プログラミング入門 ver 0.32beta

natade2sc@yahoo.co.jp

2010/6/27

はじめに

よく、C 言語を知っている人は他の言語も簡単に覚えられるといいます。なぜかというと、C 言語を知ったことで、プログラミングの基本的なことを学んだからです。それはとてもいいことですが、他の言語に移動することが目的で C 言語を勉強した人は、C 言語特有の言語仕様の勉強が必要なかったということです。ここでは、そのような言語仕様を関係なしに、プログラミングの本質を勉強するという、ちょっと今までにない説明をしていきます。この基本さえ分かれば、他の言語もそれとなく分かるはずです。よってこの本では、この基本となる大事なことのみを説明していきます。

ところで、プログラミングを普通に本とか見て覚えようと思っても、なかなか眠くなり大変です。どうやって勉強するのが一番いいのでしょうか。それは、目標を持つことです。STG を作りたいとかでもいいです。プログラミングの命令などは覚えなくてもいいです。作っていく最中に「どうすればいいのだろう」と悩み、インターネットで検索します。そして、答えを見つけるうちに STG は完成。いつのまにかプログラミングも習得している、というわけです。しかし、プログラミングを始める前に、最低限覚えておく必要のある前知識があります。この「プログラミング入門」では、それらについて、解説していくわけです。

本書では、C・C++・C#・Java の４つの言語をターゲットとして説明していきます。ですが、私は C と Java しか知らないので、内容の正確性の保障はしません。。。。"(ノ＞＜)ノ

また、ここで説明することは、あくまで一般的な話です。方法は無限にあると思うので、知りたい人は独自で調べてみてね！＼(^o^)／

2009 年 6 月 11 日

natade

目次

1 プログラミング言語について 1

2 コンピュータ上での数値の扱い 3

3 コンピュータ上での文字の扱い 6

4 コンピュータ上での真偽値の扱い 13

5 四則演算について 13

6 ビット演算とシフト演算について 16

7 関係演算と論理演算について 19

8 条件演算子について 20

9 変数について 22

1 プログラミング言語について

今回、C・C++・C#・Java で使える基本的知識を説明しようと思っています。その前に、プログラミングとは一体どういうものなのか、そういう前知識について説明したいと思います。

現在、パソコンはノイマン型とよばれる構造になっています。ノイマン型の一番の特徴は、決められた命令を 1 つ 1 つ順番にこなしていく、これだけです。こんな単純なもので、今の Windows の OS、そしてそこで起動しているソフトが動いているわけです。すごいですよね。最近、CPU はマルチコアになったりしていますが、基本的[*1]には、1 つの命令を、1 つずつ実行していくような形。例えば、1GHz の CPU が乗ったパソコンなら 1 ギガ回、1 秒に計算するという考え方でいいです。ソフトがたくさん起動していても、これらのソフトを、何回も高速で切り替えることにより、並列で動いているように見せているわけです。あっ。なんかプログラミング言語の話から脱線しているような気がしてきました。この本では、これからもよくあることですが、ごめんなさい。

ところで、この順番に命令を実行するっていうのが、「命令型プログラミング」と言うものだそうです。つまり、プログラミングは、どの命令を順番に実行するかを書いていく作業なわけです。

さて、この命令というのは、機械語[*2]なわけです。機械語だから、一般的な人間が読んでも分かるはずがありません。ですので、この機械語へ変換できる、ちょっと分かりやすい言語、アセンブリ言語が生まれました。ですが、C 言語とかに比べたら、これもまだまだ大変読みづらいものなのです。なぜ読みづらいのでしょうか。それは、カギ括弧を利用した数式の計算など、ちょっとしたことでも、何行も、しかもコンピュータの動作のことを考えて、書かないといけないからです。つまり、コンピュータ様のための言葉へ、人間が翻訳しないといけないのです。何でも出来るという点で自由度はありますが、人間は完全でもないので、どこかしら間違えてバグを生み出したり、簡単な計算でも何行も書かないといけなかったり、とにかく生産性がないのです。

ここで、もうちょっとなんとかできないのか、ということで、このアセンブリ言語へ翻訳する、さらに高レベルな言語が生み出されていったわけです。例えば、C 言語です。C 言語は、カギ括弧とか使った複雑な計算式も 1 行で書けます。C 言語のコンパイラ[*3]が、C 言語のソースコードから、何行ものアセンブリ言語へ、自動で変換してくれるわけなのです。

C 言語では、大規模なプログラムを作るために、関数を作成していく、イメージ的には**図 1.1** のような方法をとっていました。ちょっと分かりづらいかもしれませんが、目的の機能が動作するための機能をたくさん作って、それらをまとめていくような感じです。1 つの機能だけにしぼって考えることが出来るので、作成しやすいですし、大きなプログラムも作りやすいのです。機能 A を、機能 B の中で使用したりすることももちろん可能です。その後の言語とかは、さらに考えを上位的に発展させて、**図 1.2** のようなものも、作成できるようになっていきました。この方法では、機能を上位レベルで分担できるので、命令の名前の衝突[*4]などを防ぐことができるなど、さらに大規模なプログラミングを行えるようになります。この上位的なプログラミングが出来る言語の 1 つと

[*1] 処理の並列化を考えていない場合

[*2] 実行ファイルのプログラム部分とか。

[*3] ソースコードを翻訳してアセンブリ言語に変えてくれたりするもの。

[*4] C 言語では命令を作ることができます。その名前がすでにある命令と同じ場合はエラーが起きます。

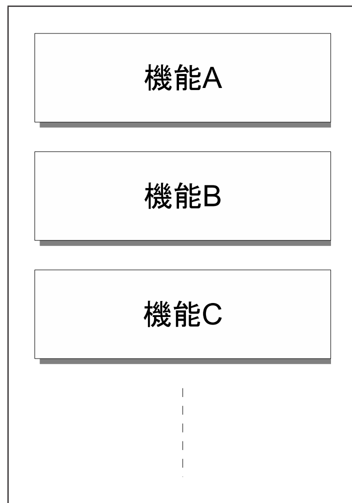

図1.1　プログラミングの考え方 1

図1.2　プログラミングの考え方 2

して C++ があります。C++ は、C の互換性を維持したまま、さらに高レベルなプログラミングが出来るようになったのです。

ところで、ここまでで C・C++ の説明を文章に散りばめておいたので、なんとなくこの 2 つの言語については、どのようなものかが分かったような気がしていると思います。これらの言語は、パソコンでそのまま動かせる機械語に変換するという考え方の言語です。これらを利用して作成した実行ファイルは、長所も多いのですが、短所があります。その短所は、互換性[*5]の問題です。機械語に変換した後の実行ファイルは、16 ビット CPU が 32 ビット CPU になって 32 ビット用アプリケーションに変わったように、CPU によって最適化され機械語に変換されたものです。よって、64 ビット用でコンパイルした実行ファイルは、32 ビット用の OS・CPU では動かないのです。また、既存の OS にある機能も利用できるのですが、この機能を使った場合、OS が変わると動く保障は一般的[*6]にはないのです。もちろん、他の会社が作った OS では、普通はサポートされていませんので、実行することが出来ません。

これらのことから、最近は仮想マシンというものを考えるようになりました。仮想マシンが、OS 間の動作を繋いでくれるのです。こうなれば、この仮想マシンだけを考えてプログラミングをすれば、実行環境のことは何も考えなくて作ることができるようになるのです。それらが、Java(JVM 上で動作) や C#(.NET 上で動作) といった言語です。

C++ の後に開発された、Java 言語や、さらに新しい C#言語は、仮想マシン上で動くだけではなく、人間に優しい設計をしています。人間に優しい設計とは、Unicode を前提とした文字列の扱い方や、真偽値型の追加や、メモリリーク[*7]を考えなくてもよいように、自動開放してくれるガベージコレクション機能などがついたものです。

[*5] OS の機能を使わずにソースコードのみを配布すれば、他の環境でも動きます。
[*6] Windows9X 時代用の機能が、WindowsNT 世代の OS でも使用できるように、サポートされているならば、動くというわけで「一般的に」という言葉を使いました。
[*7] メモリを確保したら、開放をする必要があります。開放をし忘れると、メモリをどんどん確保し続けることになり、プログラムが止まってしまう原因になります。

2 コンピュータ上での数値の扱い

実は、コンピュータで扱う数値には、いくつかの種類があります。それは整数か実数かという種類です。また、正負についても考えなくてはいけませんね。まあそれは後にして、整数と実数の違いを説明します。

整数は、小数点が存在しない数値です。といっても、固定少数点も整数に含めると思うので、この解説だと誤解がありそうですが。とにかく例を挙げてみます。次のような値、−12、123、23、439249、123103、123942、0。これらは整数です。また、数値を $\frac{1}{100}$ として考えるならば、−0.12、1.23、4392.49、1231.03、1239.42、0.00 と整数の考え方で、小さな数値も扱うことができますね。これを固定少数点と呼びます。

実数の説明です。実数は、ちょっと説明しにくいのですが、$\frac{1}{3}$ や $\sqrt{2}$ や π、e などは実数です。これらの値は、小数点が無限に続くので、先ほど説明した整数の考えでは表すことができませんね。虚数を含めない数値一般と考えてみてください。

よく、コンピュータでは 2 進数で計算していると話は聞いたことがあると思います。そのとおりです。0b0100100101000 [*8]みたいな感じでコンピュータ上では値を表しています。しかし、そんな 0b1001 とあっても分かりづらいので、適当なところで区切って、その組を 1 バイトとして考え使用しようということを、誰かが決めましたわけです。その 1 組であらわす 2 進数の値の数は 8 個です。そう、一般的に 8 ビットが 1 バイトを表すのです。ここで、1 バイトで 10 進数ではいくつまで数えることができるでしょうか。ちょっと**表 2.1** を作ったので見てください。

すみません。表が足りなくなってしまいました。8 ビットの情報量があれば、最大 10 進数でいくつまで数えられるか、この表から想像できますでしょうか。もう 1 つ同じ大きさの表があれば……と思った方、残念ながら間違いです。もっと大きくなります。表全体でこの 16 倍の大きさが必要なのです。結果から述べると、16 進数では 0xFF、10 進数では 255[*9]まで数えることができるのです。ここで、言いたいことは、コンピュータ上では、通常 8 ビット＝ 1 バイトが最小単位であり、1 バイトでは 256 種類の表現ができるということです。

表2.1 自然数の場合の 2 進数・10 進数・16 進数

16 進数	2 進数	10 進数	16 進数	2 進数	10 進数
0x00	0b00000000	0	0x08	0b00001000	8
0x01	0b00000001	1	0x09	0b00001001	9
0x02	0b00000010	2	0x0A	0b00001010	10
0x03	0b00000011	3	0x0B	0b00001011	11
0x04	0b00000100	4	0x0C	0b00001100	12
0x05	0b00000101	5	0x0D	0b00001101	13
0x06	0b00000110	6	0x0E	0b00001110	14
0x07	0b00000111	7	0x0F	0b00001111	15

[*8] 0x を最初に付けた場合 16 進数を、0 のみをつけた場合は 8 進数を、0b を最初に付けた場合 2 進数を表します。
[*9] 0 含めて 256 種類あるので 255 まで数えられる。

では数値は、自分が決めた自由なバイト数で表現できると思いますよね。しかし、実は決められたルールが数値には存在して、その中からしか選べないのです。**Σ（'◇'*）エェッ**!? それは、1 バイト、2 バイト、4 バイト、8 バイトのメモリを利用する数値です。*10なぜ、この 4 種類なのでしょうか。きっと、これらの数値は、コンピュータに関連のある 2 の累乗だからであり、それ以上の数値はあんまり扱わないから、この 4 種類にしたのでしょう。

ここで、先ほど話した負数について、整数の場合を考えます。負数とは –（マイナス）がつく数値ですね。先ほど 1 バイト（8 ビット）では、256 種類の数値を数えられるといいました。たしかにその通りで、0 ~ 255 まで表すことが出来ますね。しかしよく考えてみてください。このままでは正数しか表現できません。そこで、『ある 1 ビットが 1 ならば負としよう。』という考え方が生まれました。（**表 2.2**）

表2.2 整数の場合の 2 進数・10 進数・16 進数

16 進数	2 進数	10 進数	16 進数	2 進数	10 進数
0x00	0b00000000	0	0xFF	0b11111111	-1
0x01	0b00000001	1	0xFE	0b11111110	-2
0x02	0b00000010	2	0xFD	0b11111101	-3
0x03	0b00000011	3	0xFC	0b11111100	-4
0x04	0b00000100	4	0xFB	0b11111011	-5
0x05	0b00000101	5	0xFA	0b11111010	-6
0x06	0b00000110	6	0xF9	0b11111001	-7
0x07	0b00000111	7	0xF(	0b11111000	-8

表 2.2 を見ると分かるとおり、負数は 2 進数の左端のビットが 1 になっていますね。つまり、一番左端のビットを見て、負数か正数か判断しようというわけなのです。しかし、この場合は 2^7 まで数値しか表せませんね。負数の表現に 1 ビット使用しているわけなので、ここで表すことができる数値の範囲は –128 ~ 127 *11です。正数だけ表す必要があるなら 0 ~ 255。これが 1 バイトの限界です。

1 バイトは、主にバイナリ*12の表現で使用されます。理由としては、ファイルはバイト単位なので、ファイルの 1 バイトのみに処理を加えたいときとか、そういう場合で、1 バイトの数値が利用されるのです。後は、JIS 文字とかです。これは文字の章で説明します。話は脱線しますが、拡張子に.bin というのを見たことありますよね。あれは binary data の略です。

2 バイトの数の説明は、4 バイトの数も含めて説明します。『なぜ含めてしまうの』と思うかもしれません。まあ説明を聞いてください。整数の表現に関しては、バイトが増えた分、表すことができる数値も、その分増えますが、**表 2.1**、**表 2.2** のような感じです。ちなみに、1 バイトの数で表すことができる値の種類が、2^8 種類だったように、2 バイトでは 2^{16} 種類、4 バイトでは $2^3 2$ 種類も表現が出来るようになります。では、話を戻して、なぜ含めちゃうのかという話です。CPU で 32

*10 C++ では long double といった 16 バイト型などと、他にもあると思いますが、この一般的な 4 種類で話を進めていきます。

*11 –127 ~ 128 ではないのは、上の表をじっくり眺めると分かります。

*12 本来は 2 進という意味ですが、バイナリデータとはバイトの並びを表します。

ビットまで扱うことが出来るとか、NINTENDO64 は 64 ビットまで扱うことができるとか、そのような話を聞いたことありませんか。最近は Windows の 64 ビットバージョンもあるようですね。これらは一体何を表しているのか考えたことがありますか。実は 1 回の処理で、どのビット数までの数値を扱えるかと言うことなのです。

ちょっと話が長いので次の段落にしました。例えば、現在 2009 年の一般的なパソコンの CPU は 32 ビットまでの数値を一辺に扱うことが出来ます。[*13]ということは、1 バイトの数を扱う時間と、4 バイトの数を扱う時間は同じになってしまうということです。8 バイトは単純計算で、その 2 倍ですね。CPU が 32 ビットまで扱うことができるので、32 ビットで表すことができる限界の数値が、一番効率がいいということです。そのために、4 バイト（32 ビット）の数値が、よく利用されるというわけです。2 バイト（2^{16} ビット）では自然数のみ数えるとしても、65535 までしか扱えなく、使いづらいので、あまり使用されていないようです。[*14]

8 バイト（64 ビット）の整数は、あんまり使用されませんが存在します。説明のとおり、整数として利用することはあまりありません。UNIX 時間[*15][*16]を表す場合や暗号化に利用するぐらいです。というわけで、8 バイトの数値表現は、実数として利用することが多いです。

ここで、実数の話に戻ってきましたね。今まで話したのは整数の場合の話です。といっても使用するバイト数に関しては、整数と同じです。先に言ってしまうと、4 バイトと 8 バイトで表す実数が使用されています。

実数は、内部では浮動小数点という形で処理をしています。整数とは扱い方・考え方が大きく異なります。これについて詳しく説明するのは難しいので、小数を使いたい方はなんとなくこちら選んでください。先ほど整数を利用した固定小数点の話にふれましたが、こちらは浮動小数点です。0.1 も 0.001 も 0.000001 も扱えます。安心してください。大きい数も使えます。これは内部では、0.004×2^N というように情報を扱っているためです。化学の勉強で有効桁数とかやりましたよね。あれの概念です。有効桁数の数と、10^N という[*17]わけなのです。ところで、この概念の場合注意することがあります。有効桁数が 3 の場合、$1.01 \times 2^{10} + 1.00 \times 2^0 \rightarrow 1.01 \times 2^{10}$ なのです。小さな数は、大きな数に押されて消えてしまうわけです。これを意識せずに、1 ずつ実数で足し算していくようなプログラムを作った場合、ある大きな値で有効桁数からはみでてしまい、それ以上足しても増えない、という状況が生まれたりするのです。これは**情報落ち**という現象です。

ここで学んだことをまとめると、**表 2.3**、**表 2.4** のようになります。

ここで、改めてメモリの中ではどうなっているかを比較（**表 2.5** を参照）してみましょう。基本となる 1 バイトをつなげることで、大きな整数を表していることが分かります。2 バイトだけで数値上足りるから、節約のために 2 バイトを選ぶという人もいるかもしれませんが、基本的に 4 バイトを選べばいいです。理由として、メモリの容量を気にしなくてもいい世の中になったからです。32 ビット CPU なので、4 バイトの整数を選ぶのが基本なわけです。[*18]

[*13] 最近は 64 ビットも扱える CPU が多いですが OS が 32 ビット用なので、32 ビットと考えます。
[*14] CD の波形は 1 サンプルあたり 16 ビット整数値で保存されています。
[*15] 1970 年 1 月 1 日 0 時 0 分 0 秒からの経過秒数。
[*16] 32 ビット整数値で、UNIX 時間を数えた場合、2038 年までしか数えることができません。この問題は 2038 年問題と呼ばれています。
[*17] コンピュータ上では 2^N
[*18] 将来、8 バイトの整数が基本になる可能性も十分あります。

表2.3　整数について

バイト数	1 バイト	2 バイト	4 バイト	8 バイト
ビット数	8 ビット	16 ビット	32 ビット	64 ビット
表現の種類	256	65536	4294967295	18446744073709551616
最小自然数	0	0	0	0
最大自然数	256	65536	4294967295	18446744073709551616
最小整数	−128	−32768	−2147483648	−9223372036854775808
最大整数	127	32767	2147483647	9223372036854775807

表2.4　実数について

バイト数	4 バイト	8 バイト
ビット数	32 ビット	64 ビット
名称	単精度実数	倍精度実数
指数部	8 ビット	11 ビット
仮数部	23 ビット	52 ビット
最大桁数	256 桁	2048 桁
有効数字	約 6 桁	約 16 桁

表2.5　指定した使用バイト数で、ある整数を表した時のメモリ内部

種類	数値	メモリの内部
1 バイト整数	128	0x80
1 バイト自然数	255	0xFF
2 バイト整数	255	0x00FF
2 バイト整数	-256	0xFF00
4 バイト整数	-256	0xFFFFFF00

整数で 4 バイトを選ぶように、実数では 8 バイトを選択するのが一般的です。実数の場合は、4 バイトより 8 バイトを使用したほうが、計算速度が早いと言われています。これは最適化が下手なコンパイラの場合なので、実際は 4 バイトの方が計算速度が早いです。ですが、ゲーム以外の一般的なソフトを作る場合は、8 バイトでも十分です。

16 バイトを使用して、10 進数の浮動小数点を表す数値[*19]も存在しますが、ここで紹介した 2 種類（整数と浮動小数点）に関しては、大体どの言語にもあるので覚えておいてください。

3　コンピュータ上での文字の扱い

数値表現の次は、文字列表現について説明します。プログラミング中は特に意識する必要は無いと思いますが、実は結構ややこしいです。ゲームとか作る分にはあんまり関係ないことが多いかもしれませんが、文字列操作をするときは、十分この知識を知っておく必要があると思います。文字列操作とは、文字を検索したり、文字と文字をくっつけたり、文字を大文字にしたり、n 文字数抜

[*19] C#の decimal 型。

き出したりというような、文字郡に対しての操作です。事務系には欠かせない処理ですね。普通の言語では、これ専用の自動で処理してくれる命令が入っているので、そっちを使えば基本はいいのです。しかし、そういう命令を使ってはいけない場面や、使えない言語とか使用すると悲惨です。まあ知っておいて損は無いので、豆知識程度で知っておくといいかもしれません。

文字列もコンピュータ上では、10101 のように 2 進数で表現されているわけなのです。ここで 1 バイト（8 ビット）は、説明したとおりコンピュータの基本単位です。1 バイトで 256 種類の何かを、表現することができますよね。用は、この 256 種類に対応するように文字列をあてはめていけばいいわけです。英語圏の人はね。（^^;）

え？英語圏は？……はい。そうです。日本語が大体 256 種類に収まるわけないじゃないですか。漢字とか特に。というわけで、日本語用の変換など、世界中には数多くの変換の約束事が決められました。文字コードというものです。でも、世界中にたくさんの種類があるのだったら、全部覚えるのも大変だし、無理じゃないかと思うかもしれません。もちろんその通りです。というわけで重要な文字コードの種類をしぼって説明していきます。

まず、基本的な文字コード ASCII について説明します。これは日本語が使えないので、日本語については、またちょっと後で説明します。この ASCII は 1 バイトで 1 文字表します。簡単ですね。全部ではありませんが、そのコード表（**表 3.1**・**表 3.2**）を提示します。

表3.1 英語の大文字のコード表

10 進数	16 進数	文字	10 進数	16 進数	文字	10 進数	16 進数	文字
65	0x41	A	75	0x4B	K	85	0x55	U
66	0x42	B	76	0x4C	L	86	0x56	V
67	0x43	C	77	0x4D	M	87	0x57	W
68	0x44	D	78	0x4E	N	88	0x58	X
69	0x45	E	79	0x4F	O	89	0x59	Y
70	0x46	F	80	0x50	P	90	0x5A	Z
71	0x47	G	81	0x51	Q	91	0x5B	[
72	0x48	H	82	0x52	R	92	0x5C	¥
73	0x49	I	83	0x53	S	93	0x5D	]
74	0x4A	J	84	0x54	T	94	0x5E	ˆ

せっかく表を用意しましたが、こんな対応表は覚えなくていいです。とりあえず、注目してほしいのは 16 進数です。大文字の文字コードと小文字の文字コード、何かよく似ていませんでしょうか。勘の鋭い方は分かったはず。つまり大文字の文字コードに 0x20 足せば、小文字になるのです。面白いですね。よく出来ています。

ところで、大文字の文字コードの 92 に注目。円マークですよね。「なんで英語圏向けに作られた文字コードに円マークが入っているのだ」って思いますよね。全くあなたは勘の鋭い方です。その通り、実際は、¥ マークではなくバックスラッシュ（\）が入ります。でも日本語版の Windows 使っている方は、残念ですが表示できません。文字コードの一種である Unicode を使えばいいのですが。またこれは後に。

話は変わりますが、ASCII は 1 バイト 8 ビットで表すと書きましたよね。これ実は、元々 7 ビッ

表3.2　英語の小文字のコード表

10 進数	16 進数	文字	10 進数	16 進数	文字	10 進数	16 進数	文字
97	0x61	a	107	0x6B	k	117	0x75	u
98	0x62	b	108	0x6C	l	118	0x76	v
99	0x63	c	109	0x6D	m	119	0x77	w
100	0x64	d	110	0x6E	n	120	0x78	x
101	0x65	e	111	0x6F	o	121	0x79	y
102	0x66	f	112	0x70	p	122	0x7A	z
103	0x67	g	113	0x71	q	123	0x7B	{
104	0x68	h	114	0x72	r	124	0x7C	|
105	0x69	i	115	0x73	s	125	0x7D	}
106	0x6A	j	116	0x74	t	126	0x7E	˜

ト (128 種類) で文字表すことが出来るように設計されているのです。英数字だから 128 種類もあれば事足りるわけです。[*20]昔は、1 ビットでも大切に使いたかったのでしょうね。まあそれはおいておいて、この残りの 1 ビットは何かに使えそうですよね。これについてもまた後に。

なんかどんどん後回しにしているような感じですが気にしないでください。スペースの関係上です。ホントです。

表3.3　数値の文字コード表

10 進数	16 進数	文字
48	0x30	0
49	0x31	1
50	0x32	2
51	0x33	3
52	0x34	4
53	0x35	5
54	0x36	6
55	0x37	7
56	0x38	8
57	0x39	9

では話を、ASCII の話に戻します。**表 3.3** は、文字としての数値を表すコードの表です。これを示して何がいいたいのかというと、数値を表す文字コードから、0x30 を引けば、実際の 10 進数の数値となることです。逆にすれば、文字になります。ASCII コードについては、これぐらい知っていれば大丈夫です。`(^^)b`

ここで一旦メモリの話に移りたいと思います。ASCII なら 1 文字が 1 バイトなので、例えば ABC と 3 文字の文章は 3 バイトが必要だと思いますよね。（**表 3.4**）

惜しいですが、実は違います。ウソと思いになるかもしれません。本当は 4 バイト使用しなくて

[*20] 6 ビットのもあったようですがそれはまたのお話。

表3.4　ABC を格納した場合のメモリの内部？

文字	A	B	C
メモリ	0x41	0x42	0x43
0 からのアドレス	0	1	2

はいけません。文字だけで 3 バイト。では後の 1 バイトはなんでしょうか。NUL 文字[*21]と呼ばれる文字が必要なのです。NUL 文字は文章の終わりということを知らせるために必要な文字で、制御文字と呼ばれる文字です。制御文字は、他にも改行用の文字、タブ用の文字などがあります。というわけで正解は**表 3.5** のような感じで、4 バイト分をメモリに入れなくていけません。文字列を表現する場合は、この最後の 1 バイトを忘れないようにしてください。

表3.5　ABC を格納した場合のメモリ

文字	A	B	C	NUL
メモリ	0x41	0x42	0x43	0x00
0 からのアドレス	0	1	2	3

ところで、この NUL 文字の文字コードは 0x00 です。これを**表 3.6** のように、アドレス 1 に入れてみます。これをディスプレイで表示させた場合、A しか表示されません。「えっ。C は表示されないの？」と思うかもしれませんが、B の変わりに入った NUL によって文章が終了しているので、A しか表示されないのです。

表3.6　ABC を格納した場合のメモリの B を NUL にしました

文字	A	NUL	C	NUL
メモリ	0x41	0x00	0x43	0x00
0 からのアドレス	0	1	2	3

改行のために必要な制御文字は、1〜2 バイトが必要です。なんでこんな曖昧なんだと思うでしょう。実は 3 種類方法があるのです。ここでキーワードとなるのは**表 3.7** です。このうちどれを使っても、改行になります。Windows では CR+LF を採用しているので、CR+LF を選んでおけば大丈夫です。効果は、タイプライター時代の名残のようなものです。

表3.7　改行のコード表

文字	LF	CR	CR+LF
メモリ	0x0A	0x0D	0x0D 0x0A
効果	改行	復帰	改行復帰
バイト数	1	1	2

ところで、「PC 上で NUL 文字を入力するのにどうやってやればいいんだ」と思うかもしれません。これはプログラミング言語によるのですが、自動で文字の後ろに付けてくれるので気にしない

[*21] ヌル文字・ナル文字と読みます。

でください。心配なら、最初に用意されたメモリの容量だけ、全て NUL で埋めてしまうのがいいでしょう。改行コードを入力したい時は、「¥」と入れてください。内部で勝手に改行コードに置き換えてくれます。え、改行したくないけど「¥」って打ち込みたいって。心配無用です。改行コードを入力したいときは「¥n」、¥（0x5C）マークを入力したい時は「¥¥」として下さい。これらはエスケープ文字とも呼ばれます。プログラミング言語によって、使えるエスケープ文字がありますが、一応リストは**表 3.8** のようになっています。改行のエスケープ文字は、普通は使えるので覚えておいて下さい。なお、表を見ると分かるのですが、エスケープ文字は最初に 0x5C の ¥ が必要です。

表3.8　エスケープ文字の一例（言語によって使用可能・不可能はあります。）

効果	エスケープ文字
NUL	¥0
タブ文字	¥t
改行	¥n
¥	¥¥

そろそろ、日本語の話に移りたいと思います。多分、ASCII コードでいっぱいいっぱいかもしれませんが、がんばって下さい。今までで覚えたことは、他の文字コードでも基本となっていて、使えることが多いのです。

日本語を扱うために、Shift-JIS という文字コードについて説明します。この文字コードは Windows で昔[*22]、使用されていたので覚えておくというわけです。昔というのは、Windows9X 時代です。Windows2000 からは内部で違う文字コードに変えたようです。これは Unicode というものなのですが、後で話します。

では、Shift-JIS について話します。先ほどちょっと触れたとおり、「ASCII が 7 ビットで表現できる」というのを利用して表現されます。1 ビットを、数字でいう負数のチェックのように、日本語かどうかのチェックに利用するというわけです。具体的に言ってしまうと、日本語は 2 バイト使用します。携帯電話のメールで、残り何文字分送れるかが表示されていて、日本語を入力すると 2 文字減るというのは、これが原因です。

ところで、1 バイトの半角カタカナというものもあります。これが日本語を PC で表現する時の最初でした。だって、英語しか打てないのは、さすがに日本では使いにくいですよね。ASCII コードに、半角カタカナを余りの 1 ビットを使用して実装したのです。これが JIS8 コード（8 ビット JIS）と呼ばれるものです。昔のパソコンって半角カタカナしか画面に表示できない、とかそういうイメージありますよね。それです。

Shift-JIS はさらにこれを拡張しました。メモリを 2 バイト分を使用して、ひらがな・漢字を表現できるようにしたというわけです。ところで、ASCII コードと JIS8 コードの関係は、**表 3.9** となっています。ASCII コードで使用しなかった 8 ビット目の、空いているメモリ領域に半角カタカナを定義したのです。

新しく 2 バイト使用する Shift-JIS コードを考えるのは、これらに重複しないように配置しなくていけません。つまり、未使用の部分、「0x81〜0x9F（129〜159）」と「0xE0〜0xFC（224〜252）」

[*22] 今も普通に使うことも多いです。

表3.9 ASCII コードと JIS8 コードの関係

文字コード	種類	使用しているメモリ領域
ASCII/JIS8	制御文字	0x00 ～ 0x1F,0x7F（0 ～ 31,127）
ASCII/JIS8	記号と英数字	0x20 ～ 0x7E（32 ～ 126）
未定義	未定義	0x81 ～ 0x9F （129 ～ 159）
JIS8	半角カタカナ	0xA0 ～ 0xDF （160 ～ 223）
未定義	未定義	0xE0 ～ 0xFC （224 ～ 252）

を Shift-JIS コードの 1 バイト目に使用したのです。1 バイト目といいましたが、2 バイト目は、この 1 バイト目と範囲がこれとは異なります。1 バイト目で、Shift-JIS だと分かれば、2 バイト使用することはあきらかになるので、2 バイト目の範囲をこれより広くとったわけです。つまり、日本語を表示するための Shift-JIS は、**表 3.10** のようになります。

表3.10 2 バイト使用する Shift-JIS コードの範囲

1 バイト目	2 バイト目
0x81～0x9F（129～159） 0xE0～0xFC（224～252）	0x40～0x7E（64～126） 0x80～0xFC（128～252）

実際に、どんな日本語が、どのようなコードかは覚えなくてもいいです。では、「なぜここまで詳しく説明したのか。別に覚えなくてもいいじゃん」と思うかもしれません。もちろんそうですが、このことを知らないと、『メモリの中に格納された文字列が、何文字であるか』を、調べることが出来ないのです。単純に 1 バイト 1 文字と数えてみて下さい。日本語が含まれると、バイト数が大きくなるため、文字数が実際よりたくさん判定されてしまうのです。よって、もしメモリに格納された文字の数を調べたり、指定した文字数のみ抜き出したりする場合は、1 バイト目が、「0x81～0x9F（129～159）」と「0xE0～0xFC （224～252）」であるかを調べて、この範囲なら、次のバイトをカウントせずに飛ばす。という処理を組み込まなくてはいけないのです。面倒なら「英数字のみ入力を受け付ける」とかにすればいい話なのですが、知っておいてください。

話が脱線するのですが、2 バイト目に注目してみて下さい。「0x40～0x7E （64～126）」という範囲がありますよね。これ、中に重大な文字が含んでいます。それは、先ほど話したエスケープ文字 0x5C の ¥ です。よって、これが含まれる日本語は、ちゃんと処理していないソフトの場合、文字化けする可能性があるのです。もうこれはどうしようもないのですが、ちゃんとそういうのを考えて、文字コードを設計しておいてほしいですね。

というわけで、文字扱うのにも大変ということが分かりましたね。本来ならば、こういうのは人間側でいろいろやるのではなく、パソコン側が自動でやってくれればいいのです。そもそもの原因は、文字を数値型と同じように扱うということから始まったのです。そのため、最近は、データ型において、文字は文字型、数値は数値型と分けるプログラミング言語もあります。後で話しますが、C 言語と C++ 言語では、文字型は厳密には存在しなく、このように数値型として扱っています。でも、これより後に出来た Java 言語、C#言語は文字型というのがあるので、そう混乱しなくてもいいです。

文字型として、しっかり考える場合、文字コードとして Unicode が使用されます。これは、英数字用文字コード・日本語用文字コードとかそういう枠組みではなく、全ての文字を同じ文字コードの中に含めてしまうというものです。もちろんその分バイト数は増えますが、これらのことをいちいち考えなくてもいいので、とても便利です。Shift-JIS の「ひらがな」を使用したソフトは、海外では指定したコードにしないと表示されないです。しかし、Unicode の「ひらがな」ならば、海外でもどの国でも、「ひらがな」として認識されるのです。すばらしい話です。といっても Unicode にも、種類がありややこしい事態になっています。ちなみに、Windows では UTF-16 というコードを利用しています。Java の char 型でも、この UTF-16 を使用しています。

ここで、UTF-16 の話を進めていきます。UTF-16 では、基本的[*23]に全ての文字を 2 バイトで表すので、半角英数字や、ひらがなが混じった文字列の文字数も全バイト数に 2 を割った数字になります。先ほどの**表 3.5** で示した ASCII で ABC を格納したように、Unicode で、格納した場合のメモリを**表 3.11** のようになります。1 文字に 2 バイト使うようにはなりましたが、よくみると ASCII とメモリの内容は 00 と 1 バイト増えたぐらいで変わっていませんね。英数字は ASCII コードが基本となっているのです。というわけで、ASCII コードは Shift-JIS や Unicode にも使われている大事なコードというわけです。

表3.11　UTF-16 で ABC を格納した場合のメモリ

文字	A	B	C	NUL
メモリ	0x0041	0x0042	0x0043	0x0000
0 からのアドレス	0	2	4	6

プログラミングで文字列を表したい場合は、"（ダブルクォーテーション）で囲む必要があります。例として「"日本語"」のようにです。では、この文章の途中で"を文字として打ち込みたい場合はどうするのでしょうか。これもエスケープ文字として「¥"」のように打つことが出来ます。「¥"」を画面で表示させると「"」のように「¥」が消えて、しっかりダブルクォーテーションのみが表示されます。

文字列とは、文字の塊です。1 文字の半角英数字の場合は、’（シングルクォーテーション）で表すことがあります。「’A’」などです。この場合、コンパイラは文字としてではなく、自動で「0x41」のように数値に変換してくれます。Unicode を特殊なことをしなくても、使用できる言語であれば、全角の日本語も、シングルクォーテーションで囲んで大丈夫です。（**表 3.12**）

表3.12　ダブルクォーテーションとシングルクォーテーションについて

種類	使用目的	使用例
ダブルクォーテーション	文字列（文章）	"ABC" or "日本語"
シングルクォーテーション	文字（’1 文字’ の数値等）	’A’ or 0x41
	半角の文字の塊（C 言語）	’ABC’

[*23] サロゲートペアの場合は、2 バイトより大きくなります。

4 コンピュータ上での真偽値の扱い

真偽値（真理値）について説明します。真偽値とは、「はい」か「いいえ」か、「真」か「偽」か、「true」か「false」かなどの 2 択の情報です。なぜ、これを勉強しないといけないかというと、真偽を使用しなければ、何も使えないプログラムしかかけないからです。いや。使えなくはないのですが。ただ毎回何か入力があっても、同じ結果しか出ないようなものしか作れません。真偽という概念と制御文[*24]があれば、それを利用して「こういう場合はこちら」「こういう場合はこちら」という場合分けの処理を作ることが可能となるのです。というわけで、この概念について説明しようと思いますが、日常でも、「はい」か「いいえ」なんて使っているので、特に説明する必要もありませんよね。

では、メモリ上ではどうなっているのでしょうか。これは特に気にする必要もなく定義も決まっている言語もあれば、コンパイラに任せた言語もあります。1 ビット[*25]で表現できるからといって、メモリの容量を 1 バイトしか使用していないという保証もないのです。

ところで注意があります。C++ 言語には真偽型があるのですが、有名な C 言語では、真偽型というのがない[*26]のです。では、C 言語での真偽値の判断はどうやって行っているのでしょうか。どうなっているかは、**表 4.1** のように、0 以外か 0 というように、真偽値ではなく数値で管理しています。実は、C++ 言語でも互換性を保つために、真偽型はあっても内部的には C 言語と同じような感じになっています。また、Java や C#は、数値とはまた違った真偽型というものを、しっかり切り分けて考えていることが分かりますね。

表4.1 各言語における真偽値の扱いについて

言語	型の名前	真	偽
C	未定義	0 以外	0
C++	bool	0 以外 or true	0 or false
C#	bool	true	false
Java	boolean	true	false

5 四則演算について

四則演算について話します。四則演算は整数と実数に対して行える演算で、**表 5.1** のようなものがあります。

計算は基本、左から行っていきます。掛け算と割り算は、優先度が高いので先に計算します。内部での計算は、**式 5.1, 5.2** のように「数式を優先度が高いのから、2 組にして計算する」という方法で計算しています。**式 5.2** では、掛け算の計算順位が高いので、掛け算を先に計算を行っていま

[*24] 制御文については、後で話します。
[*25] **はい**か**いいえ**。
[*26] C99 という規格からはあるようです。

表5.1　四則演算

演算子	意味
+	加算
−	減算
*	掛け算
/	割り算

す。ですが、掛け算や割り算といった計算の順位を、無視する言語も存在しますので注意してください。

$$1 + 2 + 6 \rightarrow ((1 + 4) + 6) \rightarrow (5 + 6) \rightarrow 11 \tag{5.1}$$

$$\begin{aligned} 6 + 3 + 5 \times 3 - 5 &\rightarrow (((6 + 3) + (5 * 3)) - 5) \rightarrow (((6 + 3) + 15) - 5) \\ &\rightarrow ((9 + 15) - 5) \rightarrow (24 - 5) \rightarrow 19 \end{aligned} \tag{5.2}$$

整数の中でも、「自然数と整数」「利用するメモリサイズ」など、いろいろな種類がありました。これらを混同して計算した場合どうなるのでしょうか。言語によって異なるのですが、「コンパイラ内部の優先する数値型」又は「整数と実数なら実数優先」のように合わせられることが多いです。結果をコンパイラが一般的に使用している型へ変換する場合が**式 5.3**。メモリの容量が大きい方の数値を計算の基準にする場合が**式 5.4**。式の左側の型を計算の基準にする場合が**式 5.5**。見た目は同じなのですが、これも言語やコンパイラによって動作が違うので、注意が必要です。

$$(1\text{バイト整数})100 + (1\text{バイト整数})100 \rightarrow (4\text{バイト整数})200 \tag{5.3}$$

$$(1\text{バイト整数})100 + (4\text{バイト整数})100 \rightarrow (4\text{バイト整数})200 \tag{5.4}$$

$$(1\text{バイト整数})100 + (4\text{バイト整数})100 \rightarrow (1\text{バイト整数})200 \tag{5.5}$$

実数の場合も、整数と同じように注意が必要です。結果をコンパイラが一般的に使用している型へ変換する場合が**式 5.6**。実数を優先する場合が**式 5.7**。式の左側の型を計算の基準にする場合が**式 5.8**。

$$(4\text{バイト実数})10.0 + (4\text{バイト実数})10.0 \rightarrow (8\text{バイト実数})20.0 \tag{5.6}$$

$$\begin{aligned} (4\text{バイト整数})6/(8\text{バイト実数})2.9 &\rightarrow (8\text{バイト実数})6.0/(8\text{バイト実数})2.9 \\ &\rightarrow (8\text{バイト実数})2.068... \end{aligned} \tag{5.7}$$

$$(4\text{バイト整数})6/(8\text{バイト実数})2.9 \rightarrow (4\text{バイト整数})6/(4\text{バイト整数})2 \rightarrow (4\text{バイト整数})3 \tag{5.8}$$

他にも、いろいろなタイプがあると思います。実際どうなっているか把握するのは面倒なので、**明示的に指定する方法**をおすすめします。明示的というのは、この数値は、このタイプなのだとコンパイラに教えるということです。ですが基本的に、整数では 4 バイト整数*27、実数では 8 バイ

*27 自然数ではないので注意。

ト実数が多いのです。よって、これを基準して、他の型の場合は、明示的に指定するのがおすすめです。明示する方法は、言語によって異なります。

整数同士の割り算や、実数から整数に明示的に変換する場合、**式 5.10** のように切り捨てになるので、そういう点は注意してください。また、0 で割り算しようとしても、エラーが出るので気をつけてください。実数の場合は、Inf*28となりエラーにならない仕様の言語もあります。実数では結果が定義されないような計算。例えば、$\frac{0}{0}$ の計算では、NaN*29になります。

整数同士の割り算で、結果の小数点以下が切り捨てになる場合の例が**式 5.9**。実数同士の割り算で、結果も実数なので小数点がつく場合の例が**式 5.10**。整数同士の演算で 0 を割った場合に、除算のエラーが出て、プログラムが停止する場合の例が**式 5.11**。実数同士の演算で 0 を割った場合に、除算のエラーが出て、プログラムが停止する場合の例が**式 5.12**。実数同士の演算で 0 を割った場合に、除算のエラーが出ずに、無限大が値として入る場合の例が**式 5.13**。実数同士の演算で 0 を 0 を割った場合に、結果が定義されていないために非数が値として入る場合の例が**式 5.14**。

$$(4\text{バイト整数})100\ /\ (4\text{バイト整数})30 \rightarrow (4\text{バイト整数})3 \tag{5.9}$$

$$(8\text{バイト整数})100.0\ /\ (8\text{バイト整数})30.0 \rightarrow (8\text{バイト整数})3.333... \tag{5.10}$$

$$(4\text{バイト整数})100\ /\ (4\text{バイト整数})0 \rightarrow 0\text{の除算エラー} \tag{5.11}$$

$$(8\text{バイト実数})100\ /\ (8\text{バイト実数})0 \rightarrow 0\text{の除算エラー} \tag{5.12}$$

$$(8\text{バイト実数})100\ /\ (8\text{バイト実数})0 \rightarrow (8\text{バイト実数})\mathrm{Inf} \tag{5.13}$$

$$(8\text{バイト整数})0\ /\ (8\text{バイト整数})0 \rightarrow (8\text{バイト実数})\mathrm{NaN} \tag{5.14}$$

剰余算について説明します。剰余算は、割り算の余りの計算です。数学では mod と書きます。符号は「%」です。実数は割り算しても余りがありません。よって実数の余りを計算しようとするとエラーが出ます。整数同士の剰余算の例が**式 5.15**。実数同士の剰余算の例が**式 5.16**。

余りは、倍数を知るときによく使用されます。例えば、数値が 2 の倍数かを知りたい場合は、2 の余りが 0 であるかどうかを調べればいいのです。**パチッ☆-(^-‘ *)b ナルホド**

なお、HSP*30の場合は「%」の代わりに「¥」を使用します。

$$(4\text{バイト整数})100\ \%\ (4\text{バイト整数})30 \rightarrow (4\text{バイト整数})10 \tag{5.15}$$

$$(8\text{バイト整数})100.0\ \%\ (8\text{バイト整数})30.0 \rightarrow \text{エラー} \tag{5.16}$$

実数同士の剰余算をしたい場合は、C 言語では fmod 関数を利用するとエラーが起こりません。また、HSP の最新バージョンでは、実数を気にせずに剰余算を行えます。

*28 infinity:無限大
*29 Not a Number:非数
*30 Hot Soup Processor:おにたまさんが開発したプログラミング言語

6 ビット演算とシフト演算について

ビット演算とシフト演算について話します。両方の演算ともに、基本的に整数に対して使う演算方法です。実数には基本通用しないので使用しません。*31 これらは、前回話したように、数値が内部ではビットで表現していることに着目して、演算する方法です。

まず先にビット演算について説明します。ビット演算には、**表 6.1**〜**表 6.6** のようなものがあります。それぞれ、１つのビットごとに着眼して、演算を行うのがビット演算です。論理積と論理和は、漢字のとおり、掛け算 (積) と足し算 (和) みたいな演算です。

表6.1 ビット演算の種類

名前	演算子	意味
ビット論理積	&	ビットごとの論理積 (AND)
ビット論理和	|	ビットごとの論理和 (OR)
ビット排他的論理和	^	ビットごとの排他的論理和 (XOR)
ビット反転	~	ビットごとに 1 と 0 を反転させる。

表6.2 ビット演算の演算結果

計算例	計算例の結果	計算の効果 (1 ビットごとに着眼する)
0b10101010&0b11110000	0b10100000	両方とも 1 なら 1
0b10101010|0b11110000	0b11111010	どちらか 1 なら 1
0b10101010^0b11110000	0b01011010	2 つとも違うなら 1
~0b10101010	0b10101010	1 と 0 を反転させる

表6.3 論理積

A	B	A&B
0	0	0
0	1	0
1	0	0
1	1	1

表6.4 論理和

A	B	A|B
0	0	0
0	1	1
1	0	1
1	1	1

表6.5 排他的論理和

A	B	A^B
0	0	0
0	1	1
1	0	1
1	1	0

表6.6 ビット反転

A	~A
0	1
1	0

「ビット演算なんて、使い道があまりないじゃないか」と思う方も多いと思いますが、使い道は後で説明するので、シフト演算の説明に行きたいと思います。シフト演算は**表 6.7** のこれだけです。ビットシフトは**式 6.1**〜**式 6.5** のように、内部のビットを指定の方向へずらすということです。

$$0b00000001 << 1 \rightarrow 0b00000010 \quad (6.1)$$

$$0b00000001 << 2 \rightarrow 0b00000100 \quad (6.2)$$

*31 ビット演算に関しては、真偽値にも利用できることが一般的ですが、これについては後ほど説明します。

表6.7 シフト演算の種類

名前	演算子	意味
左シフト	<<	左へビットシフト
右シフト	>>	右へビットシフト

$$\text{0b00000100>>1} \rightarrow \text{0b00000010} \tag{6.3}$$

$$\text{0b00000100>>2} \rightarrow \text{0b00000001} \tag{6.4}$$

$$\text{0b00000100>>3} \rightarrow \text{0b00000000} \tag{6.5}$$

ここまでのビットシフトの説明を見て、普通はあんまり使い道がないと考える方が多いと思います。ここで、10 進数にしてみると**式 6.6**〜**式 6.10** のようになっています。

$$(\text{0b00000001<<1} \rightarrow \text{0b00000010}) \rightarrow (\text{1<<1} \rightarrow 2) \tag{6.6}$$

$$(\text{0b00000001<<2} \rightarrow \text{0b00000100}) \rightarrow (\text{1<<2} \rightarrow 4) \tag{6.7}$$

$$(\text{0b00000100>>1} \rightarrow \text{0b00000010}) \rightarrow (\text{4>>1} \rightarrow 2) \tag{6.8}$$

$$(\text{0b00000100>>2} \rightarrow \text{0b00000001}) \rightarrow (\text{4>>2} \rightarrow 1) \tag{6.9}$$

$$(\text{0b00000100>>3} \rightarrow \text{0b00000000}) \rightarrow (\text{4>>3} \rightarrow 0) \tag{6.10}$$

これを見て、何か気が付きませんか。実は左へビットシフトは、2 倍。右へビットシフトは、2 で割るのと同じなのです。[*32]2 進数だからこのような結果になるのです。例えば、よく使用している 10 進数で、1 を左にビットシフトした場合 10 になり、10 倍になりますよね。それと同じです。

じゃあ、「2 で掛けたり、2 で割ったりしたほうが分かりやすいじゃん」と思いになるかもしれませんが、こちらのビットシフトを利用した計算の方が計算速度がはやいのです……。というのは、本当ですが冗談です。本当に便利な点としては、ビット演算と関連しているため、これらの処理と組み合わせる時は、シフト演算のほうが、見たときに判りやすいという利点があります。普通に 2 倍、4 倍、8 倍などしたいときなどは、シフト演算でもいいのですが、見たときの分かりやすさから、素直に掛け算したほうがいい場合もあります。本当に「速度を求めたい」という時は、シフト演算を利用してください。速度を重視することは、そんなにはないとは思いますが。というのも、普通の掛け算・割り算でも、今の PC では速いので、体感的に分からないからです。ゲームとか、そういう速度優先でも、シフト演算は使用しない方がいいです。もっと低レベルなところで速度を速くしたいときに利用しましょう。(>_<)

ビットシフトの利用にも注意点があります。それは負の数字が入った場合です。負の数字を表すには、一番左のビットを、符号の目印にすると数値の章で勉強しましたね。これが問題になってくるのです。例えば、**式 6.11** と**式 6.12** は、どちらが正しいでしょうか。普通に考えれば、符号の目印の 1 が右にずれるため、**式 6.11** が正解だと思います。ですが、**式 6.12** が実は正解です。これ

[*32] 2 つビットシフトした場合は 4 倍、3 つビットシフトした場合は 8 倍。

は、多分言語ごとに異なってくると思いますが、**式 6.12** の結果になる言語が一般的なようです。心配なら一度試してみたり、調べたりすることが大事です。**式 6.11** の結果になってほしい場合は「>>>」を利用することで、行えるような言語[*33]もあります。

$$-128>>1 \rightarrow 0b10000000>>1 \rightarrow 0b01000000 \rightarrow 64 \tag{6.11}$$

$$-128>>1 \rightarrow 0b10000000>>1 \rightarrow 0b11000000 \rightarrow -64 \tag{6.12}$$

速度とは別に、これらのビット演算・シフト演算を組み合わせれば、バイトではなく、ビットを意識することができます。そのため、のように 1 バイトの数値にいくつも情報をうめることができます。左側の数値の N ビット目を 1 にする例が**式 6.13**。左側の A の 4 ビットの情報のみを抜き出す例が**式 6.14**。左側の B の 4 ビットの情報のみを抜き出す例が**式 6.15**。左側の G の 16 ビットの情報のみを抜き出す例が**式 6.16** です。

$$0b00000000|(1<<N) \tag{6.13}$$

$$(0bAAAABBBB>>4)\&0b00001111 \tag{6.14}$$

$$0bAAAABBBB\&0b00001111 \tag{6.15}$$

$$(0xAARRGGBB>>8)\&0xFF \tag{6.16}$$

「&」で他の情報を消すことを、「ビットマスク」と呼びます。マスクするわけです。この時「>>」と「&」の順番を間違えないでください。なぜ、順番を間違えたらダメなのでしょうか。これは、先ほど説明した、負の整数の場合を考慮しなくてはいけないからです。(+o+) **式 6.17** は、左側のビットが残ってしまい負の値になってしまいます。**式 6.18** のように、順番を考えれば、正しく情報をマスクすることができます。

$$(0bAAAABBBB\&0b11110000)>>4 \rightarrow 0bAAAA0000>>4 \rightarrow 0bA000AAAA \tag{6.17}$$

$$(0bAAAABBBB>>4)\&0b00001111 \rightarrow 0b0000AAAA\&0b00001111 \rightarrow 0b0000AAAA \tag{6.18}$$

式 6.14 と**式 6.15** で取り出した A の値と B の値を元のビットの位置に戻す場合は、**式 6.19** のように、逆の手順で「<<」と「|」を使えば戻すことができます。

$$(0b0000AAAA<<4)|(0b0000BBBB) \tag{6.19}$$

排他的論理和（XOR）の**式 6.20** の特性を利用すれば、暗号化にも使えます。0b00001111 を、0b10010100 を鍵にして暗号化するのが、**式 6.21** です。**式 6.22** のように、暗号化した値を、同じ鍵で排他的論理和することで、複合化ができます。

$$A\hat{}B\hat{}B = A \tag{6.20}$$

[*33] Java では >>> が使用できます。

$$0b00001111\hat{}0b10010100 = 0b10011011 \tag{6.21}$$

$$0b10011011\hat{}0b10010100 = 0b00001111 \tag{6.22}$$

ここでのポイントは、ビット演算・シフト演算は、四則演算と違って、整数に対して行う演算であり、ビットを意識した演算に利用する演算方法ということです。まだまだ、演算関係の話は続きます。がんばってください。**(・ω・)ノ**

7 関係演算と論理演算について

ここで学ぶのは、**結果が全て真偽値になる演算**です。まず関係演算子（**表 7.1**）の話から。関係演算子は、2 つのものを比較する演算です。そして、その演算の結果は、必ず真偽値になります。また、イコールとノットイコール以外は、a と b ともに数値ではないといけないです。真は偽より大きいなんて比較できませんよね。**式 7.1** は、**真偽値同士と数値同士のみ**に使用可能な関係演算の例です。**式 7.2** は、**数値同士のみ**に使用可能な関係演算の例です。

表7.1 関係演算の種類

名前	演算子と例	意味
イコール	$A == B$	A と B は同じ値であるか。
ノットイコール	$A! = B$	A と B は違う値であるか。
大なり（より大きい）	$a > b$	a のほうが b より大きいか。
小なり（未満）	$a < b$	a のほうが b より小さいか。
大なりイコール・以上	$a >= b$	a は b 以上か。
小なりイコール・以下	$a <= b$	a は b 以下か。

$$\begin{aligned} \boldsymbol{false} == \boldsymbol{false} \rightarrow \boldsymbol{true}, \boldsymbol{true}! = \boldsymbol{true} \rightarrow \boldsymbol{false} \\ 100 == 100 \rightarrow \boldsymbol{true}, 100! = 100 \rightarrow \boldsymbol{false} \end{aligned} \tag{7.1}$$

$$\begin{aligned} 100 > 10 \rightarrow \boldsymbol{true}, 10 < 100 \rightarrow \boldsymbol{false} \\ 100 \geq 100 \rightarrow \boldsymbol{true}, 100 \leq 100 \rightarrow \boldsymbol{true} \end{aligned} \tag{7.2}$$

C 言語の場合は、true、false の代わりに、1 と 0 になります。C++ 言語でも互換性のためこうなります。この場合、**式 7.1** が**式 7.3** に、**式 7.2** が**式 7.4** になります。

$$\begin{aligned} 0 == 0 \rightarrow 1, 1! = 1 \rightarrow 0 \\ 100 == 100 \rightarrow 1, 100! = 100 \rightarrow 0 \end{aligned} \tag{7.3}$$

$$\begin{aligned} 100 > 10 \rightarrow 1, 10 < 100 \rightarrow 0 \\ 100 \geq 100 \rightarrow 1, 100 \leq 100 \rightarrow 1 \end{aligned} \tag{7.4}$$

論理演算について説明します。論理演算は、**表 7.2** のようなものがあります。ビット論理演算によく似ていますが違うので注意してください。これら、**A と B には、真偽値しか入力することが出来ません。**真偽値同士から論理演算して、結果を真偽値で返すものです。ただし、C 言語の場合は、真偽値が数値で表されるため、**式 7.5** のような演算も利用できます。

表7.2　論理演算の種類

名前	演算子と例	意味
論理和	*A&&B*	A と B は両方、true であるか。
論理積	*A*‖*B*	A と B は、少なくともどちらか一方が true であるか。
否定	!*A*	A は true ではないか。

$$1\&\&1 \rightarrow 1, 1\&\&0 \rightarrow 0, !1 \rightarrow 0, !0 \rightarrow 1 \tag{7.5}$$

ところで、論理演算は、**ショートサーキット**になる場合が多いです。ショートサーキットとは、**式 7.6** のように、式の途中で結果が分かった場合に、評価を打ち切ります。もし、ビット論理演算子を利用した場合は、**式 7.7** のように、1 つ 1 つ計算されます。ショートサーキット演算子は、演算の速度を上げるために有効な演算子ですが、論理演算する順番によって、結果が変わってしまうようなプログラムを書くと、バグの元になるので注意しましょう。

$$\boldsymbol{true}\|\boldsymbol{false}\|\boldsymbol{true} \rightarrow [\boldsymbol{true}]\|\boldsymbol{false}(\text{評価しない})\|\boldsymbol{true}(\text{評価しない}) \rightarrow \boldsymbol{true} \tag{7.6}$$

$$\boldsymbol{true}|\boldsymbol{false}|\boldsymbol{true} \rightarrow (\boldsymbol{true}|\boldsymbol{false})\,|\boldsymbol{true} \rightarrow (\boldsymbol{true})\,|\boldsymbol{true} \rightarrow \boldsymbol{true} \tag{7.7}$$

基本的には、数値同士にはビット論理演算[*34]、また真偽値同士には論理演算[*35]と使い分けをすると、分かりやすいプログラムになるので、これらの違いについてしっかりと理解しておいてください。

8　条件演算子について

最後の演算子の紹介として、条件演算子（三項演算子）について説明します。演算子といっても、**式 8.1** のように、構文のような使い方をします。**式 8.1** では、A が true なら a、A が false なら b というものです。つまり、A の真偽によって、a か b の 2 種類の値に分かれるわけです。

$$A?a : b \tag{8.1}$$

[*34] 「&」と「|」。
[*35] 「&&」と「||」。

式 8.2 は、a と b に数値を利用した例です。**式 8.3** は、a と b に真偽値を利用した例です。**式 8.4** は、a と b に文字列を利用した例です。このように、**式 8.1** の A に真偽値さえくれば、a と b にはどんな値でも大丈夫です。

$$\begin{aligned} \textbf{\textit{true}}?100 : 20 \rightarrow 100 \\ \textbf{\textit{false}}?100 : 20 \rightarrow 20 \end{aligned} \tag{8.2}$$

$$\begin{aligned} \textbf{\textit{true}}?\textbf{\textit{false}} : \textbf{\textit{true}} \rightarrow \textbf{\textit{false}} \\ \textbf{\textit{false}}?\textbf{\textit{false}} : \textbf{\textit{true}} \rightarrow \textbf{\textit{true}} \end{aligned} \tag{8.3}$$

$$\begin{aligned} \textbf{\textit{true}}?\text{"真です"} : \text{"偽です"} \rightarrow \text{"真です"} \\ \textbf{\textit{false}}?\text{"真です"} : \text{"偽です"} \rightarrow \text{"偽です"} \end{aligned} \tag{8.4}$$

式 8.1 の A には真偽値が入ればいいので、真偽値の結果が出る関係演算や論理演算などでも大丈夫です。よって、四則演算と関係演算を組み合わせて、N の値が偶数か奇数かで値を変える**式 8.5**。関係演算と論理演算を組み合わせて、N の値がどの範囲に含まれているかで値を変える**式 8.6** のようなことなどもできます。

$$\begin{aligned} ((N\%2) == 0)? \\ \text{"}N\text{ は偶数です"} : \text{"}N\text{ は奇数です"} \end{aligned} \tag{8.5}$$

$$\begin{aligned} ((0 \le N)\,\&\&\,(N \le 100))? \\ \text{"}N\text{ の値は 0〜100 までのどれかの数値です。"} : \text{"(空白)"} \end{aligned} \tag{8.6}$$

条件演算子を、2 つ組み合わせれば、**式 8.7** のような式も作れます。**式 8.7** はどんな意味か分かりますか。これは「N が 10 未満なら 10 に。N が 100 より大きい場合は 100 に。その間は数値 N のまま通す」というような数式です。つまり、N は必ず 10〜100 までの間になるわけです。ですが、正直どのようになっているか、分かりづらいですよね。条件演算子は、とても強力で便利な反面、**分かりづらい数式になってしまいがち**です。よって、if などの制御文*36や、コメントを入れることが大事です。

$$N < 10?N > 100?100 : N \tag{8.7}$$

C 言語の場合は、true が 1 になることを利用して、**式 8.7** を、条件演算子を利用せずに、**式 8.8** のように、ひねくれた感じで表記することもできます。

*36 後に説明します

$$
\begin{aligned}
&((N < 10) * 10)| \\
&((N \leq 10)\&(N \leq 100) * N)| \\
&((100 < N) * 100)
\end{aligned}
\tag{8.8}
$$

式 8.8 は、本当にどんな計算をしているのか分かりにくいですよね。正しい結果が出ますが、**分かりやすい数式を書くことが大事**です。「自分の趣味の範囲ならやってもいいじゃないか」と思いますが、これを書いて、半年や 1 年後に同じ数式を見て、何をしているか分かるでしょうか。保守性も考えるのが、プログラミングのコツなのです。1 度書いたコードは、再利用することが多いのですが、このように書くと何をやっているか分からなくなるので、再利用がしにくくなるのです。

これで、演算関係の話は終わりです。お疲れ様です。次から、変数などの話になります。

9 変数について

変数と聞いて何を思い浮かべますか。x とか y ですか。変数は、プログラミング上では、よく箱に例えられます。変数はメモリのある番地に名前をつけるようなもの……と言っても分かりづらいですね。何かに例えますか。箱にしますか。箱にしましょう。

変数とは箱です。箱には、先ほど勉強したように、『4 バイト整数が入る箱』『文字が入る箱』『8 バイト実数が入る箱』のように、**どのような種類の値を入れる箱かを決める必要**があります。また、その箱には、**リスト 1** のように**名前を付ける必要**もあります。

リスト 1　箱は「何が入るものか」と「箱に付ける名前」が必要です

```
1 4バイト整数の箱 variable1
2 8バイト実数の箱 variable2
3 文字列の箱 variable3
```

ところで、4 バイト整数・8 バイト実数・文字列など、今まで型の名前を日本語で説明していましたが、実際は、型の名前は英語で決まっています。プログラムをする場合は、その言語にあった、英語の型名を利用しなくてはいけません。しかし、今回説明の元としている、C・C++・C#・Java は大体良く似ているので、あまり覚え直す必要は無いです。**表 9.1** で紹介します。C の型は全て C++ でも利用できます。

C には文字列はなく、文字の配列（char[]）があります。配列については後で説明します。また載っていないものについては、その言語で基本的に利用できないものです。ところで整数・実数に関しては、種類がたくさんあるので、覚えるのが面倒だと思うかも知れません。実は、一応これらの言語では、**表 9.2** を利用すればうまくいきます。型のサイズに気にしない場合は、**表 9.2** を利用しましょう。

補足として、○バイト整数や、○バイト実数というのは、データの格納の方法であり、真偽値や文字というものは、使用目的によるものです。C では、データの格納の方法に関する型は存在するのですが、真偽値や文字列と言った概念的な型はありません。そのため C では、真偽値や文字列といった区別がないのです。C での文字列は、あくまで ASCII コードが収まるデータ型 (char) が、並んだものにすぎないのです。それが、データのバイナリを表すものとして利用されているか、文

表9.1　各言語で利用できる型の一覧

	C	C++	C#	Java
1 バイト整数	(signed) char	C と同じ	byte	byte
1 バイト自然数	unsigned char	C と同じ	ubyte	-
2 バイト整数	(signed) short	C と同じ	short	short
2 バイト自然数	unsigned short	C と同じ	ushort	-
4 バイト整数	(signed) long	C と同じ	int	int
4 バイト自然数	unsigned long	C と同じ	uint	-
8 バイト整数	(signed) long long	C と同じ	long	long
8 バイト自然数	unsigned long long	C と同じ	ulong	-
4 バイト実数	float	C と同じ	flaot	float
8 バイト実数	double	C と同じ	double	double
真偽値	- (int)	bool	bool	boolean
文字 (' で囲む)	char	C と同じ	char	char
文字列 ("で囲む)	- (char[])	std::string	String	String
型なし	void	C と同じ	void	void

表9.2　これだけは覚えておこう。各言語で利用できるデータ型の型名

整数	int
実数	double

字列として利用されているのか。あるいは、それが整数として利用されているのか、真偽値として利用されているのかが、外見からは区別がつきません。これらは、バグの原因になるものとして、C より高級言語（Java や C#）では区別されています。

さらに補足として、**表 9.2** には書いていませんが、C と C++ でも型名として int 型というのがあります。C と C++ での int 型は、一番計算するときに早い整数型が割り当てられたものです。これらは、CPU の計算ビットによって使用するバイト数が変わります。32 ビット動作の CPU なら、long（C と C++ では 32 ビット）と同じ長さになるわけです。ですので、**表 9.2** に書いてある int も C と C++ で使用できます。

これまでの説明をふまえると、**リスト 1** は、実際には**リスト 2** と書くということが分かります。

リスト 2　型名を実際にある言語にのっとって書いた場合 (Java)

```
1 int variable1
2 double variable2
3 String variable3
```

ここまでは変数の型について説明してきました。ここからは変数をどのように用いるか説明します。例を挙げたほうが早いので、先に例を挙げます。

変数の使用方法の基本として、代入文というものを覚えましょう。代入文は、その名の通り、値（数値以外にも文字やいろいろ）を変数に入れるために利用するものです。代入演算子は「=」です。数学では「:=」と書く演算子です。関係演算子の「等しい (==)」とは違うので注意してください。実際に代入する例を、**リスト 3** に示します。

リスト 3　代入演算子を利用する例 (Java)

```
int variable1 = 100
double variable2 = 3.14159265
String variable3 = "文字列"
```

リスト 3 では、variable1 の型は整数型なので、100 という小数を含まない値を代入しています。また、variable2 の型、実数型なので、小数を含む値を代入しています。variable3 の型は、文字列型なので、文字を代入しました。文字列型と文字配列とは、異なるものなのですが、分かりやすくするために、今回はこのようにしました。

では、**リスト 4** のように、もしも変数にその変数の型と異なる値を代入したらどうなるでしょうか。

リスト 4　異なる値を代入したらどうなるのかな

```
int variable1 = 6.3
double variable2 = 24
```

通常は、型とは合わない値を入れた場合はエラーを起こします。しかし、代入できないといっても、コンパイラが自動で、値を直してくれることがあります。例えば、**リスト 4** では、6.3 という値を 6 として、整数と考えてくれたり、24 という値を 24.0 として、実数と考えてくれたりするのです。自動変換してくれる例には、**リスト 5** のようなものがあります。但し、これらの暗黙の型変換は言語によって異なるので注意しましょう。バグを生み出す原因になります。

リスト 5　暗黙の型変換の例

```
short 型の整数 = long 型整数
// short 型整数の最大 < long 型整数の最大なので変換できない。危険。
long 型の整数 = short 型の整数
// long 型整数の最大 > short 型整数の最大なので変換できる。安全に変換可能。
float 型の実数 = double 型の実数
// float 型実数の表現力 < double 型実数の表現力。情報落ちする。
double 型の実数 = float 型の実数
// double 型実数の表現力 > float 型実数の表現力なので変換できる。安全に変換可能。
```

ここからは、変数の名前の付け方について説明していきます。**リスト 3** では、整数の変数の名前を「variable1」と実数の変数の名前を「variable2」と名づけをしました。variable は変数を英単語です。たしかに、変数なので、変数の名前としてはその通りです。ですが、自分が分かればいいような名付けや、その場限りの適当な名付けなどは、絶対に避けた方がいいです。

ある程度の変数の命名は決まっているので、それを踏まえて名前をつけましょう。これらは、**命名規則**と呼ばれており、一度それに従ったら、そのソースコード上では、その命名規則を守るというものです。それによって、ソースコードの書き方に統一性が生まれて、読むのが楽になるからです。

例えば、命名規則として、**変数名は小文字**というものです。変数名はコンパイラにもよるのですが、大文字と小文字を区別します。（SEISU や Seisu や seisu は別の変数として捕らえられる）変数名は大文字で書いたりしてもよいということになるのですが、定数（後で説明します）と変数を分けるためという時代風景からです。どういうことかというと、定数名として大文字、変数名として小文字を利用することで、目でソースコードを読んだときに、定数なのか変数なのか、どちらなのかが分かりやすくなるからです。

他には、ハンガリアン記法というのもあります。変数の名前の前に、その変数がおおよそ何を示すのかを略字で書くというものです。ハンガリアン記法には、2 種類あり「アプリケーション」レベルの命名と「システム」レベルの命名があります。「アプリケーション」というのは、アプリケーションレベル、つまりアプリケーションにおいて、この変数はどういう役割に位置づけるかというものの略称を加える方法です。例えば、画面の横幅なら、width だけではなく、scWidth というような感じです。「システム」というのは、型の名前の略称を変数に付けるというものです。整数型の横幅なら、iWidth というような感じです。前者は、分かりやすいといっちゃ分かりやすいのですが、それだったら、screenWidth とか略さないほうが、もっと分かりやすいと思います。また、後者については、コンパイラが型を把握しているので、メリットがありません。（型が分からない場合は別ですが、普通は変数を作るときに、型も書くので分かるはずです。）しかし後者は、実は昔マイクロソフトがこの命名を利用していました。サイズを表す 2 バイトの数値を dwSize などです。今は.NET にうつり、このシステムハンガリアンを禁止しています。ハンガリアン記法を使用する場合は、メリットデメリットをよく考えて使用してください。

最後に個人的なものになるのですが、変数名はローマ字とかよりは、英語の名前の方がかっこいいということです。和英辞典とかを利用するとよいです。最初のうちはつけ方が、迷うと思いますがそのうち慣れて、すぐに名前を付けられるようになります。文字を打つのが面倒なので、英単語を略すというのもありますが、あんまり略すと、なんという英単語なのか分からなくなるので、ほどほどにしましょう。特に関数名（機能の名前）は、昔は略す（To → 2、For → 4、Standard → std）のが流行っていたようですが、最近は、略さないほうが流行です。これは、コード全体のファイルサイズを気にするより、人間に分かりやすくが傾向となっているからです。今は、変数名に関しては、IDE（後で説明します）が補完してくれるので、長くても問題ありません。分かりやすければいいのです。Java の場合は、命名規則（Java 言語規定の名前の項）も存在するので、それを読みながらつけるといいです。

ところで、先ほど変数は「変数はメモリのある番地に名前をつけるようなものです」と言いました。しかし、やっぱり重要な概念なので、少し難しいですが説明したいと思います。ちょっとはしょって説明しますが、雰囲気だけはつかめたらいいなと思います。メモリというのは、聞いたことありますよね。記憶できる領域です。変数も、このメモリ上に記憶されています。この領域は、OS が管理しているので、プログラム側でメモリを利用したい場合は、OS に〇バイトのサイズの領域を欲しいと頼むわけです。例として、**リスト 6** のように変数を利用したいと宣言してみます。float は 4 バイトなので、4 バイト分の自由に使用できるメモリ領域を OS からもらいます。今回は、0x9a0320 のアドレスから 4 バイトもらえたようなので、**表 9.3** のように、ここから 4 バイト分に weight のデータが入ることになったようです。変数は、この先頭アドレスである、0x9a0320 に名前を付けたようなものなのです。0x9a0320 というのは決まった数字ではなく、OS やコンパイラが決めます。コンパイラ側で名前とメモリアドレスとの領域との関係を関連付けているので、使っている私たちにとっては、あんまり関係ないことになりますが、この概念が後で必要になります。また、この不明のアドレスにアクセスをしたり、何か書き込んでしまったりすると、エラーの原因になります。

リスト 6　weight という名前の実数型の変数を宣言

```
1 float weight = 9.8;
```

表9.3 確保された変数とそのメモリアドレスの関係

使用目的	不明		weight				不明	
メモリアドレス	…	0x9a031f	0x9a0320 先頭アドレス	0x9a0321	0x9a0322	0x9a0323	0x9a0324	…

変数は、計算式の途中でも使用できます。**リスト 7** を見てください。3 行目まで使用する変数の宣言と代入を行った後に、4 行目で、変数同士の掛け算をし、その結果を kotae に代入しています。kotae の変数には、1000 が代入されます。なお『;』は、式の終わりにつける目印です。

リスト 7 計算式での変数の利用

```
1 int x = 10;
2 int y = 100;
3 int kotae = 0;
4 kotae = x * y;
```

計算と代入は別々に行われます。そのため、**リスト 8** のように自らの変数を利用した計算をし、その結果をその変数に代入することも可能です。**リスト 9** のようなイメージを持つと分かりやすいです。

リスト 8 計算式で変数を利用した計算を行い、結果を使用した変数へ代入する

```
1 int i = 0;
2 i = i + 100;
```

リスト 9 **リスト 8** のイメージ

```
1 int i = 0;
2 結果 = i + 100;
3 i = 結果;
```

ある変数に対して『何かを足したい』『何かを掛けたい』というのはよくあるので、「=」以外に、**表 9.4** のようなそれ専用の代入演算子も用意されています。**表 9.4** を見て分かる通り、四則演算子の前に、『=』がついたものになります。ここには書いていないですが、このように演算子の前に『=』をつけることで、ビット演算や、シフト演算でも利用できます。

表9.4 いろいろな代入演算子の例

代入演算子	使い方	意味
+=	a += 100	a = a + 100
−=	a −= 100	a = a − 100
*=	a *= 100	a = a * 100
/=	a /= 100	a = a / 100
%=	a %= 100	a = a % 100

このように、代入演算子はたくさんあります。変数に何か足したり掛けたりするときに使ってみてくださいね。（＾＾）

ところで、ある変数に **1 を加算**、ある変数に **1 を減算**という計算はよく利用するので、これ専用の演算子として、**表 9.5** のような演算子があります。加算の方をインクリメント、減算の方をデクリメントと呼びます。

表9.5　インクリメントとデクリメント

演算子	使い方	意味
++	a++	a = a + 1
−−	a−−	a = a − 1

インクリメントとデクリメントは、**表 9.6** のように、変数の後ろにつけるか前に付けるかで、効果が異なります。式の途中に、インクリメントとデクリメントを行うような書き方は、バグを生み出す原因になるので、おすすめしません。計算がややこしいので、特に長い計算式の途中に、この演算子を利用するのは危険です。もし、**先に計算か後に計算かで内容が異なるような式**に使用する場合は、コメントは必ず付けたほうがいいです。

表9.6　インクリメントとデクリメントの注意点

使い方	意味
a = i++	a = i; i++;
a = ++i	i++; a = i;

変数の最後として、配列変数について説明します。配列とは、同じ型（タイプ）の１つの変数が、ずらっと並んだものです。言葉の説明じゃ分かりづらいと思うので、使用する例を**リスト 10** に示します。

リスト 10　配列変数の利用例

```
1  // int x の 3つ分の配列変数を作る
2  //（宣言の仕方は言語によって異なる）
3  x[0] = 130;
4  x[1] = 2333;
5  x[2] = -123;
6  x[3] = 23;  //←エラー　4つめなので使用できない
```

配列変数の宣言方法（配列変数を使うよとコンパイラに教えること）は、言語ごとに異なります。1 度宣言すれば、変数名と添え字の番号の 2 つを書くことで、利用できます。注意する点は、添え字の番号は **0 から始める**ということです。3 つ分作った場合、1〜3 まで使えると普通は思いますが、このように実際は 0 から番号が始まるので、0〜2 までということになります。最初のうちは混乱すると思いますが、慣れるとこれが当たり前な感覚になるので、心配しないでください。

今回、**リスト 10** の例では、int 型の配列変数を利用しています。もちろん、文字の配列型や実数の配列型など、どのようなデータ型でも配列として宣言することで利用できます。

さて、配列変数を使用した場合のメモリアドレスの関係も紹介します。確保された領域が、あちこちばらばらになるのか、数珠のように繋がって確保されるのか、どのように、メモリが確保されるのでしょうか。正解は後者で、**表 9.7** のような感じです。『こんな内部的なこと覚えても意味ないんじゃない』と思うかも知れませんが、これが後先役立ちます。ポイントは、配列変数がメモリ

上では、変数同士が隣あっているということです。これは、先頭アドレスと、その変数がいくつ繋がっているか分かれば、メモリが確保されている領域が分かることを示してます。

表9.7 確保された配列変数とそのメモリアドレスの関係

使用目的	不明		x[0]				x[1]	
メモリアドレス	…	0x9a031f	0x9a0320 先頭	0x9a0321	0x9a0322	0x9a0323	0x9a0324	0x9a0325
使用目的	x[1]		x[2]				不明	
メモリアドレス	0x9a0326	0x9a0327	0x9a0328	0x9a0329	0x9a032a	0x9a032b	0x9a032c	…

配列変数は、同じ変数名で、添え字の数字を変えるだけで、たくさんの値を格納することができるのです。では、一般的にはどのように使用されることが多いのでしょうか。例えば、画面のピクセルなどは、配列変数で考えるとわかりやすいと思います。画面が「1280 × 1024」のディスプレイで、RGB（赤と緑と青）を 4byte の整数に格納するとするとします。pixel という変数名にして、1280 × 1024 個分の配列を確保すればいいのです。他にも、シューティングゲームなどで、弾を管理したいときは、最大 200 発画面に登場させるとして、200 個分のメモリを配列として確保すればいいのです。いちいち、200 個分に名前をつけて、それぞれ１つの変数として管理するのは、大変ですよね。

最初のうちは、中々使う場面を思いつかないかも知れませんが、書いていくうちに、必ずプログラムで使うようになります。それぐらい配列変数は重要なのです。

索引

記号／数字

! 20
!= 19
" 12
+ 14
+= 26
++ 27
− 4, 14
−= 26
−− 27
/ 14
/= 26
:= 23
; 26
< 19
<< 17
<= 19
= 23
== 19
> 19
>= 19
>> 17
>>> 18
’ 12
? 20
¥ 7
% 15
* 14
\ 7
| 16
|| 20
&& 20
& 16
ˆ 16
˜ 16
¥ 15
%= 26
*= 26
¥¥ 10
¥0 10
¥n 10
¥t 10
10 進数 3
16 進数 3
1970 年 1 月 1 日 0 時 0 分 0 秒 5
2038 年問題 5
2 進数 3
64 ビットバージョン 5
8 ビット JIS 10

A

AND 16
ASCII 7

C

CPU 4
CR 9
CR+LF 9

D

decimal 6

F

false 13

I

Inf 15

J

JIS8 10
JVM 2

L

LF 9
long double 4

M

mod 15

N

NaN 15
.NET 2
NUL 文字 9

O

OR 16

S

Shift-JIS 10

T

true 13

U

Unicode 12
UNIX 時間 5
UTF-16 12

X

XOR 16

あ

アセンブリ言語 1
アドレス 25
余り 15
暗号化 18
暗黙の型変換 24
以下 19
イコール 19
以上 19
インクリメント 27
英数字 11
エスケープ文字 10, 11
円マーク 7
大文字 7

か

改行 9
鍵 18
掛け算 14
加算 14
仮想マシン 2

型名の一覧 23
ガベージコレクション 2
関係演算子 19
漢字 10
偽 13
機械語 1
切り捨て 15
計算 13
減算 14
固定少数点 3
小文字 8
コンパイラ 1

さ

サロゲートペア 12
三項演算子 20
自然数 3
四則演算 13
実行環境 2
実数 3, 5, 14
シフト演算 16
条件演算子 20
小なり 19
小なりイコール 19
情報落ち 5
剰余算 15
ショートサーキット 20
除算のエラー 15
真 13
真偽型 13
真偽値 13
シングルクォーテーション 12
真理値 13
数式 13
数値 3
数値の文字コード 8
制御文字 9
整数 3, 14
先頭アドレス 28
添字 27

た

大なり 19
大なりイコール 19
代入 23
代入演算子 23, 26
タブ文字 10
ダブルクォーテーション 12
デクリメント 27

な

2 進数 3
日本語 10
ノイマン型 1
ノットイコール 19

は

倍数 15
排他的論理和 18
バイト 3
バイナリ 4
配列変数 27
バックスラッシュ 7
半角カタカナ 10
ハンガリアン記法 25
比較 19
非数 15
左シフト 17
ビット 3
ビット演算 16
ビットシフト 16
ビット排他的論理和 16
ビット反転 16
ビットマスク 18
ビット論理積 16
ビット論理和 16
否定 20
ひらがな 10
複合化 18
負数 4
浮動小数点 5
プログラミング 1
変数の名前の付け方 24

ま

右シフト 17
未満 19
無限大 15
明示的 14
命名規則 24
命令型プログラミング 1
メモリリーク 2
文字型 11
文字コード 7
文字化け 11
文字列 7

や

有効桁数 5
優先度 13
より大きい 19

ら

論理演算 20
論理積 20
論理和 20

わ

割り算 14