

メンバの初期化の順番はクラス内の宣言の順番による。そのため、あるデータメンバを別のデータメンバを使って初期化するときには気をつけなければならない。このような仕組みでエラーを出さないためには、初期化子よりコンストラクタの中身（{}の中身）でメンバを初期化する方がよい。

## 課題

- 9-0 この章のプログラムをコンパイルし、実行し、試してください。
- 9-1 学生のデータを読み込んだときに最終成績を計算し、それをオブジェクト内に保持するように `Student_info` クラスを書き直してください。これにともない、`grade` 関数がこの値を使うように書き換えてください。
- 9-2 もし `name` 関数を `const` でないメンバ関数として定義したなら、プログラムのどこを書き換える必要が出てきますか。それはなぜですか。
- 9-3 まだデータを読み込んでいない `Student_info` オブジェクトに対して `grade` 関数を使うと例外を投げるようにしておきました。きちんとコードを書くユーザはこの例外を捉えることになります。ここで例外を投げてもそれを捉えないプログラムを書いてください。それから、例外を捉えるものも書いてください。
- 9-4 `valid` 関数を使って、例外の発生そのものを避けるように、上の問題のプログラムを書き換えてください。
- 9-5 学生の合否判定だけをするプログラムのためにクラスと関連する関数を書いてください。宿題はないものとし、中間試験と期末試験の平均が 60 点以上なら合格にすることにします。出力は、学生をアルファベット順に並べ、その横に合格なら P (Pass の P)、不合格なら F (Fail の F) と記すようにしてください。
- 9-6 上のプログラムを書き換え、はじめに合格した学生をまとめて出力し、次に不合格の学生を出力するようにしてください。
- 9-7 § 4.1.3 の `read_hw` 関数は、（シーケンシャルなデータを `vector` に読み込むという）一般的な仕事をしましたが、その名前は `Student_info` オブジェクトの詳細の一部のようです。もちろん、名前を変えることができますが、逆に、これを `Student_info` の他の部分に組み込み、（一般的な仕事をするにもかかわらず）一般的のユーザがアクセスすべきでないと示すにはどうすればよいでしょう。それにはどうしますか。

# 第 10 章

## メモリ管理と低レベルのデータ構造

ここまで、データは普通の変数か `vector` のような標準ライブラリの提供するコンテナに入れてきました。このようにした理由は、標準ライブラリが提供するデータ構造は、大抵、言語自身が提供するものより柔軟性があり使いやすいからです。

ライブラリの使い方がわかれば、次にするべきことは、それがどのように動作しているかを理解することでしょう。これを理解するためには、ライブラリ抜きの言語そのもののプログラミングのための仕組みや、別の状況でも使えるテクニックを理解する必要があります。ここで、これらを指す言葉として低レベル (low level) という言葉を使いますが、それは標準ライブラリの根底にあるものであり、特にコンピュータのハードウェアがする仕事に近いからです。そのため、使い方は難しくなり危険にもなっていきますが、きちんと理解さえすれば、標準ライブラリの対応物を使うより効率的なデータ構造を使うこともできるのです。標準ライブラリがすべての問題を解決できるわけではないので、ときに C++ プログラムでは、低レベルのテクニックが使われることになるのです。

この章の解説は、問題をまず提起しその解決を見るというこれまでのスタイルをとります。これから紹介する仕組みは、低レベルすぎて難しく、有用な問題解決の例を簡単にお見せできないからです。その代わり、ここでは配列とポインタという関連した 2 つのものから考え始めましょう。それから、`new` エクスプレッションと `delete` エクスプレッションを使う動的なメモリ確保（メモリ割付）の説明をします。この方法によって、プログラマは標準ライブラリの提供する `vector` や `list` を使うより直接的にメモリを管理できるようになります。

第 11 章では、標準ライブラリのコンテナのコードで、これら配列やポインタがどのように使われているかを見てみましょう。

### 10.1 ポインタと配列

配列はコンテナの一種で `vector` に似ていますが、それほどの機能はありません。ポインタはランダムアクセス反復子の一種で、いろいろな用途がありますが、配列の要素にアクセスするのにはとても重要です。ポインタと配列は C や C++ の基本的なデータ構造です。ポインタを使わなければ配列で有用な操作はできないこと、また、配列があることでポインタの重要性が増すことで、この 2 つは分かちがたいと言えます。

この 2 つはとても深く結びついているので、何か意味のある問題を考える前に、両方を説明しておきます。配列を説明する前にポインタの説明をする方が、その逆よりも簡単なので、まずポインタから説明しましょう。

### 10.1.1 ポインタ



他の組み込み型のときと同様に、ローカル変数として生成したポインタは、意味のある値を代入される前には、ゴミが入っています。ポインタの初期値として0を与えることがよくありますが、0を与えられたポインタはどうぞオブジェクトも指さないことになっているのです。また、定数整数の0だけが、ポインタの値として解釈されるものなのです。0をポインタとして解釈したものは、ヌルポインタ(null pointer)とよばれます。これは他のポインタと比較するのによく使われます。

C++の他の値と同様に、ポインタにも型があります。Tという型のオブジェクトのアドレスなら、「T\*」のポインタ(Tを指すポインタ)」とよばれ、定義などでその型はT\*で表されます。

たとえば、`x`を`int`型のオブジェクトとします。これは

```
int x;
```

と定義されますが、次に、`x`のアドレスを格納する変数 `p`を作りたいとします。このとき、`p`は `int`へのポインターと言います。これは、`*p`が `int`ということになりますので、`p`の定義は次のようになります。

```
int *p; // *pがint型
```

ここで`*p`は定義子 (declarator) とよばれるもので、`p`の定義の一部です。`*`と`p`が組で定義子になっているわけですが、大抵の C++ プログラムは、`p`が特定の型 (今は `int*`) を持つことを強調するため、

int a[3]; // aがint\*型

とも書きます。\*のまわりの空白はあってもよいので、上の 2 つは同じ意味なのです。しかし、下の書き方では重要な点を目落としてしまうかもしれません。たとえば、

「どういう意味で」とう?

すると、pは「intへのポインター」になりますが、qはint型変数になります。これは、下のように書くのと同じです。こうするとよりわかりやすいでしょう。

```
int *p, q; // *pとqがint型
```

### あるいは 特に

`int (*p), q; // (*p)とqがint型`

と書くこともできます。もっと、章図することをはっきりさせなければ

```
int* p;      // *pがint型  
int q;      // qがint型
```

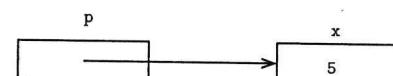
とします。これで以下のようなポインタを使った簡単なプログラムが書けます。

```
int main()
{
    int x = 5;
    // xを指すポインタp
    int* p = &x;
    cout << "x = " << x << endl;
    // xの値をpを使って変更する
    *p = 6;
    cout << "x = " << x << endl;
    return 0;
}
```

このプログラムの出力は

x = 5

です。上のプログラムでは `p` を定義した直後には、下図のよう構造ができます。



最初の出力時には、当然  $x$  は 5 です。次のステートメントでは、 $*p = 6$  を実行することで、 $x$  の値を 6 にしているのです。 $p$  が  $x$  のアドレスを保持しているなら、 $*p$  と  $x$  は同じオブジェクトを表すのです。それゆえ、2番目の出力が実行されるときには、 $x$  は 6 なのです。

1つのオブジェクトを指すポインタを、要素が1つだけでそのオブジェクトを格納しているコンテナの反復子と考えることができます。

### 10.1.2 関数へのポインタ

§ 6.2.2 で、関数を他の関数の引数に使っている例を見ましたが、そこで、これにはもう少し深い意味があるというコメントをしておきました。実は、引数に使った関数はオブジェクトではないのです。関数をコピーしたり代入したり、また直接引数にすることもできないのです。特に、関数をプログラム内で生成したり変更することは

はできません。それができるのはコンパイラだけです。プログラム中でできることは関数を呼び出すことか、そのアドレスを使うことです。

にもかかわらず、§ 6.2.2 の `write_analysis` 関数の引数に、他の関数を使いました。これは、コンパイラが関数そのものではなく、関数へのポインタを使ったということなのです。関数へのポインタは、他のポインタと同様の振る舞いをします。ただ、関数へのポインタでできることは、もとの関数を呼び出すことか、このポインタの値を他のポインタに渡すだけなのです。

関数へのポインタの宣言は他のポインタの宣言と同様です。たとえば、

```
int *p;
```

で、`*p` が `int` であることを示し、結果として `p` がポインタであると宣言するように、

```
int (*fp)(int);
```

と書いて、`*fp` が `int` 型の引数を取り `int` 型の値を戻す関数であることを示すことになります。これにより、`fp` が `int` 型の引数を取り `int` 型の戻り値を持つ関数へのポインタになるのです。

これを使ってできることは、関数のアドレスを代入することか、それを呼び出すことだけです。そこで、関数呼び出しでなければ、たとえ&を使わなくても、アドレスの代入と考えることができます。ここで `fp` と型の合う次のような関数があったとします。

```
int next(int n)
{
    return n + 1;
}
```

このとき、`fp` が `next` を指すようにできるのですが、それには、次のような 2通りの書き方があるのです。

```
// 以下の2つの書き方は同じ意味
fp = &next;
fp = next;
```

同様に、`i` という名前の `int` 型変数があるとすると、`next` を呼び出すのに `fp` を次のように使って `i` をインクリメントできるのです。

```
// 以下の2つの書き方は同じ意味
i = (*fp)(i);
i = fp(i);
```

最後に、もし他の関数を引数に取るようなエクスプレッションを書けば、コンパイラはそれを関数そのものではなく関数へのポインタとして扱うのです。そのため、たとえば § 6.2.2 の `write_analysis` 関数では、パラメータを

```
double analysis(const vector<Student_info>&)
```

と書くことができたのです。これは、

```
double (*analysis)(const vector<Student_info>&)
```

と同じ意味なのです。しかし、この自動の変換は、関数の戻り値には適用されません。もし、`write_analysis` の引数と同じ型の関数へのポインタを戻す関数を書きたい場合は、戻り値の型をきちんと書く必要があるので。この場合、`typedef` を使うのも 1つの方法です。以下のようにすることで、`analysis_fp` を関数へのポインタとして定義できます。

```
typedef double (*analysis_fp)(const vector<Student_info>&);
```

これで、関数の宣言は次のようにできるのです。

```
// get_analysis_ptr関数は関数へのポインタanalysis_fpを戻す
analysis_fp get_analysis_ptr();
```

あるいは、この関数は

```
double (*get_analysis_ptr())(const vector<Student_info>&);
```

のように宣言することもできますが、わかりにくいですね。これは、`get_analysis_ptr()` を呼び出し、その戻り値から関数を取り出すと、それは引数の型が `const vector<Student_info>&` で、戻り値の型が `double` である関数になるということです。幸い、関数へのポインタを戻す関数が使われることはめったにありません。この本では、このような書き方は、§ A.1 で詳細を説明する以外、登場しません。

関数へのポインタは、大抵、他の関数の引数として使われるのです。たとえば、ライブラリの `find_if` 関数の実装の例があります。

```
template<class In, class Pred>
In find_if(In begin, In end, Pred f)
{
    while (begin != end && !f(*begin))
        ++begin;
    return begin;
}
```

この例では、`Pred` は `f(*begin)` が意味を持つものならなんでもよいわけです。たとえば、次のような判定関数を定義したとします。

```
bool is_negative(int n)
{
    return n < 0;
}
```

すると、`find_if` を使って、`vector<int>` である `v` の中の最初の負の要素の場所を見つけることができるのです。

```
vector<int>::iterator i = find_if(v.begin(), v.end(), is_negative);
```

ここで `&is_negative` ではなく `is_negative` と書いたのは、関数の名前は自動的に関数へのポインタと解釈されるからです。また、`find_if` の定義で、`(*f)(*begin)` と書くべきところも、`f(*begin)` と同じ意味になるのです。これはこのようにポインタを使うと、自動的にそのポインタの指す関数の呼び出しと解釈されるからです。

### 10.1.3 配列

配列 (array) は、標準ライブラリではなく、言語そのものにあるコンテナの一種です。配列は、1つ以上の同じ型のオブジェクトを保持します。配列の要素数はコンパイル時に決められていないければならず、したがって、ライブラリのコンテナと違い、配列は途中で大きくしたり小さくしたりできないのです。

配列はクラス型ではないので、メンバを持っていません。特に、配列の大きさを表す値の型として `size_type` のようなものはありません。そのため、`<cstddef>` ヘッダでは、もっと一般的な `size_t` という型を定義しています。実際には、`size_t` は適当な `unsigned` 型で、どのようなサイズもこれで扱えるほど範囲の広い型なのです。それゆえ、配列のサイズは `size_t` 型の変数で扱える（そして、扱うべき）のです。これはコンテナのサイズを扱うのに、`size_type` を使ったのと同じです。

たとえば、3次元の幾何学の問題では、点の座標を次のようなもので表すかもしれません。

```
double coords[3];
```

これは3次元物理空間の座標はいつも3個あるからです。経験を積んだプログラマなら、これを次のように書いたかもしれません。

```
const size_t NDim = 3;
double coords[NDim];
```

ここで、`NDim` はコンパイル時に確定（それは `cost size_t` で、定数として初期化されるから）しているという事実を使っています。このように3という数字ではなく `NDim` を使うことで、3次元の3という数字を3角形の3などから区別できるのです。

配列をどのように定義したとしても、配列とポインタには常に基本的な関係が存在します。それは、配列の名前を値として使うと、それは配列の最初の要素のアドレスを表すということです。たとえば、`coords` を配列として定義しましたが、`coords` を値として使うと、これはこの配列の最初の要素のアドレスを意味するのです。そしてポインタの場合はいつもそうですが、\*演算子を使うとそのポインタの指す要素を取り出すことができます。つまり、

```
*coords = 1.5;
```

を実行すると、これは `coords` の最初の要素を 1.5 にセットしたことになります。

### 10.1.4 ポインタの算術

さて配列の定義の仕方と配列の最初の要素のアドレスの取得について説明しました。他の要素のアドレスはどうすれば得られるのでしょうか。§ 10.1 でも書きましたが、ポインタは反復子の一種なのです。さらにいうと、ポインタはランダムアクセス反復子なのです。これから、配列に関する2番目の基本事項が導かれます。それは、「`p` が配列の第  $m$  番の要素を指すとき、`p + n` は第  $(m + n)$  番の要素を指し、`p - n` は第  $(m - n)$  番の要素を指す（もちろん、その番号の要素がある場合のみです）」ということです。

前節の例の続きを考えましょう。`coords` の最初の要素は第 0 番です。そこで `coords + 1` はこの配列の第 1 番の要素（最初の要素の次の要素）を指すのです。そして、`coords + 2` は第 2 番の要素であり、配列 `coords`

### 10.1.1 ポインタと配列

の要素数が 3 なので、これはまた最後の要素を指すということになります。

それでは `coords + 3` とするはどうなるでしょう。これは、もし配列に第 3 番の要素があれば、それを指すことになるのですが、そのような要素はないのです。

にも関わらず、`coords + 3` は、どこも指さない有効なポインタなのです。これは `vector` や `string` の反子に似ています。`n` 要素の配列の最初の要素を指すアドレスに `n` を足したもののは、どこを指すかどうかに関わらず、比較のために使えるのです。さらに、`p` と `p + n`、`p - n` を結びつける規則も、それらが最後の要素の 1 後（1つよりも後のものはだめです）なら、成り立つののです。

そのため、たとえば、`coords` を次のように `vector` にコピーできるのです。

```
vector<double> v;
copy(coords, coords + NDim, back_inserter(v));
```

ここで、`NDim` は 3 の代わりに使っている変数です。この例では、`coords + NDim` はどの要素も指さないので、最後の要素の 1 後を指す有効な反復子なのです。そのため、`copy` の第 2 引数に使っても何も問題がないわけです。

もう 1 つ例をあげましょう。2 つの反復子から `vector` を作れますか、同様に `coords` の要素をコピーし `vector` オブジェクト `v` を生成できるのです。

```
vector<double> v(coords, coords + NDim);
```

`a` を `n` 要素の配列とすると、標準ライブラリのアルゴリズムは `a` に適用できるということです。その場合 `vector` の `v` に対してなら `v.begin()`、`v.end()` と書くところを `a`、`a+n` と書くと、`a` に対してそのアルゴリズムが使えるのです。

今 `p` と `q` を同じ配列のポインタとすると、`p - q` は `p` と `q` の指す要素の間の距離を表します。これを正確に書くと、`p - q` は  $(p - q) + q$  が `p` に等しくなるように定義されるということです。ここで `p - q` は負の数になるので、その型は符号付整数です。これが正確に `int` か `long` かは環境によって異なるので、ポインタの差を現すために、ライブラリは `ptrdiff_t` という型名を定義しています。これは `size_t` と同様に `<cstddef>` ヘッダに定義されています。

§ 8.2.7 で反復子を計算した結果、コンテナの最初の要素の 1 つ前を指すようにできるとは限らないと指摘しました。同様に、ポインタが配列の前を指すようになる計算は無効です。言葉を換えれば、`n` 要素の配列 `a` に対して、`a + i` が有効なのは、 $0 \leq i \leq n$  のときだけで、特に `a + i` がどれかの要素を指すのは  $0 \leq i < 1$  ( $i$  と  $n$  は等しくないとします) のときだけということになります。

### 10.1.5 インデックスを使う

§ 10.1 で、ポインタは配列のランダムアクセス反復子だと言いました。これは、他のすべてのランダムアクセス反復子の場合と同様に、インデックスも使えるということです。特に、`p` が配列の第  $m$  番要素を指しているとき、`p[n]` は第  $m+n$  番の要素を意味します。これはアドレスではなく要素そのものです。

ここで § 10.1.3 を思い出してください。配列の名前はその最初の要素のアドレスです。これと `p[n]` という記法から、`a` を配列とすると `a[n]` が第  $n$  番の要素になるのです。もっと形式的に言えば、`p` をポインタ、`n` を整数とすると、`p[n]` は  $*(p + n)$  と同じということです。

大抵の言語では、このようなインデックス付けは、基本的で明確です。しかし、C++においては、これは配列の直接的な性質ではないのです。むしろ、これは、配列の名前とアドレスの関係、また、ポインタがランダムアクセス反復子であるという事実から來るのです。

### 10.1.6 配列の初期化

配列には標準ライブラリのコンテナにない重要な性質があります。それは、配列の各要素に初期値を与える簡単な記法があるということです。さらに、この記法には、配列の大きさを明示しなくてもよいという便利さがあります。

たとえば、日付を扱うプログラムを書く場合、それぞれの月に日数がどれだけあるかを保持しておく必要があるかもしれません。これは、たとえば次のようにするとよいのです。

```
const int month_lengths[] = {
    31, 28, 31, 30, 31, 30,      // うるう年以外を考えています
    31, 31, 30, 31, 30, 31
};
```

ここで、1月が第0番要素に対応し、順番に12月が第11番要素に対応しています。そして、上の書き方で、それぞれの要素の値を、その月の日数に初期化できるのです。このような配列を定義しておくと、以後、第*i*番の月の日数はmonth\_length[i]で表すことができるのです。

上の定義では、month\_lengthの要素数を明示していないことに注意してください。ここでは、要素の初期化を具体的にしているので、コンパイラが要素数を数えてくれるのです。このような作業はプログラマがするよりコンパイラが行う方が間違いが少ないでしょう。

## 10.2 文字列リテラル再論

ここまで来てようやく文字列リテラルの本当の意味を理解できるだけの知識を得ました。文字列リテラルとは、実際には、文字数より1つ大きいサイズのconst charの配列なのです。文字列に付け足す要素はヌル文字(つまり\0)で、これはコンパイラが文字列の最後に付け加えるのです。これはつまり、

```
const char hello[] = { 'H', 'e', 'l', 'l', 'o', '\0' };
```

のように書くと、helloが"Hello"と同じ意味になるということです。もちろん、helloという変数そのものは文字列リテラルとは異なるオブジェクトであり、したがってアドレスも違うことはいうまでもありません。

コンパイラが文字列の最後にヌル文字を入れる理由は、こうすると、最初の文字のアドレスを渡すだけで文字列の最後が見つけられるようになるからです。ヌル文字は文字列の最後を示すマークとなり、プログラマにその事實を使えるのです。たとえば、<cstring>にstrlenという関数が定義されています。これは、文字列リテラル、つまりヌル文字で終わっている文字の配列の文字数(ただしヌル文字は数えない)を戻す関数です。strlenの実装には、たとえば次のようなものが考えられます。

```
// 標準ライブラリ関数の実装例
size_t strlen(const char* p)
{
```

### 10.3 文字ポインタ配列の初期化

```
size_t size = 0;
while (*p++ != '\0')
    ++size;
return size;
}
```

§ 10.1.3でsize\_tは配列のサイズを表すのに適当な符号なし整数を表すと書きました。これがsizeの適当な型なのです。このstrlen関数はpで示される配列の文字数をヌル文字は別に数えて戻す関数です。

helloは"Hello"と同じ意味を持つので、

```
string s(hello);
```

はhelloの中身をコピーしたstringオブジェクトsの定義になります。これは

```
string s("Hello");
```

が"Hello"をコピーしたstringオブジェクトsの定義になるのと同じです。また、stringは2つの反復子から生成することもできました。そのやり方では、

```
string s(hello, hello + strlen(hello));
```

ともできます。ここでは、配列の名前である、helloをその配列の最初を指すポインタとして使い、「\0」を指すhello + strlen(hello)をこの文字列配列の最後の要素、\0の1つあとを指すポインタとして使っています。ポインタは反復子なので、§ 6.1.1で紹介した2つの反復子から、stringを生成する方法で、使えるのです。§ 6.1.1のときも、上の例も、前の反復子が初期化に使う文字列の最初の文字を指し、後の反復子がその最後の文字の1つ後を指していることに注意してください。

## 10.3 文字ポインタ配列の初期化

§ 10.2では、文字列リテラルはヌル文字で終わる文字列の始めの文字のアドレスを書く、簡便な方法であるという話をしました。また、中カッコで要素を与えて簡単に配列を初期化する方法も説明しました。この2つの事実から、文字列リテラルを与えることで、ポインタの配列を初期化する方法がわかります。

これは一言では言えないので、例を出して説明しましょう。ここで点数をABC評価(成績を点数ではなく、A, B, C, ..., Fなどで表す評価法)に変えることを考えてみます。これは次のようなルールにします。

点数が以下のもの以上のとき	97	94	90	87	84	80	77	74	70	60	0
ABC評価は	A+	A	A-	B+	B	B-	C+	C	C-	D	F

点数をABC評価に変換する関数は次のように書けます。

```
string letter_grade(double grade)
{
    // 評価を分ける点数
    static const double numbers[] = {
        97, 94, 90, 87, 84, 80, 77, 74, 70, 60, 0
    };
    // 評価
```

```

static const char* letters[] = {
    "A+", "A-", "B+", "B-", "C+", "C-", "D-", "F"
};
// 配列のサイズと要素1つの大きさから
// 配列の要素数を計算
static const size_t ngrades = sizeof(numbers)/sizeof(*numbers);
// 点数から評価を決める
for (size_t i = 0; i < ngrades; ++i) {
    if (grade >= numbers[i])
        return letters[i];
}
return "?\?\?";
}

```

`numbers` の定義には `static` というキーワードを使いましたが、これはすでに § 6.1.3 で説明しました。今の場合、`numbers` と `letters` の初期化はこの関数が最初に呼ばれたときだけにし、その後はこれをそのまま使うようにと、コンパイラに指示することになります。この `static` がないと、コンパイラはこの関数が呼ばれるたびに配列を初期化することになり、これはプログラムの実行速度を不要に落とすことになります。また、配列の要素を `const` していますが、これは配列の要素を変更するつもりがないからです。だからこそ、値の設定を一度だけにできるわけです。

`letters` 配列は `const char` へのポインタの配列です。そして、その要素を、ABC 評価の文字列リテラルで初期化しています。

`ngrades` の定義では、新しいキーワード、`sizeof` を使ってています。これは、配列 `numbers` の要素数を、プログラマが数えずに決める方法になっているのです。`e` をあるエクスプレッションとすると、`sizeof(e)` が、`e` の使うメモリの量を `size_t` 型の値で戻してくれるのです。これはエクスプレッションを実際に評価せずに行います。このようなことができるの、型を調べるときには評価する必要が無く、同じ型のものは、みな同じ決まった量のメモリを使うからです。

さて、その `sizeof` の戻す値は、バイトで測った量です。バイトは、コンピュータによって内部の扱いが異なりますが、メモリの単位として使われるものです。バイトに関して保証されていることは、これが少なくとも 8 ビットを含み、どのようなオブジェクトも 1 バイト以上を使い、`char` が必要とするメモリが正確に 1 バイトである、ということです。

もちろん、今必要なのは、配列 `numbers` の要素数であってバイト数ではありません。そのため、配列全体のバイト数を要素 1 つ分のバイト数で割っているのです。§ 10.1.3 を思い出すと、`numbers` は配列であり、`*numbers` は配列の最初の要素です。これは、正確には配列の最初の要素ですが、すべての要素のサイズは同じですから、何番目かということに意味はありません。重要なことは、`sizeof(*numbers)` が、配列 `numbers` の要素の大きさであり、したがって、`sizeof(numbers)/sizeof(*numbers)` がこの配列の要素数を表すということです。

点数と ABC 評価の対応表はできているので、点数から評価を決めるることは簡単です。まず、`grade` と `numbers` の要素を順番に比べていき、`grade` がそれより大きいか等しい要素を見つけます。それから、それに対応する `letters` の要素を戻すのです。この要素はポインタですが、§ 10.2 で見たように、これは `string` オブジェクトに変換されます。

もし、適当な評価が見つけられなかった場合は、マイナスの点数がつけられているということになりますが、意味不明という文字列を戻すことにしてあります。ここで \ を使ってています。詳細は § A.2.1.4 で説明しますが、

C++ プログラムでは ? を 2 つ以上つなげて書けないのです。そのため、"?\\?" としましたが、これは ??? を表すのです。

## 10.4 main の引数

ポインタと文字配列の説明が終わりました。これで `main` 関数に渡す引数の説明もできます。大抵のオペレーティングシステムでは、`main` に受け取る用意があれば、引数として文字列を与えることができます。`main` に引数を受け取る準備をさせるには、これに `int` と「`char` へのポインタのポインタ」という 2 つのパラメータを与えます。他のパラメータと同様に、パラメータの名前は何にしてもよいのですが、`argc` と `argv` がよく使われます。`argv` はポインタの配列の最初の要素へのポインタです。`argc` は `argv` の指すポインタの配列の要素数を表します。また、`argv` の指す最初の要素は、常にそのプログラムの名前（の文字列へのポインタ）です。このため、`argc` はいつも 1 以上です。そして（あれば）プログラムに与えられる引数が、順に配列の次の要素になっていくのです。

例として、引数があればそれを出力するプログラムを書きます。ここで、引数は複数個あってかまいませんが、それらの間には空白を入れることに注意してください。

```

int main(int argc, char** argv)
{
    // 引数があればそれを書き出す
    if (argc > 1) {
        int i; // ループ後に必要なので i をループの外で定義する
        for (i = 1; i < argc-1; ++i) // 最後の引数以外を空白付きで出力
            cout << argv[i] << " "; // argv[i] は char*
        cout << argv[i] << endl; // 最後の引数を空白なしで出力
    }
    return 0;
}

```

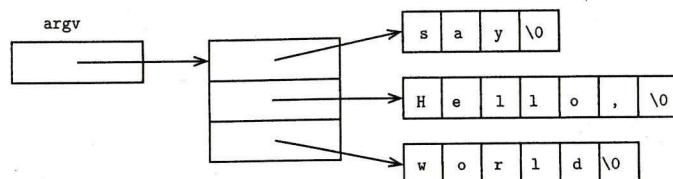
このプログラムをコンパイルし、その実行可能ファイルを `say` とすると、システムに対して、次のように使えます。

`say Hello, world`

これで、`say` が実行され、

`Hello, world`

と出力されます。この場合、`argc` は 3 であり、`argv` には 3 つの要素があります。これらは、文字列 `say`、`Hello,`、`world` の最初の文字を指すポインタなのです。これを図に描くと次のようになります。



## 10.5 ファイルの読み書き

この本では、入出力に `cin` と `cout` しか使いません。しかし、大きなプログラムでは、入力や出力のファイルが必要になります。C++にはそのための仕組みがたくさんありますが、ここではそのうちいくつかを紹介しましょう。

### 10.5.1 標準エラーストリーム

通常の出力先とは別の場所に、プログラムの動作状況を出力すると便利な場合がしばしばあります。これは、たとえばユーザにエラー状況をコメントしたり、プログラムの重要な動作を記録する（ログをとる）などの場合です。

このような出力を通常の出力と区別するため、C++ライブラリは、標準入力と標準出力に加えて、標準エラー（standard error）ストリームというものを定義しています。このストリームは標準出力と一緒にされている場合も多いのですが、大抵のシステムでは、これらを分離する方法があります。

標準エラーストリームに出力するときは、C++では `cerr` と `clog` の両方を使うことができます。この2つのストリームは同じ出力場所を示しています。その違いは、バッファ処理の違いだけです（§ 1.1）。

`clog` ストリームは、名前が示すとおり、ログ（記録、log）が目的です。そのため、バッファの方法は `cout` と同じです。まず出力するデータをバッファにため、システムが適当だと判断すると、それを書き出すというものです。一方、`cerr` ストリームはデータを直ちに出力します。このようにすることで、出力をできる限り早く目に見えるようにすることができます。もちろん、そのために、おおきな無駄を引き起こすかもしれません。つまり、即座に出力したいものには `cerr` を使い、プログラムの正常な動作を記録するためには `clog` を使うべきだということです。

### 10.5.2 様数の出力ファイル、入力ファイルを扱う

標準の入力、出力、エラーストリームはファイルに対応するかもしれませんし、しないかもしれません。たとえば、ウインドウシステムで動作する C++ プログラムは、これらのストリームをプログラムのウインドウに結びつけるかもしれません。そうすることで、ディスクファイルとは別の仕組みを使うかもしれません。

このため、C++ 標準ライブラリでは、ファイルへの入出力には標準の入出力とは違うオブジェクトを使います。入力ファイル、出力ファイルを使いたいなら、`ifstream`、`ofstream` オブジェクトを生成するのです。これは複雑なだけだと思うかもしれません。これまで、ライブラリは入出力用に `istream` と `ostream` を用意しているこ

## 10.5 ファイルの読み書き

とを見てきました。ライブラリは別に `ifstream` と `ofstream` を用意しているのでしょうか。

幸い、そうではありません。第13章で見るよう、標準入出力の型とファイル入出力の型はよく似ています。そこで、標準ライブラリでは、`ifstream` を `istream` の一種として定義し、`ofstream` を `ostream` の一種として定義しているのです。結果として、`istream` の使えるところならどこでも `ifstream` が使え、`ostream` が使えるところならどこでも `ofstream` が使えるのです。これらのクラスは `<fstream>` ヘッダで定義されています。

ところで、`ifstream` や `ofstream` を定義するときには、使いたいファイルの名前を保持している `string` オブジェクトを使うかもしれません。しかし、実際には、`string` オブジェクトではなくヌル文字で終わる文字配列の先頭のアドレスを使うことになっているのです。少々不思議な気がするかもしれません、このようにする理由の1つは、`string` を使わずに入出力ライブラリを使えるようにするためにです。また、歴史的な理由もあります。入出力ライブラリは `string` より何年か前に作られているのです。3番目の理由として、特にポインタをデータのやり取りに使うオペレーティングシステムでは、こうすることで入出力システムとのインターフェースが取りやすいことがあります。理由が何であれ、プログラムでは、ファイル名を `string` オブジェクトではなく、ヌル文字で終わる文字配列の先頭へのポインタにしなければならないのです。

例として、`in` という名前のファイルを `out` という名前のファイルにコピーするプログラムをお見せしましょう。

```
int main()
{
    ifstream infile("in");
    ofstream outfile("out");
    string s;
    while (getline(infile, s))
        outfile << s << endl;
    return 0;
}
```

このプログラムでは、文字列リテラルが「ヌル文字で終わる文字配列の先頭を指すポインタ」であるという事実を積極的に利用しています。もし、ファイル名をこのような文字列リテラルで表したくなれば、ファイル名を `string` に格納し、`string` のメンバ関数 `c_str` を使うのが一番よいと思います。この関数は § 12.6 で説明しますが、たとえば、`file` が読み込みたいファイル名を格納している `string` オブジェクトであるとすると、そのファイルを読み込むための `ifstream` オブジェクトは次のように生成できます。

```
ifstream infile(file.c_str());
```

最後の例として、`main` の引数に与えた名前のファイルを、標準出力に出力するプログラムを示します。

```
int main(int argc, char **argv)
{
    int fail_count = 0;
    // 入力されたファイル名の1つひとつについて
    for (int i = 1; i < argc; ++i) {
        ifstream in(argv[i]);
        // ファイルが存在すればその内容を出力し、存在しなければエラーメッセージを出す
        if (!in) {
            string s;
            while (getline(in, s))
```

```

        cout << s << endl;
    } else {
        cerr << "ファイルを開けません" << argv[i] << endl;
        ++fail_count;
    }
}
return fail_count;
}

```

`main` のすべての引数について (§ 10.4)、プログラムでは、その名前のファイルを読み込むため、`ifstream` オブジェクトを生成しています。ただし、その名前のファイルが存在しないような場合、そのオブジェクトを条件として使うと「`false`」と評価されることになっています。これは、何らかの理由で読み込みができないということです。その場合には、`cerr` にエラーメッセージを出力し、読み込み失敗のカウント `fail_count` を 1 つ増やします。また、`ifstream` オブジェクトが正しく生成されたときは、そのファイルを 1 行ずつ `s` に読み込み、それを標準出力に出力していきます。

このプログラムが終了し制御がコントロールに戻るときには、読み込みに失敗したファイル数をシステムに渡します。いつものように、それが 0 ならプログラムは成功したということになります。それは、指定されたすべてのファイルを読み込めたということです。

## 10.6 3 種類のメモリ管理

明確に指摘しませんでしたが、ここまでで 2 種類のメモリ管理方法を見てきました。まず最初は、自動（オートマティック、automatic）とよばれるもので、これは普通のローカル変数に適用されるメモリ管理です。これは、変数の定義に行き当たると、システムが必要なメモリを確保・割り当てるというものです。この変数の定義があるブロックが終わると、システムはメモリを自動で解放します。

変数のメモリが解放されると、この変数を指すポインタは無効になります。そこで、そのような無効なポインタを使わないようにするのはプログラマの責任になります。たとえば、

```

// この関数は意図的に無効なポインタを戻します
// わざとお見せする悪い例なので真似しないでください
int* invalid_pointer()
{
    int x;
    return &x; // インスタンス災害！
}

```

この関数はローカル変数 `x` のアドレスを戻しています。残念ながら、この関数が `return` を実行すると `x` の定義のあるこの関数のブロックを終了させることになるので、`x` のメモリは解放されてしまうのです。次に何が起こるかは不定です。特に C++コンパイラはこれをエラーとして報告しないかもしれません。成り行きませう。

もし、`x` のような変数のアドレスを戻したいなら、1 つの方法として、別のメモリ管理をするのです。これは静的なメモリ確保 (static allocation) をするということです。

```

// 文法上の問題はまったくありません
int* pointer_to_static()
{

```

```

static int x;
return &x;
}

```

`x` を `static` にしたことで、このメモリ確保は `pointer_to_static` 関数が最初に実行される前に一度だけに行われることになり、プログラムが終了するまでそのメモリは解放されないので。そのため、`static` 変数のアドレスを戻すことにも問題はありません。このポインタはプログラム実行中は有効で、終了してから無効になります。

しかし、`pointer_to_static` では、この関数が呼ばれる度に同じオブジェクトへのポインタを戻すことになるので、`static` メモリ確保の方法は後で問題を起こすかもしれません。ここで、呼び出すたびに新しいオブジェクトのポインタを戻すような関数がほしいとします。また、そのオブジェクトはプログラムが破棄したくなるまで存在するものにしたいとします。このようにするには、動的なメモリ確保 (dynamic allocation) を使います。これには、`new` と `delete` というキーワードが必要です。

### 10.6.1 オブジェクトの生成と破棄

`T` をあるオブジェクトの型とすると、`new T` によってデフォルト初期化された `T` のオブジェクトが生成され（メモリが確保され、それがオブジェクトに割り当てられ）ます。このエクスプレッションは新しく生成された（名前のない）オブジェクトへのポインタを戻します。また、オブジェクトの生成時に、特別に値を引数として与え `new T(引数)` として、初期化することもできます。このオブジェクトはプログラムが終了するか、`p` を `new` が戻すポインタ（か、そのコピー）とするとき `delete p` が実行されるまで破棄されません。なお、ポインタに `delete` を適用するには、そのポインタが `new` で生成されたオブジェクトを指しているか、0 でないなりません。0 であるポインタには `delete` を適用しても何も起こりません。

たとえば、

```
int* p = new int(42);
```

とすると、名前のない `int` オブジェクトがメモリ上に生成され、その値が 42 に初期化され、`p` がそのオブジェクトを指すようになります。このオブジェクトは次のようにして変更することができます。

```
++*p; // pの指すオブジェクトの保持する値は43になる
```

これでオブジェクトの持つ値が 43 になるわけです。そして、オブジェクトを使い終わったら、

```
delete p;
```

を実行します。これにより、`*p` の使っていたメモリ領域は解放され、`p` は無効なポインタになります。`p` に新しい有効なアドレスを代入するまで、`p` が保持している値に意味は無くなるのです。

もう 1 つ例を挙げましょう。これは `int` オブジェクトのメモリを確保し、初期化し、そのポインタ（アドレス）を戻す関数です。

```

int* pointer_to_dynamic()
{
    return new int(0);
}

```

この関数を呼び出した側では、戻されるポインタの指すオブジェクトを適当なときに破棄しなくてはなりません。

### 10.6.2 配列の生成と破棄

`T` をある型とし、`n` を負でない整数すると、`new T[n]` は、型 `T` の要素が `n` 個の配列を生成し、その最初の要素へのポインタ（型は `T*`）を戻します。すべての要素はデフォルト初期化されます。`T` が組み込み型であるなら、ここでオブジェクトは初期化はされないということです。もし `T` がクラスなら、それぞれの要素はデフォルトコンストラクタで初期化されます。

`T` がクラスのときには、初期化のプロセスに 2 つの重要な意味があります。まず、もしクラスがデフォルト初期化を許可しなければ、コンパイラがこのプログラムをエラーとするということです。それから、配列のすべての要素が初期化されるので、かなりのオーバーヘッドになるということです。第 11 章では、標準ライブラリに要素へのポインタ（型は `T*`）を戻します。すべての要素はデフォルト初期化されます。この関数を使うと、前の例と同様に、関数のユーザが受け取ったメモリを解放するコードを書かなくてはなりません。一般的に、動的に確保したメモリを的確なタイミングで解放することは難しいものです。この仕事を自動化する方法を § 11.3.4 で紹介します。

すべての普通の配列は少なくとも要素を 1 つ持つ必要があるのですが、`new T[n]` の方法では `n` を 0 として、要素のない配列を作ることができます。この場合、`new` の戻す最初の要素へのポインタに問題があります。というのは、最初の要素などないからです。この場合に戻されるのは、最後の要素の後のポインタで、これは本来なら最初の要素があるべき場所を示し、後述の `delete []` のために必要となるのです。

このような奇妙さは次のようなコードのために許されるのです。

```
T* p = new T[n];
vector<T> v(p, p + n);
delete[] p;
```

上のコードは `n` が 0 でもエラーにならないわけです。`n` が 0 のときは、`p` がどの要素も指さないので、`p` と `p + n` の比較は可能で、その結果は等しいことがわかります。その場合でも、また `n` が 0 でない場合でも、`vector` は `n` 要素を持つことになるのです。このようなコードが `n` が 0 の場合でも動作すると、とても便利なのです。

この例では、`delete []` というを使いました。ここで角カッコは、配列の最初の要素だけではなく、配列全体を破棄するために必要です。`new []` という形で確保されたメモリはプログラムが終わるか `delete [] p` が実行されるまで解放されません。ただし、この `p` は `new []` の戻したポインタ（かそのコピー）という意味です。なお、配列が破棄されるときには、各要素が逆順に破棄されます。

次の関数を例としてお見せしましょう。これは、文字列リテラルのようなヌル文字で終わる文字配列の先頭の文字へのポインタを受け取り、新しく配列を生成し、そこに文字をすべて（ヌル文字も含めて）コピーし、その先頭の文字へのポインタを戻す関数です。

```
char* duplicate_chars(const char* p)
{
    // ヌル文字も含めて要素数分のメモリを確保
    size_t length = strlen(p) + 1;
    char* result = new char[length];
    // 新しい配列にコピー
    copy(p, p + length, result);
```

### 10.7 詳細

```
return result;
}
```

§ 10.2 で見ましたが、`strlen` はヌル文字で終わる文字配列の文字数を、ヌル文字は入れずに数えて戻す関数です。そこで、コピー先にヌル文字の分のメモリも確保するため、`strlen` の戻す値に 1 を足したもので `length` にしています。ポインタは反復子なので、`p` で示される配列の要素を `result` の配列にコピーするのに `copy` 関数が使えます。`length` にはヌル文字まで含めた文字数が保持されているので、`copy` を実行すると `result` には、オリジナルと同様にヌル文字までコピーされるわけです。

この関数を使うと、前の例と同様に、関数のユーザが受け取ったメモリを解放するコードを書かなくてはなりません。一般的に、動的に確保したメモリを的確なタイミングで解放することは難しいものです。この仕事を自動化する方法を § 11.3.4 で紹介します。

### 10.7 詳細

ポインタ： オブジェクトのアドレスを保持するランダムアクセス反復子。たとえば、

`p = &s`      `p` が `s` を指すようになる。  
`*p = s2`      `p` の指すオブジェクトにアクセスし、それに新しい値 `s2` を代入する。

`vector<string> (*sp)(const string&) = split;`  
  `sp` を `split` 関数を指す関数のポインタとして定義。

```
int nums[100];
    nums を 100 個の int オブジェクトの配列として定義。
int* bn = nums;
    bn を配列 nums の最初の要素を指すポインタとして定義。
int* en = nums + 100;
    en を配列 nums の最後の要素の 1 つ後を指すポインタとして定義。
```

ポインタは 1 つのオブジェクト、配列、関数を指すことができる。関数へのポインタはその関数の呼び出しのみに使われる。

配列： ポインタを反復子とする、固定サイズの組み込み型コンテナ。配列の名前は、自動的に配列の最初の要素へのポインタと解釈される。文字列リテラルはヌル文字で終わる文字配列を指す。配列のインデックスはポインターを通じて付けられる。配列 `a` とインデックス `n` に対し、`a[n]` は `*(a + n)` と同じ。`a` の要素数が `n` のとき、`a` の全要素は範囲 `[a, a + n)` のポインタで表される。

配列は定義されるときに初期化できる。

```
string days[] = { "Mon", "Tues", "Wed", "Thu", "Fri", "Sat", "Sun" };
```

コンパイラは初期化に使う値の数から要素数を決める。

main 関数： (必要であれば) 2 つの引数を取れる。前の引数は `int` で、第 2 引数に何個の文字配列があるかを示す。第 2 引数は `char**` 型であるが、しばしば、