

宣言演算子を用いると、これらの型から、以下の型を作ることができる：

§7.2　　ポインタ型（int*など）
§7.3　　配列型（char[]）
§7.7　　参照型（double&や vector<int>&&など）

また、以下の型も定義できる：

§8.2　　構造体、クラス(第 16 章)
§8.4　　特定の値の集まりを表現する列挙体(enum と enum class)

論理型、文字型、整数型の総称が、汎整数型(integral type)であり、汎整数型と浮動小数点数型の総称が、算術型(arithmetic type)である。列挙体とクラス(第 16 章)は、型の内容を事前に定義することなく利用できる基本型とは異なり、ユーザによる型の定義が必要なため、ユーザ定義型(user-defined type)と呼ばれる。また、基本型、ポインタ型、参照型の総称が、組込み型(built-in type)である。標準ライブラリは、数多くのユーザ定義型が定義されている。(第 4 章、第 5 章)。

汎整数型と浮動小数点数型の大きさには、いくつかの種類がある。そのため、演算に必要な記憶容量、演算精度、値の範囲などから、プログラマが型を選択できる(§6.2.8)。なお、その前提は、文字を格納するためのバイト、整数値を格納・演算するためのワード、浮動小数点数演算に最適な実態、これらの実体を参照するのに必要なアドレスを、コンピュータが提供することだ。C++の基本の型とポインタと配列は、これらのマシンレベルの概念を、処理系に依存することのない、理にかなった形でプログラマに提供する。

ほとんどのアプリケーションでは、論理値に bool を、文字に char を、整数値に int を、浮動小数点数値に double を用いればすむ。これら以外の基本型は、最適化、特殊用途、互換性のためのものであって、使う必要がなければ、無視しても構わない。

6.2.2 論理型

論理型 bool は、true と false のいずれか一方の値をもつ型だ。論理型は、論理演算の結果を表すために使われる。たとえば：

```
void f(int a, int b)
{
        bool b1 {a == b};
        // ...
}
```

ここで、a と b が同じ値であれば、b1 は true になり、そうでなければ false になる。bool が良く使われるのは、何らかの条件をテストする関数(述語)の結果の型を表すときだ。たとえば：

```
bool is_open(File *);

bool greater(int a, int b) { return a > b; }
```

定義によって、true を整数に変換すると 1 になって、false を変換すると 0 になる。逆に、整数値は暗黙裏に bool へと変換できる。その際、非ゼロの整数は true に変換されて、0 は false に変換される。たとえば：

```
bool b1 = 7;            // 7 != 0 なので b1 は true になる
bool b2 {7};            // エラー：縮小変換(§2.2.2、§10.5)

int i1 = ture;          // i1 は 1 になる
int i2 = {true};        // i2 は 1 になる
```

情報の縮小変換を防ぐために、{}構文による初期化を行いたいのだが、int から bool への変換も行いたい、という場合は、次のように明示的に行う：

```
void f(int i)
{
        bool b {i != 0};
        // ...
}
```

算術式やビット単位の論理式内では、bool や int に変換される。変換された値に対しては、算術演算や論理演算が、整数として実行される。演算結果を bool に戻す必要があれば、0 は false に変換されて、非ゼロは true に変換される。たとえば：

```
bool a = true;
bool b = ture;

bool x = a + b;         // a + b は 2 なので x は true になる
bool y = a || b;   // a || b は true なので y は true になる("||"は"または"を意味する)
bool z = a - b;         // a - b は 0 なので z は false になる
```

ポインタは暗黙裏に bool に変換できる(§10.5.2.5)。空でないポインタは true へと変換される。値が nullptr であるポインタは false に変換される。たとえば：

```
void g(int* p)
{
        bool b = p;                         // true または false に縮小変換される
        bool b2 {p != nullptr};             // nullptr でないかどうかの明示的な判定

        if (p) {
                // ...                      // p != nullptr と等価
        }
}
```

私は、if(p != nullptr)よりも if(p)を好んで利用する。というのも、“p が有用ならば”を直接的に表現できる上に、記述が短くなるからだ。記述が短いほうが、エラーにつながりにくい。

6.2.3 文字型

文字セットとそのエンコーディングには、数多くのものが利用されている。C++でも、それらの型を反映できるように、困惑するほど豊富な文字型を提供している：

- char：プログラムテキストでりようする。デフォルトの文字型である。char は、処理系の文字セットにも利用されており、通常は 8 ビットである。
- signed char：char に似た型だが、符号が保証される。すなわち、正負いずれの値も保持できる。
- unsigned char：char に似た型だが、符号無しであることが保証される。
- wchar_t：Unicode などの大規模文字セットの文字を保持する型である(§7.3.2.2)。この型の大きさは処理系定義であり、処理系のロケールで利用可能な文字セットのうち最大のものを保持できる大きさだ(第 39 章)
- char16_t：UTF-16 など、16 ビット文字セットの文字を保持する型である。
- char32_t：UTF-32 など、32 ビット文字セットの文字を保持する型である。

これら 6 個は(_t で終わる名前は、別名であることが多いという事実にもかかわらず：§6.5)、いずれも個別の型である。すべての処理系で、char は、signed char と unsigned char のいずれか一方と同一となる。ただし、それら 3 個の型は、それぞれが別々の型とみなされる。

char 型の変数は、処理系の文字セット内の文字を保持できる。たとえば：

```
char ch = 'a';
```

char の大きさは、ほとんどの場合 8 ビットなので、256 種類のうち 1 個の値をもつことになる。文字セットは、ISO-646 の変種、たとえば ASCII であることが多いので、キーボード上の文字は表現可能だ。しかし、この文字セットは、一部分しか標準化されていないため、多くの問題が発生している。

異なる自然言語をサポートする複数の文字セット間には大きな違いがあるし、同じ自然言語を異なる方式でサポートする文字セット間にも大きな違いがある。そのため、本書では、C++の規則にどのような影響を与えるかといった点だけに注目する。複数言語や複数文字セット環境下のプログラミングという、広範囲で興味深い問題は、そのたびに(§6.2.3、§36.2.1、第 39 章)言及するものの、基本的には本書では扱わない。

処理系の文字セットに、10 進表記の数字と、26 個の英語のアルファベットと、基本的な句読文字が含まれることを想定しても問題ない。しかし、以下の点を想定することは安全ではない：

- 8 ビットの文字セット内に 127 個よりも多い文字が含まれないこと(たとえば、255 個の文字をもつ文字セットも存在する)
- 英語以外のアルファベット文字が含まれないこと(多くのヨーロッパの言語は、æ、þ、β などの文字をもっている)。

・アルファベット文字が連続していること(EBCDIC では’i’と‘j’は連続していない)。
・C++の記述で使うすべての文字が利用できること(一部の言語の文字セットは、{、}、[、]、|、\ もっていない)。
・char が、一般的な 8 ビット 1 バイトに収まること。char が 32 ビット(一般的には 4 バイト)であって、バイト単位にアクセスするハードウェアをもたない組込み用プロセッサも存在する。また、char を 16 ビット Unicode エンコーディングとする環境もある。

可能な限り、オブジェクトの表現に対して何らかの仮定を設けるべきではない。この一般則は、文字についても適用される。

すべての文字は、処理系が利用する文字セットにおける整数値をもつ。たとえば ASCII 文字セットの’b’の値は 98 だ。次に示すのは、入力された文字の整数値を出力するループである:

```
void inval()
{
        for (char c; cin >> c; )
        cout << "the value of '" << "'is " << int{c} << '¥n';
}
```

ここでの int{c}という表記は、文字 c の整数値(“c から構築できる int”)を返す。char から整数への変換にあたって、ある疑問が浮かぶ。char は、符号付なのだろうか、それとも符号なしなのだろうか?8 ビット 1 バイトが表現できる 256 種類の値は、0 から 255 まで、あるいは、-127 から 127 と解釈される。-128 から 127 と考えたかもしれないが、それは違う。C++標準では、ハードウェアが 1 の補数で動作する可能性を配慮しているので、表現できる値は 1 個少なくなるのだ。そのため、-128 を使うと可搬性が失われる。残念ながら、単なる char を、符号付と符号無しのどちらとするかは処理系定義である。その代わりに、C++では 2 種類の文字型を提供する。少なくとも-127 から 127 の範囲を表現する singned char 型と、少なくとも 0 から 255 を表現する unsigned char 型だ。幸いなことに、一般的な文字の値は 0 から 127 の範囲に収まっているので、その範囲外の文字だけが問題となる。

範囲外の値を単なる char に代入するコードは、移植の際に、微妙で厄介な問題につながる。複数種類で char を利用する場合や、char 型の変数に整数を代入する場合について、§6.2.3.1 を参照しょう。

文字型が汎整数型であることに注意が必要だ(§6.2.1)。そのため、文字型には、算術演算とビット単位の論理演算が適用できる(§10.3)。たとえば:

```
void digits()
{
        for (int i = 0; i != 10; ++i)
                cout << static_cast<char>('0' + i);
}
```

これは、0 から 9 までの 10 個の数字を cout に出力する。文字リテラルの’0’は整数値に変換され、それに i が加算される。加算結果の int は char に変換された上で cout に出力される。単なる’0’+i は int なので、static_cast<char>を省略すると、出力は 48、49・・・となってしまい、0、1・・・とはならない。

6.2.3.1 符号付き文字と符号無し文字

単なる char が、符号付き型と符号無し型のいずれとみなされるのかは処理系定義である。そのため、驚かされることがあるだけでなく、処理系依存の問題が浮上することもある。たとえば:

```
char c = 256;       // 255 は“全ビット 1”すなわち 16 進の 0xFF
int i = c;
```

ここで i の値はいくつになるだろうか？残念ながらその答えは、定義されない。8 ビット 1 バイトの処理系での int への拡張では、“全ビットが 1”の char の意味に依存するのだ。char が符号無しの環境では、答えは 255 である。しかし、符号付きであれば、-1 である。その場合、整数リテラル 255 を、char 型の-1 に変換したことを、コンパイラが警告するかもしれない。しかし、C++は、この問題を検出する一般的な機構をもたない。単なる char は利用せず、符号付き型と符号無し型のいずれかの char だけを利用するというのも一つの解決法だ。ただし、strcmp()のような標準ライブラリ関数は、単なる char のみを引数として受け取る(§43.4)。

単なる char は、必ず signed char と unsigned char の一方と同じように振る舞わなければならない。しかし、3 種類の char は別々のものの中で、ポインタでの混用は不可能だ。たとえば：

```
void f(char c, singned char sc, unsigned char uc)
{
        char* pc = &uc;                 // エラー：ポインタの変換は行われない
        singed char* psc = pc;          // エラー：ポインタの変換は行われない
        unsigned char* puc = pc;        // エラー：ポインタの変換は行われない
        psc = puc;                      // エラー：ポインタの変換は行われない
}
```

3 種類の char 型の変数は相互に代入できる。ただし、signed char に対して範囲を越える値を代入した結果(§10.5.2.1)は、処理系定義だ。たとえば：

```
void g(char c, singed char sc, unsigned char uc)
{
        c = 255;        // 単なる char が符号付きで 8 ビットであれば処理系定義
        c = sc;         // OK
        c = uc;         // 単なる char 符号付きで uc が大きすぎる値であれば処理定義
        sc = uc;        // uc が大きすぎる値であれば処理系定義
        uc = sc;        // OK：符号無しへの変換
        sc = c;         // 単なる char が符号付きで c が大きすぎる値であれば処理系定義
        uc = c;         // OK：符号無しへの変換
}
```

具体的には、char は 8 ビットと仮定すべきだ：

```
signed char sc = -140;
unsigned char uc = sc;      // uc == 116 (256-140==116 だから)
cout << uc;                 // 't'を表示

char count[256];            // char が 8 ビットと仮定(要素は初期化されない)
char c1 = count[sc];        // 災害の可能性：範囲外のアクセス
char c2 = count[uc];        // OK
```

全体を通じて単なる char を利用した上で、負数にならないようにすれば、上記の問題や混乱が避けられる。

6.2.3.2 文字リテラル

文字リテラル(charcter literal)は、'a'や'0'のように、単一引用符記号で囲んだ単一の文字のことだ。その型は、char である。文字リテラルは、C++プログラムを実行するマシンの文字セットでの整数値へと暗黙裏に変換される。たとえば、ASCII 文字セットのマシンでは、'0'の値は 48 である。このような 10 進表記よりも、文字リテラルのほうがプログラムの可読性が向上する。

いくつかの文字は、エスケープ文字として逆斜線 \ を利用する標準名をもつ：

名前	ASCII 名	C++名
改行	NL(LF)	\n
水平タブ	HT	\t
垂直タブ	VT	\v
後退	BS	\b
復帰	CR	\r
改頁	FF	\f
警報	BEL	\a
逆斜線	\	\\
疑問符	?	\?
単一引用符	'	\’
二重引用符	″	\”
8 進数	ooo	\ooo
16 進数	hhh	\hhh…

見かけとは違い、これらはすべて、単一の文字を表す。

処理系の文字セット内の文字を記述する際は、1 桁～3 桁の 8 進数(先頭に\を付ける)、あるいは 16 進数(先頭に\x を付ける)としても表現できる。16 進数では、桁の制限はない。8 進数では、8 進数字ではない文字の直前までが 8 進数とみなされる。同様に、16 進数では、16 進数字でない文字の直前までが 16 進とみなされる。たとえば：

8 進数	16 進数	10 進数	ASCII
‘\6‘	‘\x6‘	6	ACK
‘\60‘	‘\x30‘	48	‘0‘
‘\137‘	‘\x05f‘	95	‘_‘

この記述では、マシンの文字セットがもつすべての文字が表現できるし、文字列の中にも埋め込めるようになる(§7.3.2)。しかし、このような数値表記は、プログラムの可搬性を損なう。

なお’ab’のように、2 文字以上を文字リテラルとすることもできる。しかし、古い時代の記法であって、処理系依存となるため、極力避けるべきだ。なお、このような複数文字リテラルは、int とみなされる。

文字列内で 8 進記法を用いる場合は、常に 3 桁としておくとよい。そうすると、8 進記法の直後の文字が、数字であるかどうかを迷わなくてすむ。同様に、16 進数の場合は常に 2 桁とする。たとえば：

```
char v1[] = "a¥xah¥129";          // 6 個の文字：'a' '¥xa' 'h'   '¥12' '9' '¥0'
char v2[] = "a¥xah¥127";          // 5 個の文字：'a' '¥xa' 'h'   '¥127' '¥0'
char v3[] = "a¥xad¥127";          // 4 個の文字：'a' '¥xad' 'h' '¥127' '¥0'
char v4[] = "a¥xad¥0127";         // 5 個の文字：'a' '¥xad' '¥012' '7'   '¥0'
```

ワイド文字リテラルは、L’ab’という形式であり、その型は、wchar_t だ。単一引用符記号内の文字数とその意味は、処理系定義である。

C++プログラムでは、Unicode などのような、127 文字の ASCII 文字セットよりも多くの文字をもつ文字セットも利用できる。その場合、先頭に u が付いた 4 桁の 16 進数、または先頭に u が付いた 8 桁の 16 進数を用いてリテラルを表現する。たとえば：

```
U'¥UFADEBEEF'
u'¥uDEAD'
u'¥xDEAD'
```

X を任意の 16 進数 1 桁としたとき、u' \uXXXX'という短い表記は、U' \U0000XXXX'と等価である。4 桁または 8 桁以外の 16 進数は、字句上のエラーとなる。16 進数の意味を定義するのは ISO/IEC 10646 であり、その数値は国際文字名(universal character name)と呼ばれる。C++標準では、§iso.2.2、§iso.2.3、§iso2.14.3、§iso2.14.5、§iso.E で解説している。

6.2.4 整数型

char と同様に、整数型にも"単なる"int、signed int、unsinged int の 3 種類がある。それに加えて、大きさとして、short int、"単なる"int、 long int、long long int の 4 種類がある。long int は long と表記できるし、long long int は long long と表記できる。同様に、short は short int と同様で、unsigned は unsigned int と同義で、signed は signed int と同義である。ただし、long short int とすれば int と同じなる、ということはない。

メモリ領域をビット配列として扱うのであれば、unsigned の整数型が理想的だ。しかし、大きな正の数を表現するための 1 ビットをかせぐために unsigned を利用する、というのは、良い考えではない。unsigned と宣言することで、値の正の数だけにしようとしても、暗黙の変換規則(§10.5.1, §10.5.2.1)によって打ち砕かれることが多い。

整数型の大きさを細かく制御する必要がある場合は、<cstdint>が定義する int64_t(64 ビットの符号付き整数。存在する場合のみ定義される)、uint_fast16_t(少なくとも 16 ビットの符号無し整数。もっとも速く処理できるはずのもの)、int_least32_t(少なくとも 32 ビットの符号付き整数。多くの場合、単なる long と同じビット数である)などの別名が利用できる。(§43.7)。組込み型の整数は、的確に定義された最小の大きさをもっている(§6.2.8)。そのため、<cstdint>は、冗長になりやすいだけでなく、必要以上に利用されることがある。

標準の整数型に加えて、(符号付きと符号無しの)拡張整数型(extended integer type)を処理系が提供することがある。それらの型は、必ず整数として振る舞い、変換時や整数リテラルとしての値としても整数とみなされるが、通常は、大きいが大きい(より多くのメモリを消費する)。

6.2.4.1 整数リテラル

整数リテラルは、表記上、10 進数、8 進数、16 進数の 3 種類がある。もっとも多く利用されてるのは 10 進数リテラルであり、まさに読者が期待する通りのものだ：

```
7   1234   976 12345678901234567890
```

表現できないほど長いリテラルは、コンパイラが警告すべきだ。ただし、確実にエラーになるのは、{}構文の初期化子に対してだけだ(§6.3.5)。

ゼロで始まって、直後に x または X が続く(0x、0X)リテラルは、16 進数(基数が 16)である。ゼロで始まって、直後に x または X が続かないものは 8 進数(基数が 8)である。たとえば：

10 進数	8 進数	16 進数
	0	0x0
2	02	0x2
63	077	0x3f
83	0123	0x53

そう、単なる 0 は、10 進数ではなく、8 進数だ。

16 進数の文字の a、b、c、d、e、f は、小文字でも大文字でも、それぞれ 10、11、12、13、14、15 を表す。ビットパターンを表現する際は、8 進数や 16 進数表現はもっとも都合がよいものだ。しかし、そうでな

い、いわゆる数値を表現する場合に使うと、混乱しやすくなる。たとえば、int を 16 ビットの 2 の補数で表現するマシン上で 0xffff と記述すれば、負の 10 進数-1 を意味するが、int のビット数がもっと大きければ、正の 10 進数 65535 になってしまう。

リテラルに接尾語 U を付加すると、明示的な unsigned の指定となる。同様に L を付加すると long の指定だ。たとえば、3 は int、3U は unsigned int、3L は long int である。

複数の接尾語を組み合わせることも可能だ。たとえば：

```
cout << 0xF0UL << ' ' << 0LU << '¥0';
```

接尾語をまったく付加しない場合は、リテラルの値と、その処理系での整数型の大きさとから、コンパイラが適切な型を決定する(§6.2.4.2)。

意図が分かりにくい定数を使うときは、若干のコメントを加えた const(§7.5)、constexpr(§10.4)、列挙子(§8.4)の初期化子だけにしておくとよいだろう。

6.2.4.2 整数リテラルの型

一般的に整数リテラルの型は、その形式と、値と、接尾語とに基づいて判断される：

- 接尾語をもたない 10 進数の型は、次の並びにおいて、その値を表現できる先頭のものとなる：int、long int、long long int。
- 接尾語をもたない 8 進数あるいは 16 進数の型は、次の並びにおいて、その値を表現できる先頭のものとなる：int、unsigned int、long int、unsigned long int、long long int、unsigned long long int。
- u または U の接尾語をもつ場合の型は、次の並びにおいて、その値を表現できる先頭のものとなる：int、unsigned long long int。
- l または L の接尾語をもつ 10 進数の型は、次の並びにおいて、その値を表現できる先頭のものとなる：long int、long long int。
- l または L の接続語をもつ 8 進数または 16 進数の型は、次の並びにおいて、その値を表現できる先頭のもとなる：long int、unsigned int、long long int、long long int、unsigned long long int。
- ul、lu、uL、Lu、Ul、lU、UL、LU の接続語をもつ場合の型は、次の並びにおいて、その値を表現できる先頭のものとなる：unsingned long int、unsigned long long int。
- ll または LL の接尾語をもつ 10 進数の型は、long long int である。
- ll または LL の接尾語をもつ 8 進数あるいは 16 進数の型は、次の並びにおいて、その値を表現できる先頭のものとなる：long long int、unsigned long long int。
- llu、llU、ull、Ull、LLu、LLU、uLL、ULL の接続語をもつ場合の型は、unsigned long long int である。

たとえば、100000 は、int が 32 ビットのマシンでは int だが、int が 16 ビットで long が 32 ビットのマシンでは long int となる。同様に 0XA000 は、int が 32 ビットのマシンでは int だが、int が 16 ビットでは unsigned int となる。このような処理系依存は、接尾語を用いることで回避可能だ。たとえば、1000000L とすればすべてのマシン上で long int となり、同様に 0XA000U はすべてのマシン上で unsigned int である。

6.2.5 浮動小数点数型

浮動小数点数型は、浮動小数点値を表現する。浮動小数点数値は、固定サイズのメモリサイズで、実数値を近似して表現したものだ。浮動小数点数型には、float(単精度)、double(倍精度)、long double(拡張倍精度)の 3 種類がある。

単精度、倍精度、拡張倍精度という言葉の意味するところは、処理系定義だ。

いろいろな状況に応じて、適切な精度を選択できるようになるには、まずは浮動小数点数演算を理解する必要がある。浮動小数点数演算を知らない読者は、アドバイスをもらうか、時間をかけて理解する必要がある。それができないのであれば、うまくいくように願いながら double を利用するといいだろう。

6.2.5.1 浮動小数点数リテラル

浮動小数点数リテラルの型は、デフォルトで double となる。先ほども述べたように、型が表現できないほど大きな値のリテラルに対して、コンパイラが警告すべきだ。浮動小数点数リテラルの例を示そう：

```
1.23    .23    0.23    1.    1.0    1.2e10    1.23e-15
```

浮動小数点数リテラルでなく、次に示す 4 個の構文トークンとみなされる(文法エラーだ)：

```
65.43 e - 21
```

浮動小数点数リテラルの型を float にしたければ、接尾語 f または F を付ける：

```
3.14159265f  2.0f  2.997592F  2.9e-3f
```

浮動小数点数リテラルの型を long double にしたければ、接尾語 l または L を付ける：

```
3.14159265L  2.0L  2.997925L  2.9e-3L
```

6.2.6 接頭語と接尾語

リテラルの接頭語と接尾語

記法		位置	意味	例	参照	ISO
0		前	8 進数	0776	§6.2.4.1	§iso.2.14.2
0x	0X	前	16 進数	0xff	§6.2.4.1	§iso.2.14.2
u	U	後	unsigned	10U	§6.2.4.1	§iso.2.14.2
l	L	後	long	20000L	§6.2.4.1	§iso.2.14.2
ll	LL	後	long long	20000LL	§6.2.4.1	§iso.2.14.2
f	F	後	float	10.3f	§6.2.5.1	§iso.2.14.4
e	E	中	浮動小数点数	10e-4	§6.2.5.1	§iso.2.14.4
.		前	浮動小数点数	12.3	§6.2.5.1	§iso.2.14.4
’		前	char	’c’	§6.2.3.2	§iso.2.14.3
u’		前	char16_t	u’c’	§6.2.3.2	§iso.2.14.3
U’		前	char32_t	U’c’	§6.2.3.2	§iso.2.14.3
L’		前	wchar_t	L’c’	§6.2.3.2	§iso.2.14.3
"		前	文字列	"mess"	§7.3.2	§iso.2.14.5
R″		前	原文字列	R"(¥b) "	§7.3.2.1	§iso.2.14.5
u8″	u8R″	前	UTF-8 文字列	u8"foo"	§7.3.2.1	§iso.2.14.5
u″	uR″	前	UTF-16 文字列	u"foo"	§7.3.2.2	§iso.2.14.5
U″	uR″	前	UTF-32 文字列	U"foo"	§7.3.2.2	§iso.2.14.5
L″	LR″	前	wchar_t 文字列	L"foo"	§7.3.2.2	§iso.2.14.5

なお、ここでの”文字列”は、”文字列リテラル”(§7.3.2)のことであって、“型が std::string” という意味ではない。

当然、.と e は接中語ともみなせるし、R”と u8”は区切り文字の最初の部分とみなせる。しかし、ここでは

学術的な用語体系よりも、混乱するほど豊富なリテラルの全体を示すことを優先した。

接尾語の l、L を、u、U と組み合わせて、unsigned long 型を表すことができる：

```
1LU      // unsigned long
2UL      // unsigned long
3ULL     // unsigned long
4LLU     // unsinged long long
5LUL     // エラー
```

接尾後の l、L は浮動小数点リテラルにも利用でき、その場合は long double 型を表す：

```
1L       // long int
1.0L     // long double
```

接頭語 R、L、u の組合せも可能であり、たとえば uR"**(foo¥(bar))** "のように利用できる。接尾語の U は整数に利用するもの(unsigned)であり、接頭語の U は文字または文字列に利用するもの(UTF-32 エンコーディング：§7.3.2.2)であることに注意しよう。

これだけでなく、ユーザ定義用に、ユーザが接尾語を新しく定義することもできる。たとえば、ユーザ定義リテラル演算子(§19.2.6)を定義すると、次のような記述が行える：

```
"foo bar"s     // std::string 型のリテラル
123_km       // Distance 型のリテラル
```

_以外の文字で始まるユーザ定義接尾語は、標準ライブラリが予約ずみだ。

6.2.7 void

void 型は、文法的には基本型である。しかし、利用できる箇所は、より複雑な型の一部に限られており、void 型のオブジェクトというものは存在しない。返却値を返さない関数の型、あるいは、型が分からないオブジェクトというものは存在しない。返却値を返さない関数の型、あるいは型が分からないオブジェクトを指すポインタの型として利用する。たとえば：

```
void x;          // エラー：void オブジェクトというものはない
void& r;         // エラー：void への参照というものもない
void f()         // 関数 f は値を返却しない(§12.1.4)
void* pv()       // 型が分からないオブジェクトへのポインタ(§7.2.1)
```

関数を宣言する際は、返却値の型の指定が必要だ。論理的には、返却値型を省略するれば、何の値も返さないことを意味すると考えられるが、そうすると見苦しい文法になってしまう(§iso.A)。そのため、関数が値を返さないことを表す void を、“疑似戻り型”として利用する。

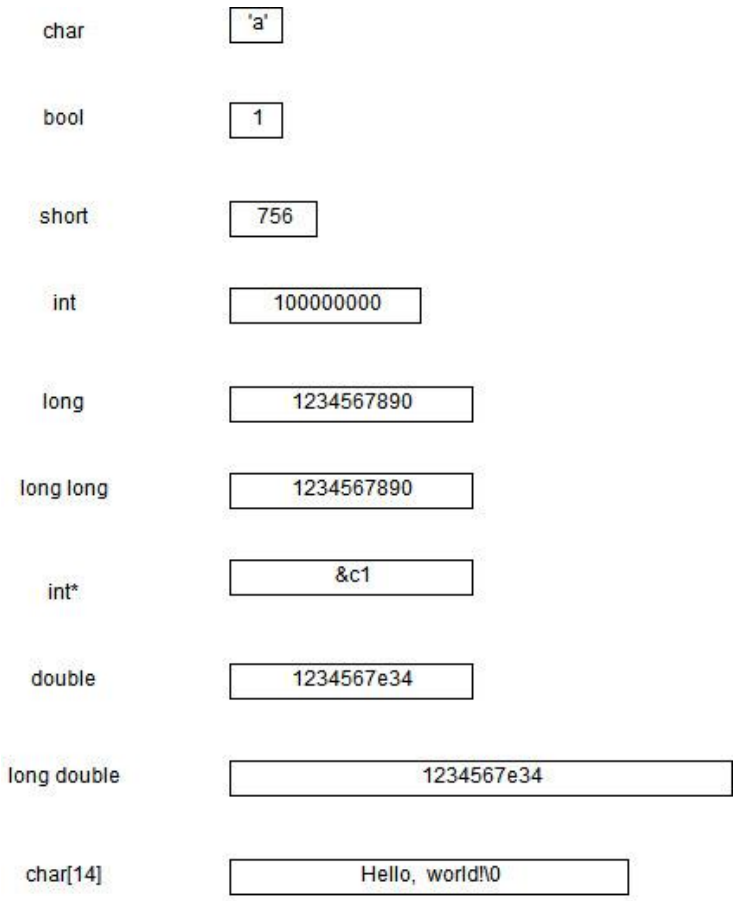
6.2.8 大きさ

int の大きさなど、C++の基本型のいくつかの側面は処理系定義である(§6.1)。私は、これらの依存症について指摘するとともに、それらを避けるべきだあり、避けられなければ影響を最小に抑えるべきだと、勧めることが多い。どうしてだろうか？さまざまなシステムで開発するプログラマや、さまざまなコンパイラを利用するプログラマは、多くの点に注意を払う。そうしないと、つかみどころのないバグの発見と修正に大きな労力を浪費せざるを得なくなるからだ。可搬性を気にしないと主張して、実際にそうしている人がいる。ただし、彼らは単一のシステムだけを利用して、“自分が使っているコンパイラが実装するものこそが、プログラミング言語だ”という考えをもっている。これは視野が狭くて浅はかである。もし開発したプログラムが成功すれば、移植されることになる。そうすると、処理系依存の機能に関する問題を、誰かが見つけて修正しなければならなくなる。また、同じシステムを異なるコンパイラでコンパイルしなければならないことは、よくあることだし、お気に入りのコンパイラが、将来のリリースでは現在とは異なる実装となる可能性だってある。混乱した問題を後で解くよりも、プログラムを書く時点で、処理系依存の影響を把握して抑えるほうが、ずっと簡単だ。

処理系依存の影響を抑えるのは、比較的容易である。システムに依存する非標準ライブラリの影響を抑えるほうが、はるかに難しい。標準ライブラリ機能を可能な限り利用しておくことが、一つの対策である。

複数の整数型、符号無しの整数型、浮動小数点数型を提供しているのは、ハードウェアの特性をプログラマが活用できるようにするためだ。型が消費するメモリ、メモリアクセス時間などはマシンによる違いが大きい。異なる基本型を使う場合の演算速度も同様だ。マシンに関する知識があれば、ある変数に対して適切な整数型を選択するのは、それほど難しくない。ところが、本当に可搬性の高い、低レベルのコードを書くのは困難である。

基本型と文字リテラル(§7.3.2)を図で表したものを示す：



もし、(1 バイトを 0.2 インチ幅とする)縮尺どおりに、1 メガバイトのメモリを図にすると、およそ 3 マイル(5km)も右に延ばすことになる。

C++でのオブジェクトの大きさは、char の大きさの整数倍となるので、char の大きさは 1 と定義されている。オブジェクトや型の大きさは、sizeiof 演算子(§10.3)によって取り出せる。なお、基本型の型については以下のことが保証される：

- 1≡sizeof(char)≦sizeof(short)≦sizeof(int)≦sizeof(long)≦sizeof(long long)
- 1≦sizeof(bool)≦sizeof(long)
- sizeof(char)≦sizeof(wchar_t)≦sizeof(long)
- sizeof(float)≦sizeof(double)≦sizeof(long double)
- sizeof(N)＝sizeof(signed N)＝sizeof(unsigned N)

最終行の N は char、short、int、long、long long のいずれかである。さらに、char の大きさが 8 ビット以上であること、short が 16 ビット以上であること、long が 32 ビット以上であることも保証されている。char は、マシンの文字セット内の 1 文字を保持できる大きさだ。マシン上で文字を操作して保持するのに、もっとも適切な大きさとなる。通常は 4 バイト(32 ビット)のワードである。これ以上のこと、たとえば 32 ビットを char 型とするマシンなどを想定するのは賢明ではない。int とポインタの大きさが同一であるという想定は、もっと悪い。ポインタの大きさが整数よりも大きいマシン(“64 ビットアーキテクチャ”)は、多数ある。sizeof(long) < sizeof(long long)や sizeof(double) < sizeof(long double)が保証されないことにも注意しよう。
基本型の処理系定義の部分は、sizeof を用いるだけで確認できるし、<limits>でも関連した事項が定義されている。たとえば：

```
#include <limits>               // §40.2
#include <iostream>

int main()
{
        cout << "size of long " << sizeof(1L) << '¥n';
        cout << "size of long long " << sizeof(1LL) << '¥n';

        cout << "largest float == " << std::numeric_limits<float>::max() << '¥n';
        cout << "char is signed == " << std::numeric_limits<char>::is_singed << '¥n';
}
```

<limits>(§40.2)が定義する関数は constexpr(§10.4)なので、実行時オーバヘッドなしに利用できるし、定数式が必要な場面でも利用できる。

基本型は、代入や式の中で、自由に組み合わせられる。可能な場合には、情報を失うことなく値が変換される(§10.5)。

たとえば、値 v が、T 型の変数として表現可能であれば、v を T に変換する際に必ず値が保持される。逆に、値が維持されない類の変換は、最大限避けるべきだ(§2.2.2、§10.5.2.6)。

特定の大きさの整数、たとえば 16 ビットの整数が必要であれば、各種の型(あるいは型別名：§6.5)を定義した標準ヘッダ<cstdint>を#include した上で利用する。たとえば：

```
int16_t  {0xaabb};                    // 2 バイト
int64_t xxx {0xaaaabbbbccccdddd};     // 8 バイト
int_least16_t y;                      // 少なくとも 2 バイト(ちょうど int と同様)
int_least32_t yy;                     // 少なくとも 4 バイト(ちょうど long と同様)
int_fast32_t z;                       // 少なくとも 4 バイトをもつ最速の計算可能な整数型
```

標準ヘッダ<cstddef>は、標準ライブラリ内の宣言と、ユーザコードの、どちらにも広く利用される別名を定義する。たとえば、size_t は、処理系定義の符号無し整数型だ。すべての種類のオブジェクトのバイト単位の大きさを保持するものであり、オブジェクトの大きさを保持する場面で利用される。たとえば：

```
void* allocate(size_t n);             バイトを取得
```

<cstddef>では、符号付整数型 ptrdiff_t も定義してる。これは、要素数を求めるための、ポインタどうしの差の演算結果を保持できるものだ。

6.2.9 アライメント

オブジェクトは、メモリ上で内部データ表現を保持さえすればいい、というものではない。マシンアーキテクチャによっては、ハードウェアが効率よくアクセスできるように(極端な場合では一度にすべてアクセスできように)、値を保持するバイトを適切にアライメント(alignment)する必要がある。たとえば、4 バイトの int は(4 バイトの)ワード境界にアライメントしなければならないことが多く、また 8 バイトの double は(8 バイトの)ワード境界にアライメントしなければならないこともある。もちろん、これは極めて処理系特有の話であって、ほとんどのプログラマは完全に無視して構わない。明示的なアライメントが必要な場面に出会うことなく、数十年間もよい C++コードを書き続けることもできる。アライメントを意識しなければならない場面の多くは、オブジェクトの配置に関するものだ。たとえば、アライメント補正のために、struct 内に“隙間”が設けられることがある(§8.2.1)。

alignof()演算子は、引数のアライメントを返す。たとえば：

```
auto ac = alignof(char);              // char のアライメント
auto ai = alignof(int);               // int のアライメント
auto ad = alignof(T);                 //  何らかの型 T のアライメント

int a[20];
auto aa = alignof(decltype(a));       // int のアライメント
```

宣言の中で、式としてアライメントが必要になることもある。ところが、alignof(x+y)のような式は利用できない。その場合は、型指定子 alignas を用いる。alignas(T)は“T とはまったく同じようにアライメントせよ”という意味だ。たとえば、次の例では、未初期化のメモリ領域が、X 型として利用できるようになる：

```
void user(const vector<X>& vx)
{
        constexpr int bufmax = 1024;
        alignas(X) char buffer[bufmax];                 // 初期化されない

        const int max = min(vx.size(), bufmax / sizeof(X));
        uninitalized_copy(vx.begin(), vx.begin() + max, reinterpret_cast<X*>(buffer));
        // …
}
```

6.3 宣言

C++のプログラムでは、名前(識別子)は、利用する前に宣言しておかなければならない。すなわち、名前が表す実体をコンパイラに伝えるために、型を指定するのだ。たとえば：

```
char ch;
string s;
auto count = 1;
extern int error_number;

const char* name = "Njal";
const char* season[] = { "spring", "summer", "fall", "winter" };
vector<string> people { name, "Skarphedin", "Gunnar" };
struct Data { int d, m, y };
int day(Data* p) { return p->d; }
double sqrt(double);
template<typename T>T abs(T a) { return (n < 2)? 1 : n * fac(n - 1); }      // コンパイル時評価が可能
                                                                                (§2.2.3)

constexpr double zz { ii * fac(7) };                // コンパイル時初期化

using Cmplex = std::complex<double>                                  // 型別名（§3.4.5、§6.5)
struct User;                                                                 // 型名
enum class Beer { Carlsberg, Tuborg, Thor };
namespace NS { int a; }
```

この例からも分かるように、宣言は、型と名前との単なる関連付け以上のことを行える。ここに示した宣言(declaration)の多くは、定義(definition)でもある。定義とは、その実体の利用にあたってプログラムが必要とする情報をすべて含む宣言のことだ。特に何かを表すのにメモリが必要な場合は、定義によってメモリを割り当てる。インターフェース部を宣言として、実装部を定義とする用語体系がある。宣言からインターフェースを抜き出して別ファイル内に置くこと(§15.2.2)もあるので、その用語体系の観点では、メモリを設定する定義をインターフェースに入れることはできない。

さて、これらの宣言が広域スコープ(§6.3.4)にあれば、次のようになる：

用語集

・イテレータ・・・イテレータ(iterator)とはコンテナの要素へのアクセスを抽象化したもの
・STL・・・Standard Template Library (**STL**) は、プログラミング言語 C++の規格で定義された標準ライブラリの一つ。
・コンテナ・・・オブジェクトの集まりを表現するデータ構造、クラス
・Vector・・・C++ライブラリ で提供されるテンプレートを利用した動的な配列
・コンストラクタ・・・オブジェクト指向のプログラミング言語で新たなオブジェクトを生成する際に呼び出されて内容の初期化などを行なう関数、メソッドのことである。
・セマンティクス・・・プログラミング言語において、ソースコード中で利用されている変数や文が正しく動作するかを判断する基準のこと。たとえば、ヌルが定数として定義されている値がアドレス参照のために使われていると、セマンティクスに違反しているとみなして、エラーと判断する。
・スループット・・・一般に単位時間当たりの処理能力のこと。一定時間内に処理できるデータ量のこと。