第１章

画面への出力とキーボードからの入力

画面に表示を行ったりキーボードから数値や文字を読み込んだりするプログラムを通じて、C++ に慣れましょう。

1-1 C++の歴史

本書で学習する C++ の歴史を紹介します。

AT&T ベル研究所の Bjarne Stroustrup 博士は 1980 年頃、事象駆動型のシミュレーションの記述にあたり C 言語を拡張した言語を作りました。それは、Simula67 から取り入れたオブジェクト指向の基礎となる**クラス**の概念や、強力な関数引数型チェックなどの機能をもったものでした。後に C++ と呼ばれることになるこの言語は、C 言語と Simula67 を両親とする言語であるといえます（**Fig.1-1**）。

▶ 右（子）から左（親）に矢印が向いている理由は、第 14 章を学習すると分かるようになります。

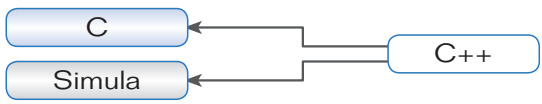

Fig.1-1 C++とその両親

それ以来この言語は、次の名称で呼ばれることになります。

> **クラス付きの C**（*C with classes*）

1983 年には、仮想関数や演算子多重定義などの機能が導入されました。この言語には、Rick Mascitti によって、

> **C++**（シープラスプラス）

という名称が与えられます。これは、もとになった言語の名前 C に ++ という記号を付加したものとなっています。その ++ は、C 言語の演算子の一つであり、

> 値を 1 単位だけ増やす

という機能をもちます（p.76）。C++ は、C 言語を拡張したものであって、まったく異なる言語ではないということが分かります。“D” などといったネーミングに比べると控え目ですよね。Stroustrup 博士が、C 言語に対して敬意を払っていることの表れであるとも考えられます。

さて、現実の C++ には多くのバージョンが存在します。1983 年には C++ の大学への頒布が始まり、1985 年に商業ベースの Release 1.0 の販売が開始されます。

Stroustrup 博士自身が 1986 年に出版した

> The C++ Programming Language*

は、その Release 1.0 に相当する C++ の解説書です。このバージョンに対して、限定公

開部などを導入し、若干の改良を施した Release 1.1 や 1.2 などが相次いで発表されます。

その後、多重継承などの機能が追加されて、大幅な改良が行われます。これが Release 2.0 です。

Stroustrup 博士は、1990 年に Margaret A. Ellis との共著で

> The Annotated C++ Reference Manual**

を発表しました。この書は C++ の完全な文法書であり、Release 2.1 に相当します。ここではテンプレートと例外処理が、今後追加されるであろう試行的な機能であると紹介されています。Release 3.0 では、テンプレートが正式に導入されました。

Stroustrup 博士は、1997 年に、

> The C++ Programming Language Third Edition***

において、新しい C++ を解説しています。

*

C 言語や C++ などのプログラミング言語の国際的な規格や各国の国内規格は、以下の機関で“標準規格”として制定されています。

> 国 際 規 格：国際標準化機構（ISO：International Organization for Standardization）
> 米国の規格：米国国内規格協会（ANSI：American National Standards Institute）
> 日本の規格：日本工業規格（JIS：Japan Industrial Standards）

体裁などの細かい点が異なることを除くと、これらは（基本的には）同一のものです。現在、C 言語と C++ の規格は、いずれも第 2 版となっています。

■ 標準 C

第 1 版：1989 年に ANSI 規格が制定され、翌 1990 年には ISO 規格が制定されました。JIS 規格が制定されたのは 1993 年です。ANSI の制定年から C_{89} と呼ばれます。

第 2 版：1999 年に ANSI および ISO 規格が制定され、2003 年に JIS 規格が制定されました。ANSI ／ ISO の制定年から C_{99} と呼ばれます。第 1 版との互換性に乏しいため、この規格の C 言語はあまり使われていないのが実情です。

■ 標準 C++

第 1 版：1988 年に ANSI および ISO 規格が制定されました。

第 2 版：第 1 版に対して小改訂を施したものです。2003 年に ANSI および ISO 規格が制定され、ほぼ同時に JIS 規格が（C++ としては初めて）制定されました。本書で解説する C++ は、この規格に基づいています。

---

* 邦訳：斎藤信男訳『プログラミング言語 C++』，トッパン，1988

** 邦訳：足立高徳訳『注解 C++ リファレンスマニュアル』，トッパン，1992

*** 邦訳：㈱ロングテール／長尾高弘訳『プログラミング言語 C++ 第 3 版』，アスキー，1998

1-2 まずは画面に表示

コンピュータで処理を行ったときは、その結果を何らかの形で人間に伝えなければなりません。本節では、コンソール画面へ表示を行うことによって、コンピュータから人間に情報を伝える方法を学習します。

■ コンソール画面への出力

最初に作るのは、コンソール画面に表示を行うプログラムです。テキストエディタなどを使って **List 1-1** のプログラムを打ち込みましょう。

大文字と小文字、半角文字と全角文字は区別されますので、この通りにします。

List 1-1 Chap01/list0101.cpp

```
// 画面への出力を行うプログラム

#include <iostream>
using namespace std;

int main()
{
    cout << "初めてのC++プログラム。\n";
    cout << "画面に出力しています。\n";

    return 0;
}
```

実行結果

```
初めてのC++プログラム。
画面に出力しています。
```

▶ プログラム中のスペースや " などの記号を全角文字で打ち込まないよう注意しましょう。

バックスラッシュ \ の代わりに**円記号 ¥** を使う日本独自の文字コード体系が採用されていることもあります。みなさんの環境に応じて、必要ならば読みかえてください。

なお、本書に示すプログラムは、ホームページからダウンロードできます（p. ivの『本書の構成』を参照してください）。

C++ のプログラムは、アルファベット・数字・記号などで構成されます。こんなに短いプログラムですが、/, \, #, {, }, <, >, (,), ", ; と数多くの記号が使われています。

▶ C++ のプログラムで利用する記号文字の読み方は、**Table 1-1**（p.11）にまとめています。

本書では、みなさんが読みやすく理解しやすくなるよう、青文字、*斜体字*、**太字**、**ゴシック体**などを使い分けてプログラムを表記しています。

■ ソースプログラムとソースファイル

私たち人間が《文字の並び》として作成する **List 1-1** のようなプログラムを**ソースプログラム**（*source program*）と呼び、それを格納するファイルを**ソースファイル**（*source file*）と呼びます。

▶ **ソース**（*source*）は、“もとになるもの”という意味です。ソースプログラムは、**原始プログラム**と呼ばれることもあります。

打ち込んだソースファイルは list0101.cpp という名前で保存します。ただし、拡張子が .c や .cc や .C でなければならない処理系もあります。みなさんの環境に応じて、必要ならば変更してください。

▶ **処理系**とは、C++ プログラムの開発に必要なソフトウェアのことです。Microsoft Visual C++、Borland C++ など数多くの処理系があります。

■ プログラムの実行

コンピュータは、C++ のソースプログラムを直接理解・実行することはできません。そのため、**Fig.1-2** に示すように、ソースプログラムを**コンパイル**したり**リンク**したりする作業を行って**実行プログラム**を作成する必要があります。

私たち人間が読み書きしやすい《**文字の並び**》を、コンピュータが理解しやすい 0 と 1 の並びである《**ビットの並び**》に変換するのです。

▶ **ビット**とは 0 あるいは 1 の値をもつデータ単位です。0 と 1 のみを表すことのできる 1 桁（けた）のデータと理解しましょう。

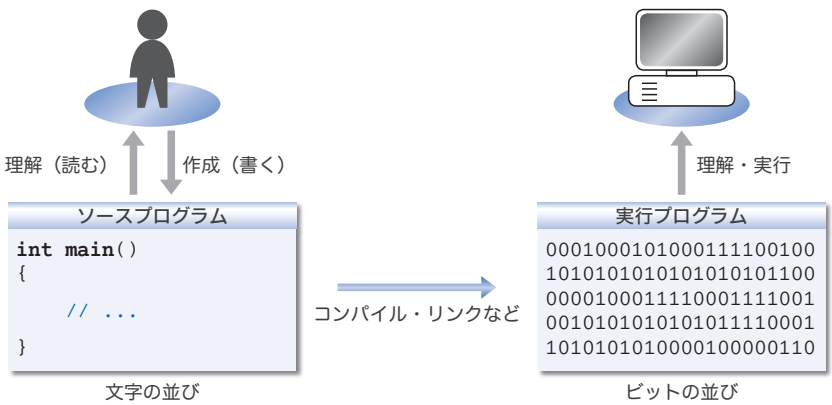

Fig.1-2　プログラムの作成から実行まで

コンパイルの手順やプログラムの実行方法は処理系によって異なりますので、マニュアルなどを参照してやってみましょう。

▶ ソースプログラムに綴（つづ）り間違いなどがあると、コンパイル時にエラーが発生し、その旨の**診断情報**（*diagnostic message*）が表示されます。その際は、打ち込んだプログラムをよく読み直して、ミスを取り除いた上で、再度コンパイル・リンクの作業を試みましょう。

なお、プログラム中の余白（空白）に、全角文字のスペースを打ち込むと正しくコンパイルできません。半角文字のスペースか、タブを打ち込みます。

コンパイルが完了したらプログラムを実行します。そうすると、実行結果（左ページ）に示すように、コンソール画面への出力が行われます。

■ コメント（注釈）

プログラムの先頭行は//で始まっています。連続する２個のスラッシュ記号 // は、

```
// 画面への出力を行うプログラム
```

この行のこれ以降は、プログラムの＜読み手＞に伝えることです。

と指示します。伝える内容は、プログラムに対する**注釈＝コメント**（*comment*）です。

コメントの内容は、**プログラムの動作には影響を与えません**。作成者自身を含め、プログラムの読み手に伝えたいことがらを、簡潔な言葉（日本語や英語など）で記述します。

重要 ソースプログラムには、作成者自身を含めた《読み手》に伝えるべきコメントを簡潔に記入せよ。

コメントには /* と */ とで囲む記述法もあります。開始を表す /* と終了を表す */ とが**同一行になくてもよい**ため、右のように複数行にわたるコメントの記述に効果的です。

```
/*
   画面への出力を行うプログラム
*/
```

▶ この記述法を使う場合は、コメントを閉じるための */ を、/* と書き間違えたり書き忘れたりしないように注意しましょう。

■ ヘッダとインクルード

コメントの次の行は、以下の指示を行います。

```
#include <iostream>
```

画面やキーボードなどへの入出力を行うためのライブラリ（処理実現のための部品群）に関する情報が格納されている <iostream> の内容を取り込みます。

その結果、**Fig.1-3** に示すように、**#include** 指令の行は <iostream> の内容とそっくり入れかえられます。こうすることによって、入出力のライブラリを利用するのに必要な情報を入手するのです。

入出力を扱う <iostream> の他にも、文字列を扱う <string>、時間を扱う <ctime> などが提供されます。それらは**ヘッダ**（*header*）と呼ばれます。

なお、**#include** 指令によってヘッダの内容を“取り込む”ことを、**インクルードする**（*include*）といいます。

重要 ヘッダにはライブラリに関する重要な情報が格納されている。プログラムで利用するライブラリに関する情報が格納されているヘッダをインクルードせよ。

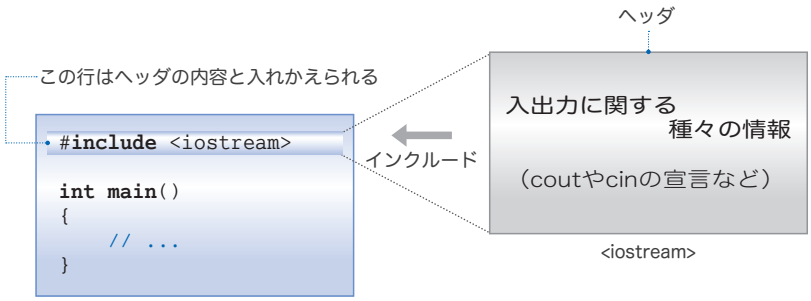

Fig.1-3　#include指令によるヘッダのインクルード

▶ 各ヘッダは、単独のファイルとして提供されるとは限りません。というのも、テキストファイルではなく、コンパイル済みの特殊な形式で提供されることもあるからです（そのため、ヘッダファイルではなく、単に**ヘッダ**と呼ばれます）。

なお、単独のファイルとして提供される場合でも、<iostream> のファイル名が "iostream" であるという保証はありません（"iostream.h" や "iostream.hpp" といった名前かもしれません）。

■ std 名前空間の利用

#include の次の行は、using 指令と呼ばれる指令です。この指令は、

```
using namespace std;
```

std という名前空間（*name space*）を使います。

という宣言です。

名前空間については第 8 章で学習しますので、現在の段階で理解する必要はありません。とりあえずは、C++ が提供する**標準ライブラリ**の利用に必要な《決まり文句》として覚えておきましょう。

▶ std は standard（標準）の略です。なお、using namespace std; は省略することもできます。ただし、その場合はプログラム中の cout を std::cout に変更しなければなりません（p.284）。

□ 演習 1-1

ヘッダ <iostream> をインクルードする指令が欠如していると、どうなるであろうか。プログラムをコンパイルして確認せよ。

□ 演習 1-2

using 指令を削除して、cout を std::cout に変更したプログラムを作成せよ。

■ コンソール画面への出力とストリーム

コンソール画面への出力を行っている箇所を理解しましょう。

```
cout << "初めてのC++プログラム。\n";
cout << "画面に出力しています。\n";
```

コンソール画面やファイルなどに対する入出力には**ストリーム**（*stream*）を利用します。ストリームとは、文字が流れる“河”のようなものです（**Fig.1-4**）。

重要 外部への入出力は、文字が流れる河であるストリームを経由して行う。

cout は、コンソール画面に対する文字の流れである**標準出力ストリーム**（*standard output stream*）です。

ストリームに対する出力は、文字を《挿入する》ことによって行います。ストリームへの挿入を指示するのが、左向きの不等号が二つ並んだ << です。この記号は**挿入子**（*inserter*）と呼ばれます。

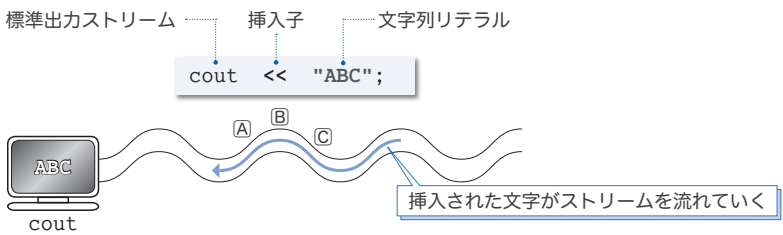

Fig.1-4 コンソール画面への出力とストリーム

▶ 以下、コンソール画面のことを単に「画面」と呼ぶことにします。

ヘッダ名 iostream は**入出力ストリーム**（*input-output stream*）の略で、cout はコンソール・アウト（*console out*）の略です（一般に“シーアウト”と発音します。cont とか count と書き間違えないようにしましょう）。

また、二つの << は連続しなければなりません。< と < の間にスペースやタブを入れないようにしましょう。

■ 文字列リテラル

"初めてのC++プログラム。\n" や "ABC" のように、二重引用符 " で囲んだ文字の並びは、**文字列リテラル**（*string literal*）と呼ばれ、ひとまとまりの《文字の並び》を表します。

▶ 二重引用符 " は、文字列リテラルの開始と終了を表す記号です。cout に挿入したときに " が画面に表示されるわけではありません。文字列リテラルの詳細は第 7 章で学習します。

■ 改行

文字列リテラル中の \n は《改行文字》です。改行を出力すると、それに続く表示は、

次の行の先頭から行われます。そのため、まず『初めての C++ プログラム。』が表示され、それから行を改めて『画面に出力しています。』が表示されます。

▶ `\n` は、見かけは \ と n の 2 文字ですが、《改行文字》という**単一の文字**を表します。このように、目に見える文字として表記が不可能あるいは困難な文字は、\ で始まる**拡張表記**によって表します。拡張表記の詳細は第 3 章（p.98）で学習します。

■ main 関数

プログラムの本体となる部分を抜き出したのが **Fig.1-5** です。

この部分は **main 関数**（*main function*）と呼ばれます。プログラム実行時は、**main** 関数中の**文**（*statement*）が順次実行されていくことになっています。

重要 C++ のプログラムの本体は **main** 関数である。プログラム実行時には、その中の文が順次実行される。

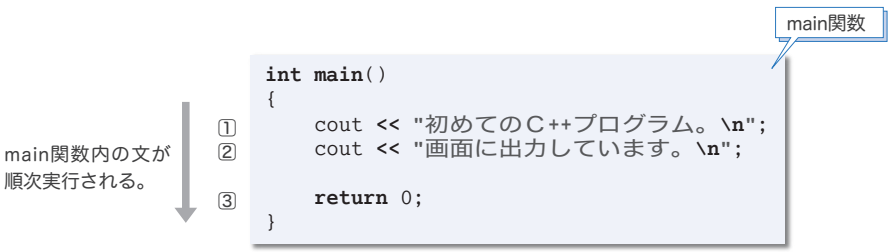

Fig.1-5 プログラムの実行とmain関数

▶ `int main()` や `{ }` や `return 0;` は後の章で学習しますので、とりあえずは《決まり文句》として覚えましょう。なお、関数については第 5 章や第 8 章などで詳しく学習します。

■ 文

本プログラムの **main** 関数には三つの**文**（*statement*）があります。これらの共通点はセミコロン ; で終わることです。日本語の文の末尾に句点 。があるのと同様に、C++ の文の末尾には原則としてセミコロン ; が必要です（一部の例外があります）。

重要 文は、原則としてセミコロンで終わる。

▶ コメントは文では**ありません**ので、《コメント文》といった文は存在しません。

□ 演習 1-3

文の終端を示すセミコロン ; が欠如しているとどうなるか。プログラムをコンパイルして確認せよ。

■ ストリームへの連続した出力

List 1-2 は、二つの文字列 "はじめまして。" と "こんにちは。" を連続して表示するプログラムです。

List 1-2 Chap01/list0102.cpp

```
// 挿入子<<を連続適用して画面に出力

#include <iostream>
using namespace std;

int main()
{
    cout << "\aはじめまして。" << "こんにちは。\n";

    return 0;
}
```

警告

改行

実行結果

♪はじめまして。こんにちは。

このように、出力ストリーム cout に対して複数の挿入子を連続して適用すると、**先頭側（左側）のものから順に出力されます。**

■ 警告

文字列リテラル中の **\a** は《警告》です。cout に対して警告を出力すると、視覚的あるいは聴覚的な注意を促すことができます。多くの実行環境では、いわゆる“ビープ音”が鳴ります（画面が点滅するような実行環境もあります）。

▶ 本書の実行例では、警告を♪と表記します。

□ 演習 1-4

右に示すように、各行に 1 文字ずつ名前を表示するプログラムを作成せよ。著者の名前ではなく、自分の名前を表示すること。

```
柴
田
望
洋
```

□ 演習 1-5

右に示すように、各行に 1 文字ずつ名前を表示するプログラムを作成せよ。姓と名の間は 1 行あけることとし、自分の名前を表示すること。

```
柴
田

望
洋
```

■ 記号文字の読み方

C++ のプログラムで利用する記号文字の読み方を **Table 1-1** に示します。

▶ ここに示す読み方は、通称・略称・俗称を含んでいます。

■Table 1-1 記号文字の読み方

+	プラス符号、正符号、プラス、たす
-	マイナス符号、負符号、ハイフン、マイナス、ひく
*	アステリスク、アスタリスク、アスター、かけ、こめ、ほし
/	スラッシュ、スラ、わる
\	逆斜線、バックスラッシュ、バック　※ JIS コードでは ¥
¥	円記号、円、円マーク
%	パーセント
.	ピリオド、小数点文字、ドット、てん
,	コンマ、カンマ
:	コロン、ダブルドット
;	セミコロン
'	一重引用符、引用符、シングルクォーテーション
"	二重引用符、ダブルクォーテーション
(	左括弧、左丸括弧、左小括弧、パーレン
)	右括弧、右丸括弧、右小括弧、パーレン
{	左波括弧、左中括弧、ブレイス
}	右波括弧、右中括弧、ブレイス
[	左角括弧、左大括弧、ブラケット
]	右角括弧、右大括弧、ブラケット
<	小なり、左向き不等号
>	大なり、右向き不等号
?	疑問符、はてな、クエッション、クエスチョン
!	感嘆符、エクスクラメーション、びっくりマーク、びっくり、ノット
&	アンド、アンパサンド
~	チルダ、なみ、にょろ　※ JIS コードでは ‾（オーバライン）
‾	オーバライン、上線、アッパライン
^	アクサンシルコンフレックス、ハット
#	シャープ、ナンバー
_	下線、アンダライン、アンダバー、アンダスコア
=	等号、イクオール、イコール
|	縦線

■ 自由形式記述

List 1-3 に示すプログラムを見てください。**List 1-1**（p.4）のプログラムと本質的には同一であり、実行結果も同じです。

List 1-3 Chap01/list0103.cpp

```
/*
    画面への出力を行うプログラム     */

#include <iostream>

using
namespace std;

int main(

                                                    ) {

cout << "初めてのC++プログラム。\n";   cout
<< "画面に出力しています。\n";
    return
0
 ;                                                                  }
```

読みにくいけれども正しいプログラム

実行結果

```
初めてのC++プログラム。
画面に出力しています。
```

一部のプログラミング言語が課す「プログラムの各行を、ある決められた桁位置から記述せねばならない。」といった制約は、C++ にはありません。ソースファイルの自由な桁位置にプログラムを記述できる**自由形式**（*free formated*）が許されます。

このプログラムは、思いきり自由に（？）記述した例です。もっとも、いくら自由であるとはいっても、いくつかの制限があります。

1 単語の途中にホワイトスペースを入れてはいけない

int, **return**, **main**, cout, **<<**, //, /*, */ などは、それぞれが『単語』です。これらの途中に**ホワイトスペース**（スペース文字・タブ文字・改行文字など）を入れて、

```
ret
    urn
```

と記述することはできません。

2 文字列リテラルの途中で改行してはいけない

二重引用符で文字の並びを囲んだ文字列リテラル "…" も、一種の単語のようなものです。したがって、以下のように途中で改行してはいけません。

```
cout << "初めての
                C++プログラム。\n";
```

プログラム中に長い文字列リテラルを記述する必要がある場合は、文字列リテラルを区

切ってそれぞれを " " で囲みます。たとえば、

```
cout << "昔々、あるところに、お爺さんとお婆さんが住んでいました。\n";
```

は、以下のように記述できます。

```
cout << "昔々、あるところに、お爺さんと"
        "お婆さんが住んでいました。\n";
```

このように、**ホワイトスペースをはさんで隣接している文字列リテラルは、単一の文字列とみなされます。**

③ インクルード指令の途中で改行してはいけない

自由形式であるとはいえ、先頭が # 文字である **#include** などの指令は特別扱いです。原則として、単一行の中で書かなければなりません（複数行にわたって記述する方法もあります）。

▶ このような指令は**前処理指令**（*preprocessing directive*）と呼ばれ、**#include** の他にも **#define** 指令（p.110）や **#if** 指令（p.343）などがあります。

■ インデントのすすめ

最初に示したプログラム **List 1-1**（p.4）をもう一度よく見てください。**main** 関数の中に書かれている文は、すべて左から数えて 5 桁目から記述されています。

{ } は、まとまった文（日本語での“段落”）をくくったものです（p.50）。段落中の記述を右に数桁ずらして書くと、プログラムの構造が見やすくなります。このような余白のことを**インデント**といい、インデントを用いて記述することを**インデンテーション**（段付けする）と呼びます。

本書のプログラムは、左端から 4 桁ごとのインデントを与えて表記します（**Fig.1-6**）。

```
int main()
{
    for (int i = 1; i <= 9; i++) {
        for (int j = 1; j <= 9; j++)
            cout << setw(3) << i * j;
        cout << '\n';
    }

    return 0;
}
```

▶ ここに示しているプログラムは、**List 3-12**（p.88）の一部です。
九九の表を出力します。

Fig.1-6 インデント

1-3 変数

画面への出力法が分かりましたので、単純な計算を行って、その結果を表示するプログラムを作ってみましょう。

■ 演算結果の出力

足し算を行って、その結果を表示するプログラムを作りましょう。**List 1-4** に示すのは、二つの整数値 18 と 63 の和を求めて表示するプログラムです。

List 1-4 Chap01/list0104.cpp

```
// 二つの整数値18と63の和を求めて表示

#include <iostream>
using namespace std;

int main()
{
    cout << "18と63の和は" << 18 + 63 << "です。\n";

    return 0;
}
```

実行結果

```
18と63の和は81です。
```

■ 整数リテラル

18 や 63 のように、整数を表す定数のことを**整数リテラル**（*integer literal*）と呼びます。

▶ 18 は整数リテラルで "18" は文字列リテラルです。整数リテラルの詳細は第 4 章で学習します。

■ 演算結果の出力

プログラム網かけ部による出力の様子を示したのが **Fig.1-7** です。

cout に挿入されている二つの文字列リテラル "18と63の和は" と "です。\n" は、画面にそのまま表示されます（ただし \n は《改行文字》として出力されます）。

一方、文字列リテラルではない 18 + 63 は、そのまま表示されるのではありません。整数と整数を加算した結果である 81 が表示されます。

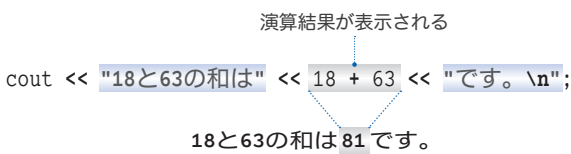

Fig.1-7 ストリームへの整数値の出力

■ 変数

このプログラムは、18 と 63 以外の数値の和を求めることができません。和を求める数値を変更する際は、プログラムに手を加える必要があります。

値を自由に出し入れすることのできる《変数》を使うと、そのような煩わしさから解放されます。

■ 変数の宣言

変数とは、数値を格納するための“箱”のようなものです。いったん箱に値を入れておけば、その箱が存在する限り**値が保持されます**。また、値を書きかえるのも取り出すのも自由に行えます。

プログラム中に複数の箱があると、どれが何のための箱なのかが分からなくなってしまいます。ですから、箱には《**名前**》がないと困ります。

変数を使うには、箱を作るとともに、その箱に名前を与えるための**宣言**（*declaration*）が必要です。*x* という名前の変数を宣言する**宣言文**（*declaration statement*）は次のようになります。

```
int x;          // xという名前をもつint型変数の宣言
```

int は《整数》という意味の語句 integer に由来します。この宣言によって、名前が *x* である変数（箱）が作られることになります（**Fig.1-8**）。

重要 変数を使いたいときには、まず宣言をして名前を与えよ。

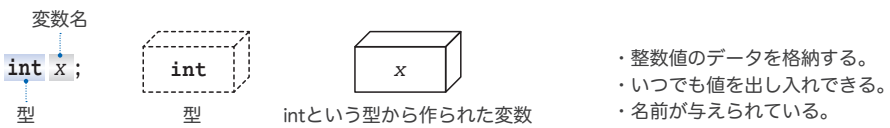

Fig.1-8　変数と宣言

変数 *x* が扱えるのは《整数》だけです（たとえば 3.5 といった実数値は扱えません）。これは、**int** という**型**（*type*）の性質です。

int は**型**であり、その型から作られた変数 *x* は **int** 型の**実体**というわけです。

▶ **int** 以外にもたくさんの型が提供されます。型に関する詳細は第 4 章以降で、名前の付け方に関する規則は第 2 章（p.61）で学習します。

なお、二つ以上の変数をまとめて一度に宣言することもできます。以下のようにコンマ文字で区切って宣言します。

```
int x, y;          // int型の変数xとyを一度に宣言
```

二つの変数 x と y に値 63 と 18 を代入して、その合計と平均を表示するプログラムを **List 1-5** に示します。

List 1-5 Chap01/list0105.cpp

```
// 二つの変数xとyの合計と平均を表示

#include <iostream>
using namespace std;

int main()
{
    int x;              // xはint型の変数
    int y;              // yはint型の変数

1   x = 63;             // xに63を代入
    y = 18;             // yに18を代入

2   cout << "xの値は" << x << "です。\n";                  // xの値を表示
    cout << "yの値は" << y << "です。\n";                  // yの値を表示
3   cout << "合計は" << x + y << "です。\n";               // xとyの合計を表示
    cout << "平均は" << (x + y) / 2 << "です。\n";         // xとyの平均を表示

    return 0;
}
```

実行結果

```
xの値は63です。
yの値は18です。
合計は81です。
平均は40です。
```

▶ 二つの変数を 1 行にまとめて `int x, y;` と宣言せず、別個に宣言しています。このほうが、個々の宣言に対するコメント（注釈）が記入しやすくなり、宣言の追加や削除が容易になります（ただしプログラムの行数は増えてしまいます）。臨機応変に使い分けましょう。

■ 代入演算子

二つの変数に値を入れるのが 1 です。= は、右辺の値を左辺に**代入**するように指示する記号であり、**代入演算子**（*assignment operator*）と呼ばれます。

Fig.1-9 に示すように、変数 x には 63 が代入され、変数 y には 18 が代入されます。

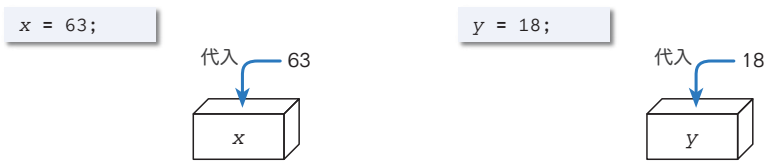

Fig.1-9 変数への値の代入

▶ 演算子については、p.21 で学習します。

代入演算子は、数学のように『x と 63 が等しい』とか『y が 18 と等しい』といっているのではないことに注意しましょう。

なお、代入演算子には、演算と代入を同時に行う複合形式のものもあります（p.80）。

■ 変数の値の表示

変数に格納されている値は、いつでも取り出すことができます。2の最初の行では、**Fig.1-10** に示すように、変数 x の値を取り出して表示しています。

▶ cout に挿入する x は文字列リテラルではありませんから、画面に x の《変数名》が表示されるのではなく、その《値》が表示されます。

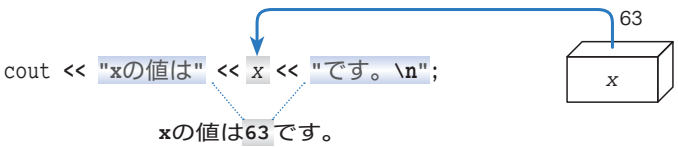

Fig.1-10　ストリームへの変数の値の出力

■ 算術演算子と演算のグループ化

3では、x と y の合計と平均を表示しています。

式 x + y を囲む（ ）は、優先的に演算を行うための記号です。**Fig.1-11** a に示すように、まず x + y の加算が行われ、それを 2 で割る除算が行われることになります（スラッシュ記号 / は除算を行う演算子です）。

図 b のように、（ ）がなく x + y / 2 となっていれば、x と y / 2 との和を求めることになります。私たちが日常行っている計算と同じで、**加減算よりも乗除算のほうが優先される**からです。

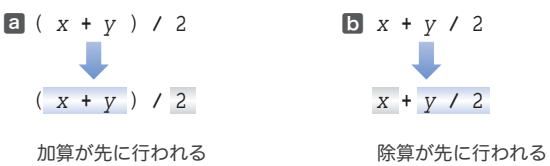

Fig.1-11　( )による演算順序の変更

なお、整数 / 整数の演算では、**小数点以下の部分が切り捨てられます**。実行結果が示すように、63 と 18 の平均値は 40.5 ではなく 40 となります。

▶ すべての演算子と優先順位は p.64 にまとめています。

□ 演習 1-6

List 1-5 のプログラムを変更して小数点以下をもつ実数値を x や y に代入せよ。その実行結果から、**int** 型変数が整数値のみしか扱えないことを確認せよ。

□ 演習 1-7

三つの **int** 型変数に値を代入し、それらの合計と平均を求めるプログラムを作成せよ。

■ 変数と初期化

前のプログラムから、変数に値を代入する1の部分を削除するとどうなるかを検討しましょう。**List 1-6** を実行してみてください。

List 1-6 Chap01/list0106.cpp

```
// 二つの変数xとyの合計と平均を表示（変数は不定値）

#include <iostream>
using namespace std;

int main()
{
    int x;          // xはint型の変数（不定値となる）
    int y;          // yはint型の変数（不定値となる）

    cout << "xの値は" << x << "です。\n";                  // xの値を表示
    cout << "yの値は" << y << "です。\n";                  // yの値を表示
    cout << "合計は" << x + y << "です。\n";               // xとyの合計を表示
    cout << "平均は" << (x + y) / 2 << "です。\n";         // xとyの平均を表示

    return 0;
}
```

実行結果一例

```
xの値は6936です。
yの値は2358です。
合計は9294です。
平均は4647です。
```

変数 x と y が**妙な値**となっていることが実行結果から分かります。

▶ この値は、実行環境や処理系によっても異なります。プログラムを実行するたびに異なる値となる可能性もあります。

変数が生成される際は、原則として**不定値**すなわち**ゴミの値**が入れられます。そのため、値が正しく設定されていない変数から値を取り出して演算を行うと、思いもよらぬ結果となるのです。

▶ ただし、静的記憶域期間をもつ変数に限り、自動的に 0 が入れられます。詳しくは第 5 章（p.199）で解説します。

■ 初期化を伴う宣言

変数に入れるべき値が分かっているのであれば、その値を変数に最初から入れておいたほうがよいでしょう。

そのように宣言するプログラムが **List 1-7** です。

変数 x と変数 y は、生成時に 63 と 18 という値で**初期化**（*initialize*）されます。

網かけ部に示すように、変数に入れるべき値を = 記号の右側に与えます。初期値を与える部分は**初期化子**（*initializer*）と呼ばれます（**Fig.1-12**）。

変数が生成される際に入れるべき値を設定する

```
int x = 63;
        初期化子
```

Fig.1-12 初期化を伴う宣言

重要 変数には、特に不要でない限り初期化子を与えて確実に初期化せよ。

List 1-7 Chap01/list0107.cpp

```
// 二つの変数xとyの合計と平均を表示（変数を明示的に初期化）

#include <iostream>
using namespace std;

int main()
{
    int x = 63;      // xはint型の変数（63で初期化）
    int y = 18;      // yはint型の変数（18で初期化）

    cout << "xの値は" << x << "です。\n";                  // xの値を表示
    cout << "yの値は" << y << "です。\n";                  // yの値を表示
    cout << "合計は" << x + y << "です。\n";               // xとyの合計を表示
    cout << "平均は" << (x + y) / 2 << "です。\n";         // xとyの平均を表示

    return 0;
}
```

実行結果

```
xの値は63です。
yの値は18です。
合計は81です。
平均は40です。
```

■ 初期化と代入

本プログラムで行っている《初期化》と、**List 1-5**（p.16）で行った《代入》は、値を入れるタイミングが異なります。以下のように理解しましょう（**Fig.1-13**）。

- **初期化**：変数を生成するときに値を入れる作業。
- **代　入**：生成済みの変数に値を入れる作業。

▶ ここに示したような短く単純なプログラムでは、代入と初期化の違いは大きくありません。ただし、第９章以降の《クラス》を用いたプログラムでは、その違いが明確になります（p.378）。
なお、本書では、初期化を指定する記号 = を細字で、代入演算子 = を**太字**で示して区別しています。

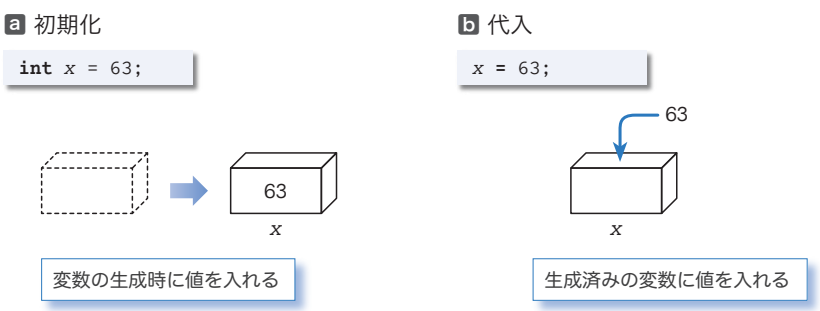

Fig.1-13　初期化と代入

□ 演習 1-8

int 型の変数に実数値の初期化子を与えるとどうなるか。プログラムを作成して確認せよ。

1-4 キーボードからの入力

変数を使うことの最大のメリットは、自由に値を変えられることです。キーボードから読み込んだ値を変数に入れる方法を学習します。

■ キーボードからの入力

キーボードから二つの整数値を読み込んで、それらに対して加算・減算・乗算・除算を行った結果を表示しましょう。そのプログラムを **List 1-8** に示します。

List 1-8 Chap01/list0108.cpp

```
// 二つの整数値を読み込んで加減乗除した値を表示

#include <iostream>
using namespace std;

int main()
{
    int x;          // 加減乗除する値
    int y;          // 加減乗除する値

    cout << "xとyを加減乗除します。\n";

    cout << "xの値：";      // xの値の入力を促す
    cin >> x;               // xに整数値を読み込む

    cout << "yの値：";      // yの値の入力を促す
    cin >> y;               // yに整数値を読み込む

    cout << "x + yは" << x + y << "です。\n";   // x + yの値を表示
    cout << "x - yは" << x - y << "です。\n";   // x - yの値を表示
    cout << "x * yは" << x * y << "です。\n";   // x * yの値を表示
    cout << "x / yは" << x / y << "です。\n";   // x / yの値を表示（商）
    cout << "x % yは" << x % y << "です。\n";   // x % yの値を表示（剰余）

    return 0;
}
```

実行例

```
xとyを加減乗除します。
xの値：7 ↵
yの値：5 ↵
x + yは12です。
x - yは2です。
x * yは35です。
x / yは1です。
x % yは2です。
```

キーボードから入力された数値を変数に格納するのが網かけ部です。

初登場の cin（一般に“シーイン”と発音します）は、キーボードと結び付いた**標準入力ストリーム**（*standard input stream*）です。そして、その cin に対して使う >> は、入力ストリームから文字を取り出す働きをする**抽出子**（*extractor*）です。

入力ストリーム cin から流れてくる文字を数値として取り出し、その値を変数に格納する様子を示したのが **Fig.1-14** です。

▶ int 型では無限に大きな（あるいは小さな）値を表現できないため、キーボードから入力する値は **List 4-2**（p.111）の実行によって得られる範囲に収まっていなければなりません。また、アルファベットや記号文字など数字以外の文字を入力しないようにしましょう。

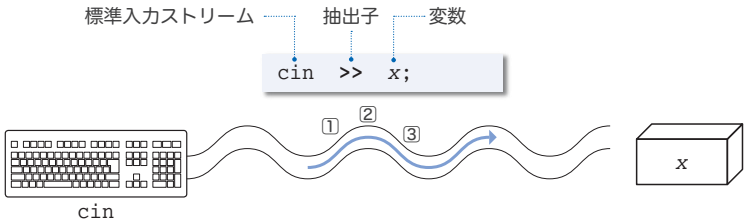

Fig.1-14　キーボードからの入力とストリーム

■ 演算子とオペランド

本プログラムで初めて使っているのが、減算を行う -、乗算を行う *、除算の剰余すなわち“余り”を求める % です。

演算を行う + や - などの記号を**演算子**（*operator*）と呼び、演算の対象となる式のことを**オペランド**（*operand*）と呼びます。

たとえば、*x* と *y* の和を求める式 *x* + *y* において、演算子は + であって、オペランドは *x* と *y* の二つです（**Fig.1-15**）。

▶ 左側のオペランドを第１オペランドあるいは左オペランドと呼び、右側のオペランドを第２オペランドあるいは右オペランドと呼びます。

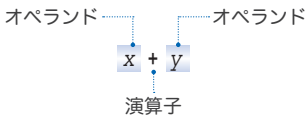

Fig.1-15　演算子とオペランド

このように二つのオペランドをもつ演算子を**２項演算子**（*binary operator*）と呼びます。

２項演算子のほかにも、オペランドが一つの**単項演算子**（*unary operator*）と、オペランドが三つの**３項演算子**（*ternary operator*）があります。

List 1-8 で利用している演算子 +，-，*，/，% をまとめて、一般に**算術演算子**（*arithmetic operator*）と呼びます。これらの演算子の概略を **Table 1-2** と **Table 1-3** に示します。

■Table 1-2　加減演算子（additive operator）

x + *y*	*x* に *y* を加えた結果を生成。
x - *y*	*x* から *y* を減じた結果を生成。

■Table 1-3　乗除演算子（multiplicative operator）

x * *y*	*x* に *y* を乗じた値を生成。
x / *y*	*x* を *y* で割った商を生成（*x*，*y* ともに整数であれば小数点以下は切り捨て）。
x % *y*	*x* を *y* で割った剰余を生成（*x*，*y* ともに整数でなければならない）。

■ 連続した読込み

cout に対して挿入子 << を連続して適用できるのと同様に、cin に対して抽出子 >> を連続して適用すると、2 個以上の変数の値を一度に読み込むことができます。

そのように書きかえたプログラムを **List 1-9** に示します。

List 1-9 Chap01/list0109.cpp

```
// 二つの整数値を読み込んで加減乗除した値を表示

#include <iostream>
using namespace std;

int main()
{
    int x;          // 加減乗除する値
    int y;          // 加減乗除する値

    cout << "xとyを加減乗除します。\n";

    cout << "xとyの値：";          // xとyの値の入力を促す
    cin >> x >> y;                 // xとyに整数値を読み込む

    cout << "x + yは" << x + y << "です。\n";    // x + yの値を表示
    cout << "x - yは" << x - y << "です。\n";    // x - yの値を表示
    cout << "x * yは" << x * y << "です。\n";    // x * yの値を表示
    cout << "x / yは" << x / y << "です。\n";    // x / yの値を表示（商）
    cout << "x % yは" << x % y << "です。\n";    // x % yの値を表示（剰余）

    return 0;
}
```

実行例

```
xとyを加減乗除します。
xとyの値：7 5↵
x + yは12です。
x - yは2です。
x * yは35です。
x / yは1です。
x % yは2です。
```

読込みを行うのが網かけ部です。このように抽出子 >> を連続して適用した場合は、**先頭側（左側）の変数から順に値が読み込まれます。**

抽出子 >> を使った入力では、スペース文字・タブ文字・改行文字などの空白文字は**読み飛ばされる**ことになっています。ここに示す《実行例》では、二つの整数値 7 と 5 の間にスペース文字を入れています。7 が x に入力され、5 が y に入力されることになります。

スペース文字が読み飛ばされますから、

```
7    5↵
```

と 7 の前にスペースを入れたり、7 と 5 の間に複数のスペースを入れたり、5 の後にスペースを入れたりすることができます。

また、改行文字が読み飛ばされることを利用して、以下のように数値ごとに Enter キー（Return キー）を打ち込むこともできます。

```
7↵
5↵
```

▶ 負の値に / 演算子や % 演算子を適用した演算結果は処理系に依存します。p.25 の **Column 1-1** で解説しています。

■ 単項の算術演算子

整数値を読み込んで、その値の符号を反転した値を表示するプログラムを **List 1-10** に示します。

List 1-10 Chap01/list0110.cpp

```
// 整数値を読み込んで符号を反転した値を表示

#include <iostream>
using namespace std;

int main()
{
    int a;                      // 読み込む値

    cout << "整数値：";         // 値の入力を促す
    cin >> a;                   // aに整数値を読み込む

1   int b = -a;         // aの符号を反転した値でbを初期化
2   cout << a << "の符号を反転した値は" << b << "です。\n";

    return 0;
}
```

実行例 1

```
整数値：7 ↵
7の符号を反転した値は-7です。
```

実行例 2

```
整数値：-15 ↵
-15の符号を反転した値は15です。
```

変数 b を宣言する 1 に着目しましょう。このように、（たとえ main 関数の途中であっても）必要になった箇所で変数を宣言するのが一般的です。

重要 変数は必要になった時点で宣言せよ。

変数 b は -a で初期化されています。ここでの - 演算子は単項演算子であり、**オペランドの符号を反転した値**を生成します。

左オペランドから右オペランドを引いた値を求める加減演算子（p.21）ではありません。すなわち - には、**単項演算子版と 2 項演算子版の二つがある**わけです。

＊

なお、+ 演算子にも単項演算子版があります。冗長となるため、あまり使われることはありませんが、+a は a の値そのものを表します。

この演算子を利用すると、2 の部分は以下のようにも実現できます。

```
cout << +a << "の符号を反転した値は" << b << "です。\n";
```

単項版の演算子 + および - の概略を **Table 1-4** に示します。

■Table 1-4 単項の算術演算子

+x	x そのものの値を生成。
-x	x の符号を反転した値を生成。

■ 実数値の読込み

整数を表す **int** 型は、小数点以下の部分をもつ実数を扱うことができないのでしたね（p.15）。実数は、**double** という型によって扱うことができます。

二つの実数値を読み込んで加減乗除するプログラムを **List 1-11** に示します。

List 1-11 Chap01/list0111.cpp

```
// 二つの実数値を読み込んで加減乗除した値を表示

#include <iostream>
using namespace std;

int main()
{
    double x;           // 加減乗除する値
    double y;           // 加減乗除する値

    cout << "xとyを加減乗除します。\n";

    cout << "xの値：";       // xの値の入力を促す
    cin >> x;               // xに実数値を読み込む

    cout << "yの値：";       // yの値の入力を促す
    cin >> y;               // yに実数値を読み込む

    cout << "x + yは" << x + y << "です。\n";   // x + yの値を表示
    cout << "x - yは" << x - y << "です。\n";   // x - yの値を表示
    cout << "x * yは" << x * y << "です。\n";   // x * yの値を表示
    cout << "x / yは" << x / y << "です。\n";   // x / yの値を表示

    return 0;
}
```

実行例

```
xとyを加減乗除します。
xの値：7.5
yの値：5.25
x + yは12.75です。
x - yは2.25です。
x * yは39.375です。
x / yは1.42857です。
```

▶ もちろん小数点以下の部分のない値を入力しても構いません。たとえば、*x* の値として 5 を入力したい場合は、5.0 と入力しても、単に 5 と入力してもよいことになっています。

本プログラムでは剰余を求めていません。**Table 1-3**（p.21）に示すように、剰余を求める **%** 演算子を**実数型に適用することはできない**からです。

重要 実数型の値に % 演算子を適用することはできない。

これ以降、原則として、整数は **int** 型の変数で表し、実数は **double** 型の変数で表すことにします。

▶ 実数の剰余を求めるプログラムは **List 2-14**（p.55）に示します。実数型を表すための**浮動小数点型**に関する詳細は第 4 章で学習します。

演習 1-9

右に示すように、キーボードから読み込んだ整数値に 10 を加えた値と 10 を減じた値を出力するプログラムを作成せよ。

```
整数値：7
10を加えた値は17です。
10を減じた値は-3です。
```

演習 1-10

二つの実数値を読み込み、その和と平均を求めて表示するプログラムを作成せよ。

```
xの値：7.5
yの値：5.25
合計は12.75です。
平均は6.375です。
```

演習 1-11

実数値として半径を読み込んで、円の面積を表示するプログラムを作成せよ。

円周率は 3.14 とすること。

```
円の面積を求めます。
半径：7.2
面積は162.7776です。
```

■ Column 1-1　除算の演算結果

除算を行う / 演算子と % 演算子は、オペランドのどちらか一方でも負であれば、演算結果が処理系によって異なります。

■オペランドが両方とも正符号

すべての処理系で、商も剰余も正の値となります。例を示します。

	x / y	x % y
正 ÷ 正　※ x = 22 で y = 5	4	2

■オペランドの少なくとも一方が負符号

/ 演算子の結果が“代数的な商以下の最大の整数”と“代数的な商以上の最小の整数”のいずれとなるのかは、処理系に依存します。以下に例を示します。

	x / y	x % y	
負 ÷ 負　※ x = -22 で y = -5	4	-2	} どちらになるかは処理系依存
	5	3	
負 ÷ 正　※ x = -22 で y = 5	-4	-2	} どちらになるかは処理系依存
	-5	3	
正 ÷ 負　※ x = 22 で y = -5	-4	-2	} どちらになるかは処理系依存
	-5	3	

■ 乱数の生成

キーボードから値を読み込むのではなく、コンピュータに値を作ってもらうこともできます。そのプログラムを **List 1-12** に示します。

List 1-12 Chap01/list0112.cpp

```
// 0〜9のラッキーナンバーを乱数で生成して表示

#include <ctime>                                    1
#include <cstdlib>
#include <iostream>
using namespace std;

int main()
{
    srand(time(NULL));                              2
                                                    3
    int lucky = rand() % 10;          // 0〜9の乱数

    cout << "今日のラッキーナンバーは" << lucky << "です。\n";

    return 0;
}
```

実行例

```
今日のラッキーナンバーは6です。
```

このプログラムは、0 から 9 まの数値の一つを《ラッキーナンバー》として生成して表示します。

コンピュータが生成するランダムな値のことを**乱数**と呼びます。1と2は、乱数の生成に必要な《決まり文句》です。

▶ 2は必ず3より前になければなりません。したがって、本プログラムのように、**main** 関数の最初に2を書くとよいでしょう。

肝心なのは3です。***rand()*** と書いた部分は、**0 以上のランダムな整数値である乱数となります**（負にはなりません）。

生成される乱数は大きな値となる可能性がありますから、本プログラムは 10 で割った余りをラッキーナンバーとして求めています。非負の整数値を 10 で割った余りを求めるのですから、*lucky* の値は必ず 0 以上 9 以下の整数値となります。

□ 演習 1-12

以下に示す三つのプログラムを作成せよ。

・1 桁の正の整数値（すなわち 1 以上 9 以下の値）をランダムに生成して表示。
・1 桁の負の整数値（すなわち -9 以上 -1 以下の値）をランダムに生成して表示。
・2 桁の正の整数値（すなわち 10 以上 99 以下の値）をランダムに生成して表示。

□ 演習 1-13

キーボードから読み込んだ整数値プラスマイナス 5 の範囲の整数値をランダムに生成して表示するプログラムを作成せよ。

※キーボードから読み込んだ値が 100 であれば、95 〜 105 の整数値を表示すること。

■ Column 1-2　乱数の生成について

乱数の生成に必要な❶、❷、❸について現時点では理解する必要はありません。第４章や第６章などの学習が終了した後に、本 **Column** を読むとよいでしょう。

＊

乱数を生成する ***rand*** 関数は、0 以上 **RAND_MAX** 以下の乱数を返します。<cstdlib> ヘッダで定義される **RAND_MAX** の値は処理系に依存しますが、少なくとも 32,767 であることが保証されます。

以下に示すのは、二つの乱数を生成するプログラム部分です。

```
#include <cstdlib>
// … 中略 …
int x = rand();          // 0以上RAND_MAX以下の乱数を生成
int y = rand();          // 0以上RAND_MAX以下の乱数を生成
cout << "xの値は" << x << "で、 yの値は" << y << "です。\n";
```

上記のプログラムを実行すると、*x* と *y* は異なる値として表示されます。

ところが、このプログラムを何度実行しても常に同じ値が表示されます（*x* と *y* の値は異なるのですが、*x* の値は毎回同じになり、*y* の値も毎回同じになります）。

このことは、生成される乱数の系列、すなわちプログラム中で１回目に生成される乱数、２回目に生成される乱数、３回目に生成される乱数、… が決まっていることを表しています。たとえば、ある処理系では、常に以下の順で乱数が生成されます。

16,838 → 5,758 → 10,113 → 17,515 → 31,051 → 5,627 → …

というのも、***rand*** 関数は“種”を利用した計算によって乱数を生成しているからです。“種”の値が ***rand*** 関数の中に埋め込まれているため、毎回同じ系列の乱数が生成されるのです。

種の値を変更するのが ***srand*** 関数です。たとえば、

```
srand(50);                    // 種の値を50に設定
```

と呼び出すだけで、種の値を変更できます。

もっとも、このように定数を渡して ***srand*** 関数を呼び出しても、その後に ***rand*** 関数が生成する乱数の系列は決まったものとなってしまいます。先ほど例を示した処理系では、種を 50 に設定すると、生成される乱数は以下のようになります。

22,715 → 22,430 → 16,275 → 21,417 → 4,906 → 9,000 → …

そのため、***srand*** 関数に与える引数は、ランダムな乱数でなければなりません。

しかし、『乱数を生成する準備のために乱数が必要である』というのも、おかしな話です。

そこで、よく使われる手法の一つが、***srand*** 関数に対して《現在の時刻》を与える方法です。プログラムは以下のようになります。

```
#include <ctime>
#include <cstdlib>
// … 中略 …
srand(time(NULL));       // 現在の時刻から種を決定
int x = rand();          // 0以上RAND_MAX以下の乱数を生成
int y = rand();          // 0以上RAND_MAX以下の乱数を生成
cout << "xの値は" << x << "で、yの値は" << y << "です。\n";
```

time 関数が返却するのは **time_t** 型で表現された《現在の時刻》です。プログラムを実行するたびに時刻は変わるわけですから、その値を種にすると、生成される乱数の系列もランダムなものとなります。

なお、***time*** 関数の詳細は **Column 10-4**（p.328）で解説しています。

■ 文字の読込み

文字を読み込むプログラムを作りましょう。文字を 1 文字だけ読み込んで、それを表示するプログラムを **List 1-13** に示します。

List 1-13 Chap01/list0113.cpp

```
// 文字を読み込んで表示

#include <iostream>
using namespace std;

int main()
{
    char c;      // 文字

    cout << "文字を入力してください：";      // 文字の入力を促す
    cin >> c;                                // 文字を読み込む

    cout << "打ち込んだ文字は" << c << "です。\n";       // 表示

    return 0;
}
```

実行例

```
文字を入力してください：x↵
打ち込んだ文字はxです。
```

文字を扱うのは **char** 型です。抽出子 **>>** はスペースや改行などの空白文字を読み飛ばしますから（p.22）、スペースやタブは無視されます。したがって、空白以外の最初の文字が変数 *c* に読み込まれることになります。

▶ **char** 型の詳細は第 4 章で学習します。

■ 文字列の読込み

次に文字列（文字の並び）を読み込むプログラムを作りましょう。名前を入力させて、その名前に対する挨拶を表示するプログラムを **List 1-14** に示します。

List 1-14 Chap01/list0114.cpp

```
// 名前を読み込んで挨拶する

#include <string>
#include <iostream>
using namespace std;

int main()
{
    string name;     // 名前

    cout << "お名前は：";        // 名前の入力を促す
    cin >> name;                 // 名前を読み込む（スペースは無視）

    cout << "こんにちは" << name << "さん。\n";      // 挨拶する

    return 0;
}
```

実行例 1

```
お名前は：柴田望洋↵
こんにちは柴田望洋さん。
```

実行例 2

```
お名前は：柴田 望洋↵
こんにちは柴田さん。
```

文字列を扱うのは ***string*** 型です。この型を利用する際はヘッダ <string> をインクルードする必要があります。

▶ ***string*** については、第 13 章で学習します。

抽出子 >> による読込みでは空白文字が読み飛ばされます。文字列の途中にスペース文字を入れて入力する実行例②では、"柴田" のみが *name* に読み込まれています。

＊

スペースも含めて 1 行分全体を読み込むプログラムを **List 1-15** に示します。

List 1-15 Chap01/list0115.cpp

```
// 名前を読み込んで挨拶する（スペースも読み込む）

#include <string>
#include <iostream>
using namespace std;

int main()
{
    string name;     // 名前

    cout << "お名前は：";          // 名前の入力を促す
    getline(cin, name);           // 名前を読み込む（スペースも読み込む）

    cout << "こんにちは" << name << "さん。\n";      // 挨拶する

    return 0;
}
```

実行例 1

```
お名前は：柴田望洋↵
こんにちは柴田望洋さん。
```

実行例 2

```
お名前は：柴田 望洋↵
こんにちは柴田 望洋さん。
```

スペースを含めた文字列の読込みは、***getline***(cin, 変数名) によって行います。Enter キー（Return キー）より前に打ち込んだすべての文字が、文字列型の変数に格納されることになります。

＊

string 型の変数の初期化や代入も可能です。プログラム例を **List1-16** に示します。

List 1-16 Chap01/list0116.cpp

```
// 文字列の初期化と代入

#include <string>
#include <iostream>
using namespace std;

int main()
{
    string s1 = "ABC";       // 初期化
    string s2 = "XYZ";       // 初期化
    s1 = "FBI";              // 代入（値を書きかえる）

    cout << "文字列s1は" << s1 << "です。\n";        // 表示
    cout << "文字列s2は" << s2 << "です。\n";        // 表示

    return 0;
}
```

実行結果

```
文字列s1はFBIです。
文字列s2はXYZです。
```

まとめ

- C++ は、C 言語と Simula67 をもとにして作られた、オブジェクト指向プログラミングをサポートする言語である。
- C++ のプログラムはコンパイルやリンクを行って実行できる形式に変換しなければならない。
- C++ のプログラムは自由形式である。インデントを与えて読みやすいものとし、作成者自身を含めた《読み手》に伝えるべき適切なコメントを記入すべきである。
- 標準ライブラリの利用にあたっては、該当するヘッダをインクルードするとともに、指令 **using namespace** `std;` が必要である。
- C++ のプログラムの本体は **main** 関数であり、その中に書かれた文が順次実行される。原則として文はセミコロンで終わる。
- 画面やキーボードなどのストリームへの入出力を行うには `<iostream>` ヘッダのインクルードが必要である。
- コンソール画面を表す標準出力ストリームは `cout` であり、挿入子 **<<** によって出力を行うことができる。
- キーボードを表す標準入力ストリームは `cin` であり、抽出子 **>>** によって入力を行うことができる。空白文字は読み飛ばされる。
- 整数定数は整数リテラルとして表し、文字の並びは文字列リテラル `"…"` として表す。ホワイトスペースをはさんで隣接した文字列リテラルは連結される。
- 改行文字は **\n** で表し、警告文字（一般にはビープ音）は **\a** で表す。
- 変数は型から作られた実体である。変数を使うには名前を与える宣言が必要である。
- 変数を生成する際に値を入れるのが初期化であり、生成済みの変数に値を入れるのが代入である。明示的に初期化しない変数は原則として不定値となる。
- 整数を表すのは **int** 型、実数を表すのは **double** 型、文字を表すのは **char** 型である。

- 文字列を表すのは ***string*** 型である。***string*** 型を利用するには <string> ヘッダのインクルードが必要である。

- 演算を行うための + や * などの記号が演算子であり、単項演算子、２項演算子、３項演算子がある。演算の対象となる式がオペランドである。

- （ ）で囲まれた演算は優先的に実行される。

- 整数 / 整数によって得られる商は、小数点以下の部分が切り捨てられた値である。

- 商を求める / 演算子と剰余を求める % 演算子は、オペランドの一方でも負であれば演算結果が処理系に依存する。% 演算子のオペランドは整数でなければならない。

- 乱数は ***rand*** 関数によって生成できる。