

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プログラムの実行
- #include 指令によるヘッダのインクルード
- using 指令
- コメント（注釈）
- 自由形式記述とインデント
- main 関数
- 文
- 入出力ストリームと挿入子・抽出子
- 改行と警報
- 型（int 型、double 型、char 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
- 文字列と string 型
- 初期化と代入
- 演算子とオペランド
- 定値オブジェクト
- 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

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 (**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。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、もとも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、以下の機能ももちます。

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。

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。

■ 標準Cと標準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。

なお、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『C₈₉』、第2版のみに該当する内容を『C₉₉』と呼びます。

* 邦訳：斎藤信男訳『プログラミング言語C++』、トッパン、1988

** 邦訳：足立高德ら訳『注解C++リファレンスマニュアル』、トッパン、1992

*** 邦訳：(株)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

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ることになります。本節では、コンソール画面への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、ここに示すとおりにします。

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";
}
```

実行結果

初めてのC++プログラム。
画面に出力しています。

- ▶ プログラム中の余白や"などの記号を全角文字で打ち込まないように注意しましょう。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、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、ディレクトリ名を含むファイル名です。

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、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。

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、**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。

■ 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、“もともになるもの”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++、GNU C++ など数多くの処理系があります。

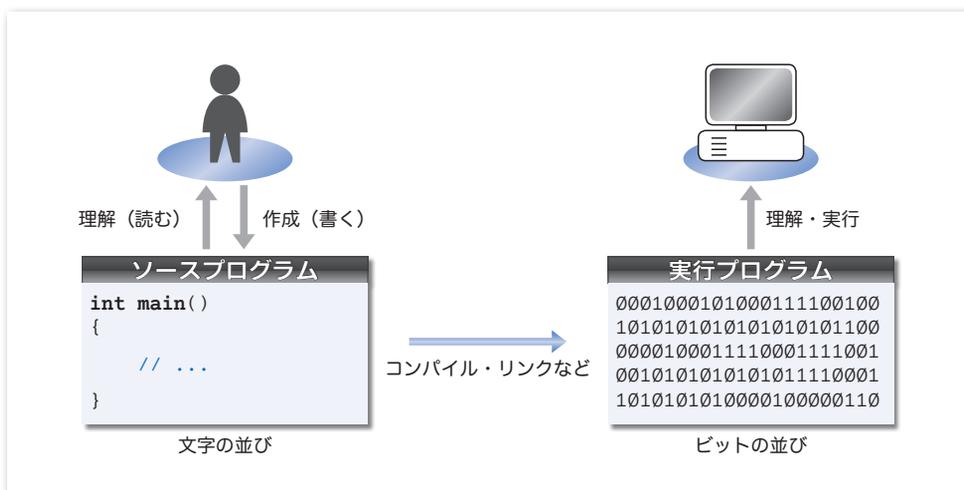
■ プログラムの実行

コンピュータは、C++のソースプログラムを直接理解して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** (*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ね。それが0と1だけに制限されていると考えましょう)。

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きましょう。



● **Fig.1-2** プログラムの作成から実行まで

- ▶ ソースプログラムに^{フタ}綴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）

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**注釈**=**コメント**（comment）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、簡潔な言葉（日本語や英語など）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ので、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。本書では、コメントを青文字で表記します。

■ 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>
```

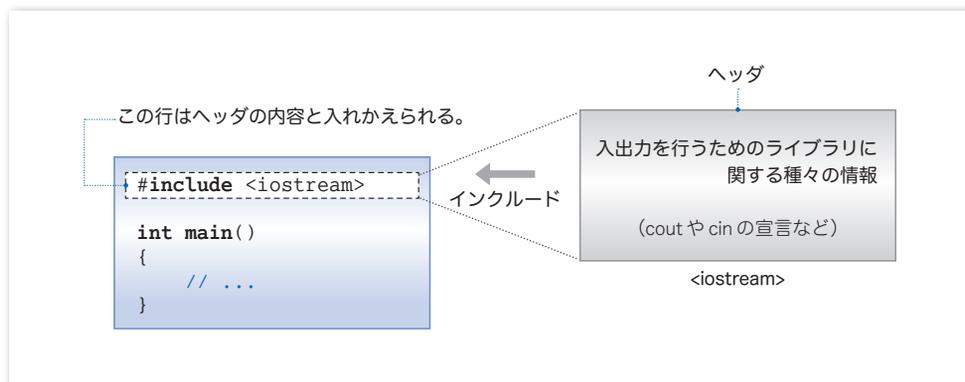
画面やキーボードなどへの入出力を行うためのライブラリ（処理実現のための部品群）に関する情報が格納されている`<iostream>`の内容を取り込みます。

Fig.1-3に示すように、`#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ります。

なお、`<iostream>`の他に、文字列を扱う`<string>`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。

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

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

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であるとは限りません (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、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。

なお、using namespace std; の指令は省略可能です。ただし、省略する場合は、プログラム中の cout を std::cout に変更しなければなりません。

▶ その理由は、第9章で学習します。

■ 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検証せよ。

■ 演習 1-2

List 1-1 (p.4) 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。

■ 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";
```

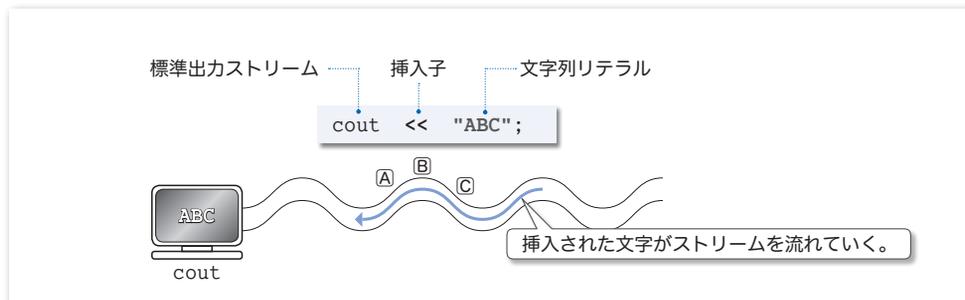
コンソール画面を含め、ファイルなどの外部に対する入出力には、**ストリーム** (stream) を利用します。ストリームとは、文字が流れる“川”のようなものです (**Fig.1-4**)。

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。

- ▶ 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。



● **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) の略です。また、`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、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。本書では、文字列リテラルを“少し薄い黒文字”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。文字列リテラルの詳細は、第8章で学習します。

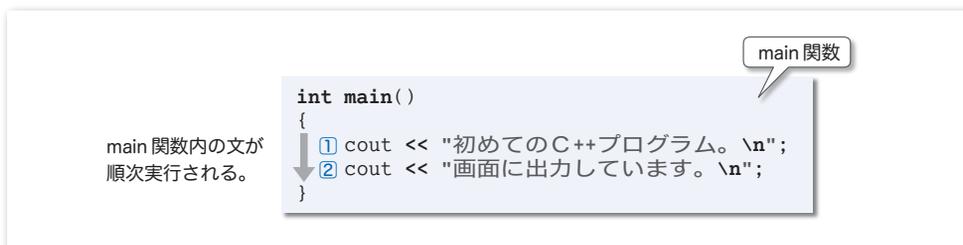
■ 改行

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。拡張表記の詳細は、第3章で学習します。

■ 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。



● **Fig.1-5** プログラムの実行とmain関数

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**実行される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。

- ▶ `int main() { }` 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。

■ 文

本プログラムの **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）。

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。《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。

画面への出力を一つの文にまとめたプログラムを **List 1-2** に示します。

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。

文字列リテラル内に2個の改行文字 `\n` が含まれています。「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ることを確認しましょう。

- ▶ これ以降、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」と表現することにします。

■ ストリームへの連続した出力

List 1-3 は、二つの挨拶「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 1-3

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";
}
```

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“ビーブ音”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。

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。なお、ここに示す読み方は、通称・略称・俗称を含んでいます。

- ▶ 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ははじめまして。" << "こんにちは。¥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。

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
*	アスタリスク、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、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、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、による ※JIS コードでは ` (オーバーライン)
-	オーバーライン、上線、アッパライン
^	アクセント記号、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 1-4 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"
;
}

```

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++ 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単語の途中で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。これらの途中にホワイトスペース**空白類文字** (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) を入れて、右のように記述す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、左下に示すように、途中で改行してはいけません。

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 に示す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 << "ABCDEFGHJKLMNOPQRSTUVWXYZ" // 空白類をはさんで並んだ
           "abcdefghijklmnopqrstuvwxyzn"; // 文字列リテラルは連結される
}
```

実行結果

```
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyzn
```

文字列リテラル "ABCDEFGHJKLMNOPQRSTUVWXYZ" と "abcdefghijklmnopqrstuvwxyzn" が連結されて 1 個の文字列リテラルとなっていることが、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間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、ホワイトスペース空白類 (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。このような指令は前処理指令 (preprocessing directive) と呼ばれます。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。

前処理指令は、原則として、単一行の中に書かなければなりません。途中で改行する必要がある場合は、行末に逆斜線 \ を書くことになっています。

```
#include
<iostream>
```



```
#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。

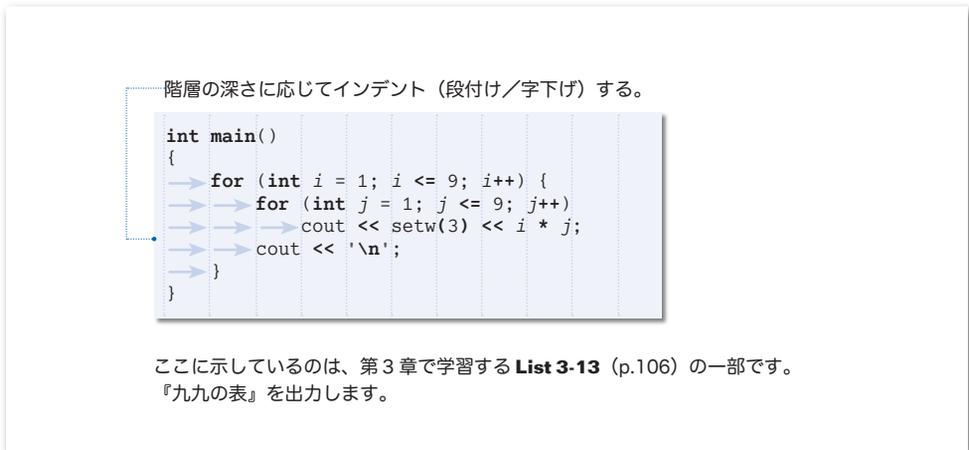
■ インデント

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、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**インデントーション**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**）。

- ▶ 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。



● **Fig.1-6** ソースプログラム中のインデント

- ▶ 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。

■ 演習 1-4

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

柴田
望洋

■ 演習 1-5

右に示すように、1 行に 1 文字ずつ名前を表示するプログラムを作成せよ。姓と名のあいだは 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

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な作業となります。

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。

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。青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。』は表示されなくなります。

List 1C-1

chap01/list01c01.c

```
// 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.c

```
// 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しましょう。

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.415) で学習します。

1-2

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 1-6** 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。

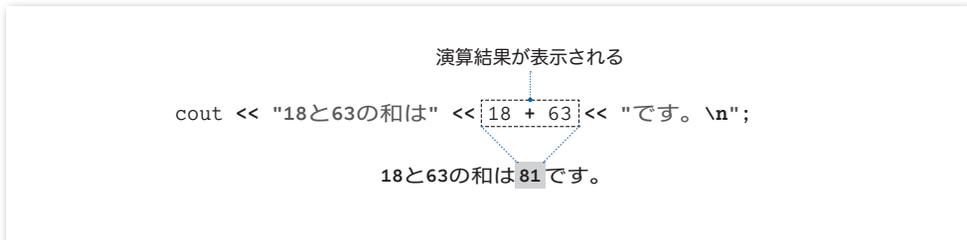
- ▶ 整数リテラル 18 は単一の数値 じゅうはち 18 で、文字列リテラル "18" は 2 個の文字 いち 1 と はち 8 が並んだものです。整数リテラルの詳細は、第 4 章で学習します。

■ 演算結果の出力

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わけではありません。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ることができません。数値を変更する際は、プログラムに手を加えるだけでなく、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
```

^{イント}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。

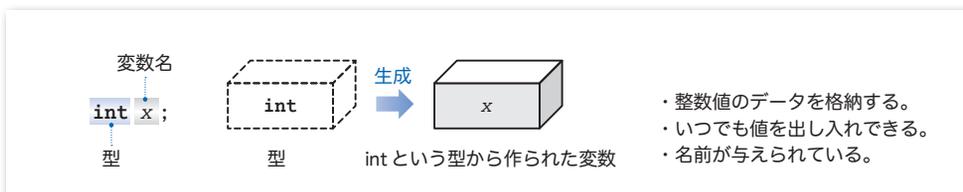
変数xが保持できる値は整数に限られています。たとえば3.5といった小数部をもつ実数値は扱えません。これは、intという型 (type) の性質です。

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。

▶ 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

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

    cout << "xの値は" << x << "です。 \n";
    cout << "yの値は" << y << "です。 \n";
    cout << "合計は" << x + y << "です。 \n";
    cout << "平均は" << (x + y) / 2 << "です。 \n";
}

```

実行結果

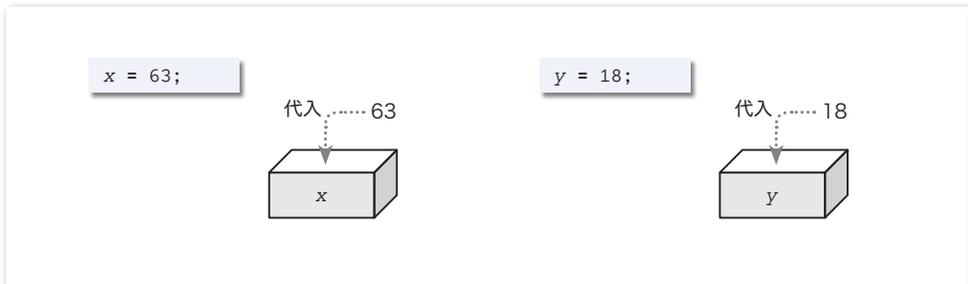
xの値は63です。
yの値は18です。
合計は81です。
平均は40です。

▶ 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。このほうが、個々の宣言に対する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）。

■ 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** 変数への値の代入

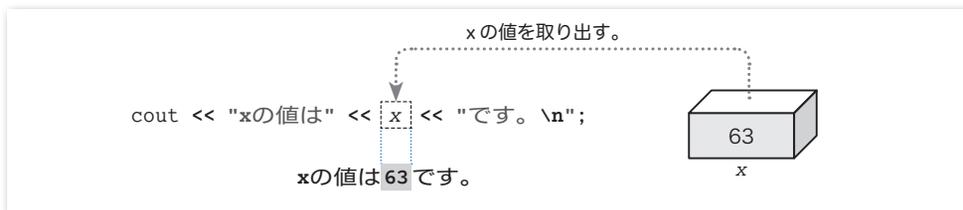
代入演算子は、数学のように『 x と 63 が等しい』とか『 y が 18 と等しい』と解釈されているのではないことに注意しましょう。

▶ 演算子については、p.22 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から、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** ストリームへの変数の値の出力

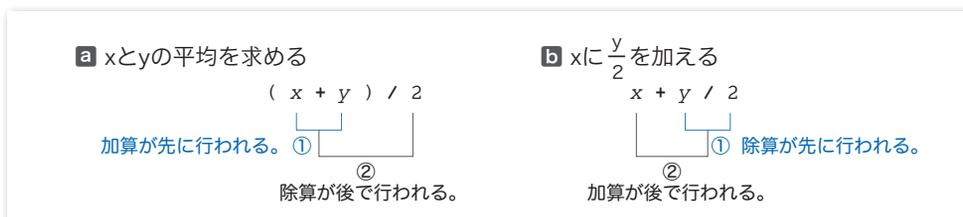
■ 算術演算子と演算のグループ化

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、**Table 2-10** (p.76) 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。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。

■ 演習 1-6

List 1-7 のプログラムを変更して、小数部をもつ実数値を x や y に代入するプログラムを作成せよ。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すること。

■ 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。

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。
```

1-3

変数

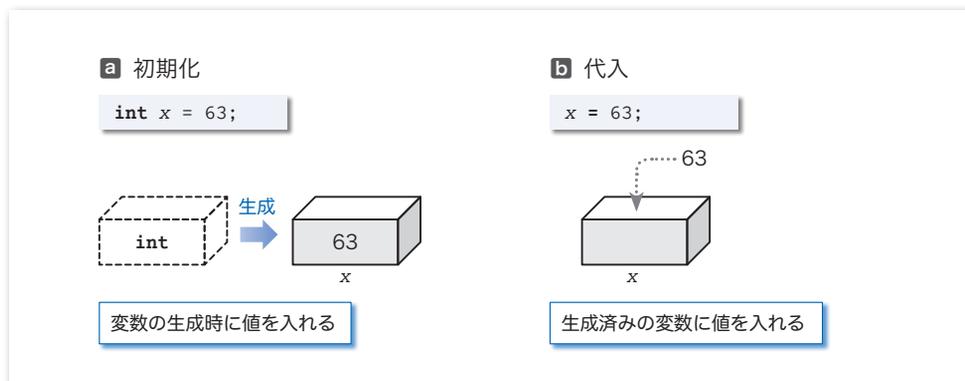
■ 初期化と代入

本プログラムで行っている《初期化》と、**List 1-7** (p.18) 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**：生成済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=を太字で示すことによって区別しやすくしています。



● 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

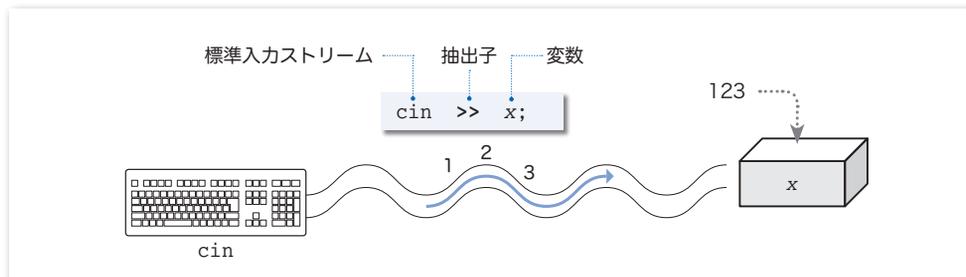
キーボードから二つの整数値を読み込んで、それらに対して加算・減算・乗算・除算を行った結果を表示しましょう。そのプログラムを **List 1-10** に示します。

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 4-1** (p.129) 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。
```

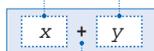
1-4

キーボードからの入力

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。

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。

オペランド オペランド (演算の対象となる式)



※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。

演算子 (演算を行う記号)

● 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)。

連続した読み込み

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みます。

そのことを使っ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 への読み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。

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 のあいだ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
```

- ▶ 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。 **Column 1-2** (p.27) 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 **List 1-12** に示します。

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。

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原則です。

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.23) ではありません。演算子 $-$ 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 (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。

■ 実数値の読み込み

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を p.17 で学習しました。実数は、`double` という型によって扱えます。

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。

本プログラムでは剰余を求めていません。**Table 1-3** (p.23) 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。

重要 実数型のオペランドに `%` 演算子を適用することはできない。

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` 型の変数で表すことにします。

- ▶ 実数の剰余を求める方法は、**List 2-16** (p.63) 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。

■ 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をそのまま反復して表示するプログラムを作成せよ。

整数値：7
7と入力しましたね。

■ 演習 1-10

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 1-11

右に示すように、二つの実数値を読み込み、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。

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 1-12

右に示すように、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。

三角形の面積を求めます。
底辺：7.8
高さ：3.2
面積は12.48です。

Column 1-2

除算の演算結果

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。

■ 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。そのプログラムを **List 1-14** に示します。

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 $2\pi r$ と面積 πr^2 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 を表すのが、**網かけ部**の3.14です。このように、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。

*

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 $2\pi r$ と面積 πr^2 をより正確に求めるために、円周率を3.1416に変えることを考えましょう。そのためには、**網かけ部**を3.1416に変更することになりますね。

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。定値オブジェクトを用いて書きかえたプログラムを **List 1-15** に示します。

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。

▶ “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、と理解しておくといでしょう。

List 1-15

chap01/list0115.cpp

// 円周の長さや円の面積を求める (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。

1-4

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 3.1416 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から、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、何を表すものである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るとよい。

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。右下のプログラムは、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えます。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>
#include <cstdlib>
#include <iostream>
```

```
using namespace std;
```

```
int main()
```

```
{
    srand(time(NULL)); // 乱数の種を設定
    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";
}
```

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**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から、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ますので、**lucky**の値は必ず0以上9以下の整数値となります。

演習 1-13

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。

演習 1-14

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。

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。

Column 1-3

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、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.404 の **Column 11-4** 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かからず、本当の乱数は、次に生成される数値の予測がつきません。

文字の読み込み

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.24)、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。変数 `c` に読み込まれるのは、空白以外の最初の文字となります。

▶ **char** 型の詳細は、第4章で学習します。

文字列の読み込み

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：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と同じです (p.7)。

抽出子 `>>` による読み込みでは、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、“柴田”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：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み込みは、`getline(cin, 変数名)` によって行います。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ることとなります。

*

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。

まとめ

1

- C++ は、C 言語と Simula 67 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プログラミング言語である。
- ソースプログラムは、文字の並びとして作成する。ソースプログラムをそのまま実行することはできないので、コンパイルやリンクを行って、実行できる形式に変換する必要がある。
- C++ のプログラムは自由形式である。スペースやタブによって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注釈コメントを記入するとよい。
- 標準ライブラリの利用にあたっては、該当するヘッダを `#include` 指令によってインクルードしなければならない。`using namespace std;` の指令を行うと使いやすくなる。
- C++ のプログラムを起動すると、`main` 関数内に置かれた文が順次実行される。原則として文の終端はセミコロン ; である。
- 文字の流れる川であるストリームへの入出力に関するヘッダは `<iostream>` である。画面を表す標準出力ストリーム `cout` への出力は、挿入子 `<<` によって行える。キーボードを表す標準入力ストリーム `cin` からの入力、抽出子 `>>` によって行える。
- 値の性質を表すのが型である。整数、実数、文字は、それぞれ `int` 型、`double` 型、`char` 型で表せる。
- 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。
- 整数の定数は整数リテラルとして表し、実数の定数は浮動小数点リテラルとして表す。また、文字の並びは二重引用符で囲んだ文字列リテラル `"..."` として表す。空白類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、警報文字（一般にはビーブ音）は `\a` で表す。
- 数値などのデータを自由に出し入れできる変数は、型から作られた実体である。変数を使うには、必要になった時点で型と名前を与えた宣言文によって宣言する。
- 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。
- `const` 付きで宣言された変数は、値を変更できない定値オブジェクトとなる。定数に対して名前を与えるのに有効である。
- 演算を行うための `+` や `*` などの記号が演算子であり、演算の対象がオペランドである。オペランドの個数に応じて、単項演算子、2項演算子、3項演算子がある。

- 演算子によって優先度が異なる。() で囲まれた演算は優先的に実行される。
- 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。“整数 / 整数” の演算で得られる商は、小数部が切り捨てられた値である。% 演算子のオペランドは整数でなければならない。
- 非負のランダムな値である乱数は、**rand** 関数によって生成できる。

```

注釈 (コメント)
/* 複数行にわたれる */
// その行の行末まで

```

加減演算子	$x + y$	$x - y$
乗除演算子	$x * y$	x / y $x \% y$
単項の算術演算子	$+x$	$-x$

#include 指令：インクルードするヘッダの内容と置きかわる。

```

chapg01/summary.cpp
/*
 サンプル・プログラム
*/

#include <ctime>
#include <cstdlib>
#include <string>
#include <iostream>

using namespace std;

int main()
{
    int a;
    a = 1;
    int b = 5;

    srand(time(NULL));
    int lucky = rand() % 10;
    cout << "今日のラッキーナンバーは" << lucky << "です。\\n";
    cout << "2 で割った商は" << lucky / 2 << "です。\\n";
    cout << "2 で割った剰余は" << lucky % 2 << "です。\\n";

    // 定値オブジェクト (値を書きかえられない変数)
    const double PI = 3.14;
    double r;
    cout << "半径：" << r;
    cin >> r;
    cout << "半径" << r << "の円の面積は"
         << (PI * r * r) << "です。\\n";

    string name; // 名前
    cout << "お名前は：" << name;
    cin >> name;
    cout << "aこん" "にちは" << name << "さん。\\n";
}

```

各文が順次実行される

乱数の生成に必要
文字列の利用に必要
ストリーム入出力に必要

using 指令

整数リテラル	5
浮動小数点リテラル	3.14
文字列リテラル	"半径："

型
変数名
初期化子
// aはint型の変数
// 代入 (生成済みの変数に値を入れる)
// 初期化 (変数の生成時に値を入れる)
乱数生成の準備
0以上RAND_MAX以下の乱数
// 乱数の種を設定
// 0~9の乱数
オペランド
演算子
オペランド

実行例

```

今日のラッキーナンバーは5です。
2で割った商は2です。
2で割った剰余は1です。
半径：4.5
半径4.5の円の面積は63.585です。
お名前は：Shibata BohYoh
♪ こんにちはShibataさん。

```

警告

改行

空白類をはさむ文字列リテラルは連結される。