C＋＋言語におけるデータ抽象

Data abstraction in C＋＋

小　林　健一郎

Ken-ichiro KOBAYASHI

（平成９年10月13日受理）

アブストラクト：
C＋＋言語におけるデータ抽象の実現を具体例「行列を表すクラス」で考察する．この例は「一般的なC＋＋教授法」へのひとつのヒントになると考える．

１．導入

C＋＋はCと並んで，パーソナルコンピュータの世界では事実上の標準とまで言えるようになっている．しかし，一方，例えば，科学技術計算の分野ではそれ程普及していないようである．その理由には，FORTRAN等による過去の資産を放棄できないというものに加えて，科学技術計算に携わる研究者のC＋＋への理解不足もあるように思う．また，C＋＋の技法について書かれた多くの書籍は，言語そのものを専門とする研究者やライブラリの製作者の手によるものが多く，これらは非常に優れた教科書であっても，とにかく答えを出すことが至上の研究者の多くには興味のわかないものなのかもしれない．

本論文では科学技術計算を例にC＋＋の長所のひとつ「データ抽象のサポート」を考察する．ただし，分野を科学技術計算とするのは，具体性を持たせるためであり，本論文のテーマは特定の分野に限定されるものではない．著者は，むしろ，多くの分野の研究者との議論を望むものである．

実際，うまく活用すれば，C＋＋は，あらゆる分野で，したがって科学技術の計算においても非常に有効な言語である．C＋＋言語の長所には，

１．コードの保守管理・再利用が容易である．
２．Cと同等以上の実行速度も得られる．
３．Cの書籍が多くあり，C＋＋に利用しやすい．

などが挙げられる．

「コードの保守管理」とは，一度書いたコードのバグを訂正し，また機能を拡張するという意味である．「再利用」とは，一度書いたコードを（ほとんど）変更せずに別のプロジェクトに利用するという意味である．この項目についてはFORTRAN等とＣ＋＋の両方を経験した研究者の多くが賛成すると著者は考える．

「Ｃの書籍が多い」ということに関しては，少し説明が必要かもしれない．まず，Ｃの書籍と書いたのは，入門書のことではなく，主にアルゴリズムの教科書を念頭においてのことである．著者の印象では，いかに専門的な科学技術計算でも，随所で一般的なアルゴリズムの実装が必要になり，そのために多くの書籍が存在することは実際に有用であると思われる．Ｃ＋＋は基本的にＣを含んでいるので，Ｃのコードを流用するのは容易なのである．

では，なぜ，ＣではなくＣ＋＋を推薦するかというと，Ｃでは保守管理・再利用がＣ＋＋に比べてかなり困難だからである．この点では，著者は，ＣもFORTRANもそれ程変わらないと考える．ＣやＣ＋＋を知らない人は，Ｃ＋＋がＣから「生まれた」言語であるという事実から，両者はほとんど変わらないと考える傾向にあるようだが，プログラミングの基本思想においては，Ｃ＋＋とＣは全く異なる言語であり，むしろＣとFORTRANの方が近いということができるだろう．

「Ｃの書籍が多い」であって「Ｃ＋＋の書籍が多い」ではない理由についても述べたい．Ｃ＋＋は内にＣを含んでいるため，大部分のアルゴリズム，すなわち，算法はＣと共通であるということがひとつある．一方，ＣではできないＣ＋＋のアルゴリズム実現法もあるが，その教科書は少ないようである．これについては，そのような教科書を書くのが難しいから，というのがその理由のひとつではないかと著者は推測している．さらに言うと，Ｃ＋＋での特徴的なアルゴリズム実現法というと，それは仮想関数を多用した，いわゆる「オブジェクト指向型」のプログラムということになる．このような教科書の具体例は，「社員リスト」や「文字列クラス」や「簡易コンパイラ」などである．冒頭でも書いたが，このような例に興味を持つ科学技術研究者は，コンピュータそのものの研究者を除くと，あまり多くないのではないだろうか．本論文がわずかでもこの穴を埋められれば幸いである．

ところで，若手研究者の中には，FORTRANよりＣを選ぶ研究者も多くなってきている．（FORTRANよりＣを選ぶ理由は，著者にはわからない．）にもかかわらず，彼らの多くはＣ＋＋は不要と考えているようである．「Ｃでできる事をなぜＣ＋＋にする必要があるのか」「オブジェクト指向が科学技術計算に必要だろうか」というのが彼らの意見・感想である．

はじめの問に関する著者の答は「保守管理・再利用のしやすさを考慮すると，Ｃ＋＋はＣよりはるかに勝っている（と思う）」である．次の問いに関する答は微妙であり，少し説明したい．

オブジェクト指向という概念は，一般に「データ抽象，継承，動的結合」から成り立っている．Ｃ＋＋では，「クラス，継承クラス，仮想関数」がこれらの概念を実現している．ここで，普通，特に強調されるのは，動的結合，あるいは，仮想関数である．これにより，Ｃで必要だった条件分岐（switch文）がかなり簡略化されるのが特徴のひとつであるが，おかげで，Windowsプログラムで「悪夢」と言われたswitch文が一掃されたことは有名である．しかし，科学技術計算ではどうであろうか．条件分岐が決定的でない計算では，そのためだけに新しい言語を学ぶ動機にはならないだろう．また，条件分岐を多用するプログラムでは，仮想関数の多用が実

行速度の低下という科学技術計算にとって致命的な問題を引き起こしてしまうかもしれない．以上の理由から，現段階では，著者は「オブジェクト指向（特に動的結合の部分）は科学技術計算に極めて有効である」と結論しない．（「不要である」という主張はしていないと明言しておきたい．これについては別の場所で議論したい．）

ただ，それでも，「C＋＋は非常に有効な言語である」と主張する．それは，つまり，「C＋＋言語によるプログラミング＝オブジェクト指向プログラミング」では無いという単純な主張でもある．本論文の主題は「オブジェクト指向」の一部である「データ抽象」にある．「オブジェクト指向」という言葉は一種の流行語となり，喧伝された感があるが，当然，万能ではない．「データ抽象」というプログラミング概念は，より単純であり，従って，より広範囲に適用できるものである．「C＋＋ではコードの保守管理・再利用が容易である」という時，その理由は主にC＋＋の「オブジェクト指向性」にある．しかし，「オブジェクト指向性」の一部である「データ抽象」において，既に「保守管理・再利用の容易さ」が存在し，科学技術計算においても極めて有効であると考えるのである．

以下では，上の主張を「行列」という具体例で考えていきたい．これは「初級・中級者にC＋＋の長所を理解させるための一例」でもあり，C＋＋教授法の部分的研究でもある．はじめに書いたが，本論文のテーマ（「データ抽象」）は科学技術計算に限定されるものではない．読者にはCの経験とC＋＋に関する基本的な知識を仮定する．

なお，C＋＋使用を拒否する単純な理由のひとつに「難しい」という声も聞く．本論文とは無関係にも見えるがコメントしておきたい．C＋＋を完全にマスターするにはCを完全にマスターするより時間がかかるだろう．しかし，C＋＋は基本方針さえ理解すれば，簡単な知識でもプログラムが書ける言語なのである．すなわち，とりあえず「動くプログラム」を書けば，それを自分の知識（言語に関する知識と対象に関する知識）の増加に従って改良すること，つまり，保守管理が（何度も繰り返したが）容易な言語なのである．この点を理解せずにC＋＋を難しく教えている教授者がいるとすれば，それこそが問題なのだと考える．

2．データ抽象

人間の行動を考えるときに，「抽象化」という概念は重要である．ここでいう「抽象化」とは極めて日常的な人間の考え方のパターンを指している．

例えば，自動車の運転を考える．自動車は数多くの部品からなり，それらは走行中複雑な役割を果たしている．しかし，運転者が普通に走行している際に必要な知識は，ハンドル，アクセル，ブレーキ,,, といったわずかな部品に関する部分的な知識のみである．もちろん，走行が普通でない場合や，自動車が故障した場合には，より深い知識が必要になるだろう．しかし，普通の走行で必要な知識はわずかなのである．（例えば，ハンドルに関する知識と言っても，ハンドルの生産工程について知っている必要はなく，単に，右に回せば車は右に行き，左に回せば左に行く，という程度の知識である．）このように，事実の詳細を無視し，必要な部分だけに着目することが，ここでいう「抽象化」の意味である．

古いスタイルのプログラムには詳細が並べ立てられていた．例えば，普通の人間が「ａとｂを足して,,,」などと考えるときに，機械語のプログラマはレジストリやメインメモリのことを考えていたのである．いわゆる高級言語は，機械語に比べ高度に抽象的である．例えば，FOR-

TRANなら，加減乗除や簡単な数学関数など普遍的な演算は自動で処理してくれる．これらの演算をCPUやメモリ上でどのように実現するか考えなくて済むということは，プログラマには福音である．繰り返すが，このように（この例ではCPUなどの）詳細を省いた表現，人間が実行したいことを直接的に言い表す表現を「抽象的な表現」と言う．言うまでもないが，ここでは，否定的な意味合いは無い．

FORTRANで特殊な演算を実現したい場合，例えば，商社Aの月別収支を計算したい場合，FORTRANの組み込み関数・演算子を使って新たにそのような計算の仕組みを定義しなければならない．往々にして，こういう計算は，メインプログラムに書き込まず別にサブルーチンとして定義し，そのサブルーチンを呼び出すということが行われる．これによりメインプログラムにはサブルーチンの呼び出しのみが書かれることになるが，サブルーチンの仕事が明らかである場合には，このメインプログラムは，サブルーチンを使わないものよりかなり読み易いものになっているだろう．このようなサブルーチンの定義と使用は，プログラマが行う「抽象化」である．

抽象化されたプログラムは読みやすいものであるが，変更や拡張が容易であるという長所もある．例えば，上の例では，商社Aの月別収支の計算方法が変更された場合でも，プログラマはそのサブルーチンのみを書き換えれば良いからである．

「構造化された機械語」という異名もあるCは，機械語の効率の良さを保ちながら，プログラムをサブルーチン（Cでは関数という）に分割しやすい言語である．Cのこのような構造は「構造化プログラミング」という手法を実現するためのものであるが，Cの「抽象化」へのサポートということもできる．

もちろん，プログラマが「何々をする関数」を定義する際には，Cの技法を使って詳細を記述しなければならない．(この点でプログラマの負担が絶対的に軽減される汎用言語は無いであろう．) しかし，一度詳細を記述（実装）してしまえば，次からはその関数を呼び出すだけで良い．そのような（良い）関数の呼び出しで記述されるプログラムは「抽象的」なのである．

繰り返すが，本論文では「抽象的＝理解し易い，読み易い」「具体的（詳細）＝理解し難い，読み難い」である．「抽象的すぎてわかり難い」などと言うときの「抽象的」とは違うので注意されたい．言うまでもなく，プログラムは詳細な部分(具体的な部分)こそ理解し難いのである．もちろん，詳細（実装）はいずれにせよ必要であるが，一度記述した詳細な部分をメイン部分では人間に理解し易い「言葉」で使うのが，本論文で言う「抽象化されたプログラム」である．当然，関数を使えば無条件に「抽象的」になるのではなく，意図的にそのようにプログラミングする必要がある．

しかし，Cは「手続き」を関数という形で抽象化する一方で，データに関しては，ほとんど組み込み型以上の抽象化をサポートしていない．すなわち，プログラマは常に，組み込みデータ型に加えて，それらの配列や構造体くらいしか使えないのである．

C＋＋はCにクラスを付け加えることで，この問題を解決している．つまり，クラスは「抽象データ型」をプログラマ自身が定義するための道具立てなのである．クラスは一般に変数と関数からなり，それぞれがプライベート（プロテクテッドも含む）とパブリックに分類されている．プライベートな変数や関数はアクセスが制限され，パブリックな変数や関数はどこからでもアクセス可能である．

この仕組みこそが「抽象データ型」の定義に必要な仕組みである．つまり，パブリックな部

分は，そのデータ型に期待される振る舞いを「抽象的」に記述しているのである．そして，プライベートな部分は，隠されるべき詳細である．例えば，クラス「自動車」を考えよう．

```
class Car
{
    詳細
public :
    start();              //自動車を発進させる関数
    stop();               //自動車を止める関数
    turn(int angle);  //自動車の方向をangle度だけ変える関数
    その他
};
```

上で宣言した関数start，stop，turnなどは，このクラスのユーザが普通に期待するか，期待しなくても，容易に理解できるほど抽象的である．実際に，どのように発進させるのか，どのように自動車をとめるのかなどの詳細は，このクラスの普通のユーザは知る必要もないのである．（実際の自動車でエンジンがどのように動いて車輪を回しているか，，，などを理解している人は少ないだろうし，理解する必要も無い．これと同じである．）このような詳細は，これらの関数の定義やプライベートな部分に記述されるだろう．しかし，Carのユーザは，パブリックな部分の「働き」だけ理解すれば良いのである．メインプログラムは

```
void main()
{
    Car theOne ;
    theOne.start();
    theOne.turn(30);
    theOne.stop();
}
```

などとなるだろう．上のメインの部分だけを見て，何をしているか非常にわかり易い．（自動車theOneを生成し，発進させ，30度向きを変え，止めている．）このようなわかり易さは，データ（ここでは自動車）を十分「抽象的」に表現できているから可能なのである．

さて，実際の自動車がエンジンを付け替えたりするように，Carのstartの実装がより良いコードに変更されることもあるだろう．これは，Car::start()の定義の書き換えを意味するが，その他の部分は全く変更しないで良いのである．（正確には，「C＋＋プログラマが意識的にコードを書いていれば，」という条件が付くことは言うまでもない．）このような変更の容易さは「データの抽象性」と密接に結びついているのである．

なお，プライベート宣言により，詳細（実装）を一般ユーザからアクセス不可にしていることが，「データの抽象性」を保証するのに重要であることも指摘しておきたい．ユーザがプライベートであるべきデータを使えるならば（つまり，クラスの定義でプライベート宣言が適切に

なされていなければ)，そのデータは「抽象的なデータ」ではなく，「具体的な詳細を見せているデータ」である．この場合，一般には，プログラムの変更は困難になる．

一方，パブリックな部分は一般にインターフェースと呼ばれる．まとめると，本論文における「抽象データ型」とは「適切なインターフェースと隠蔽された詳細を持つプログラマ定義のデータ型」である，ということができる．

3．行列クラスとデータ抽象

第2節の議論を簡単な行列クラスを作る例で考えたい．地味であるが，科学技術計算にはほぼ必須のデータ型の例である．この例で「抽象データ型」を定義するときに出てくる問題をひとつひとつ具体的に指摘する．

はじめに書いたようにCによるアルゴリズムの教科書は初等的なものからかなり高級なものまで数多く出版されている．このような教科書はアルゴリズムとその実現（実装）の説明が主であるため，Cの持つ問題点には触れないように書かれていることが多い．著者は，Cの特に大きな問題はメモリ管理の難しさにあると考える．

例えば，連立方程式の解法のアルゴリズムなどには，行列（2次元配列）が使われる．その場合，多くの教科書では行列状のデータがプログラム内で，

```
#define N 3

static double a[N][N+1]={{...},{...},{...}};
```

などと与えられる．

しかし，実際のプログラムでは，プログラム実行中に行列の大きさを決め，データを受け取れるようにしたいと考えるだろう．(科学技術計算の専門家には，プログラム中にデータを書き込むことをあまり気にしない人も多いようだが，データを書き換えるたびにコンパイルし直すのはあまり効率の良い方法ではないだろう．)

まず，問題になるのはCでは動的に2次元配列を作るのが難しいということである．上の例でNをユーザから受け取るようにすることはできない．そこで普通行われるのは，malloc系の関数でメモリを確保し，これを行列に割り当てる方法である．ポインタに慣れているCのプログラマなら簡単にできることだが，そうしてできたコードは往々にして読み難く保守管理の難しいものである．また，そのコードを他のプログラムで再利用できるか疑問である．

一方，C++では，行列は比較的容易に実現できる．簡単のため要素がdoubleの行列を考えるが，もっとも簡単なクラスは，doubleの（1次元）配列を利用するものであるだろう．次のようなコードは，おそらく，C++を習いはじめたばかりの初心者でもそれ程苦労せずに理解できるだろう．

```
class Matrix
{
    double data[100];
```

```
    unsigned i_width, j_width;   //行数と列数
public:
    Matrix(int i, int j):i_width(i),j_width(j){}
    unsigned GetIWidth() const { return i_width; }
    unsigned GetJWidth() const { return j_width; }
    void SetData(unsigned i, unsigned j, double x){
        data[i * j_width+j]=x;
    }
    double GetData(unsigned i, unsigend j) const {
        return data[i * j_width+j];
    }
};
```

（intではなくunsignedを使っている点，constの使い方，コンストラクタの定義の仕方など，やや初心者的でないが，説明すれば理解できるだろう．）

Matrixは，data[100]の一部を長さj_widthのi_width個の行に分解し，これを行列に見立てているのである．このクラスを使えば，

```
//iとjをユーザから受け取り,,,
Matrix theOne(i,j);
```

のように行列をプログラム実行時に「定義」することができる．この行列を使って，Cの教科書にあるような簡単な計算を，実行時にデータを与える形で行うことができるのである．上記のような簡単なコードで「動的に」行列を作れるようになるのは驚きである．

もちろん，このコードは実用的でない．i_width * j_widthが100以下の時のみに使えるからである．せめてi_width * j_widthに対するチェックは付けておきたいが，初心者用のコードでもあり，チェックをつけてもあまり実用的にならないという理由でチェックは省略した．（このクラスのユーザにはこの点を警告すべきではあるだろう．）しかし，ここで強調したい点は，Cの教科書にあるアルゴリズムを（制限付きとは言え）より使いやすくするコードが簡単に書けてしまうという点である．

このクラスの実用版への書き換えは後でするとして，「データ抽象」について考察したい．このクラスはプライベートなデータとパブリックなインターフェースがきちんと別れているので，（適正な）「抽象データ」である．まず，dataとi_width, j_widthであるが，これらはプライベートなデータであり，外部からアクセスすることは（普通の方法では）できない．これらのデータにアクセスするためには，パブリックなメンバ関数GetIWidth(), GetJWidth(), SetData(unsigned i, unsigned j, double x), GetData(unsigned i, unsigend j)を使うしかないのである．

実は，初心者には少し難しいが，上のパブリックなメンバ関数，すなわち，インターフェースはある程度の考察から定義されてる．

まず，行列の行数，列数はいろいろな場面で必要となるはずなので，GetIWidth()とGet-

JWidth()を定義した．i_widthとj_widthをプライベートに定義しておいて，それらの値を得るために，わざわざ関数を定義していることを初心者は不思議に思うかもしれない．ポイントはi_width, j_widthを読み出す関数は定義したが，設定する関数は定義していないということである．そもそも行列の行数や列数を普通の計算途中に変えるということは考え難いだろう．また，すでにデータを保持している行列の行数や列数が変更できたとすると，それは非常に危険だと言わざるを得ない．少なくとも科学技術計算に使われる行列の行数や列数は途中で変更されるべきではない，と著者は考えたのである．（もちろん，別の考え方もあるだろうが，今は，そのように考える．）それこそが，i_width, j_widthをプライベートにし，アクセス関数を別に定義した理由である．もし，i_width, j_widthがパブリックなメンバであれば，プログラムの中で変更することが可能である．そして，不用意に変更すれば，プログラムは暴走するであろう．

繰り返しになるが，「普通の（科学技術計算に使われる）行列の行数や列数が途中で変更されることはない」とクラスの製作者が考え，それ故，このクラスはそのように定義されたのである．Cで同じようなことをするのはかなり大変であろう．

なお，蛇足になるが，「変更可能」と「実際に変更すること」は別のことであり，「変更可能でも変更しなければ良いはず」と考えるかもしれない．しかし，気を付けていても「可能なこと」は，「起こり得ること」である．危険は排除するべきである．

さて，実際のデータはdata[100]内に格納されているが，このデータには，読み出し・書き込み両方のアクセス関数（SetData, GetData）を定義した．もちろん，行列の要素そのものは，計算途中で常に，読まれたり変更されたりするからである．dataを（ユーザから）隠蔽し，SetDataとGetDataを定義したメリットは明らかである．これにより，行列ではないdata[100]を行列として使うことができるようになったのである．ここでdata[100]をパブリックにした場合の危険性を繰り返す必要はないだろう．

著者は，このクラスを「データ抽象」の良い例であると考える．Matrixは行列を表しており4つのアクセス関数を持つ「抽象データ型」なのである．

次に，より実用的なMatrixを考えたい．問題は，i_width * j_widthに関する制限であった．もちろん，data[100]の100をより大きな数にすることでは解決できない．この制限をなくすためには，i_widthとj_widthが与えられてからメモリを確保するようにすれば良いのである．このためにはC++にはnewとdeleteが用意されている．

ここで代入演算子やコピーコンストラクタという初心者には難しい問題があるが，ここでは1例を挙げるのみとする．

```
class Matrix{
    double *data;
    unsigned i_width, j_width;
public:
    Matrix(unsigned, unsigned);
    Matrix(const Matrix&);
    ~Matrix();
```

```
    Matrix& operator=(const Matrix&);

    unsigned GetIWidth() const{ return i_width; }
    unsigned GetJWidth() const{ return j_width; }
    void SetData(int i,int j,float x){
        data[i * j_width+j]=x;
    }
    double GetData(int i,int j) const {
        return data[i * j_width+j];
    }
};

Matrix::Matrix(unsigned i, unsigned j)
    :i_width(i),j_width(j),data(new double[i_width * j_width]){}

Matrix::Matrix(const Matrix &m)
    :i_width(m.i_width),j_width(m.j_width)
        ,data(new double[m.i_width * m.j_width])
{
    for(int i=0;i<i_width * j_width;i++)
        data[i]=m.data[i];
}

Matrix::~Matrix(){ delete [] data; }

Matrix& Matrix::operator=(const Matrix &m)
{
    if(&m!=this){
        delete [] data;
        i_width=m.i_width;
        j_width=m.j_width;
        data=new double[i_width * j_width];
        for(int i=0;i<i_width * j_width;i++)
            data[i]=m.data[i];
    }
    return  * this;
}
```

ここで重要なことは，はじめに定義したバージョンのMatrixと全く同じアクセス関数を持つという点である．そもそもMatrixのアクセス関数は熟考の末考えられたもの（とする）なので，

バージョンアップの際にもできる限り変更するべきではない．変更がなければ，以前のバージョンのMatrixを使って動いていたプログラムは新しいバージョンのMatrixでも動くはずである．

この事実は非常に重要なことを表している．

プログラムを組む場合，大きなプログラムであれば，複数のプログラマが手分けをし，プログラムは分割して作られる．例えば，行列を使ったプログラムを作る場合，行列の定義と行列を使う部分を分けて考えることは，大いに有り得るだろう．しかし，行列を使う部分を作る際には，デバッグ用に行列クラスがあらかじめ必要である．もし，行列クラスが完成するまで待つのなら，プログラムを分割して製作しているメリットはほとんど無くなってしまうだろう．ここで，上で示したように，すぐ書けて（ある種の制限内では）使える行列クラスをまず使って行列を使う部分をプログラミングし，平行して完成させたより良い行列クラスと最後に結合することができるわけである．

もちろん，行列クラスはそれ程手間のかかるものではないが，より大きなクラス，大きなプログラムに適用できる手法であることは明らかだろう．これは「データ抽象」により可能になっているのである．

また，プログラムの最初のバージョンで使っていたクラスをより効率の良いものに変える必要がでることもあるだろう．その場合も，「データの抽象化」が適正に行われていれば，そのクラスのパブリックな部分は変えずに実装のみを変更できるので，保守管理が楽なのである．

クラスのパブリックな部分は，インターフェースとも呼ばれると書いた．実際，このインターフェースがそのクラスのオブジェクトの性質を規定しており，逆に，その性質を持ったデータを必要とするあらゆるプログラムでこのクラスが使えるのである．これが最初の節で「再利用が容易」と述べた内容である．さらに，バージョンアップの際に，インターフェースを変更しなければ，このクラスを使うすべてのプログラムが即座にバージョンアップされるのである．

実際，余程の事情がなければインターフェースを変更すべきではない．クラスを製作する際に非常に重要なことは適正なインターフェースを考えるということである．インターフェースを頻繁に変更することは，コードの他の場所でのバグを呼ぶことになるだろう．適正なインターフェースを定義することが「抽象データ型」（の中心部分）を定義することなのであり，「実装」はプログラマの知識やコーディングに避ける労力に応じて順次改良していけば良いのである．

4．もうひとつの行列クラス

前節では，「抽象データ」の重要性とそれに対するＣ＋＋の能力を見た．実際，Ｃ＋＋の抽象化能力は極めて強力で，プログラマはその能力をうまく使いこなさなければならないのである．その際に重要なことは，将来のバージョンアップ後も使える有効かつ適当なインターフェースをデータに与えることである．データのインターフェースの頻繁な変更は，プログラム全体の変更につながり，「保守管理の容易さ」などのＣ＋＋の利点を損なうことになるからである．

インターフェースの決定は必ずしも簡単なことではなく，プログラムを開発するグループよりさらに大きな単位で十分議論しておくことが望ましい．著者としては，ひとつの業種全体等で標準が作られることが理想であると思う．ただ，実際には，個別の企業からプログラマ個人の利益にまで関わるので私企業では難しいものかもしれない．また，変化の激しいコンピュータ業界などでは，そもそも標準化は物理的に不可能かもしれない．

一方，利益競争にそれほど関わらず，また，数年以上に渡るプログラム資産を使う組織，例えば，公的研究機関などでは，インターフェースの標準化は望ましいし，また可能であると考える．標準化はときに新しい発展を阻害するものであり，そのデメリットも考えなければならないが，公的機関での標準化は，多数のプログラマがうまく協働するために是非考えるべきことだろう．

本論文は標準化の具体案を提出するものではないが，行列クラスを例に，少し具体的に考えてみたい．行列クラスを考えたのは，C（やC＋＋）に組み込みの行列が無いためである．

まず，行列の要素へのアクセスを考えたい． 前節の例では，要素の値を読み出すのに，Matrix::GetDataという関数を使い，書き込むのにはMatrix::SetDataという関数を使った．このように読み出しと書き込みに別の関数を使うことは，「データ抽象」の考えに沿っている．つまり，データを保護するという意味で安全であり，データの内部構造を変更するのも容易なはずである．

しかし，Cの教科書では，例えば，行列oneの(i,j)要素をone[i][j]と表していることが多い．（これらの例では，普通，行列のデータはプログラムに埋め込まれているため，このようなことが容易にできる．しかし，oneをプログラム実行時に動的に生成するのはそれほど簡単ではない．）このようなアクセスは，読み出しと書き込みが同時に可能なので危険である反面，わかり易いという大きなメリットもある．Cの教科書にあるコードをそのまま使えるという点も実際上は大きなメリットである．それゆえ，このインターフェースは捨て難いかもしれない．以下で，このインタフェースを持つ簡単な例を考えてみる．

```
//SimpleKMatrix.h
#ifndef KMATRIX_H
#define KMATRIX_H
////////////////////////////////////////////////////////////////////////
//  class KMatrix
//
//  Copyright (c) Ken Kobayashi  1997 Oct. 3
//
//  matrix, unresizable
//
////////////////////////////////////////////////////////////////////////
//  USAGE
//
//  KMatrix<T> one(n,m);
//  one is a matrix named KMatrix.
//  one[i], whose type is T*, represents the i-th row of the matirx.
//  one[i][j], whose type is T, is the (i,j) element of the matrix.
//  one[i] and one[i][j] can be used both to get and to set the
//  elements' value
//
```

```
//  The sample code is shown at the end of this file.
////////////////////////////////////////////////////////////////////////
//  WARNING!
//  0.This class is simple and efficient, the auther belives, but
//    not safe.
//
//  1.There is no check on the validity of scripts.
//    Be totally sure that
//          0 <= i < n
//          0 <= j < m
//    with "one[i][j]" for "KMatrix one(n,m);".
//    Scripts out of the ranges may cause a serious damage!
//
//  2.The definiton of operator= should be checked to see whether it is
//    the one which you really want. For example, if the sizes of the
//    matrices are different from each other, what do you want?
//
//  3.The author of this class strongly suggest not to use a pointer as in
//          T * p=&one[i][j];
//    This is dangerous and will be more so after the revision.
//
////////////////////////////////////////////////////////////////////////

template <class T> class KMatrix
{
    T * * data;
    unsigned i_width, j_width;
    void CreateDataSpace();
    void CopyData(const T * const * );
    void DeleteData();
public:
    KMatrix(unsigned i=1, unsigned j=1);
    KMatrix(const KMatrix& m);
    ~KMatrix();
    KMatrix& operator=(const KMatrix& m);

    const T * & operator[](unsigned n) const {
        return data[n];
    }
    T * & operator[](unsigned n){
```

```
        returndata[n];
    }

    void SetAllElements(T);

    unsigned GetIWidth() const { return i_width; }
    unsigned GetJWidth() const { return j_width; }
};

template <class T>
void KMatrix<T>::CreateDataSpace(){
    data=new T * [i_width];
    for(unsigned k=0; k<i_width; k++)
        data[k]=new T[j_width];
}

template <class T>
void KMatrix<T>::CopyData(const T * const * d){
    for(unsigned i=0; i<i_width; i++)
        for(unsigned j=0; j<j_width; j++)
            data[i][j]=d[i][j];
}

template <class T>
void KMatrix<T>::DeleteData(){
    for(unsigned i=0; i<i_width; i++)
        delete [] data[i];
    delete [] data;
}

template <class T>
KMatrix<T>::KMatrix(unsigned i, unsigned j)
    :i_width(i),j_width(j){
    CreateDataSpace();
}

template <class T>
KMatrix<T>::KMatrix(const KMatrix<T>& m)
    :i_width(m.i_width),j_width(m.j_width){
    CreateDataSpace();
```

```
    CopyData(m.data);
}

template <class T>
KMatrix<T>& KMatrix<T>::operator=(const KMatrix<T>& m){
    if(i_width!=m.i_width||j_width!=m.j_width)
        throw "Illegal assignment of matrices.";

    if(&m!=this){
        i_width=m.i_width;
        j_width=m.j_width;
        DeleteData();
        CreateDataSpace();
        CopyData(m.data);
    }
    return *this;
}

template <class T>
KMatrix<T>::~KMatrix(){
    DeleteData();
}

template <class T>
void KMatrix<T>::SetAllElements(T e){
    for(unsigned i=0; i<i_width; i++)
        for(unsigned j=0; j<j_width; j++)
            data[i][j]=e;
}
#endif
```

この実装は，実質的にdataへの直接アクセスを許し，かつエラーチェックをしていないので，危険な実装であるという事実は否めない．しかし，組み込みの配列と同様のインターフェースを持つ，おそらくもっとも簡単な実装で，エラーチェックが無い分高速であるという利点もある．危険性についてはコメントで警告しておいた．これはあくまでも一例であって，研究所等では，必要に応じてより安全な行列クラスを作れば良いと考える．

著者の提案は，より安全で同じインターフェースを持つクラスを作り，それによってプログラムをデバッグし，より高速にしたい場合は，デバッグ後に上の様なクラスを使えば良い，というものである．

ところで，少し論点が変わるようだが，私の印象では，C＋＋の優秀な伝道師たちは，行列クラスにあまり熱意が無いようにみえる．行列を論じる書籍は少ないのである．これは行列が簡単すぎるということもあるが，一方で，行列＝2次元配列はライブラリ製作者（C＋＋の伝道師たちの実際の職業は，言語の専門家かライブラリの製作者であることが多い）の興味を特に引かないのではないだろうか．

ライブラリの製作者に興味があるのは汎用のクラスである．その視点から言えば，2次元の配列を特別に取り上げる理由はあまり無い．まず，考えるのは，汎用の（1次元）配列クラスだろう．汎用であるがゆえに，それは「簡単に」高次元の配列を作る部品となる．例えば，

```
template <class T> calss Array{
    ...
};
```

から，整数を要素に持つ行列は次のように定義できる．

```
Array< Array<int> > one;
```

ここで，整数の行列は「整数の1次元配列の1次元配列」と考えた．これは美しい方法である．また，この方法により，より高次元の配列も，例えば，

```
Array< Array < Array<int> > > one;
```

などと作ることができる．

実際，このような汎用クラスは極めて有効であり，そのようなクラスを必要に応じて使うことに，著者は大いに賛成である．しかし，常に必要なのかは疑問である．

上の方法では，行列のサイズを生成時に決められない．生成時には，デフォルトのサイズ（おそらく0サイズ）を受け入れ，生成後にサイズを変えなければならないのである．実装の仕方によって多少異なるが，例えば，4行5列の行列なら，

```
Array< Array<int> > one(4);
for(int i=0; i<4; i++)
    one[i].Resize(5);
```

などとする必要があるだろう．

これは，少々だが，面倒な作業かもしれない．前節の例のように，例えば，4行5列の整数行列を

```
Matrix<int> one(4,5);
```

などと生成できる方が，使いやすいと感じるクラスユーザは多いだろう．

また，サイズを後から変更するということは，この「行列」クラスはサイズを変更するパブリックな関数を持つということである．

行列のサイズを変更する関数は，果たして必要だろうか．これは場合によるだろうが，普通の科学技術計算では，一度定義した行列のサイズを変更することは無い．そのような計算に使う行列のクラスがサイズ変更関数を持っていても，使わなければ良い，ということは言える．しかし，そのような関数の存在が危険をもたらさないとも言い切れない．長いコードの中で，誰かが行列のサイズを何かの理由（おそらくはメモリ節約のためなど）で変更したとしても，普通の研究者は気が付かないだろう．

また，高次元の配列を作れるということは，ある種の分野では魅力的なことは確かだが，4次元以上の配列はまず使わないという分野もある．汎用性を持たせるためには，クラスの定義が確実に複雑になっていくので，もし，独自の行列を定義したい場合は，Arrayを使った上のような定義（インターフェースの定義）が本当に良いかどうか考える必要があると思う．本論文では，行列クラスを定義する方針で考えを進めてきた．

ここで，1次元配列KVectorを作り，それから行列KMatrixを定義する例を以下にひとつ挙げる．この定義のKMatrixは上で定義したKMatrixと全く同じインターフェースを持つので，同じプログラムで使用できる．（これらをテストするコードを付録に付ける．）

```
//KVectorKMatrix.h
#ifndef KMATRIX_H
#define KMATRIX_H
////////////////////////////////////////////////////////////////////////
//   class KVector
//
//   Copyright (c) Ken Kobayashi  1997 Oct. 5
//
//   private array, unresizable
//
////////////////////////////////////////////////////////////////////////
//   USAGE
//
//   KVector<T> one(n);
//   one is an array named KVector.
//   one[i], whose type is T, is the i-th element of the vector.
//   one[i] can be used both to get and to set the elements' value.
//
////////////////////////////////////////////////////////////////////////
//   WARNING!
//
//   1.There is a check on the validity of scripts.
//
```

```
//  2.The definiton of operator= should be checked to see whether it is
//    the one which you really want. For example, if the sizes of the
//    vectors are different from each other, what do you want?
//
//  3.The author of this class strongly suggest not to use a pointer as in
//        T * p=&one[i];.
//
////////////////////////////////////////////////////////////////////////////
template <class T> class KMatrix;

template <class T> class KVector
{
    friend class KMatrix<T>;

    T * data;
    unsigned size;
    void CopyData(const T * );
    void ResizeData(unsigned);
public:
    KVector():size(0),data(0){}
    KVector(unsigned sz):size(sz),data(new T[sz]){}
    KVector(const KVector&);
    KVector& operator=(const KVector&);
    ~KVector(){ delete [] data; }

    void SetAllElements(T);

    const T& operator[](unsigned n) const{
        if(n>=size||data==0)
            throw "Subscript out of range.";
        return data[n];
    }
    T& operator[](unsigned n){
        if(n>=size||data==0)
            throw "Subscript out of range.";
        return data[n];
    }

unsigned GetSize() const { return size; }
```

```
};
template <class T>
void KVector<T>::CopyData(const T * d){
    for(unsigned i=0; i<size; i++)
        data[i]=d[i];
}

template <class T>
void KVector<T>::ResizeData(unsigned sz){
    if(sz!=size){
        delete [] data;
        size=sz;
        data = new T[size];
    }
}

template <class T>
KVector<T>::KVector(const KVector<T>& a)
    :size(a.size),data(new T[a.size]){
    CopyData(a.data);
}

template <class T>
KVector<T>& KVector<T>::operator=(const KVector<T>& a){
if(size!=a.size)
throw "Illegal assignment of KVectors";

    if(&a!=this){
        CopyData(a.data);
    }
    return *this;
}

template <class T>
void KVector<T>::SetAllElements(T e){
    for(unsigned i=0; i<size; i++)
        data[i]=e;
}

//////////////////////////////////////////////////////////////////////////
```

```
//  class KMatrix
//
//  Copyright (c) Ken Kobayashi  1997 Oct. 8
//
//  matrix, unresizable
//
////////////////////////////////////////////////////////////////////////
//  USAGE
//
//  KMatrix<T> one(n,m);
//  one is a matrix named KMatrix.
//  one[i], whose type is KVector<T>, represents the i-th row of the matirx.
//  one[i][j], whose type is T, is the (i,j) element of the matrix.
//  one[i] and one[i][j] can be used both to get and to set the
//  elements' value
//
////////////////////////////////////////////////////////////////////////
//  WARNING!
//
//  1.There is a check on the validity of scripts.
//
//  2.The definiton of operator= should be checked to see whether it is
//    the one which you really want. For example, if the sizes of the
//    matrices are different from each other, what do you want?
//
//  3.The author of this class strongly suggest not to use a pointer as in
//        T * p=&one[i][j];.
//
////////////////////////////////////////////////////////////////////////
template <class T> class KMatrix
{
    KVector< KVector<T> > * data;
    unsigned i_width, j_width;
    void ResizeData(unsigned, unsigned);
    void CopyData(const KVector< KVector<T> > * );
public:
    KMatrix(unsigned i=1, unsigned j=1);
    KMatrix(const KMatrix&);
    ~KMatrix(){ delete data; }
    KMatrix& operator=(const KMatrix&);
```

```
    const KVector<T>& operator[](unsigned n) const{
        return (*data)[n];
    }
    KVector<T>& operator[](unsigned n){
        return (*data)[n];
    }
    void SetAllElements(T e);
    unsigned GetIWidth() const { return i_width; }
    unsigned GetJWidth() const { return j_width; }
};

template <class T>
void KMatrix<T>::ResizeData(unsigned i, unsigned j){
if(i_width==i&&j_width==j)
return;
i_width=i;
j_width=j;
delete [] data;
    data=new KVector< KVector<T> >(i_width);
    for(unsigned k=0; k<i_width; k++)
        (*data)[k].ResizeData(j_width);
}

template <class T>
void KMatrix<T>::CopyData(const KVector< KVector<T> >* d){
    (*data)=(*d);
}

template <class T>
KMatrix<T>::KMatrix(unsigned i, unsigned j)
  :i_width(i),j_width(j){
    if(i_width==0||j_width==0)
        throw "The matrix can have no element!";
    data=new KVector< KVector<T> >(i_width);
    for(unsigned k=0; k<i_width; k++)
        (*data)[k].ResizeData(j_width);
}

template  <class T>
KMatrix <T> ::KMatrix(const KMatrix& m) {
```

```
    i_width=m.i_width;
    j_width=m.j_width;
    data=new KVector < KVector <T> > (i_width);
    for(unsigned k=0; k <i_width; k++)
        (*data) [k] .ResizeData(j_width);
    CopyData(m.data);
}

template <class T>
KMatrix <T> & KMatrix <T> ::operator=(const KMatrix& m) {
    if(i_width!=m.i_width||j_width!=m.j_width)
        throw "Illegal assignment of matrices.";

    if(&m!=this) {
        ResizeData(m.i_width, m.j_width);
CopyData(m.data);
    }
    return *this;
}

template <class T>
void KMatrix <T> ::SetAllElements(T e) {
    for(unsigned i=0; i <i_width; i++)
        (*data) [i] .SetAllElements(e);
}
#endif
```

これは，1次元配列KVectorで行列KMatrixを作る例であるが，前述のように，インターフェースは前の例と同じである．例えば，行列を生成するには以前と同様，KMatrix<int> one(4, 5);の様にすれば良く，簡単である．また，KVectorのサイズ変更関数KVector::ResizeDataはプライベートなので，ユーザがサイズ変更をする可能性は（ほぼ）無い．

はじめにも述べたが，本論文で扱った行列クラスは，汎用性より使いやすさを重視している．これは当然特定の人たち（著者が想定する研究者・技術者）にとっての使いやすさであり，実際には，いろいろな異なるインターフェースと実装が考えられることは言うまでもない．

例えば，3，4次元配列への拡張は，この方法では容易である．しかし，任意の高次元配列を生成するには向いていない．そのような拡張性は犠牲にして，「使い易さ」を取ったのであるが，任意の次元の配列を頻繁に使う人には良いクラスではないだろう．また，行列のサイズを変更する必要のある人には全く使えないクラスである．(サイズを変更可能にするには，サイズ変更関数KVector::ResizeDataをパブリックにするだけなので，簡単ではある．また，場合によっては，前述の様にサイズ変更関数はパブリックにして，ユーザに「使用に際しては慎重に」

と警告を出す方が良いかもしれない.)

著者は，研究所レベルでは，汎用クラスライブラリを書くより，ある程度特化したクラスライブラリを書くべきであると考えている．本論文はその１例を示すものである．

本論文では，行列クラスのインターフェースとして２例を挙げたが，前節のMatrixと本節のKMatrixのどちらのインターフェースが使い易いかはユーザが決めることである．また，科学技術計算に使うためには，加減や乗算も定義すべきであろう．

5．まとめ

本論文では，行列のクラスを例に「データの抽象化」の様子を見た．著者は「データ抽象」の概念は今後も極めて重要であり続けると考える．

Ｃ＋＋には強力な「抽象化能力」があるが，それゆえに，選択肢も多く，データのインターフェースは慎重に作る必要がある．使い易いインターフェースは，プログラムの開発グループ内ではもちろん，できればより大きな単位でも話し合われるべきであろう．

謝辞

本論文は国際協力事業団（JICA）の短期専門家派遣で，中国潅漑排水技術開発研修センターに滞在中に考えたことを基にしています．実用に供するプログラムのあるべき姿について議論していただいた飯嶋孝史氏をはじめJICA専門家のみなさんと中国側職員のみなさんに感謝いたします．

文献

１．B．ストラウストラップ著　斎藤信男，三好博之，追川修一，宇佐見徹訳
「プログラミング言語Ｃ＋＋」アジソン・ウェスレイ・トッパン　1993年
２．S. C. デューハースト，K. T. スターク著　小山裕司訳
「Ｃ＋＋言語入門」アスキー出版局　1990年
３．A.Koenig, B.Moo著　「Ruminations on C＋＋」　Addison Wesley　1996年
４．小林健一郎「Ｃ＋＋言語のミニマル・サブセットとオブジェクト指向性」
「静岡学園短期大学研究報告」第９号，1996年，309項

付録

```
#include <iostream.h>
#include <stdio.h>
#include "SimpleKMatrix.h"  //または"KVectorKMatrix.h"

//10 * 10 KMatrix with integer elemets
class Int_10_10_KMatrix: public KMatrix<int>
{
public:
    Int_10_10_KMatrix():KMatrix<int>(10,10){}
```

```
};

template <class T>
void ShowKMatrix(const KMatrix<T>& m)
{
    unsigned i, j;
    for(i=0; i<m.GetIWidth(); i++){
        for(j=0; j<m.GetJWidth(); j++){
            cout<<"("<<i<<","<<j<<")=";
            cout<<m[i][j]<<" ";
        }
        cout<<endl;
    }
}

void HitKey()
{
    cout<<"Hit the return key to continue."<<endl;
    getchar();
}

void main()
{
    KMatrix<int> one(10,20);
    one.SetAllElements(1);
    one[2][2]=5;
    ShowKMatrix(one);
    HitKey();

    //error
    //one[2].ResizeData(3);

    KMatrix<int> two;
    two[0][0]=100;
    ShowKMatrix(two);
    HitKey();

    KMatrix<double> three(5,5);
    three.SetAllElements(-0.5);
    three[4][4]=100;
```

```
    ShowKMatrix(three);
    HitKey();

    Int_10_10_KMatrix four;
    four.SetAllElements(3);
    ShowKMatrix(four);
    HitKey();

    KMatrix<Int_10_10_KMatrix> five(3,3);
    five.SetAllElements(four);
    cout<<endl;
    cout<<"(five[2][2])[4][4]="<<(five[2][2])[4][4]<<endl;
    (five[1][1])[0][3]=15;
    cout<<"(five[1][1])[0][3]="<<(five[1][1])[0][3]<<endl;
    HitKey();

    KMatrix<int> six(one);
    ShowKMatrix(six);
    HitKey();

    KMatrix<int> seven(10,20);
    seven=six;
    ShowKMatrix(seven);
    HitKey();
}
```