第 2 章

型・値・式・変数

駐車場の自動精算機は、どのように車を管理しているでしょうか。駐車開始時刻から経過時間を知って、駐車料金を算出しているはずです。計算機のプログラムとして模倣するためには、これらをプログラミングによって実現せねばなりません。

デジタルの計算機は、数値、もっと言えば 0 か 1 しかわかりません。しかし、0 と 1 の列をどのように解釈するかによって、多種のデータを表現することができます。ここでは、その始まりとして、数値や文字がどのように表現されているか見ることにします。また、可変な値としての変数や、データが何を表すのかを意味する**型**についても見ていきます。

キーワード

- 型，定数，変数
- 整数，浮動小数点数，文字
- int，double，char
- スワップ
- キャスト
- 処理系依存

表 2.1　よく使われる型

型名	表すもの	定数の例
int	整数	256　0x100
double	浮動小数点数	365.24　3.6524e+2
char	文字（1 文字）	'A'　'#'　'0'
char*	文字列	"Aa0#"　"256"　"A"
void	値なし	—

2.1　定数と変数と型

プログラムで扱う値は、次の 2 通りに大別されます。

定数[*1]　ソースコード上に、直接記述した値です。その名の通り、変化しない値です。
変数　値を保存する入れ物です。代入したときに、保存されている値が変化します。

このような値には、**型** (type) と呼ばれる種類があります。型ごとに表せるものが決まっていて、整数、小数、あるいは文字などがあります。表 2.1 のものがよく使われるので、最初に覚えましょう。既に**??**節では、int と double を使いました。

ソースコード上の定数は、自動的に型が決まります。例えば、「10」は int 型ですし、「1.23」は double 型です。変数の型は、int などの**型名**で指定します。

2.2　変数

変数は、値を保存する**箱**のようなものです。数学の変数と似ていますが、値の変化するタイミングや、名前のつけ方も違います。詳しく見ていきましょう。

2.2.1　変数定義

変数は、**定義** (definition)[*2]してから使います。定義には、型と変数名を明記します。型が同じなら、複数の変数をまとめて定義できて、変数名をコンマで区切って書き並べます。

```
/* 文法 */
型名 変数名;
型名 変数名1, 変数名2, ...;
```

```
/* 実例 */
int a;
double x, y; // 同じ型をまとめて
```

[*1] 読みは「ていすう」と「じょうすう」の両方があります。
[*2] 「定義」の代わりに、メモリ配置を伴わない「**宣言** (declaration)」の用語を使う書籍もあります。分割コンパイルをすると区別が必要ですが、（本書の守備範囲の）単一ソースファイルでは、実質的に同じです。

鋭意作成中

2.2.2 変数への代入

変数に値を代入するには、**代入演算子** (assignment operator) = を使います。変数には何度でも代入できます。最後に代入した値一つだけが保持されて、古い値は、新しい値を代入した瞬間に忘れてしまいます。

```
/* 文法 */

変数名 = 式;
```

```
/* 実例 */
int a, b;
a = 1;
b = a + 2;     // b = 1 + 2
// a + 2 = b; // コンパイルエラー
```

$a = b$ と $b = a$ は、数学では同じ意味のこともありますが、プログラムでは違います。= の右辺と左辺で役割が異なっていて、右辺の**式の値を計算**して、左辺の**変数に格納**します。格納先である左辺は、普通の計算式ではなく**単一の変数**である必要があって、**左辺値** (lvalue) という造語で呼ぶほど重要な概念です。右上の例でわかるように、「a」や「b」は左辺値ですが、「a+2」は左辺値ではなく、代入できません。

= の**両辺の型は一致**させるのを基本としてください。C 言語は、一致していないと（おせっかいなことに）自動変換してしまうので、かえって思わぬ動作不良を引き起こします。もちろん、変換できなければコンパイルエラーになります。

右辺の「a+2」のように、計算式に変数が現れると、保存しておいた値を取り出します。この操作を「変数を**参照** (refer) する」とも「変数に**アクセス** (access) する」ともいいます。

よくある誤解に、= で関係式（恒等式）を定義していると思う人がいますが、そうではありません。C 言語では、その瞬間の式の値を格納しているに過ぎないので、代入した変数は、次に代入するまで値が変化しないことに注意してください。

コラム： 変数定義の場所

古い C 言語規格（C89 まで）では、変数定義の行える場所に強い制約があり、ブロック（☞**??**項）の開始直後に限られていました。使用変数の一覧になっていたものの、定義と使用する場所が離れて、初期化忘れを誘発する欠点もありました。

C99 からこの制限は撤廃され、ブロックの途中でも変数定義が行えます。これで C++ や Java など、多くの言語と同じになりました。同じ変数を使いまわさずにすんで、初期化忘れの検出にも役立ちます。

残念なことに、C 言語の入門書は古いスタイルのままのものが多く、本書は数少ない C99 準拠の入門書となっています。**TODO:** 初期化を先に述べている

2.2.3　変数の初期化

変数の値は、多くの場面で、代入するまでどんな値になっているのか保証がありません[*3]。代入されていない変数の状態を**未初期化** (uninitialized) とか**代入忘れ**、入っている値は**不定** (indeterminate) だとも、俗に**ゴミ**ともいいます。代入したつもりで、忘れたままゴミの値を使っていると、動作がおかしくなります。しかも、ゴミのはずの値が、特定の条件で 0 になることもあって、代入忘れに気づきにくいので要注意です。

代入忘れを防ぐために、様々な工夫がされています[*4]。ここで紹介する、変数の**定義と同時に代入**する**初期化** (initialize) の構文も、代入忘れ防止に役立ちます。

```
int a;  // 変数定義
a = 5;  // 通常の代入
```

```
/* 初期化つき */
int a = 5;
```

型が同じ変数は、やはり次のように、まとめることができます。

```
/* 変数１つずつ */
int a = 5;
int b = 2;
int c = a + b;
```

```
/* 同じ型の変数をまとめて */
int a = 5, b = 2, c = a + b;
```

コラム：初期化

初期化という言葉には、2 通りの意味があります。

狭義　変数定義と同時に行う代入

広義　変数を定義して、初めて行う代入（同時でなくてもよい）

狭義の初期化だけの機能もあります。警告の「変数が未初期化」は広義です。

[*3] C 言語は様々な操作を許すために、代入忘れをエラーにしません。他の言語に目を向けると、エラーにしたり、変数を定義したと同時に 0 で初期化されると決めている言語も多くあります。

[*4] スコープを狭くしたり（☞??ページのコラム）、コンパイラの警告（☞??節）を活用します。

鋭意作成中

2.2.4 識別子

識別子 (identifier) は、変数名や関数名に使う文字の並びです。**シンボル** (symbol) ともいいます。数学の変数名は 1 文字ですが、プログラムでは長い名前がつけられます。

- アルファベットの大文字と小文字、数字、アンダースコア（_）を組み合わせます。
- アルファベットの大文字と小文字は、区別されます。
- 長さは、多くの場面で少なくとも先頭から 31 文字は区別される[*5]ので、ほぼ気にしなくて大丈夫です。
- プログラム上の特別な役割の**予約語** (reserved word)（☞**??**節）は使えません。
- 先頭文字はアルファベットにしておきます。
  - 数字で始めると、数値と解釈されます。
  - アンダースコア（_）で始まる名前は、システムに予約されているので、プログラムで作成してはいけません。

命名の習慣は☞**??**節で紹介します。ひとまず**変数には小文字**を使っておきましょう。

```
/* 正しい識別子 */
ab
automation
power_2
seven_zip
unix__
```

```
/* 誤った識別子 */
a b （×スペース）
auto （予約語に一致）
power-2 （×ハイフン）
7zip （×先頭の数字）
__unix__ （×先頭の_）
```

コラム： 静的型付け vs. 動的型付け

C 言語では、コンパイル時に型を厳格に決定してしまいます（静的型付け）。型の一致しない代入や演算は、自動変換されたり、コンパイルエラーになったりします。型を首尾一貫することで、正しいプログラムを作る助けにする狙いがあります。言語によっては、実行時まで型を決定しないものもあります（動的型付け）。

[*5] 長すぎてもエラーにはなりませんが、先頭部分が一致すると、違う変数名でも同じと扱われることになります。実際に何文字まで区別されるかは、文脈や環境によっても異なります。

2.3 整数

整数を扱うための**整数型** (integer type) を説明します。これまで使ってきたように、1 や 20 など、**数字だけの並びは整数型の定数**です。通常はもちろん **10 進数**です。

先頭が 0 なら **8 進数**で解釈します。
先頭が 0x なら **16 進数**になり、数字に加えて a~f の英字も使います。大文字と小文字の区別はありません。

```
int a = 123;   // 123  (10進数)
int b = 0123;  // 83    (8進数)
int c = 0x123; // 291  (16進数)
int d = 0xFF;  // 255  (16進数)
```

整数型には、符号の有無などでいくつも種類がありますが、ひとまず int だけ覚えておけば大丈夫です。順番に見ていきましょう。

2.3.1 符号付き整数型

一番よく使う**符号付き整数型** (signed integer type) が int です。環境に応じて、CPU にとって一番扱いやすいものが選ばれています。そして割り当てられる領域サイズの違いで 4 種類の型があります[*6]。（本書では使い分けの必要な場面はほとんどありません。）

「short int」 ≤ 「int」 ≤ 「long int」 ≤ 「long long int」

不等号はメモリ領域サイズの大小を表していて、4 種類の中で同じものもあり得ます。また、int に long のような修飾する単語が付くと int は省略できるので、以降は省略します。

2.3.2 符号なし整数型

負の整数が不要な場面があります。ビット演算や、正の大きな値を扱うときなどです。このために**符号なし整数型** (unsigned integer type) が用意されています。

本書では積極的には用いませんが、標準ライブラリで利用されている（☞**??**節）ので、存在だけは知っておきましょう。型の名前としては、符号付き整数の型の前に unsigned のキーワードを付けて「unsigned int[*7]」「unsigned long」のように[*8]します。

> ――― 複数のバイト列で整数を表す場合の内部表現（バイトの並び順・エンディアン）[*9]や、マイナスの値の実現方法[*10]は何通りかあって、CPU によって異なります。C 言語規格では選択の幅を持たせています。 ―――

[*6] long long は C99 で正式に導入されました。
[*7] ここでも int は省略できて「unsigned」だけでも同じ意味ですが、本書では省略しないことにします。
[*8] unsigned との対称で「signed」のキーワードもあるので、「signed long」という型名も表記可能ですが、long は元から符号付きなので、特に効果はありません。2.5.1 項の signed char のみ意味があります。
[*9] 上位ビットから並べるビッグエンディアン，下位ビットから並べるリトルエンディアンなどがあります。
[*10] C99 では、次の 3 通りが想定されています。 (i) 2 の補数 (ii) 1 の補数 (iii) 符号ビット

表 2.2 <limits.h> で定義された整数型の主な定数 (1)

シンボル	意味	保証値	保証値の絶対値
SHRT_MIN	short の最小値	−32767	$2^{15}-1$
SHRT_MAX	short の最大値	+32767	$2^{15}-1$
USHRT_MAX	unsigned short の最大値	65535	$2^{16}-1$
INT_MIN	int の最小値	−32767	$2^{15}-1$
INT_MAX	int の最大値	+32767	$2^{15}-1$
UINT_MAX	unsigned int の最大値	65535	$2^{16}-1$
LONG_MIN	long の最小値	−2147483647	$2^{31}-1$
LONG_MAX	long の最大値	+2147483647	$2^{31}-1$
ULONG_MAX	unsigned long の最大値	4294967295	$2^{32}-1$
LLONG_MIN	long long の最小値	−9223372036854775807	$2^{63}-1$
LLONG_MAX	long long の最大値	+9223372036854775807	$2^{63}-1$
ULLONG_MAX	unsigned long long の最大値	18446744073709551615	$2^{64}-1$

2.3.3 整数型の表示方法

int を printf() で表示する際の書式文字列は %d です。符号なしの表記にするなら、10 進の %u、8 進の %o、16 進の %x があります。%X なら大文字です。(☞??節)

```
printf("%d", 123); // "123" (10進)
printf("%u", 123); // "123" (10進)
printf("%o", 123); // "173" ( 8進)
printf("%x", 123); // "7b"  (16進)
printf("%X", 0xff);// "FF"  (16進)

printf("%5d", 123);  // "  123"
printf("%05d", 123); // "00123"
```

% と d の間に 5 を指定すると、必ず 5 文字以上で表示して、足りなければ先頭にスペース文字が追加されます。05 なら、スペース文字の代わりに 0 で埋められます。

2.3.4 整数型の値の範囲

整数型ごとの記憶できる値の範囲は、表 2.2 のように、<limits.h> に定義されています。C 言語の規格では、最低限の範囲が規定されています。つまり、もっと広い範囲の値を格納する処理系（コンパイラ）があっても構いません。（このように処理系の自由に決めてよいものを**処理系依存** (implementation dependent) といいます[*11]。）最低限の範囲を「保証値」として載せておきました。C 言語は、パソコンだけでなく、小さな組み込みプロセッサなどでの利用も想定して、このような選択に幅を持たせています。

[*11] 処理系依存の性質をあてにしたプログラムは、違う環境では動作がおかしくなる可能性があります。このような、動作保証のない状態を「**可搬性** (portability) を失っている」といいます。

2.4　浮動小数点数

整数ばかりを扱っていても、平均を求めたくなれば、小数付きの計算が必要です。6.02×10^{23} のような（誤差を含めた）大きな数を扱いたいこともあるでしょう。

```
double a = 1.23;     // 1.23
double b = 1.23e+2; // 123.0
double c = 123E-2;  // 1.23
double d = .23;      // 0.23
```

このための型が**浮動小数点型** (floating point type)[*12]です。内部的には、仮数(かすう)部と指数部に分けて格納されます。この値は**浮動小数点数** (floating point number) と呼ばれます。

- `1.23` のように、**小数点が入る**と浮動小数点型の定数になります。
- `6.02e+23` のように、**e が入る**と**指数表記** (exponential notation) の定数です。`e` の前が仮数、後が指数と解釈され、6.02×10^{23} の意味です。仮数は 1〜10 に正規化されている必要もなく、小数点もなくても構いません。`e` は `E` でも同じです。
- 多用する機能ではありませんが、整数部分か小数部分が 0 であれば（片方だけ）省略可能で、0.5 は「`.5`」、1.00 は「`1.`」と表記できます。

2.4.1　浮動小数点型

具体的な型としては、領域サイズの違いで、次の 3 通りあります。すべて符号付きです。

`float` 浮動小数点 (floating point) から容易に想像できる名称ですが、整数型の short のような役割で、あまり積極的には用いられません。（☞**??**ページのコラム）

`double` 小数付きの数値といえば、通常はこの double を用います。float に 32 ビット、double に 64 ビットと「倍」(double) の領域を割り当てることが多いので、このような名称になっているのでしょうが、言語規格上は倍にする必要はありません。

`long double` さらに高精度の計算ができそうな名称ですが、実際には環境によって精度や計算速度が大きく異なる[*13]ので、いつでも役立つわけではありません。

2.4.2　浮動小数点型の表示方法

double を printf() で表示する際の書式文字列は `%f` です。float も同じです。（☞**??**節）

- `%f` は、小数での表記になります。
- `%e` は、`6.02e+23` のような指数表記になります。
- `%g` は、指数の値に応じて `%f` と `%e` のどちらかが自動的に選ばれます。そして小数

[*12] 実数型 (real data type) と呼ぶ人もいますが、数学的には正確ではありません。実際には「有限小数」です。FORTRAN 言語での型名は REAL なので、影響されているのかもしれません。

[*13] long double の実体の例を挙げます。 (1) double と同じ。 (2) 128 ビットの領域を割り当てて、うち 80 ビットを演算に用いる。 (3) 128 ビットの領域の全体で演算をして、計算時間が double の何倍もかかる。

表 2.3 <float.h> で定義された浮動小数点型の主な定数

シンボル	意味	保証値	
FLT_RADIX	内部表現に使用される基数 (2, 16, ...)		
FLT_MAX	float の最大値	1e+37	(10^{37})
DBL_MAX	double の最大値	1e+37	(10^{37})
LDBL_MAX	long double の最大値	1e+37	(10^{37})
FLT_DIG	float の有効精度（10 進数表記の桁数）	6	
DBL_DIG	double の有効精度（〃）	10	
LDBL_DIG	long double の有効精度（〃）	10	

※最小値は、最大値の符号違いです。DBL_MIN の意味は☞**??**項。

の末尾の 0 が省かれます。

- %8f は、8 文字以上で表示されます。8 文字より少なければ、先頭にスペース文字が補われます。8 文字より多ければ、必要な文字数まで使われます。
- %.3f は、小数部分が 3 桁になります。%8.3f のように、全体の文字数と同時に指定できます。省略時は 6 と同じです。%.0f のように 0 を指定すると整数で表示され、小数点も表示されません。

```
printf("%f", 1.2);//"1.200000"
printf("%e", 1.2);//"1.200000e+00"
printf("%g", 1.2);//"1.2"
```

```
printf("%.2f", 1.2e3);//"1200.00"
printf("%.2e", 1.2e3);//"1.20e+03"
printf("%.2g", 1.2e3);//"1.2e+03"
```

2.4.3 浮動小数点型の値の範囲と精度

型ごとの性質をあらわす定数は、表 2.3 のように <float.h> に定義されています。浮動小数点型の性質は複雑で、多くの定数があるのですが（☞**??**節）、さしあたり最大値・最小値と精度をおさえておきましょう。

表の中に、（マイナスの）最小値を表す定数はありません。整数型とは異なり、浮動小数点型は**必ず符号付き**ですから、表現できる値の範囲は**正負対称**です。この性質から、double の最小値は、最大値を利用して -DBL_MAX と表せます。

IEEE 754 という規格書で規定されたフォーマットが C 言語規格で推奨されていて、単精度（32 ビット）を float に、倍精度（64 ビット）を double に割り当てることがほとんどです。精度は float で 7 桁、**double で 15〜16 桁**、と覚えておくと役立ちます。

2.5 文字

プログラムでは、数値だけでなく、文字を扱いたい場面もあるでしょう。まずは 1 文字を表す**文字定数**です。文字定数は、シングルコーテーション (') で**1 文字だけ**を囲います。0 文字でも 2 文字以上でも、マルチバイト文字[*14]でもコンパイルエラーです。

```
char c = 'A'; // 文字定数
// char d = '12';//NG:複数文字
// char e = 'あ';//NG:マルチバイト
```

コンピュータでは一般に、文字は**文字コード** (character code) に変換してから扱います。C 言語でも、文字を扱っているかのようにみせかける文法がありますが、最終的には「整数値」に変換しています。つまりこの文字定数は、**内部的には数値**（文字コード）に置き換わります。そしてプログラムにとって、数値か文字かの違いは、**表示の際の形式**が 10 進数なのか、それとも対応する文字コードの文字なのか、だけです。

2.5.1 文字型

文字は char 型で保存します[*15]。char は int などと同じ、整数型の一員ですが、符号付きかどうかは処理系依存です。1 バイトの領域を割り当てると決まっています。

> ほかに、符号付きの signed char と、符号なしの unsigned char という型もあります。どちらも 8ビット以上の領域が割り当てられると決まっています。どちらかというと文字のためではなく、signed char は short よりもメモリを節約するためであったり、unsigned char はバイト単位のファイルや I/O 入出力の際の最小単位として使われることが多く、本書では出番がありません。ちなみに、どちらか片方が char と一致するとも決まっています。一致するほうは 1バイトになるわけです。

よく使われる文字コード体系は、**ASCII コード** （☞??節）ですが、C 言語は文字コード体系に依存しないように設計されています。

2.5.2 文字型の表示方法

char を printf() で表示する際の書式文字列は %c です。%d などで**文字コード**が表示されることも理解しておきましょう。右は ASCII コードでの例です。

```
printf("%c", 'A');// "A"  (文字)
printf("%d", 'A');// "65" (10進数)
printf("%o", 'A');// "101" (8進数)
printf("%x", 'A');// "41" (16進数)
```

[*14] マルチバイト文字にはワイド文字 (wchar_t) という型が用意されていますが、本書では立ち入りません。
[*15] 1 文字単位で扱う場合は、大は小を兼ねる、のことわざのように、int で代用する場面も多々あります。

表 2.4 <limits.h> で定義された整数型の主な定数 (2)

シンボル	意味	保証値	保証値の絶対値
SCHAR_MIN	signed char の最小値	-127	$2^7 - 1$
SCHAR_MAX	signed char の最大値	+127	$2^7 - 1$
UCHAR_MAX	unsigned char の最大値	255	$2^8 - 1$
CHAR_BIT	char のビット数	8	
CHAR_MIN	char の最小値	SCHAR_MIN または 0	
CHAR_MAX	char の最大値	SCHAR_MAX または UCHAR_MAX	

「文字定数」と「文字コード」の関係は理解しにくいので、右に例を続けます。'A' は単なる値ですから、文字コードを調べて、その値を書いても動作は同じです。

```
printf("%c", 'A'); // "A" 文字定数
printf("%c", 65);  // "A" 10進数
printf("%c", 0101);// "A"  8進数
printf("%c", 0x41);// "A" 16進数
```

ただし、「65 が A の文字を表す」とわかる人は少ないので、これでは人に解読しにくいプログラムになってしまいます。さらに、文字コード体系は環境ごとに違う可能性があるので、環境に依存したプログラムにもなります。ですから、このことは動作原理として理解しておいて、**プログラムには'A' の文字定数**を使いましょう。

文字の操作（例えば大文字変換）は、文字コード体系によって演算が違います。このような環境依存の動作は、システム（コンパイラ）が用意します。(☞??節)

```
#include <ctype.h>
 ...  省略 ...
printf("%c", toupper('a')); // "A" 大文字変換
printf("%c", tolower('X')); // "x" 小文字変換
```

2.5.3 文字型の値の範囲

文字型の記憶できる値の範囲は、表 2.4 のように、 <limits.h> に定義されています。

コラム： 文字定数と文字型

定数にも型があります。1 や 2 などの数値は皆さんの予想通り int 型です。では 'a' のように書かれた文字定数の型は何だと思いますか？

実は、**文字定数の型は int** であって、char ではありません。つまり普通の数値と同じです。そうは言っても、この文字定数は char が表現できる範囲に収まっていることが保証されているので、char に代入しても問題は起こりません。

2.5.4　文字列と文字列定数

printf() で出力するメッセージは、ダブルコーテーション（"）で囲みます。**??**項で説明した通り、これは文字列の定数（文字列リテラル）です。

文字列 (string) は、文字の連なったもので、長さは何文字でもよくて、1 文字、0 文字の文字列もあります。既に活用していますが、マルチバイト文字も含めて大丈夫です。

```
/* 文字 */
// char c = 'This'; // NG:複数文字
char c = 'A';
// char c = '';     // NG:空文字
```

```
/* 文字列 */
char *s = "This is 文字列.";
char *t = "A"; // 1文字の文字列
char *u = "";  // 空文字列
```

文字列の型は、char とポインタを組み合わせた char* ですが、この型の変数を使いこなすには、**??**章の配列とメモリ確保の仕組みを理解せねばなりません。ここでは、文字列リテラルの復習にとどめておきましょう。

- \ など、特殊な文字を含めるにはエスケープシーケンス（☞**??**節）を使います。
- 改行文字は \n と書きます。ソースコード上で本当に改行してはいけません。
- ソースコード上で連続する 2 つの文字列リテラルは、連結されます。

文字列を printf() で表示するときの書式文字列は %s です。今は定数を表示しても役立った気がしませんが、将来はプログラムで生成した変数を表示するのに使います。

```
printf("%s is a string.", "文字列"); // "文字列 is a string."
```

なお、printf() で出力するメッセージは文字列リテラルにしてきましたが、今後とも定数にしてください。これを変数にすると、セキュリティ上の危険を考慮せねばなりません。

コラム：定数の型と接尾辞

定数の末尾にアルファベットの**接尾辞**を書き加えると、定数の型を制御できます。小文字でも同じ意味ですが、l（エル）は 1（いち）と見間違えるので、大文字がよいでしょう。

整数（10 進数）は、基本的には int で、大きな数は値に応じて long や long long に自動的に切り替わります。明示的に指定するときに、L あるいは LL を使います。unsigned にするなら U です。同時に指定すると 123ULL のようになります。なお、short や char の定数は存在しないので、どうしても必要ならキャスト（☞2.7.3 項）します。

```
long long a = 65536LL;
long double x = 360.0L;
```

浮動小数点の場合は、基本的に double で、自動的には切り替わりません。接尾辞で精度を選ぶことになって、F で float、L で long double になります。

表 2.5 四則演算の演算子

演算子	演算の種類
+	加算（和）
-	減算（差）
*	乗算（積）
/	除算（商）
%	剰余

表 2.6 単項演算子（抜粋）

演算子	演算の種類
+	単項プラス
-	単項マイナス
++	インクリメント
--	デクリメント

2.6 式と単純な演算子

「1+2+3」のような式は、演算が行なわれて、最終的には 1 つの値に置き換わります。式中の「+」のような、演算の種類を表すものを**演算子** (operator)、「1」や「2」のような演算対象となる値を**被演算子**あるいは**オペランド** (operand) といいます。

四則演算 (four arithmetic operations) の演算子は、表 2.5 のように 5 種類あります。

- **2 項演算子**なので、「2+3」のように、演算子の前後に 2 つのオペランドをとります。
- 数学では $2x$ のように掛け算記号を省略できますが、プログラムでは省略できないので「2*x」と書きます。
- 2 つのオペランドの型が**整数なら**、演算結果も**整数**です。「5/2」は 2 になります。double 同士の演算は、もちろん double になりますから、「5.0/2.0」は 2.5 です。
- 2 つのオペランドの型が揃っていないと、（int→double のように、情報量の多い型のほうに）自動的に格上げして揃えられます。

最後の % は、余りを求める**剰余演算子** (modulus operator)[*16]です。割り算の、商と剰余が別々の演算子になっているので、5 種類の演算子があります。整数で演算を行なうと、商も剰余も整数で得られます。商は整数に切り捨てられているようにも見えますが、実際のところは (被除数)=(除数)×(商)+(剰余) の関係を満たすようにしているのでしょう。マイナスの値が入ると状況が複雑になります（☞14 ページのコラム）。

整数演算の 0 の割り算は、規格上の動作は未定義ですが、**実行時エラー** (run-time error) で強制終了されることが多く、Cygwin では以下のようなメッセージが表示されます。

```
Floating point exception (コアダンプ)
```

符号付きの整数演算で**桁あふれ**（表現できる範囲を越えたオーバーフローやアンダーフロー）が起こった場合も、規格上の動作は未定義です。現実には演算結果がおかしくなるだけがほとんどですが、プログラムとしては、どれも起こらないように対策が必要です。

[*16] C 言語では、% は整数専用です。浮動小数点の場合は、数学関数の fmod(x,y) を使います。Java など、型の区別なく % で扱える言語も多数あります。

2.6.1　単項演算子

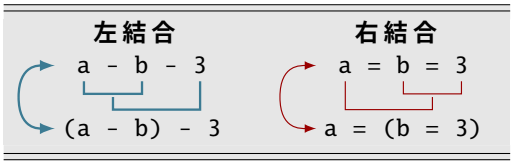
例えば式の先頭に「−3」が出てきた場合、このマイナスは 2 項演算子ではなく、オペランドが 1 つの**単項演算子** (unary operator) として働き、「3」の符号を反転します。表 2.6 のように、マイナスとの対称で、「+3」のような、プラスの単項演算子もあります。

2.6.2　優先順位

数学の式では、カッコ（）で**優先順位** (priority) を上げる（先に計算することを指示）しますが、プログラムでも同じです。カッコの種類は、プログラムでは一通りだけです。

和や差よりも**積や商を優先**するのも、数学と同じです。つまり「`1+2*3`」は「`1+(2*3)`」と同じです。さらに**??**節を参照してもらうと、単項演算子のほうが優先順位が高いので、「`(-2)*(-3)`」は「`-2 * -3`」とカッコがなくても同じだとわかりますが、細かい規則を覚える必要はありません。

同じ優先順位の演算が並んだら、多くの演算子は左のものが優先です（左結合）。代入演算子の = は、（数少ない）右結合なので、右から評価して、しかも = 演算の値は、代入した値そのものですから、同じ値を連続して代入するのに使えます。



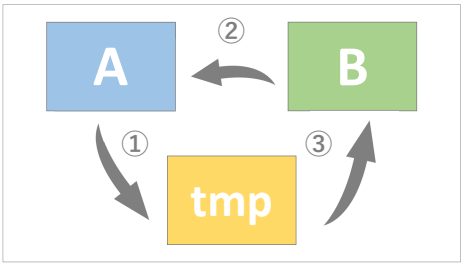
コラム：剰余はプラスに揃えてから

剰余は、日常生活ではそれほど意識して使うことはありませんが、コンピュータ上では、周期的な動作をさせるなどの場面で多用されます。

この計算にマイナスの値が含まれると、商とともに、剰余はややこしくなります。(被除数)=(除数)×(商)+(剰余) の関係は常に満たされるのですが、a % b の符号は (X) a と同符号　(Y) b と同符号　(Z) 常に 0 以上　の 3 通りの動作が考えられます。どれにするかは C89 まで処理系依存でしたが、C99 で (X) に定められました。(X) は Java や、近年の C++ と同じで、多数派ともいえます。しかし、Perl や Python では (Y) を採用していますし、数学の有力な定義では (Z) になっていたりと、まちまちです。

		(X)		(Y)		(Z)	
a	b	/	%	/	%	/	%
+10	+7	+1	+3	+1	+3	+1	+3
+10	−7	−1	+3	−2	−4	−1	+3
−10	+7	−1	−3	−2	+4	−2	+4
−10	−7	+1	−3	+1	−3	+2	+4

細かな言語規格をあてにせず、a も b もプラスの値に限定するのが安心です。



2.7 変数と複雑な演算子

ここでは、プログラミング特有の変数の使い方や、数学にはない演算子を見ていきます。

2.7.1 値の入れ替え（スワップ）

2 つの変数の値を入れ替える操作は**スワップ** (swap) といい、慣用句になっています。変数は、代入された瞬間に、それまで記憶していた値を忘れます。このため 2 つの変数同士で代入していたのでは、代入された方の変数の値は失われるので、工夫が必要です。

記憶してきた値を失わないためには、3 つめの**待避用の変数**が必要になります。つまり a と b をスワップするには、a=b の前に、(1) 3 つめの変数 tmp に a の値を待避します。(2) a=b が終わってから、(3) 待避しておいた tmp（元は a の値）を b に代入します。待避用の変数を思いつくところがポイントです。

tmp（あるいは temp）という名前は temporary（一時の、臨時の）の省略形です。使い捨てにする変数名によく使われます。

```
int a = 3, b = 5;
int tmp = a;  a = b;  b = tmp;   // a = 5,  b = 3
```

2.7.2 インクリメント・デクリメント・複合代入

プログラムでは「`x = x + 1;`」という表現をよく見かけます。数学の方程式だと思うと、x を移項して「0=1」の “解なし” になる、おかしな式ですが、プログラムでの動作は、右辺を計算して、左辺に代入するので、結果として **x の値が 1 増えます**。同様に「`x = x - 1;`」なら x が 1 減ります。

```
int x = 100;

x = x + 1; // x に 1 加える
printf("%d\n", x); // "101"

x = x - 1; // x から 1 減らす
printf("%d\n", x); // "100"

x++; // x に 1 加える
printf("%d\n", x); // "101"
```

1 加えるという操作はよく使うので、**インクリメント** (increment) との呼び名があって、「`x++;`」の専用の単項演算子（表 2.6）があります。**1 減らす**のは**デクリメント** (decrement) で、「`x--;`」です[*17]。

「`x = x + 5;`」の**5 加える**操作には、専用の呼び名はありませんが、右辺と左辺に同じ変数が現れるので、**複合代入演算子** (compound assignment operator) を用いると「`x += 5;`」と少し短く記述できます。

```
x = x + 5;    ⇔    x += 5;
x = x - 5;    ⇔    x -= 5;
x = x * 5;    ⇔    x *= 5;
x = x / 5;    ⇔    x /= 5;
x = x % 5;    ⇔    x %= 5;
```

- 四則演算すべてに、この複合代入があります。変数部分が長いときに有用です。
- + と = の間にはスペース文字を挟んではいけません。
- + と = を間違えて「`x =+ 5;`」と入れ替えると**単なる代入になる**ので、要注意です。

2.7.3　型変換（キャスト）

整数と浮動小数点数とでは、プログラムでの扱いが違いました。整数の 5 と 2 の割り算では、商は 5 / 2 = 2、余りは 5 % 2 = 1 と整数で得られますが、小数付きの 2.5 を得るにはどうしたらよいでしょうか。

小数付きの結果を得るには、演算前から小数付きの値にしておきます。つまり 5.0/2.0 なら 2.5 が得られます。では、整数の 5 や 2 から 2.5 を得るにはどうしたらよいでしょう。

整数を小数付きにするには、**キャスト** (cast) 演算子で、強制的な型変換を行います。

```
/* 文法 */
(型名)値
```

```
/* 実例 */
(double)5 // →5.0
```

この機能を用いて、「`(double)5/(double)2`」とすると double の「2.5」が得られます。ここで注意してほしいのが、**キャストの順序**です。「`(double)(5/2)`」だと、整数同士の 5/2 (=2) の計算をすませてから double に変換するので、「2.0」になってしまいます。

```
/* 成功 */
int a = 5, b = 2;
double c = (double)a / (double)b;
      2.5      5.0         2.0
```

```
/* 失敗 */
int a = 5, b = 2;
double c = (double)(a / b);
      2.0              2
```

三角形の底辺 w と高さ h を int 型で保持しているとします[*18]。面積の計算は、次のように double 型に揃えるのがよいでしょう。数学の公式では $w \times h \div 2$ と、理想的な 2 で割りますが、プログラムでは、誤差を含む可能性のある 2.0 で割ることになります。

[*17] 前置の「`++x;`」や「`--x;`」もあります。後置ともほぼ同じ動きです。（☞??ページのコラム）

[*18] もっとも、面積を求めるつもりがあるのなら、初めから w, h を double 型にしておくのが現実的です。

```
int w = 10, h = 5;
printf("三角形の面積は %f\n", (double)w * (double)h / 2.0);
```

キャストの優先順位は四則演算よりも高いので（☞??節）、double の計算結果を整数値に丸めるときなど、キャストする式全体をカッコで囲う場面がよく出てきます。

コラム： キャストの使用は最小限に

キャストによる**情報の欠落**には注意しましょう。キャストは double を整数型に丸め込むことも可能です。元の値の小数部分はなくなります。また、大きな整数値を char にキャストすることもできます。上位ビットがなくなって、下位の 1 バイト分だけが取り出されます。

```
(int)3.14 // →3  （注意）
(char)257 // →1  （注意）
```

意図的ならもちろん構わないのですが、キャストは危険でも**強制的**に操作を行うよう指示するものですので、おかしな指示でも警告は出ません。（もちろん不可能ならコンパイルエラーです。）ですから、「試行錯誤してキャストでエラーがなくなった」というのは、まったくあてになりません。つじつま合わせのために、むやみにキャストするのはやめておきましょう。

2.8 マクロ定数

変化しない値である**定数**は、プログラムではどう扱うのでしょうか。変数に代入してもよいのですが、マクロという機能による**マクロ定数**のほうが安全です。間違って値を代入するとエラーになるからです。表 2.2 などのシンボルもマクロ定数で実現されています。

マクロ (macro) は、#define 命令[*19]で「文字の置き換え」を指示したものです[*20]。置き換え文字に定数を指定したものが**マクロ定数** (macro constant) です。マクロ名にも識別子を用いますが、変数とすぐに区別がつくよう**すべて大文字**にするのが習慣です。

```
/* 文法 */
#define マクロ名 置き換え文字
```

```
/* 実例 */
#define EPS      0.000001
#define POW3_3   (3*3*3)
```

#define 文はインデントせずに、**プログラムの冒頭**（#include 文のすぐ下）に記述するのが習慣です。（☞??ページのソースコード??，??ページのソースコード??など）

[*19] # から始まるキーワードは**プリプロセッサ命令**に分類され、1 行の終わりが命令の一区切りです。セミコロンは関係ありません。命令の途中で改行したいなら、行末に \ をつけて、行継続することを示します。

[*20] このため、数値以外にも使い道がありますが、逆に式の優先順位は考慮されないので、定義する数値全体をカッコで囲う必要もあります。（☞??ページのコラム）

2.9 型ごとの限界値を探る

型の上下限の値をソースコード 2.1 で確かめてみましょう。INT_MAX などのシンボルを使う前には、#include <limits.h> や #include <float.h> が必要です。「sizeof(型名)」で型の占める領域のバイト数がわかります。printf() の "%zu" の意味は??項で説明します。

GCC Intel 64 ビットでの実行例は右のようになりました。int が 4 バイト（32 ビット）、long long が 8 バイト（64 ビット）だとわかります。double の精度は保証値（10 桁）よりもかなり良いです。環境によっては、また違った結果になります。

ソースコード 2.1 の実行例（環境依存）

```
INT_MAX=+2147483647
INT_MIN=-2147483648
sizeof(int)=4

LLONG_MAX=+9223372036854775807
LLONG_MIN=-9223372036854775808
sizeof(long long)=8

FLT_MAX=+3.40282e+38
FLT_DIG=6
sizeof(float)=4

DBL_MAX=+1.79769e+308
DBL_DIG=15
sizeof(double)=8
```

ソースコード 2.1　型ごとの上下限の値を表示

```
1  #include <stdio.h>
2  #include <limits.h> // INT_MAX, LLONG_MIN など
3  #include <float.h>  // DBL_MAX, FLT_DIG など
4
5  int main(void) {
6      printf("INT_MAX=%+d\n", INT_MAX);          // 最大値
7      printf("INT_MIN=%+d\n", INT_MIN);          // 最小値
8      printf("sizeof(int)=%zu\n", sizeof(int)); // バイト数
9      printf("\n");
10
11     printf("LLONG_MAX=%+lld\n", LLONG_MAX);
12     printf("LLONG_MIN=%+lld\n", LLONG_MIN);
13     printf("sizeof(long long)=%zu\n", sizeof(long long));
14     printf("\n");
15
16     printf("FLT_MAX=%+g\n", FLT_MAX);
17     printf("FLT_DIG=%d\n",  FLT_DIG);          // 有効精度（桁数）
18     printf("sizeof(float)=%zu\n", sizeof(float));
19     printf("\n");
20
21     printf("DBL_MAX=%+g\n", DBL_MAX);
22     printf("DBL_DIG=%d\n",  DBL_DIG);
23     printf("sizeof(double)=%zu\n", sizeof(double));
24     return 0;
25 }
```

コラム： 多倍長演算

もっと大きな値を扱いたくなったら、あるいは、もっと高精度の小数を扱いたくなったら、多倍長演算が必要になります。CPU（ハードウェア）の直接の命令では計算できなくなくなって、ソフトウェアで模倣（エミュレート）するため、計算速度が数倍も遅くなるのは、原理的に避けられません。

残念ながら C 言語の規格には、多倍長演算のための標準ライブラリはありません。GNU Multi-Precision Library (GMP) という有名なライブラリがありますが、C 言語からは使いにくいので、C++ からの利用を考えたほうがよいでしょう。Java なら BigInteger / BigDecimal、Perl にも bigint / bignum などの標準のライブラリがあります。Python3 なら標準機能に組み込まれています。

コラム： 四捨五入

double 型の `x` を四捨五入して、整数を得たいとしましょう。int にキャストすると小数部分がなくなるので、0.5 を足してからキャストする `(int)(x+0.5)` が四捨五入になる、というのが慣用句でした。しかし、状況次第で問題があります。

- `x` がマイナスなら `(int)(x-0.5)` と、0.5 を引かねばなりません。キャストは**0 方向への丸め**ですから、正負で丸め方向が違います。
- `x` が絶対値の大きな値なら、int ではあふれるかもしれません。

double 型のまま四捨五入するには、数学関数（☞**??**節）を用いて `floor(x+0.5)` あるいは C99 で新設された `round(x)` が簡便です。さらに long long 型で得るなら `llround(x)` もあります。なお、`printf("%.0f", x)` で表示される値の丸め方向は、C99 では `FLT_ROUNDS` （☞**??**項）に従うことが期待されていて、現実には四捨五入とは微妙に異なる、偶数丸めであることが多いです。

2.10 練習問題

1. **[代入・左辺値・初期化・複合代入（☞2.2.2 項、2.2.3 項、2.7.2 項）]**
   プログラム中の `int` 型変数 `x` について、以下を説明せよ。
   (a)「`x = 10;`」と「`10 = x;`」の違いは何か。（コンパイルエラーに着目せよ。）
   (b)「`int x;`」の定義直後の `x` が 0 である保証はあるか。
   (c)「`x = x + 5;`」は、どのような作用があるか。また += 演算子を用いて同じ動作を実現せよ。

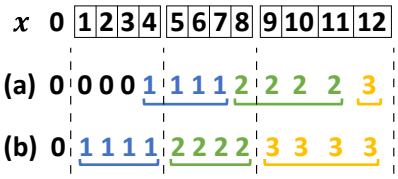
2. **[変数への代入（☞2.2.2 項）]**
   右のプログラムから、無駄を省け。

```c
#include <stdio.h>

int main(void) {
    int a = 10;
    a = 20;
    a = 30;
    a = 40;
    a = 50;
    printf("%d\n", a);
    return 0;
}
```

3. **[左辺値（☞2.2.2 項）]**
   1 元 1 次方程式 $ax+b=c$ の解 x をプログラムで求め、表示せよ。x, a, b, c はすべて double 型の変数として定義せよ。
4. **[スワップ（☞2.7.1 項）]**
   変数 a, b, c がある。$\overrightarrow{a \leftarrow b \leftarrow c}$ とプログラムで 3 つの値を入れ替えてみよ。
5. **[剰余と周期性（☞2.6 節）]**
   度数法の 2 つの角度 a, b $(0 \le a, b < 360)$ の和 $(a+b)$ と差 $(a-b)$ の値を、それぞれ 0 以上 360 未満で求めよ。（つまり、例えば 367 なら 7、−1 なら 359 にせよ。）a, b とも int 型とし、剰余演算子 % のオペランドは負にならない式にせよ。
6. **[剰余と十進数表記]**
   正の整数 x を 123 とするとき、以下の値を表示して確かめよ。（商は整数とする。）
   - x を 10 で割った余りはいくらか。
   - x を 10 で割った商を y とする。y を 10 で割った余りはいくらか。
   - y を 10 で割った商を z とする。z を 10 で割った余りはいくらか。

   この 3 つの値が、x の十進数表記とどのような関係になっているのか考察せよ。x を別の値に変えて試し、考察を補強せよ。
7. **[除算の切り捨て・除算の切り上げ]**
   x 人が 4 人がけの長椅子に、隙間を空けずに座る。次の (a) (b) を、それぞれ x の式で表して、プログラムで表示せよ。（**ヒント**：(b) は、x の 3 ずれた (a) と同じ。）
   (a) 満席になった長椅子の台数（高々 1 台の満席ではない椅子を含めない）
   (b) x 人が座るのに必要な長椅子の台数（満席ではない椅子を含める）



8. **[除算の切り上げ・マクロ定数（☞2.8 節）]**
   整数 x (>0) について、西暦 x 年が何世紀であるかをプログラムで表示せよ。西暦 1 年から 100 年までは 1 世紀、101 年から 200 年までは 2 世紀と、100 年ごとに新しい世紀になる。定数 100 は、マクロ YEAR_PER_CENTURY と定義してから用いよ。西暦 0 年は存在しないが、誤って計算しても「1 世紀」にならない式にせよ。

索引