

1-4 次のプログラムはどうでしょうか。下から3行目のところで、}}を};}にしたらどうでしょうか。

```
#include <iostream>
#include <string>
int main()
{
    const std::string s = "a string";
    std::cout << s << std::endl;
    const std::string s = "another string";
    std::cout << s << std::endl; }}
return 0;
}
```

1-5 以下は正しいプログラムでしょうか。その理由も説明してください。また、正しくなければ、正しく書き直してください。

```
#include <iostream>
#include <string>
int main()
{
    std::string s = "a string";
    std::string x = s + ", really";
    std::cout << s << std::endl;
    std::cout << x << std::endl;
}
return 0;
}
```

1-6 入力のところで、Samuel Beckett のように、名前を2つ入力したらどうなるでしょうか。まず、結果を予想してから、実行して確かめてください。

```
#include <iostream>
#include <string>
int main()
{
    std::cout << "What is your name? ";
    std::string name;
    std::cin >> name;
    std::cout << "Hello, " << name
            << std::endl << "And what is yours? ";
    std::cin >> name;
    std::cout << "Hello, " << name
            << "; nice to meet you too!" << std::endl;
    return 0;
}
```

第2章

ループとカウント

§ 1.2 では、あいさつのまわりに決まった形のフレーム（枠）をつけるプログラムを書きました。この章では、プログラムを書き換えることなく、枠の大きさを変えられるような、フレキシブルなプログラムを書きましょう。

その過程で C++ の基礎を学び、いかにループとカウントができるかを見ていきます。また、関連してループの不变量という考え方も紹介します。

2.1 問題

§ 1.2 ではあいさつのまわりにフレームをつけるプログラムを書きました。たとえば、ユーザが Estragon という名前を入力したなら、

```
*****
*          *
* Hello, Estragon! *
*          *
*****
```

となるのでした。このプログラムは1度に1行ずつ出力します。まず、first と second という名前の変数を定義し、最初の行と2行目を格納させ、それらを出力してから、まわりに*をつけたあいさつを3行目に出力するというものでした。4行目と5行目は2行目と1行目と同じため、別に変数を定義する必要はありませんでした。

このアプローチには大きな欠点があります。それはすべての行が変数、つまりプログラムの一部であるということです。そのため、あいさつのまわりの空白をなくすといったわずかな変更でも、プログラム全体を書き直さなければならないのです。そこですべての行をローカルな変数に格納するよりもフレキシブルな方法を考えてみましょう。

この目的のため、あいさつ以外の出力は1文字1文字することになります。ただし、あいさつだけは string という変数に入っています。出力した文字をまた使うことはないので、出力する文字すべてを変数に格納する必要もないことがわかります。

2.2 全体の構造

まず、書き換えなくてよい部分から見ていきましょう。

```
#include <iostream>
```

```
#include <string>
int main()
{
    // ユーザに名前を聞く
    std::cout << "あなたの姓を入力してください：";
    // 姓を読み込む
    std::string name;
    std::cin >> name;
    // あいさつのメッセージを作る
    const std::string greeting = "Hello, " + name + "!";
    // この部分を書き換える
    return 0;
}
```

コメントで「この部分を書き換える」と書いた部分にはいる実際のコードを書きながら、`name` や `greeting` の扱いを決め、また標準ライブラリについて、いくつか説明します。これからプログラムを分割して説明しますが、§ 2.5 でそれらをまとめます。

2.3 あらかじめ行数のわからない行を書く

出力は、1度に1行ずつ書いていく四角形と考えることもできます。あらかじめ何行書かなければいけないかわからなくとも、行数を計算することができます。

あいさつの行と、最初と最後の行はそれぞれ1行です。これで3行ですから、あいさつと最初の行の間に何行空欄の行を入れるかわかれば、その数を2倍して3を足したものが、全体の行数になるわけです。

```
// 1行目とあいさつの間の空白の行数
const int pad = 1;
// 全体の行数
const int rows = pad * 2 + 3;
```

プログラムの中で空白の行数をはっきりさせたかったので、この数を示す変数に `pad` (詰め物) という名前をつけました。この `pad` はあいさつのまわりの空白の数を表しています。また、`pad` を使って、出力する全行数 `rows` は上のように計算できます。

組み込み型の `int` は整数の入れものにはもっとも自然な型なので、`pad` と `rows` は `int` にしました。また、§ 1.2 で紹介した `const` を使って、`pad` も `rows` も変更しないことを示しています。

さらに、1行目とあいさつの間の空白の行の数と同じ数だけ、あいさつの左右にも空白をいれることにしましょう。すると、この `pad` という変数は、上下左右の4方向（側面）に使われることになります。いつも `pad` を空白の数を使っていれば、この値を変えるだけであいさつのフレームの大きさを変えられることに注意してください。

これで書くべき行数がわかったので、実際の出力を考えてみましょう。

```
// 入力と出力の間をあける
std::cout << std::endl;
// 行数rowsだけ出力する
int r = 0;
// 不変な表明：出力した行数はr
```

2.3 あらかじめ行数のわからない行を書く

```
while(r != rows) {
    // この部分に各行の出力を書く (§ 2.4で説明)
    std::cout << std::endl;
    ++r;
}
```

§ 1.2 のときと同じように最初に空白の行を1つ入れて入力と出力を分けました。それ以降の部分は新しいことで詳しい説明が必要です。まずこの説明をすませてから「各行の出力」を考えましょう。

2.3.1 while 文

このプログラムでは `while` 文 (while statement) というものを使って、出力する行数をコントロールします。この `while` 文は、条件が成り立つ限り与えられた命令の実行を繰り返すというものです。その形は

```
while (conditon)
    statement
```

と書きます。このステートメント `statement` が `while` の中身 (body) です。

`while` 文はまず条件 `conditon` を調べるところから始めます。もし、この条件が成り立っていないければ、中身はまったく実行されません。もし、条件が成り立っていれば、中身が1度実行され、また条件が調べられ…と、これが繰り返されます。つまり、`while` 文は、条件を調べることと中身の実行を交互に、条件が成り立たなくなるまで繰り返すのです。条件が成り立たなくなれば `while` 文全体が終了し、プログラムはその次に進んでいきます。

今この例を簡単に言ってしまえば、「`r` が `rows` に等しくない限り、{}の中身を実行する」ということになります。プログラムを見やすくするために、ステートメントを `while (conditon)` と違う行にし字下げをしました。実は、

```
while (conditon) statement
```

となっていても問題はありません。しかし、これではこのプログラムを読む人に、苦痛を与えることになるでしょう。

ところで、ステートメントの後にはわざと；(セミコロン)をつけませんでした。これは、1つの；つきのステートメントでもブロック (block) でもよいからです。ブロックとは、一般に複数の；つきのステートメントを併せて囲んだもののことです。ステートメントが1つの場合、それに；がついているので、その後に書く必要はありません。また、ブロックの場合、}がステートメントの終わりを意味するので、やはりその後に；は要らないのです。なお、ブロックは{}に囲まれたステートメントの集まりなので、§ 0.7 で説明したようにスコープになっています。

さて、`while` は条件 (condition) を調べるところからはじまるのでした。条件とは真偽が定められる式です。たとえば `r != rows` はそういう式なのです。この条件式は、`r` と `rows` を比べるのに、非等価演算子 (inequality operator) `!=` を使っています。このような条件式は、一般に、`bool` という真偽を表す組み込み型の値を戻します。`bool` 型には、`true` (真) と `false` (偽) の2しかありません。¹

¹ 訳注：`r != rows` は、`r` と `rows` が等しいときは `false` という値、`r` と `rows` が等しくないときは `true` という値を戻す (そういう値になる) わけです。

それから、while の中身の最後にある

```
++r;
```

も新しく出てきたものです。++はインクリメント (increment) とよばれ、元のものを 1 増やすという働きを持ちます。これは

```
r = r + 1;
```

と書いても同じです。しかし、「1 増やす」ということはプログラムにはよくあることなので、特に ++ という演算子があるのです。また、詳しくは § 5.1.2 で見ますが、「1 増やす」ように、「あるものの値をすぐ次の値にする」ということは、抽象データ構造の考え方の中では重要な役割を果たします。その意味でもこの演算子は重要なのです。

2.3.2 while 文をデザインする

while 文の条件を考えることはいつもやさしいとは限りません。同様に、while 文全体を理解することはときに難しいものです。§ 2.3 の while 文が rows 行の出力をしているのを理解することはそれほど難しくありませんが、しかし、出力される行数が正確であるということは保証されているでしょうか。出力される行数が rows ではなく、誤って rows - 1 や rows + 1 に、あるいは全然違うものになることはないでしょうか。もちろん、1 行 1 行コードを追っていってその結果を調べることはできますが、それでも間違えることはないでしょうか。

実は、while 文を書いたり理解するのに便利なテクニックがあります。このテクニックは、while 文の意味とプログラムの一般的な動作という 2 つの重要なポイントに関連しています。

最初のポイントは while 文が終了したときには、その条件が false になっているという事実です。そうでなければ、while 文は終了しないからです。たとえば、§ 2.3 の while が終了したときには、r != rows が false、つまり r が rows と等しくなっているわけです。

もう 1 つのポイントは、ループの不变量 (loop invariant) とよばれるものです。これは「while の条件が調べられている間も必ず成立しているべきこと」です。私たちは、プログラムが正しく動作しているかチェックするためにこの不变量の表明（「不变な表明」）を使うことができるし、逆に、不变な表明が適宜成り立つようにプログラムを書いていくこともできるのです。不变な表明はプログラムの一部ではありませんが、プログラムをデザインする上で貴重な知的ツールなのです。そもそも、有用な while ループには、必ずそれに伴う不变な表明を考えることができます。これをコメントとして書いておくと、プログラムを理解するのにも役立ちます。

より具体的に説明するため、§ 2.3 の while 文をもう一度考えましょう。while の直前のコメント「出力した行数は r」が不变な表明です。

この不变な表明がこの部分で成り立つことを確かめるには、while の条件が調べられる度に、この表明が正しいことを確かめる必要があります。これは 2 つの箇所で不变な表明をし、それを確かめるということになります。

それは、まず while 文の直前です。今の例でこれを確かめるのは簡単です。というのは、while の直前には何も出力しておらず、r は 0 に設定されているので、明らかなのです。

もう 1 つの箇所は、while の中身の最後のところです。ここで不变な表明が正しければ、条件を調べる直前に不变な表明が成立していると言えるわけです。これにより、不变な表明は条件を調べる前にはいつも正しいと言

2.4 行の出力

えます。

不变な表明が while の直前で成り立ち、各ステップの最後にも成り立つようにプログラムを書くことで、この不变な表明は while の後でも正しいことが言えます。ここまでをきちんとしておけば、最初は不变な表明が成り立っていたのに、途中で成り立たなくなるということはあり得ないわけです。

ここまで学んだことをまとめておきましょう。

```
// 不変な表明：出力した行数はr
int r = 0;
// rを0にすることで、不变な表明が正しいと言える
while(r != rows) {
    // ここで不变な表明が正しいと仮定できる
    // 各行の出力を書くと、不变な表明が成り立たなくなる
    std::cout << std::endl;
    // rを1増やして、また不变な表明を成り立たせる
    ++r;
}
// ここでは不变な表明が正しいと言える
```

while の不变な表明は、「出力した行数は r」でした。r は定義したときに 0 にしておきますが、この時点では何も出力していないので、不变な表明は明らかに成り立ち、最初の箇所でのチェックが済みます。

while の条件を調べる直前が不变な表明の 2 番目のチェック箇所ですが、これは条件を調べて while の中身を実行した後ということもできるのでした。

行を出力すると、r と出力した行数がずれるので、一時的に不变な表明は成り立たなくなります。そこで、r を 1 増やすことで出力行数と一致させ、不变な表明がまた成り立つようにしたのです。while の中身の最後で不变な表明が成り立つようになったので、条件はすべて満たしたことになります。

こうして 2 箇所でチェックしたので、while 終了後にも r 行出力されていることが保証されるわけです。さらに、このとき、r == rows であることは前に書きました。結局、出力された行は全部で rows になるわけです。

このようなループの理解の仕方はいろいろな場面で有用です。一般に、ループ内の変数の適当な性質を示す不变な表明（今の例では「出力した行数は r」）を見つけ、ループの終了時には条件より、その変数が有用な値（今の例では r == rows）になっていることを見ると良いでしょう。ループの中身は、各ステップで、そして結果的にループ終了後にも、不变な表明が成り立つように変数を操作することになるのです。

2.4 行の出力

前の節で正しい数の行を出力する方法を見たので、これから 1 行をどう出力するかを考えることにします。つまり、§ 2.3 のコード中で「// この部分に各行の出力を書く（§ 2.4 で説明）」と書いたところを実際に書くわけです。

まず、出力する行はみな同じ長さであることに注目しましょう。出力を四角い行列と考えると各行の長さはこの四角の列数ということになります。これはあいさつの長さに片側の空白の数の 2 倍と *2 つ分の 2 を足せば求められます。

```
const std::string::size_type cols = greeting.size() + pad * 2 + 2;
```

まず `const` の意味は簡単で、`cols` という変数の持つ値は、ここで定義した後にはプログラムでは変えないとということでした。`std::string::size_type` という型は難しく見えるかもしれません。ここで`::`はスコープ演算子とよばれるもので、`std::stirng` は、`std` という名前空間の `string` の完全な名前でした。名前空間やブロックと同じく、クラスも独自のスコープを持ちます。`std::stirng` が「保持する文字列の文字数を格納するための変数」の型として、`size_type` というものを定義しているのです。`string` の文字数を格納するために、いつでも `std::string::size_type` という型の変数を使うべきなのです。

`cols` の型を `std::string::size_type` にした理由は、`greeting` に格納されている文字列がどんなに長くなっても、`cols` にその長さを格納したいと考えたからです。`cols` の型としては単純に整数の `int` を考えるかもしれません。実際、それで大抵はうまくいくでしょう。しかし、`cols` に格納される値は、ユーザのこのプログラムへの入力によって決まるもので、その長さを私たちがコントールする方法はないのです。誰かが `int` の範囲を超えるほど長い入力をする可能性はあるのです。

`int` は `rows` の型としては十分でした。`rows` は `pad` によって決まり、それはプログラマが決めるものだからです。どのような環境でも `int` は、最低でも 32767 という大きな整数までが使えるように決められています。しかし、ライブラリが特別な目的で定義している型は、その目的に使うような習慣をつけたほうがよいでしょう。

まず `string` が負の数の文字数を持つということはありません。そのため、`std::string::size_type` は符号なし (`unsigned`) の型とよばれます。これは負の数を格納できないという意味です。この性質は今のプログラムでは使われませんが、§ 8.1.3 では極めて重要な例を見ます。

何行出力するかがわかったので、また `while` 文を使うことができます。

```
std::string::size_type c = 0;
// 不変な表明：現在の行で出力した文字数はc
while (c != cols) {
    // ここに文字を出力する部分を書く
    // 不変な表明が正しくなるようにcを調整する
}
```

この `while` は § 2.3 のものとほぼ同じように動作しますが、中身に違う点があります。§ 2.3 のループでは 1 度に 1 行ずつ書くと考えたのに対して、ここでは 1 度に 1 文字とは限らないということです。1 度に 1 文字ずつ書かなければいけない理由などないです。もちろん、1 文字でも書けば、仕事は先に進んだことになります。ここで考えるべきことは、出力する文字数を正確に `cols` にするということだけです。

2.4.1 フレームの文字を書く

最後の仕事は、出力する文字を決めるということです。まず、最初と最後の行上にいるときと、最初と最後の列上にいるときは、*を出力することがわかっています。

たとえば、`r` が 0 のときは、まだ 1 行も出力していないので、今は最初の行を出力中であることがわかります。また、`r` が `rows - 1` なら、これまで `rows - 1` 行出力したので、今は最後の行を出力中ということになります。同様に、`c` が 0 のときは、各行の最初の文字を書くところであり、`c` が `cols - 1` なら最後の文字を書くところです。これにより、コードを書くことができます。

```
// 不変な表明：現在の行で出力した文字数はc
while (c != cols) {
```

2.4 行の出力

```
if (r == 0 || r == rows - 1 || c == 0 || c == cols - 1) {
    std::cout << "*";
    ++c;
} else {
    // ここに文字を出力する部分を書く
    // 不変な表明が正しくなるようにcを調整する
}
}
```

ここで新しいことたくさんが出てきましたので、少し細かく説明しましょう。

2.4.1.1 if 文

`while` の中身はブロックですが、ここには `if 文` (`if statement`) というものが含まれています。ここで*を出力すべきかどうかを判定しているのです。`if` 文には次のような 2 つの形式があります。

```
if ( condition)
    statement
```

または、今の例のように

```
if ( condition)
    statement1
else
    statement2
```

です。`while` 文のときと同様に、条件は真偽値 (bool 値) を持ちます。もし、条件が真であれば、`if` に続くステートメントが実行されます。2 つ目の形式の場合、条件が偽であれば、`else` 以下のステートメントが実行されます。

`while` の場合と同様に、字下げなどは、必ずしも上のような書式にする必要がないことも覚えておいてください。とは言え、みなさんも、この本を読み進めば、この本のサンプルのような書式で書くほうが、ずっとわかりやすいと思うようになるでしょう。

2.4.1.2 論理演算子

次に条件の式について考えます。

```
r == 0 || r == rows - 1 || c == 0 || c == cols - 1
```

この条件は `r` が 0 または `rows - 1` のときか、`c` が 0 または `cols - 1` のとき真になります。ここには 2 種類の演算子`==`と`||`が使われています。C++では、等しいということ (equality) を調べるのに、`==`という演算子を使います。`=`が 2 つあるのは、代入で使われる`=`と区別するためです。たとえば `r == 0` という条件式は、`r` が 0 に等しいかどうかによって bool 値 (true か false) になります。`||`は論理的or (logical-or) 演算子とよばれるもので、この両側にある条件のどちらかが true なら、全体も true になります。

このような比較の演算子は数学的な演算子より優先度 (precedence) が低いものです。この優先度とは、複数の演算子がある場合に、どちらから実行されるかを決めるものです。たとえば、

```
r == rows - 1
```

とあれば、

```
r == (rows - 1)
```

という意味になるのです。

```
(r == rows) - 1
```

という意味ではありません。それは、数学的演算子である-の方が、比較の演算子==より優先度が高いからです。これにより、まず rows から 1 を引き、その結果と r が比較されるのです。これは望ましい動作です。また、本當は (r == rows) - 1 としたいなら、そのようにカッコを使えばよいのです。この場合、r が rows に等しいかを調べた結果から 1 を引いているので、結果は、r が rows に等しいかどうかに従って、0 か-1 になります*2。

論理的 or 演算子は両側のオペランドのどちらかが true であるかどうかを調べます。その形は

```
condition1 || condition2
```

です。ここで条件 condition1 と条件 condition2 はエクスプレッションで真偽値を与えるものです。

この || 演算子は比較演算子より優先度が低く、他の多くの C++ の演算子と同じく、左結合（左から結合して評価される）です。しかし、他の多くの演算子と違う性質もあります。それは、左側のオペランドが true の場合右側はまったく調べないということです。これはしばしば短絡評価（short-circuit evaluation）と言われます。たとえば § 5.6 で見ますが、この性質がプログラムに決定的な影響を与えることもあります。

|| が左結合であるのと、||、==、- の相対的な優先度によって、

```
r == 0 || r == rows - 1 || c == 0 || c == cols - 1
```

は、カッコを下のようにつけたものと同じです。

```
((r == 0 || r == (rows - 1)) || c == 0) || c == (cols - 1)
```

これは、短絡評価のため、まず最初は、一番右の || 演算子の左オペランドである

```
(r == 0 || r == (rows - 1)) || c == 0
```

が最初に調べられます。そのためには、この右の || の左オペランドである

```
r == 0 || r == (rows - 1)
```

が調べられます。結局、これは最初に

```
r == 0
```

*2 訳注：r == rows の戻り値は true か false ですが、これらは 1,0 と解釈されるのです。

が調べられるということです。もし、r が 0 なら以下のすべての条件式は true です。

```
r == 0 || r == (rows - 1)
(r == 0 || r == (rows - 1)) || c == 0
((r == 0 || r == (rows - 1)) || c == 0) || c == (cols - 1)
```

もし、r が 0 でなければ、次には、r と rows - 1 が等しいかどうかが調べられます。もし、等しくなければ、今度は c が 0 と等しいかが調べられ、等しくなければ、c と cols - 1 が等しいかどうかが調べられ、結果が与えられるのです。

言い換えると、|| でいくつかの条件をつなげると、その条件が順番に調べられるということです。そして、そのうちのどれかが true なら全体の結果が true になり、どの条件も true でなければ全体が false になります。また、どれかの条件が true の場合、それ以降の条件は調べられないのです。そこで上の条件式を細かく見直すと、今書いている文字が、最初の行にあるか、最後の行にあるか、最初の列にあるか、最後の列にあるかを調べていることがわかるでしょう。それらの場合は * を出力するのです。そうでない場合は、別に考える必要があります。

2.4.2 フレーム以外の文字を書く

いよいよ § 2.4.1 のプログラムで

```
// ここに文字を出力する部分を書く
// 不変な表明が正しくなるように c を調整する
```

と書いたところのコードです。ここではフレーム（枠）以外の文字を出力しなければなりません。これらの文字は、当然、あいさつの一部か空白文字です。問題は、そのどちらかをどう判定し、それをどうするかです。

まず、あいさつの最初の文字を出力するところかどうかを調べることを考えます。これは、正しい行で正しい列かどうかを見ればよいのです。あいさつを書く行は、最初の行の後に、* と空白の行を pad 行出力した後です。そしてその行で * を出力して空白文字を pad 個出力してから、あいさつがはじまります。ループの不变な表明からわかることは、r が pad + 1 のときがあいさつの行で、その行で c が pad + 1 であれば、あいさつの文字がはじまるということです。

結局、あいさつの最初の文字を出力し始めるのは、r と c が pad + 1 のときだということです。出力場所がここに来たときに、あいさつを書き出せばよいのです。そして、そうでないときは、空白文字を出力していればよいわけです。また、どちらの場合も、c を正しく更新するのを忘れてはいけません。それは次のようになるでしょう。

```
if (r == pad + 1 && c == pad + 1) {
    std::cout << greeting;
    c += greeting.size();
} else {
    std::cout << " ";
    ++c;
}
```

で、if 文の条件には論理的 and (logical-and) 演算子 && を使いました。|| のように && は両側の条件式を調べます。この演算子は左結合で短絡評価します。ただし、|| と違って、&& は両側のオペランドがそろって true

のときだけ `true` になるのです。もし、どちらかが `false` になれば、全体も `false` になります。右側の条件は、左側の条件が `true` のときのみ調べられます。

この条件が満たされたときにあいさつを書きます。このとき、`c` が出力した文字数とは等しくなくなるので、一時的に不变な表明が成り立たなくなります。そこで出力した文字数を足すことで `c` を更新し、不变な表明がまた成り立つようになります。`c` に出力した文字数を足すためには、新しく、複合代入演算子 (compound-assignment) とよばれるものを使いました。`c += greeting.size()` と書くと、`c = c + greeting.size()` と同じ意味になり、`c` を `greeting.size()` だけ増やすことになるのです。

この条件が成り立たないときは、フレーム上にいるのではなく、あいさつを書いているのでもないので、単に空白を出力し、不变な表明を成り立たせるため、`c` を 1 増やします。これが、`else` のブロックに書かれていることです。

2.5 フレームを描くプログラムの完成

プログラムは完成しました。コードは少しづつ書いたのでここでまとめることにしましょう。しかし、プログラムを短くする方法を 3 つ紹介し、その方法を使って書くことにします。

最初に紹介するのは、標準ライブラリにある名前を使うのに、ある種の宣言を 1 度だけして、その後のタイプ量を減らす方法です。これにより、`std::` を繰り返し書く必要がなくなります。2 番目の方法は、`while` 文を短くするのによく使われる方法です。最後に紹介するのは、`c` を増やすのに 2箇所ではなく 1 箇所にすることで、プログラムを少しだけ短くする方法です。

2.5.1 std::を省略する方法

たぶん、もう標準ライブラリのすべての名前の前にある `std::` を、見たり書いたりするのに疲れてきたのではないかでしょうか。`std::` を毎回書くのは、標準ライブラリの何を使っているかを覚えておくのにはよい習慣です。しかし、もうこれはよくわかったことでしょう。

C++では、特別な名前は特別な名前空間に属していることを示すことができます。たとえば、

```
using std::cout;
```

と書いておけば、`cout` が `std::cout` を意味するようになります。このとき、また、他の `cout` は定義しないという意味もあります。これにより、`std::cout` を単に `cout` と書けるようになります。

このような宣言を、論理的に、**using宣言** (using-declaration) と言います。ここで宣言された名前は他の名前と同様に振る舞います。たとえば、`using` 宣言が中カッコ{}の中にあれば、それはそこから}までののみ有効となります。

以後、プログラムを短くするために、`using` 宣言を使うことにします。

2.5.2 for文を使ってコードを短くする

§ 2.3 で見たプログラムの制御構造をもう一度見てみましょう。一番外側の構造は

```
int r = 0;
```

2.5 フレームを描くプログラムの完成

```
while (r != rows) {
    // rを変えないコード
    ++r;
}
```

です。このような `while` 文はよく見かけるものです。まず、`while` の前に、`while` の条件で調べる変数 `r` を初期化しています。それから、`while` の中身では `r` の値を変え、最後には条件が成り立たなくなります。このような構造がとても多いので、これを短く書く方法が言語にあるのです。それは、

```
for (int r = 0; r != rows; ++r) {
    // rを変えないコード
}
```

です。どちらの形式を使っても、`r` には 0 から `rows - 1` までの間の数が順番に入れられていきます。ここで 0 が範囲の始まり、`rows` が範囲の終わりの後 (off-the-end value) を表わすと考えることができます。このような範囲は、半分開いた範囲 (half-open range) とよばれ、`[begin, off-the-end)` と書かれます。カッコが「」のようにアンバランスなのは、範囲の表現のため、わざとです。たとえば、`[1, 4)` と書けば、それは 1、2、3 のことで、4 は含まれません。同様に `r` の範囲は `[0, rows)` となります。

`for` (for statement) は一般に次のように書けます。

```
for (init-statement condition; expression)
    statement
```

ここで最初の行は `for` ヘッダ (for header) などと言われます。この部分が、次の `for` の中身 (for body) を制御するのです。ここで初期ステートメント `init-statement` と `condition` の間にはわざと ; を書かなかったことに注意してください。というのは、`init-statement` はステートメントであり、通常はそれ自身が ; を最後に持っているからです。

`for` 文の実行は `for` ヘッダの `init-statement` の実行から始まります。普通はそこで、`condition` に使うループ変数を定義し初期化します。`for` ヘッダ内で定義された変数は、`for` 文の終了時に破棄されるので、それ以後は使えません。³

ループの各ステップ (最初のステップから) で `condition` は調べられます。`condition` が `true` のときのみ、`for` の中身が実行されるのです。そして、その後に `expression` が実行されます。それから、また、`condition` が調べられ、中身が実行され、`expression` が実行され...が、`condition` が `false` になるまで繰り返されます。

より一般的に `for` 文の意味を書くと次のようになります。

```
{
    init-statement
    while (condition) {
        statement
        expression;
    }
}
```

³ 訳注：まだこの仕様にしたがっていないコンパイラも見受けられます。

初期ステートメント *init-statement* と *while* 文を、わざわざで囲んだのは、*init-statement* で定義される変数の寿命を示すためです。; の無い場所、ある場所には注意してください。*init-statement* と *statement* はそれ自身が最後に; を持っているので書かなかったのです。*expression* は; をつけてステートメントになるので、; が後についています。

2.5.3 条件チェックをまとめる

§ 2.4 のプログラムで「文字を出力」と書いたところは、3つの部分に分解することができます。それは、*、空白、あいさつ全体の出力です。私たちが書いたコードでは、不变な表明を成り立たせるため、*を出力して c を更新し、また空白を出力して c を更新しています。これ自身に問題はありませんが、プログラムでは、しばしば条件を調べる順番を変えることで、2つ以上の同じステートメントをひとつにまとめることができるものです。

今考えている3つの場合は、同時に成り立つことはないので、条件の検査をどういう順番に行なっても問題ないはずです。そこで、あいさつを書き出すところかどうかを先に調べれば、それ以外の場合には、両方とも c の更新としては 1 増やすだけなので、これをまとめれば ++c は 1 つで済むようになります。

```
if (あいさつを書くところかどうか) {
    cout << greeting;
    c += greeting.size();
} else {
    if (フレームの上かどうか)
        cout << "*";
    else
        cout << " ";
    ++c;
}
```

c の更新を 1 つにまとめると、if と else のブロックが 1 行になるので中カッコもはずしました。++c; の字下げをよく見てください。これにより、フレームを書いていても空白を書いていても適用されることを示しているのです。

2.5.4 プログラムの完成

ここまですべてをまとめ、コードを短くする3つの方法を使うと、プログラム全体は次のようになります。

```
#include <iostream>
#include <string>
// 標準ライブラリで使う名前を宣言
using std::cin; using std::endl;
using std::cout; using std::string;
int main()
{
    // ユーザの姓を入力してもらう
    cout << "あなたの姓を入力してください: ";
    // 姓を読み込む
    string name;
    cin >> name;
    // あいさつメッセージを作る
```

```
const string greeting = "Hello, " + name + "!";
// あいさつのまわりに入れる空白の数
const int pad = 1;
// 出力する行 (rows) と列の数 (cols)
const int rows = pad * 2 + 3;
const string::size_type cols = greeting.size() + pad * 2 + 2;
// 入力と出力をするために空の行を出力
cout << endl;
// rows行の出力
// 不要な表明：出力した行数はr
for (int r = 0; r != rows; ++r) {
    string::size_type c = 0;
    // 不要な表明：現在の行で出力した文字数はc
    while (c != cols) {
        // あいさつの書き出しがチェック
        if (r == pad + 1 && c == pad + 1) {
            cout << greeting;
            c += greeting.size();
        } else {
            // フレームを書いているかチェック
            if (r == 0 || r == rows - 1 ||
                c == 0 || c == cols - 1)
                cout << "*";
            else
                cout << " ";
            ++c;
        }
    }
    cout << endl;
}
return 0;
```

2.6 カウント

大抵の経験を積んだ C++ プログラマは、一見奇妙な習慣を持っています。それは、プログラム中でものを数えるときに、1 からではなく 0 から数えるということです。たとえば、前の節の外側の for ループは次のような形をしていました。

```
for (int r = 0; r != rows; ++r) {
    // 行を書く
}
```

これは、

```
for (int r = 1; r <= rows; ++r) {
    // 行を書く
}
```

とも書けるでしょう。前のものは 0 から数え、条件には != を使っているのに対し、後のものは 1 から数え条件には <= を使っています。繰り返しのステップ数は同じです。前の方は後のものよりよいという理由はあるのでしょうか。

0 から数える 1 つの理由は、それによって、非対称な範囲が使いやすくなるということです。たとえば、`for` 文の範囲としては、`[0, rows)` を使う方が、`[1, rows]` を使うより自然です。

このような非対称な範囲を使うには重要な理由があるのです。それは範囲 $[m, n]$ の個数は $n - m$ になるということです。 $[m, n]$ なら $n - m + 1$ です。そのため、`[0, rows)` の個数は `rows - 0` より、明らかに `rows` なのです。`[1, rows]` の個数よりわかりやすいでしょう。

この対称と非対称の違いは、範囲が実際には空の場合、特に重要です。非対称な表示では、空の範囲が $[n, n]$ のように書けますが、対称な書式では $[n, n-1]$ と書くことになります。終わりの方が始めより後になるということは、プログラムデザインにおいて、尽きないトラブルのもとになるでしょう。

0 から数え始める別の理由は、それによってループの不变な表明が書きやすいということです。今の例でも 0 から数えることで、不变な表明をストレートにしています。つまり、「不变な表明：出力した行数は `r`」です。ここで 1 から数えた場合にこの表明がどうなるか考えてみてください。

たとえば、「これから `r` 行目を書くところ」としたいかもしれません、これは不变な表明ではありません。それは `while` の最後の条件検査では、`r` が `rows + 1` になっているからです。私たちは `rows` 行だけ書きたいのです。したがって、「これから `r` 行目を書くところ」というのは誤りです。

不变な表明は「出力した行数は `r - 1`」とすることもできます。しかし、0 から数え始めた場合はうが簡単ではないですか。

また、0 から数える別の理由として、<= の代わりに != を使えるというのもあります。これはたいしたことではないように思えるかもしれません、ループ終了後のプログラムの状態に影響があるのです。たとえば、条件が `r != rows` なら、ループが終わったときには、`r == rows` になっているわけです。不变な表明は「出力した行数は `r`」と言っているのですから、これで `rows` 回の行を出力していることがはっきりします。しかし、条件が `r <= rows` の場合は、出力したのが少なくとも `rows` 回の行を出力したことしか言えません。もしかしたら、誤って、もっと多く出力してしまっているかもしれないのです。

つまり、0 から数えることで != を使い、正確に繰り返しを `rows` 回行うことが保証できるのです。もちろん、`rows` 以上であればよいという場合は、条件 `r < rows` でできます。1 から数えると、繰り返しの数を最低 `rows` 回にするには、条件を `r <= rows` とすればよいのですが、繰り返しの数を正確に `rows` 回にするのは複雑です。それは `r == rows + 1` のことです。このようなわかりにくい表現を使つてもメリットはありません。

2.7 詳細

エクスプレッション（式）：すでにいくつか紹介してあるように、C++はCからたくさんの演算子を引き継いでいる。さらに、入出力の演算子で説明したように、これらの演算子はクラスとよばれるもののオブジェクトに対して動作が定義され、言語そのものを拡張することにもなる。複雑なエクスプレッションを正確に理解することは、C++で効率的なプログラムを書くためには必須である。そのようなエクスプレッションを理解するために、以下のことを理解する必要がある。

- 演算子の優先度と結合の性質によって、オペランドがどのようにグループ化されるか。

2.7 詳細

- オペランドが別の型に変換される場合は、それがどのように変換されるか。
- オペランドのグループがどのような順で評価されるか。

演算子が違えば優先度も異なる。大抵の演算子は左結合だが、代入演算子とオペランドを 1 つしか持たない演算子は右結合になる。以下に、この章で紹介しなかったものも含めて、一般的な演算子をまとめておく。これは優先度の高い順で、二重線で囲まれたものは同じ優先度のものになっている。

<code>x.y</code>	オブジェクト <code>x</code> のメンバ <code>y</code>
<code>x[y]</code>	オブジェクト <code>x</code> の要素でインデックスが <code>y</code> のもの。
<code>x++</code>	<code>x</code> を 1 増やす（インクリメント）。もとの <code>x</code> の値を戻す。
<code>x--</code>	<code>x</code> を 1 減らす（デクリメント）。もとの <code>x</code> の値を戻す。
<code>++x</code>	<code>x</code> を 1 増やす（インクリメント）。増やした <code>x</code> の値を戻す。
<code>--x</code>	<code>x</code> を 1 減らす（デクリメント）。減らした <code>x</code> の値を戻す。
<code>!x</code>	論理的否定。 <code>x</code> が <code>true</code> のとき <code>!x</code> は <code>false</code> 。
<code>x * y</code>	<code>x</code> と <code>y</code> の積。
<code>x / y</code>	<code>x</code> わる <code>y</code> 。両方が整数の場合、小数部を切り捨てるか、0 に近づくように丸めるかは処理系によって異なる。
<code>x % y</code>	<code>x</code> を <code>y</code> でわった余り。 <code>x - ((x / y) * y)</code> と同じ。
<code>x + y</code>	<code>x</code> と <code>y</code> の和。
<code>x - y</code>	<code>x</code> と <code>y</code> の差。
<code>x >> y</code>	<code>x, y</code> が整数の場合、 <code>x</code> を <code>y</code> ビット右シフト。 <code>y</code> は非負でなければならない。 <code>x</code> が <code>istream</code> の場合、 <code>x</code> から <code>y</code> に読み込み。
<code>x << y</code>	<code>x, y</code> が整数の場合、 <code>x</code> を <code>y</code> ビット左シフト。 <code>y</code> は非負でなければならない。 <code>x</code> が <code>ostream</code> の場合、 <code>y</code> を <code>x</code> に書き出す。
<code>x relop y</code>	関係演算子。 <code>relop</code> には、 <code><</code> 、 <code>></code> 、 <code><=</code> 、 <code>>=</code> が入るが、それぞれの関係の真偽を <code>bool</code> 値で戻す。
<code>x == y</code>	<code>x</code> が <code>y</code> に等しいかどうかを <code>bool</code> 値で戻す。
<code>x != y</code>	<code>x</code> が <code>y</code> に等しくないかどうかを <code>bool</code> 値で戻す。
<code>x && y</code>	<code>x</code> と <code>y</code> 両方が真であるかどうかを <code>bool</code> 値で戻す。 <code>y</code> は <code>x</code> が <code>true</code> のときのみ評価される。
<code>x y</code>	<code>x</code> と <code>y</code> のいずれかが真であるかどうかを <code>bool</code> 値で戻す。 <code>y</code> は <code>x</code> が <code>false</code> のときのみ評価される。
<code>x = y</code>	<code>y</code> の値を <code>x</code> に代入し、その結果の <code>x</code> の値を戻す。
<code>x op= y</code>	複合代入演算子。 <code>op</code> には算術演算子とシフト演算子が入り、 <code>x = x op y</code> と同等になる。
<code>x ? y1 : y2</code>	<code>x</code> が <code>true</code> なら <code>y1</code> を戻し、そうでなければ <code>y2</code> を戻す。 <code>y1</code> か <code>y2</code> のどちらかのみが評価される。

一般に、演算子では、どのオペランドから評価されるか決まっていない。したがって、1つのオペランドが別のオペランドの値に依存するようなエクスプレッションは避けることが重要。そのような例は§4.1.5で見る。

オペランドの型は可能な場合、適当なものに変換される。エクスプレッションの中の数値型オペランドや比較は、通常の算術的変換（usual arithmetic conversion）を受ける。これについては、§A.2.4.4に詳しくまとめる。基本的に算術的変換は精度を保つように行われる。小さい型は大きい型に変換され、符号ありの型は符号なしの型に変換される。また、数値はbool値に変換される。0はfalseで、他のすべての値はtrueと考えられる。クラス型のオペランドは特定の型に変換される。第12章でこのような変換のコントロールの仕方を見る。

型：

```
bool      真偽を表す組み込み型。trueとfalseの2つの値のみを持つ。
unsigned   負でない整数の型。
short     少なくとも16ビット分の値を持つ整数の型。
long      少なくとも32ビット分の値を持つ整数の型。
size_t    (<cstddef>に定義されている)すべてのオブジェクトのサイズを格納できる型。
string::size_type
        stringのサイズを格納できる非負の整数型。
```

半分開いた範囲： 端を片方のみ含む範囲。たとえば、[1, 3)は1と2を含むが3は含まない。

条件： 真偽値を与えるエクスプレッション。条件に使われる数値はbool値に変換される。0以外はtrueに変換され、0はfalseに変換される。

ステートメント：

```
using namespace-name::name;
        名前 name を namespace-name::name の代わりに使えるようにする。
type-name name;
        型が型名 type-name の変数である名前 name を定義する。
type-name name = value;
        型が型名 type-name の変数である名前 name を値 value のコピーとして定義する。
type-name name (args);
        適当な引数 args から型が型名 type-name の変数である名前 name を定義する。
expression;
        副作用を目的としてエクスプレッション expression を実行する。
{ statement (複数可) }
        ブロックとよばれる。ステートメント statement を順番に実行する。複数のステートメントをまとめて、1つのステートメントのように扱える。中カッコの中で定義された変数はブロックのスコープ内にある。
while(condition) statement
```

2.7 詳細

条件 condition が false のときは何もせず、true のときはステートメント statement を実行し、while 全体を繰り返す。

`for(init-statement condition; expression) statement`

{ init-statement while(condition) {statement expression;} }と同じ。（ただし、continueというステートメントが使われる場合は異なる。）

`if(condition)statement`

条件 condition が true のときステートメント statement を実行する。

`if(condition) statement else statement2`

条件 condition が true のときステートメント statement を実行し、そうでないときはステートメント statement2 を実行する。else は一番近い if を補完する。

`return val;`

関数を終了し、値 val を呼び出し側に戻す。

課題

2-0 この章のすべてのプログラムをコンパイルし実行してください。

2-1 この章のプログラムを変更して、フレームとあいさつの間に空白がないようにしてください。

2-2 この章のプログラムを変更して、上下と左右のフレームからあいさつまでの間の空白の数が違うようにしてください。

2-3 この章のプログラムを変更して、ユーザがフレームとあいさつの間の空白の数を指定できるようにしてください。

2-4 この章のプログラムは、あいさつとフレームをわける空白が大部分の行を、1字ずつ出力しています。これを変更して、必要な空白はまとめて1つのエクスプレッションで出力するようにしてください。

2-5 *のみを使って正方形、長方形、三角形を書くプログラムを作ってください。

2-6 次のコードを実行するとどうなるでしょうか。

```
int i = 0;
while (i < 10) {
    i += 1;
    std::cout << i << std::endl;
}
```

2-7 10から-5までカウントダウンするプログラムを書いてください。

2-8 [1, 10)の範囲の整数を足すプログラムを書いてください。

2-9 ユーザに2つの整数を入力させ、その後、どちらが大きいかを表示するプログラムを書いてください。

2-10 次のプログラム中の std:: の使い方を説明してください。

```
int main()
{
    int k = 0;
    while (k != n) { // 不変な表明：ここまでに出力した*の数はk
```