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。

- C++ の歴史
- ソースプログラムとコンパイル／リンク／実行
- #include 指令によるヘッダのインクルード
- using 指令と std 名前空間
- main 関数と文
- コメント（注釈）
- 自由形式記述とインデント
- <iostrem> ヘッダと入出力ストリーム
- cout への << による挿入／ cin からの >> による抽出
- 改行 \n と警報 \a
- 型（int 型／ double 型／ char 型）
- <string> ヘッダと文字列と string 型
- 文字列リテラル／整数リテラル／浮動小数点リテラル
- 変数の宣言
- 初期化と代入
- 定値オブジェクト
- 演算子とオペランド
- 算術演算子
- 乱数の生成

1–1 C++の歴史

まずは、C++の歴史を簡単に学習しましょう。

C++の歴史

1979年、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**などの機能をもっていました。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**）。

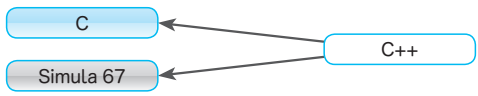

Fig.1-1 C++とその両親

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。1983年には、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の後ろに++という記号を付加したものです。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（第3章で学習します）。

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。

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。

Stroustrup博士ら多くの人々の努力によって《標準規格》が制定されるとともに、改訂を続けています。正式には、『第1版』、『第2版』、…ですが、一般的には、制定された西暦年の下2桁を付して、『**C++98**』、『**C++03**』、『**C++11**』、『**C++14**』、『**C++17**』、…と呼ばれています。

▶ 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：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。

第2版の『C++03』は、第1版の『C++98』のマイナーチェンジ版です。

なお、親言語であるC言語も、『C89』、『C99』、『C11』、…と呼ばれています。

2013年にStroustrup博士が出版した

The C++ Programming Language Fourth Edition****

では、C++11のすべてが詳細に解説されています。

▶ **クラス付きのC**の初期の時点で、クラス、クラスの派生、アクセス制御、コンストラクタ、デストラクタ、関数引数型チェックなどの、核となる機能をもっていました。

仮想関数、多重定義、演算子多重定義、参照、入出力ストリームライブラリ、複素数ライブラリが加えられ、名称が**C++**へと変更されました。

その後、テンプレートを用いたジェネリックプログラミング、例外処理、名前空間、動的キャスト、汎用コンテナ・アルゴリズムライブラリなどが追加されてC++98が完成しました。

C++11では、統一形式の初期化構文、ムーブの概念、可変個引数テンプレート、ラムダ式、型別名、並行処理に適したメモリモデル、スレッドライブラリ、ロックライブラリなどが追加されています。

なお、本書はC++03をベースにしており、C++11の新機能などは補足的に解説しています。

---

* 邦訳：斎藤信男訳『プログラミング言語C++』，トッパン，1988

** 邦訳：足立高徳ら訳『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』，アジソンウェスレイパブリッシャーズジャパン，1998

**** 邦訳：柴田望洋訳『プログラミング言語C++第4版』，SBクリエイティブ，2015

1–2 まずは画面に表示

本節では、コンソール画面への表示を行って、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、ここに示すとおりにします。

▶ 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.v）。各プログラムリストの右上に示しているのは、ディレクトリ（フォルダ）名を含むファイル名です。

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";
}
```

実行結果

```
初めてのC++プログラム。
画面に出力しています。
```

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

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。余白や " などの記号文字を全角文字で打ち込まないよう注意しましょう。

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、p.11 の **Table 1-1** にまとめています。なお、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る環境があります。みなさんの環境に応じて、必要ならば読みかえましょう。

ソースプログラムとソースファイル

私たち人間は、プログラムを《**文字の並び**》として作成します。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。

▶ source は、『もとになるもの』という意味です。そのため、ソースプログラムは、**原始プログラム**とも呼ばれます。

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、ソースファイルの拡張子(かくちょうし)が .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変更しましょう。

▶ **処理系**は、C++ プログラムの開発に必要なソフトウェアです。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

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。私たち人間が読み書きする《**文字の並び**》を、コンピュータが理解できる 0 と 1 の並びである《**ビットの並び**》に変換する必要があります。

そこで、**Fig.1-2** に示すように、ソースプログラムを**コンパイル**（**翻訳**(ほんやく)）したり、**リンク**（**結合**(けつごう)）したりする作業を行って、**実行プログラム**を作成します。

▶ **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（10 進数の 1 桁では 0，1，2，…，9 の 10 種類の数を表せます。それが 0 と 1 だけに限定されています）。

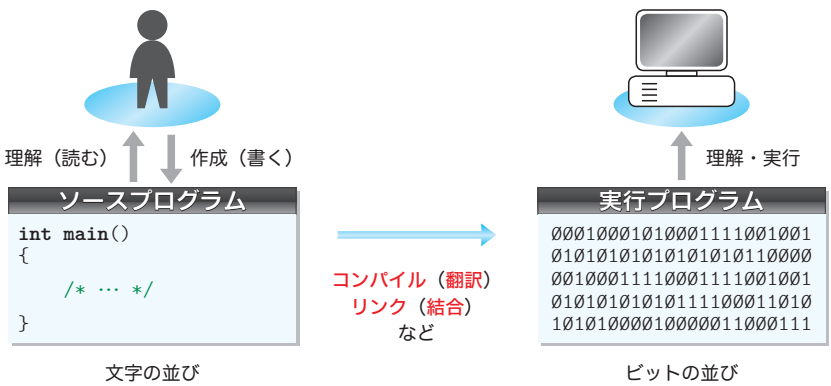

Fig.1-2　プログラムの作成から実行まで

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作業を行いましょう。

▶ 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。

コメント（注釈）

まずはプログラムの先頭行に着目します。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。

という表明です。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コメント（*comment*）すなわち注釈（ちゅうしゃく）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。作成者自身を含めて、プログラムの読み手に伝えたいことを、日本語や英語などの簡潔な言葉で記述します。

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。

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しよう。

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、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注意が必要です。なお、本書では、コメントを色のついた文字で表記します。

```
A /* 複数行にまたがることが可能な注釈 */
B // その行の終端までが注釈
```

Aは、C言語から引き継がれた形式のコメントです。Bは、C言語の祖先であるBCPLで利用されていた形式のコメントです。C言語では長いあいだ採用されていませんでしたが、C言語の誕生から30年近くたったC99で復活採用されました。

＊

形式Aを《入れ子》にする（コメントの中にコメントを入れる）ことはできません。そのため、以下のコード（プログラム）は、コンパイルエラーとなります。

```
/*  /* このようなコメントは駄目!! */   */
```

というのも、最初の */ がコメントの終了とみなされるからです。

＊

形式Aのコメントの中では // を自由に使えますし、形式Bのコメントの中では /* や */ を自由に使えます（特別扱いされずに、注釈として書かれた文字とみなされます）。以下に示すのは、いずれも正しいコメントであり、コンパイルエラーにはなりません。

```
A /* // このコメントはＯＫ!! */
B // /* このコメントもＯＫ!! */
```

なお、コメントは、プログラムがコンパイルされる最初のほうの段階で、1個の空白文字に置換されます（p.13）。

ヘッダとインクルード

コメントの次の行は、# で始っています。これは、

```
#include <iostream>
```

以下の表明です。

画面やキーボードなどに対する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。

Fig.1-3 に示すように、この **#include 指令**の行は、**<iostream>** の中身とそっくり入れかえられて、入出力ライブラリの利用に必要な情報が埋め込まれます。

<iostream> の他にも、<string> などが提供され、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。

なお、ヘッダの内容を“取り込む”ことを、**インクルード**（*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**しよう。

#include <iostream> を削除すると、プログラムはコンパイルできなくなります。確かめてみましょう（**"chap01/list0101a.cpp"**）。

▶ ヘッダファイルではなく、単に**ヘッダ**と呼ぶのは、個々のヘッダが、単独のファイルとして提供されるとは限らないからです。さらに、文字の並びであるテキストファイルではなく、コンパイルずみの特殊な形式でヘッダを提供する処理系もあります。

std 名前空間の利用

#include の次の行は、**using 指令**と呼ばれる指令であり、以下のことを表明します。

```
using namespace std;
```

std という名前空間（*name space*）を使います。

名前空間は第 9 章で学習しますので、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）。

using namespace std; の指令は削除可能です。ただし、削除する場合は、プログラム中のすべての cout を std::cout に変更する必要があります（**"chap01/list0101b.cpp"**）。

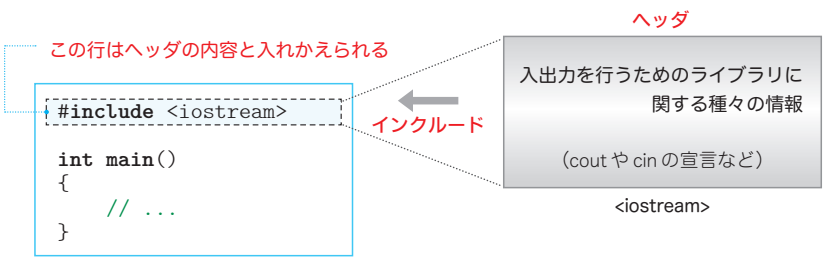

Fig.1-3 #include 指令によるヘッダのインクルード

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";
```

Fig.1-4 に示すように、コンソール画面などの外部への入出力には、**ストリーム**（*stream*）を利用します。ストリームは、文字が流れる“川（かわ）”のようなものです。

重要 外部への**入出力**は、文字が流れる川である**ストリーム**を経由して行う。

cout は、コンソール画面と結び付く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、文字の**挿入**によって行います。それを指示するのが、左向きの不等号 < が二つ並んだ **<<** です。この記号は、**挿入子（そうにゅうし）**（*inserter*）と呼ばれます。

▶ < と < のあいだにスペースやタブを入れてはなりません。

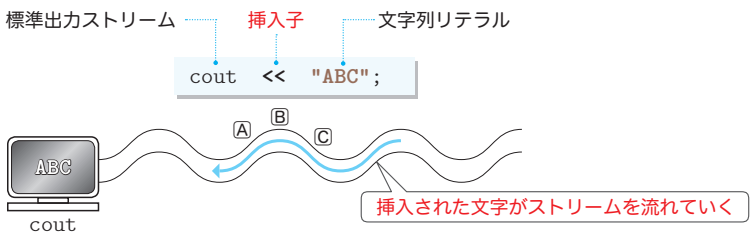

Fig.1-4 コンソール画面への出力とストリーム

▶ 以下、コンソール画面のことを、単に「画面」と呼びます。
iostream は**入出力ストリーム**（*input-output stream*）の略で、cout は、character out の略です。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。

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、《**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。本書では、文字列リテラルを "少し薄い文字" で表記します。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**を表します。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。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（第3章）。

main 関数と文

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。プログラムを起動して実行すると、main 関数中の**文**（*statement*）が順次実行されます。

重要 C++ のプログラムの本体は **main 関数**であり、プログラムを起動すると main 関数中の文が順次実行される。

▶ `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。

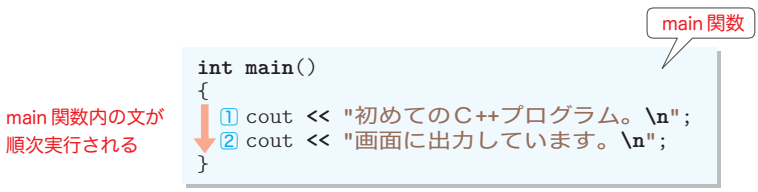

Fig.1-5　プログラムの実行と main 関数

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）。

重要 **文**は、原則としてセミコロンで終わる。

▶ 文のセミコロンが欠如していると、プログラムはコンパイルできなくなります。確かめてみましょう（`"chap01/list0101c.cpp"`）。なお、コメントは文ではありません。《コメント文》といった文は C++ には存在しません。

List 1-2 に示すのは、画面への出力を一つの文にまとめたプログラムです。

List 1-2　chap01/list0102.cpp

```
// 文字列リテラル内の改行文字\nの働きを確認

#include <iostream>

using namespace std;

int main()
{
    cout << "初めてのC++プログラム。\n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。
```

『初めての C++ プログラム。』の後ろに置かれた改行文字によって、『画面に出力しています。』が次の行に表示されます。

▶ これ以降、以下のように表現します。

- 最後に改行文字を出力する場合は　“『ABC』と表示”と表現します。
- 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。

ストリームへの連続した出力

List 1-3 は、二つの挨拶を連続して表示するプログラムです。

出力ストリーム cout に対して、複数個の挿入子 **<<** を連続して適用しています。このような場合、**先頭側（左側）から順に出力されます。**

List 1-3 chap01/list0103.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";
}
```

警報 ← `\a`　　改行 ← `\n`

実行結果

♪はじめまして。こんにちは。

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 ♪ で表記します。

インデント

main 関数の中の文は、すべて左から数えて５桁目から記述されています。

{ } は、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（**Fig.1-6**）。

階層の深さに応じてインデント（段付け／字下げ）する

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

これは、第３章で学習する **List 3-14**（p.98）の一部です。《九九の表》を出力します。

▶ インデントは、タブキーとスペースキーのいずれでもタイプできます。

ただし、エディタやその設定によっては、タブをタイプした文字と、保存したソースファイル上の文字とが一致しないことがあります。

Fig.1-6　ソースプログラム中のインデント

記号文字の読み方

C++ で利用する記号文字の読み方を、俗称(ぞくしょう)を含めてまとめたのが **Table 1-1** です。

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
%	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
(	左括弧、開き括弧、左丸括弧、始め丸括弧、左小括弧、始め小括弧、左パーレン
)	右括弧、閉じ括弧、右丸括弧、終り丸括弧、右小括弧、終り小括弧、右パーレン
{	左波括弧、左中括弧、始め中括弧、左ブレイス、左カーリーブラケット、左カール
}	右波括弧、右中括弧、終り中括弧、右ブレイス、右カーリーブラケット、右カール
[	左角括弧、始め角括弧、左大括弧、始め大括弧、左ブラケット
]	右角括弧、終り角括弧、右大括弧、終り大括弧、右ブラケット
<	小なり、左アングル括弧、左向き不等号
>	大なり、右アングル括弧、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、チルド、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、カレット、キャレット
#	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

注意 !!

▶ 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、**List 1-3** の表示を行う文は、以下のようになります。

```
cout << "¥aはじめまして。" << "こんにちは。¥n";
```

みなさんの環境が ¥ を使う環境であれば、本書のすべての \ を ¥ と読みかえてください。

自由形式記述

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

List 1-4 chap01/list0104.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"
    ;
          }
```

実行結果

```
初めてのC++プログラム。
画面に出力しています。
```

読みにくいけれども正しいプログラム

一部のプログラミング言語は『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。』などの制約を課します。しかし、C++ のプログラムは、そのような制約は受けません。自由な桁位置にプログラムを記述できる**自由形式**（*free formatted*）が採用されています。

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、いくつかの制約があります。

①単語の途中に空白類文字を入れてはならない

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。

```
ma
   in
```

②文字列リテラルの途中で改行してはならい

文字の並びを二重引用符 " で囲んだ文字列リテラル "…" も、一種の単語ですので、左下に示すように、途中での改行は不可能です。

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。すなわち、右下に示すように記述します。

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

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

このように、**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。**

List 1-5 のプログラムで確認しましょう。

List 1-5 chap01/list0105.cpp

```
// 空白類をはさむ文字列リテラルが連結されることの確認

#include <iostream>

using namespace std;

int main()
{
    cout << "ABCDEFGHIJKLMNOPQRSTUVWXYZ"      // 空白類をはさんで並んだ
            "abcdefghijklmnopqrstuvwxyz\n";   // 文字列リテラルは連結される
}
```

実行結果

```
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
```

文字列リテラル `"ABCDEFGHIJKLMNOPQRSTUVWXYZ"` と `"abcdefghijklmnopqrstuvwxyz\n"` が連結されて 1 個の文字列リテラルとなることが、実行結果からも分かります。

なお、本プログラムのように、二つの文字列リテラルのあいだには注釈（コメント）があっても構いません。プログラムをコンパイルす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。

重要 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、分割して表記できる。

▶ 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。

③前処理指令の途中で改行してはならない

先頭が `#` 文字で始まる `#include` などの指令は**前処理指令**（まえしょりしれい）（*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、行末に逆斜線 \ を書きます。

```
#include
 <iostream>
```
✕

```
#include \
 <iostream>
```
○

なお、前処理指令には、`#include` 指令の他にも、後の章で学習する `#define` 指令や `#if` 指令などがあります。

▶ 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 文字が取り除かれます（その結果、次の行とつながります）。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。

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りましょう。

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 1-6** 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 1-6 chap01/list0106.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";
}
```

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。

▶ 整数リテラル 18 は単一の数値 1 8（じゅうはち）で、文字列リテラル "18" は 2 個の文字 1（いち）と 8（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。

演算結果の出力

本プログラムでの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なく、整数と整数を加算した結果である「81」として表示されます。

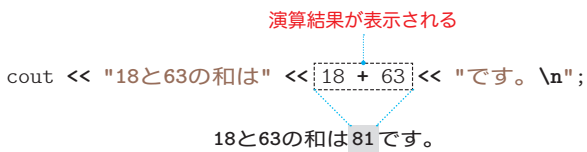

Fig.1-7 ストリームへの文字列リテラルと整数値の出力

変数

このプログラムは、18 と 63 以外の数値の和を求められません。数値を変更する際は、プログラムに手を加えた上に、コンパイル・リンクの作業も必要です。値を自由に出したり入れたりできる《**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。

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 の略です。この宣言によって、名前が *x* である変数（箱）が作られます（**Fig.1-8**）。

重要 **変数**を使うときは、まず**宣言**をして名前を与えよう。

変数 *x* が保持できる値は整数に限られます。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**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。

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
```

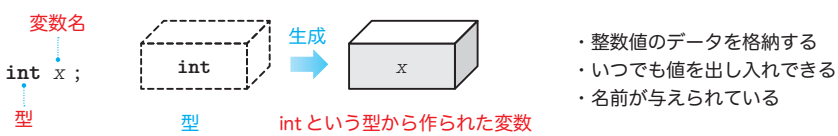

Fig.1-8　変数と宣言

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作りましょう。**List 1-7** に示すのが、そのプログラムです。

List 1-7 chap01/list0107.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
}
```

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、個別に宣言しています。こうすると、個々の宣言に対するコメントが記入しやすくなるだけでなく、宣言の追加や削除も容易になります（ただしプログラムの行数は増えてしまいます）。

代入演算子

二つの変数に値を入れる 1 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.1-9** 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。

Fig.1-9 代入演算子による変数への値の代入

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」と解釈されるのでは**ありません**。

▶ 演算子については、p.20 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。

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。2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.1-10** です。

▶ cout に挿入する x は文字列リテラルではありませんので、画面に表示されるのは、変数名である「x」ではなく、その値である「63」です。

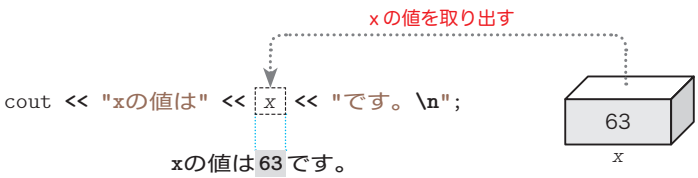

Fig.1-10 変数の値の取出しとストリームへの出力

算術演算子と演算のグループ化

3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.1-11** 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 との和が求められます。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、**Table 2-10**（p.70）で学習します。

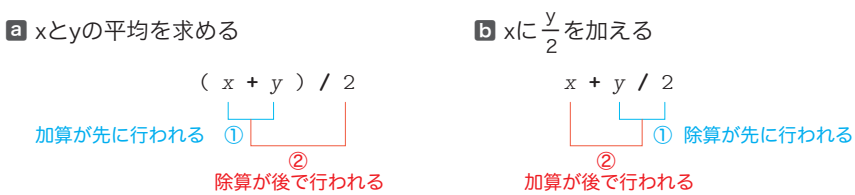

Fig.1-11 （）による演算順序の変更

なお、“整数 / 整数”の演算では、**小数部（小数点以下の部分）が切り捨てられます。**そのため、63 と 18 の平均値は 40.5 ではなく 40 となります。

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実験してみます。**List 1-8** を実行しましょう。

List 1-8 chap01/list0108.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
}
```

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異なります（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）。また、同一環境であっても、プログラムを実行するたびに値が異なる可能性もあります。

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っては、その生成時に自動的に 0 が入れられることが保証されます。詳しくは第 6 章（p.224）で学習します。

初期化を伴う宣言

変数に入れる値が事前に分かっていれば、その値を最初から変数に入れておくべきです。そのように修正したプログラムが **List 1-9** 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.1-12** 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。

重要 変数の宣言時には、**初期化子**を与えて確実に初期化しよう。

▶ 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 が**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。

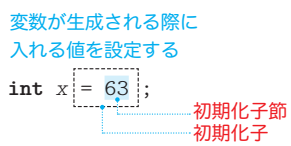

Fig.1-12 初期化を伴う宣言

List 1-9 chap01/list0109.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
}
```

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 1-7**（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こと。**
- **代　入**：**生成ずみの変数に値を入れること。**

▶ 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。

なお、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**で示すことで区別しやすくしています。

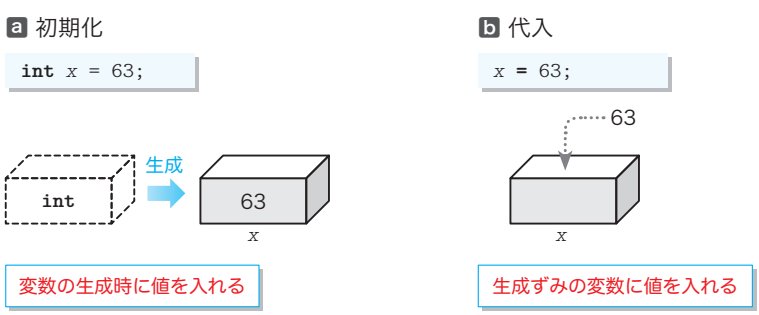

Fig.1-13　初期化と代入

▶ なお、以下の形式の初期化も行えます。

```
int x(63);
int x{63};        // この形式の宣言はC++11以降
int x = {63};     // この形式の宣言はC++11以降
```

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、自由に値を入れたり出したりできることです。本節では、キーボードから読み込んだ値を変数に入れる方法などを学習します。

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、それらに対して加減乗除の算術演算を行った結果を表示しましょう。そのプログラムを **List 1-10** に示します。

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 に対して適用している **>>** は、入力ストリームから文字を取り出す**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**抽出**して、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）値は表現できず、キーボードから入力する値は **List 4-1**（p.121）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。

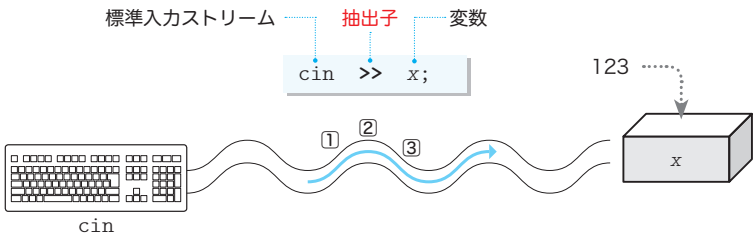

Fig.1-14 キーボードからの入力とストリーム

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《**あまり**》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**）。

本プログラムで利用している演算子 +，-，*，/，% の概略をまとめたのが、**Table 1-2** と **Table 1-3** です。なお、これらの演算子は、一般に**算術演算子**（*arithmetic operator*）と呼ばれます。

List 1-10 chap01/list0110.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）
}
```

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。
```

いずれも２個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。

２項演算子のほかに、オペランドが１個の**単項演算子**（*unary operator*）と、オペランドが３個の**３項演算子**（*ternary operator*）とがあります。

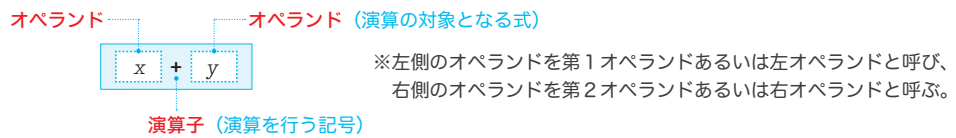

Fig.1-15 演算子とオペランド

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 を乗じた積を生成。
`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）。

値の連続した読込み

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。それを利用して書きかえたプログラムを **List 1-11** に示します。

List 1-11 chap01/list0111.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）
}
```

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。
```

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ます。**ここに示す《実行例》では、二つの整数値 7 と 5 のあいだにスペース文字を入れています。そのため、7 が x に入力され、5 が y に入力されます。

右のように、7 の前、7 と 5 のあいだ、5 の後のいずれにも（1 個以上の）スペースを入れることが可能です。

```
7 5↵
```

また、改行文字が読み飛ばされることを利用して、右のように、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↵
```

▶ 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（次ページで学習します）。

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作りましょう。**List 1-12** に示すのが、そのプログラムです。

変数 b を宣言する 1 に着目しましょう。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（**Table 1-4**）。

List 1-12 chap01/list0112.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

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

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。
```

演算子 + にも単項演算子版があります。2の +a は a の値そのものを表します。

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。

1の宣言に戻りましょう。この宣言は、main 関数の途中にあります。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原則です。

重要 変数は必要になった時点で宣言しよう。

▶ 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	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。

実数値の読込み

整数を表す **int** 型が、小数部をもつ実数を扱えないことを p.15 で学習しました。実数は、**double** という型で扱えます。

List 1-13 に示すのが、二つの実数値を読み込んで加減乗除するプログラムです。

List 1-13 chap01/list0113.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
}
```

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。
```

▶ 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 とも、5.0 とも、5. とも入力できます。

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 % 演算子のオペランドは整数型でなければならないからです。**

重要 実数型のオペランドには % 演算子は適用できない。

もし本プログラムに、以下の文を追加すると、コンパイルエラーとなります。

```
cout << "x % yは" << x % y << "です。\n";    // コンパイルエラー
```

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

▶ 実数の剰余を求める方法は、**List 2-17**（p.61）で学習します。また、実数を表すための**浮動小数点型**に関する詳細は、第 4 章で学習します。

Column 1-1 デバッグとコメントアウト

プログラムの欠陥や誤りのことを**バグ**（*bug*）といいます。また、バグを見つけたり、その原因を究明したりする作業が、**デバッグ**（*debug*）です。

デバッグの際に、『この部分が間違っているかもしれない。もしこの部分がなかったら、実行時の挙動はどう変化するだろうか。』と試しながらプログラムを修正することがあります。その際に、プログラムの該当部を削除してしまうと、もとに戻すのが大変です。

そこで、よく使われるのが**コメントアウト**という手法です。コメントとしてではなくプログラムとして記述されている部分を、コメントにしてしまうのです。

List 1-1 のプログラムを以下のように書きかえて実行してみましょう。色つき文字の部分がコメントとみなされますから、『初めての C++ プログラム。』が表示されなくなります。

List 1C-1 chap01/list01c01.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";
}
```

実行結果

```
画面に出力しています。
```

行の先頭に2個のスラッシュ記号 // を書くだけで、その行全体をコメントアウトできるわけです。プログラムをもとに戻すのも簡単です。// を消すだけです。

なお、複数行にわたってコメントアウトする際は、以下に示すように /* … */ 形式を使うとよいでしょう。

List 1C-2 chap01/list01c02.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";
*/
}
```

実行結果

```
何も表示されません。
```

なお、コメントアウトされたプログラムは、読み手にとって紛らわしく、誤解されやすいものとなります。というのも、コメント化の根拠が、その部分が不要になったためなのか、何らかのテストを目的とするものなのか、などが分からないからです。

コメントアウトの手法は、あくまでもその場しのぎのための一時的な手段と割り切って使いましょう。

なお、**#if** 指令を用いると、よりよい方法でのコメントアウトが実現できます。**Column 11-7**（p.401）で学習します。

定値オブジェクト

円の半径をキーボードから読み込んで、その円の“円周の長さ”と“面積”を求めて表示するプログラムを作りましょう。**List 1-14** に示すのが、そのプログラムです。

List 1-14 chap01/list0114.cpp

```
// 円周の長さと円の面積を求める（その１：円周率を浮動小数点リテラルで表す）

#include <iostream>

using namespace std;

int main()
{
    double r;                   // 半径

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

    cout << "円周の長さは" << 2 * 3.14 * r << "です。\n";    // 円周
    cout << "面積は" << 3.14 * r * r << "です。\n";          // 面積
}
```

実行例

```
半径：7.2
円周の長さは45.216です。
面積は162.778です。
```

本プログラムでは、公式どおりに、円周の長さと面積を求めています（半径 r の円周の長さは $2\pi r$ で、面積は πr^2 です）。

円周率 π を表す網かけ部の 3.14 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。

*

さて、円周率は 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円周率を 3.1416 にするのであれば、２箇所の網かけ部を 3.1416 に変更します。

本プログラムでは、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 3.1416 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ます。**

*

このようなケースで効力を発揮するのが、**定値オブジェクト**（*const object*）です。定値オブジェクトを用いて書きかえたプログラムが **List 1-15** です。

宣言に付いている **const** によって、変数 *PI* は 3.1416 で初期化された定値オブジェクトになります。**定値オブジェクトの値を書きかえることはできません。**

▶ 《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。

List 1-15 chap01/list0115.cpp

```
// 円周の長さと円の面積を求める（その２：円周率を定値オブジェクトで表す）

#include <iostream>

using namespace std;

int main()
{
    const double PI = 3.1416;    // 円周率
    double r;                    // 半径

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

    cout << "円周の長さは" << 2 * PI * r << "です。\n";      // 円周
    cout << "面積は" << PI * r * r << "です。\n";            // 面積
}
```

実行例

```
半径：7.2↵
円周の長さは45.239です。
面積は162.861です。
```

本プログラムでは、円周率が必要な計算では、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 3.1416 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）に変えるとしても、プログラムの変更は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 3.1416 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ので、プログラムが読みやすくなります。

重要 プログラム中に埋め込まれた数値は、何を表すのかが理解しにくい。**定値オブジェクト**として宣言して名前を与えよう。

本プログラムのように、定値オブジェクトの変数名は大文字にします。**const** でない普通の変数と見分けやすくなるからです。

▶ プログラム中に埋め込まれた、意図が分かりにくい数値は、**マジックナンバー** (*magic number*) と呼ばれます。定値オブジェクトを導入すると、マジックナンバーを除去できます。

*

定値オブジェクトの宣言時には必ず初期化子を与えなければなりません。右下のコードはコンパイルエラーとなります。

```
const double PI = 3.1416;
```

○

```
const double PI;
PI = 3.1416;
```

×

▶ **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。

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ができます。その方法を、**List 1-16** に示すプログラムで学習しましょう。

List 1-16 chap01/list0116.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));                  // 乱数の種を設定    2

    int lucky = rand() % 10;            // 0〜9の乱数
                                                        3
    cout << "今日のラッキーナンバーは" << lucky << "です。\n";
}
```

実行例

```
今日のラッキーナンバーは7です。
```

このプログラムは、0 から 9 までの数値の一つを《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（らんすう）と呼びます。1 と 2 と 3 は、乱数の生成に必要な“決まり文句”です。

▶ 2 は必ず 3 より前に置く必要があります。

肝心なのは 3 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ので、本プログラムは 10 で割った剰余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 で割った剰余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。

▶ 実行例では、乱数を 10 で割った剰余が 7 の例を示しています。乱数によって実行結果が変わる数値や、実行結果が処理系に依存する数値などは、*青の斜め文字*で表します。

*

生成する乱数の範囲を変えてみましょう。いくつかの例を示します。

```
1 + rand() % 9              //  1〜 9の乱数を生成
1 + rand() % 10             //  1〜10の乱数を生成
rand() % 100                //  0〜99の乱数を生成
10 + rand() % 90            // 10〜99の乱数を生成
```

▶ C++11 以降では、より性能で高機能な **<random> ヘッダ**によるライブラリが提供されます。

Column 1-2 乱数の生成について

乱数の生成に必要な1、2、3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、乱数を 2 個生成するプログラム部分です（"chap01/column0102a.cpp"）。

```
#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。

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、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**（"chap01/column0102b.cpp"）。

```
#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になります（time*** 関数については、**Column 11-4**（p.390）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。

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、それを反復して表示するプログラムを **List 1-17** に示します。

List 1-17 chap01/list0117.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
}
```

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（p.22）、キーボードから入力された空白文字は変数には格納されません。変数 *c* に読み込まれるのは、空白以外の最初の文字です。

▶ **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。名前を文字列として読み込んで、挨拶するプログラムを **List 1-18** に示します。

List 1-18 chap01/list0118.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
}
```

実行例 1

```
お名前は：福岡五郎↵
こんにちは福岡五郎さん。
```

実行例 2

```
お名前は：福岡 五郎↵
こんにちは福岡さん。
```

文字列を扱うのは ***string*** **型**です。この型の利用時は、**<string> ヘッダ**のインクルードが不可欠です。

▶ 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 と同じです：p.7）。

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、文字列の途中にスペース文字を入れて入力する実行例②では、"福岡" のみが *name* 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プログラムを **List 1-19** に示します。

List 1-19 chap01/list0119.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
}
```

実行例 1

```
お名前は：福岡五郎↵
こんにちは福岡五郎さん。
```

実行例 2

```
お名前は：福岡　五郎↵
こんにちは福岡　五郎さん。
```

スペースを含めた文字列の読込みを行うのが、***getline***(cin, 変数名) です。リターン（エンター）キーより前に打ち込んだすべての文字が、文字列型の変数に格納されます。

＊

List 1-20 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例です。

List 1-20 chap01/list0120.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
}
```

実行結果

```
文字列s1はFBIです。
文字列s2はXYZ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。