```
    return 0;
}
```

0-8 それから、以下のものはどうでしょうか。

```
#include <iostream>
int main()
{
    // これは複数行にまたがるコメントです。
    // このコメントには// を使い、/*や
    // あるいは*/は使いません。
    std::cout << "うまくいくかな?" << std::endl;
    return 0;
}
```

0-9 もっとも短い正しいプログラムを書いてください。

0-10 Hello, world!プログラムについて、空白が許されているすべての場所に改行を入れて、プログラムを書き直してください。

第1章

string を使う

第0章では、小さなプログラムを詳細に調べ、コメント、標準ヘッダ、スコープ、名前空間、エクスプレッション、ステートメント、文字列リテラル、出力といった、C++の基礎になる仕組みを驚くほどたくさん見ました。この章では、文字列を使う同様に簡単なプログラムを書いて、引き続きC++の基礎を見ていきます。この過程で、宣言、変数、初期化、さらに入力やC++のstringライブラリを紹介します。ここで見るプログラムはとても簡単なもので、制御構造は扱いません。それは第2章で見ることにします。

1.1 入力

前の章では出力方法を学びましたので、ここでは入力方法を学びます。たとえば、「Hello, world!プログラム」を改良して、それぞれのユーザにあいさつする（Helloという）プログラムを書いてみます。

```
// ユーザに名前（姓）を聞いて、あいさつするプログラム
#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;   // nameの定義
    std::cin >> name;   // nameに読み込む
    // あいさつを書く
    std::cout << "Hello, " << name << "!" << std::endl;
    return 0;
}
```

このプログラムを実行すると、まず、画面に

あなたの姓を入力してください：

と書かれます。ここで、たとえば、Vladimir（ウラジミール）[*1]と入力します。

Vladimir

[*1] 訳注：このVladimirと後に出てくるEstragonは、Samuel Beckettの1952年発表の戯曲『ゴドーを待ちながら』(En attendant Godot)の登場人物です。VladimirとEstragonの2人がGodotという人を待つのですが、Godotは現れず何も起こらないという不条理劇です。

すると、画面には

```
Hello, Vladimir!
```

と出力されます。それでは、詳細を見てみましょう。入力を読み込むためには、それを入れておく場所が必要です。そのような場所を**変数**（variable）と言います。変数は名前を持つオブジェクトです。**オブジェクト**（object）とは、コンピュータのメモリ領域の一部で型が指定されている部分のことです。オブジェクトと変数の違いは重要です。実際、§ 3.2.2 や § 10.6.1 で見るように、名前のないオブジェクトも作れるからです。

変数を使いたいときには、プログラム中で変数の名前と型を指定する必要があります。名前と型を同時に指定することで、コンパイラは、より簡単に、プログラムから効率的なコードが作れるのです。また、コンパイラはスペルミスを見つけることができます。もちろん、スペルミスが偶然他の有効な名前になっていなければの話ですが。

今の例では、変数の名前を name としました。そして、その型は std::string です。§ 0.5 と § 0.7 で見たように、std::があるということは、string が C++言語そのもののので定義された型ではなく、標準ライブラリの一部であるということです。標準ライブラリにあるものはいつもそうですが、それを使うためには、特別なヘッダが必要です。std::string の場合、それは<string>であるので、プログラムの最初にその#include ディレクティブをつけたのです。

最初のステートメント

```
std::cout << "あなたの姓を入力してください：";
```

は、もう説明の必要はないでしょう。これはユーザに姓を入力するようにメッセージを出力しているだけです。ここで重要なのは、ここに std::endl がないということです。std::endl 操作子（マニピュレータ）がないため、メッセージの出力後に改行されないのです。そのため、このメッセージがある行で、ユーザの入力を受け取ることになります。

次のステートメントは、

```
std::string name;   // nameの定義
```

ですが、これは、型が std::string で名前が name という変数の**定義**（definition）なのです。この定義は関数の中にあるので、この変数は**ローカル変数**（局所変数、local variable）とよばれます。その意味は、この変数は中カッコの中という限られた場所でのみ有効であるということです。プログラムの}に達すると、name は**破棄**（destroy）され、name が使っていたメモリ領域は他の利用のために解放されます。ローカル変数の寿命が限られているため、変数と他のオブジェクトを区別することが重要になることもあります。

オブジェクトには、明示的でなくとも、**インタフェース**（interface）というものがあります。これはそのオブジェクトでできる動作・作用の集まりという意味です。name を string 型の変数（つまり名前つきのオブジェクト）に定義することで、ライブラリが string オブジェクトでできるとしたことのすべてを、name でできるようになったのです。

string でできることの 1 つに、**初期化**（initialize）というものがあります。string 型変数を定義すると、その値は自動的に初期化されます。それは、string オブジェクトは常に値を持つようにと、ライブラリが決めているからです。もう少し先では、自分で string 変数の初期値を与える方法も見ます。その方法を使わない場合

は、string オブジェクトは、最初、空（empty）またはヌル（null）string という文字を 1 つも含まないオブジェクトになります。

name を定義した後は、

```
std::cin >> name;   // nameに読み込む
```

としています。これは std::cin から name に読み込むという意味です。<<演算子と std::cout と同様に、標準ライブラリは**>>演算子**と std::cin を入力に使います。今の例では、>>は標準的な入力場所から文字列を読み込んで、name にそれを格納します。標準ライブラリは文字列を読むときに、始めにある**空白文字**（whitespace）（スペース、タブ、バックスペース、行末記号など）は入力から捨て去り、次の空白文字かファイルの終わりに達するまでを name に読み込みます。したがって、std::cin >> name は、標準入力から 1 単語読み込み、その文字列を name に格納するのです。

入力には別の副作用があります。それは、名前を入力させるため、プロンプト（入力を促すサイン）を出力デバイスに表示するということです。一般に、入出力ライブラリは、バッファ（buffer）とよばれる内部的なデータ構造に、出力すべきものを貯め、出力を最適化します。大抵のシステムは、書き出す文字数にあまり関係なく、出力にとても時間がかかります。そのため、出力の要求がある度に出力をするのではなく、出力するものをバッファに貯め、必要になったときにのみ、それを出力デバイスに出力するようにしてあるのです。このような出力は**フラッシュ**（flush）といわれます。こうすることで、複数の出力は、実際に 1 つにまとめられます。

バッファをフラッシュするには 3 つの場合があります。まず、バッファが一杯になった場合です。このとき標準ライブラリは自動的にフラッシュします。2 番目は、ライブラリが標準入力ストリームから、読み込むよう要求された場合です。この場合は、バッファが一杯になっていなくても、即座に出力バッファをフラッシュします。3 番目の場合は、プログラマが明示的にフラッシュを要求した場合です。

プログラムが cout にプロンプトを書き出すときには、それはまず、標準出力ストリームに付随するバッファに行きます。次に、cin から入力を読み込もうとすると、これがバッファをフラッシュするのです。そのため、確かに、ユーザは「ユーザ入力を促すもの」を見ることになります。

次のステートメントは出力ですが、実は、これは明示的にバッファのフラッシュを要求しています。ステートメント自身はプロンプトとあまり変わりません。まず、"Hello, "と出力させ、次に name などを出力させ、最後に std::endl を出力させています。std::endl は改行を出力しますが、その後でバッファをフラッシュさせるものなのです。これにより、出力が即座に表示されるのです。

バッファを適切なときにフラッシュすることは、実行時間が長いと思われるプログラムを書くときには重要です。そうしないと、出力したはずのものが、実際に画面に現れるまでに、長い時間バッファ内に留まるかもしれないからです。

1.2 名前にフレームをつける

ここまでで扱ったのは、あいさつのプログラムです。今度は、あいさつをもう少し複雑にし、入力と出力を次のようにしたいと思います。

```
あなたの姓を入力してください：Estragon
**********
*        *
```

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

このプログラムは最後に5行出力します。その最初の行はフレーム（枠）の始めですが、これは、*という文字の列です。その長さは、ユーザの名前に「Hello, 」という文字列と前後の空白と*の長さを足したものに等しくなるよう調節してあります。2行目は両端に*があり、あとは適当な数の空白です。3行目は、始めに*と空白があり、次にあいさつのメッセージがあり、空白と*で終わります。4行目と5行目は、それぞれ、2行目と1行目と同じです。

プログラムでは、最初に出力するものを作っておくのが普通でしょう。ユーザに名前を入力させたら、それをもとにあいさつのメッセージを作り、それを後で出力のときに使うのです。そのような方針でプログラムを書くと以下のようになります。

```
// ユーザに姓を聞き、フレーム付きのあいさつを出力するプログラム
#include <iostream>
#include <string>
int main()
{
    std::cout << "あなたの姓を入力してください：";
    std::string name;
    std::cin >> name;
    // 後で使うため、あいさつのメッセージを作る
    const std::string greeting = "Hello, " + name + "!";
    // 2行目と4行目を作る
    const std::string spaces(greeting.size(), ' ');
    const std::string second = "* " + spaces + " *";
    // 1行目と5行目を作る
    const std::string first(second.size(), '*');
    // すべてを出力する
    std::cout << std::endl;
    std::cout << first << std::endl;
    std::cout << second << std::endl;
    std::cout << "* " << greeting << " *" << std::endl;
    std::cout << second << std::endl;
    std::cout << first << std::endl;
    return 0;
}
```

このプログラムは、まずユーザに姓を聞き、それを name という変数に格納します。それから、あいさつメッセージを greeting という変数に入れています。それから、greetng に格納されている文字の数と同じ数の空白を spaces という変数に格納しています。そして、出力する2行目を作って second に格納し、second にある文字の数と同じ数の*を含む1行目を作って first に格納しています。その後は、それらを1行ずつ出力しているのです。

#include ディレクティブと main の中の最初の3行はもう難しくないでしょう。一方、greeting の定義には新しいことが3つほどあります。

まず、変数には最初から値を与えることができるということです。それには、変数の名前の後に=をつけ、その後に、与えたい値を書き、最後に; を書けばよいのです。変数と値の型が違う場合（§ 10.2 で string と文字

列リテラルの場合の例を見ます）、値の最初の型を変数の型に**変換**（convert）します。

string と文字列リテラルをつなぐ（concatenate）のには、+を使うことができます。これは実は、2つの string をつなぐときにも使えます。（ただし、2つの文字列リテラルには使えません。）第0章では、3+4が7になるという話をしました。文字列をつなぐということはまったく別の働きです。実は、どちらの場合も、+演算子はオペランドの型を調べて、それに合うような作用をするのです。このように、ある演算子が異なる型のオペランドについて違った作用をするということを、その演算子は**オーバーロード**（overload）されていると言います。

最後に、変数の定義の一部に **const**というものが使われているということです。こうすることで、この変数の値は最後まで変えないという約束をすることになるのです。厳密な言い方をすれば、const を使ってもこのプログラムに良いことは何もありません。しかし、値を変えない変数をはっきりさせておくことで、プログラムを理解しやすいものにできるのです。

変数を const 宣言するときには、そこで値を与えなければならないことに注意してください。宣言後には値を変えることができないからです。また、const 変数を初期化するのに使う変数は const でなくても良いことも覚えておいてください。今の例では、greeting の値は name に値が読み込まれるまでわからないのです。そして、name に値が読み込まれるのは、プログラムが実行されているときだけです。このように実行時に値を読み込むため、name 自身は const にはできないのです。

結合の性質は、決して変えられない演算子の性質です。第0章では<<は左結合ということを説明しました。これにより、std::cout << s << t は (std::cout << s) << t と同じ意味になります。+演算子も（>> 演算子も）左結合です。したがって、"Hello, " + name + "!"は、「Hello, 」に name の値をつなげ、その結果に「!」をつなげるということになります。もし name に Estragon という文字列が入っていれば、"Hello, " + name + "!"の値は「Hello, Estragon!」になるのです。

ここまできて、出力するあいさつが決まり、それが変数 greeting に格納されたことになります。次の仕事はこのあいさつを囲むフレームを作ることです。そのために次の1行があるのですが、ここには新しいことが3つ含まれています。

```
const std::string spaces(greeting.size(), ' ');
```

greeting を定義するときには、その初期化に=を使いました。ここでは spaces という string 変数を初期化するのに、カッコで囲まれ、カンマで区切られた2つのエクスプレッションを使っています。=を使う場合には、変数の持つべき値を直接示しました。今のようにカッコを使う場合には、与えられたエクスプレッションからの変数（ここでは spaces）の**構築方法**（construct）を示しているのです。もちろん、この構築方法は変数によって異なるものです。言葉を換えていうと、2つのエクスプレッションから string 変数がどのように構築されるのかを知らなければ、上の spaces の定義はわからないということです。

変数の構築方法は型によって違います。今の場合は、何から変数を構築しているのでしょう。登場している2つのエクスプレッションはどちらもまだ説明していないものです。どういう意味でしょうか。

前のエクスプレッションである greeting.size() は、**メンバ関数**（member function）の呼び出しと言われるものの例になっています。greeting というオブジェクトは size という関数要素を持っているのです。関数というものは値を得るために呼び出せるものです。greeting の型は std::string でしたが、greeting.size() が greeting に格納されている文字数を戻すようになっているのです。

後ろのエクスプレッションは**文字リテラル**（character literal）とよばれるものです。これは文字列リテラルとはまったく違うものです。文字リテラルにはクォーテーション'が使われ、文字列リテラルにはダブルクォーテーション"が使われます。文字リテラルの型は組み込み型（C++にもとからある型を「組み込み型」とよびます）の**char**ですが、この型は複雑なので§10.2で説明します。文字リテラルは1つの文字を表すものです。文字列リテラルの中で特殊な意味を持っていた文字は、文字リテラルの中でも同じ意味を持ちます。たとえば、'や\という文字を表すには、その前に\を置きます。同様に、\n、\t、\"は、それぞれ、第0章の文字列リテラルで説明したように、改行、タブ、"という文字を表します。

さて、spaces の定義を理解するためには、整数と文字から string がどう構築されるかを知る必要があるのです。実はそれは、その整数の個数だけその文字をコピーして作るということなのです。たとえば、

```
std::string stars(10, '*');
```

とあれば、stars.size() は 10 になり、stars 自身は**********になるのです。

したがって、spaces は greeting と同じ数の空白が並んだ文字列になるのです。

second では、もう新しいことは使っていません。これは、*と空白の文字列と*をつなげて、フレーム付きあいさつの2行目にしているだけです。first の定義も同様に簡単です。これは second の文字数と同じ数の*の文字列です。

プログラムの残りの部分は、§1.1と同様に、文字列を出力しているだけです。

1.3 詳細

型：

`char` コンパイラが定義している型で、普通の英語の文字を保持する組み込み型。

`wchar_t` 日本語などのワイド文字を保持できる大きな組み込み型。

string 型： 標準ヘッダ<string>の中で定義されている。この型のオブジェクトには、0個から複数個の文字の列を格納できる。n を整数、c を文字、is を入力ストリーム、os を出力ストリームとすると、string には次のような使い方がある。

`std::string s;`
: s を空の std::string オブジェクトとして定義。

`std::string t = s;`
: t を std::string のオブジェクトとして定義し、その値を s の文字をコピーしたものとして初期化。ここでの s は他の string オブジェクトでも文字列リテラルでもかまわない。

`std::string z(n, c);`
: z を std::string のオブジェクトとして定義し、その値を c を n 個コピーしたものとして初期化。ここで c は char であって、string オブジェクトや文字列リテラルではない。

`os << s` フォーマットを変えずに、os で表される出力ストリームに、s の中の文字列を書き出す。このエクスプレッションの戻り値は os。

`is >> s` is で表される入力ストリームから、まず空白文字を破棄し、最初に出てくる空白文字でない文字から次に空白文字が出てくるまでを、s に読み込む。この際、最初から s にあった文字列は上書きされてしまう。戻り値は is。

`s + t` このエクスプレッションの戻り値は、s のコピーと t のコピーをつなげた文字列。s か t のどちらかは、文字列リテラルか char オブジェクトでかまわない。ただし、どちらかは std:string でなければならない。

`s.size()` s 内の文字列の文字数。

変数： 変数は次の3つのうちのどれかで定義できる。

```
std::string hello = "Hello";      // 初期値を明示して定義する
std::string stars(100, '*');      // 型のルールに従って、エクスプレッションから構築する
std::string name;                 // 型のルールにしたがって初期値を明示せずに定義する
```

中カッコの中で定義された変数はローカル変数とよばれ、その中カッコの中が実行されている間だけ存在している。実行が}に達するとその変数は破棄され、その変数が使っていたメモリ領域はシステムに返される。

変数を const つきで定義すると、その変数の値は変数が破棄されるまで変更できないという約束になる。このような変数は後で値を代入することができないので、定義のときに初期化しなければならない。

入力： std::cin >> v が実行されると、標準入力ストリームの空白文字が破棄され、空白でない文字列が v に読み込まれる。std::cin の型は istream。入力の命令が連続して書けるよう std::cin 自身が戻される。

課題

1-0 この章のプログラムをコンパイルし、実行し、試してみてください。

1-1 以下の定義は有効でしょうか。その理由も説明してください。

```
const std::string hello = "Hello";
const std::string message = hello + ", world" + "!";
```

1-2 以下の定義は有効でしょうか。その理由も説明してください。

```
const std::string exclam = "!";
const std::string message = "Hello" + ", world" + exclam;
```

1-3 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。

```
#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;
}
```