第1章

画面への出力とキーボードからの入力

問題1–1

▶『新版 明解C++ 入門編』List 1-1(p.4)

コンソール画面に『初めてのC++プログラム。』と『画面に出力しています。』を、連続して一行ずつ表示するプログラムを作成せよ。

```
// 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";
}
```

実行結果

```
初めてのC++プログラム。
画面に出力しています。
```

ソースプログラムとソースファイル

コンソール画面に表示を行うプログラムです。大文字と小文字は区別されますので、テキストエディタなどを使って、ここに示すとおりにプログラムを打ち込みましょう。

▶ プログラム中の余白や " などの記号を全角文字で打ち込んではいけません（記号文字の読み方は、p.5 の **Table 1-1** にまとめています）。余白の部分は、スペース・タブ・リターン（エンター）のキーを使って打ち込みます。

なお、環境によっては、**逆斜線＝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。

私たち人間は、アルファベット・数字・記号などで構成された《文字の並び》としてプログラムを作成します。このようなプログラムを**ソースプログラム**（*source program*）と呼び、ソースプログラムを格納したファイルのことを**ソースファイル**（*source file*）と呼びます。

打ち込んだソースファイルは、mondai0101.cpp などの適当な名前を与えた上で保存します。ただし、拡張子は .cpp ではなく .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、必要であれば変更しましょう。

▶ source は、“もとになるもの”という意味です。処理系は、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++、GNU C++ など数多くの処理系があります。

プログラムの実行

C++ は、C 言語や Simula をもとに作られた、**オブジェクト指向プログラミングをサポートするプログラミング言語です**。コンピュータは、C++ のソースプログラムを直接理解して実行することはできません。そのため、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**（*bit*）は、binary digit（2 進数字）の略であり、0 または 1 の値をもつデータ単位です。1 ビットでは、0 と 1 の 2 種類の数を表せます。

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作業を行いましょう。

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（左ページのプログラムリスト内）に示すように、コンソール画面への出力が行われます。

▶ ソースプログラムに綾（あや）り間違いなどがあると、コンパイルエラーが発生し、その旨の**診断メッセージ**（*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。

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、

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。

という表明です（**Fig.1-1** a）。すなわち、プログラムそのものというよりも、プログラムに対する**注釈＝コメント**（*comment*）です。他人が作成したプログラムに適切なコメントが書かれていれば、読むときに理解しやすくなります。また、自分が作ったプログラムのすべてを永遠に記憶することなど不可能ですから、コメントの記入は作成者自身にとっても重要です。

コメントの有無や内容は、**プログラムの動作に影響を与えません。作成者自身を含めた《読み手》に伝えるべきことを、簡潔な言葉（日本語や英語など）で記述しましょう。**

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、図 b のように複数行にわたるコメントの記述に効果的です。

▶ 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。

a //形式 … 行末までがコメント

```
// 画面への出力を行うプログラム
```

b /* … */形式 … 複数行にまたがれる

```
/*
    画面への出力を行うプログラム
*/
```

Fig.1-1 コメントの二つの形式

ヘッダとインクルード

コメントの次の行は、以下の表明を行う指令です。

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。

#include 指令の行が、<iostream> の内容とそっくり入れかえられます。その結果、入出力ライブラリの利用に必要な情報が手に入るのです。

<iostream> の他にも、<string>, <iomanip>, <cstdlib> などが提供されます。これらは**ヘッダ**（*header*）と呼ばれます。< > の中の iostream や string がヘッダ名です。

#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。

▶『新版 明解C++ 入門編』演習 1-1(p.7)

問題1–2

前問のプログラムのヘッダ <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検証せよ。

```
// 画面への出力を行うプログラム（#include指令が欠如）
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";
}
```

実行結果

コンパイルエラーとなるため実行できません。

インクルード指令の欠如

前問のプログラムから、“**#include** <iostream>”のインクルード指令を削除したプログラムです。コンパイルエラーとなるため、実行することはできません。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ます。**ライブラリ利用にあたっては、ヘッダのインクルードが不可欠です。**

コンソール画面への出力とストリーム

コンソール画面を含め、ファイルなどの外部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“川”のようなものです（**Fig.1-2**）。**外部への入出力は、文字が流れる川であるストリームを経由して行います。**

網かけ部で利用されている cout は、コンソール画面と結び付いたストリームであって、**標準出力ストリーム**（*standard output stream*）と呼ばれます。ストリームへの出力は、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。

▶ これ以降、コンソール画面のことを、単に「画面」と呼ぶことにします（cout は“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）。

なお、二つの < は連続しなければなりません。< と < のあいだにスペースやタブを入れないようにしましょう。ヘッダ名 iostream は**入出力ストリーム**（*input-output stream*）の略です。

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。

▶ リテラルとは、『文字どおりの』『文字で表された』という意味です。なお、cout に挿入したときに画面に " が表示されることはありません。

改行

文字列リテラル中の **\n** は《改行文字》を表す特別な表記です。改行文字を出力すると、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。

▶ 二つの文字 \ と **n** が表すのは、《改行文字》という**単一の文字**です。改行文字のように、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。

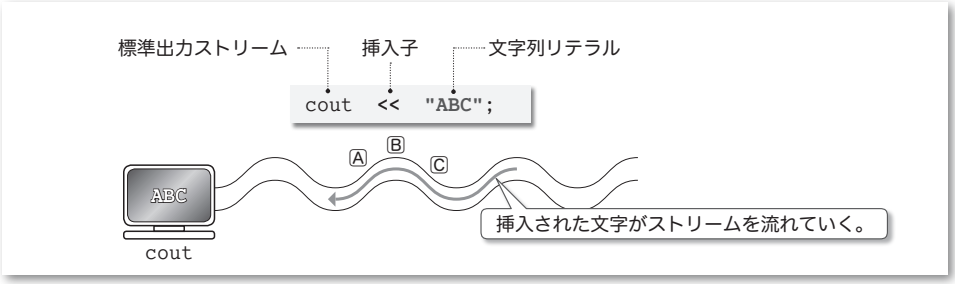


Fig.1-2 コンソール画面への出力とストリーム

▶ 注意：日本語版の MS-Windows などでは、逆斜線 \ の代わりに円記号 ¥ を使います。たとえば、本プログラムの網かけ部は、以下のようになります。

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

みなさんの環境に応じて、必要ならば読みかえるようにしてください。

本書では、最後に改行文字を出力する場合は“『ABC』と表示”と表現して、最後に改行文字を出力しない場合は“「ABC」と表示”と表現します。

Table 1-1 記号文字の読み方

記号	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、バックスラ、バック　※JIS コードでは ¥
¥	円記号、円、円マーク　注意!!
%	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
'	単一引用符、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
?	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
< >	小(大)なり、左(右)向き不等号
( )	左(右)括弧、左(右)丸括弧、左(右)小括弧、パーレン
{ }	左(右)波括弧、左(右)中括弧、ブレイス
[ ]	左(右)角括弧、左(右)大括弧、ブラケット

問題1–3

▶『新版 明解C++ 入門編』演習 1-2(p.7)

問題 1-1（p.2）のプログラムから、**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。

```
// 画面への出力を行うプログラム（using指令を利用しない：解答例Ａ）
#include <iostream>
int main()
{
    std::cout << "初めてのＣ++プログラム。\n";
    std::cout << "画面に出力しています。\n";
}
```

実行結果

```
初めてのＣ++プログラム。
画面に出力しています。
```

```
// 画面への出力を行うプログラム（using指令を利用しない：解答例Ｂ）
#include <iostream>
int main()
{
    std::cout << "初めてのＣ++プログラム。\n"
              << "画面に出力しています。\n";
}
```

std 名前空間と using 指令

問題 1-1 のプログラムから、**using 指令**（*using directive*）である **"using namespace** std;**"** を削除したプログラムです。この指令は、以下のことを表明します。

> **std という名前空間**（*name space*）**を使います。**

名前空間については第 9 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）に由来します。

本プログラムのように **using** 指令を省略する場合は、cout を std::cout に変更しなければなりません。もし、本プログラムの std::cout を、単なる cout に書きかえると、コンパイルエラーとなります。

ここでは、二つのプログラム例を示しています。

▫ 解答例Ａ

出力を二つの文で行っています（**問題 1-1** と同じです）。

▶ 文については、次問で学習します。

▫ 解答例Ｂ

出力を一つの文で行っています（最初の出力の箇所にはセミコロンが付いていないことに注意しましょう）。出力ストリーム cout に対して複数個の挿入子 **<<** 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る**ことを利用しているのです。

もちろん、以下のように、一行にまとめることもできます。

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

▶『新版 明解C++ 入門編』演習 1-3(p.9)

問題1–4

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検証せよ。

```
// 画面への出力を行うプログラム（文の終端のセミコロンが欠如）
#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"
}
```

実行結果

コンパイルエラーとなるため実行できません。

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.1-3** です（ここには、セミコロンを削除する前の、正しいプログラムを示しています）。

この部分は **main 関数**（*main function*）と呼ばれます。**main 関数は C++ のプログラムの本体です**。プログラムを起動して実行すると、**main** 関数中の**文**（*statement*）が順次実行されることになっています。

▶ **int main**() や { } は、後の章で学習しますので、いずれも《決まり文句》として覚えましょう。なお、“関数”については、第 6 章以降で詳しく学習します。

Fig.1-3 プログラムの実行と main 関数

文

図中に示しているプログラムの **main** 関数には二つの文があります。**文はプログラムの実行単位です**。日本語の文の末尾に句点 。を置くのと同様で、C++ の文の末尾にはセミコロン；が必要です（例外もあります）。

本プログラムでは、文の末尾に必要な；が欠如しているため、コンパイルエラーとなります。

▶ コメントは文ではありません。《コメント文》といった文は存在しません。

問題1–5

▶『新版 明解C++ 入門編』演習 1-4(p.14)

1 行に 1 文字ずつ自分の名前を表示するプログラムを作成せよ。

```
// 1 行に 1 文字ずつ自分の名前を表示
#include <iostream>
using namespace std;
int main()
{
    cout << "柴\n田\n望\n洋\n";
}
```

実行結果

```
柴
田
望
洋
```

自由形式記述

本プログラムでは、1 文字表示するたびに改行文字を出力することによって、名前を 1 行に 1 文字ずつ表示しています。

さて、C++ のプログラムは、「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。」といった制約を受けません。自由な桁位置にプログラムを記述できる**自由形式**（*free formatted*）が許されます。自由とはいえ、いくつかの制限があります。

▪ 単語の途中に空白類文字を入れてはいけない

int, **main**, cout, **<<**, //, /*, */ などは、それぞれが『単語』です。これらの途中に**空白類文字**（ホワイトスペース）（空白文字・改行文字・水平タブ文字・垂直タブ文字・書式送り文字）を入れることはできません。

▪ 文字列リテラルの途中で改行してはいけない

文字の並びを二重引用符 " で囲んだ文字列リテラル "…" も、一種の単語です。したがって、途中で改行することはできません。

なお、**空白類文字をはさんで隣接している文字列リテラルは、連結されて単一の文字列リテラルとみなされます**。そのことを利用すると、本プログラムは、以下のように実現できます。

```
    cout << "柴\n田\n"              // 姓
            "望\n洋\n";             // 名
```

二つの文字列リテラル "柴\n田\n" と "望\n洋\n" が連結されて 1 個の文字列リテラルとなります。なお、このように、二つの文字列リテラルのあいだには注釈（コメント）があっても構わないことになっています。プログラムが翻訳される最初のほうの段階で、注釈が 1 個の空白文字に置換されることになっているからです。

なお、空白類文字と注釈の総称が、**空白類**（ホワイトスペース）（*white space*）です。

長い文字列リテラルは、空白類（空白類文字と注釈）をはさんで、分割して表記するとよいでしょう。

▶ 連結される文字列リテラルは 2 個に限られるわけではありません。たとえば、空白類をはさんだ 3 個の文字列リテラル "ABCD" "EFGH" "IJKL" も、きちんと連結されて 1 個の文字列リテラル "ABCDEFGHIJKL" となります。

問題1–6

▶『新版 明解C++ 入門編』演習 1-5(p.14)

1行に1文字ずつ自分の名前を表示するプログラムを作成せよ。姓と名のあいだを1行あけること。

```
// 1行に1文字ずつ自分の名前を表示（姓と名のあいだを1行あける）
#include <iostream>
using namespace std;
int main()
{
    cout << "柴\n田\n\n望\n洋\n";
}
```

実行結果

```
柴
田

望
洋
```

空の行の出力

姓と名のあいだに改行文字を表す拡張表記を2個並べています（網かけ部）。姓を表示した後に、改行文字が2個出力されるため、姓と名とのあいだが1行あくことになります。

自由形式記述と前処理指令

自由形式記述には、もう一つ制限があります。

▪ 前処理指令の途中で改行してはいけない

先頭が # 文字である **#include** などの指令は**前処理指令**（*preprocessing directive*）と呼ばれます。前処理指令は、単一行で書くのが原則です。途中で改行する必要がある場合は、以下のように、行末に逆斜線 \ を書くことになっています。

```
#include \
    <iostream>
```

▶ 逆斜線文字と改行文字が連続していると、コンパイルの最初の段階で、それらの2文字が取り除かれる（その結果、次の行とつながる）ことになっています。そのため、逆斜線 \ は、改行文字の直前に置かなければなりません。

インデント

main 関数の中に書かれている文は、左から数えて5桁目から記述されています（これまでのプログラムは、すべてこのようになっています）。

{ } は、ひとまとまりの文をくくったものであり、日本語での“段落”のようなものです（詳細は次章で学習します）。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はっきりします。そのための余白のことを**インデント**（段付け／字下げ）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**と呼びます。

本書のプログラムは、4桁ごとのインデントを与えて表記しています。

▶ すなわち、左側の余白は、階層の深さに応じて 0, 4, 8, 12, … 個分の空白となります。以下に示すのは、**問題 3-21**（p.102）のプログラムの一部です。

```
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++)    // n個の'*'を表示
        cout << '*';
    cout << '\n';
}
```

問題1–7

▶『新版 明解C++ 入門編』演習1-6(p.19)

二つの **int** 型変数に 63 と 18 を代入した上で、その合計と平均を求めて表示するプログラムを作成せよ。小数部をもつ実数値を変数に代入するとどうなるのかの確認も行うこと。

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xとyの合計と平均を表示
#include <iostream>
using namespace std;
int main()
{
❶   int x;       // xはint型の変数
    int y;       // yはint型の変数

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

❸   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

❹   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
}
```

変数と宣言

二つの変数に値を代入して、その合計と平均を表示するプログラムです。

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です。

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。箱には《**名前**》がないと困ります。

そのため、変数を使うには、箱を作るとともに、名前を与える**宣言**（*declaration*）が必要です。❶は、*x* と *y* という名前の変数を宣言する**宣言文**（*declaration statement*）です。

int（イント）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）と、名前が *y* である変数（箱）が作られます（**Fig.1-4**）。

変数 *x* と *y* が保持できる値は整数に限られています。たとえば 3.5 といった小数部をもつ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* が **int** 型の**実体**というわけです。

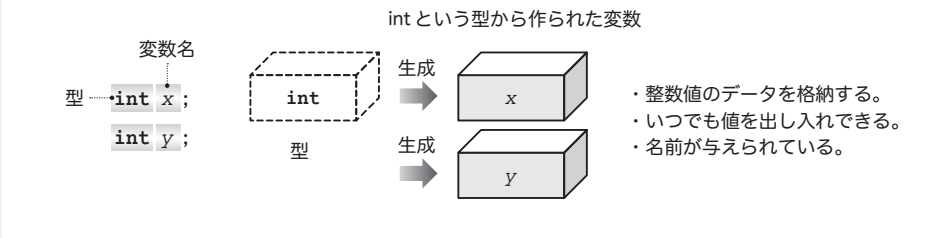


Fig.1-4　変数と宣言

なお、二つ以上の変数を一度にまとめて宣言することもできます。以下のように、各変数名をコンマ文字 , で区切って宣言します。

```
int x, y;          // int型の変数xとyをまとめて宣言
```

ただし、解答プログラムのように、各変数を個別に宣言したほうが、個々の宣言に対する注釈（コメント）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）。

▶ 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 章以降で学習します。また、名前の与え方に関する規則は次章で学習します。

代入演算子

二つの変数に値を入れる**2**に着目しましょう。ここで使われている = 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.1-5 **a**に示すように、変数 *x* に 63 が代入され、変数 *y* に 18 が代入されます。

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』と解釈されているのではないことに注意しましょう。

▶ 演算子の詳細は、次章で学習します。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。

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。

▶ 整数リテラル 18 は単一の数値 十（じゅう）八（はち）で、文字列リテラル "18" は 2 個の文字 一（いち）と 八（はち）が並んだものです。整数リテラルの詳細は、第 4 章で学習します。

2を以下のように書きかえてみましょう（図**b**）。

```
x = 63.7;   // xに63.7を代入
y = 18.9;   // yに18.9を代入
```

```
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。
```

そうすると、右に示す実行結果が得られます。**int 型の変数に実数値を代入すると、小数部は切り捨てられる**のです（四捨五入されるわけではないことに注意しましょう）。

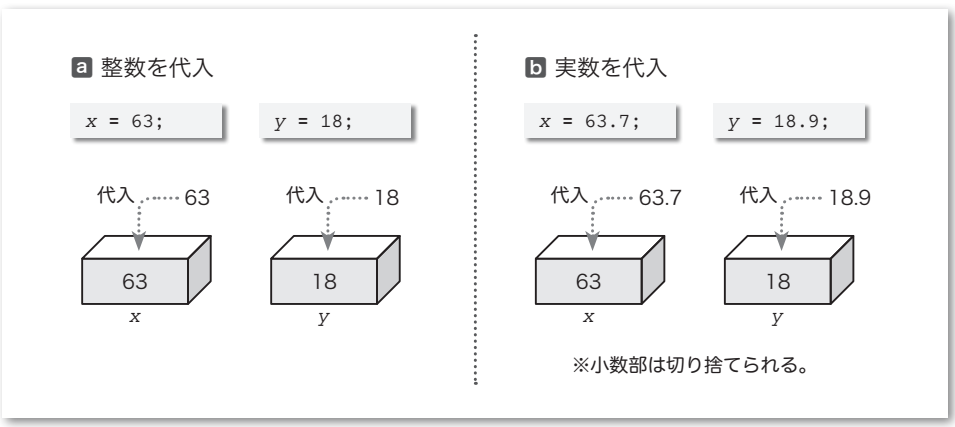


Fig.1-5 int 型変数への値の代入

変数の値の表示

変数に格納されている値は、いつでも取り出せます。3では、変数の値を取り出して表示しています。変数 *x* の値を表示する様子を示したのが **Fig.1-6** です。

cout に挿入されている二つの文字列リテラル "**x**の値は" 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 *x* は、変数名「x」が表示されるのではありません。変数の値である「63」として表示されます。

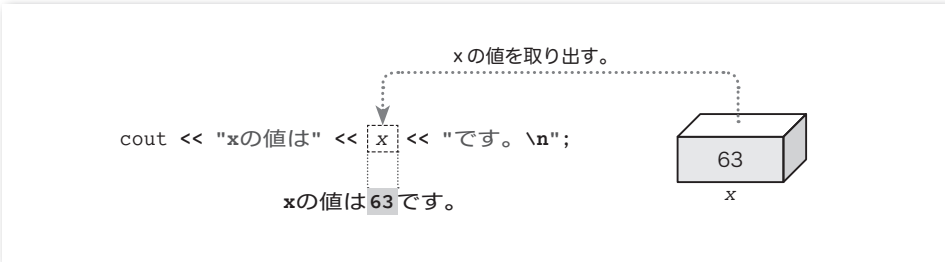


Fig.1-6　ストリームへの変数の値の出力

算術演算子と演算のグループ化

4で表示しているのは、*x* と *y* の合計 *x* + *y* と、平均（*x* + *y*）/ 2 です。

平均を求める計算では、式 *x* + *y* が（）で囲まれています。この（）は、優先的に演算を行うための記号です。**Fig.1-7** a に示すように、まず *x* + *y* の加算が行われ、それから 2 で割る除算が行われます。スラッシュ記号 / は除算を行う記号です。

もしも図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。

▶　すべての演算子と優先順位は、次章で学習します。

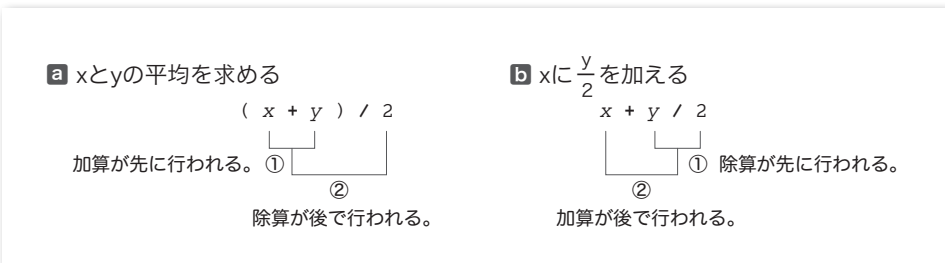


Fig.1-7　（）による演算順序の変更

なお、“整数 / 整数”の演算では、**小数部（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。

▶『新版 明解C++ 入門編』演習1-7(p.19)

問題1–8

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。

```
// 三つの変数xとyとz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
    int z;      // zはint型の変数

    x = 63;     // xに63を代入
    y = 18;     // yに18を代入
    z = 34;     // zに34を代入

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

実行結果

```
xの値は63です。
yの値は18です。
zの値は34です。
合計は115です。
平均は38です。
```

三値の合計と平均

前問のプログラムは、二つの変数の合計と平均を求めるプログラムでした。本問のプログラムは、変数が三つに増えています。

行っていることは、前問のプログラムと、基本的に同一です。

変数と初期化

本プログラムから、変数に値を代入する網かけ部を削除するとどうなるかを実験してみましょう。この部分を削除した上で、コンパイル・実行してみてください。おそらく、三つの変数が妙な値として表示されるはずです。

```
xの値は6936です。
yの値は2358です。
zの値は-3597です。
合計は5697です。
平均は1899です。
```

▶ この値は、実行環境や処理系によっても異なりますし、同一環境であっても、プログラムを実行するたびに異なる値となる可能性があります。なお、**実行時エラーが発生して、プログラムの実行が中断される場合もあります。**

変数が生成される際は、**不定値**すなわち**ゴミの値**が入れられます。そのため、値が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、その生成時に自動的に 0 が入れられます。詳しくは第 6 章で学習します。

問題1–9

▶『新版 明解C++ 入門編』演習1-8(p.21)

二つの **int** 型変数を 63 と 18 で初期化した上で、その合計と平均を求めて表示するプログラムを作成せよ。初期化子として小数部をもつ実数値を与えるとどうなのかの確認も行うこと。

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
}
```

初期化を伴う宣言

変数に入れる値が事前に分かっているのであれば、その値を最初から変数に入れておくべきです。本プログラムは、その方針に基づいて作られたプログラムです。

網かけ部の宣言によって、変数 *x* と変数 *y* は 63 と 18 という値で**初期化**（*initialize*）されます。**Fig.1-8** に示すように、変数の宣言における = 記号以降の部分は、変数の生成時に入れる値を指定するものであり、**初期化子**（*initializer*）と呼ばれます。

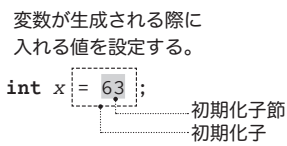


Fig.1-8 初期化を伴う宣言

▶ 標準 C++ では、= 記号を含めた = 63 が**初期化子**と呼ばれ、= 記号より右側の 63 は**初期化子節**（*initializer clause*）と呼ばれます。ただし、C 言語を含めた、他のプログラミング言語では、後者を初期化子と呼ぶのが一般的です。本書でも、文法的な厳密性が要求されない文脈では、後者のことを初期化子と呼びます。

初期化と代入

本プログラムで行っている《初期化》と、これまでのプログラムで行っていた《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.1-9**）。

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

▶ 本書では、初期化を指定する記号 = を細字で示し、代入演算子 **=** を**太字**で示すことによって区別しやすくしています。

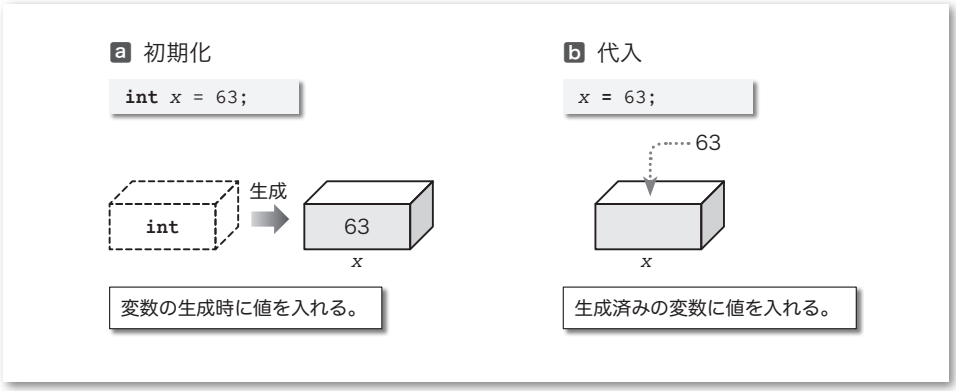


Fig.1-9 初期化と代入

▶ 本問のような、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 10 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。

プログラムの網かけ部を以下のように書きかえてみましょう。

```
int x = 63.7;    // xを63.7で初期化
int y = 18.9;    // yを18.9で初期化
```

```
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。
```

そうすると、右に示す実行結果が得られます。**int** 型の変数を実数値で初期化すると、小数部は切り捨てられることが分かりますね（代入の場合と同じです。四捨五入されるわけではないことに注意しましょう）。

C++ の誕生

1980 年頃、AT&T ベル研究所の Bjarne Stroustrup 博士が事象駆動型のシミュレーションの記述のために、C 言語を拡張したプログラミング言語を作りました。それは、**クラス付きの C**（*C with classes*）と呼ばれる言語であり、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**などの機能をもつものでした。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。

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。その後、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 の後ろに ++ という記号を付加したものとなっています。ちなみに、++ は、C 言語の演算子の一つであり、以下の機能をもちます。

値を 1 単位だけ増やす。

“D” などといったネーミングに比べると、C++ という名称は控え目です。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ことを示しています。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。

▶『新版 明解C++ 入門編』演習1-9(p.27)

問題1–10

キーボードから読み込んだ整数値をそのまま反復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整数値をそのまま反復して表示
#include <iostream>
using namespace std;
int main()
{
    int x;                      // 読み込む値
    cout << "整数値：";         // xの値の入力を促す
    cin >> x;                   // xに整数値を読み込む
    cout << x << "と入力しましたね。\n";          // xの値を反復して表示
}
```

実行例

```
整数値：7↵
7と入力しましたね。
```

キーボードからの入力

整数値を読み込んで、その値を反復表示するプログラムです。

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ている**標準入力ストリーム**（*standard input stream*）です。そして、その cin に対して適用している **>>** 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.1-10** です。

▶ プログラムから **using** 指令を削除する場合、cin は std::cin としなければなりません（cout の場合と同じです）。

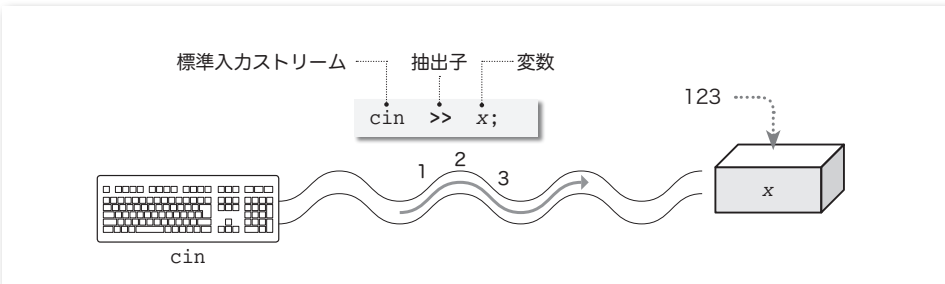


Fig.1-10 キーボードからの入力とストリーム

▶ **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**問題 4-4**（p.132）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。

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。

▶『新版 明解C++ 入門編』演習1-10(p.27)

問題1–11

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。

実行例

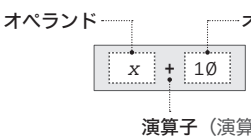
```
整数値：7↵
10を加えた値は17です。
10を減じた値は-3です。
```

```
// キーボードから読み込んだ整数値の±10の値を表示
#include <iostream>
using namespace std;
int main()
{
    int x;                      // 読み込む値
    cout << "整数値：";         // xの値の入力を促す
    cin >> x;                   // xに整数値を読み込む
    cout << "10を加えた値は" << x + 10 << "です。\n";    // 10を加えた値を表示
    cout << "10を減じた値は" << x - 10 << "です。\n";    // 10を減じた値を表示
}
```

演算子とオペランド

これまでのプログラムでは、加算を行う+、減算を行う-、乗算を行う*、除算を行う/といった記号を利用してきました。これらの記号を**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 + 10において、演算子は+であって、オペランドはxと10の二つです（**Fig.1-11**）。



※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ぶ。

Fig.1-11 演算子とオペランド

四則演算で利用される演算子+，-，*，/，%は、**算術演算子**（*arithmetic operator*）と呼ばれます。算術演算子の概略をまとめたのが、**Table 1-2**（左ページ）と**Table 1-3**です。

いずれも二つのオペランドをもつ演算子です。このような演算子は、**２項演算子**（*binary operator*）と呼ばれます。

なお、２項演算子のほかに、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*）とがあります。

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る）。
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）。

問題1–12

キーボードから読み込んだ整数値の最下位桁を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整数値の最下位桁を表示
#include <iostream>
using namespace std;
int main()
{
    int n;                      // 読み込む値
    cout << "整数値：";         // nの値の入力を促す
    cin >> n;                   // nに整数値を読み込む
    cout << "最下位桁は" << n % 10 << "です。\n";    // nの最下位桁を表示
}
```

実行例

```
整数値：3176↵
最下位桁は6です。
```

剰余

本プログラムでは、除算の剰余すなわち“あまり”を求める%演算子を利用しています。たとえば、3176を10で割った剰余は6です。

ただし、変数nに読み込んだ値が負の場合は、うまくいきません。というのも、除算を行う/演算子と%演算子の演算結果は、処理系によって異なるからです。

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

xとyの符号とは無関係に(yが0でない限り)、(x / y) * y + x % yの値はxと一致します。

本プログラムを実行して、変数nに読み込まれた数値が-3176であれば、最下位桁として表示されるのは、-6あるいは4となります。

*

前ページの**Table 1-3**に示すように、**剰余を求める%演算子のオペランドは整数型でなければなりません。実数型のオペランドに%演算子を適用することはできません。**

▶『新版 明解C++ 入門編』演習1-11(p.27)

問題1–13

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。

```
// 二つの実数値を読み込んで和と平均を表示
#include <iostream>
using namespace std;
int main()
{
    double x;       // 加える値
    double y;       // 加える値
    cout << "xとyの値：";    // xとyの値の入力を促す
    cin >> x >> y;          // xとyに実数値を読み込む
    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
}
```

実行例

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

実数値の取扱い

整数を表すための **int** 型が、小数部をもつ実数を扱えないことは、既に学習しました。実数は、**double** という型によって扱えます。本プログラムでは、変数 *x* と *y* を **double** 型としています。

連続した読込み

挿入子 **<<** を cout に連続適用すると複数の値を一度に出力できるのでしたね。抽出子 **>>** を cin に対して連続適用すると、複数の変数の値を一度に読み込めます。

二つの変数 *x* と *y* への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・タブ・改行などの空白文字が読み飛ばされる**ことになっています。ここに示す実行例では、二つの数値 7.5 と 5.25 のあいだにスペース文字を入れています。そのため、7.5 が *x* に入力され、5.25 が *y* に入力されます。

スペース文字が読み飛ばされますから、

```
7.5    5.25↵
```

と 7.5 の前にスペースを入れたり、7.5 と 5.25 のあいだに複数のスペースを入れたり、5.25 の後にスペースを入れたりすることもできます。

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エンターキー（リターンキー）を打ち込むこともできます。

```
7.5↵
5.25↵
```

小数部のない値を打ち込む際は、小数点を含めて、それ以降は省略できます。たとえば 5.0 は、5 と入力しても、5.0 と入力しても、5. と入力してもよいことになっています。

問題1–14

▶『新版 明解C++ 入門編』演習1-12(p.27)

三角形の底辺と高さを読み込んで、その面積を表示するプログラムを作成せよ。

```
// 三角形の底辺と高さから面積を求めて表示
#include <iostream>
using namespace std;
int main()
{
    double width;       // 底辺
    double height;      // 高さ
    cout << "底辺：";  cin >> width;          // 底辺を読み込む
    cout << "高さ：";  cin >> height;         // 高さを読み込む
    double area = width * height / 2.0;      // 底辺
    cout << "面積は" << area << "です。\n";   // 面積を表示
}
```

実行例

```
底辺：7.8↵
高さ：3.2↵
面積は12.48です。
```

浮動小数点リテラル

本プログラムでは、三角形の面積を（底辺 * 高さ）/ 2.0 で求めています。

2.0 のように、小数部をもつ実数を表す定数のことを**浮動小数点リテラル**（*floating-point literal*）と呼びます。

▶ これ以降は、次問（右ページ）の解説です。

定値オブジェクト

半径が r である球の表面積と体積を求める公式を以下に示します（π は円周率です。）。

- 表面積 … $4\,\pi\,r^2$
- 体　積 … $\frac{4}{3}\,\pi\,r^3$

これらの値は、以下に示す式で求めることができます。

- 表面積 … 4 * 3.14 * r * r
- 体　積 … 4 * 3.14 * r * r * r / 3

もっとも、円周率 π は 3.14 ではなくて、3.1415926535…と無限に続く値です。円周の長さと面積をより正確に求めるためには、3.14 を 3.1416 とか 3.14159 などに変えることになります。この場合、変更は２箇所だけです。ただし、大規模な数値計算プログラムであれば、プログラム中に 3.14 が数百箇所あるかもしれません。

エディタの《置換》機能を使えば、すべての 3.14 を 3.14159 に変更するのは容易です。とはいえ、円周率ではない値として、たまたま 3.14 を使っている箇所がプログラム中にあるかもしれません。そのような箇所は、置換の対象から外す必要があります。すなわち、選択的な置換が要求されるわけです。

このようなケースで効力を発揮するのが、値を書きかえることのできない**定値オブジェクト**（*const object*）です。網かけ部の宣言に付いている **const** によって、変数 *PI* は 3.14 で初期化された定値オブジェクトになります。

▶ “オブジェクト”については、第４章で学習します。現時点では、『変数』を意味する専門用語である、と理解しておくとよいでしょう。

問題1–15

球の半径を読み込んで、その表面積と体積を表示するプログラムを作成せよ。

```
// 球の半径から表面積と体積を求めて表示
#include <iostream>
using namespace std;
int main()
{
    const double PI = 3.14; // 円周率
    double r;               // 半径

    cout << "球の半径：";   // 半径の入力を促す
    cin >> r;               // 半径を読み込む

    cout << "表面積は" << 4 * PI * r * r << "です。\n";
    cout << "体　積は" << 4 * PI * r * r * r / 3 << "です。\n";
}
```

実行例

```
球の半径：5.7↵
表面積は408.074です。
体　積は775.341です。
```

本プログラムでは、円周率が必要な計算で変数*PI*の値を利用しています。定値オブジェクトを利用するメリットは、以下のとおりです。

① 値の管理を一箇所に集約できる

円周率の値3.14は、変数*PI*の初期化子として与えられています。もし他の値（たとえば3.14159）に変える場合は、プログラムの変更が一箇所だけですみます。

タイプミスやエディタ上での置換操作の失敗などによって、たとえば3.14と3.14159とを混在させてしまう、といったミスを防げます。

② プログラムが読みやすくなる

プログラムの中では、数値ではなく変数名*PI*で円周率を参照できますから、プログラムが読みやすくなります。

プログラム中に埋め込まれた数値は、何を表すものであるのかが理解しにくくなります。定値オブジェクトとして宣言して名前を与えるとよいでしょう。

本プログラムのように、定値オブジェクトの変数名を大文字にすると、**const**でない普通の変数と見分けやすくなります。

＊

定値オブジェクトの値を書きかえることはできません。そのため、宣言時には必ず初期化子を与えなければなりません。以下のプログラムは、コンパイルエラーとなります。

```
const double PI;     // コンパイルエラー（初期化子が欠如）
PI = 3.14;           // コンパイルエラー（constオブジェクトに対する代入）
```

▶ **const**は、オブジェクトの型の属性を指定する**cv修飾子**の一つです。cv修飾子には、**const**の他に**volatile**があります。

問題1–16

▶『新版 明解C++ 入門編』演習1-13(p.30)

以下に示す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。

実行例

```
1桁の正の乱数：4です。
1桁の負の乱数：-7です。
2桁の正の乱数：93です。
```

```
// 1桁あるいは2桁の乱数を生成
#include <ctime>                                    ❶
#include <cstdlib>
#include <iostream>

using namespace std;

int main()
{
    srand(time(NULL));  // 乱数の種を設定          ❷
    cout << "1桁の正の乱数：" <<  1 + rand() %  9 << "です。\n";
    cout << "1桁の負の乱数：" << -1 - rand() %  9 << "です。\n";
    cout << "2桁の正の乱数：" << 10 + rand() % 90 << "です。\n";
}                                                   ❸
```

乱数の生成

乱数を生成して表示するプログラムです。乱数とは、コンピュータが生成するランダムな値のことです。生成するのは、1桁あるいは2桁の値です。

プログラム中の❶と❷と❸は、乱数の生成に必要な《決まり文句》です。

▶ ❷は必ず❸より前に置く必要があります。

肝心なのは❸です。*rand*()と書かれた部分は、**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、9あるいは90で割った剰余を利用して、以下のように求めています。

- 1桁の正の乱数 … 1に0～8を加えた値（1～9）。
- 1桁の負の乱数 … -1から0～8を減じた値（-1～-9）。
- 2桁の正の乱数 … 10に0～89を加えた値（10～99）。

▶ 非負の整数値を9で割った剰余は0以上8以下の整数値となり、90で割った剰余は0以上89以下の整数値となります。

単項の算術演算子

-1における - は、**2項演算子でなく単項演算子です。Table 1-4** に示すように、演算子 + と - には、単項演算子版があります。単項 + 演算子は、オペランドの値そのものを生成し、単項 - 演算子は、オペランドの符号を反転した値を生成します。

Table 1-4 単項の算術演算子（正符号演算子と負符号演算子）

+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。

Column 1-1 乱数の生成について

乱数の生成に必要な❶、❷、❸は、現時点では理解する必要はありません。後半の章まで学習が進んでから本 **Column** を読むとよいでしょう。

乱数を生成する ***rand*** 関数が返却するのは、0 以上 **RAND_MAX** 以下の値です。<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。

さて、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>
using namespace std;
// … 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";
```

こ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示しています。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（たね）”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) とか ***srand***(23) 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>
using namespace std;
// … 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（***time*** 関数については、p.364 の **Column 11-3** で学習します）。

なお、***rand*** 関数が生成するのは、擬似乱数と呼ばれる乱数です。擬似乱数は、乱数のように見えますが、ある一定の規則に基づいて生成されます。擬似乱数と呼ばれるのは、次に生成される数値の予測がつくからです。本当の乱数は、次に生成される数値の予測がつきません。

▶『新版 明解C++ 入門編』演習1-14(p.30)

問題1–17

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。たとえば、キーボードから読み込んだ値が 100 であれば、95 以上 105 以下の乱数を生成して表示すること。

```
// キーボードから読み込んだ整数値プラスマイナス5の範囲の乱数を生成して表示
#include <ctime>
#include <cstdlib>
#include <iostream>

using namespace std;

int main()
{
    srand(time(NULL));      // 乱数の種を設定
    int n;                  // キーボードから読み込む値

    cout << "整数値：";  cin >> n;                    // 整数値を読み込む

    cout << "その値の±5の乱数を生成しました。それは" << n - 5 + rand() % 11
         << "です。\n";
}
```

実行例

```
整数値：100 ↵
その値の±5の乱数を生成しました。それは103です。
```

一定範囲の乱数の生成

キーボードから読み込んだ整数値プラスマイナス 5 の範囲の乱数を生成するプログラムです。乱数として出力しているのは、網かけ部の式の値です。

実行例のように、変数 n に読み込んだ値が 100 であれば、n - 5 は 95 です。その 95 に対して、0 以上 11 未満（0 以上 10 以下）の乱数を加えると 95 以上 105 以下の整数となります。

標準 C++

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：Japanese 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++ の規格は第 2 版までが制定されています。

- 第 1 版：1998 年に制定された規格です。ANSI 規格と ISO 規格がありますが、JIS 規格はありません。
- 第 2 版：第 1 版に対して小改訂を施した規格です。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。

問題1–18

姓と名のイニシャルを読み込んで挨拶するプログラムを作成せよ。

```
// イニシャルを読み込んで挨拶
#include <iostream>
using namespace std;
int main()
{
    char initial1, initial2;
    cout << "姓のイニシャルは：";   cin >> initial1;
    cout << "名のイニシャルは：";   cin >> initial2;
    cout << "\aこんにちは" << initial2 << "." << initial1 << ".さん。\n";
}
```

実行例

```
姓のイニシャルは：S↵
名のイニシャルは：B↵
♪こんにちはB.S.さん。
```

char 型と文字の読込み

文字を読み込むプログラムです。

文字を表すのは **char** 型です。抽出子 **>>** がスペースや改行などの空白文字を読み飛ばすため、キーボードから入力された空白文字が変数に格納されることはありません。各変数に読み込まれるのは、空白以外の最初の文字となります。たとえば、□□S（スペース２個の後ろに S）と入力されると、読み込まれるのはスペースではなく S です。

▶ **char** 型の詳細は、第４章で学習します。

警報

文字列リテラル中の **\a** は《警報》を表す拡張表記です。cout に対して警報文字を挿入すると、視覚的あるいは聴覚的な注意を促せるようになっており、ほとんどの実行環境では、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、警報を♪と表記します。

標準C

C++ の親である C 言語の標準規格には、以下に示す二つの版があります。

- 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 **“C89”** 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 と ISO の制定年から **“C99”** と呼ばれます。

新しく制定された第２版は、第１版との互換性が乏しいこともあり、あまり使われていないのが実情です。本書では、第１版と第２版共通の内容を『**標準C**』と呼び、第１版のみに該当する内容を『$\mathbf{C}_{89}$』、第２版のみに該当する内容を『$\mathbf{C}_{99}$』と呼びます。

問題1–19

キーボードから姓と名を読み込んで挨拶するプログラムを作成せよ。

```
// 姓と名を読み込んで挨拶
#include <string>
#include <iostream>
using namespace std;
int main()
{
    string last, first;
    cout << "姓は：";   cin >> last;
    cout << "名は：";   cin >> first;
    cout << "こんにちは" << last << first << "さん。\n";
}
```

実行例1

```
姓は：柴田↵
名は：望洋↵
こんにちは柴田望洋さん。
```

実行例2

```
姓は：柴 田↵
名は：こんにちは柴田さん。
```

文字列

単一の文字ではなく、文字の並びである文字列を読み込んで表示するプログラムです。文字列を扱うのは ***string*** 型です。この型の利用時は、<string> ヘッダのインクルードが必要であることを覚えておきましょう。

▶ もしプログラム冒頭に“**using namespace** std;”の指令がなければ、プログラム中のすべての ***string*** を std::***string*** に変更する必要があります。cout の場合と同じです。

抽出子 **>>** による読込みでは、空白文字が読み飛ばされます。そのため、文字列の途中に空白文字を入れて入力する実行例②では、"柴" が *last* に読み込まれ、"田" が *first* に読み込まれます。

スペースも含めて 1 行分全体を読み込むのであれば、網かけ部は以下のようにしなければなりません。

```
    getline(cin, last);          // 姓を読み込む（スペースも読み込む）
    getline(cin, first);         // 名を読み込む（スペースも読み込む）
```

***getline*(cin, 変数名)** では、スペースを含めた文字列の読込みが行えます。リターンキー（エンターキー）より前に打ち込んだすべての文字が、文字列型の変数 *last* と *first* とに格納されることになります。

＊

以下に示すのは、***string*** 型の変数の初期化と代入を行うプログラム部分です。

変数 *s1* は、いったん "ABC" で初期化された後に、"FBI" が代入されています。表示されるのは、代入後の文字列です。

```
    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
```

```
文字列s1はFBIです。
文字列s2はXYZです。
```