

A S C I I A d d i s o n W e s l e y P r o g r a m m i n g S e r i e s

◆ Effective C++ Second Edition 50 Specific Ways to Improve Your Programs and Designs

Effective C++

改訂2版

Scott Meyers 著 吉川邦夫 訳

アスキー出版局

序章

プログラミング言語の基礎を学習するのと、その言語で**効果的に**プログラムを設計し実装する方法を学習するのは、まったく別の話だ。C++という、パワーと表現力の守備範囲が非常に広いことを特徴とする言語では、その違いはとくに大きい。すでに基本的な機能を完備した従来言語(C)の上に構築されたC++は、オブジェクト指向的な特徴も幅広く備えており、テンプレートや例外処理もサポートしている。

正しく使えば、C++での仕事は快適に進む。オブジェクト指向設計であれ従来の方式による設計であれ、実にさまざまな設計を直接的に表現し、効率良く実装することが可能である。あなたが定義する新しいデータ型は、組み込みのデータ型とまったく区別なく使用でき、しかも、より柔軟に使うことができる。慎重に吟味され注意深く組み立てられたクラス群、すなわち、メモリ管理、エイリアシング、初期化とクリーンアップ、型変換など、ソフトウェア開発者の悩みの種であるすべての難題を自動的に処理してくれるクラス群を使うことによって、アプリケーションのプログラミングは簡単かつ直感的で効率の良いものとなり、エラーもほとんど追放できる。C++で効果的なプログラムを書くのは、**正しい方法を知っていれば**、さほど難しいことではない。

けれども、訓練なしでC++を使った場合、そのコードはしばしば読みにくく、保守がたいへんで、拡張もできず、効率も悪く、要するに使えないものができあがってしまうことが多い。

そうならないようにするコツは、つまずきがちなC++の難所を知り、それらを避けて歩く方法を学ぶことだ。それが本書の目的である。私はこの本を、すでに**言語としてのC++**は知っていて、多少は使った経験もある読者を想定して書いている。私が提供したいのは、この言語を**効果的に**使うことによって、ソフトウェアを読みやすく、保守が簡単で、拡張可能で、効率が良く、思いどおりに動くものにするための手引きである。

本書で提供するアドバイスには、大きく分けて2種類のものがある。一方は一般的な設計上の方針、もう一方は言語の個々の機能について押さえるべき基本である。

設計に関する議論は、ある目的をC++で達成するためにはさまざまなアプローチのどれを選べば良いか、という点に絞られる。継承とテンプレートは、どのように使い分ければいいのか。テンプレートと総称ポインタ(void*)の

使い分けは？ public な継承と、private な継承とでは？ private な継承と層を重ねること(40 項)とでは？ 関数オーバーロードとデフォルト引数とでは？ 仮想関数とそうでない関数とでは？ 値渡しと参照(リファレンス)渡しとでは？ これらを正しく判断し、最初に決定しておくことが大切である。なぜなら、もしここで判断を誤ったら、そのミスは開発プロセスのずっと後期になるまで表面化しないかもしれず、その時期に修正しようとしても、たいがい困難で時間がかかり、チームの士気は低下し、費用も嵩んでしまうからだ。

何をしたいか正確に理解している場合でも、それをキチンとやるにはコツが要ることがある。たとえば代入演算子の戻り値の型は何か。new 演算子は、メモリが足りないときには、どうふるまえばいいのか。デストラクタを仮想化するのは、どんなときなのか。メンバの初期化リストは、どのように書けばいいのか。このような細部も、決しておろそかにしてはならない。これらを間違えると、そのプログラムのふるまいは、必ずといっていいほど予想外の(たぶん謎のような)ものになってしまう。それどころか、異常なふるまいが即座に現れない場合は、もっと深刻な事態となる。さまざまな未検出のバグをかかえたまま、幽霊のようなコードがテストをすり抜けて市場に出てしまったら……。それはいつ爆発するかわからない時限爆弾のようなものだ。

さて、この本は、最初から最後まで通して読まなければ意味がない、というタイプの本ではない。ページの順に読まなくたって構わない。本書の内容は 50 項目に細分されていて、それぞれの項目は多かれ少なかれ独立した読み物になっている。ただし、項目が別の項目を参照していることは、しばしばある。だから、とくに興味のある項目から読み始め、参照を追って別の項目に進むというやり方もある。

同じジャンルの項目を集めた章立てにしてあるので、たとえばメモリ管理やオブジェクト指向設計など、特定の話題に関心を持っているのなら、それに該当する章を読み進んでいくなり、あるいはそこを出発点として、参照を追って別の項目に進めばいいだろう。もっとも本書の内容は、C++ で効果的なプログラミングを行う上で、まったく基本的なことばかりである。だから、本書のどの項目も、他のすべての項目と結局は何らかの関係があるはずだ。

本書は C++ 言語のリファレンスブックではないし、この言語を最初から勉強するための教科書でもない。たとえば私は、new 演算子を自作する上で注意すべき点については、何でも喜んでお教えするが(7 項から 10 項)、この関数が void* を返すこと、最初の引数は size_t 型でなければならないことなどは、他の本を読めばわかることだから省略する。そういう情報を載せた C++ の入門書は数多く出版されている。

本書の目的は、C++ でのプログラミングのうち、通常は表面的にしか扱われていない(あるいはまったく無視されている)側面を強調することにある。他の本は、この言語のさまざまなパーツについて説明しているが、本書は、

それらをどう組み合わせれば効果的なプログラミングができるのかを説明する。他の本は、プログラムをコンパイルする方法を解説しているが、本書は、コンパイラがチェックしてくれない問題の予防方法を解説する。

ほとんどの言語と同じく、C++にも豊富な「民間伝承」がある。それらは、いわば聖なる言い伝えの一部として、プログラマからプログラマへと、たいがいは口述によって伝承されてきたものだ。本書は、こうして蓄積された知恵の一部を、よりアクセスしやすい形式で記録しようという、私なりの試みでもある。

ただし本書で扱うC++は、規則を守り**可搬性のある**C++だけである。本書では、ISO/ANSIの標準規格にある機能だけを使っている。可搬性は本書の主眼点の1つである。だから、実装に依存するようなハックや小細工の手法を探している方の役には立たないだろう。

とはいえ、標準規格に記述されているC++は、残念ながら、読者がお馴染みのコンパイラベンダーがサポートしているC++とは、どこかしら異なっている場合がある。このため、C++言語の比較的新しい機能を使うと便利だということを指摘する場合、それが使えないときでも効果的なソフトウェアを作成できるような別の方法も同時に示している。将来必ず使えるようになる機能を知らないで無駄な努力をするのはばかばかしいことだ。しかしその反面で、究極無比なる至高のC++コンパイラが手に入るまでじっと待ち続けるわけにもいかない。結局、いま手に入る道具を使って仕事をこなさなければならぬわけで、本書はまさにその作業を援助するためのものだ。

C++コンパイラは、どれも同じではない。コンパイラは標準に準拠して実装されているわけだが、どこまで忠実かはさまざまだ。だから、少なくとも2種類のコンパイラを使ってコードを開発してみることをお勧めする。そうすれば、ベンダーが独自に行った拡張や、標準の誤った解釈などに依存してしまうというミスはある程度は避けることができる。また、コンパイラ技術の危険な最先端(つまり、まだ1社しかサポートしていないような最新機能)で痛い目に会うこともなくなるだろう。そのような機能は、しばしば実装が貧弱であり(バグがあるか、遅いか、たいがいは両方)登場したばかりなのでC++コミュニティにも正しい使い方をアドバイスできるほどの経験が蓄積されていない。最先端に行くのは魅力的かもしれないが、信頼性の高いコードを作ることがあなたのゴールだとしたら、みんなの先頭に立って藪をかき分ける役は、他の誰かにやってもらうほうが賢明というものである。

本書はC++を礼賛するものではない。完璧なC++ソフトウェアへの唯一正しい道を示す福音など、どこを探しても見つからないだろう。本書の50の項目は、それぞれより良い設計を考え出すための方法、よくある問題を回避する方法、より効率を高めるための方法などのガイダンスを提供しているが、無条件にそうすべきだというような普遍的な項目は1つも無い。ソフトウェアの設計と実装は複雑な仕事であり、しかもハードウェアやオペレーティング

グシステムやアプリケーションの制限によって必ず歪められてしまうものだから、私にできるのは、より良いプログラムを作るためのガイダンスを提供することがせいぜいである。

すべてのガイドラインを常に守っていれば、C++の中でも最もありふれた種類の罠に陥ることはまずないだろう。しかしガイドラインには例外が付きものである。各項に解説がついているのはそのためだ。これらの解説は、本書の中で最も重要な要素である。各項目は「こうしよう」、「こうしてはならない」と断言するが、それを支えている論理を理解して初めて、あなたが開発しているソフトウェアや、あなたが悩んでいるユニークな制約に、その断言を本当にあてはめていいかどうかを正しく判断できるのである。

だから本書は、C++のふるまいを知り、なぜそのようにふるまうのか、そのふるまいをどのように利用できるかを学ぶつもりで読むのが最も適している。本書の項目を無条件に厳守するのは明らかに不適切であるが、それと同時に、確かな理由もなしに本書のガイドラインを破るのもやめておいたほうがいい。

このような本では、言葉の定義にこだわるのは無意味である。その手の楽しみは、法律家気取りの言語マニアに任せておこう。ただし、C++には誰もが理解しなければならない特殊な用語も少ないが存在する。以下に示す用語は本書で非常によく登場するから、どういう意味で使っている言葉なのかをここで明らかにしておこう。

宣言は、オブジェクト、関数、クラス、テンプレートの名前と型をコンパイラに知らせるが、細部は省かれている。以下に示すものはいずれも宣言である。

```
extern int x; // オブジェクト宣言
int numDigits(int number); // 関数宣言
class Clock; // クラス宣言
template<class T>
class SmartPointer; // テンプレート宣言
```

これに対して、**定義**は、コンパイラに詳細な情報を伝える。オブジェクトの定義は、そのオブジェクトのためにコンパイラがメモリを割り当てるべき場所を示す。関数または関数テンプレートの定義は、コードの本体を提供する。クラスまたはクラステンプレートの定義は、そのクラスまたはテンプレートのメンバのリストを示す。

```
int x; // オブジェクト定義

int numDigits(int number) // 関数定義
{ // (この関数は、
    int digitsSoFar = 1; // パラメータに含まれる
                        // 数字の数を返す)

    if (number < 0) {
        number = -number;
    }
}
```

```

    ++digitsSoFar;
}

while (number/=10) ++digitsSoFar;

return digitsSoFar;
}

class Clock {                // クラス定義
public:
    Clock();
    ~Clock();

    int hour() const;
    int minute() const;
    int second() const;
    ...
};

template<class T>
class SmartPointer {        // テンプレート定義
public:
    SmartPointer(T *p = 0);
    ~SmartPointer();

    T * operator->() const;
    T& operator*() const;
    ...
};

```

次はコンストラクタの種類。**デフォルトコンストラクタ**は、引数なしで呼び出せるコンストラクタである。このようなコンストラクタは、パラメータが1つもないか、あるいは、すべてのパラメータについてデフォルトの値が決まっているかのどちらかである。オブジェクトの配列を定義したいときは、一般にデフォルトコンストラクタが必要だ。

```

class A {
public:
    A();                // デフォルトコンストラクタ
};

A arrayA[10];          // コンストラクタが10回呼び出される

class B {
public:
    B(int x = 0);      // デフォルトコンストラクタ
};

B arrayB[10];          // コンストラクタが、それぞれ
                        // 引数を0として、10回呼び出される

class C {
public:
    C(int x);          // デフォルトコンストラクタではない
};

```

```
C arrayC[10]; // これはエラー
```

クラスのデフォルトコンストラクタにデフォルトのパラメータ値がある場合、あなたが使っているコンパイラは、オブジェクトの配列を許さないかもしれない。一部のコンパイラは、C++標準規格案準拠と称しているのに、上記の arrayB の配列定義を受け付けてくれないのだ。これは、標準規格による C++ の記述と、特定のコンパイラによる C++ 言語の実装との間に見られる不一致の一例にすぎない。私が知っているコンパイラは、全部が全部、こういう欠点を何かしら持っているのだ。コンパイラベンダーが標準規格に追いつくまでは、柔軟に対処するつもりでいなければならない。そして、そう遠くない将来、標準として記述された C++ と実際の C++ コンパイラが受け付ける言語とが一致することに望みを託そう。

ちなみに、デフォルトのコンストラクタを持たないオブジェクトの配列を作りたいときは、代わりにポインタの配列を作るというのが常套手段である。こうすると各ポインタは new を使って個別に初期化できる。

```
C *ptrArray[10]; // コンストラクタは呼び出されない

ptrArray[0] = new C(22); // 1個のCオブジェクトを割り当てて
                        // 構築する
ptrArray[1] = new C(4); // 上に同じ
...

```

用語の定義に戻ろう。**コピーコンストラクタ**は、同じ型の別のオブジェクトを使って、オブジェクトを初期化するのに使用される。

```
class String {
public:
    String(); // デフォルトコンストラクタ
    String(const String& rhs); // コピーコンストラクタ
private:
    char *data;
};

String s1; // デフォルトコンストラクタを呼び出す
String s2(s1); // コピーコンストラクタを呼び出す
String s3 = s2; // コピーコンストラクタを呼び出す

```

コピーコンストラクタの最も重要な役割は、オブジェクトを値で渡す(あるいは値で返す)方法を定義することだろう。一例として、次に示すのは2個の String オブジェクトを連結する関数を書く方法である(ただし、この方法は効率が悪い)。

```
const String operator+(String s1, String s2)
{
    String temp;

    delete [] temp.data;

```

```

temp.data =
    new char[strlen(s1.data) + strlen(s2.data) + 1];

strcpy (temp.data, s1.data);
strcat (temp.data, s2.data);

return temp;
}

String a("Hello");
String b(" world");
String c = a + b;    // c = String("Hello world")

```

この operator+[加法演算子]は、パラメータとして 2 個の String オブジェクトを受け取り、結果として 1 個の String オブジェクトを返す。パラメータと結果の両方は値によって渡されるはずなので、コピーコンストラクタが、s1 を a で初期化するとき 1 回、s2 を b で初期化するとき 1 回、そして c を temp で初期化するときにも 1 回呼び出される。実際には、コピーコンストラクタは、もっと何度も呼び出されるかもしれない。コンパイラが、中間的なテンポラリ[一時的]オブジェクトを生成したい場合もあり、そうするのは別に違反ではないからだ。ここで重要なのは、値渡しは「コピーコンストラクタの呼び出し」を意味するということである。

ところで、String のための operator+を実際に作る時は、こんな方法は使わないのが普通だ。const String オブジェクトを返すのは正しいが(21 項、23 項)、2 個のパラメータはリファレンス渡しにしたいからだ(22 項)。

もっとも実際には、String のための operator+など、できれば書かずに済ませたいものだし、たいがいは書かずに済むはずだ。なぜなら C++標準ライブラリ(49 項)には、文字列型が含まれていて(その名もずばり、string という)、上記の operator+とそっくり同じようなことを行う string オブジェクトのための operator+も、これに含まれているからだ。本書では、私は String と string の両方のオブジェクトを使い分けている(前者は大文字の S で始まる点に注意)。単に汎用的な文字列を使う必要があり、その実装については問わないような場合、C++標準ライブラリの一部である string クラスを使う。あなたも、そうすべきだ。けれども、C++のふるまいを説明したいときもあり、そういう場合には実装コードを示さなければならない。標準ではない String クラスを使うのはそのような場合である。プログラマとしてあなたがコーディングするときは、文字列オブジェクトが必要なときは常に標準の string 型を使うべきだ。通過儀礼のように、自分で文字列クラスを開発していたのは、もはや昔話である。しかしながら、string と似たようなクラスの開発に関わる問題については、今でも理解しておく必要があり、そんなときには String を使うのが便利なのだ(そして他の目的には使わない)。char*を使った生の文字列について言えば、よほどの事情でもない限り、そういう古めかしいものは、もう使わないほうがいい。string 型は、い

まや上手に実装すれば、実質的にはすべての面において(効率も含めて 49 項)char*よりも優れたものにできる。

次に取り組むべき2つの用語は、**初期化と代入**だ。オブジェクトが初期化されるのは、初めて値を与えられるときに限られる。クラスのオブジェクトやコンストラクタを持つ構造体の初期化は、**必ず**コンストラクタの呼び出しによって行われる。オブジェクトの代入は、これとはまったく別のもので、すでに初期化されているオブジェクトに新しい値を与えるのが代入である。

```
string s1;           // 初期化
string s2("Hello"); // 初期化
string s3 = s2;     // 初期化
s1 = s3;           // 代入
```

処理についてだけ考えると、初期化はコンストラクタによって行われ、代入は operator=[代入演算子]によって行われるという違いがある。別の言い方をすれば、この2つの処理は、別々の関数呼び出しによって行われるのである。

このように区別している理由は、この2種類の関数では考慮すべきことがらが違うからだ。一般にコンストラクタは引数の妥当性をチェックしなければならないが、たいがいの代入演算子は、(すでにオブジェクトは構築済みなので)引数は正しいものとみなすことができる。一方、代入される対象のほうには、これから構築するオブジェクトとは違って、すでにリソースが割り当てられているだろう。そうしたリソースは、一般に新しいリソースを割り当てる前に解放しなければならない。リソースにはメモリも含まれているのが普通だから、代入演算子は新しい値のためにメモリを割り当てる前に、古い値のために割り当てられていたメモリを解放する必要がある。

以下に、Stringのコンストラクタと代入演算子の実装例を示す。

```
// Stringのコンストラクタの一例*1
String::String(const char *value)
{
    if (value) { // もしvalueがnullポインタでなければ
        data = new char[strlen(value) + 1];
        strcpy(data, value);
    }
}
```

*1 訳注：著者の正誤表によれば、Stringクラスのデフォルトコンストラクタには、デフォルトのパラメータ値0(ヌルポインタ)を与えるべきではない。なぜならC++標準のstring型は、ヌルポインタを受け取ったときのふるまいが不定だからである。この例を、もっと標準のstring型に近づけるには、次のようなコンストラクタが必要である。

```
class String {
public:
    String(); // デフォルトコンストラクタ
    String(const char *); // Cの文字列用のコンストラクタ
    ...
};
```

```

else {
    data = new char[1];
    *data = '\0';
}
}

// Stringの代入演算子の一例
String& String::operator=(const String& rhs)
{
    if (this == &rhs)
        return *this;
    delete [] data;
    data =
        new char[strlen(rhs.data) + 1];
    strcpy(data, rhs.data);
    return *this;
}

```

このコンストラクタが、パラメータの妥当性を必ずチェックし、データメンバが正しく初期化されるように注意を払っている(null 文字で終結している char* を指していなければならない)ことに、注目されたい。一方、代入演算子は、パラメータが正しいことを前提としているが、そのかわりに、自分自身への代入(17 項)などといった異常なケースの検出や、新しいメモリを割り当てる前に古いメモリを解放することに注意を払っている。上記の2つの関数の違いは、オブジェクトの初期化とオブジェクトの代入との違いの典型を示している。ところで、delete に [] が付いているのを初めて見たという方は、5 項を参照していただきたい。なるほど、と納得できるだろう。

最後に、**クライアント**という言葉にも説明が必要だ。クライアントとは、あなたが書いたコードを使うプログラムのことである。本書でクライアントと言う場合、私が想定しているのは、あなたのコードを見てそれが何を行うのか解読しようとする人、あなたのクラス定義を読んでそのクラスを継承すべきかどうか判断しようとする人、あなたが下した設計上の判断を調べてその根本的な理由を読み取ろうとする人である。

クライアントについて考慮する習慣がない読者もおられると思うが、私は「クライアントにかかる負担は可能な限り軽くすべきだ」と読者に納得していただけるように、じっくりと時間をかけて説明するつもりである。結局あなた自身も、他の人が開発したソフトウェアのクライアントなのだ。自分にまわってくる仕事は、なるべく楽にしておいて欲しいと思うのは、誰でも同じことである。それに、読者もいつか、自分が昔書いたコードを使わなければならないという居心地の悪い状態に陥るかもしれない。その時は、あなた自

2 ここに示した const char 型の引数を取る String のコンストラクタは、ヌルポインタが渡されるケースを別個に処理しているが、標準の string 型には、このような対処は要求されていない。ヌルポインタによって string を作成しようとすると、その結果は不定である。ただし、char* の形式による空文字列("")によって string オブジェクトを作成するのは安全である。

身がクライアントになるのだ。

本書では、あるいは読者に馴染みがないかもしれない2つの構造を使う。これらは比較的最近になってC++に追加されたものだ。その1つはbool型で、値はtrueとfalseという2つのキーワードのどちらかである。組み込みの関係演算子(<, >, ==など)が返す型や、if、for、while、doのステートメントの条件部で評価される型には、いまではbool型が使われる。もしコンパイラが、まだboolを実装していなかったら、近似の効果を得るには、typedefを使ってboolを定義し、trueとfalseには定数オブジェクトを使うのが簡単である。

```
typedef int bool;
const bool false = 0;
const bool true = 1;
```

この方法ならばCとC++の伝統的な用語法と互換性があり、このエミュレーションを使ったプログラムのふるまいは、boolをサポートするコンパイラに移植しても変わらないはずである。

もう1つの新しい構造とは、実際には4つのキャスト書式、すなわちstatic_cast、const_cast、dynamic_cast、reinterpret_castである。

従来のC言語スタイルのキャストは、このようなものだった。

```
(type) expression           // 式 expression を、
                             // 型 type にキャストする
```

新しいキャストは、以下のようになる。

```
static_cast<type>(expression) // 式 expression を、
                              // 型 type にキャストする
const_cast<type>(expression)
dynamic_cast<type>(expression)
reinterpret_cast<type>(expression)
```

キャストの書式に4種類あるのは、それぞれの目的が違うからだ。

const_castは、オブジェクトやポインタの定数性(constと定義されていること)をキャストによって取り払うのが目的である(21項)。

dynamic_castは、「安全なダウンキャスト」(39項)を行うのに使う。

reinterpret_castは、実装に依存する結果を生み出すようなキャスト(たとえば関数ポインタ型同士のキャスト)のために考案された(読者がreinterpret_castを必要とすることはほとんどないだろう。本書でも使っていない)。

static_castは、「その他もろもろ」のキャストである。ほかのどのキャストを使うのも不適切である場合は、このキャストを使う。従来のC言語ス

タイトルのキャストに最も近い意味を持つのがこのキャストである。

従来のキャストは今でも有効だが、新しいキャスト書式を使ったほうがいい。コードの中で識別するのがずっと簡単になるし(人間にとっても、grepのようなツールでも)書式によってそれぞれのキャストの目的がより厳密に指定されるため、コンパイラが間違った使い方を検出できる。たとえば、何かが `const` であることをキャストによって取り払う目的で使えるのは `const_cast` だけである。他の新キャスト書式によって、オブジェクトやポインタの定数性を取り払おうとしたら、そのキャストはコンパイルを通らないだろう。

新しいキャストについてもっと知りたい方は、C++に関する最新の入門書を探すが、あるいは筆者の『*More Effective C++*』の item 2 を参照されたい(『*More Effective C++*』の概要は本書のあとがきにある)。

本書のコーディング例では、オブジェクト、クラス、関数などに、なるべく意味のある名前を付けるようにした。他の本には、識別子を選ぶのに「簡潔は機智の神髄」という古いことわざを信奉しているものが多いけれど、私は自分の機智を示すよりも、ものごとをはっきりさせることに興味がある。だから私は、プログラミング言語の本に謎めいた識別子を使うという伝統を破る努力をしている。しかし、そうは言っても自分で使い慣れた2つのパラメータ名については、どうしても使いたいという誘惑に勝てないときがあった。それらの意味は、ここで説明しておかないと、とくにコンパイラ書きに手を染めた経験がある人以外はわかりにくいだろう。

その2つの名前とは `lhs` と `rhs` で、それぞれ左側(left-hand side) 右側(right-hand side)の意味である。これらの名前は、二項演算子(`operator==` [等価演算子]や、`operator*`などの算術演算子)を実装する関数のパラメータ名に使う。たとえば、`a` と `b` が有理数(rational number)を表すオブジェクトで、有理数をメンバではない `operator*`関数によって乗算することが可能だとしたら、式

```
a * b
```

は関数呼び出し

```
operator*(a, b)
```

と等価である。

23項に出てくるが、私は `operator*`を次のように宣言する。

```
const Rational operator*(const Rational& lhs,
                          const Rational& rhs);
```

おわかりのように、左側のオペランド `a` は関数内では `lhs` と呼ばれ、右側のオペランド `b` は関数内では `rhs` と呼ばれる。

また、ポインタ名も、同じルールに従って省略することにした。たとえば `T`

型のオブジェクトへのポインタには、pt という名前を付けることがある(pt とは“ pointer to T ”の略)。以下に、いくつか例を示す。

```
string *ps;           // ps = string へのポインタ

class Airplane;
Airplane *pa;        // pa = Airplane へのポインタ

class BankAccount;
BankAccount *pba;    // pba = BankAccount へのポインタ
```

リファレンスも同じように命名する。rs は string へのリファレンス、ra は Airplane へのリファレンスという意味である(rs とは“ reference to string ”の略)。

また、メンバ関数について説明するとき、mf(“ member function ”の略) という名前を使うこともある。

混乱があるといけないので、一応念のために書き添えておくと、本書で「C 言語」という場合は、ISO および ANSI の規格によって定められた標準の C 言語を意味している。昔の、型付けの弱い「古典的な」C 言語のことではない。

第 1 章

CからC++への移行

誰でも C++に慣れるまでには少し時間がかかるものだが、年季の入ったベテランの C プログラマ諸氏の中には、C++への移行のプロセスを考えると気が滅入るという方も多いただろう。C 言語は実質的には C++のサブセットなので、C 言語の古いトリックも全部使えるのだが、その多くはもはや適切ではない。たとえば C++プログラマにとって、ポインタへのポインタは、おかしなものに見える。どうしてポインタへのリファレンスを使わないのだろう、と私たちは思うのだ。

C は、実にシンプルな言語だ。C 言語が提供しているものは、結局はマクロであり、ポインタであり、構造体、配列、そして関数である。どのような問題も、必ずマクロ、ポインタ、構造体、配列、関数によって解決するのが C 言語だ。C++は違う。C++にも、マクロ、ポインタ、構造体、配列、関数はもちろん残っている。しかし C++には `private` メンバと `protected` メンバがある。関数のオーバーロードがあり、デフォルトパラメータがあり、コンストラクタとデストラクタがある。ユーザー定義の演算子、インライン関数、リファレンス、フレンド、テンプレート、例外、名前空間などもある。C++は、C よりもずっと広大な空間を設計者に与えている。考慮すべき選択肢の数が、ずっと多いのである。

このようにさまざまな選択肢を前にすると、ほとんどの C プログラマは自分の殻に閉じこもり、慣れ親しんだやり方に固執しようとする。ほとんどの場合、それが罪悪だということにはならないが、C のやり方の一部は C++の流儀に反している。とくに以下に示すようなアドバイスには、**とにかく従ったほうがいい**。

1 項 #defineではなく、constとinlineを使おう

この項は、「プリプロセッサよりもコンパイラを使おう」と呼んだほうが正しいかもしれない。#define は、しばしば言語そのものの一部ではないかのように扱われるからだ。それが、#define の問題点の1つである。たとえば次のように書いたとしよう。

```
#define ASPECT_RATIO 1.653
```

このシンボル名 ASPECT_RATIO を、コンパイラが読むことはないかもしれない。この名前は、コンパイラがソースコードを読み込むよりも前に、プリプロセッサによって解釈され取り除かれてしまう場合があるのだ。その結果、ASPECT_RATIO という名前はシンボルテーブルに入らない。コンパイル中に、この定数の使い方に関してエラーが起きると混乱するかもしれない。そういうコンパイラのエラーメッセージには ASPECT_RATIO ではなく 1.653 という数字が出てくるからだ。もし ASPECT_RATIO が他人の書いたヘッダファイルの中で定義されていたとしたら、この 1.653 というのが何の数字だかわからないので、たぶんそれを調べるのに長い時間がかかってしまうだろう。同じ問題は、シンボリックデバッグを使うときにも生じる。やはり、プログラミングで使用した名前が、シンボルテーブルに入らない場合があるからである。

この悲惨なシナリオを救うのに、シンプルにして十分な方法がある。プリプロセッサ用のマクロを使うのではなく、定数を定義すればいい。

```
const double ASPECT_RATIO = 1.653;
```

このアプローチは、魔法のようにうまく働く。ただし、注意すべき特別なケースが2つある。

第一に、定数ポインタの定義には、ちょっとしたコツがある。定数の定義は(いろいろなソースファイルによってインクルードされる)ヘッダファイルに入れるのが典型的なので、そのポインタが指す対象を const 宣言するだけでなく、ポインタそのものも const 宣言することが、しばしば重要となるのだ。ヘッダファイルの中で、たとえば char* を使った文字列定数を定義するときは、次のように const を2回使わなければならない。

```
const char * const authorName = "Scott Meyers";
```

const の意味と、その使い方について、とくにポインタに関連した議論は、21 項を参照されたい。

第二に、クラスに特有の定数を定義したほうが便利な場合があり、その場合は書き方も少し違ってくる。定数のスコープをクラス内に限定するには、定数をクラスのメンバにしなければならない。また、その定数のコピーがいくつも存在しないように、static メンバにしなければならない。

```
class GamePlayer {
private:
    static const int NUM_TURNS = 5;    // 定数の宣言
    int scores[NUM_TURNS];           // 定数の使用
    ...
};
```

ここで、ひとこと助言しておく、上にあるのは NUM_TURNS の定義ではなく、宣言である点に注意。実装ファイルで、static なクラスメンバを定義する必要がある。

```
const int GamePlayer::NUM_TURNS; // 必要な定義
                                // クラス実装ファイルに入れる
```

もっとも、気になって夜も眠れないというほど深刻な問題ではない。定義を書き忘れたら、リンカがちゃんと教えてくれる。

古いコンパイラは、この構文を受け付けないかもしれない。static なクラスメンバに、宣言時に初期値を与えることは、昔は違法だったからである。そのうえ、クラス内の初期化が許されるのは整数型(すなわち、int、bool、char など)だけで、しかも使えるのは定数だけだったのだ。上記の構文を使えない場合は、定義のときに初期値を与えればいい。

```
class EngineeringConstants {    // これはクラスの
private:                        // ヘッダファイルに入れる
    static const double FUDGE_FACTOR;
    ...
};
```

```
// これはクラスの実装ファイルに入れる
const double EngineeringConstants::FUDGE_FACTOR = 1.35;
```

ほとんどの場合は、これだけで十分である。唯一の例外は、クラスのコンパイル時にクラス定数の値が必要な場合だ。上記の、GamePlayer::scores の配列がその例である(コンパイラが、コンパイル時に配列のサイズがわからないと文句を言う場合)。このようなとき、整数型クラス定数の初期値をクラス宣言で指定することを頑迷にも禁止するコンパイラをごまかすには、「enum ハック」という愛称で親しまれている手口を使うのが定石である。このテクニックは、列挙型の値は int を使えるところならどこでも使えるという事実を利用するもので、GamePlayer の場合は次のように定義する。

```
class GamePlayer {
private:
    enum { NUM_TURNS = 5 };      // これが"enum ハック"
                                // NUM_TURNS は 5 を表す
                                // シンボル名になる
    int scores[NUM_TURNS];     // 問題なし
    ...
};
```

主に歴史的な価値しか認められない(つまり1995年より前に書かれた)コンパイラを扱うときを除いて、このenumハックを使う必要はない。それでも、それがどんなものかは知っておく価値がある。まだものがシンプルだった当時のコードに遭遇することは、今でも稀ではないからだ。

話をプリプロセッサに戻そう。#define ディレクティブには、もう1つ、よくある(しかし、良くない)使い方がある。マクロは関数に似ているが、関数呼び出しのオーバーヘッドがないので、#define でそういうマクロを実装することが多い。典型的な例は、2つの値の最大値を計算するマクロである。

```
#define max(a,b) ((a) > (b) ? (a) : (b))
```

この一節には、考えるだけでも憂鬱になるくらい、数多くの危険が隠れている。ラッシュアワーにフリーウェイで遊ぶくらい危険でいっぱいなのだ。

このようなマクロを書くときは、次のことを決して忘れてはならない。マクロの本体を書くときは、すべての引数をカッコで囲むこと。さもないと、誰かがマクロの中に式を使ったとき、トラブルのもとになる。しかし、その点をきちんと守ったとしても、次のようなおぞましい事態は起こるのだ。

```
int a = 5, b = 0;

max(++a, b);      // aは2回インクリメントされる
max(++a, b+10);  // aは1回インクリメントされる
```

この場合、maxの中でaに何が起こるかは、aが何と比較されるかに依存する!

ありがたいことに、こんなものを我慢して使う必要はない。インライン関数(33項)を使えば、マクロに匹敵する効率が得られるだけでなく、通常の間数と同じく、予測可能なふるまいと型安全性を得ることができる。

```
inline int max(int a, int b) { return a > b ? a : b; }
```

これは上記のマクロとまったく同じではない。このインライン版のmaxは、intを使っての呼び出しにしか使えないからだ。けれどもテンプレートを使えば、その問題はうまく解決できる。

```
template<class T>
inline const T& max(const T& a, const T& b)
{ return a > b ? a : b; }
```

このテンプレートによって生成されるのは、一個の間数ではなく関数ファミリーである。この家族の一員は、いずれも同じ型に変換可能な2つのオブジェクトを受け取って、その2つのオブジェクトのうち大きいほう(のconstバージョン)へのリファレンスを返す。型Tに何が使われるかわからないので、効率良く処理するために、引数にも戻り値にもリファレンスを使う(

22 項)

ところで、max のように汎用的に使える便利な関数のテンプレートを書くと考えたときは、それがすでに存在していないか、標準ライブラリ(49 項)をチェックしたほうがいい。max の場合、読者は一番乗りの栄冠が他人の頭に載っているのを見て、心地よい驚きを経験するだろう。max はすでに C++ 標準ライブラリに含まれている。

const と inline が使えるようになって、プリプロセッサの需要は減少したが、まったく不要になったわけではない。#include を捨てる日が来るのはまだ先のことであり、#ifdef と #ifndef も、しばらくはコンパイル制御のための重要な役割を果たし続けるだろう。プリプロセッサ先生、老いたりといえ、まだ引退させるわけにはいかない。しかし彼には、もっと長く頻繁な休暇をぜひとも与えるべきである。

2 項 <stdio.h>ではなく、<iostream>を使おう

確かに、stdio.h は可搬性に優れている。なるほど、効率もいい。使い方もよく知っている。まったくそのとおり。stdio.h は偉大な存在だが、しかし実際にどうかというと、scanf だの printf だのといった連中には大いに改良の余地がある。とくに問題なのは、型安全性と拡張性を欠いている点だ。型安全性と拡張性は、C++ 式のライフスタイルには不可欠なものだから、連中との縁は今すぐ切ったほうがいい。それに、printf/scanf ファミリーの関数では、読み書きの対象になる変数と、その読み書きを制御する書式情報とが別扱いになっているが、これではまるで FORTRAN だ。1950 年代の遺物には、そろそろ別れを告げなければならない。

これら、printf/scanf にあった短所が、operator>> と operator<< の長所になっているのは当然である。

```
int i;
Rational r;    // r は有理数
...
cin >> i >> r;
cout << i << r;
```

このコードをコンパイルするのに必要なのは、Rational 型のオブジェクトを扱える operator>> と operator<< の関数だ。これらがなければエラーになる(int 用の関数は標準である)。しかも、型の異なる変数について、どのバージョンの演算子を呼び出せばいいかは、コンパイラが判断してくれるので、「最初に読み書きされるオブジェクトは int で、次は Rational で」などと、いちいち指定する必要はない。

そればかりか、読み出すべきオブジェクトも書き込むべきオブジェクトも、同じ構文を使って渡せるのだから、scanf のようなばかばかしい決まりごと

を覚える必要はない。なにしろ `scanf` の場合は、ポインタを作っていないければ忘れずにアドレスを取るよう注意する必要があり、ポインタを作っているときはアドレスを取らないように注意する必要があるのだ。そんな細かいことは、C++コンパイラに任せてしまおう。コンパイラにできるのは、せいぜいその程度の仕事だ。人間にはもっと別の仕事がある。最後に、`int` のような組み込み型が、`Rational` のようなユーザー定義型と同じ方法で読み書きされるという点にも注目されたい。`scanf` と `printf` では、逆立ちしたってこんな芸当はできない。

自然数を表すクラスの出力ルーチンを、あなたは次のように書くかもしれない。

```
class Rational {
public:
    Rational(int numerator = 0, int denominator = 1);
    ...
private:
    int n, d;    // 分子(numerator)と分母(denominator)
friend ostream& operator<<(ostream& s, const Rational& r);
};

ostream& operator<<(ostream& s, const Rational& r)
{
    s << r.n << '/' << r.d;
    return s;
}
```

`operator<<`の、このバージョンには、本書の別の項で説明する微妙な(しかし重要な)ポイントも含まれている。たとえば、`operator<<`は、メンバ関数ではない(その理由は19項で説明する)。また、出力すべき `Rational` オブジェクトは、オブジェクトとしてではなく、`const` へのリファレンスとして、`operator<<`に渡す(22項)。これと対になる `operator>>`も、同じように宣言し、実装する。

しかし、実証済みの枯れた技術を使うのが正解というような状況も、やはり存在することは認めざるを得ない。第一に、`iostream` 処理の実装の一部には、それに対応するCのストリーム処理と比べて効率の劣るものもあり、それがアプリケーションにとって大きな違いとなることも(たまには、だが)あるかもしれない。ただし、これは `iostream` が一般にそうだということではなく、そういう実装もあるという意味だから、取り違えないでいただきたい。第二に `iostream` ライブラリは、標準化の過程を経るうちに、かなり基本的なところも変わってきているから(49項)、アプリケーションに可能な限り可搬性を持たせたいときには、ベンダによって標準への追従程度が異なる点が問題になるかもしれない。最後に、`iostream` ライブラリのクラスにはコンストラクタがあるが、`<stdio.h>`の関数にはコンストラクタがないので、`static` オブジェクトの初期化の順序(47項)の問題で、標準Cライブラリ

のほうが使いやすい(要するに、いつ呼び出しても叱られないということがわかっているから)というようなケースも稀には存在するだろう。

`iostream` ライブラリのクラスと関数によってもたらされる型安全性と拡張性は、あなたの第一印象よりもずっと便利なものかもしれない。だから、いくら`<stdio.h>`に慣れ親しんでいるからといって、`iostream` を無視してはいけない。あなたがC++に移行したあとも、思い出だけは残しておけるのだから。

ところで、この項のタイトルは誤植ではない。`<iostream.h>`の間違いでなくて、本当に`<iostream>`なのである。厳密に言うと、`<iostream.h>`などというものは存在しないのだ。標準化委員会は、Cの標準にない他のヘッダ名を切りつめたときに、`<iostream>`を残して“.h”の付く長いほうは排除した。なぜそういうことをしたかは49項で説明するが、ここで理解しておくべき重要なことは、もしあなたのコンパイラが`<iostream>`と`<iostream.h>`の両方をサポートしていたら(これは、ありそうなことだ) 両者には微妙な違いがあるという点である。とくに、`#include <iostream>`と書いた場合、`iostream` ライブラリの諸要素は、名前空間“`std`”の中に包み込まれる(28項)が、`#include <iostream.h>`と書くと、それらの要素はグローバルスコープに置かれる。これらをグローバルスコープに置くと、名前の衝突が起きる可能性がある。そもそも名前空間は、名前の衝突を防ぐために考案されたのだ。ついでに言うと、`<iostream>`とタイプするのは`<iostream.h>`とタイプするよりも楽だ。それだけでも前者を選ぼうという人は多い。

3項 malloc と free よりも、new と delete を使おう

`malloc` と `free` のどこが悪いのか。答えは単純で、このペアは(その同類たちも)コンストラクタとデストラクタを知らないのである。

次に、10個の `string` オブジェクトのための空間を取得する2つの方法を示す(1つは `malloc` を、もう1つは `new` を使う)。

```
string *stringArray1 =
    static_cast<string*>(malloc(10 * sizeof(string)));

string *stringArray2 = new string[10];
```

ここで、`stringArray1` は10個の `string` オブジェクトを格納するのに十分なメモリを指しているが、そのメモリにはまだオブジェクトが構築されていない。そればかりか、綱渡りのようなハックでも発明しない限り、配列の中のオブジェクトを初期化する方法はないのだ。言い換えると、`stringArray1` はまったく使いようがない。一方、`stringArray2` は、完全に構築された10個の `string` オブジェクトによる配列を指しているし、それぞれのオブジェ

クトは、1個の string を受け取るどのような演算にも、安全に使用することができる。

それでも、火の輪くぐりか何かの曲芸を使って、配列 stringArray1 のオブジェクトをなんとか初期化できたものと仮定しよう。プログラムは、あとのほうで、次のことを行わずだ。

```
free(stringArray1);

delete [] stringArray2;    // []が必要な理由は、5項を参照
```

free の呼び出しは stringArray1 が指しているメモリを解放するが、そのメモリにある string オブジェクトに対してデストラクタが呼び出されることはない。もし string オブジェクトが自分でもメモリを割り当てていたとしたら(string オブジェクトは、そういうことをするのが常である)、そのようにして割り当てられたメモリは、返却されずにすべて失われてしまう。ところが、stringArray2 に対して delete を呼び出した場合は、メモリがまだ解放されないうちに、この配列の中にあるオブジェクトのそれぞれに対してデストラクタが呼び出される。new と delete は、コンストラクタとデストラクタに対して、適切な相互作用を行うのだ。明らかに、こちらを選ぶのが正解である。

一般に、new と delete、malloc と free を、混ぜ合わせて使うのだけは避けるべきだ。new で取得したポインタに対して free を呼び出したり、malloc で作ったポインタに delete をかけたりすると、その結果は不定である。そして、われわれは「不定」とはどういう意味かを知っている。不定とは、開発中は正常に動作し、テスト中も正常に動作し、一番大事な顧客の目の前で突然動かなくなる、という意味である。

new/delete と malloc/free との間に互換性がないおかげで、なかなか興味深い複雑な事態が生じることがある。たとえば、普通は<string.h>に入っている strdup 関数は、char*式の文字列を受け取って、そのコピーを返す。

```
char * strdup(const char *ps); // psが指しているものの
                             // コピーを返す
```

あるサイトでは、C と C++ の両方で同じバージョンの strdup を使っている。だから、この関数の内部ではメモリを割り当てるのに malloc が使われる。このため、strdup の呼び出しを書いたら、strdup から返されたポインタに対して必ず free を使わなければならないのだが、ある C++ プログラムは、そのことをすっかり忘れてしまった。ところが、である。このような複雑さを未然に防ごうというので、もう1つのサイトでは C++ の strdup を書き直すことに決めた。その改良版では、関数の中で new を呼び出すので、これを呼び出したからには、あとで delete を使わなければならない。その結果は、ご想像のとおりだ。異なった形式の strdup を使う2つのサイトの間

を、コードが行き来するにつれて引き起こされる可搬性の問題は、まさに悪夢さながらである。

とはいえ、C++のプログラマたちもC言語プログラマたちと同じく、コードの再利用に関心を持っている。そして、mallocとfreeをベースとするCのライブラリにも、再利用する価値が十分にあるコードを含むものが数多く存在するというのは、疑いようのない事実である。そのようなライブラリを活用するときは、ライブラリによってmallocされたメモリを解放したり、ライブラリによってfreeされるメモリをmallocしたりするのは、結局あなた自身の責任で行わなければならない。それでもいいのなら、結構だ。C++プログラムの中でmallocとfreeを呼び出すのは、別に悪いことではない。ただしそれは、mallocから得たポインタは必ずfreeで元に戻し、newから得たポインタは必ずdeleteに行き着くことを保証できればの話である。集中力がなくなって、newとfree、mallocとdeleteをごちゃ混ぜに使い始めたときから、問題が始まるのだ。そうなったら、トラブルを呼び寄せるようなものである。

mallocとfreeがコンストラクタとデストラクタを意識していないこと、malloc/free組とnew/delete組をごっちゃにするのは大学の新入生歓迎パーティーよりも危なっかしいことを考えれば、可能な限りnewとdeleteだけを使うのが最善の策である。

4項 コメントはC++スタイルで書こう

あの懐かしいC言語のコメント構文は、C++でも使える。しかし、C++に取り入れられた新しい「行末までコメント」の構文には、はっきりとした長所がある。たとえば、次のような場合を考えてみよう。

```
if ( a > b ) {
    // int temp = a;           // aとbを交換する
    // a = b;
    // b = temp;
}
```

ここに見られるコードブロックは、何か理由があってコメントアウトされたのだろうが、最初にコードを書いたプログラマは、このコードが何をするのかを示すためにコメントを残している。ソフトウェア書法の驚嘆すべき実例であろう。さて、ブロックをコメントアウトするのにC++方式のコメントを使う場合、それに元のコメントが埋め込まれることには、別に注意を払う必要はない。しかし仮にこの2人がC言語方式のコメントを使っていたとしたら、ある深刻な問題が生じていたはずである。

```
if ( a > b ) {  
    /* int temp = a;    /* aとbを交換する */  
    a = b;  
    b = temp;  
    */  
}
```

ご覧のように、コードブロック全体をコメントアウトするつもりで始まったコメントは、埋め込まれたコメントの末尾で、思いがけず中途のまま終わってしまう。

C言語スタイルのコメントにも、まだ使いみちはある。たとえばCとC++の両方のコンパイラで処理されるヘッダファイルなどだ。とはいえ、C++のコメントを使えるときは、それで通したほうがいい。

ついでに大事な話をしておこう。C言語だけを対象として書かれた時代遅れのプリプロセッサは、C++スタイルのコメントをどう扱えばいいのか知らないで、次のようなコーディングが予期せぬ結果を生むことがある。

```
#define LIGHT_SPEED 3e8    // m/sec( in a vacuum )
```

C++を知らないプリプロセッサは、この行の最後にあるコメントを、なんとマクロの一部として解釈してしまうのだ。もちろん、1項で述べたように、そもそも定数を定義するのにプリプロセッサを使うべきではない。

第2章

メモリ管理

C++のメモリ管理について考慮すべきことは、おおざっぱに分けて「まともに動かすこと」と「効率良く実行させること」の2つである。優秀なプログラマは、前者を優先させなければならないということをよく理解しているはずだ。目が眩むほど高速で、驚くほど小さいプログラムができたとしても、意図したとおりに動かなければ、およそ使い物にはならない。ほとんどのプログラマにとって、「まともに動かすこと」とは、メモリの割り当てルーチンと解放ルーチンを正しく呼び出すということである。一方、「効率良く実行させること」は、しばしばメモリの割り当てルーチンと解放ルーチンのカスタムバージョンを自作するという意味になる。そうすると「まともに動かすこと」は、なおさら重要である。

正確な動作を重視する立場から言うと、C++がメモリリークの潜在という欠点をC言語から受け継いでいるのは、まったく頭の痛い問題である。仮想記憶という偉大な発明もあるが、それにしただけ無限ではないし、誰もが仮想記憶を持っているとは限らない。

C言語のメモリリークは、`malloc`によって割り当てられたメモリが、いつまでたっても `free` で解放されない場合に生じる。C++の舞台では役者が `new` と `delete` に変わったが、ストーリーに大した違いはない。もっとも、デストラクタのおかげで状況はいくらか改善された。オブジェクトを破壊するときには、`delete` の呼び出しを必ず行わなければならないが、そういう呼び出しはデストラクタに入れればいい、ということになったからだ。しかし、同時に注意事項も増えてしまった。つまり、`new` は暗黙のうちにコンストラクタを呼び出すし、`delete` は暗黙のうちにデストラクタを呼び出す。しかも、`operator new` と `operator delete` は、プログラマが自分のバージョンを(クラスの内側と外側の両方で)勝手に定義できるのだ。これだけ複雑になると、どこでミスをするか、わかったものではない。以下の項目は、一般的なミスを防止するためのアドバイスである。

5 項 newとdeleteのペアは、同じ形式に揃えよう

下記のコードには間違いがある。何だろう？

```
string *stringArray = new string[100];

delete stringArray;
```

newを使ったあとに、ちゃんとdeleteを使っているのだから、とくに問題はなさそうに見える。ところが、それでも1つたいへんな間違いがある。これではプログラムの動作が不定になってしまうのだ。少なくとも、stringArrayが指している100個のstringオブジェクトのうち99個は正しく破壊されないはずだ。なぜなら、それらのデストラクタは、おそらく永遠に呼び出されないからである。

newを使うと、2つのことが起きる。第一に、メモリが割り当てられる(これはoperator newという関数を介して行われるのだが、その点については7項から10項で詳しく述べる)。第二に、そのメモリについて1個または複数のコンストラクタが呼び出される。そして、deleteを使うと、別の2つのことが起きる。まず、1個または複数のデストラクタが、そのメモリに対して呼び出され、それから割り当てられていたメモリが解放される(operator deleteという関数を介して、8項)。deleteにとって、削除すべきメモリに入っているオブジェクトの数は重要な問題である。それによって、デストラクタを何回呼び出せばいいかが決まるのだ。

この問題は、実はもっと単純化できる。あなたが削除用に渡したポインタは、1個のオブジェクトを指しているのか、それともオブジェクトの配列を指しているのか？ どちらなのか、あなたが教えなければdeleteにはわからないのだ。deleteを使うときにブラケット([])を書かないと、deleteはポインタが指しているのは1個のオブジェクトであると想定する。ブラケットがあれば、配列を指しているものと判断する。

```
string *stringPtr1 = new string;
string *stringPtr2 = new string[100];
...
delete stringPtr1;          // 1個のオブジェクトを削除する
delete [] stringPtr2;      // オブジェクトの配列を削除する
```

stringPtr1に“ [] ”を使ったとしたら、どうなるか。その結果は不定である。では、stringPtr2に“ [] ”を使わなかったら、どうなるか。それも、また不定なのだ。それだけではない。intのような組み込み型でさえも、結果は不定である。組み込み型にはデストラクタがないが、それでも不定である。だから、ルールは簡単だ。newを呼び出すときに“ [] ”を使ったら、deleteを呼び出すときも“ [] ”を使わなければならない。newを呼び出すときに“ [] ”を使わなかったら、deleteを呼び出すときに“ [] ”を使ってはならない。

とくに、あなたが書くクラスのデータメンバにポインタがあり、しかも複数のコンストラクタを提供するときには、このルールに十分気をつけなければならない。その場合は、ポインタメンバの初期化を行うのに、すべてのコンストラクタで**同じ形式の new** を使わなければならない。さもないとデストラクタは、ポインタにどの形式の delete を使えばいいのか、わからなくなってしまう。この問題については 11 項で詳しく説明する。

typedef をよく使う人にも、このルールは重要である。typedef で定義した型のオブジェクトを出現させるのに new を使う場合、typedef を書く人は、どの形式の delete を使うべきかを書き残しておかなければならない。下の typedef を例として、考えてみよう。

```
typedef string AddressLines[4]; // 個人の住所
                                // 4行からなる
                                // 各行は1個のstring
```

AddressLines は配列なので、次のような new を使ったら

```
string *pal = new AddressLines; // "new AddressLines"が
                                // 返すのは、
                                // "new string[ 4 ]"の場合と
                                // 同じく string* である
```

次のように配列形式の delete を対応させなければならない。

```
delete pal; // 不定!

delete [] pal; // これが正解
```

このような混乱を防ぐには、配列型に typedef を使うのは控えるのが一番だろう。もっとも、それは簡単なことだ。C++の標準ライブラリ(49 項)には string と vector のテンプレートが含まれているので、組み込み型の配列を作る必要はほとんどないのである。たとえばこの例の場合、AddressLines は string の vector によって定義できる。すなわち、AddressLines の型は、vector<string>となる。

6 項 デストラクタでポインタメンバに delete を使うのを忘れないようにしよう

動的なメモリ割り当てを行うクラスは、ほとんどの場合、(1 個または複数の)コンストラクタでは new を使ってメモリを割り当て、そのあと、デストラクタでは delete を使ってそのメモリを解放する。クラスを最初に書くとき、これをきちんと行うのは別に難しいことではない(ただし、当然ながら、コンストラクタのどれかがメモリを割り当てた可能性のあるメンバには、す

べて delete をかけることを忘れてはならない)。

けれども、クラスが長い期間にわたって保守され拡張されていくのにしたがって、状況は難しくなる。クラスを変更するプログラマが、そのクラスを最初にしたのと同じ人物とは限らないからだ。ポインタメンバを追加するときには、たいがいの場合、次のことが必要になるが、そんな状況では、うっかり忘れてしまうことが多いのだ。

それぞれのコンストラクタで、そのポインタを初期化すること。そのポインタにメモリを割り当てないコンストラクタは、ポインタを 0 (すなわち、ヌルポインタ) で初期化すること。

代入演算子では、既存のメモリを削除して、新しいメモリを割り当てること(17 項)。

デストラクタで、そのポインタに対して delete を行うこと。

コンストラクタでポインタを初期化するのを忘れて、代入演算子の中でポインタを扱うコードを書き忘れてしまった場合、問題はすぐに明らかになるから、実際にはそれほど深刻なことにはならないだろう。ところが、デストラクタでポインタの削除を忘れると、目に見える症状が出てこないことが多い。その代わりに、微妙なメモリーリークが生じる。それは癌のようにゆっくりと増殖し、あなたのアドレス空間を蝕み、プログラムを早すぎる死に追い込む。このように、この問題は症状がすぐに現れないので、ポインタメンバをクラスに追加するときは必ず注意が必要である。

ところで、ヌルポインタを削除するのは、いつでも安全である(何も起こらない)。だから、コンストラクタや代入演算子や他のメンバ関数を、そのクラスのポインタメンバがすべて有効なメモリを指すか、またはヌルとなるように書いておけば、デストラクタの中では、ポインタに対して new が使われたかどうかいちいち気にせずに、気楽に delete を連発してかまわない。

もっとも、この項を無条件に強制する理由はまったくない。new で初期化しなかったポインタに対して delete を使おうとは誰も思わないし、元々は外から渡されたポインタを削除したいというケースもほとんどないだろう。言い換えると、ポインタに new を使ったのがあなたのクラスのメンバでなければ、あなたのクラスのデストラクタは、そのポインタを delete しないのが普通である。

7 項 メモリ不足に備えよう

operator new は、要求されたメモリを割り当てられないときは例外を送出する(以前は 0 を返した。一部の古いコンパイラは、今でもそうしている。新しいコンパイラでも、必要ならば 0 を返すようにできるが、この件については本項の最後に述べることにする)。メモリ不足による例外には対処しな

なければならない。そうするのが唯一正しい行いだということは、あなたの胸の奥底にある良心がよく知っているはずだ。しかし同時に、実際にそうするのはたいへんだという事実も、あなたははっきりと認識しているだろう。その結果、あなたは例外処理を、ときどき省略することになる。ときどきではなくて、常に、かもしれない。しかしあなたは不安を抱き続けることになる。`new` が本当に例外を送出したらどうしよう。

この事態に対処するには、どうすればいいか。昔のやり方に戻ってプリプロセッサを使うという手もあるじゃないか、とお思いかもしれない。たとえば C 言語の常套句としては、メモリの割り当てを試み、成功したかどうかをチェックする、型に依存しないマクロを定義するというものがある。C 言語の場合、そういうマクロは次のように書くことができるだろう。

```
#define NEW(PTR, TYPE)          \
try { (PTR)= new TYPE; }      \
catch (std::bad_alloc&) { assert(0); }
```

(「ちょっと待った。その `std::bad_alloc` ってのは?」`bad_alloc` というのは、operator `new` がメモリ割り当ての要求に応えられなかったとき送出する例外で、`std` は `bad_alloc` が定義されている名前空間 (28 項) の名称だ。「なるほど。で、`assert` ってのは?」C の標準インクルードファイルに `<assert.h>` というのがあるけれど(名前空間にうるさい C++ の場合は、`<cassert>` がこれに相当する。49 項) これを見れば、`assert` が単なるマクロであることがわかる。このマクロは渡された式が 0 以外の値を取ることを確認し、もし 0 だった場合はエラーメッセージを出して、`abort` を呼び出す。)

この `NEW` というマクロは、これ自体は問題なさそうだが、使いものにはならないだろう。そもそも、`new` には無数の使い方があるというのに、このマクロには、それらに対処しようという考えがない。型 `T` の新しいオブジェクトを得る方法には、一般的な構文だけでも 3 種類ある。だから以下に示す 3 つの形式について、それぞれに可能性のある例外を扱う必要があるのだ。

```
new T;
new T(コンストラクタの引数);
new T[サイズ];
```

しかし、問題はもっと複雑である。クライアントは operator `new` の独自のバージョンを(オーバーロードによって)定義できるから、プログラムは `new` を使う構文の形式を何種類でも含むことが可能なのだ。

では、どうすればいいのか。エラー処理を非常にシンプルに行いたいときは、要求したメモリが割り当てられなかったら指定のエラー処理ルーチンが呼び出されるようにしておけばいいだろう。この方法は、「operator `new` が要求を満足させられないときには、例外を送出する前に、クライアントが指定するエラー処理関数(しばしば `new-handler` と呼ばれる)があれば、それ

を呼び出す」という規約に依存している(実際には、operator new が行う処理はもう少し複雑だ。詳しくは8項で説明する)。

クライアントがメモリ不足を処理する関数を指定するには、set_new_handler を呼び出せばいい。これはヘッダ<new>の中で、だいたい次のように書かれているはずだ。

```
typedef void (*new_handler)();
new_handler set_new_handler(new_handler p) throw();
```

ご覧のとおり、new_handler は関数ポインタの typedef であり、その関数は何も受け取らず、何も返さない。そして set_new_handler は、new_handler を受け取り new_handler を返す関数である。

set_new_handler のパラメータは、operator new が要求されたメモリを割り当てられなかったときに呼び出してもらいたい関数へのポインタである。set_new_handler の戻り値は、この set_new_handler が呼び出される前に指定されていた new_handler 関数へのポインタである。

set_new_handler は、次のように使う。

```
// operator new が十分なメモリを割り当てられなかったときは、
// この関数を呼び出す
void noMoreMemory()
{
    cerr << "要求されたメモリを割り当てられません\n";
    abort();
}

int main()
{
    set_new_handler(noMoreMemory);
    int *pBigDataArray = new int[100000000];
    ...
}
```

もし operator new が 100,000,000 個の int のための空間を割り当てることができなければ(たぶん無理だろう)noMoreMemory が呼び出され、プログラムはエラーメッセージを出したあと、アボートする。プログラムの終了方法としては、コアダンプするよりも、いくらかマシである(ところで、エラーメッセージを cerr に書く途中でメモリを動的に割り当てる必要があったら、いったいどうなるだろうか.....)。

operator new が、1つのメモリ要求を満足させられなかったときは、new_handler 関数を1回呼び出すのではなく、十分なメモリを見つけられるまで何度も繰り返して呼び出す。この繰り返して呼び出すコードは8項に示すが、今ここでやっている抽象レベルの高い説明では、正しく設計された new-handler は、次のうちの1つを行わなければならない、という結論を示すだけで十分だろう。

もっと多くのメモリを使えるようにする。

そうすれば、operator new が次にメモリを割り当てようとしたときには成功するかもしれない。この手法を実装するには、たとえばプログラムの起動時に予備として大きなメモリブロックを割り当てておき、new-handler が初めて呼び出されたときに、そのブロックを解放するという方法がある。予備のメモリを解放するときには、ユーザーにメモリが少ないことを警告し、もっとメモリを追加しないと今後の要求が失敗する恐れがあることを知らせるメッセージを出すことが多い。

別の new-handler をインストールする。

現在の new-handler では、もうメモリを割り当てられないときでも、もっと資源を持っている別の new-handler があるかもしれない。それがわかっている場合、現在の new-handler は(set_new_handler によって)別の new-handler をインストールしてあとを任せることができる。operator new が次に new-handler 関数を呼び出すときには、最後にインストールされたものが呼び出される(この手法の変形として、new-handler が自分のふるまいを変更して、次に呼び出されたときは別の処理を行うというものもある。それには、new-handler が自分のふるまいを変えるために、静的データかグローバルデータを変更するという方法がある)。

インストールされている new-handler を外す。

つまり set_new_handler にヌルポインタを渡す。new-handler がインストールされていないと、operator new は、メモリ割り当てが失敗したとき、std::bad_alloc 型の例外を送出するようになる。

例外を送出する。

std::bad_alloc 型の例外か、std::bad_alloc から派生した型の例外を送出する。そのような例外は、operator new によって捕獲されないの、メモリを要求したところまで伝播されるだろう(他の種類の例外を送出するのは、operator new の例外に関する規約に違反する。違反が起きたときはデフォルトで abort が呼び出されてしまうので、new-handler が例外を送出するときは、必ず std::bad_alloc の階層構造に属した例外を送出するように注意すべきだ)。

リターンしない。

典型的には、abort または exit を呼び出して終了する。どちらも標準 C ライブラリに含まれている(だから、標準 C++ ライブラリにもある。49 項)。

以上のものから選択すれば、new-handler 関数の実装に、ある程度の柔軟性を持たせられるだろう。

割り当てるオブジェクトのクラスによっては、メモリ割り当ての失敗を別の方法で扱いたい場合もあるだろう。

```

class X {
public:
    static void outOfMemory();
    ...
}

class Y {
public:
    static void outOfMemory();
    ...
};

X* p1 = new X;    // もし割り当てが失敗したら
                 // X::outOfMemory を呼び出す
Y* p2 = new Y;    // もし割り当てが失敗したら
                 // Y::outOfMemory を呼び出す

```

C++はクラスごとの new-handler をサポートしていないが、その必要はない。そういうふるまいは、自分で実装できる。クラスごとに、set_new_handler と operator new の専用バージョンを提供すればいいのだ。クラスの set_new_handler によって、クライアントはそのクラス用の new-handler を指定できる(これは、標準の set_new_handler によって、クライアントがグローバルな new-handler を指定できるのと同じことだ)。また、クラスの operator new によって、クラスオブジェクトのためにメモリを割り当てるときには、グローバルな new-handler ではなくクラス独自の new-handler が使われることが保証される。

クラス X で、メモリ割り当てが失敗したときの処理を行いたいと仮定しよう。operator new が X 型のオブジェクトのためのメモリを割り当てられないときに呼び出される関数を、動的に管理しなければならないから、このクラス用の new-handler 関数を指す、new_handler 型の static メンバを宣言することになるだろう。すると、クラス X は以下ようになる。

```

class X {
public:
    static new_handler set_new_handler(new_handler p);
    static void * operator new(size_t size);
private:
    static new_handler currentHandler;
};

```

クラスの static メンバは、クラス定義とは別に、その外側で定義しなければならない。static オブジェクトはデフォルトで 0 に初期化されるので、これを利用し、X::currentHandler は初期化なしで定義する。

```

new_handler X::currentHandler;    // currentHandler には、
                                 // デフォルトで 0(ヌル)が
                                 // セットされる

```

クラス X の set_new_handler 関数は、渡されたポインタをすべて保存す

る。また、その呼び出しの1つ前に保存したポインタを返す。これは、標準の `set_new_handler` が行うことと、まったく同じである。

```
new_handler X::set_new_handler(new_handler p)
{
    new_handler oldHandler = currentHandler;
    currentHandler = p;
    return oldHandler;
}
```

最後に、`X` の operator `new` は次のことを行う。

1. `X` のエラー処理関数を引数として標準の `set_new_handler` を呼び出す。これによって `X` の new-handler がグローバルな new-handler としてインストールされる。以下に示すコードで、“`::`”を使って `std` スコープを明示的に参照している点に注目されたい。標準の `set_new_handler` は、このスコープの中にある。
2. グローバルな operator `new` の呼び出しによって、要求されたメモリを実際に割り当てる。この割り当て要求が最初の試みで失敗したら、グローバルな operator `new` は、先ほどグローバル new-handler 関数としてインストールされたばかりの、`X` の new-handler を呼び出すだろう。そして、グローバル operator `new` が要求されたメモリを割り当てる方法を、結局見出せなかったら、`std::bad_alloc` の例外を送出するが、これは `X` の operator `new` によって捕獲される。すると、`X` の operator `new` は、グローバル new-handler を元々インストールされていたものに戻し、例外を伝播することによってリターンする。
3. グローバル operator `new` が、`X` 型のオブジェクトのために十分なメモリを割り当てるのに成功したとすると、`X` の operator `new` は、やはり標準の `set_new_handler` を呼び出して、グローバルなエラー処理関数を元々インストールされていたものに戻す。そして、割り当てたメモリへのポインタを返す。

以上のことを C++ で表現すると、次のようになる。

```
void * X::operator new(size_t size)
{
    new_handler globalHandler =           // Xのハンドラを
        std::set_new_handler(currentHandler); // インストール
    void *memory;
    try {
        memory = ::operator new(size);     // メモリ割り当てを
                                           // 試みる
    }
    catch (std::bad_alloc&) {             // ハンドラを
        std::set_new_handler(globalHandler); // 元に戻し
        throw;                             // 例外を伝播する
    }
}
```

```

std::set_new_handler(globalHandler);    // ハンドラを
                                        // 元に戻す
return memory;
}

```

クラス X のクライアントは、new-handler を以下のように使用する。

```

void noMoreMemory();                // X のオブジェクトに対する
                                    // メモリ割り当てが失敗した
                                    // とき呼び出される関数を
                                    // 宣言する
X::set_new_handler(noMoreMemory);  // X の new-handler 関数として
                                    // noMoreMemory をセットする
X *px1 = new X;                    // もしメモリ割り当てが失敗したら
                                    // noMoreMemory が呼び出される
string *ps = new string;           // もしメモリ割り当てが失敗したら
                                    // (もしあれば)グローバルな
                                    // new-handler 関数が呼び出される
X::set_new_handler(0);              // X 固有の new-handler 関数を
                                    // 設定解除(ヌルポインタにする)
X *px2 = new X;                    // もしメモリ割り当てが失敗したら
                                    // 即座に例外を送出する(クラス
                                    // X にエラー処理関数が設定されて
                                    // いないため)

```

この方式を実装するコードは、クラスが違って共通に使えるということにお気づきだろうか。そうだとしたら他の場所でも再利用したくなるのが当然だ。41 項で説明するように、再利用可能なコードを作るには、継承とテンプレートという 2 つの方法がある。けれども、この場合は、この両方を組み合わせると望みの結果が得られる。

必要なのは、「混入スタイル」の基底クラスだ。混入スタイル (mixin-style) の基底クラスとは、ある特別な能力 (この場合なら、クラス独自の new-handler を設定する能力) だけを派生クラスが継承できるように設計された基底クラスのことだ。次に、その基底クラスをテンプレートにする。この混入型設計のうち、基底クラスは、派生クラスが必要とする set_new_handler と operator new 関数を継承できるようにする役割を果たし、同じくテンプレートは、これを継承する各クラスが、データメンバとして、それぞれ異なった currentHandler を持てるようにする。なんだか複雑そうだが、できあがるコードは見覚えのあるものだから、安心してほしい。実際、本当に違っているのは、これを必要とするどんなクラスからも再利用できるようになったという点だけである。

```

template<class T>                  // クラス固有の set_new_handler
class NewHandlerSupport {        // をサポートするための
public:                            // 「混入スタイル」の基底クラス
    static new_handler set_new_handler(new_handler p);
    static void * operator new(size_t size);
private:
    static new_handler currentHandler;
};

```

```

template<class T>
new_handler NewHandlerSupport<T>::set_new_handler(new_handler p)
{
    new_handler oldHandler = currentHandler;
    currentHandler = p;
    return oldHandler;
}

template<class T>
void * NewHandlerSupport<T>::operator new(size_t size)
{
    new_handler globalHandler =
        std::set_new_handler(currentHandler);
    void *memory;
    try {
        memory = ::operator new(size);
    }
    catch (std::bad_alloc) {
        std::set_new_handler(globalHandler);
        throw;
    }
    std::set_new_handler(globalHandler);
    return memory;
}

// それぞれの currentHandler に 0 をセットする
template<class T>
new_handler NewHandlerSupport<T>::currentHandler;

```

このクラステンプレートを使えば、クラス X に `set_new_handler` のサポートを追加するのは簡単だ。X は、`NewHandlerSupport<X>` を継承するだけでいい。

```

class X: public NewHandlerSupport<X> { // 混入基底クラスの
    // テンプレートから
    // 継承している点に注意
    ... // 前と同じ。ただし、set_new_handler と
}; // operator new の宣言はない

```

X のクライアントには、舞台裏のすべての処理は見えないままであり、クライアントの以前のコードはそのまま使える。クライアントには、普通は舞台裏を見せたくないものだから、そのほうがありがたい。

`set_new_handler` を使うのは、メモリ不足の可能性に対処するには手軽で簡単な方法である。new を使う部分をすべて try ブロックで囲むよりは、そのほうがずっと魅力的なのは確かだ。そればかりか、`NewHandlerSupport` のようなテンプレートを使えば、クラス固有の new-handler を、それを必要とするどのクラスにも追加することができる。ただし、混入スタイルの継承について説明すると、どうしても多重継承の話になってしまう。多重継承は「滑り落ちやすい坂道」なので、そちらに出かける前には、ぜひとも 43 項を読んでおくべきだ。

1993年までのC++では、operator newはメモリ要求を満足させられなかったら0を返すことになっていた。現在は、std::bad_alloc例外を送出するのがoperator newの正しいふるまいだが、コンパイラがこの新しい仕様をサポートする前に書かれたC++プログラムの数は非常に多い。すでに確立している「0かどうかのテスト」を使った膨大な量のコードを破棄するわけにはいかないので、C++の標準化委員会は、伝統的な「失敗のときは0を返す」ふるまいを提供する、operator newの(そしてoperator new[]の)もう1つの形式を提供した(8項)。これらが「送出なし(nothrow)」形式と呼ばれているのは、要するに決して送出を行わないからである。この形式では、newの呼び出しのところで(標準のヘッダファイル<new>で定義されている)nothrowオブジェクトを使うようにする。

```
class Widget { ... };
Widget *pw1 = new Widget;           // メモリ割り当てに失敗すると
                                    // std::bad_allocを送出する

if (pw1 == 0) ...                   // このテストには失敗するはず
    Widget *pw2 =
        new (nothrow) Widget;       // 割り当てに失敗すると
                                    // 0を返す

if (pw2 == 0) ...                   // このテストは成功するかもしれない
```

通常の(例外を送出する)newを使うにしても、「送出なし」のnewを使うにしても、メモリ割り当ての失敗に対処できるよう、あらかじめ準備することが重要である。それには、どちらの形式のnewにも対処できる、set_new_handlerを利用するのが最も簡単である。

訳者追記：著者の正誤表によれば、例外は値ではなくリファレンスで受け取るのが正しい。これは効率のためである。詳しくは『*More Effective C++*』のItem 13を参照のこと。

8項 operator newとoperator deleteを書くときは規約を守ろう

operator newの作成を自分で行うことに決めたら(そうする理由は10項で説明するが)、あなたの関数のふるまいがデフォルトのoperator newのそれと矛盾しないようにすることが重要だ。具体的に言うと、正しい戻り値を返すこと、メモリが不足したときはエラー処理関数を呼び出すこと(7項)、サイズ0のメモリ要求にも対応できるように準備しておくことである。また、「通常の」形のnewを間違えて掩蔽しないように注意する必要もあるが、それは9項のテーマだ。

戻り値の問題は簡単だ。要求されたメモリを供給できるときは、そのメモリへのポインタを返せばいい。メモリが足りなければ、7項のルールに従い、`std::bad_alloc` 型の例外を送出する。

もっとも、これは実はそれほど簡単ではない。operator new は、メモリ割り当てに失敗するたびにエラー処理関数を呼び出して、メモリ割り当てを何度も繰り返そうとする。エラー処理関数がメモリの一部を解放してくれることを当てにしているからだ。operator new が例外を送出するのは、エラー処理関数へのポインタがヌルのときだけである。

さらに標準 C++ によれば、operator new は、たとえ 0 バイトが要求されたときでも正規のポインタを返さなければならない(信じてもらえないかもしれないが、この奇妙なふるまいを要求すると、C++ 言語の他の部分を単純化できるというメリットがあるのだ)。そういうわけで、メンバではない operator new は、疑似コードで書くと次のようになる。

```
void * operator new(size_t size) // あなたの operator new には、ほかに
{                               // パラメータがあってもいい
    if (size == 0) {             // 0 バイトの要求は
        size = 1;                // 1 バイトの要求として扱う

        while (1) {
            size バイトのメモリを割り当てようと試みる;

            if (割り当てに成功した)
                return (そのメモリを指すポインタ);

            // 割り当てに失敗した: 現在の
            // エラー処理関数を探す( 7 項)
            new_handler globalHandler = set_new_handler(0);
            set_new_handler(globalHandler);

            if (globalHandler) (*globalHandler)();
            else throw std::bad_alloc();
        }
    }
}
```

0 バイトの要求を、1 バイト要求されたときと同じように扱うトリックは、ずいぶん狡猾に見えるだろうが、この方法はシンプルだし、違法でもなく、ちゃんと使える。だいたい、0 バイトの要求など、めったにあるものではない。そうだろう？

それから、この疑似コードにはエラー処理関数へのポインタとしてヌルを設定し、すぐまた元の値に戻している部分があるが、ここも疑いの目で見られるかもしれない。だが、エラー処理ルーチンへのポインタを直接取得するにはほかに方法がないので、それを知るためには `set_new_handler` を呼び出さなければならない。みっともないかもしれないが効果的だ。

operator new に永久ループが含まれることは7項で述べたが、上記のコードには、そのループが明確に示されている。while(1) のループは、いつま

でも無条件に繰り返される。このループから脱出するには、メモリの割り当てに成功するか、あるいは new-handling 関数が(7項で説明したように)次のどれかを行うしかない。つまり、入手できるメモリを増やすか、別の new-handler をインストールするか、インストールされている new-handler を外すか、std::bad_alloc またはそれから派生した例外を送出するか、または、リターンしないかである。new-handler が、このうちのどれかを行わなければならない理由は、もうはっきりしているだろう。そうしなければ operator new の内部のループが永遠に終了しないのである。

operator new がサブクラスによって継承されるということ認識していない人は多い。だから、次に述べるような、まことに興味深い、複雑な事態が生じることがある。上記の operator new の疑似コードでは、この関数は(sizeが0の場合を除いて)size バイトのメモリを割り当てようとする。size は、この関数に渡される引数なのだから、当然のことだ。しかし、クラス専用の operator new の場合、たいがい(10項にあるものもそうだが)特定のクラスのために設計されるのであり、不特定のクラスや、そのサブクラスのために設計されるわけではない。つまり、クラス X の new operator を作るとしたら、その関数のふるまいは、ほとんど常に sizeof(X) のサイズを持つオブジェクトに合わせてチューニングされるはずだ。それより大きいものも少ないものも無視される。ところが継承が行われると、基底クラスの operator new が、派生クラスのオブジェクトにメモリを割り当てる目的で呼び出される可能性がある。

```
class Base {
public:
    static void * operator new(size_t size);
    ...
};

class Derived: public Base    // 派生クラス Derived が
{ ... };                    // operator new を宣言しないと.....

Derived *p = new Derived;    // Base:operator new が呼び出される
```

もし Base クラス独自の operator new が、この事態に対処するように設計されていないとしたら(そういう設計は、あまり期待できない)、「間違っただ」量のメモリが要求されたら標準の operator new に任せてしまうという方法がベストだろう。

```
void * Base::operator new(size_t size)
{
    if (size != sizeof(Base))    // サイズが「間違っ」たら
        return ::operator new(size);    // 標準の operator new に
    ...                            // 要求を処理させる
    ...                            // そうでなければ、
    ...                            // ここで要求を処理する
}
```

「ちょっと待った!」という叫び声が聞こえる。「病的だけど可能性がないこともないというサイズが0のケースをチェックするのを忘れてるぞ!」いやいや、別に忘れてはいない。長々と指摘していただかなくても、そのテストは、ちゃんと存在する。それはサイズが `sizeof(Base)` に等しいかどうかのテストに組み込まれたのだ。C++の標準は、いろいろと謎めいた機構によって支えられているのだが、その1つに、すべての独立したクラスのサイズは0であってはならない、という決まりごとがある。この定義により、`sizeof(Base)` が0になることはあり得ない(たとえメンバが1個もなくとも)。だから、もし `size` が0ならば、その要求は `::operator new` に渡される。そうなったら、その要求を妥当な方法で扱うのは、こちらの責任ではないということになる。

配列のためのメモリ割り当てをクラス単位で制御したければ、`operator new` の配列バージョンである `operator new[]` を実装しなければならない(この関数は、普通“array new”と呼ばれる。“operator new[]”をどう発音したらいいかというのは難しい問題だ)。`operator new[]` を書くことに決めたら、行うのは生のメモリを割り当てることだけだ、ということをお忘れなく。配列の中の、まだ存在していないオブジェクトについては、何も行うことはできない。実際、それぞれのオブジェクトの大きさも不明なのだから、配列にオブジェクトがいくつ入るのかもわからない。結局のところ、基底クラスの `operator new[]` は、(継承を介して)派生クラスのオブジェクトの配列のためにメモリを割り当てる目的で呼び出されるかもしれない、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よりも大きいのが普通である。したがって、`Base::operator new[]` の中からは、配列の中に入るオブジェクトのサイズが `sizeof(Base)` だと決め付けることはできず、それゆえに、配列の中のオブジェクトの個数が必ず(要求バイト数)/`sizeof(Base)` になると決め付けることもできない。

`operator new`(と `operator new[]`)を書くときに従わなければならない規約については、以上で十分だろう。`operator delete`(と、配列のための `operator delete[]`)に関しては、話はずっとシンプルになる。忘れてならないことは、ただ1つ、「ヌルポインタの削除は常に安全だ」とC++が保証していることだ。この保証を無にはならない。メンバではない `operator delete` の疑似コードを以下に示す。

```
void operator delete(void *rawMemory)
{
    if (rawMemory == 0) return; // 削除の対象がヌルポインタだったら
                                // 何もしない

    rawMemoryが指しているメモリを解放する;

    return;
}
```

同じ関数のメンバ版も、同様にシンプルだが、ただし削除される対象のサ

イズをチェックしなければならない。クラス固有の operator new が「間違っ
た」サイズのメモリ要求を ::operator new に渡しているとしたら、「間違っ
たサイズ」の削除要求は、 ::operator delete に渡さなければならない。

```
class Base {
public:
    static void * operator new(size_t size);
    static void operator delete(void *rawMemory, size_t size);
    ...
};

void Base::operator delete(void *rawMemory, size_t size)
{
    if (rawMemory == 0) return; //ヌルポインタをチェック
    if (size != sizeof(Base)) { //もしサイズが「間違っていたら
        ::operator delete(rawMemory); //標準の operator delete に
    } //要求の処理をまかせる
    return;
}

rawMemoryが指しているメモリを解放する;

return;
}
```

このように、operator new と operator delete(と、それらの配列版)に
関する規約は、とくに面倒なものではないが、それらに従うことは重要であ
る。あなたのメモリ割り当てルーチンは、new-handler 関数をサポートし、
サイズ0の要求を正しく扱っていれば十分だし、メモリ解放ルーチンは、ヌ
ルポインタにさえ対処していればいい。メンバ版の関数には継承のサポート
を追加して、ほ～ら、これでもう完成だ。

9 項 「普通の」形式のnewを掩蔽しないように注意しよう

内側のスコープで名前を宣言すると、外側のスコープにある同じ名前は掩
蔽される(見えなくなる)。だから、f という関数がグローバルスコープとク
ラススコープの両方にあるときは、メンバ関数によってグローバル関数が掩
蔽される。

```
void f(); // グローバル関数

class X {
public:
    void f(); // メンバ関数
};

X x;

f(); // グローバル関数 f を呼び出す
```

```
X.f();           // X::f を呼び出す
```

これは別に驚くべきことではないし、グローバル関数とメンバ関数は普通は別の構文で呼び出されるから、混乱が起きることもほとんどないだろう。けれどもこのクラスに、パラメータ数を増やした `operator new` を追加すると、びっくりするような結果が生じることが多い。

```
class X {
public:
    void f();

    // new-handler 関数を指定できる
    // operator new
    static void * operator new(size_t size, new_handler p);
};

void specialErrorHandler();           // どこかで定義されている

X *px1 = new (specialErrorHandler) X; // X::operator new を呼び出す

X *px2 = new X;                       // エラー!
```

“`operator new`”という名前の関数をクラスの内側で宣言すると、「普通の」形式の `new` へのアクセスを、そのつもりがなくても妨害してしまう。なぜそうなるのかは、50 項で説明するので、ここでは、この問題を回避する方法を考えることにしよう。

「普通の」呼び出し方をサポートする、クラス固有の `operator new` を書くのも解決策の 1 つである。グローバル版と同じことをするだけなら、インライン関数の中に効率的かつエレガントにカプセル化することができる。

```
class X {
public:
    void f();

    static void * operator new(size_t size, new_handler p);

    static void * operator new(size_t size)
    { return ::operator new(size); }
};

X *px1 =new (specialErrorHandler) X; // X::operator new( size_t,
                                     // new_handler )を呼び出す
X* px2 = new X;                     // X::operator new( size_t )
                                     // を呼び出す
```

もう 1 つの対策は、`operator new` に追加するパラメータのそれぞれに、デフォルトパラメータ値(24 項)を提供することである。

```
class X {
public:
    void f();
```

```

static
    void * operator new(size_t size,          // p にデフォルト値
                       new_handler p = 0); // があることに注意
};

X *px1 = new (specialErrorHandler) X;      // 問題なし

X* px2 = new X;                            // これも問題なし

```

どちらにしても、あとで「普通の」形式の new のふるまいをカスタマイズすることになったときは、関数を書き直すだけで済む。これら呼び出すプログラムは、再リンクによって、カスタマイズされたふるまいを自動的に取得できる。

10 項 operator new を書くなら、operator delete も書こう

ここで一歩後退して、基本的な事項に戻ろう。operator new や operator delete の自分のバージョンは、そもそも何のために書くのだろうか？

たいていの場合、その答えは「効率」だ。operator new と operator delete のデフォルトのバージョンは、汎用的な使い方には最適だけれども、柔軟性に富んでいるということはすなわち、より限定されたコンテキストで性能を向上させる余地を必然的に含んでいるということでもある。それがとくに顕著なのは、小さいオブジェクトを数多く動的に割り当てるアプリケーションだ。

一例として、飛行機 airplanes を表現するクラスを考えてみよう。Airplane クラスに含まれるのは、飛行機オブジェクトの実際の表現へのポインタだけだ(このテクニックは 34 項で説明する)。

```

class AirplaneRep { ... };          // Airplane オブジェクトの表現

class Airplane {
public:
    ...
private:
    AirplaneRep *rep;              // 表現へのポインタ
};

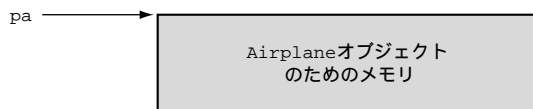
```

Airplane オブジェクトは、決して大きいものではない。ポインタが 1 つ入っただけだ(14 項で説明するように、もし Airplane クラスが仮想関数を宣言していると、暗黙のうちに第二のポインタを含むことがある)。しかし、Airplane オブジェクトを operator new の呼び出しによってアロケートすると、おそらくポインタ 1 個(あるいは 2 個)を格納するのに必要なメモリよりも、多くのメモリが割り当てられてしまう。この理不尽とも思えるようなふるまいには、operator new と operator delete が相互に通信しなければならないという事情が関係している。

operator new のデフォルトバージョンは汎用的なアロケータなので、どんなサイズのブロックでもアロケートできるように設計しなければならない。同様に、operator delete のデフォルトバージョンは、operator new が割り当てたブロックは、どんなサイズでもデアロケートできなければならない。operator delete がメモリをどのくらいデアロケートしたらいいかを知るには、そもそも new がどれだけのメモリを割り当てたかを、何かの方法で知る必要がある。operator new が operator delete に割り当てたメモリの量を知らせるには、new が返すメモリの直前の位置に、割り当てたブロックのサイズを示すデータを付加するのが、一般的な方法だ。つまり、

```
Airplane *pa = new Airplane;
```

のように書いたとき、以下のようなメモリブロックが得られるとは限らず、



下記のようなメモリブロックが割り当てられることが多いのである。



Airplane クラスのようにオブジェクトが小さい場合、この管理用の追加データ領域は(とくに仮想関数を含まないクラスの場合)動的に割り当てるオブジェクトがそれぞれ必要とするメモリの2倍以上になることさえある。

メモリが貴重な資源であるような環境をターゲットとしてソフトウェアを開発する場合、このように浪費的なアロケーションは許されないだろう。Airplane クラスのために自分で operator new を書くことにすれば、すべての Airplane オブジェクトが同じサイズであるという事実を活用でき、割り当てたメモリブロックのそれぞれについて管理情報を持つ必要はなくなる。

クラス固有の operator new の実装方法としては、デフォルトの operator new から生のメモリブロックを、大きなサイズで何個か要求する方法がある。1個のメモリブロックは、かなりの数の Airplane オブジェクトを格納できる大きさとする。Airplane オブジェクト自身のためのメモリチャンク[ひとつかたまりのメモリ]は、これらの大きなブロックから割り当てる。現在使用されていないチャンクは、チャンクの連結リスト(空きリスト)として構造化し、将来 Airplane が使えるようにしておく。こう書くと、すべてのオブジェクトに(リストを連結するための)next フィールドを持たせなければな

らず、そのオーバーヘッドがかかるのではないかと、思われるかもしれないが、その必要はない。Airplane オブジェクトとして使用中のメモリチャンクにだけ必要な rep(表現)フィールドは、next ポインタを格納する場所としても使える(なぜなら、そのポインタが必要なのは、Airplane オブジェクトとして使用されていないチャンクだけだからだ)。2 役を兼ね備えるデータフィールドには普通そうするように、ここは union を使えばいい。

この設計を実現するには、カスタム仕様のメモリ管理をサポートできるように Airplane の定義を変更しなければならない。それは次のように行う。

```
class Airplane {           // 変更後のクラス - カスタム仕様の
public:                   // メモリ管理をサポートする
    static void * operator new(size_t size);
    ...
private:
    union {
        AirplaneRep *rep;    // 使用中のオブジェクト用
        Airplane *next;     // 空きリスト上のオブジェクト用
    };

    // このクラス固有の定数( 1 項 )により、大きなメモリブロックを
    // Airplane オブジェクト何個分の大きさにするかを指定する。
    // ( 定数を初期化する )
    static const int BLOCK_SIZE;
    static Airplane *headOfFreeList;
};
```

以上で、operator new の宣言、rep と next のフィールドを同じメモリで兼用する union、ブロックを割り当てるサイズを決めるクラス固有の定数、空きリストの先頭を管理するための static ポインタが追加された。この、最後のポインタメンバを static とするのは重要なことだ。空きリストは Airplane のオブジェクトごとに 1 個あるのではなく、**クラス全体で 1 個の空きリストを管理するからである。**

次の仕事は、新しい operator new を書くことだ。

```
void * Airplane::operator new(size_t size)
{
    // 「間違った」サイズの要求は ::operator new( ) に転送する
    // 詳しくは 8 項を参照

    if (size != sizeof(Airplane))
        return ::operator new(size);

    Airplane *p =           // p は、空きリストの先頭を指す
        headOfFreeList;    // ポインタになる

    // もし p が有効ならば、単にリストの先頭を
    // 空きリストの次の要素に移すだけ
    if (p)
        headOfFreeList = p->next;
    else {
```

```

// 空きリストは空( から )である。BLOCK_SIZE 個の Airplane
// オブジェクトを格納できる大きさのメモリブロックを割り当てる
Airplane *newBlock =
    static_cast<Airplane*> ( (::operator new(BLOCK_SIZE *
                                        sizeof(Airplane)) ) );

// メモリチャンクを連結して新しい空きリストを作る
// 0 番目の要素を飛ばしているのは、operator new の呼び出し側に
// 返さなければならないから
for (int i = 1; i < BLOCK_SIZE-i; ++i)
    newBlock[i].next = &newBlock[i+1];

// ヌルポインタで連結リストを終結する
newBlock[BLOCK_SIZE-1].next = 0;

// p をリストの先頭に、headOfFreeList を、そのすぐあとに
// 続くチャンクに、それぞれ設定する
p = newBlock;
headOfFreeList = &newBlock[1];
}
return p;
}

```

8 項を読まれた方はご存知のとおり、operator new がメモリ要求を満足させられなかったときは、new-handler 関数と例外を伴う儀式めいた一連のステップを実行することが求められている。上記のコードには、そのようなステップが含まれていない。その理由は、この operator new が、自分が管理するメモリをすべて ::operator new から得ているからである。ということはすなわち、この operator new が失敗するのは ::operator new が失敗するときだけに限られるということになる。けれども、::operator new が失敗したら、::operator new 自身が new-handler 関数(と、おそらく最終的には例外の送出)を伴う儀式を行うわけだから、Airplane の operator new が同じことを行う必要はない。言い換えると、new-handler 関連のふるまいは、存在しないわけではなく、::operator new の中に隠されていて見えないのである。

これで operator new ができたから、あとは Airplane の static データメンバを定義するだけだ。

```

Airplane *Airplane::headOfFreeList; // これらの定義は
// ヘッダファイルではなく
const int Airplane::BLOCK_SIZE = 512; // 実装ファイルに
// 入れる

```

static メンバはデフォルトで 0 に初期化されるので、headOfFreeList をヌルポインタに初期化する必要はない。BLOCK_SIZE の値は、もちろん、::operator new から獲得する各メモリブロックのサイズを決定する。

このバージョンの operator new は、何の問題もなく動作する。デフォルトの operator new を使う場合と比べて、Airplane オブジェクトのために

使われるメモリが少ないだけでなく、たぶん高速になるはずだ(数十倍も速くなる可能性がある)。これは別に驚くべきことではない。汎用バージョンの operator new は、さまざまなサイズの要求に対処しなければならないし、内部および外部のフラグメンテーション[管理メモリの断片化]も処理しなければならない。それに比べて、このバージョンの operator new は、連結リストの中の2、3のポインタを操作するだけだ。柔軟性を求められなければ、高速化は簡単である。

さて、ようやく operator delete について議論できる段階に到達した。読者はもう忘れてしまったかもしれないが、本項は operator delete についての項である。ここまで書いてきた今の時点で、Airplane クラスは operator new を宣言しているが、operator delete は宣言していない。そこで、クライアントが次のように書いたとしたらどうだろうか。これはすばらしく理に適っているとは言わないまでも、何でもないように見える。

```
Airplane *pa = new Airplane;    // Airplane:operator new を呼び出す
...
delete pa;                      // ::operator delete を呼び出す
```

しかし、このコードを読みながら耳を澄ませば、墜落して炎上する飛行機の轟音や、それを知ったプログラマたちが泣き叫ぶ声が聞こえてくるだろう。何が悪かったのか。Airplane で定義されている operator new は、まったくヘッダ情報を持たないメモリへのポインタを返すが、デフォルトの、グローバルな operator delete は、渡されたメモリにヘッダ情報があることを前提としているのだ! これでは大災害を免れない。

この例が示しているのは、次の原則である。operator new と operator delete は、両者の前提が一致するように書かなければならない。もしメモリ割り当てルーチンを自分で書くのであれば、そのメモリを解放するルーチンも必ず書くようにすることだ。

Airplane クラスの問題は次のように解決できる。

```
class Airplane {                // 前と同じだが、operator delete の
public:                          // 宣言が追加された
    ...
    static void operator delete(void *deadObject,
                                size_t size);
};

// operator delete にはメモリチャンクが渡される
// もしそれが「正しい」サイズなら、空きチャンク
// リストの先頭に追加される
void Airplane::operator delete(void *deadObject,
                               size_t size)
{
    if (deadObject == 0) return;    // 8項

    if (size != sizeof(Airplane)) { // 8項
```

```

        ::operator delete(deadObject);
        return;
    }

    Airplane *carcass =
        static_cast<Airplane*>(deadObject);

    carcass->next = headOfFreeList;
    headOfFreeList = carcass;
}

```

operator new では「間違っただ」サイズの要求をグローバルな operator new に回すように注意を払ったのだから(8 項) ここでも同じ注意を払って、そういう「サイズが不適切な」オブジェクトはグローバル版の operator delete に処理させなければならない。さもないと、今まで一生懸命になって防ごうとしていたトラブル、すなわち new と delete の意味上の不一致が、まさに生じてしまう。

ここで注意すべき点が1つ。削除しようとしているオブジェクトが仮想デストラクタのない基底クラスから派生されている場合、C++が operator delete に渡す size_t の値が正しいサイズではないときがある。これだけでも、基底クラスに必ず仮想デストラクタを作る理由になるが、14 項では、もう1つの、たぶんもっと重要な理由について説明する。しかし今のところは、もし基底クラスに仮想デストラクタがないと、operator delete が正しく機能しない場合がある点に、まずは注意していただきたい。

これで万事オーケーと言いたいところだが、読者の眉間に皺が寄っているところを見ると、どうもメモリリークが気になって仕方がないようだ。長年ソフトウェア開発に携わってきた経験から言って(と、読者は言うだろう)見逃すはずがない。Airplane の operator new は大きなメモリブロックを ::operator new から取得しているのに、Airplane の operator delete は、それらのブロックを解放していないではないか*1。メモリリーク! メモリリーク! あなたの頭の中で鳴り響く警報の音が聞こえるようだ。

しかし、落ち着いて私の言うことを聞いて欲しい。**メモリリークはない。**

メモリリークが生じるのは、メモリが割り当てられたあと、そのメモリへのすべてのポインタが失われたときである。ガーベジコレクションや、その他、言語に追加された機構がなければ、そのようなメモリは再利用できなくなる。けれども、この設計にメモリリークはない。なぜなら、メモリへのすべてのポインタが失われることは決してないからだ。大きなメモリブロックは、それぞれ、まず Airplane の大きさのチャンクに分割され、次にそれらのチャンクが空きリストに置かれる。クライアントが Airplane::operator

*1 これは確信を持って書いている。本書の第1版では、この点について触れるのを忘れたので、数多くの読者に不備を咎められたのである。人は過ちを犯すものだということを実証するのに、数千人のブルーリーダーより優れたものはないだろう(...sigh)


```

delete pa;                                // よし。ブロックはまた空だ
                                           // 解放しよう

...                                        // という具合に.....

return 0;
}

```

この小悪魔のようなプログラムを実行すると、プール版の operator new と operator delete との比較は言うに及ばず、デフォルトの operator new と operator delete を使った場合と比べても、ずっと遅くなり、メモリも多く使ってしまうだろう。

もちろん、このような病的なコーディングに対処する方法はあるが、めったにない特別なケースに対処するコーディングを書けば書くほど、あなたのコードは、メモリ管理ルーチンのデフォルトの実装に近づいていってしまう。結局、何の得にもならないのだ。メモリプールは、メモリ管理に関するあらゆるケースに最適というわけではないが、多くの場合は、これで十分である。

実際、あまりにも多くのケースにあてはまるので、メモリプールを別のクラスに再実装するのが面倒になるかもしれない。そこで、あなたは考えるだろう。「固定サイズのメモリアロケータという考えを、再利用できるようにパッケージングする方法が、きっとあるはずだ」と。確かにそれはあるけれども、この項はずいぶん長くなってしまったから、その詳細は(あなたが恐れていたとおり)読者への課題という形にしておこう。

その代わりに、ここでは Pool クラスへの最小限のインターフェイス(18 項)だけ示しておくことにする。ここで、Pool 型の各オブジェクトは、Pool のコンストラクタによって指定されたサイズのオブジェクトのアロケータである。

```

class Pool {
public:
    Pool(size_t n);                // サイズ n のオブジェクトのための
                                  // アロケータを作成する

    void * alloc(size_t n);        // 1 個のオブジェクトに必要なだけの
                                  // メモリを割り当てる。8 項で述べた
                                  // operator new の規約に従う

    void free(void *p, size_t n); // p が指すメモリをプールに返却する
                                  // 8 項で述べた operator delete の
                                  // 規約に従う

    ~Pool();                       // プールのすべてのメモリを
                                  // 解放する
};

```

このクラスを使うと、Pool オブジェクトを作成し、メモリ割り当てと解放の処理を行い、Pool を破壊することができる。Pool オブジェクトを破壊すると、割り当てたすべてのメモリが解放される。つまり、Airplane のメモ

り管理関数に見られたメモリリークまがいのふるまいは、これで防ぐことができる。しかし同時に、もし Pool のデストラクタを早まって(そのメモリを使っているオブジェクトをすべて破壊する前に)呼び出してしまうと、一部のオブジェクトはまだメモリを使い終わっていないのに足元をすくわれてしまう結果になる。その結果ふるまいが不定になる、とは穏やかすぎる表現だろう。

この Pool クラスを使えば、Airplane にカスタムのメモリ管理を追加するのは簡単である。Java プログラマでも楽に書けるくらいだ。

```
class Airplane {
public:

    ... // 通常の Airplane 関数

    static void * operator new(size_t size);
    static void operator delete(void *p, size_t size);

private:
    AirplaneRep *rep; // 表現へのポインタ
    static Pool merePool; // Airplane 用のメモリプール
};

inline void * Airplane::operator new(size_t size)
{ return memPool.alloc(size); }

inline void Airplane::operator delete(void *p, size_t size)
{ memPool.free(p, size); }

// Airplane オブジェクトのための新しいプールを作成
// この部分はクラス実装ファイルに入れる
Pool Airplane::merePool(sizeof(Airplane));
```

このほうが、前に示した設計よりもずっと明快だ。Airplane クラスの中に散らばっていた、飛行機と無関係な詳細はきれいになくなっている。union も、空きリストの先頭も、生のメモリブロックのサイズを決める定数も、みんな消えた、これらはすべて Pool の中に隠されてしまったが、実はそこが本来の居場所だったのである。メモリ管理についての心配は、Pool の著者にまかせてしまえばいい。あなたの仕事は Airplane クラスを正しく動かすことである。

さて、カスタムのメモリ管理ルーチンによってプログラムの性能が向上するのは興味深いことであるし、そういったルーチンを、どうすれば Pool のようなクラスにカプセル化することが可能か、という問題も重要だが、しかし、本項の主題を見失わないようにしよう。重要なポイントは、operator new と operator delete は共同作業をしなければならないということだ。だから、もし operator new を書いたら、必ず operator delete も忘れずに書くことである。