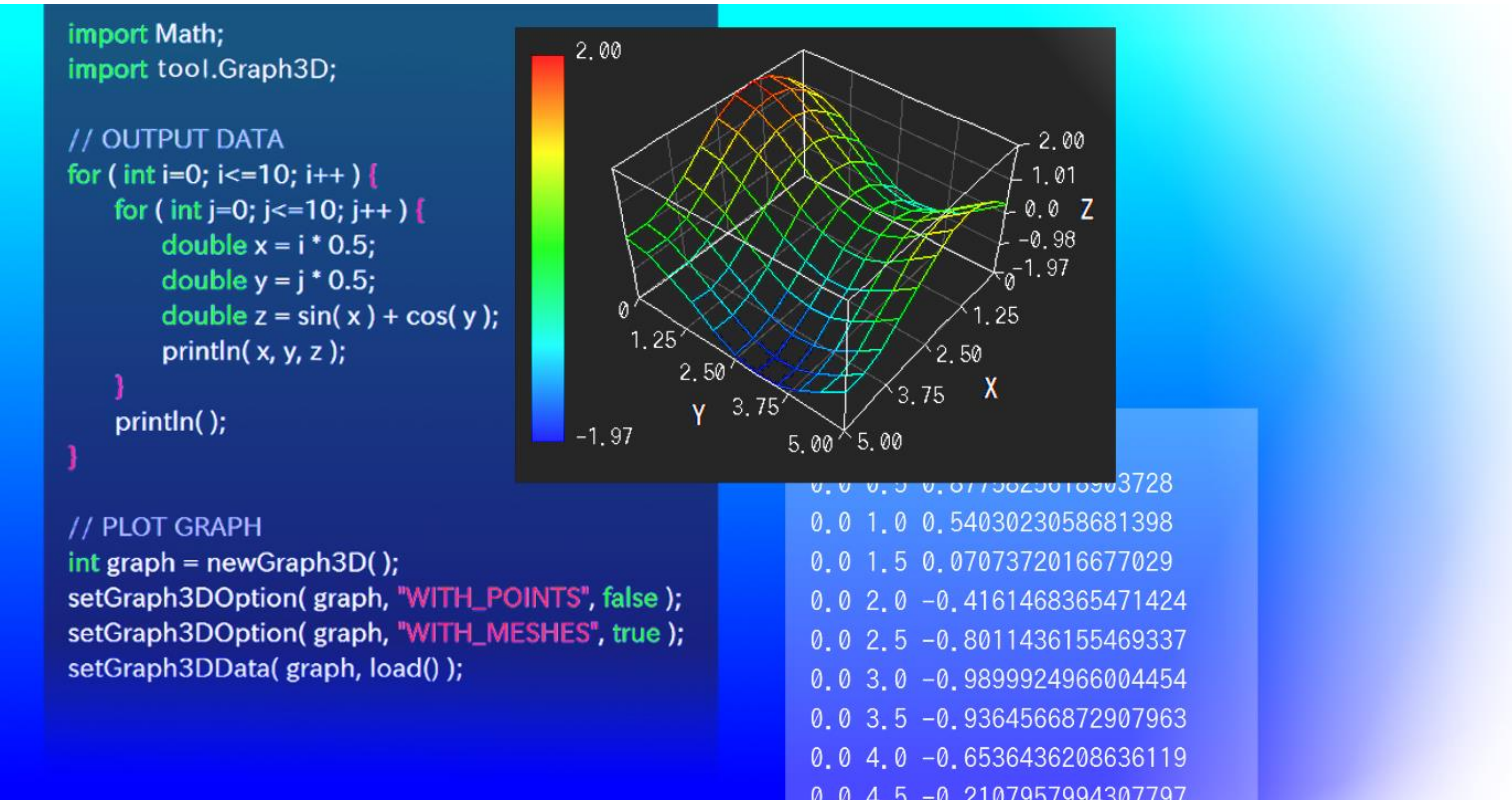
RINEARN
www.rinearn.com



プログラミング言語 VCSSL
スタートアップガイド

第 3 版

はじめに

プログラミングは、コンピューターを自在に制御して、様々な処理を行わせる事ができる、強力な手段です。わたしたちが日常で利用している、ソフトやアプリのいろいろな機能も、すべて誰かがプログラミングして作ったものです。

自分の手でプログラミングをすれば、必要な機能をもつソフトを自作したり、面倒な作業をある程度自動化できたりと、コンピューターをより深く効率的に活用できます。ゲームを作ってみるのも楽しいかもしれません。さらに、膨大な処理量の数値計算やデータ解析、科学技術分野でのシミュレーションなどにも、プログラミングは欠かせないものです。そしてなによりも、自分でなにかを設計して作るのが好きな人にとっては、最高の趣味としても楽しめるのが、プログラミングです。

この文書は、プログラミングがはじめての方を対象とした、VCSSL(ブイシーエス・エスエル)というプログラミング言語の入門ガイドです。単純に VCSSL の扱い方を解説する事よりもむしろ、VCSSL を足がかりとして、プログラミングそのものの感覚を伝える事に重点を置いています。

VCSSL は、C 言語に似た文法をもっている、とてもシンプルなプログラミング言語です。本文書で扱う内容は、プログラミングの基礎知識として特に重要な範囲にしぼっているので、あとで同じように C 言語風の文法の言語(もちろん C 言語も含みます)へとステップアップするときにも、きっと役立つでしょう。

ところで、プログラミング言語は、なんといっても「 コンピューターを使うために作られた言葉 」です。後々で泥沼にはまらないためには、プログラムの書き方だけでなく、コンピューターのしくみについても、一緒に理解していく事が大切です。このガイドでも、そのような解説をところどころで交えながら、できるだけ本質的な、ごまかしの無い説明を行う事を目指しました。

なお、このガイドは上でも述べた通り、完全にプログラミングがはじめてという方を想定しています。そのため、すでに他のプログラミング言語を習得されている方にとっては、VCSSL のガイドとしては内容が冗長かもしれません。その場合は同梱の「プログラミング言語 VCSSL リファレンスガイド」や、C/C++などを習得されている方には「C 系言語ユーザー向け 即席 VCSSL ガイド」をおすすめします。

それでは、一緒に楽しいプログラミングをはじめましょう!

目次

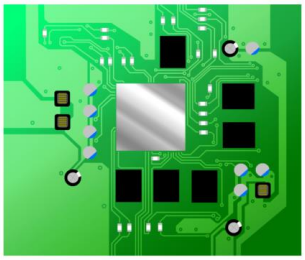
コンピューターとプログラム、プログラミング言語

■ 色々な処理を行える「コンピューター」

現代に生きる私たちにとって、コンピューターはとても身近で、便利で、無くてはならない存在です。コンピューターといっても PC、いわゆるパソコンだけではありません。スマートフォンやゲーム機だって、構造的にはれっきとしたコンピューターと呼べるものです。他にも、色々な家電製品の中にだって（最近は炊飯器や洗濯機の中にも）、よく小さなコンピューターが入っています。

PC、パソコン
（パーソナルコンピューター）

スマートフォン、
ゲーム機



家電製品に入っている
小さなコンピューター

それでは、コンピューターは一体なぜ便利なのでしょうか？ なぜ、こんなにも色々な所で使われているのでしょうか？

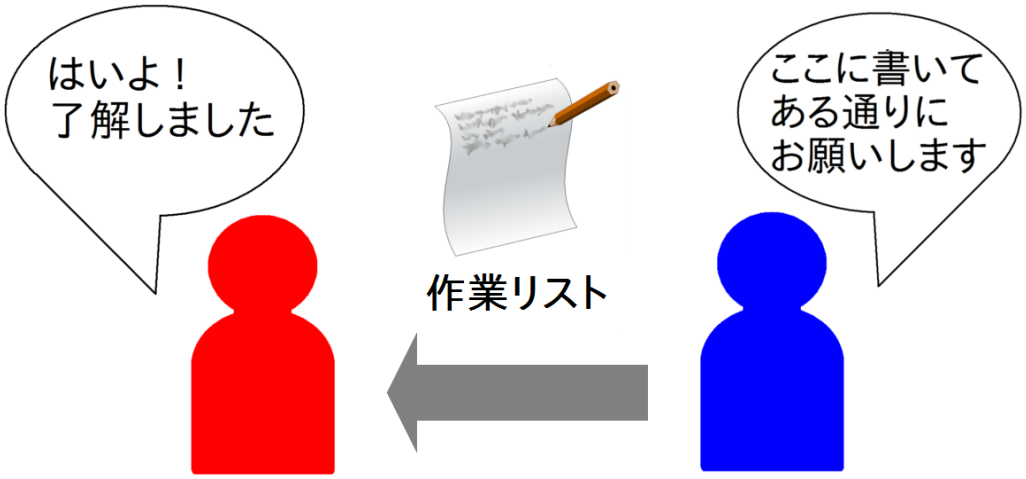
それは、「コンピューターが色々な処理をできるから」に他なりません。たとえばコンピューターは、複雑な計算を行う事も、文章を書くことも、デジカメで撮った写真を編集する事も、面白いゲームをプレイする事も、特殊な機器（デバイス）を制御する事だってできます。実際にみなさんの持っている PC は、基本的にこれらの事を全部できます。

これって、道具としてはすごく特別な事だと思いませんか？ 普通の道具は、たとえばテレビや音楽プレイヤーなど、何か決まった目的や機能を持っています。でもコンピューターは、特定の何ができるというよりも、「色々できる」という事が便利なわけです。

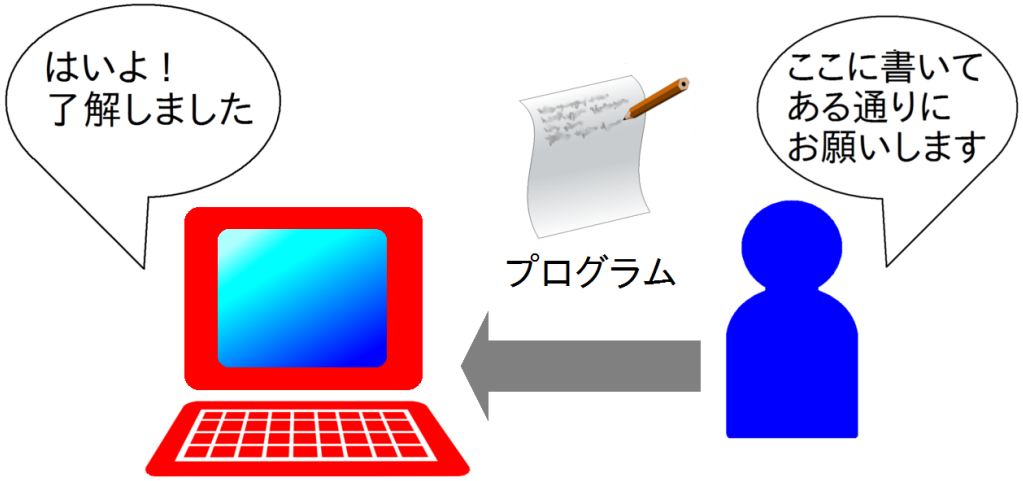
■ コンピューターに処理内容を伝える「プログラム」

さて、色々な処理をできるコンピューターですが、何も指示されなければ、何もしません。 そこで、どんな処理をしてほしいかをコンピューターに伝えるものが、「**プログラム**」です。

たとえば、あなたが他の人に、何か複雑な作業を頼むとき、どうするでしょうか？ 恐らく、作業手順のリストを紙などに書いて、作業する人に渡すのではないでしょうか。



これと全く同じで、コンピューターに何か複雑な作業（処理）を頼みたければ、プログラムに書いて渡せばよいわけです。「プログラム」というと、なんだか専門的な用語で難しそうなイメージがありますが、 要するに「 **プログラム ＝ コンピューターにとっての作業リスト** 」なのです。 コンピューターは、プログラムを渡されると、そこに書かれた通りの内容を、忠実に実行してくれます。



一般に、プログラムを書く作業を「**プログラミング**」や「コーディング」などと呼び、 書く人を「プログラマー」などと呼びます（仕事としての肩書きには色々なものがあります）。

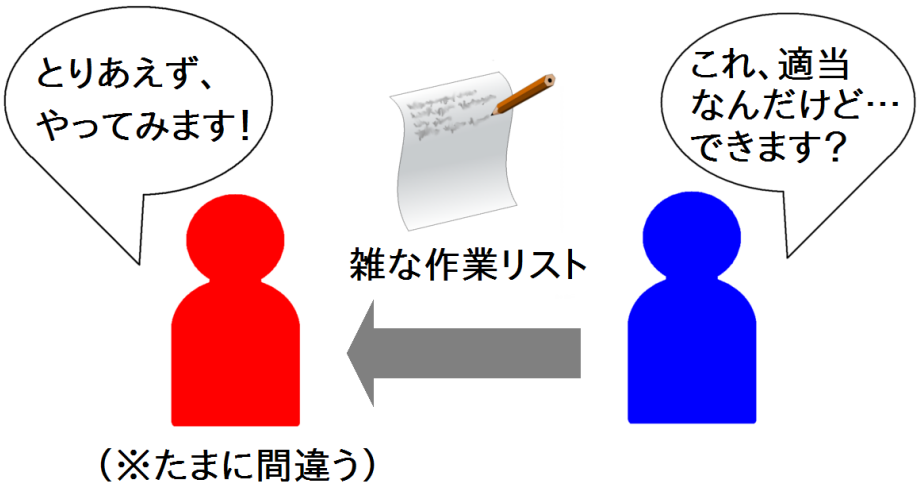
■ 作業（処理）における、コンピューターと人間の大きな違い

ところで、上の例でも触れた通り、リストに書かれた通りの事を行うだけなら、人間にだってできますよね。なぜわざわざ、コンピューターの使い方やプログラムの書き方を覚えてまで、コンピューターに頼むのでしょうか？

実は、**何か作業（処理）を行うにあたって、コンピューターと人間には大きな違いがあり、それぞれに得意・不得意がある**のです。なので、プログラミングを始める前に、この「違い」を理解しておく事がとても重要です。人間に作業を頼むのと全く同じ感覚で、コンピューターに頼もうとすると、融通の利かなさや細かさなどにイライラしてしまう事でしょう。でも、相手の得意・不得意を理解しようという心構えが最初からあれば、意外と腹は立たないものです。

■ コンピューターは、自分でどうすべきか考えられない。そのため細かい指示が必要だが、その代わり動作は正確！

それでは、具体的に人間とコンピューターの違いを見ていきましょう。まず人間は、**一から十から百まで細かく指示されなくても、ある程度は自分でどうすべきか考えながら、作業を進める事ができます。**場合によっては、「 例の件、適当にやっておいてくれ 」といった指示でさえも、ちゃんとこなす事ができます。

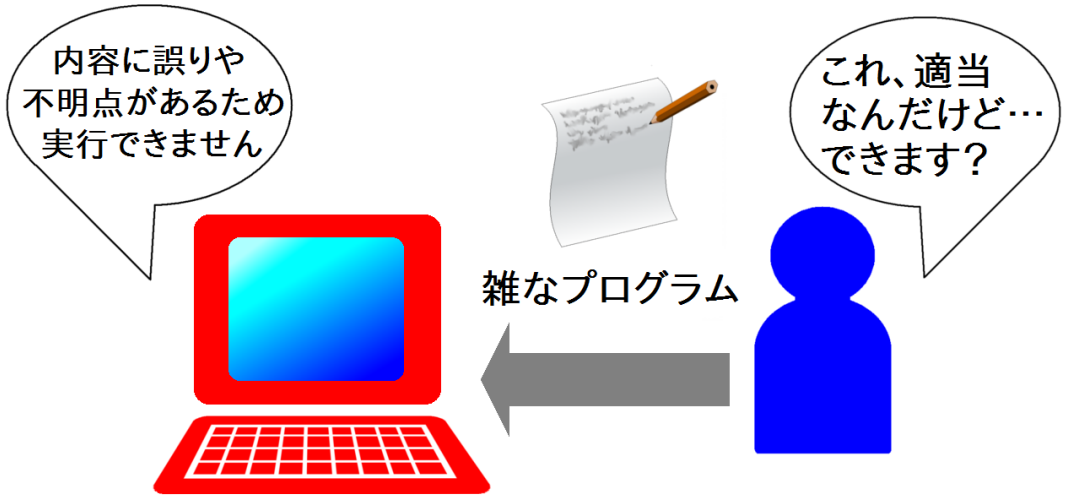


このように、**人間は常にある程度考えながら作業をしている**ため、未知の問題が発生しても、その場その場で適当に対処できます。これは人間の大きな強みです。その一方で、**考えた結果として間違った事をしてしまったり、**単純にケアレスミスをしてしまったり、そもそも作業内容を勘違いしてしまう事もあります。

それでは、コンピューターはどうでしょうか。コンピューターは人間と違い、**一手一手どうすべきか考えながら作業(処理)するのは苦手です(※)。** 原則として、プログラムに書かれた通りの内容を、忠実に実行する事しかできません。

(※ 近年では、機械学習などの人工知能技術も普及してきていますが、ここではより原理的な、コンピューターの基本動作の話に割り切っています。)

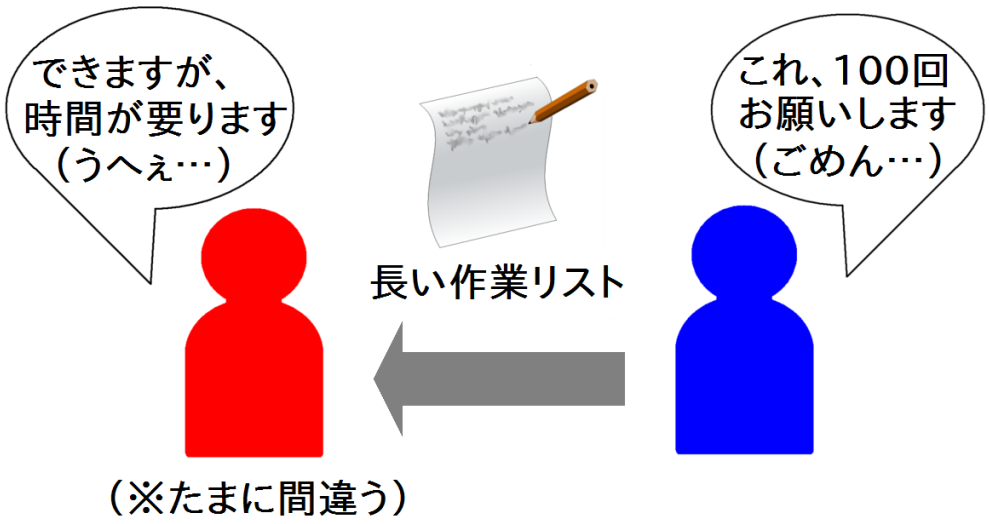
なので**プログラムには、すべての手順を、誰が読んでも考える余地が無いくらいに、細かく明確に書いてあげる必要がある**のです。 色々と省いたり記述ミスがあるような、雑なプログラムを渡しても、最初から実行してもらえないか、途中で止まってしまいます。どうすべきか考える事ができないからです。



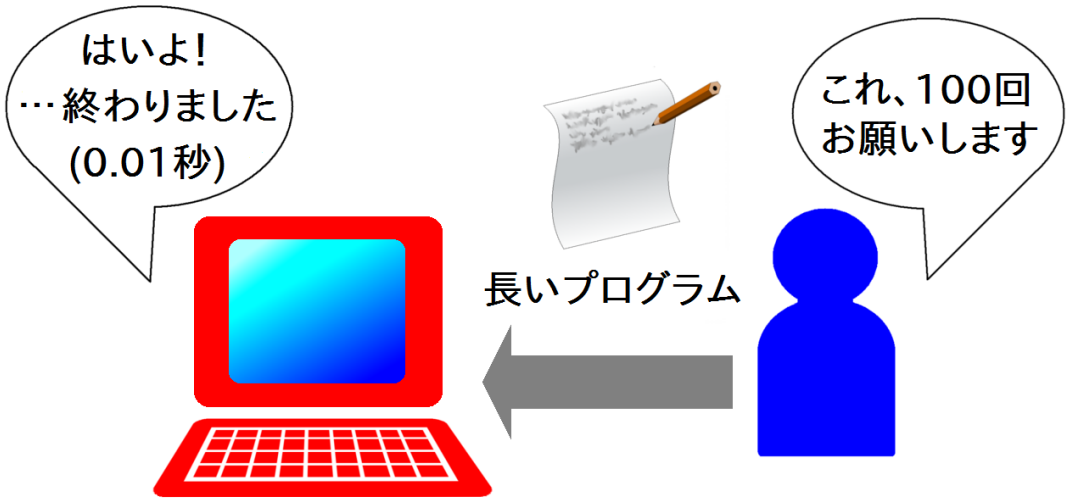
その代わり、ちゃんとコンピューターに十分わかるようにプログラムを書いてあげれば、 その通りに**非常に正確に実行**してくれます。 想定外のトラブルや不具合などを除けば、原則として間違える事はありません。 この「正確さ」が、コンピューターの強みの一つです。

■ コンピューターの動作は超高速！ 膨大な回数の繰り返しも余裕！

コンピューターには、正確さの他にも、いくつも大きな強みがあります。その一つは、**動作がとても高速**な事です。 たとえば、同じ作業(処理)を 100 回、繰り返し行う場合を考えてみましょう。 100 回となると、人間が行うと結構な時間がかかるはずです。 慣れない内は何度か間違う事もあるでしょう。 特に、疲れてくると注意が続かなくなるので、休憩も必要です。



それに対してコンピューターは動作が高速なため、**そもそも 100 回程度であれば一瞬で終わってしまう事が多い**でしょう。具体的にかかる時間はプログラムの内容によって異なりますが、人間が同じ手順を行うよりは、はるかに高速です。



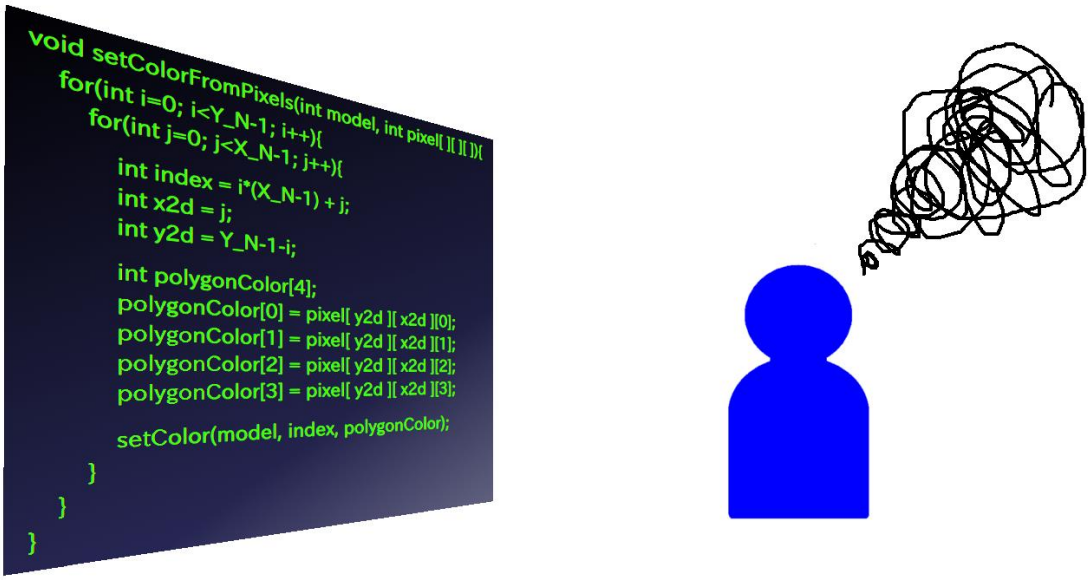
さらに **1 万回、1 億回…と繰り返す必要がある場合**には、それなりに時間がかかるかもしれません。それでも、先に述べた通り、**コンピューターは毎回ちゃんと指示通りに正確に処理を続けてくれます。**基本的に、慣れや休憩なども無くて大丈夫です。さすがに年単位でずっと稼働させ続ける場合などは、耐久性や冷却性などに配慮し、たまに再起動やメンテナンスなども必要になったりしますが、それでも実際にほぼ一年中ずっと動き続けているコンピューターは、世の中にたくさんあります。人間にとっては、昼も夜も休憩なしに、一年中ずっと同じ作業をミスせず続けるのは、非常に厳しいですね。この「繰り返しへの強さ」も、コンピューターの大きな強みです。

■ コンピューターと人間を繋ぐ「 プログラミング言語 」

■ プログラムはどう書くべき？ コンピューターのつもりになって考えてみよう

さて、コンピューターとプログラムのイメージがつかめた所で、もう少し具体的なプログラミングの話に入りましょう。**実際に、プログラムはどのように書けばよいのでしょうか？**

まず答えを先に言ってしまうと、「プログラミング言語」という特別な言葉で書いてあげる必要があります。しかし、プログラミング言語はかなり独特の雰囲気を放つものなので、いきなり「 このように書いてください 」と言われて、すんなりと受け入れられるものではありません。



そこで、ここはひとつ、まずはコンピューターのつもりになって、「 プログラムをどう書いてほしいか 」を一緒に考えてみましょう。たとえば、人間に頼むのと全く同じように、以下のようなプログラムを書くのはどうでしょうか：

すみませんが、昼食代を計算してもらえますか？ えーと、お茶が 100 円で、お弁当が 200 円です。

あなたは上のプログラム（のようなもの）を読んで、どう答えますか？「300 円」でしょうか。恐らくそれが最も普通の答えでしょう。**でも、それが唯一の正解でしょうか？ 100 人が読んで 100 人とも「300 円」と答えるでしょうか？** そのように疑ってみると、上のようなプログラムの書き方の問題点が見えてきます。

■ まずは内容を細かく、具体的に!

まず大きな問題点として、 上のプログラムには「 **何を何個ずつ買うのか** 」という事が書かれていません。10 人で会議中に出前をとろうという状況かもしれないので、10 個ずつ要るかもしれません。 300 円と答えた人はきっと、「 個数や他に必要なものが書かれていないという事は、単純にお茶とお弁当を 1 個ずつ買うのだろう 」と考えたはずです。でも思い出してください。**コンピューターは、自分でどうすべきか考えて対応するのは苦手** です。ちゃんと何を何個買うのか、はっきり書いてほしいはずです。

もう一つの問題点として、上のプログラムには「 **昼食代をどう計算すべきか** 」という事が書かれていません。300 円と答えた方は、きっと頭の中で「 **昼食代 ＝ お茶の値段×お茶の個数 ＋ お弁当の値段×お弁当の個数** 」という計算を行った事でしょう。なぜなら、そう計算すればよい事は、小学校で習うような常識だからです。でもコンピューターからすれば、上のような計算式も具体的に書いてほしいはずです。また、値段が定価か税別かわからないので、**場合によっては消費税がかかる**かもしれません。その場合は、消費税も含んだ計算式を書く必要がありますね。

問題点はまだあります。 たとえば、上のプログラムには「 何も答えない 」というのも一つの正解です。なぜなら、**「 計算してもらえますか? 」と尋ねられているだけで、「 答えを教えてください 」とまでは要求されていない**からです。 人間からすれば、言われなくても当然わかる事ですが、 コンピューターにとっては、答えを教えてほしいならそう書いてほしいわけです。

また、確実に計算してほしいなら、「すみませんが、〜 してもらえますか?」のような社交辞令も不要でしょう。 仮に、本当に疑問形でそう尋ねたいなら、「 どういう場合に断るべきか 」まで書くべきです。「えーと、」などのような意味のない記述も、コンピューターには不要ですね。

このあたりで一度、プログラムを書きなおしてみましょう。

昼食代を計算してください。 お茶の値段が 100 円で、お弁当の値段が 200 円です。 買うものは、お茶を 1 個と、お弁当を 1 個です。 昼食代の計算式は「 昼食代 ＝ お茶の値段×お茶の個数 ＋ お弁当の値段×お弁当の個数 」です。 最後に、計算した昼食代を教えてください。

内容が細かく具体的になって、行うべき処理ががはっきりしましたね。でもその代わり、少し読みづらくなってしまいました。このように細かい指示を書く場合、**箇条書き（ かじょうがき ）**の形のほうがすっきりします。箇条書きは、上から下に読みながら作業を進めるように書くとわかりやすいので、ここでもそうしましょう。

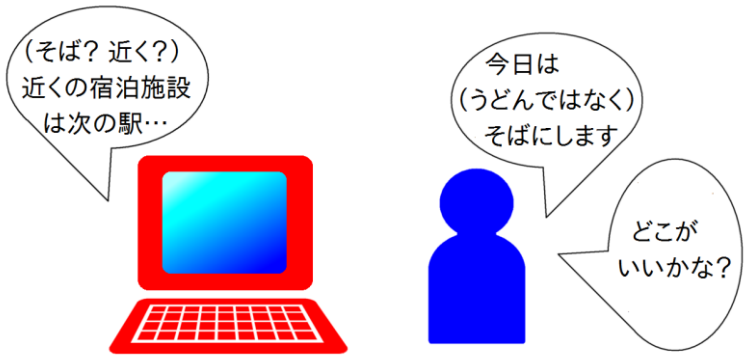
・お茶の値段は 100 円とします。
・お弁当の値段は 200 円とします。
・お茶の個数は 1 個です。
・お弁当の個数は 1 個です。
・昼食代を計算してください。計算式は、
　「昼食代＝お茶の値段×お茶の個数 ＋ お弁当の値段×お弁当の個数」 です。
・昼食代の計算結果を教えてください。

最初に比べると、だいぶコンピューターに伝わりやすそうな内容になってきた気がしませんか？実際にコンピューターに渡せるレベルまで、あと少しです。

■ 人間の言葉 ＝ 自然言語は、コンピューターには難しい ！ そこで記号やキーワードなどを使って、書き方を単純化

さて、上で書いたプログラムには、まだコンピューターにとって「最後の壁」が残っています。それは、各行が、普通の日本語の文章で書かれているという事です。と言っても、英語で書いてもだめです。日本語や英語のように、人間が日常で用いる言葉を一般に「 **自然言語** 」と呼びますが、実は**コンピューターにとって、自然言語は複雑すぎて、正しく理解するのがなかなか難しい**のです。

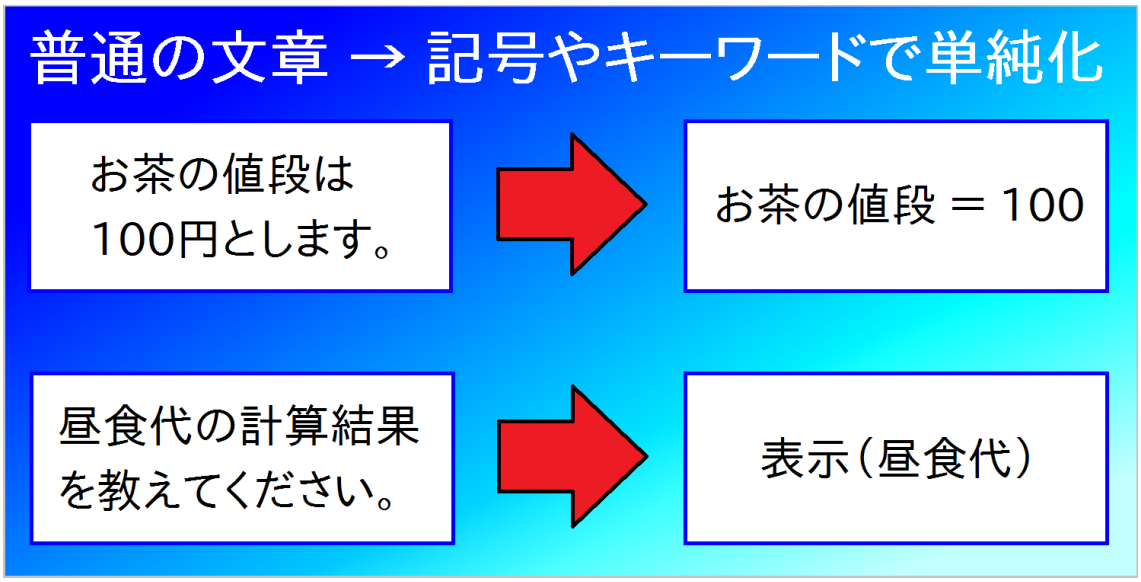
一応、自然言語を処理する研究は進んでいるので、 中途半端な正確さでよければ、自然言語でコンピューターに指示を出す事自体は不可能ではありません。実際、最近の PC やスマートフォンには、そのような機能が搭載されていたりもします。 言った事をちゃんと理解してくれる事もあれば、内容が間違って伝わる事もあります。



しかしながら、コンピューターの強みの一つとして、「 間違えず正確に処理を行える 」という事がありました。 なので、用途にもよりますが、 **そもそも指示内容を勘違いして理解されたのでは、本末転倒で意味がない**場合も多いはずです。 実際、いまはコンピューターに正確に指示したいので、それでは困ります。

そこで、プログラムの各行も、 普通の文章で書くのはなく、 もっとコンピューターが内容を簡単に解釈できるように、 ずっと単純化した形で書く事にしましょう。

たとえば「 **お茶の値段は 100 円とします。**」 の行のように、 何かの値をコンピューターに知らせたい場合は、「 **お茶の値段 ＝ 100** 」 といった具合に「 ＝ 」記号を使って書く事にしましょう。 こうすれば、「 ＝ 」記号の左右を見るだけで、「 『お茶の値段』に『100』をインプットすればいいんだな 」と簡単に解釈できます。



また、「 **昼食代の計算結果を教えてください。** 」 という行は、 要するに昼食代の値を画面に表示させたいわけですね。 であれば、「 表示 」というキーワード（※）を用意しておいて、 その後に値を「 ( ) 」記号で囲って書くと、その値を画面に表示するというルールを作ってしまいましょう。 そうすれば、この行は 「 **表示(昼食代)** 」 と単純に書けます。

(※ 厳密な話では、言語によっては構文キーワードではなく関数名（識別子）だったりもしますが、難しくなるのでここでは触れません。)

このように書き方を単純化してみると、先ほどのプログラムは次のように書けます：

```
・お茶の値段 ＝ 100
・お弁当の値段 ＝ 200
・お茶の個数 ＝ 1
・お弁当の個数 ＝ 1
・昼食代＝お茶の値段×お茶の個数 ＋ お弁当の値段×お弁当の個数
・表示 ( 昼食代 )
```

■ そして「 プログラミング言語 」へ

さて、私たちはここまで、プログラムをどのように書くのがよいかを、コンピューターの視点に立って考えてきました。そして最終的に、上で示したような「 **独特な書き方** 」に到達しました。これはもはや日本語の文章でも何でもない、一つの言葉 ＝ 言語と言えるでしょう。このように、プログラムを書くのに特化した言語を「 **プログラミング言語** 」と呼びます。つまり私たちは、ここまでの説明の中で、架空のプログラミング言語を一つ作ったわけです。

実際にこの架空の言語をコンピューターで使えるようにするのも面白そうですが、それは話の本筋から外れてしまいます。世の中には、すでに多くのプログラミング言語があります。手っ取り早くコンピューターで実行して結果を得るには、それらを使うのがよいでしょう。

本文書は「 **VCSSL** 」というプログラミング言語のガイドなので、例として、プログラムを VCSSL の形に書き直してみましょう:

```
int お茶の値段 = 100 ;
int お弁当の値段 = 200 ;
int お茶の個数 = 1 ;
int お弁当の個数 = 1 ;
int 昼食代 = お茶の値段 * お茶の個数 + お弁当の値段 * お弁当の個数 ;
print ( 昼食代 ) ;
```

「 int 」というキーワードが登場していたり、「 表示 」の代わりに「 print 」と書かれているなど、細かい部分は異なりますが、私たちが上で作ってきた架空の言語とよく似ています。

実際には「 お茶の個数 」などの項目名も、英語で「 numberOfTeas 」などと書くのが普通で、日本語でもローマ字にして「ochaKosuu」や「 kosuu_ocha 」などのように書きますが(読みやすさのため、単語の区切りの位置を大文字にしたり、アンダースコア「 _ 」をはさんだりします)、そうすると SF 映画などで見る、いかにも「プログラム」という雰囲気になります:

```
int ochaNedan = 100 ;
int obentouNedan = 200 ;
int ochaKosuu = 1 ;
int obentouKosuu = 1 ;
int daikin = ochaNedan * ochaKosuu + obentouNedan * obentouKosuu ;
print ( daikin ) ;
```

これまでの説明なしに、いきなり上のプログラムを見せられると、私たちの言葉とは全く違うので、理解に苦しんでいたかもしれません。 でも、ここまで読み進んだ皆さんは、なぜプログラムがこのような独特の書き方で書かれているのか、もうわかりますね。

さて、**上のプログラムはもちろん、実際に皆さんの PC で実行する事が可能**です。 詳しい実行方法はまた後の回で説明するとして、とりあえず実行してみると、画面に次のように表示されます：

```
300
```

そう、コンピューターが計算してくれた答えは、やはり 300 円。 長い道のりでしたが、私たちはようやく、コンピューターに昼食代を計算してもらう事ができたわけです。

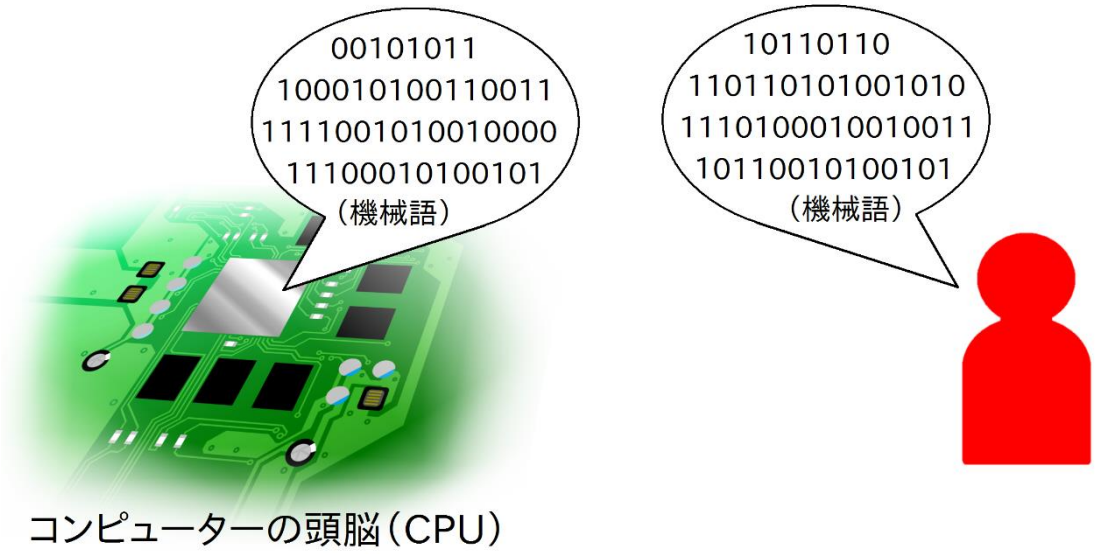
■ コンピューターの頭脳と プログラミング言語のしくみ

■ CPU

無事コンピューターにプログラムを実行してもらう事ができたところで、最後におまけ話です。プログラムが実際にどのように実行されているのか、そのしくみを見ておきましょう。

ここまで「 コンピューターは考えるのが苦手 」や「 複雑な言葉はわからない 」 などと言ってきましたが、 そもそも**コンピューターの頭脳**はどんなものなのでしょうか？ それは、**「 CPU 」と呼ばれる特殊な電子回路部品**です。

実は、上で書いたプログラムも、CPU が直接理解できるわけではありません。信じられないかもしれませんが、あれだけ細かくかみ砕いても、まだ複雑すぎるのです。CPU はもっとずっと単純な「 **機械語** 」と呼ばれる言葉を解釈して動くように設計され、造られています。従ってコンピューターは、機械語で書かれたプログラムをそのまま理解できます。



コンピューターの頭脳（CPU）

機械語は、 人間よりもやはり電子回路や電気信号にとって都合がいいようにできています。たとえば機械語は 1 と 0 を並べて記述されます。「 たし算 」命令は「 10001000 」と書くといった感じで、値も 2 進数で扱います。そうすると解釈や計算を行う回路を単純にできたり、電気信号として正確に扱いやすかったりするからです。電気的に値を覚えるのにも有利です。

また、機械語には非常に単純な命令しか存在しません。というのも、あまり機能が複雑になると、CPU のしくみや電子回路としての都合で、処理効率や消費電力などに不利が生じます。また

CPU の製造でも、なるべく回路規模が小さいほうが、一度にたくさん作れる上に不良品率も下がり、コストを抑えられます（シリコンの板の上に量産されます）。従って**機械語・CPU の機能は比較的単純にしておき、複雑な処理はそれらを組み合わせて使うのが合理的**なのです。

■ 言語処理系

実際に、（機械語ではない）プログラミング言語で書かれたプログラムは、各言語に専用の「 **言語処理系** 」というものによって、自動的に機械語の命令を組み合わせて実行されます。（実は、この言語処理系そのものもプログラムです。）

言語処理系には、実行しながら（逐次的に）機械語の処理を組み合わせてプログラム通りに動かす「 **インタープリタ** 」や、そもそも実行前にプログラム全体をまとめて機械語に変換してしまう「 **コンパイラ** 」など、複数のタイプがありますが、いずれにしても最終的に CPU を動かすものは機械語です。

もちろん、人間が直接、機械語でプログラムを書く事も不可能ではありません。実際、単純に 1 と 0 の代わりにキーワードで置き換えただけの「 **アセンブリ言語** 」を人間が使う事はあります。しかし CPU の機能は非常に単純で細かいので、それを人間が組み合わせて長く複雑な処理を書くのはたいへんです。

それよりは、ふつうの（ アセンブリ言語と比べると "高級" な ）プログラミング言語の方が、まだ人間寄りです。一方でプログラミング言語は、私たちの普通の言葉（自然言語）に比べると、ずっとコンピューター寄りでしたね。つまるところプログラミング言語は、対極的な「 人間 」と「 コンピューター 」、および「 自然言語 」と「 機械語 」との間をつなぐ存在であるとも言えます。

VCSSL の概要と導入

■ プログラミング言語 VCSSL の概要

今回からは、いよいよ実際にプログラムを書き、みなさんの PC の上で動かしていきましょう。

プログラムを書くために、「 プログラミング言語 」という専用の言葉を使う必要がある事は、前回説明した通りですね。 世の中にはたくさんのプログラミング言語があり、適材適所で使われています。 ただ、プログラミング言語は互いに共通点が多く、どれか一つで感覚やコツを掴めば、別の言語も習得しやすくなります。

このコーナーは VCSSL という言語の公式ガイドなので、 VCSSL を使ってプログラミングをしていきます。 そこでまず、VCSSL とはどんな言語なのかを見ていきましょう。

■ 書き方(文法)は C 言語風

数多くあるプログラミング言語の中で、「 C 言語 」はとても有名です。 みなさんの中にも、聞いた事がある方や、実際に使用されている方も多いかもしれません。 この C 言語は歴史も長く、幅広い分野で使われてきたため、 後に登場した多くの言語にも、いわば原型のような形で影響を与えています。 特に、いわゆる「 C 系 」と呼ばれる言語 ― C++、Java(R)、C# 等々 ― は、 プログラムをぱっと見たときに C 言語風の印象を受けますが、 これは書き方(文法)の基本部分に C 言語風のスタイルを採用しているためです。 VCSSL でもその通りで、基本的な処理の記述スタイルは C 言語風になっています。

C言語(※C99以降)のプログラム例

```
#include <stdio.h>
int main(void) {
    int a[ 100 ] ;

    for ( int i = 0 ; i < 100 ; i++ ) {
        if ( i % 2 == 0 ) {
            int value = 2 * i + 3 ;
            a[ i ] = value;
        } else {
            int value = 2 * i - 3 ;
            a[ i ] = value ;
        }
    }

    printf( "%d", a[ 80 ] ) ;
    return 0 ;
}
```

VCSSLでの同じプログラムの例

```
int a[ 100 ] ;

for ( int i = 0 ; i < 100 ; i++ ) {
    if ( i % 2 == 0 ) {
        int value = 2 * i + 3 ;
        a[ i ] = value;
    } else {
        int value = 2 * i - 3 ;
        a[ i ] = value ;
    }
}

print( a[ 80 ] ) ;
```

ただしすぐ後で述べるとおり、VCSSL は簡易用途を想定したシンプル寄りの言語であるため、メジャーな言語ほどの汎用性はありません。しかしながら、VCSSL を扱いながら覚えた基本は、C 言語や他の C 系の言語へのステップアップにおいても、きっと役立つ事でしょう。

■ もともとは電卓ソフト用の言語 - 手軽さやコンパクトさを優先

VCSSL はもともと、電卓ソフト上でちょっとした計算を自動化するための簡易言語として誕生しました。そのため、言語としての方向性は、以下のような3つの要求点に基づいています:

- C 言語などをすでに習得済みのユーザーが、すぐ違和感なく使えるように。
- はじめてプログラミングに触れるユーザーにも、なるべく学習しやすいように。
- 短い計算や簡易アプリのような、小さなプログラムを手軽に書けるように。

結果として、VCSSL の骨格は、**なるべく C 言語風の標準的なスタイルを保ちつつ、記述の手軽さやシンプルさを優先させたもの**になっています。

食べ物を選択し、値段を表示するプログラム

```
// ユーザーにメッセージを表示
popup( "何を食べますか?" );

// ユーザーに食べ物を選択してもらう
string menu = select( "お寿司", "お肉", "空気" );

// 以下、選択された食べ物に応じて値段を表示
if ( menu == "お寿司" ) {
    popup( "お寿司ですね。5000円です !" );
}
if ( menu == "お肉" ) {
    popup( "お肉ですね。1500円です !" );
}
if ( menu == "空気" ) {
    popup( "空気ですね。0円です !" );
}
```

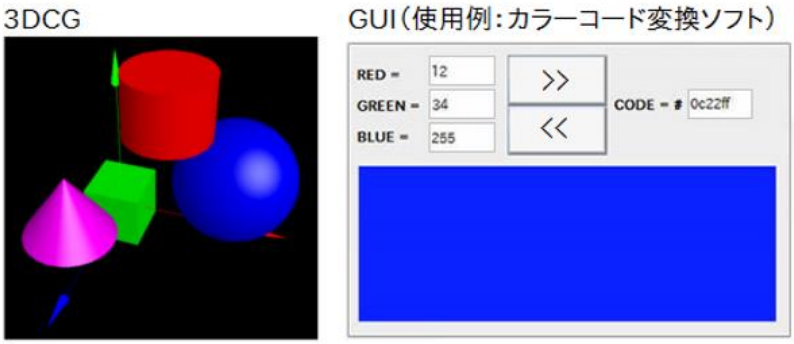
半面、VCSSL には、オブジェクト指向のような大規模開発向けの言語機能や、流行の先進的な言語機能などはありません。むしろ逆で、VCSSL の骨格は、枯れた技術でコンパクトにまとめ上げ、「**シンプルで変わらない道具**」という方向での完成形を目指しています。

■ ライブラリでは色々な用途をサポート - GUI や簡易 3DCG 機能も

一方、VCSSL が電卓用の言語であったのは初期の話で、現在はもっと色々な用途に対応できる、独立したプログラミング言語になっています。そこで、言語の骨格はシンプルなままに保ちつつ、色々な機能をまとめた「ライブラリ」というものが標準でいくつも付属しています：

・ VCSSL ライブラリ一覧
https://www.vcssl.org/ja-jp/lib/

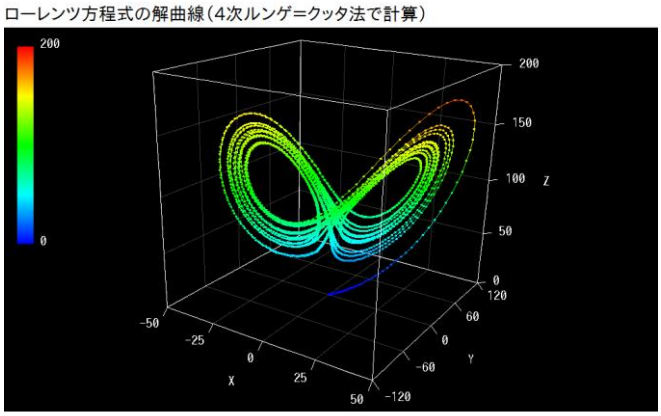
上記ページの通り、テキストやファイルの処理など日常で必要な機能をはじめ、GUI（画面作成）や 2D & 3DCG などのグラフィックス機能や、サウンド機能なども標準で利用できます。



ただし、先述した VCSSL の方向性と同様、これらの機能でも基本的に簡易用途での手軽さが優先されています。たとえば 3DCG の描画速度は数十万ポリゴン/秒くらいで、表面にテクスチャ（絵）を貼ったりもできませんが、そのかわり数学的に難しい知識がなくても扱えます。

■ 計算用途のための機能も

電卓用であったなごりとして、VCSSL には計算用途のための機能もあります。たとえば、処理結果をグラフに表示させる機能や、大きな桁数の計算なども標準でサポートしています。



インタープリタ型の言語でありつつ静的型付け（「変数と配列」の回参照 ）なのも、計算用途を考慮しての仕様です。処理速度も 1 億回演算/秒（ 100MFLOPS, 倍精度 ）程度と、ちょっとした数値計算や、データの変換・可視化などをこなせる水準です。

■「 VCSSL エンジン 」の上で動作、各種 PC でインストール不要で実行可能

VCSSL のプログラムは、「 VCSSL エンジン 」というインタープリタ型の言語処理系（ 前回の後半を参照 ）によって実行されます。この VCSSL エンジンは Java 言語で開発され、各種 PC 用 OS においてインストール不要で動作し、USB メモリーからも起動できます。

つまり VCSSL で書いたプログラムは、VCSSL の実行環境さえ一緒に置いておけば、何もインストールしなくても実行できますし、USB メモリーで持ち運んで使う事だってできます。この節の最後に述べる通り、自分で書いたプログラムを、VCSSL エンジンと一緒に配布する事もできます。

■ アプリケーション組み込み用サブセット（部分機能版）の「Vnano」も

現在、**VCSSL の中心的な機能のみを抜き出したサブセット（部分機能版）**として、**ユーザーが手軽に自作のアプリケーション等に組み込み、スクリプト処理機能として使用できる事を目指した「 Vnano （VCSSL nano の略）」**が開発進行中です。Vnano のスクリプトエンジンはオープンソースで、下記 URL の公開リポジトリ上において開発 & 公開されています：

https://github.com/RINEARN/vnano

現時点ではまだ正式リリースには至っていませんが、既にライブラリとして呼び出し可能で、大枠では動作する段階になっています。興味がある方は、ぜひ気軽に試してみてください ！

なお、将来的には、この Vnano のスクリプトエンジンを土台に、VCSSL のスクリプトエンジンを新実装に置き換えていく事で、VCSSL をオープンソースの言語にしていく計画もあります。

■ VCSSL でのプログラミング環境の準備

■ まずは VCSSL の実行環境をダウンロード ！

それでは、VCSSL でプログラミングをはじめるために、必要なものを揃えましょう。 ダウンロードと使用方法は、以下のページに詳しく載っていますので、それを参考にしてみてください。

・ 最新版 VCSSL のダウンロードと使用方法
https://www.vcssl.org/ja-jp/download/

■ ダウンロードした ZIP ファイルを展開

すでに述べた通り、VCSSL の実行環境はインストールせずに使用できます。 まず、**ダウンロードした ZIP ファイルを右クリックして、「 すべて展開 」 や「 ここに展開 」などを選んで展開します。**

※ 「問題を引き起こす可能性～」などのエラーが表示されて展開を完了できない場合、ZIP ファイルを右クリックして「プロパティ」を選択し、プロパティの画面の下にあるセキュリティ項目の「許可する」にチェックを入れて「OK」すると、以降は展開可能になります。このエラーは、インターネット等から入手した不明なプログラムを安易に実行しないための、OS のセキュリティ機能によるものです。

※ Linux®等をご使用の場合で、展開結果のファイル名の日本語が文字化けしてしまう場合があります。その場合、コマンドライン端末から以下のように unzip コマンドで展開してみてください：

```
cd　ZIP ファイルのある場所
unzip　-O　cp932　ZIP ファイル名
```

効果が無かった場合は、他の展開ソフトを使用するか、展開後に convmv コマンドなどで文字化けの修復処理を試してみてください（日本語ファイル名は CP932 でエンコードされています）。

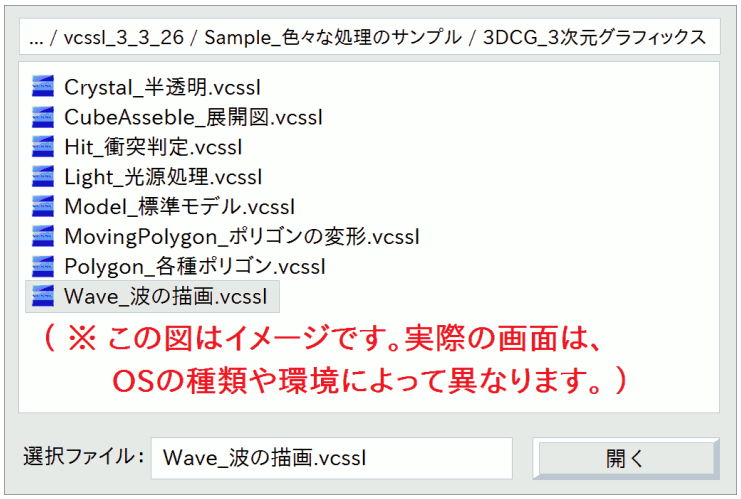
展開後、「License」フォルダ内にあるライセンス文書をご確認ください。使用には同意が必要です。要約すると「商用・非商用問わず無料で使用できますが、開発元は使用の結果に対して一切の責任を負いません」といった具合の内容になっています。

■ インストールせずに使用してみる

Microsoft® Windows® をご使用の場合は、ZIP ファイルを展開してできたフォルダ内にある「 VCSSL_?.?.?.bat※（バッチファイル、「?」の箇所はバージョン番号の数字）」をダブルクリックして実行してください。 Linux®等やその他の OS では、同フォルダ内までコマンドライン端末上で cd コマンドで移動し、「 java -jar VCSSL.jar 」と入力して「 VCSSL.jar（JAR ファイル）」を実行してください。ちなみに、この JAR ファイルが VCSSL の実行環境の本体（VCSSL エンジン）です。

※ 末尾の「 .bat 」の部分は、ご使用の環境における拡張子の表示設定によっては表示されません。

実行すると、以下のような画面が起動します。



この画面で、実行したいプログラムを選んでください。それだけで実行できます。簡単ですね。ここで試しに、同梱のサンプルプログラムをいくつか実行してみるのもおすすめです。

■ よく使う場合は、PC に VCSSL をインストール（導入）する事も可能

もちろん VCSSL の実行環境は、以下の通りに PC にインストール（導入）する事もできます：

・ VCSSL をインストールして使用する（Windows ※）
https://www.vcssl.org/ja-jp/download/#install_win

・ VCSSL をインストールして使用する（Windows 以外の OS）
https://www.vcssl.org/ja-jp/download/#install_other

インストールしておくと、書いたプログラムをダブルクリックするだけで実行できたり、コマンド入出力端末から vcssl コマンドで実行できたりと、使う頻度が高い場合に便利になります。

■ テキストエディタの用意

さて、続いてプログラムを書くために使う、テキストエディタという種類のソフトを用意します。とは言っても、PC を買った時点で、メモ用のもの（メモ帳など）が最初から付いていると思います。それでも一応プログラムは書けるので、最初のうちはそれを使うのもありでしょう。

一方で、「 テキストエディタ　プログラミング 」などで Web 検索してみると、プログラミングに適したものがたくさん見つかります。最初から本格的なものを使ってみたい場合は、その中で気に入ったものを選びましょう。たとえば Visual Studio Code などは、各種の OS で使えます：

・ Visual Studio Code（※）
https://code.visualstudio.com/

※ Windows、Visual Studio は米国 Microsoft Corporation の米国およびその他の国における登録商標です。

※ この文書は独立著作物であり、Microsoft Corporation と関連のある、もしくはスポンサーを受けるものではありません。

■ プログラムを書いて実行してみよう!

■ プログラムを書く

さあ、いよいよプログラムを書いて、実行してみましょう ！　まずは、上で用意したテキストエディタを起動して、以下の一行のプログラムを書きます：

```
print ( "Hello, World !" ) ;
```

※ 最後の「 ; 」記号を忘れてはいけません ！

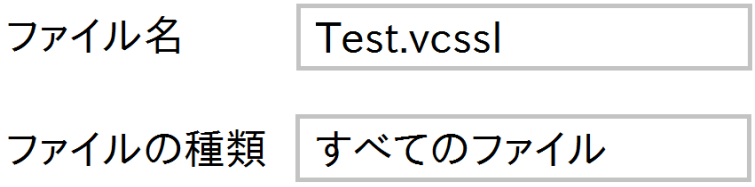
Windows 上でメモ帳を使う場合は、スタートメニューから「 Windows アクセサリ 」→「 メモ帳 」などで起動して、上のプログラム内容をコピーして貼り付けるか、書いてください。先ほど例に挙げた Visual Studio Code を使う場合は、起動してから「 ファイル 」→「 新規ファイル 」を選び、上のプログラム内容をコピーして貼り付けるか、書いてください。

■ プログラムに名前をつけて保存する

プログラムを書けたら、保存しましょう。まずテキストエディタのメニューから「 ファイル 」→「 名前を付けて保存 」を選びます。保存場所はとりあえずデスクトップでもどこでも OK です。

そして「 ファイル名 」の項目に好きな名前を書きますが、ここで名前の最後に必ず「 .vcssl 」を

付けてください。続いて、その項目の下にある「 ファイルの種類 」の選択ボックスから、「 すべてのファイル 」を選択してください。例として「 Test 」と名付けるなら以下の通りです：



上の「 .vcssl 」のように、ファイル名の「 . 」記号の後ろは拡張子と呼ばれ、種類を自動で判断するためのものです。Windows ではよく拡張子を表示しない設定になっていますが、必要に応じて「 Windows 拡張子 表示 」などと Web 検索し、表示するよう設定してもよいでしょう。

■ プログラムを実行する

それでは、上で書いたプログラムを実行してみましょう。先ほど「 インストールせずに使用してみる 」の項目で述べた通りの方法で VCSSL の実行環境を起動し、表示される画面から、上で作ったプログラム「 Test.vcssl 」を選択すると実行できます。（ .vcssl の部分は、拡張子の表示設定によっては見えません。）

実行すると、以下のように黒い画面に白い文字で「 Hello, World ! 」と表示されます：

そう、先ほどのプログラムは、画面に「 Hello, World ! 」というフレーズを表示する内容だったのです。実はこれは、プログラミングの入門で、とりあえず最初に表示させてみる、お決まりのフレーズです。これで私たちは、いよいよプログラミングの第一歩を踏み出したわけです！

■ プログラムを書きかえて再実行してみる

さて、黒い画面やテキストエディタはまだ閉じないでください（ もし閉じてしまった場合は、再び開いておいてください ）。 実際にプログラムを作る作業では、少し書きたしては、試しに実行して…をくり返す事になります。そのたびに上の手順を繰り返す必要はありません。

まずテキストエディタの画面に戻り、プログラムの内容を以下のように書きかえてみましょう：

```
print（ ”大丈夫！ ちゃんと日本語も表示できます！”）；
```

もとの内容は消してかまわないので、上の内容をそのままコピーして貼り付けるか、書いてください。 そしてテキストエディタのメニューから「 ファイル 」→「 上書き保存 」を選ぶか、または Ctrl キーと S キーを同時押し(※)してください。 すると、このプログラムの内容が、先ほどと同じ名前で保存されます。（※ エディタによっては、同時押しするキーは異なります。）

続いて、プログラムを実行すると出てきた黒い画面に戻り、メニューから「 Reset 」を選ぶか、または F11 キーを押してください。 するとプログラム内容が更新され、再び実行されます：

プログラムを書きかえた結果が反映されて、表示される内容が変わりましたね。そう、ちゃんと日本語も表示できるんです。

■ 日本語が文字化けする場合は、先頭行に文字コード宣言を書けば解決

ところで、上の手順の結果、下図のように謎の暗号のような文章が表示された方も多いかと思います。

これは「文字化け」という現象で、日本語を含むプログラムの場合にしばしば発生します。とりあえずここでの対策は簡単なので、心配は要りません。

文字化けは、**プログラムの先頭行に、以下のように「 文字コード宣言 」という特別な一行を書いておく**と、防ぐ事ができます：

```
coding UTF-8;

print（ ”大丈夫！ ちゃんと日本語も表示できます！”）；
```

または

```
coding Shift_JIS;

print（ ”大丈夫！ ちゃんと日本語も表示できます！”）；
```

のどちらかを試してみてください。これで再び、黒い画面のメニューから、Reset を選ぶか F11 キーを押して、再実行してみると：

ちゃんと表示されるようになりましたね。 最初からちゃんと表示された方も、上の 2 通りを試してみてください。どちらかで文字化けが生じる事を実感できるはずです。

上のプログラムで先頭行に登場している、「 UTF-8 」や「 Shift_JIS 」というのは、 **文字コード** と呼ばれるものです。 文字コードについて詳しい説明は「 ファイルの読み書き 」の回で行いますが、 簡単に言うと、日本語などの文字をどういうルールで扱うかを決めているものです。 標準で使用される文字コードは、OS の種類やテキストエディタによって違うため、 日本語の扱いがちぐはぐになってしまって、先ほどのように文字化けが生じるのです。

どの文字コードが正解かは使用環境によるため、 **これ以降、このガイドでは文字コード宣言は省略しますが、 基本的にはプログラムの先頭で、常に上のように文字コード宣言を書いておいた方が無難です。** プログラムがちゃんと動かない原因を苦労して探して、結局は文字化けが原因だった、というトラブルを確実に防げるからです。

■ 書いたプログラムを配布するには…

VCSSL で書いた自作プログラムは、完全に(著作権的にも)書いた人のもので、自由に使用し、配る事もできます。 その際、実行に必要な VCSSL エンジンも、標準付属のライセンスを同梱する事で、 自作プログラムと一緒に配る事ができます。手順の具体例を見てみましょう。

■ 基本的には、VCSSL エンジン(VCSSL.jar)などのファイルを一緒に置くだけ

まずフォルダ(ディレクトリ)を作り、その中に自作プログラム「 **(例) Test.vcssl** 」を置きます。そして VCSSL の実行環境をダウンロード・展開した中から、 以下のものを一緒に置きます。

・VCSSL.jar（ JAR ファイル / VCSSL エンジン本体 ）
・lib フォルダ
・License フォルダ

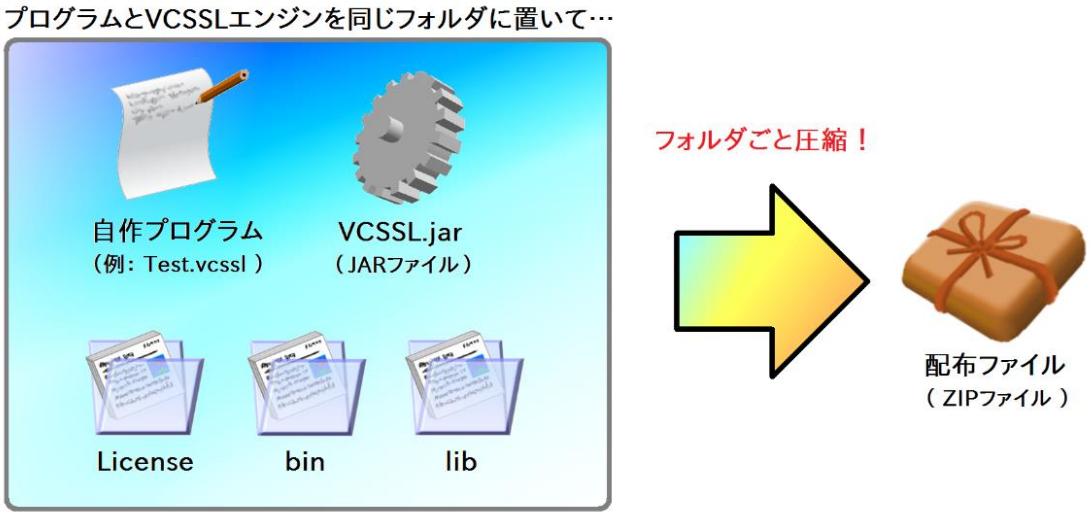
単純に java コマンドで VCSSL.jar を実行してもらうなら（配布側にとってはこれが最も手軽に済む方法です）、通常は上記だけで十分なのですが、使う機能によっては（2D/3D グラフプロットなど）、以下も一緒に必要になります：

・bin フォルダ

必要に応じて、ダブルクリック起動用のバッチファイルや、他のフォルダなども追加します※。

最後に、ユーザーへの使い方の説明などを記載した（いわゆる ReadMe などの）説明書も書いて追加しておくようにしましょう。VCSSL の実行環境を使用するには、上記 License フォルダ内のライセンスに同意する必要があるため、その旨も記載してください。

あとは、上記のものをまとめたフォルダごと右クリックして「 送る 」→「 圧縮 」 で ZIP ファイルにまとめて、自由に配布できます。



この ZIP ファイルをユーザーにダウンロード・展開してもらい、「 VCSSL.jar 」（またはダブルクリック起動用のバッチファイル）を起動してプログラムを選択してもらえば、普通に実行できます。

※「jre」フォルダやセキュリティ設定ファイルなど、同梱時に注意が必要なファイルも … 未起動の VCSSL 実行環境のものを同梱するのがおすすめ

VCSSL の実行環境をバッチファイルで起動した際に、動作に必要な Java®実行環境（JRE）が見つからなかった場合、ダウンロードして使用するかどうかユーザーに尋ねられますが、その JRE は「jre」フォルダ内に配置されます。その JRE には、その JRE 自身の（別の）ライセンスが適用されます。商用・非商用問わず無償で使用できますが、配布等には条件が存在するため、注意が必要です。詳細は ReadMe や「jre」フォルダ内の文書、および JRE 付属の説明書等をご参照ください。よくわからない場合は、「jre」フォルダは除いた形で配布するか、もしくは、VCSSL の実行環境をダウンロードした際に最初から入っていた内容だけに削ってから同梱する事をおすすめします。

また、VCSSL の実行環境を起動すると、「etc」フォルダ内（または VCSSL.jar と同じ場所）に、「 VCSSL_Security_Setting 」というファイルができますが、これも配布には適しません。これは、VCSSL のセキュリティ設定を保存するファイルで、実行環境が置かれている場所や日付などから生成されたハッシュ値も記録されており、それによって、コピーするとセキュリティ警告のメッセージと共に内容が（全てデフォルト値に）初期化されます。これは、ユーザーにとっては、入手したプログラムがいきなりセキュリティ的に緩い状態（PC 内のファイルを無確認で削除できるなど）で動作する事はリスクに繋がるためです。そのため VCSSL では、入手直後はデフォルトのセキュリティ設定で動作し、そこからユーザーの意志で設定を手動で変更するという方針を採用しており、従ってセキュリティ設定ファイルを同梱して配布しても警告と共に初期化されるのです。

このように、ダウンロード後に起動済みの VCSSL の実行環境には、再配布には適さないファイルも生成されている事に留意する必要があります。従って再配布の際は、ダウンロードしたまま未起動の VCSSL 実行環境のものを同梱する事をおすすめします。なお、同梱後にテスト実行するとやはり色々なファイルが生成されるため、配布用の内容をまとめる場所と、それをコピーしてテスト実行する場所は分けておく事もおすすめします。

■ プログラムの選択を自動化するには、設定ファイルを 1 行だけ書いて同梱

さて、VCSSL.jar(またはダブルクリック起動用のバッチファイル)を起動後、いちいちユーザーにプログラムを選択してもらうのは面倒かもしれません。これを自動化するには、メモ帳などのテキストエディタで以下の 1 行:

```
PROGRAM=Test.vcssl
```

を書き、「 VCSSLSetting.ini 」というファイル名を付けて、「 すべてのファイル 」の種類で保存してください。上の行の「 Test.vcssl 」の部分には、自分で書いたプログラムの名前を書きます。あとはこのファイルを、上で作ったフォルダ内か、または「etc」フォルダを同梱する場合はその中に置いておくと、起動時に自動でプログラムが選択されるようになります。

■ 「 VCSSL.jar (JAR ファイル) 」 のファイル名は、自由に変えて OK

ユーザーにダブルリックしてもらうファイルの名前が 「 VCSSL.jar 」という事が、なんだか特殊な印象がして嫌だ、という方も多いかもしれません。その場合は好きな名前に変えてしまいましょう。ただし、拡張子を 「 .jar 」から変えると実行できないので、注意してください。

■ 電卓ソフト「リニアンプロセッサー」での実行も

VCSSLのプログラミングと実行は、電卓ソフト「 リニアンプロセッサー 」で行う事も可能です。具体的には、リニアンプロセッサーのウィンドウ左上にある入力項目にプログラムを書き、「=」ボタンまたは F11 キーで実行します。詳細はリニアンプロセッサー付属の取扱説明書をご参照ください。

・リニアンプロセッサー
https://www.rinearn.com/ja-jp/processor/

リニアンプロセッサーでは、ウィンドウを最大化すると、入力項目が広がってテキストエディタの代わりとして使えるため、別途テキストエディタを用意・起動する必要はありません。VCSSL の実行環境も内蔵されており、書いたプログラムを電卓の機能としても使用できます。そのため、短いプログラムを手っ取り早く書いて実行したり、日常で頻繁に使う場合などに便利です。

一方で、それなりに長いプログラムになると、やはりこの節で行ったように、本格的なテキストエディタで書いて、VCSSL の実行環境で実行する方が便利です。用途に応じて使い分けてください。

プログラムの書き方の基本

前回の後半では、第一歩としてプログラムを実行しましたが、 そこではあくまでも実行手順の説明が目的だったので、プログラムの記述内容には触れませんでした。 そこで今回からは、プログラムの書き方の説明に入っていきましょう。今回はその基本です。

■ 最初に、半角と全角には注意が必要 ！

プログラムを書き始める前に、一つだけ非常に重要な、気を付けないといけない事があります。 それは、恐らくみなさんが、日本語を入力できる PC を使っているという事です。 **キーボードの左上にある「 全角 / 半角 」キーを押してみてください。 日本語を入力できるモードと、そうでないモードが切り替わりませんか？** ここでは、前者を全角モード、後者を半角モードと呼びます。 半角モードでは、以下のように文字の幅が細くなっているはずです：

```
全角（ 使用 NG！）：  ＡＢＣ　ａｂｃ　１２３　＋　＊　＝　”　；
半角（ 使用 OK！）：  A B C　a b c　1 2 3　+　*　=　”　;
```

プログラムは、英数字（ 英語のアルファベットと数字 ）と記号を組み合わせて書きますが、 **それらはすべて半角モードで入力する必要があります。** 全角の英数字や記号が混じっているとエラーになるので、気を付けましょう。（ただし後述する「文字列」では、全角も使えます。）

■ 前回のプログラムの意味を追いながら、書き方の基本を押さえていこう

さてここからが本題です。前回の後半では、以下の記述内容のプログラムを実行しました：

```
print ( ”Hello, World !” ) ;
```

※ 最後の「 ; 」記号を忘れてはいけません ！

これを実行すると、画面に以下のように表示されるのでしたね：

```
Hello, World !
```

実は、上のプログラムの記述内容には、色々と重要な要素がつまっています。 以下では、その意味を少しずつ掘り下げながら、プログラムの書き方の基本を説明していきましょう。 今回の最後まで読むと、上のプログラムの本当の意味が、すべてわかるはずです。

■「 print 」関数などで、画面に文章や数値を表示できる

まず気になるのが「 print 」という部分です。そこで以下のプログラムを書いてみてください：

```
print ( 123 ) ;
```

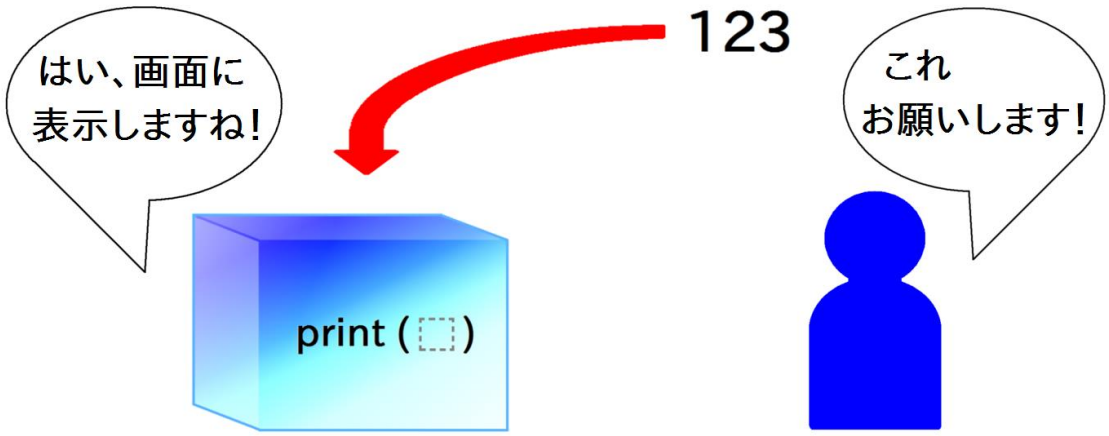
※ 最後の「 ; 」記号を忘れてはいけません ！

これを実行すると、コンソール画面（ 黒い画面 ）に以下のように表示されます：

```
123
```

この通り、**「 print ( ) 」のカッコ内に書いた内容が表示された**事がわかります。

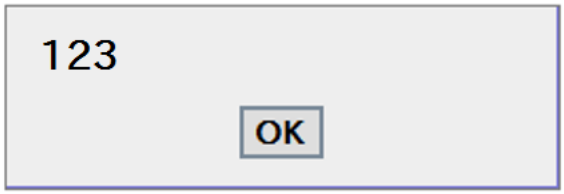
この「 print 」は、**関数**というものの一つです。 関数は、このようにカッコ内に書かれた内容に基づいて、 あらかじめ決まった処理を行ってくれます。 そう、この print 関数は、カッコ内の内容を画面に表示してくれる関数なのです。



似たような関数は他にもあります。たとえば以下のように書き直して実行してみてください：

```
popup ( 123 ) ;
```

ここで使っている「 popup 」関数は、カッコ内の内容を、小さなウィンドウに表示してくれる関数です。実際に実行してみた様子は以下の通りです：



※この図はイメージです。実際の見た目は環境などによって異なります。

ちゃんとウィンドウが起動し、「 123 」と表示されましたね。

以下に、表示関連の関数をリストアップしておきます。使い方は同じなので、気軽に試してみてください：

- print 関数 ：
  内容をコンソール画面に表示（ 追記 ）する。最後に改行しない。
- println 関数 ：
  内容をコンソール画面に表示（ 追記 ）する。**最後に改行する。**
- popup 関数 ：
  内容を小さなウィンドウなどに表示する。（「 OK 」を押すまで処理を止めて待つ。）
- output 関数 ：
  内容を小さなウィンドウなどに表示する。（ 処理を止めて待たない。）

なおこれらは、**カッコ内に「 , 」記号で区切って、複数の内容を表示してもらう事も可能**です。

■ プログラムの中に式を書いたら、自動で計算される

さて続きです。以下のプログラムを実行してみてください：

```
print ( 1 + 2 ) ;
```

実行結果は以下の通りです：

```
3
```

「 1 + 2 」という式を計算した値である、「 3 」が表示された事がわかりますね。

この通り、「 **プログラムの中に式を書いたら、コンピューターがその値を計算してくれる** 」というルールがあります。これはとても重要なルールです。というのも、print 関数のカッコ内だけの話ではなく、とにかくプログラムの中にある式は、すべて自動で計算されるからです。

この自動計算によって、上のプログラムは「 print (3) ; 」と書くのと同じ結果になったのです。コンピューターが計算してくれるのは便利なので、色々な式を書いて試してみたくなりますね。しかし、式の書き方は次回で詳しく説明するとして、とりあえず今は先に進みましょう。

■ 式ではなく文章（ 文字列 ）として扱う場合は、「 " 」記号ではさめば OK！

さて、「 式として計算してほしくない場合 」 はどうすればいいでしょうか？ たとえば計算問題プログラムなどを作っていて、「 １＋２ 」という文章をそのまま表示してほしい場合などです。

ここで最初のプログラム「 print ("Hello, World !") ; 」を思い出してください。「 Hello World ! 」という文章を 「 " 」 記号 ではさんでいましたね。そう、「 " 」 ではさめばよいのです：

```
print ( "1 + 2" ) ;
```

このプログラムを実行すると、結果は以下の通りです：

```
1 + 2
```

上の結果からもわかる通り、「 プログラムの中で、『 " 』 記号（ ダブルクォーテーションと呼びます ）ではさんだ部分は式ではなく、ただの文章として扱われる 」 というルールがあります。 これも print 関数のカッコ内だけの話ではなく、プログラム全体に通じる話です。

このように「 式ではない、ただの文章 」 の事を、プログラミングでは「 文字列 」 と呼びます。文章は 「 文字が列になっているデータ 」 だからです。この呼び方には慣れておきましょう。

■ 文字列なのに「 " 」記号を忘れると… 式と見なされてエラーに ！

よくあるミスとして、「 文字列として扱ってもらうべき内容を 『 " 』 ではさみ忘れる 」 というものがあります。 この場合、その内容が式の形をしているかどうかにかかわらず、式と見なされる事に注

意が必要です。ためしに、最初のプログラムでわざと「 " 」を忘れてみましょう:

```
print ( Hello, World ! ) ;
```

実行すると、画面の色が黒から青に変わり、以下のような内容が表示されます:

```
（青色の画面）
------------------------------------------------
SYNTAX ERROR - 構文エラー / 1
[ LINE - 場所 ] Test : LINE 1 行目付近
[ CODE - 内容 ] print ( Hello, World ! )
[ INFO - 詳細 ] 宣言されていない変数「 Hello 」を呼び出しています。

------------------------------------------------
SYNTAX ERROR - 構文エラー / 2
[ LINE - 場所 ] Test : LINE 1 行目付近
[ CODE - 内容 ] print ( Hello, World ! )
[ INFO - 詳細 ] 式を解釈できませんでした。式が正しい形をしていない可能性があります。

...
```

このように **画面が青くなった時は、プログラムの記述内容や動作にエラーがあり、実行できなかった** 事を意味しています。エラーの詳細を示すエラーメッセージが表示されています。その内容から、やはり「 Hello, World ! 」を式として解釈しようとしたものの、そもそも式ではないので、当然ながら失敗してしまった事がわかります。「 " 」を忘れないよう気を付けましょう。

■ 複数の処理は、「 ; 」記号で区切る

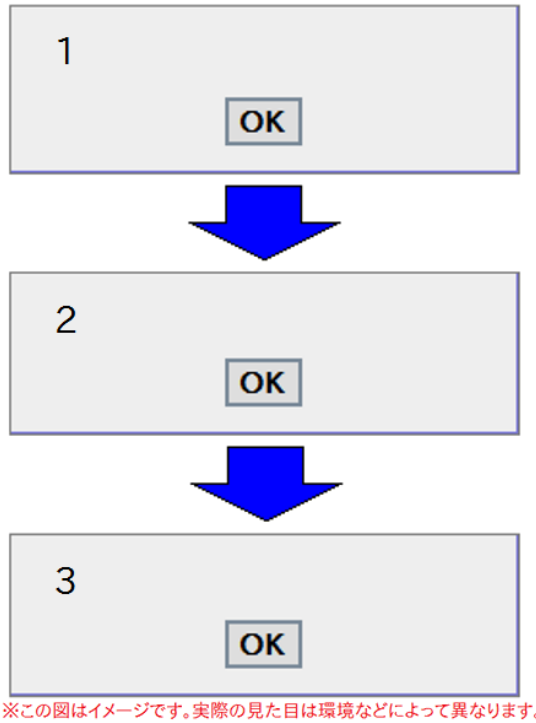
■「 ; 」記号は、一つの処理の終わりのしるし ！

さてここまでで、最初のプログラム「 print ("Hello, World !") ; 」の意味は、ほとんどわかりましたね。残っている説明は、**一番後ろにある「 ; 」記号(セミコロンと呼びます)** についてだけです。**この「 ; 」記号は、一つの処理の終わり(文末)を意味する** ものです。プログラムの中でこの記号があると、そこから先は、そこより前とは「 別の処理 」について書いていると見なされます。逆に言

えば、以下のように、**「 ; 」記号で区切って複数の処理を書ける** わけです:

```
popup ( 1 ) ;    popup ( 2 ) ;    popup ( 3 ) ;
```

このプログラムを実行すると、以下のように 3 回連続で、小さなウィンドウが表示されます:



※この図はイメージです。実際の見た目は環境などによって異なります。

この実行結果のふるまいを、プログラムを読み返しながら、頭の中で追いかけてみましょう。

まず、最初の「 ; 」記号までの内容である、「 popup (1) 」の処理が実行されます。 その結果、小さなウィンドウが起動して「 1 」と表示されます。「 OK 」を押すと、そのウィンドウが閉じます。最初の処理はこれで終わりです。

処理が終わると、コンピューターは次の処理を実行します。それは、次の「 ; 」までの内容である「 popup (2) 」です。 その結果、再びウィンドウが起動して「 2 」と表示され、ウィンドウを閉じると、この処理も終わりです。

その処理も終わったら、コンピューターはさらに次の処理を実行します。それは、さらに次の「 ; 」までの内容である「 popup (3) 」です。 その結果、ウィンドウに「 3 」と表示され、閉じると終わりです。 次は… おっと、プログラムの最後ですね。これですべての処理が終わりました。

■「 ; 」記号の後では、改行するのが普通

ところで、上のプログラムは、説明の都合で 1 行に書きましたが、実はこれは**少しお行儀の悪い**

書き方です。実際には以下のように、「 ; 」記号のあとに改行するのが普通です:

```
popup ( 1 ) ;
popup ( 2 ) ;
popup ( 3 ) ;
```

もしかすると、改行しないほうが読みやすいと思ったでしょうか。確かに慣れるまではそうかもしれません。でも、もっと大きなプログラムだと、数百個以上の処理を書く必要があります。そのような場合は 1 行におさまるはずもなく、どこかで改行する事は避けられません。

となると、各行ごとに気まぐれな長さで改行するより、毎回「 ; 」記号の後で改行するクセをつけたほうが「 1 行 = 1 つの処理 」となり、慣れると読みやすいのです。たとえば「 箇条書き 」を想像してみてください。内容は 1 行に 1 個ずつ書かれているほうが、読みやすいですね。

■ プログラム内にコメント(メモ)を書いておこう !

以上で、最初のプログラム「 print ("Hello, World !") ; 」の意味に関する説明は、すべて終わりました。図にまとめると、以下のようになります:

この内容を忘れないためには、ノートなどにメモしておくのもよいでしょうが、もっと手短な方法があります。というのも、実はプログラムの中に、「 コメント 」という形でメモを残せるのです。コメントは、プログラム実行時には完全に無視されるので、自由な内容を書く事ができます。

■ 1 行のコメントは、先頭に「 // 」を付ければ OK！

まずは 1 行のコメントを書いてみましょう。それには、コメントの先頭に「 // 」を付けます：

```
// 以下は、画面に「 Hello, World ！ 」と表示する処理 （ この行はコメント ）

print ( ”Hello, World !” ) ;
```

そうすると、**実行時に「 // 」よりも後は行末まで無視されます。**実際に実行してみましょう：

```
Hello, World !
```

この通り、確かにコメントを書いていない場合と、全く同じ結果ですね。

■ 複数行のコメントは、「 /* 」と「 */ 」ではさむと楽

コメントの行数が多い場合、すべての行の先頭に「 // 」を付けてもよいのですが、それは少し面倒です。そこで、**「 /* 」と「 */ 」ではさんだ中身はすべてコメントになり、実行時に無視される** というルールがあります。試してみましょう：

```
/*
 （この中はすべてコメント）
 以下は、画面に「 Hello, World ！ 」と表示する処理。
 「 print 」というのは、画面にカッコ内の内容を表示する関数。
 「 ” 」記号ではさんだ中身は、式ではなく文字列になる。
 「 ; 」記号は、処理の終わり（ 文末 ）の印。
*/

print ( ”Hello, World !” ) ;
```

実行結果は先ほどと全く同じです。長いコメントを書く場合に楽ですね。

どれくらいの頻度・分量でコメントを書くべきかというのは、個人差もあり、しばしば議論の対象にもなります。しかし、少なくともプログラミングに慣れるまでは、マメに書いておくのがよいでしょう。実はコメントを書くのは結構面倒なので、全く書かないクセが付きがちだからです。

式と計算、データ型

前回の途中で、式というものが登場しました。プログラムの中で式を書くと、コンピューターが自動でその値を計算してくれるのでしたね。今回は、色々な式を書いて計算してみましょう。

■ 式を書いてみよう

■ 式とは

本来ならば、まず「 式とは何か？ 」という詳しい説明が必要かもしれませんが、実はそれは少し難しい文法規則の話になってしまうため、ここではやめましょう。ここ言う「 式 」とは、あまり深く考えず、要するに普通の言葉で言う「 数式 」や「 計算式 」といった程度の意味と思っておきましょう。

$$1 + 2$$

$$1.2 / 3.1 + 2.5$$

$$y = \sin (x)$$

■ 「 式である事 」を意味するキーワードなどは不要で、結構どこにでも書ける

さて、プログラムの中に式を書くにはどうすればよいでしょうか。 **今までの話の流れからすると、「 この行は式です 」と知らせるキーワードなどが必要と思うかもしれません。しかし式に関しては、そういったものは不要です。** プログラムの中身は、式が大半を占めるからです。

プログラミングを始めたての頃は、プログラムの中の各行が、何をする行なのかを判断するのが大変です。実際、後の回で登場する「 制御文 」など、プログラム内には式以外のものもあります。しかし、それらは先頭のキーワードなどから判断できるようになっています。 なので、**『 そういった「 式でないもの 」以外は式だ 』と思ってしまうのが手っ取り早い**でしょう。

では、実際に式を計算するプログラムを書いてみましょう:

```
12345 + 67890 ;
```

これだけで OK です。簡単ですね。でもこのプログラムを実行しても、画面には何も表示されません。確かに 12345 + 67890 の値は計算されるのですが、その結果を画面に表示させる処理を書いていないからです。そこで前回のように print 関数を使いましょう:

```
print ( 12345 + 67890 ) ;
```

※ 最後の「 ; 」記号を忘れてはいけません ！

この print 関数は、**カッコの中身を画面に表示させる機能を持っている** のでしたね。実際に上のプログラムを実行すると、以下の通り、12345 + 67890 の計算結果が表示されます:

```
80235
```

上の例のように、式は関数のカッコ内など、いろいろなところに結構自由に書けます。「 式はここに書かないといけない 」というよりは、むしろ逆に「 式を書けないところもいくつかある 」くらいのイメージで、気軽にとらえておきましょう。

■ 式の途中で改行しても大丈夫!

上で書いたプログラムには、右端に**「 ; 」 記号**が付いていますが、これは前回説明した通り、一つの処理の終わりを意味するものでしたね。いちいち付けるのは面倒ですが、その代わり、とにかくこの記号の位置までは一つの処理なので、改行しても別の処理にはなりません。なので以下のように、式の途中で自由に改行したり、空白をはさんだりしても大丈夫です:

```
print (

    19800
      +
    52000
      +
    22800

) ;
```

上のプログラムを実行すると、以下のようにちゃんと計算結果が表示されます：

```
94600
```

式の長さに制限はなく、何文字でも、何行になっても大丈夫です。

■ 計算に使う記号（ 算術演算子 ）の種類と優先度

■ 足し算以外にも色々な計算ができる

ここまでは足し算を行ってきましたが、もちろん引き算もできます：

```
print ( 12345 - 67890 ) ;
```

この計算結果はマイナスになるはずですが、それでも大丈夫です。 実際に上のプログラムを実行してみると、以下のようにちゃんとマイナスの値が表示されます：

```
-55545
```

かけ算や割り算もできますが、私たちが普段使っている「 × 」記号や「 ÷ 」記号は、キーボードにはありません。 そこで代わりに、**かけ算には「 * 」記号（ アスタリスクと呼びます ）を使い、 割り算には「 / 」記号（ スラッシュと呼びます ）を使います。** 実際に使ってみましょう：

```
println ( 2 * 3 ) ;
println ( 10 / 5 ) ;
```

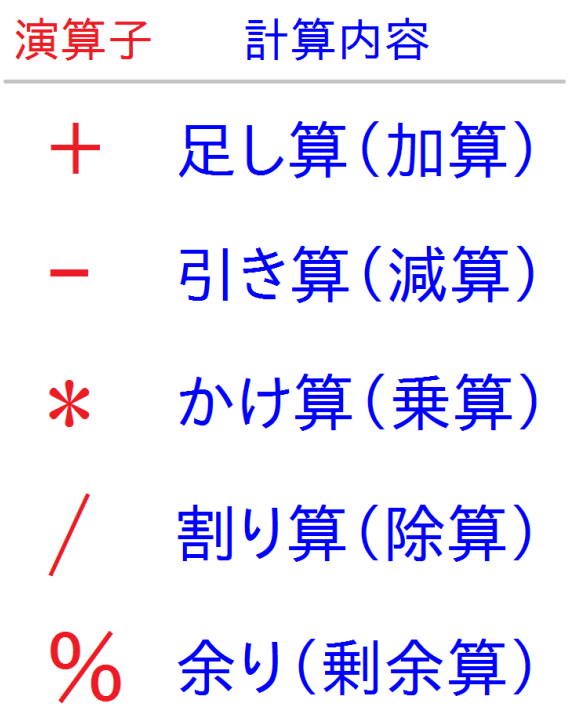
上のプログラムで使っている println 関数は print 関数とほぼ同じですが、画面にカッコ内の値を表示した後に改行してくれます。 そのため、複数の式の値を表示したいときに、行ごとに区切られて読みやすくなります。 実行結果は以下の通りです：

```
6
2
```

ちゃんと ２＊３ の値である ６ と、10 / ５ の値である ２ が、行ごとに表示されていますね。

■ 計算の記号 ＝ 算術演算子

さて、ここで計算に使ってきた「 ＋ - * / 」のような記号の事を、正確には「 **算術演算子** 」と呼びます。 難しそうな呼び方ですが、怖がる必要はありません。要するに、ただの計算の記号以上の何者でもありません。 算術演算子をまとめると、以下のようになります：



■ 計算の順序と、演算子の優先度

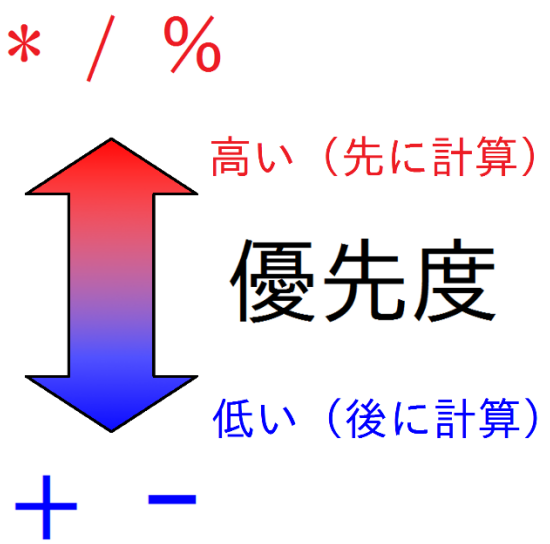
以下のプログラムのように、式の中に複数の計算が含まれている場合もあります：

```
print ( 1 + 2 * 3 ) ;
```

このような場合、「 どのような順序で計算されるか 」によって、結果が異なります。 たとえば、まず １ ＋ ２ を計算して、その値に ３ をかけた場合、計算結果は「 ９ 」になります。 逆に、まず ２＊３ を計算して、それに １ を足した場合、計算結果は「 ７ 」になります。 実際には：

```
7
```

この通り、後者のように計算されます。これは、私たちが小学校で習うのと同じように、**「 かけ算・割り算は、足し算・引き算よりも先に計算する 」**というルールがあるからです。これをもう少しプログラミングの世界の言葉で言うと、**『「 * 」と「 / 」演算子は、「 + 」と「 - 」演算子よりも優先度が高い 』**などと言います。優先度が高い演算子のほうが、先に計算されるのです。



そして、かけ算と割り算などのように、**同じ優先度の演算子が並んでいるときは、左から順番に計算**されます：

```
print ( 4 * 5 / 10 ) ;
```

上のプログラムを計算すると、左から順にまず 4 * 5 が計算されて 20 になり、続いてその値が 10 で割られて、最終的に 2 になります。実際の実行結果は以下の通りです：

```
2
```

■ カッコで囲めば、その中が優先的に計算される

たとえば「 長さの合計を 2 倍したい 」というように、かけ算よりも足し算を先に行ってほしい場合もありますね。そのような場合は、先に計算してほしい部分をカッコで囲めば OK です：

```
print (   2 * ( 30 + 50 + 80 )   ) ;
```

上のプログラムを実行すると、まずカッコ内の 30 + 50 + 80 が計算されて 160 になり、それに 2 がかけられて 320 になります。実際の計算結果は以下の通りです：

```
320
```

カッコは何個でも使う事ができ、入れ子にする（ 重ねる ）事もできます。

■ データ型

■ 小数点のある計算もできる

ところで、以下のように、小数点のある数の計算を行う事もできます：

```
print ( 1.2 + 3.4 ) ;
```

上のプログラムの実行結果は以下の通りです：

```
4.6
```

ちゃんと結果にも小数点が付いています。 上のように**小数点のある数**を、ここでは**「 小数 」**と呼ぶ事にしましょう。 **より厳密に「 浮動小数点数 」や「 実数 」と呼ぶ**べきだ、という意見があるかもしれませんが、 どちらも少し難しそうなイメージが出てしまうため、ここでは細かい厳密さは置いておく事にします。 なお、「 １ 」や「 ２ 」のように、小数点の付いていない数は**「 整数 」**と呼びます。

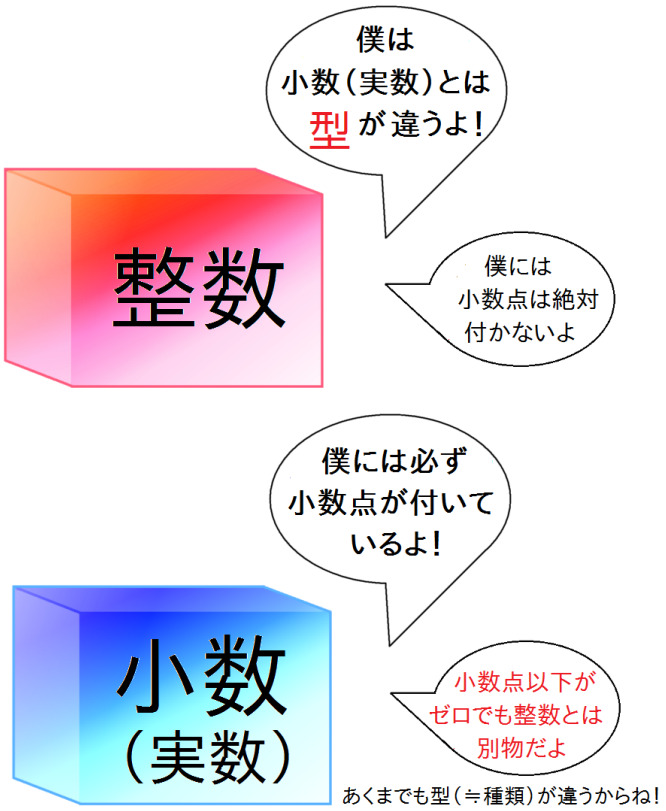
■ 整数と小数は種類（ データ型 ）が違う!

さて、ここでとても重要なルールがあります。 それは **「 プログラム内の整数と小数は、別の種類のデータとして扱われる 」** という事です。 といっても、この事自体は、そう驚く事ではありません。なぜなら、私たち人間だって、手計算で筆算をする場合などには、**整数と小数では計算のやり方が少し違うので、両者を区別して適切な計算方法を使い分けている**はずです。全般的に、整数同士の筆算よりも、小数同士の筆算の方が少し複雑になるので、小学校でもまず前者を習ってから、後で後者を習いましたね。ただ、そういった計算方法の切り替えは、小学校で毎日たくさん訓練したので、もうあたりまえすぎて、普段は意識せず行っているかもしれません。でも、区別はしているはずです。

これはコンピューターの電子回路にとっても同じで、やはり**小数と整数では計算のやり方が違うので、「小数のデータ」と「整数のデータ」は、内部で区別して扱う必要がある**のです。また、私たちが小数を書き表すためには、整数では不要な「 小数点 」というものを付ける必要があります。それと同様に、やはり**コンピューター内部で(１と０の列として)小数のデータを表す際にも、整数にはない工**

夫が必要になってきます（ ※ 詳しくは、IEEE754 という規格で決まっています ）。

このようにコンピューターの気持ちになって考えれば、プログラム内で小数と整数を別の種類のデータとして扱うのは、不思議な事ではありませんね。このような種類の事を一般に「 データ型 」、または単に「 型 」(英語では「 タイプ 」)と呼びます。整数と小数は型が違うというわけです。



ただし、プログラム中に直接書かれた数値が整数か小数かは、小数点以下がゼロかどうかではなく、単純に小数点が付いるかどうかだけで判断されるという事に注意が必要です。つまり「 1 」と「 1.0 」は、前者が整数型のデータで、後者が小数型のデータであり、別のものと見なされます。

■ 割り算は、データ型によって振る舞いや結果が違うので、注意が必要！

整数と小数の違いに特に気を付ける必要があるのは、割り算を行うときです。実は 整数と小数とで、割り算の振る舞いや結果が違うからです。一体どう違うのでしょうか？

例として「 20 / 8 」について考えてみましょう。多くの人は、この式の答えは「 2.5 」だと思うでしょうし、それは確かに一つの正解です。でも、小学校時代に、小数を習う前には、どう答えたでしょうか？「 2 余り 4 」と答えたのではないでしょうか。懐かしいですね。これも一つの正解でしょう。つまり割り算には、そもそも整数と小数で 2 種類の答え方があるわけです。

そしてプログラム内で、整数同士で割り算を行うと、後者のように整数の答えが求まります。余りは「 % 」という演算子で計算すれば求まります：

```
println ( 20 / 8 ) ;        // 割り算の結果が整数で求まる
println ( 20 % 8 ) ;        // 割り算の余りが求まる
```

上のプログラムの実行結果は以下の通りです:

```
2
4
```

ちゃんと「 2 余り 4 」という答えが求まっていますね。

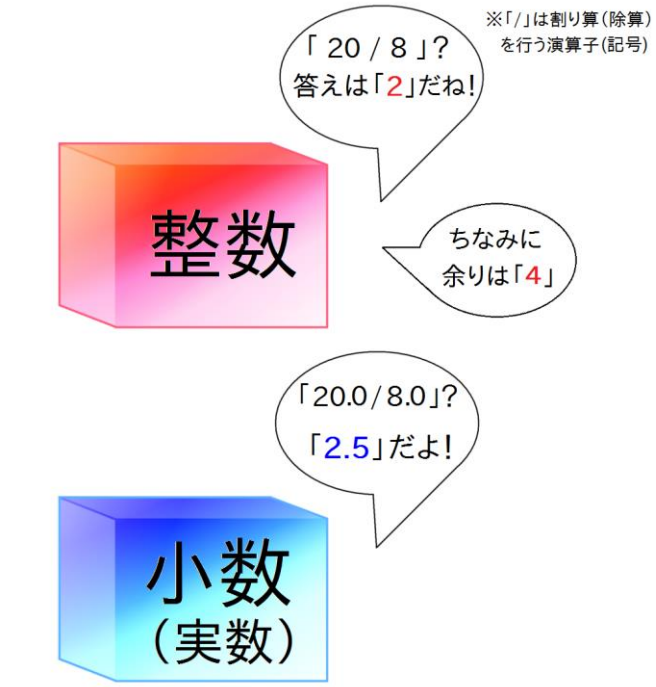
一方、小数同士で割り算を行った場合は、小数の答えが求まります。とにかく数値に小数点を付ければ、小数と見なされるのでしたね。試してみましょう;

```
println ( 20.0 / 8.0 ) ;        // 割り算の結果が小数で求まる
```

上のプログラムの実行結果は以下の通りです:

```
2.5
```

今度は「 2.5 」という答えが求まりました。慣れるまでは、両者の違いに注意が必要です。



同じ演算子(記号)を使っても、対象のデータ型(≒種類)によって、
演算(計算)のされ方や結果が違う!

■ 計算に使う数値のデータ型と、計算結果のデータ型との対応関係

　では、整数同士でもなく小数同士でもなく、以下のように整数と小数を両方使った割り算の結果は、どちらの型になるのでしょうか？

```
print ( 20 / 8.0 ) ;
```

結果は以下の通りです：

```
2.5
```

小数の結果が得られましたね。 実は、割り算だけでなく、かけ算や足し算・引き算も含めて、算術演算子の計算結果の型は、以下のルールで決まります：

- **整数同士の計算結果は整数**
- **小数同士の計算結果は小数**
- **整数と小数の計算結果は小数**（※）

（ ※ より厳密には、整数と小数を組み合わせた計算の場合、整数のほうの値が小数に変換されてから、小数同士の計算が行われます。）

■ 小数(実数)の誤差と丸め

■ 小数では誤差にも要注意

　割り算以外にも、整数と小数の扱われ方の違いには、注意が必要な点があります。 それは小数における**「 誤差 」**です。 より詳しく言うと、**「 コンピューター内で保持している小数の値や計算結果が、正確な値からずれてしまう 」**事に注意が必要なのです。 そんな事が起こったら大変じゃないか？ そう、知らないと結構大変なのです。

　論より証拠、まずは以下のように計算させてみましょう：

```
print ( 0.1 * 3.0 ) ;
```

正しい値は 0.3 のはずですね。しかし実行結果は：

```
0.30000000000000004
```

と、このように**正しい値(0.3)から 0.00000000000000004 だけ大きな値**になっています。整数で計算していた時は、こんな奇妙な事は起こりませんでしたね※。 でも、小数ではよくある事なので、要注意なのです。

(※ 整数でも、値が極端に大きくなったりして、扱える範囲を超えてしまうと、変な値になる「オーバーフロー」という現象が生じる事はあります。しかしそれは、値の範囲さえ注意すれば防げます。)

もう一つ例として:

```
print ( 0.3 * 3.0 ) ;
```

と計算させると:

```
0.8999999999999999
```

今度は**正しい値(0.9)から 0.00000000000000001 だけ小さな値**になってしまいました。どちらの場合にも、正しい値からほんの少しだけずれています。このずれが、誤差です。

このような誤差が生じる理由をちゃんと説明しようとすると、 土台となる長い説明がいくつも必要なのですが、一言で言うと、これは**コンピューター内部での、小数の扱い方の都合によって生じてしまうもの**です。もっと詳しく知りたい方は、「 浮動小数点数　誤差 」などと Web 検索すると、色々なページで解説されています。でも、誤差が生じる理由は複数あって、それぞれが結構難しいです。

しかし何よりも重要なのは、このような誤差の存在を事を知った上で、**どう対処するか**という事です。という事で、一番単純な方法で対処してみましょう。

■ 四捨五入などの「 丸め 」で誤差部分を落とす

幸い、この種の誤差は、**もとの値と比べてかなり小さい**という特徴があります。 そして多くの用途においては、上の例のように小数点以下 17 桁目とかにくる値は、もう無視して捨ててしまってもいいと思いませんか? 私たちが日常で小数の計算をする場合も、値が割り切れなかったりすると、「 小数点以下 5 桁もあれば十分だろう 」などと計算を打ち切ったりしますね。プログラムでもそうしてみましょう。

ただ、単純に「小数点以下 5 桁よりも小さい桁は捨てる」とすると、 0.30000000000000004 は 0.3 になって OK ですが、0.8999999999999999 は 0.89999 になるはずで、あまりうれしくありません。 後者は 0.9 になってほしいですね。 なので、実際には単純な切り捨てよりも、小学校で習った「 四捨五入 」などの工夫した方法で、余分な桁を落とす場合が多いです。 このような、数値の余分な部分を落とす処理の事を、**「 丸め(まるめ)」** や **「 端数(はすう)処理 」** などと

呼びます。

それでは実際に、先ほどの計算結果を、四捨五入で小数点以下 5 桁に「 丸め 」て表示させてみましょう:

```
println (     round(     0.1 * 3.0 ,     5, HALF_UP )     ) ;
println (     round(     0.3 * 3.0 ,     5, HALF_UP )     ) ;
```

実行すると:

```
0.3
0.9
```

無事、誤差の部分が「 丸め 」られて落ちましたね。このように、値や式を round(ラウンド)関数で囲むと、 少数の値の中で、使いたい桁数よりも小さな部分(端数)を丸める事ができます。 なお、上のプログラム内で、「 5, HALF_UP (ハーフアップ) 」の部分が、「 小数点以下 5 桁より小さい部分を四捨五入 」という指示になります。

■ 他の誤差対処法

小数の誤差は、上のように丸めてしまえば済む場面が多いのですが、誤差や精度がとても重要な用途では、 別の方法で対処される事もあります。

例えば、プログラミング言語によって呼び方は異なるのですが、十進型などと呼ばれる少し特別なデータ型を使って、小数の計算を行う方法などがあります。 その方法では、上で見てきたような種類の誤差は生じません。

もう一つ、ものすごく長い桁数の小数を扱いたい場合には、別の特殊な方法が用いられる事があります。 ふつうの小数のデータ型は、扱える桁数が(たかだか十数桁くらいに)限られていて、扱いきれない部分は勝手に丸められて、そこで誤差が生じます。 それでも数百桁、数千桁といった桁数を正確に扱いたい場合は、多倍長計算という特別な方法が使用されます。

ただ、誤差には色々な種類があって、「 どれか最強の方法を使えば、すべての誤差を気にする必要が無くなる 」というわけではないので、注意は必要です。むしろ逆に、そのような厳密さが要求される場面では、精度や誤差や丸めなどについて、しっかりと深く理解してから注意してプログラムを書く必要があります。

一応は、VCSSL にも十進で多倍長計算を行うためのデータ型(varfloat 型)が用意されているのですが、 具体的にふみ込むには、そのような難しい説明が必要になってしまうので、ここでは紹介だけにとどめておきましょう。

■ データ型としての文字列

■ 文字列も、整数や小数とは種類が違うデータ ＝ 別のデータ型

さて、いよいよ今回の最後です。実はデータ型は、整数と小数だけではありません。たとえば前回、「 文字列 」について少しだけ説明しました。「 "ABC" 」のように文章を「 " 」記号 ではさめば、その間は「 文字が並んだデータ 」、つまり文字列と見なされるのでしたね。この「文字列」というのも、一種のデータ型です（※）。確かに文字列は、整数や小数とは全然種類が違うものなので、データとしても区別して扱う必要がありますね。

（ ※ 文字列を扱うデータ型やクラスなどが、標準で存在するかどうかは、プログラミング言語によって異なります。例えば C 言語では、1 個の文字を扱うデータ型はありますが、文字列全体を扱うデータ型はありません。そのため、C 言語において文字列のデータは、個々の文字のデータが並んだものとして扱われます。 ）

■ 文字列に対する足し算は、文字列を結合する（つなぐ）処理になる

ところで、文字列に対しては引き算やかけ算・割り算などは行えないのですが、実は足し算だけは行えます。ただし、その振る舞いは、整数や小数とは全く違うものになります。試してみましょう：

```
println ( ”ABC” + ”DE” ) ;
```

このように文字列同士を足すと、その結果は、文字列を結合した（つないだ）ものになります。 実際の実行結果は以下の通りです：

```
ABCDE
```

確かに文字列が結合されていますね。同様に、文字列と数値を足す事もできます：

```
println ( ”ABC” + 123 ) ;
```

このような場合、まず数値「 123 」が文字列「 "123" 」に変換された上で、結合されます。 実際の実行結果は以下の通りです：

```
ABC123
```

文字列と数値を足した結果の型は、数値が整数か小数かを問わず、必ず文字列になります。これも、数値同士の足し算とは違う振る舞いですね。

■ 演算の振る舞いがデータ型によって違う事は、データ型の役割の一つ

ここまで見てきたように、足し算や割り算などの演算子の振る舞いが、データ型に応じて変化してしまう事は、まぎらわしく感じるかもしれませんが、しかしデータ型の重要な役割の一つでもあります。小学校の算数で「 ABC + DE = ？ 」といった計算を習わない通り、そもそも意味的に、文字列に対して数値の足し算と同じ事はできないはずです。なので、もしコンピューターが、文字列のデータに対して数値の足し算と同じ処理を行ってしまったら、その結果のデータは意味不明なものになってしまいます。同様に、数値でも小数点の有無によって計算のやり方は違うので（実際に学校でも別々に習いますね）、小数のデータに対して整数の演算と全く同じ処理を行ってしまうと、やはりその結果のデータはおかしなものになります。したがって、ちゃんとデータの種類に応じて、意味のある処理をコンピューターに行ってもらう必要があり、データ型はそのための重要な役割を担っているのです。

変数と配列

前回は、プログラム内にいろいろな式を書いて、コンピューターに計算させてみました。 今回は、その計算結果をメモしておける「 変数 」という仕組みを使ってみましょう。

■ コンピューターに値をメモできる「 メモリー 」

■ 実用の場面では、計算途中の値をメモするものも必要

プログラム内の式では、足し算や引き算、かけ算や割り算、小数点のある数値の計算などができましたね。 これだけでも、自分で電卓をたたく代わりなどに、それなりに便利に使えそうです。 でも、みなさんが電卓を使う場面を思い返してみてください。机の上にあるのは電卓だけでしょうか？ 恐らく、**なにか値をメモするためのもの（紙など）**も必要ではないでしょうか。

たとえば、ある計算を行って、その値をメモして、また別の計算を行って、その結果をメモして、 さらにメモした値を使って計算を行って… などです。よくありますね。 このように、**日常的な計算の場面では、計算を行うもの（電卓）と、 計算途中の値をメモしておくもの（紙など）が、大体ワンセットで必要**になります。 なので電卓自体に、値をメモしておく機能が付いている製品も数多くあります。

■ コンピューターにも、値をメモするための部品「 メモリー 」が入っている！

これはプログラムの中でも全く同じで、値をメモしたいという場面はたくさん出てきます。 でも、紙を用意しなくても大丈夫です。 コンピューターの中にはちゃんと、**値をメモするための部品である「 メモリー」**が搭載されています。

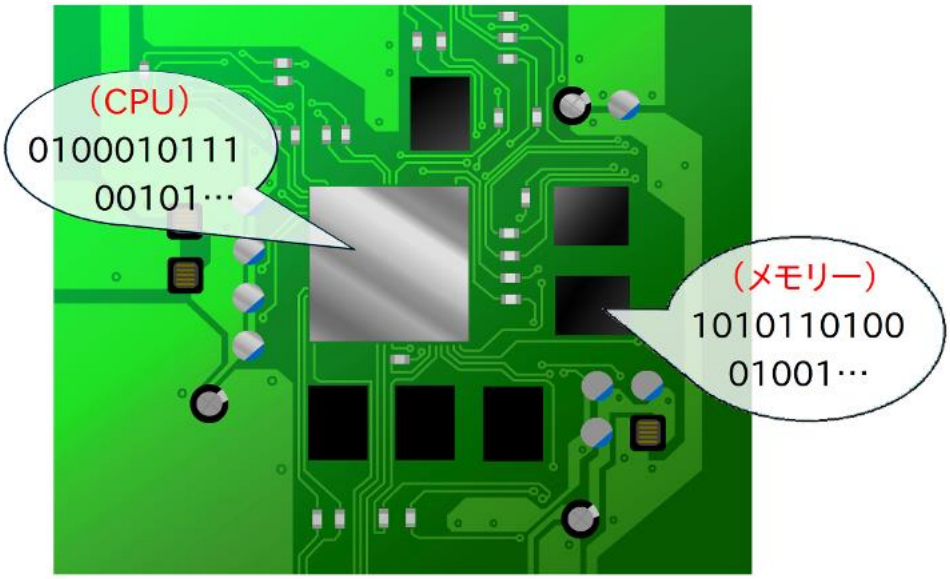
※基板上に直接付いていたり、交換可能な部品だったりします

メモリーは英語で「 Memory 」と書き、これは日本語では「 記憶 」という意味です。 つまりメモリーは、メモ内容を紙などに書きだすのではなく、自分自身の中に覚えておく事ができる部品です。そのため動作が高速で、メモするのも、メモした値を読みだすのも一瞬でできます。

さらにメモリーには、とても多くの値をメモしておく事ができます。 その量は、みなさんが PC(パソコン)を買うとき、カタログに 「 メモリー: 4GB 」 などと書かれているはずです。「 B(バイト) 」は、値を覚えておく単位のようなものです。「 G(ギガ) 」は 1000 の 1000 倍の 1000 倍を意味します。 つまり**メモリーは、何十億個以上もの値をメモしておく事ができる**のです。

■ CPU と同様、メモリーも 1 と 0 でやり取りする電子部品

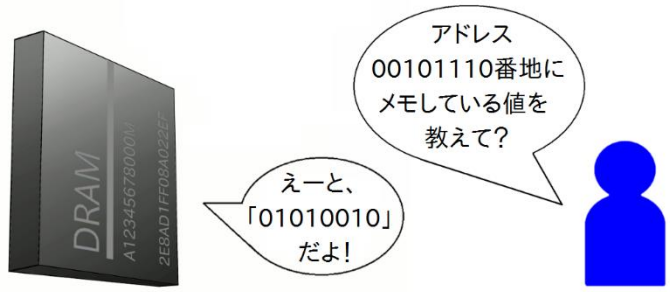
このように高性能なメモリーですが、直接使うのは、人間にとって簡単ではありません。 というのも、第 1 回の最後で触れた、コンピューターの頭脳である CPU の話をふり返ってみてください。CPU はあくまでも電子部品であるため、 人間よりも電子回路や電気信号にとって都合がよいように作られているのでしたね。 プログラムは **1 と 0 を並べて記述**する必要がありますし、 計算する数値も、CPU 内部ではすべて 1 と 0 の列として扱われています。 そしてメモリーも、CPU とやり取りする電子部品であり、実は**やり取りできるのは 1 と 0 の列だけ**なのです。 当然、記憶できるのも 1 と 0 の列だけ、という事になります。つまり、もし人間がメモリーを直接使って値をメモしたければ、 まずその値を、何らかの方法で 1 と 0 の列に置きかえる必要があります。



また、メモリーはたくさんの値を記憶できるわけですが、**「 どの値を、メモリーの中のどこに記憶させるのか 」**を、 無味乾燥な**「 アドレス (Address、住所という意味) 」という番号**で指示しないといけません。 これもコンピューター内部では 1 と 0 の列で表す必要があります。 そして、このアドレスは値を読みだす際に必要なので、忘れてはいけません。

たとえば、すごく厚いメモ帳があったとしましょう。その各ページに 1 つずつ、値をメモして使う場

合を考えてみてください。この例だと、メモリーのアドレスに相当するのは、メモ帳のページ番号です。何ページ目に何の値をメモしたかを覚えておかないと、メモした値を読み出す時に困りますね。それと同じで、アドレスを覚えておかないと、メモリー内にたくさん並ぶ値の中で、どれが目的の値かわからなくなります。ただ、アドレスは無味乾燥な番号なので、人間が覚えておくのは大変です。



■ メモリーへの値のメモを、簡単にできる「 変数 」

■「 変数 」は、アドレスの代わりに名前を付けられ、１ と ０ への変換も気にせず使える

でも、心配しなくても大丈夫です。 私たちがプログラム内に、前回のように人間にわかりやすい形で計算式を書けるのは、 言語処理系というものが、CPU とのやり取りを仲介してくれるからでしたね。 **メモリーとのやり取りも、私たちにとって使いやすいように、言語処理系が仲介**してくれます。 そのための便利なしくみが「 **変数** 」です。

変数には、覚えにくいアドレスではなく、**自由にわかりやすい名前(変数名)**をつけて、値をメモできます。 値を読みだすのも、変数名だけわかっていれば大丈夫です。 また、メモする値を、人間が１ と ０ になおす必要もありません。 そういった、メモリーとのやり取りに必要な作業は、背後で言語処理系が行ってくれます。便利ですね。

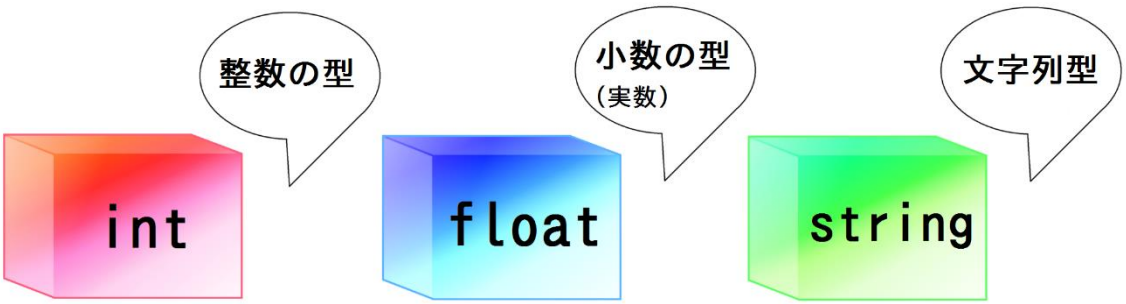
■ 実際に、変数を作ってみよう!

それでは、プログラムの中で、変数を用意してみましょう。 例として、**整数の値**をメモする、**「 memo 」という名前(変数名)**の変数を作ります。 それには、プログラムに、以下のように記述しましょう:

```
int memo ;
```

簡単ですね。ただし変数を作っただけなので、まだ実行しても何も表示されません。 このように変数を作る事を、**「 変数を宣言する 」**といいます。

さて、上のプログラムで、先頭の **「 int 」** という部分が気になると思います。 これは、メモする値が整数である事を意味しています。 ここで再び、前回も説明した**「 データ型 」**の登場です。 プログラム内のデータは、**「 整数 」**や**「 小数(実数) 」**、**「 文字列 」**などの種類に分けて扱われ、 その種類をデータ型と呼ぶのでしたね。 これらのデータ型にも名前があり、プログラム内でそれぞれ **「 int 」**、**「 float 」**、**「 string 」** と書きます。



上のプログラムのように、変数を宣言するときは、どういうデータ型の値をメモするのかを書く必要があります。 そうすると、**データ型ごとに適切な方法を言語処理系が選んで、値を自動的に 1 と 0 の列になおして、メモリーに記憶させてくれる**(※)のです。

(※ たとえば文字列の中の文字は、文字コードという対応表のようなものを使って 1 と 0 の列になおされます。 小数は、IEEE754 という規格に従って 1 と 0 の列になおされます。)

■ 変数に値をメモする = 代入する

それでは、先ほど宣言した変数に値をメモし、それを画面に表示してみましょう:

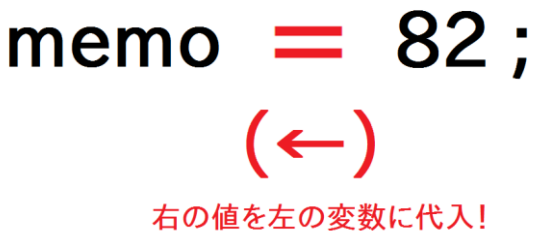
```
// 整数をメモする変数「 memo 」を宣言
int memo ;

// memo に「 82 」という値をメモ（代入）
memo = 82 ;

// memo にメモしている値を表示
print ( memo ) ;
```

「 // 」よりも後はコメントであり、プログラム実行の際は無視される事に注意してください。実際に実行すると、以下のように表示されます：

```
82
```

このように、print 関数のカッコの中に変数名を書くと、その値を画面に表示してくれます。さて、上のプログラムにおいて、変数に値をメモしているのは「 memo = 82 ; 」の行です。このように、変数に値をメモする事を、正確な言い方では「 代入する 」と言います。

ここで登場している「 = 」記号は、「 右の値を左の変数に代入する 」という意味のものであり、「 代入演算子 」と呼ばれます。算数や数学の「 = 」記号とは少し意味が違うので、注意が必要です。恐らく、イメージ的には「 ← 」記号などの方が、本来の意味に近いでしょう。慣れるまでは「 = 」記号を見たら、頭の中で「 ← 」記号に置き換えるのも手かもしれません。



変数の宣言と、値の代入は、以下のように 1 行にまとめて書く事もできます：

```
// 変数「 memo 」を宣言して「 82 」を代入
int memo = 82 ;

// memo の値を表示
print ( memo ) ;
```

実行結果は先ほどと同じです。なお、「 = 」記号の右に、計算式を書く事もできます。そうすると、式

の計算結果の値が、変数に代入されます。

■ 変数の値を上書きする

変数は、値を代入した後に、何度でも上書きで代入しなおす事ができます:

```
// 変数「 memo 」を宣言して「 82 」を代入
int memo = 82 ;

// memo の値を表示
println ( memo ) ;

// memo に 100 を代入して値を上書き
memo = 100 ;

// memo の値を表示
println ( memo ) ;

// memo に 12345 を代入して値を上書き
memo = 12345 ;

// memo の値を表示
println ( memo ) ;
```

実際結果は以下の通りです:

```
82
100
12345
```

プログラムの処理は、上の行から下の行へと順番に実行される事を思い出すと、 確かに変数の値が、代入するたびに上書きされていっている事がわかりますね。

■ 変数を使って計算する

変数に代入した値を使って計算を行うには、計算式の中に、数値などを直接書くかわりに変数名

を書けば OK です。そうすると計算結果は、式の中の変数名の部分を、その変数の値で置きかえて計算したものになります：

```
// 変数「 memo 」を宣言して「 30000 」を代入
int memo = 30000 ;

// memo を使って計算
print ( memo + 2000 ) ;
```

実際結果は以下の通りです：

```
32000
```

ちゃんと、変数「 memo 」の値である 30000 に、2000 が足された結果が得られていますね。
　もちろん、変数を使った計算結果を、別の変数に代入する事もできます：

```
// 変数「 memo 」を宣言して「 30000 」を代入
int memo = 30000 ;

// memo を使った計算結果を「 memo2 」に代入
int memo2 = memo + 2000 ;

// memo2 の値を表示
print ( memo2 ) ;
```

実行結果は先ほどと同じ「 32000 」です。別の変数ではなく、変数「 memo 」を使った計算結果を、memo 自身に代入する事もできます：

```
// 変数「 memo 」を宣言して「 30000 」を代入
int memo = 30000 ;

// memo を使った計算結果を memo 自身に代入
memo = memo + 2000 ;

// memo の値を表示
print ( memo ) ;
```

これでも同じ結果が得られます:

```
32000
```

さて、先ほど『「 = 」記号の意味は、算数や数学のイコールとは違う 』と説明しましたが、この「 memo = memo + 2000 ; 」という行は、まさにそうですね。数学だと、こんな関係を満たす memo の値は存在しません。繰り返しになりますが、ここでの「 = 」記号はあくまでも、「 右の値を左の変数に代入するもの 」です。上の行では、まず「 = 」の右にある式の値が「 32000 」と計算されてから、それがそのまま「 = 」の左の変数「 memo 」に代入されるだけです。等式を解いてくれるわけではないので、注意が必要です。

■ データ型の変換

■ いろいろなデータ型の変数を使ってみよう!

ここまでは、整数のデータ型である int 型を例に使ってきましたが、小数(実数)の float 型や、文字列の string 型でも試してみましょう:

```
// 整数の変数「 suuji (数字) 」
int suuji = 12345 ;

// 小数 (実数) の変数「 ensyuu (円周率) 」
float ensyuu = 3.14 ;

// 文字列の変数「 aisatsu (あいさつ) 」
string aisatsu = "おはよう" ;

// 変数の値を、行ごとに表示
println ( suuji ) ;
println ( ensyuu ) ;
println ( aisatsu ) ;
```

実行結果は以下の通りです:

```
12345
3.14
おはよう
```

それぞれ、ちゃんと値を代入できていますね。

■ 変数に、別のデータ型の値を代入すると、変数のデータ型に変換される。変数のデータ型は変わらない!

上の内容を読んでいて、恐らく多くの人が、次のような事を思い浮べたのではないでしょうか:

「 整数（int 型 ）の変数『 suuji 』に、『 3.14 』って代入したら一体どうなるんだろう? 」

その答えは、**「 強引に整数になおされてから代入される 」**です。試してみましょう:

```
// 整数の変数「 suuji（数字） 」を宣言
int suuji ;

// suuji に小数（実数）の値を代入
suuji = 3.14 ;

// suuji の値を表示
print ( suuji ) ;
```

実行結果は以下の通りです:

```
3
```

上の通り、**「 3.14 」の小数点以下が切り捨てられ、強引に整数に変換されてから、整数の変数「 suuji 」に代入された**事がわかります。

このように変数には、宣言したときのデータ型（ 上の例では int 型、つまり整数 ）の値しか入れられません。つまり**変数のデータ型は、宣言時に一度決めると、ずっと変わらない**のです。このようなルールを**「 静的型付け 」**と呼びます。VCSSL は、C 言語や C++などと同じく、静的型付けのプログラミング言語です。

一方で、上のような代入を行うと、変数「 suuji 」のデータ型が小数(実数)型に変わるプログラミング言語もあります。そのようなルールは**「 動的型付け 」**と呼びます。静的型付けと動的型付けには、それぞれ有利不利があり、使う人の好みも分かれます。

■ データ型の変換は、なるべく頑張ってくれるが、無理な場合もある

さて、どうやっても変数のデータ型には変換できない値を代入したら、どうなるでしょうか？ たとえば以下のように、**整数の変数「 suuji 」に "あいうえお" と代入する**場合などです：

```
// 整数の変数「 suuji（数字） 」を宣言
int suuji ;

// suuji に文字列を代入
suuji = ”あいうえお” ;

// suuji の値を表示
print ( suuji ) ;
```

実行してみましょう：

```
（青い画面）
------------------------------------------------
RUNTIME ERROR - 実行時エラー / 1

 [ LINE - 場所 ] Test : LINE 5 行目付近
[ CODE - 内容 ] suuji = ”あいうえお”
 [ INFO - 詳細 ] int 型へ変換できません。（ ”あいうえお” ）
```

おっと、青い画面になりました。という事は、**エラーの発生**です。 このように、無理な変換を行うとエラーになり、プログラムの実行が止まってしまいます。

ただし、一見すると常に無理そうな「 文字列を整数に変換 」という処理ですが、一応は頑張って試してくれます。 そして、文字列の中の文字が、すべて数字の場合などは、うまく変換できて代入が成功します。 **上のプログラムで "あいうえお" を "12345" と書きかえて実行すると、エラーは発生せず、実行結果もふつうに「 12345 」という表示されます。**

■ 式の中などで、変数を一時的に別のデータ型に変換する「 キャスト 」

ところで、変数の値を、一時的に別のデータ型と見なして、式の中などで使いたい場合もあります。よくある例は、**整数の変数同士で割り算を行う場合**でしょう。 前回説明した通り、整数同士の割り算では、結果も整数になってしまいます。 小数点のある計算結果を得たければ、分母か分子が小数

(実数)である必要があります。

　でも、式の中で、整数の変数の名前に小数点を付けても、小数には変換されません。「 そんな名前の変数はありません 」とエラーになるだけです。 そのような変換をするには、式の中で「 (float)suuji 」のように、変数名の前に、変換したいデータ型をカッコではさんで書けば OK です。これで、変数「 suuji 」がどのようなデータ型でも、float 型つまり小数になります:

```
// 整数の変数「 bunbo（分子） 」
int bunsi = 20 ;

// 整数の変数「 bunbo（分母） 」
int bunbo = 8 ;

// そのままの割り算結果を表示
println ( bunsi / bunbo ) ;

// 小数に変換した割り算結果を表示
println ( (float)bunsi / (float)bunbo ) ;
```

実行結果は以下の通りです:

```
2
2.5
```

上の行は、そのまま整数同士の割り算を行った結果で、やはり整数です。 下の行は、変数名に「 (float) 」を付けて小数に変換した値を使って、割り算を行った結果で、小数の結果が得られています。このようなデータの変換をキャストと呼びます。

■ 配列

■ いくつもの値をメモしておける「 配列 」

　これまで扱った変数は、1 個につき 1 つの値しかメモできません。でも、プログラムによっては、非常に多くの値をメモしておきたい場合などもあります。そのような場合、必要な個数だけ変数をいちいち宣言するのは面倒ですし、個数によっては人力では無理でしょう。

　そんなときに便利なのが、1 個につき複数の値をメモしておける、「 配列 」というものです。配

列がメモできる値の数を、その配列の「 要素数 」と呼びます。配列の宣言は、これまでのふつうの変数の宣言と基本は同じで、違うのは**変数名(配列名)の後に [ ] 記号で囲って要素数を書く**という点です:

```
// 要素数 100 の配列「 memo 」を宣言
int memo[100] ;
```

これで要素数が「 100 」、つまり 100 個の値をメモしておける配列「 memo 」が作れました。先頭の「 int 」は、データ型が int 型、つまり整数の値をメモする事を意味しています。これはふつうの変数と同じですね。

配列の中の値には、「 インデックス 」という番号が付いています。配列に値を代入したり、読みだす際には、**配列名の後に [ ] 記号で囲って、インデックスを指定**します。たとえば、上で作った配列において、インデックス [1] 番と [2] 番に別の値をメモしてみましょう:

```
// 要素数 100 の配列「 memo 」を宣言
int memo[100] ;

// インデックス [ 1 ] 番に値を代入
memo[1] = 11111;

// インデックス [ 2 ] 番に値を代入
memo[2] = 22222;

// [1] 番と [2] 番の値を行ごとに表示
println ( memo[1] ) ;
println ( memo[2] ) ;
```

実行結果は以下の通りです:

```
11111
22222
```

1 個の配列に、ちゃんと 2 個の値をメモしておく事ができました。このように、配列はインデックスを変える事で、あたかも複数の変数を宣言したかのように扱う事ができます。

■（重要）配列インデックスの落とし穴！　［ 要素数 ］番目は存在しない

配列のインデックスには、慣れるまで注意が必要なルールがあります。それは、**インデックスは［１］ではなく［０］から割り振られる**という事です。そのため、**使えるインデックスの上限は［ 要素数-1 ］番目であり、［ 要素数 ］番目は存在しない**のです。たとえば上で宣言した配列「 moji 」は要素数が 100 個なので、使えるインデックスは［0］番目から［99］番目までです。［100］番目を使うとエラーになります。C 言語などでもそうなので、注意しましょう。

条件分岐と条件式

今回は、場合によって分ける「条件分岐」を扱います。

■ 状況に応じて処理を切りかえたい場面もある

■ 例：変数 x の値がマイナスならゼロにしたい

これまでの例では、プログラムに書いた処理内容が、すべて実行されてきました。 でも、実際に色々な作業を自動化しようと思うと、そのプログラム内には、「 状況に応じて処理を行ったり、逆に行わなかったり、または別の処理を行ってほしい 」という部分もでてきます。

単純な例として、「 もし変数 x（エックス）の値がマイナスなら、ゼロにしたい 」という場面を考えてみてください。実際のプログラム内で、こういう状況はよくあります。 こういう場合は、プログラムをどのように書けばよいでしょうか？

■ とりあえず x がマイナスのところで、ゼロにする処理を書いてみる

たとえば、まずは直球でそのまま、以下のように書くのはどうでしょうか。

```
// 整数の変数「 x 」を宣言
int x ;

// x にマイナスの値を代入
x = -100 ;

// x にゼロを代入（ ※ x がマイナスの場合に必要 ）
x = 0 ;

// x の値を表示（ 実際の場面では x を使った計算などを実行 ）
print ( x ) ;
```

最後の部分は、実用的なプログラムでは x を使って計算などの処理を行うのでしょうが、ここではあくまでも簡単な例なので、x の値を表示するだけにしています。

さて上のプログラムでは、x の値がマイナスになっているところで、素直に **x の値をゼロにする処理「 x＝0； 」**を書いています。実際に実行してみると:

```
0
```

この通り、あたりまえですが、0 が表示されます。コンピューターにとっては、確かにマイナスだった x の値をゼロにしたわけなので、目的通りの事をやっています。ある意味で一つの正解かもしれません。**x がマイナスの状況に限っては。**

■ x の値がマイナスじゃなかったら？ ゼロにしてはいけない！

でも、上のプログラムでは、x の値をプラスの値に書きかえても、常に「 x＝0； 」の行のところで 0 にされてしまいます。それはまずいですね。あくまでも「 マイナスの場合はゼロにしたい 」というだけなので、そうでない場合には、ゼロにしてはいけません。

では「 人間がプログラムを読んで、x の値がマイナスでない場合には、『 x = 0 ; 』の行を消してしまう 」というのは、どうでしょうか？

```
// 整数の変数「 x 」を宣言
int x ;

// x にプラスの値を代入
x = 100 ;

// ※ x がプラスなので、 x にゼロを代入する行は消した

// x の値を表示（ 実際の場面では x を使った計算などを実行 ）
print ( x ) ;
```

強引な方法ですが、確かに実行すると x の値がそのまま変化せず

```
0
```

と表示されます。まあ、これも x がプラスの状況に限っては、間違いではない…かもしれません。でもこういう解決法だと、x の値が変わるたびに、一緒に「 x = 0 ; 」の行も付けたり消したりと、手動で書きかえないといけません。それはちょっと面倒ですね。うっかり作業ミスが入り込む危険性もあります。

■ 処理の切りかえを自動化する「 条件分岐（ if 文 ）」

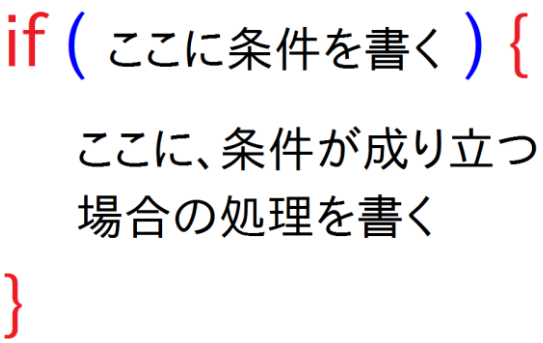
■ 上の例を、条件分岐を使って自動化してみる

という事で、上の例での「 x をゼロにする処理 」のように、状況に応じて処理を付けたり消したり（ 行ったり行わなかったり ）という事を自動化できると便利ですね。これを可能にしてくれるのが「 条件分岐 」です。詳しい説明は後にして、まずは先ほどの例で使ってみましょう：

```
// 整数の変数「 x 」を宣言
int x ;

// x にマイナスの値を代入
x = -100 ;

// 条件分岐 ( ※ 条件 x < 0 が成り立つ場合だけ実行される )
if ( x < 0 ) {

    // x にゼロを代入
    x = 0 ;
}

// x の値を表示 ( 実際の場面では x を使った計算などを実行 )
print ( x ) ;
```

このとおりです。上のプログラムで、「 if (x < 0) { 」と、「 } 」記号とで囲まれた部分が、条件分岐です。この条件分岐は if というキーワードではじまるので、**if 文(イフぶん)**とも呼ばれます。if というのは英語で「 もし 」という意味で、「 もし ～ だったら 」の「 もし 」です。

この条件分岐では、if のあとの **丸いカッコ (…)** の中に条件を書きます。そして、その条件が成り立つ場合のみ、**波っぽいカッコ { … }** の中の処理が実行されます。処理は複数行でもかまいません。



条件が成り立たない場合、{ … } の中の処理はすべて無視されます。なお、{ … } の中身を微妙に右にずらして書いてあるのは、単純に読みやすさのための慣習で、動作には関係ありません。(※「 Tab 」キーを押すとずらせます。)

いろいろな条件の書き方は後で詳しく説明しますが、ここで条件にした（ｘ＜０）は、「ｘの値がゼロより小さければ」という意味で、つまり「ｘの値がマイナスだったら」という意味と同じです。よって上の例の条件分岐は、**「もしｘの値がマイナスならば、『ｘ＝０；』の処理を実行する」**という内容になっているはずです。それでは実行してみましょう：

```
0
```

この通り、確かに上のプログラムではｘの値が「-100」つまりマイナスなので、「ｘ＝０；」の処理が実行され、その結果ｘの値がゼロになった事がわかります。

では ｘ がプラスならどうなるでしょう？ 上のプログラムで**「ｘ＝-100；」の行からマイナスをとって、「ｘ＝100；」に書きかえて再実行**してみてください。 結果は以下の通りです：

```
100
```

このとおり、x が最初の値のままになりました。 つまり今度は ｘ がマイナスではないので、条件分岐の条件 ｘ＜０ が成り立たずに、「ｘ＝０；」の処理が無視されたわけです。

以上のように条件分岐を使って、ちゃんと最初の目的 **「もし変数 ｘ の値がマイナスなら、ゼロにしたい」**という事を自動化できました。

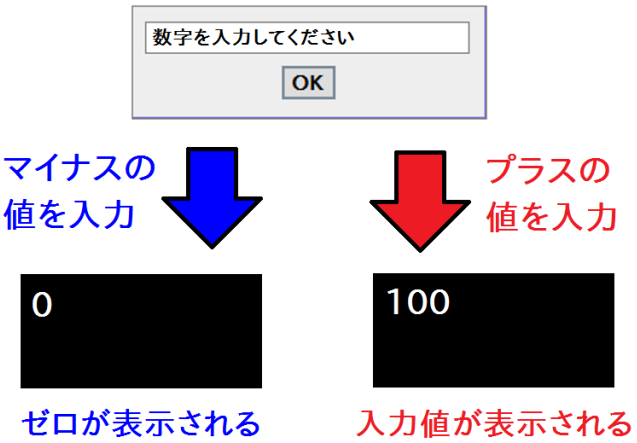
■ 実行するまで値がわからない場合などは、条件分岐でしか対応できない

ところで、今回の最初で行ったように、状況に応じて手動でプログラムを書きかえる方法は、そもそも不可能な場合もあります。というのも、実際の場面では**「プログラムが実行されるまで、どういう値になるかわからない」**という状況もよくあるからです。 典型的な例としては、ユーザーに値を入力してもらう場合などがあげられるでしょう。以下のような場合です：

```
// 整数の変数「 x 」を宣言
int x ;

// x の値を、ユーザーに入力してもらう
x = input ( ”数字を入力してください” ) ;

// 条件分岐 ( ※ 条件 x < 0 が成り立つ場合だけ実行される )
if ( x < 0 ) {

    // x に値「 0 」を代入
    x = 0 ;
}

// x の値を表示 ( 実際の場面では x を使った計算などを実行 )
print ( x ) ;
```

上のプログラムで使っている **input 関数** は初めて登場しましたが、これはユーザーに値を入力してもらう関数で、このようにイコール記号(代入演算子)の右側に書きます。すると左の変数に、ユーザーが入力した値が代入されます。例のとおり、カッコ内は無しでも OK です。

このプログラムを実行すると、**まず入力ウィンドウが表示され、そこで入力した値が変数 x に代入されます。**その後の処理は、先ほどの例と同じです。もし入力された x の値がマイナスなら、条件分岐の中の処理が実行され、その結果 x がゼロになって「 0 」と表示されます。そうでなければ、ユーザーが入力したままの x の値が表示されます。

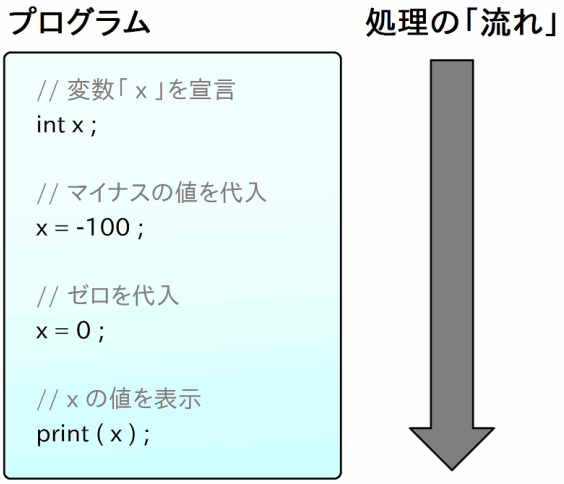


この例では、**x の値がユーザーの気分次第で変わるので、プログラムを書いている時点では、その値は全くわかりません。**なので、最初に行ったような、プログラムを手動で書きかえる方法ではどうやっても対応できませんが、このように条件分岐を使えば対応できます。

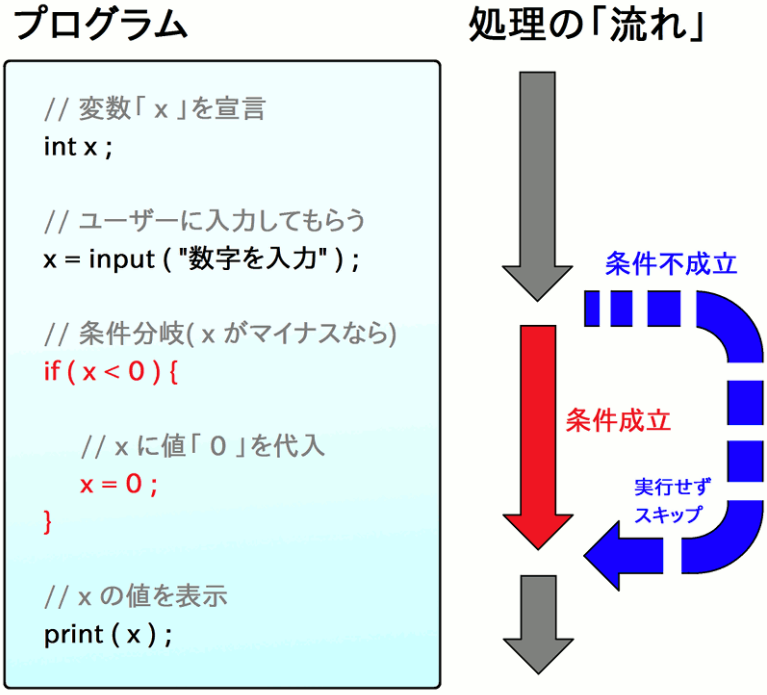
■ フローチャートと複雑な条件分岐

■ 処理の流れを整理するためには、フローチャートを描こう!

ところで、前回までの内容では、プログラムは 1 行ずつ順に実行されるものでした。つまりプログラムの **処理の「 流れ 」**は、「 最初の行から最後の行へ 」つまり **「 上から下へ 」という一本道**だったわけです。



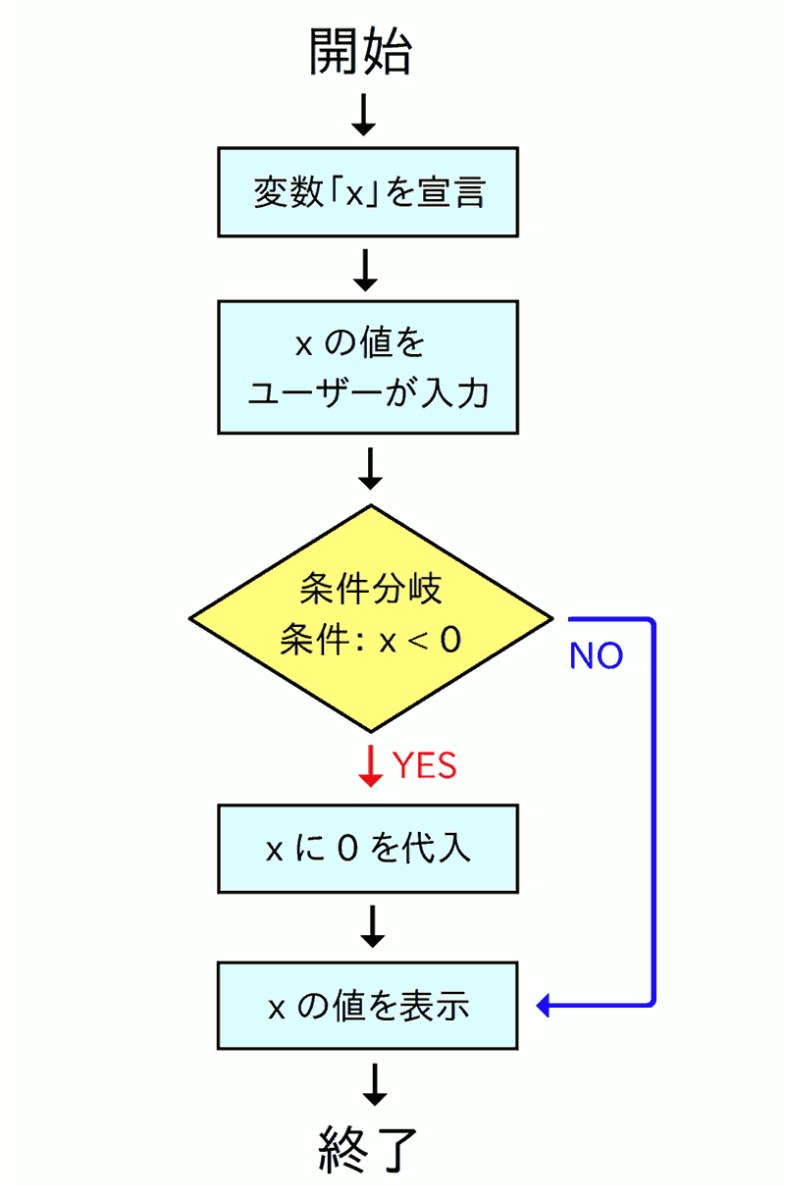
しかし今回、プログラムの中で **「 状況に応じて、実行せずに飛ばされる部分 」**というものが出てきました。その結果、処理の流れはもはや一本道ではなく、以下のように枝(えだ)わかれしたものになってしまいます:



この「 処理の流れの枝わかれ 」というものは、 実はなかなかのクセものです。 ここまでの例のように単純な場合はいいのですが、 実際の場面では、条件分岐の中にさらに条件分岐が必要になったり、 さらにその中にまた条件分岐が入ってしまう場合もあります。 こうなると、処理の流れは網のように枝わかれし、条件も複雑に組み合わさってきます。

一方で、ふつう人間が日常生活を送る上で、同時に想定している状況の数は、多い時でも 3 個くらいですよね。 それに対して、現実的なプログラムの中であり得る「 処理の枝 」や状況の数はもっと多くなるので、慣れないと頭がこんがらがってしまったり、 想定しきれていない条件パターンの「 枝 」が残って、プログラムが想定外の動きをしてしまったりします。バグですね。

そこで、処理の流れを図に描いて整理すると便利です。 その図とは一般に「 フローチャート 」と呼ばれるものです。「 フロー 」とは英語で「 流れ 」という意味です。 実際に先ほどの例で描いてみましょう:



こんな感じです。ちゃんとしたフローチャートの描き方には、いくつかの規格や種類があるのですが（ 枠線や矢印、図形の使い方など）、**そういった細かいルールをプログラミング言語と一緒に覚えるのは大変なので、ここでは思い切り簡略化**しています。描き方はシンプルで、以下の 3 つのルールに基づいています：

- 処理の流れは、矢印で表す。
- ふつうの処理は、長方形で表す。
- 条件分岐は、ひし形で表す。

では、上のフローチャートを読んでみましょう。**基本的には「 あみだくじ 」みたいな要領**で、上から下に矢印をたどって読んでいきます。 長方形はふつうの処理なので、ふつうに読み進みます。まず変数 x を宣言して、続いて、それにユーザーからの入力値を代入するわけですね。

その後にひし形があります。これは条件分岐で、ひし形の中に条件が書かれています。**ここからの処理の流れは枝分かれし、条件が成り立つ場合は「 YES 」の矢印に、成り立たない場合は「 NO 」の矢印に進みます。**前者の場合には x をゼロにする処理が続きますが、それは後者の場合ではスキップされる（ とばされる ）事がわかりますね。

その後、枝分かれした処理は合流し、x の値を表示して、プログラム終了です。どうでしょうか。処理の流れがわかりやすくなった感じがしませんか？

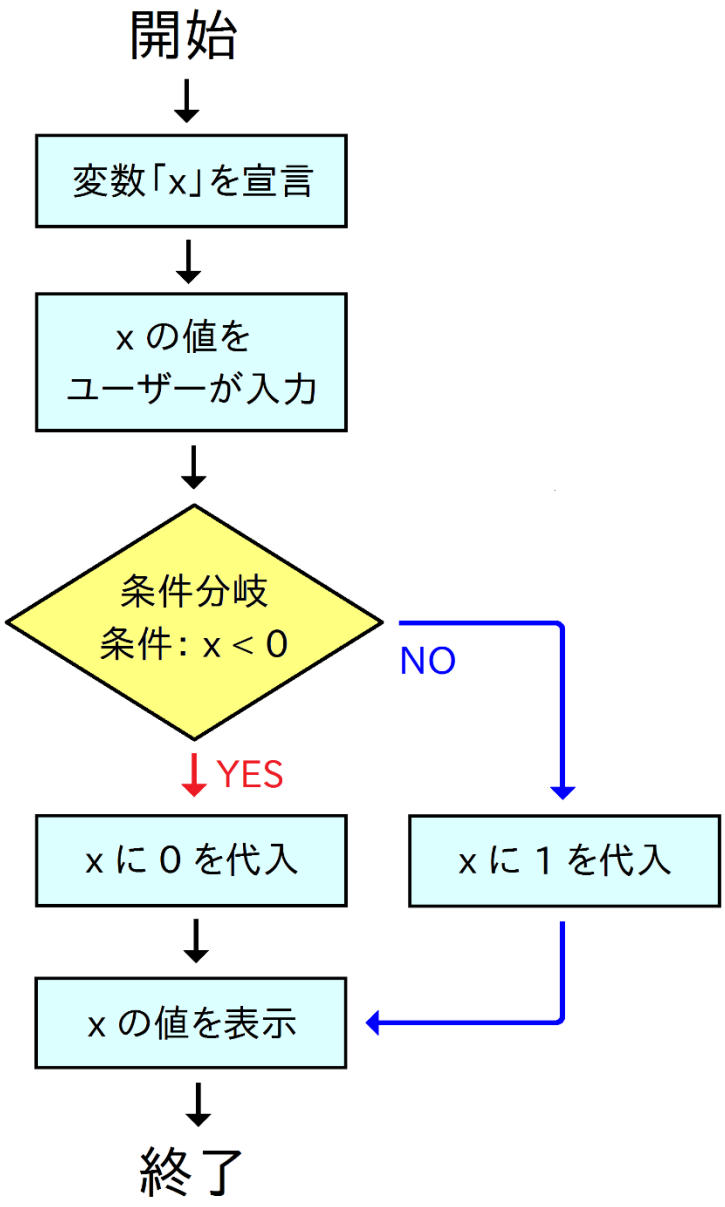
■ 条件が成り立たなかった場合は、別の処理をする（ else 文 ）

フローチャートを読めるようになったところで、もう少し複雑な条件分岐を使っていきましょう。条件分岐で、条件が成り立たなかった場合には、別の処理を行いたいという場合もよくあります。それには以下のように、「 if 文 」の後に「 else 文(エルスぶん) 」を書けば OK です：

```
// 整数の変数「 x 」を宣言
int x ;

// x の値を、ユーザーに入力してもらう
x = input ( ”数字を入力してください” ) ;

// if 文 ( ※ 条件 x < 0 が成り立つ場合だけ実行される )
if ( x < 0 ) {

    // x に値「 0 」を代入
    x = 0 ;
}

// else 文 ( ※ 上の if 文が成り立たなかった場合に実行される )
else {

    // x に値「 1 」を代入
    x = 1 ;
}

// x の値を表示 ( 実際の場面では x を使った計算などを実行 )
print ( x ) ;
```

else 文の { … } の中に書いた処理は、すぐ上の if 文の条件が成り立たなかった場合にだけ、実行されます。逆に if 文が条件成立して実行された場合、else 文の中の処理は実行されません。else は英語で「 そうでなければ ～ 」といった意味で、これは if が「 もし 」という意味だったのと、わかりやすく対応しています。

つまり上のプログラムでは、「 もし(if)変数 x がマイナスなら 0 を代入し、そうでなければ(else)変数 x に 1 を代入する 」という処理になります。フローチャートは以下の通りです：



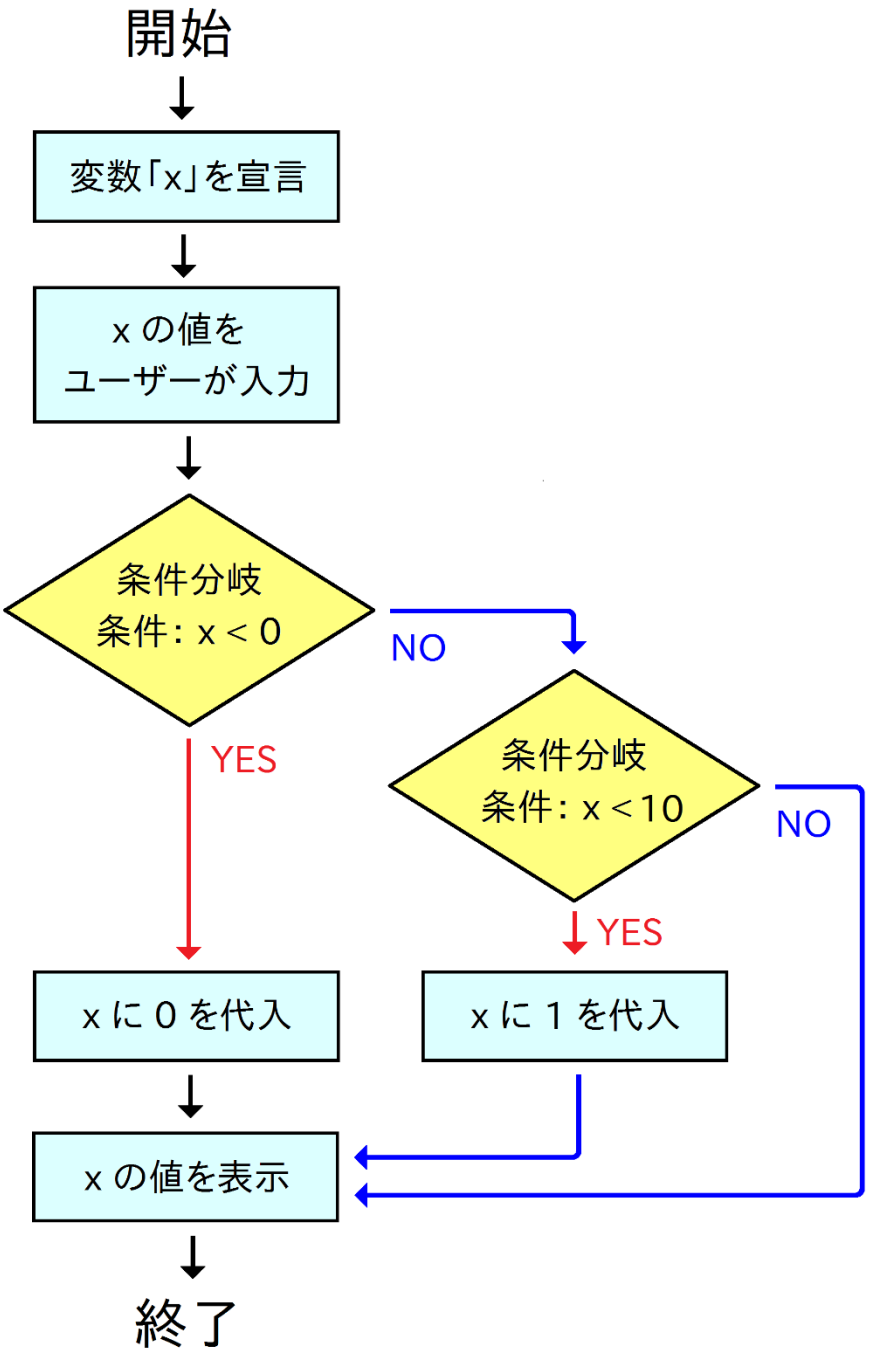
条件分岐（ ひし形 ）の「 NO 」の矢印の先に、x を 1 にする処理が追加されましたね。そこが else 文の部分です。このプログラムでは、x の値は先と同様、ユーザーの入力値が代入されるようになっています。実行していろいろな値を入力し、フローチャートと見比べてみましょう。

■ else 文の中に、さらに条件を設定する

上の例では、if 文の条件が成り立たなければ、その時点で無条件で else 文の中が実行されます。でも、この else 文の実行に、さらに条件を設けたいという場合もあります。そのような場合は以下のように、else 文のあとに if 文をくっつけます：

```
// 整数の変数「 x 」を宣言
int x ;

// x の値を、ユーザーに入力してもらう
x = input（ ”数字を入力してください”）;

// if 文（ ※ 条件 x ＜ 0 が成り立つ場合だけ実行される ）
if ( x ＜ 0 ) {

    // x に値「 0 」を代入
    x = 0 ;
}

// ※ 上の if 文が不成立で、さらに x ＜ 10 が成り立つ場合に実行
else if ( x ＜ 10 ) {

    // x に値「 1 」を代入
    x = 1 ;
}

// x の値を表示（ 実際の場面では x を使った計算などを実行 ）
print ( x ) ;
```

こうすると、まず最初の if 文が条件不成立の時点で、処理は else 文のところまで進みます。そこでさらに条件 x < 10（ x が 10 未満 ）が成り立つ場合のみ、その後の { … } の中の処理が実行されます。文章だとややこしいですが、フローチャートだと以下の通りです：



やはりフローチャートだと見やすいですね。これも、実際に実行していろいろな値を入力してみて、フローチャートと見比べてみてください。この「 else if 」は何個もつなげたり、最後に条件なしの else 文をつなげる事もできます。

なお、if 文を別の if 文や else 文の { ... } の中に入れるなど、入れ子にする事もできます。

■ いろいろな条件式

さて、最後です。いろいろな場合において、if 文のカッコ内の条件を、どのように書けばよいかを、一通り抑えておきましょう。

■ 値の大小比較や一致判定などを行う「 比較演算子 」

ここまで用いてきた「 x < 10 」は、「 x が 10 よりも小さいか？ 」という条件でしたね。この「 < 」は、一般に「 比較演算子 」と呼ばれるものの一つです。名前は難しそうですが、要するに「 比較を行う記号 」です。比較演算子は、別名として「関係演算子」と呼ばれたりもしますが、名前が似ていて別種の「条件演算子」と紛らわしいので、ここでは比較演算子と呼びましょう。

比較演算子には、全部で以下のようなものがあります：

比較演算子	意味
<	左より右が大きいか？
>	左より右が小さいか？
<=	左より右が大きいか、もしくは等しいか？
>=	左より右が小さいか、もしくは等しいか？
==	左と右は等しいか？（※ イコール記号が 2 個並んでいる事に注意！）
!=	左と右は異なるか？

どれも、YES なら条件成立になります。たとえば、**x の値がちょうど 10 である時に処理を実行したいなら、if (x == 10) { … } のように書けばよい**わけです。

なお、**「 <= 」と「 >= 」において、イコール記号は必ず右側に付ける必要がある**事に注意してください。順序が逆だと使用できません。慣れないうちはハマりやすいので、気を付けましょう。

■ 複数の条件を組み合わせる「 論理演算子 」

続いて、複数の条件を組み合わせて判定する方法についてです。

たとえば、「 x が 10 よりも小さく、かつ、y も 20 より小さい 」という条件のときだけ、処理を行いたいとしましょう。この場合、普通に if 文を 2 個使って、以下のように書く事もできます：

```
...
if ( x < 10 ) {
     if ( y < 20 ) {
          ...
          ( 行いたい処理 )
          ...
     }
}
...
```

でも、実は以下のように 1 つの if 文で書く事もできます:

```
...
if ( x < 10 && y < 20 ) {
     ...
     ( 行いたい処理 )
     ...
}
...
```

これで、全く同じ処理を実現できます。シンプルになりましたね。

ここで使用している「 && 」は一般に**論理演算子**と呼ばれるものの一つで、上の例のように、複数の条件を組み合わせてまとめるのに便利です。具体的に「 && 」は、**左右の条件が両方とも成立している場合**だけ、全体的に条件成立となります。つまりどちらか片方でも不成立であれば、全体的には不成立です。

論理演算子には、以下のようなものがあります:

論理演算子	意味
&&	左と右の条件が、両方とも成り立っているか？（論理積）
||	左と右の条件が、どちらか一方でも成り立っているか？（論理和）
!	右の条件の成立/不成立を反転する（否定）

「 || 」は「 && 」の兄弟のようなもので、たとえば **if (x < 10 || y < 20) { … }** と書けば、「 x が 10 よりも小さいか、もしくは y が 20 より小さいか 」の**どちらか片方だけでも成立していれば、その時点で if の中の処理が実行**されます。両方成立していても実行されます。

「！」はちょっとパターンが違って、たとえば **if(!(x<10)){…}** と書けば、「 x が 10 よりも小さく**ない**」場合に、if 文の中の処理が実行されるようになります。他の論理演算子と組み合わせて使うと結構便利なのですが、あまり考えずに場当たり的に使いすぎると、あとで条件を読みかえした時に、「 旗揚げゲーム 」のように紛らわしくなる事もあります。もっとも、それは他の論理演算子や比較演算子にも言える事です。

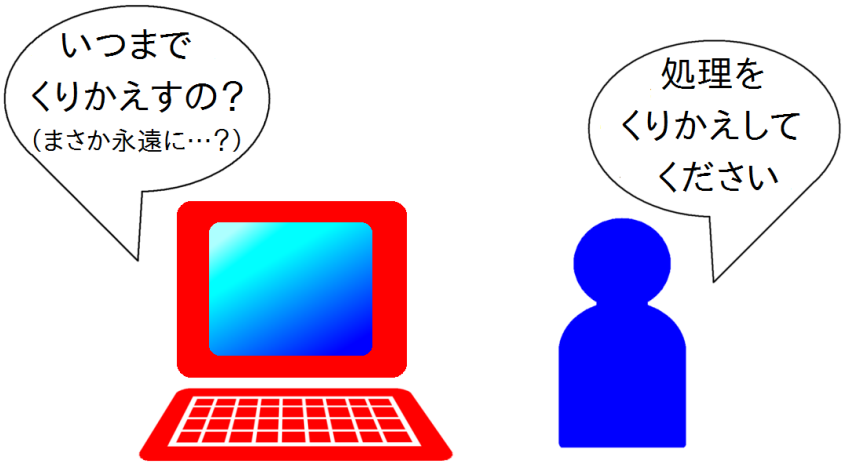
そのような場合、条件全体を図などに書いてよく見直すと、実はもっとシンプルに書ける、という可能性もあります。また、else if 文が何個も連なる場合などには、それらの順序を少し入れかえると、条件が途端にシンプルになったりもします。**解読しづらい複雑な記述内容の条件は、たくさんバグを生み出しかねない強敵**です。読みやすくなるように、うまく工夫しましょう。

くりかえし処理

前回扱った条件分岐とあわせて、実際のプログラミングに欠かせないのが、処理をくりかえす方法です。今回は、実際に繰り返し処理を行うための「 while 文 」と「 for 文 」を使ってみましょう。

■ くりかえし処理の基本となる、2 つのパターン

一言で「 くりかえし処理 」といっても、そのパターンは一通りではありません。たとえば、もしあなたが「 この作業をくりかえしてください 」と だけ 指示されても、「 一体いつまで…？ 」と困ってしまいますよね。



そう、実用的なくりかえし処理には、ふつう 「 いつまでくりかえすのか 」 という指示もいっしょに必要になります。

この指示は、たとえば以下のようなパターンが考えられます：

(1) 条件が成り立っている間だけくりかえす
(2) 決まった回数だけくりかえす

たとえば「 ストップと言われていない間はくりかえす 」は (1)、「 10 回くりかえす 」は (2) にあたります。もっと別のパターンも考えられますが、この 2 つはプログラミングにおいて特に基本となるので、必ず押さえておきたいものです。

それでは、実際にくりかえし処理を行ってみましょう。

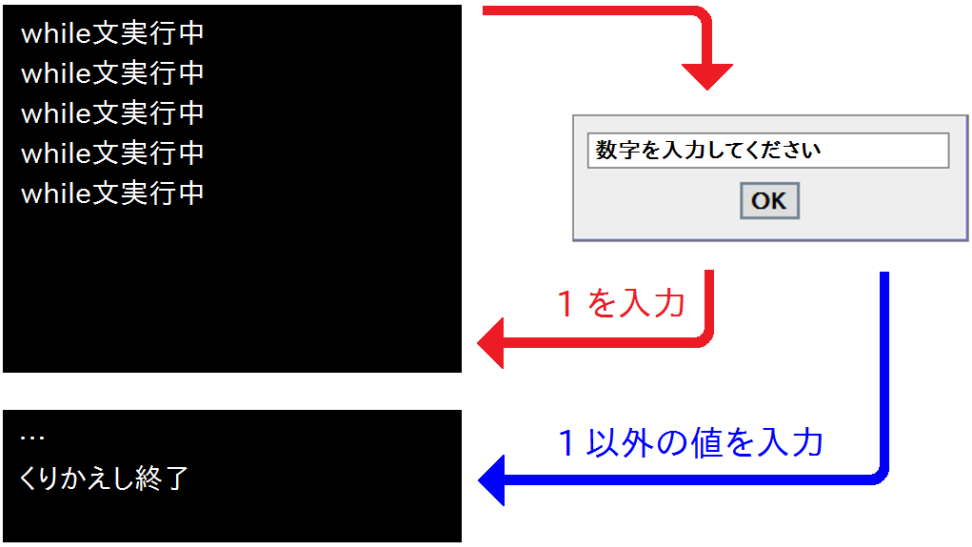
■ 条件が成り立っている間だけ処理をくりかえす

■ while 文（ ホワイルぶん ）

まずは、「 条件が成り立っている間 」だけ処理をくりかえす方法についてです。これには、「 while 文（ ホワイルぶん ）」というものを使用します。while 文は、前回扱った条件分岐の if 文とよく似ています。特に書き方は以下の例の通り、if の代わりに while と書くだけです：

```
// 整数の変数「 x 」を宣言し、最初に値「 1 」を代入
int x = 1;

// 条件 x == 1 が成り立つ間だけくりかえされる（ == は等号）
while ( x == 1 ) {

    // くりかえし処理の中で、毎回メッセージを表示
    println ( "while 文実行中" ) ;

    // x の値を、ユーザーに入力してもらう
    x = input ( "数字を入力してください" ) ;
}

// くりかえし処理が終わったら、最後にメッセージを表示
println ( "くりかえし終了" ) ;
```

内容の説明はすぐ後でするとして、とりあえず実行してみましょう。このプログラムを実行すると、以下のようにまず画面に「 while 文実行中 」と書かれて、続いて値を入力するウィンドウが表示されます。



ここで入力ウィンドウに「 1 」と答えると、また画面に「 while 文実行中 」と追記され、再び入力ウィンドウが表示されます。 **「 1 」と答えている間は、ずっと同じ事がくりかえされ続けます。** 逆に「 1 」以外の数字を入力すると、「 くりかえし終了 」というメッセージが書かれて終わります。ちゃんと、くりかえし処理になっていますね。

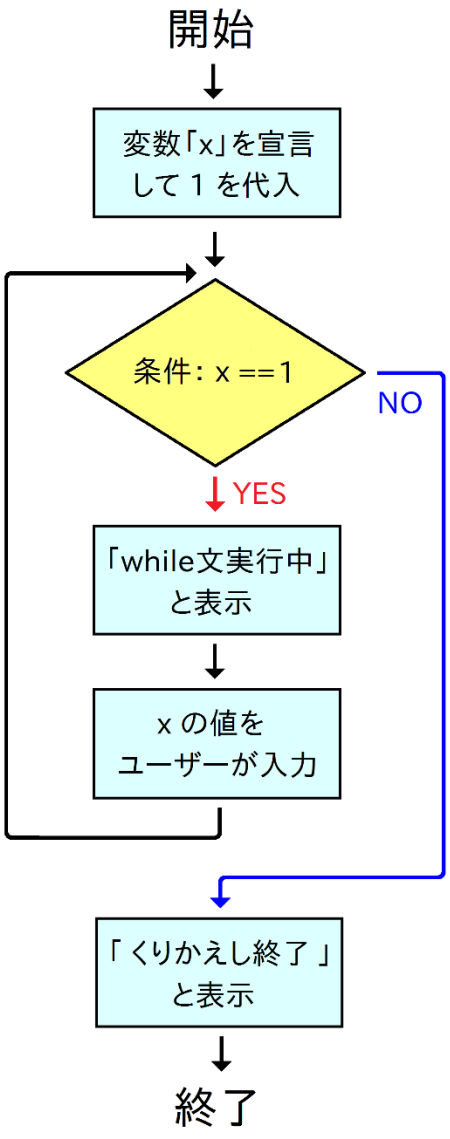
さて、プログラム内容の説明です。「 while 」の部分に注目してください。そのあとの **丸いカッコ (…)** の中には、if 文と同じように、条件式を書きます。上の例での条件式 (x == 1) は、「 x の値が 1 と等しければ 」という意味です。イコール記号「 = 」は、1 個ではなく 2 個続いている事に注意しましょう。if 文の場合は、ここに書いた条件が成り立っている場合だけ、 **波っぽいカッコ { … }** の中の処理が実行されるのでしたね。この事は、while 文でもまったく同じです。

```
while ( ここに条件を書く ) {
    ここに、条件が成り立つ間に
    くりかえしたい処理を書く
}
```

ただし if 文では、{ … } の中の処理は、1 回だけ実行されて終わりでした。それに対して while 文では、**{ … } の中の処理が終わったタイミングで、処理はまた「 while 」と書かれているところまで戻ってきます。そして、条件が成り立っているかどうかが再び確認されます。もし成り立っていれば、また { … } の中の処理が実行されます。これがくりかえされます。** 逆に、条件が成り立っていなければ、{ … } の中は実行されず、処理はもう「 while 」のところにも戻りません。くりかえ

し処理はもう終わりで、ふつうにその下の処理が実行されていきます。

文章だけで説明するのもややこしいですね。そこで上のプログラムにおける処理の流れを、前回登場したフローチャートにしてみると、以下のようになります：



上のフローチャートの真ん中～左にかけて、**「 わっか 」のように一周回って閉じている部分が、くりかえし処理の部分**になっています。条件の判断はひし形の部分で行われ、**条件が成り立っている間は YES の矢印のほうに処理が流れて、このわっか(ループ)をぐるぐると回り続ける** 事になります。確かにくりかえしですね。そして条件が成り立たなくなれば、処理はわっかを「 脱出 」して NO の矢印へ進み、くりかえし終了です。

上のフローチャートを見ながら処理を追いかけると、確かにユーザーが「 1 」を入力し続けている間は、処理がくりかえされますね。もう一度プログラムを実行し、見比べてみてください。

■ くりかえしを途中で終わらせる「 break 文(ブレイクぶん) 」

上の例では、くりかえしを続けるか終わるかを、while 文やのカッコ内に条件式として書きました。でも、その条件式とは無関係に、強制的にくりかえしを終わらせる方法も存在します。 それが **break 文(ブレイクぶん)**です。break 文の使い方は単純で、**プログラム内で「 break 」と書くだけで、その行が実行された時点でくりかえしが終了します。**
(※ くりかえしが何重にもなっている場合は、一番内側のくりかえしが終了します。)

試してみましょう。これまでは説明しませんでしたが、実は条件式に「 true 」と書くと、「 常に成り立つ条件 」になります。つまり **while (true) { … } と書けば、いつまでもくりかえされる処理**になります。これを break 文で終わらせてみましょう:

```
// くりかえしの条件が常に成り立つ処理
while ( true ) {

    // くりかえし処理の中で、毎回メッセージを表示
    println ( ”くりかえし中” ) ;

    // 変数 x を宣言し、値をユーザーに入力してもらう
    int x = input ( ”数字を入力してください” ) ;

    // もし x の値が 1 以外なら、くりかえしを終わる
    if ( x != 1 ) {
        break ;
    }
}

// くりかえし処理が終わったら、最後にメッセージを表示
println ( ”くりかえし終了” ) ;
```

if 文の中で使っている (i != 1) は、「 i が 1 以外の場合 」という条件です。この条件が成り立つ場合に、if 文の { … } 内に書かれた **break 文が実行され、くりかえしが終わる**というわけです。このプログラムの実行結果は、先ほどの例とまったく同じです。「 x が 1 の間だけくりかえす 」という事と、「 x が 1 でなければくりかえしを終わる 」という事は同じだからです。

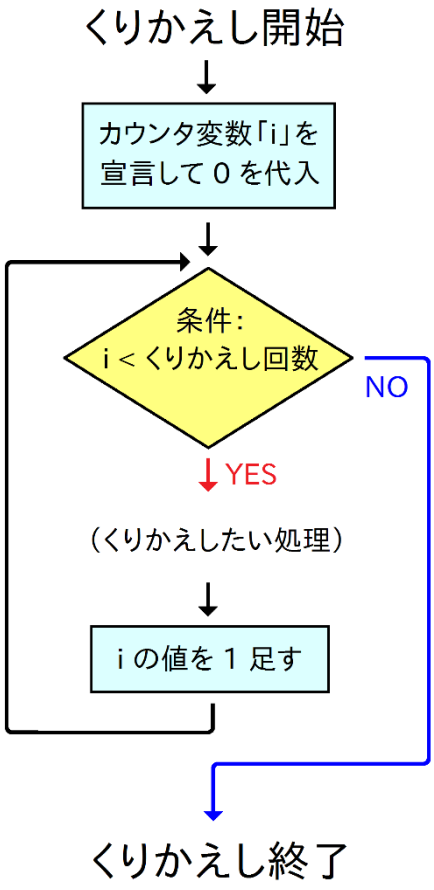
■ 決まった回数だけ処理をくりかえす

■ 回数が決まったくりかえしには、「 カウンタ変数 」で回数を数えれば OK!

続いて、「 決まった回数 」だけ処理をくりかえす方法についてです。実はこれは、条件が成り立っている間のくりかえしを使って実現できます。つまり、すでに扱った while 文でもできるのです。具体的には、回数を数えるための変数を用意して、以下のように使います。この変数は一般に「 カウンタ変数 」と呼ばれます。

- 整数型の変数（カウンタ変数）を用意し、「 0 」を代入しておく。
- 処理を 1 回くりかえすごとに、カウンタ変数に 1 ずつ足していく。
- カウンタ変数の値が、くりかえし回数より少ない間だけ、処理をくりかえす。

文章だけだと分かりづらいので、フローチャートにしてみましょう。以下の通りです：



実際にノートなどに、カウンタ変数の値を書きかえながら、上のフローチャートの処理をなぞってみてください。決まった回数だけくりかえされる仕組みのイメージがつかめると思います。

■ まずは、while 文を使って行ってみる

上のフローチャートに書いた処理は、すでに扱った while 文でも実現できます。ちょうどいい練習にもなるので、まずは while 文で行ってみましょう。プログラムは以下の通りです：

```
// 整数型のカウンタ変数 「 i 」を宣言し、値「 0 」を代入
int i = 0 ;

// カウンタ変数が 10 未満（ 10 は含まない）の間くりかえす
while ( i < 10 ) {

    // くりかえし処理の中で、毎回カウンタ変数の値を表示
    println ( i ) ;

    // 毎回のくりかえしの最後に、カウンタ変数に 1 を加算
    i = i + 1 ;
}

// くりかえし処理が終わったら、最後にメッセージを表示
println ( ”くりかえし終了” ) ;
```

このプログラムの実行結果は、以下の通りです：

```
0
1
2
3
4
5
6
7
8
9
くりかえし終了
```

この通り処理がくりかえされ、その間、毎回カウンタ変数の値が表示されています。 表示されて

いるのは「 0 」から始まって「 9 」で終わっているので、ちょうど 10 回くりかえされた事がわかりますね。このようにカウンタ変数の値が「 0 」からスタートする場合、くりかえしの最後でのカウンタ変数の値は、**「 くりかえし回数 - 1 」**になる事に、慣れるまで注意が必要です。

最後のくりかえしでのカウンタ変数の値は「 くりかえし回数 」に一致するほうが、すっきりするかもしれません。その場合はカウンタ変数を値「 1 」からスタートさせて、while 文の条件で「 i <= 10 」のように「 <= 」記号を使えば OK です。でも、くりかえし処理は配列と組み合わせて使う事が多く、**配列の要素のカウントも「 0 」から「 要素数 - 1 」まで**なので、くりかえしも 0 からスタートする方が、処理で好都合な場合が多いです。ぜひ慣れましょう。

■ こういう場合には for 文(フォーぶん)を使うと便利!

さて、上のように決まった回数のくりかえし処理を while 文で行うと、**カウンタ変数の宣言と加算のために、プログラム内で 2 行使う**必要があります。でも、決まった回数のくりかえし処理は非常によく使うので、**できればもっと短くまとめたい**ところです。

また、別の人がプログラムを読んで、くりかえし処理の流れを追いかける際など、**カウンタ変数のスタートの値や加算処理は、くりかえしの条件とワンセット**で把握する必要があります。場合によっては、カウンタ変数が中途半端な値からスタートしたり、毎回加算する数が 1 ではない場合もよくあるからです。その点でも、これらの処理は一か所にまとめたいところです。

そこで便利なのが**「 for 文(フォーぶん)」**です。for 文なら、これらを 1 つにまとめて書くことができます。先ほどの while 文を使ったプログラムと全く同じ処理を、for 文を使って書きなおすと、以下のようになります:

```
// カウンタ変数「 i 」が 10 未満 ( 10 は含まず) の間くりかえす
for ( int i = 0 ; i < 10 ; i = i + 1 ) {

    // くりかえし処理の中で、毎回カウンタ変数の値を表示
    println ( i ) ;
}

// くりかえし処理が終わったら、最後にメッセージを表示
println ( "くりかえし終了" ) ;
```

ずいぶん短くまとまりましたね。上のプログラム内で、**「 for 」**というキーワードで始まっているところが for 文です。「 for 」のすぐあとのカッコ内は **(… ; … ; …)** の形になっていて、**「 ; 」記号で区切って 3 つの式が書かれてる** 事に注目しましょう。これら 3 つの式は、それぞれ以下のような役割をもっています:

```
for (初期化; 条件; 更新処理) {
    ここに、条件が成り立つ間に
    くりかえしたい処理を書く
}
```

・ **左の式（初期化）：**

ここは、くりかえしに入る前、最初に 1 回だけ実行されます。カウンタ変数の宣言や、スタート値の代入（初期化）などを書きます。実際に、上のプログラム内に書いた「 int i = 0 」では、カウンタ変数「 i 」を宣言し、スタート値を「 0 」に設定しています。

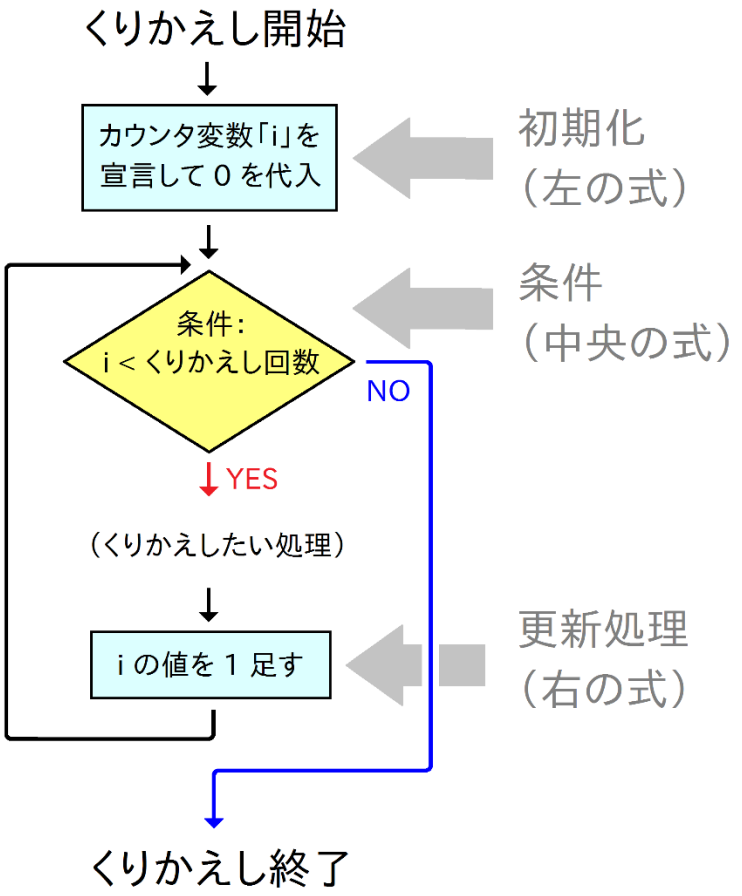
・ **中央の式（条件）：**

ここには while 文のカッコ内と同様に、処理をくりかえす条件式を書きます。上のプログラム内の処理「 i < 10 」は、while 文のプログラムでも使った通り、「 カウンタ変数の値が 10 より小さい間だけ処理をくりかえす 」という意味でしたね。

・ **右の式（更新処理）：**

ここに書いた内容は、くりかえし処理が 1 周終わるごとに、毎回実行されます。カウンタ変数の加算処理などを書きます。上のプログラム内の処理「 i = i + 1 」では、毎回カウンタ変数の値を 1 ずつ足していくという内容になります。

文章の説明だとなかなか複雑そうですが、フローチャートにすると以下の通りです：



このようにフロー自体は、普通に while 文でカウンタ変数を使った場合の、一定回数くりかえしの処理と変わりませんね。**ようするに、先ほどの while 文のプログラムで別々に書いていた、カウンタ変数の宣言と加算処理を、for 文だと条件式の両どなりに押し込めて書けるというだけ**です。なので、プログラムの実行結果も以下の通り、先ほどの while 文を使った例とまったく同じになります：

```
0
1
2
3
4
5
6
7
8
9
くりかえし終了
```

この通り、確かにまったく同じ結果になりましたね。

■ for 文と while 文は、どちらでも同じ処理を書けるものの、用途に応じて使い分けると読みやすくなる

実は for 文のカッコ内（…；…；…）の左側の式は、必ずしも「カウンタ変数の初期化処理を書かなければいけない」というわけでは**ありません。**同様に右側の式も、「 カウンタ変数の更新処理を書かなければいけない 」と決まっているわけではありません。ルール上は、前者はくりかえしの前、後者は 1 周ごとに毎回実行する処理であれば、なんでも書く事ができます。実は左右になにも書かなくても OK で、そうすると while 文と同じ、単なる条件くりかえしの処理になります。

for 文を使ったくりかえし処理は、すべて while 文でも行えますし、逆に while 文と同じくりかえし処理も for 文で行えます。でも、カウンタ変数の操作のように、**くりかえしの条件に関わる「 動き 」を、条件式の両どなりに書けるという事に、for 文を使う一つの利点がある**わけです。程度問題で人にもよりますが、あまりくりかえしの条件と無関係な処理が、for 文のカッコ内に書かれていると、読みづらいと感じる人もいるかもしれません。読みやすさを考えて、for 文と while 文をうまく使いわけましょう。

■ くりかえし処理に使うと便利な演算子

さて、ここまで例に使ってきたプログラムの通り、くりかえし処理の中では、「 なにかの変数の値を書きかえる 」という処理がよく登場します。カウンタ変数の加算などはいい例ですね。実はこういった処理は、短く書ける便利な演算子(記号)がいくつかあります。

■ インクリメント演算子とデクリメント演算子

たとえば、これまでの例ではカウンタ変数「 i 」に 1 を加算するのに、「 i = i + 1 」と書いてきました。これは、足し算の記号である「 + 」演算子と、代入の記号である「 = 」演算子を組み合わせて、**「 いまの『 i 』の値に 1 をたして、その結果で『 i 』を書きかえる 」** という処理を実現していたわけです。

でも、実はこれと同じ処理は、足し算の記号を 2 個並べた「 ++ 」という演算子だけで行えます。これは**インクリメント演算子と呼ばれ、「 ++ i 」や 「 i ++ 」のように、変数の前や後に書いて使います。** 前後のどちらに書くかは、式の中にめり込ませて使った場合に違いが出るのですが、単体で使う場合はどちらも「 i = i + 1 」と同じ結果になります。使ってみましょう:

```
// カウンタ変数「 i 」が 10 未満 ( 10 は含まず) の間くりかえす
for ( int i = 0 ; i < 10 ; i ++ ) {

    // くりかえし処理の中で、毎回カウンタ変数の値を表示
    println ( i ) ;
}

// くりかえし処理が終わったら、最後にメッセージを表示
println ( "くりかえし終了" ) ;
```

for 文のカッコ内が、より短くまとまりましたね。実行結果はこれまでの例と同じです:

```
0
1
2
3
4
5
6
7
8
9
くりかえし終了
```

このようにカウンタ変数を 1 ずつ加算する for 文では、インクリメント演算子を使って書くのが一般的です。

なお、インクリメント演算子の逆バージョンとして、変数を 1 だけ少ない値で書きかえる、つまり「 i = i - 1 」と同じ処理を行う「 --（マイナス記号が 2 個連続）」という演算子もあります。それはデクリメント演算子と呼ばれ、「 i -- 」のように書いて使います。これもくりかえし処理の中でよく用いられます。

■「 2 ずつ加算したい場合 」などに便利な、複合代入演算子

インクリメント演算子では、変数の値を 1 しか加算できないですが、1 以外の値だけ加算したい場合もよくあります。たとえば、for 文のカウンタ変数を 2 ずつ加算したい場合などです。このような場合、for 文のカッコ内の右の式に、そのまま「 i = i + 2 」と書いてもいいのですが、そのかわりに「 i += 2 」と書いても OK です。試してみましょう:

```
// カウンタ変数「 i 」が 10 未満（ 10 は含まず）の間くりかえす
// ただし、カウンタ変数は 2 ずつ加算する
for ( int i = 0 ; i < 10 ; i += 2 ) {

    // くりかえし処理の中で、毎回カウンタ変数の値を表示
    println ( i ) ;
}

// くりかえし処理が終わったら、最後にメッセージを表示
println ( "くりかえし終了" ) ;
```

実行結果は以下の通りです:

```
0
2
4
6
8
くりかえし終了
```

ちゃんとカウンタ変数が 2 ずつ増えていますね。

　このように、**演算の記号（足し算や引き算などの記号）に「 = 」を続けて書くと、左辺の変数の値を、右辺の値で足したり引いたりした演算結果で書きかえる事ができます。**かけ算や割り算でも使えます。なお、この「 += 」のような記号は**複合代入演算子**と呼ばれます。

ファイルの読み書き

ここでは、ファイルの読み書きを行う方法について説明します。

■ データをずっと覚えておく部品「 ストレージ 」

■ 変数で使う「 メモリー 」は、あくまでも一時的なメモ用の部品

これまで、値をメモするのに使ってきた「 変数 」ですが、 その値を読んで使えるのは、プログラムの実行中だけです。 実行が終わって閉じた後に、**「 さっき実行したプログラムの変数『 x 』の値を読み出したいんだけど… 」**と思っても、ふつうは無理です。一応、実行終了直後なら、特殊な方法まで含めれば絶対無理とまでは言い切れないのですが（※セキュリティ面で問題になったりします）、別のプログラムを何個も実行したり、コンピューターを再起動した後になると、もう絶望的です。

なぜでしょうか。 ここで「 変数と配列 」の節の話を、少しふり返ってみましょう。 コンピューターには、値をメモするための部品として**「 メモリー 」**が搭載されているのでした。 変数は、そもそもメモリーを簡単に使うためのものでしたね。 実はこのメモリー自体が、**あくまでも「 コンピューターの動作中に、一時的に値を記憶する 」事を想定した部品**なのです。 なので変数も、やはり一時的なメモ用の機能であり、実行終了後に値を読み出すような事を想定した機能にはなっていないのです。

コンピューターのメモリー（メインメモリー）

なお、このメモリーは、少し専門的な呼び方では**「 主記憶装置 」**、英語で**「 Main Memory ； メインメモリー 」**と呼びます。他にも「なんとかメモリー」と呼ぶものは色々あるのですが、プログラミングで単にメモリーと言うと、ふつうはこのメインメモリーを指します。これまでの説明でもそうでしたね。

■ データをずっと覚えておく「 ストレージ 」はメモリーとは別の部品

さて、プログラム終了後や電源を切った後でも、ずっとデータを覚えておいてほしい場面も多いので、もちろんコンピューターにはそのための部品も入っています。それは一般に**「 ストレージ 」**などと呼ばれます。ストレージは英語で「 Storage 」と書き、これは日本語で「 貯蔵 」といった意味です。データの貯蔵庫ですね。こちらは、専門用語では**「 補助記憶装置 」**と呼ばれます。

データを保存する部品「ストレージ」

※ 製品にもよりますが、データ保存期間や寿命は永久ではありません。
重要なデータは、こまめにバックアップをとりましょう。

なお、ストレージは、後で説明するハードディスクという機械の略称として、「 ディスク 」と呼ぶ人も結構います。ただし、CD や DVD などもディスクと呼んだりするので、混同には注意が必要です。

ところで、「 データを覚えておく 」という目的は同じなのに、メモリーとストレージで、なぜわざわざ別々の部品が必要なのでしょうか？ それは、**一言で「 データを覚えておく 」と言っても、実際にそれが可能な電子部品や機械には様々なものがあり、それぞれ得意・不得意がある**ためです。コンピューターはそれらを適材適所で組み合わせて使っているので、メモリーとストレージが別々の部品としてあるのです。

■ メモリーには、読み書きのスピードや耐久性に優れたものが使われる

メモリー（メインメモリー）は、変数などのように、計算中の一時的なメモのために使われるため、とにかく**読み書きのスピードの速さが重要**です。なぜなら、コンピューターの頭脳である CPU が計算するスピードはとてつもなく速いので、 値をメモする事にもたついていたら、台無しだからです。また、それだけとてつもない速さでメモとして使われるという事は、**膨大な回数の読み書きに耐えられる耐久性**も必要になります。メモリーには、このような要求に適したものが使われます。

コンピューターのメモリー（メインメモリー）

たとえば 2019 年時点のコンピューターに搭載されているメモリーは、ふつう DRAM という半導体素子が使われています。これは確かに読み書きが高速で、使用回数に対する耐久性も非常に高いという特徴があります。また、実は CPU の中にも「 キャッシュ 」と呼ばれる小さなメモリーが入っているのですが、それにはさらに高速な SRAM というものが使われていたりもします。

■ ストレージには、価格あたりの容量に優れたものが使われる

しかし、DRAM や SRAM は電源を切るとデータが消えてしまう（ 揮発性 ）という性質があり、ストレージには不向きです。個人向けのコンピューターのストレージには、**「 電源を切っていても、ずっとデータが残る（ 不揮発性 ）」**事が重要です。また、コンピューターを長く使っているうちに、データがたまりすぎて足りなくなると困るので、**価格あたりの容量が大きい事も重要**です。ストレージには、このような要求に適したものが使われます。

データを保存する部品「ストレージ」

■ ストレージの中身

では、具体的にストレージの中身には、どんなものが使われるのでしょうか。

それは時代によって変化しますが、特に近年主流になりつつあるものとしては、NAND 型フラッシュメモリー（以下、NAND フラッシュ）が挙げられます。名前にメモリーと付いていますが、これまでの話におけるメモリー（メインメモリー）ではなく、ストレージ用に使われます。半導体素子の一種ですが、DRAM とは違い、電源を切ってもデータが消えないという特性を持っています。半面、読み書きのスピードや、書き換え回数の耐久性では、DRAM にはかないません。

NAND フラッシュと書くとずいぶん専門的な雰囲気がしますが、実はみなさんおなじみの、USB メモリーの中身です。カメラや携帯電話・スマートフォンなどに使う各種メモリーカードの中身も、ほぼ NAND フラッシュです。実はとても身近なんですね。と言っても、コンピューターのストレージ用には、USB メモリーなどを大量に常時接続して使うのは不便なので、「 ソリッドステートドライブ（ SSD ）」という箱のような形の部品にして使われます。中には NAND フラッシュのチップがたくさん搭載されていて、大容量を実現しています（NAND フラッシュ以外を用いた製品もあります）。

もう一つ、ストレージとしてよく使われるものに、「 ハードディスク 」というものが挙げられます。ハードディスクは結構昔から使われてきたもので、現在もかなり大容量の製品が安価に買えます。中身は、回転する磁気ディスクが入った精密機械です。そのため衝撃に注意する必要があったり、機械的な寿命などもあったりしますが、一方でデータの書き換え回数に関する耐久性の高さや、電源 OFF の条件下におけるデータ保持期間の長さなどの長所があります。

■ ストレージへの読み書きは「 ファイル 」という形で行う

■ ファイルは、データの場所を覚えなくても、「 名前 」をつけて簡単に扱える

さて、メモリー（ メインメモリー ）への値の読み書きを、簡単にしてくれるのが「 変数 」でしたね。それと同様に、ストレージへの読み書きを、人間にとってわかりやすくしてくれる、「 ファイル 」というしくみがあります。「 あります 」というか、みなさもよく知っているかもしれませんね。ここでいうファイルというのは、ふつうにコンピューターを使っているときにも登場する、あのファイルの事です。そう、たとえばみなさんがコンピューターで文章を書いてファイルに保存したり、開いたりしたとき、実はコンピューターはストレージにデータを読み書きしていたのです。

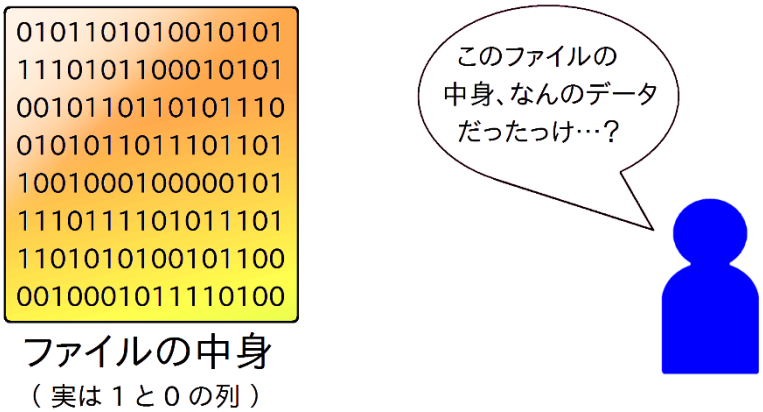
結局のところ、コンピューターは １ と ０ の世界で動作しています。ストレージも、ただ膨大な数の １ と ０ の列を記録しておける装置でしかありません。その中の、どこからどこまでの範囲に、ど

のデータが保存してあるかという事を、人間が直接管理するのは大変です。そこで「 ファイルシステム 」というものが、わかりやすく仲介してくれて、その結果、人間にはファイルという形で見えているのです。

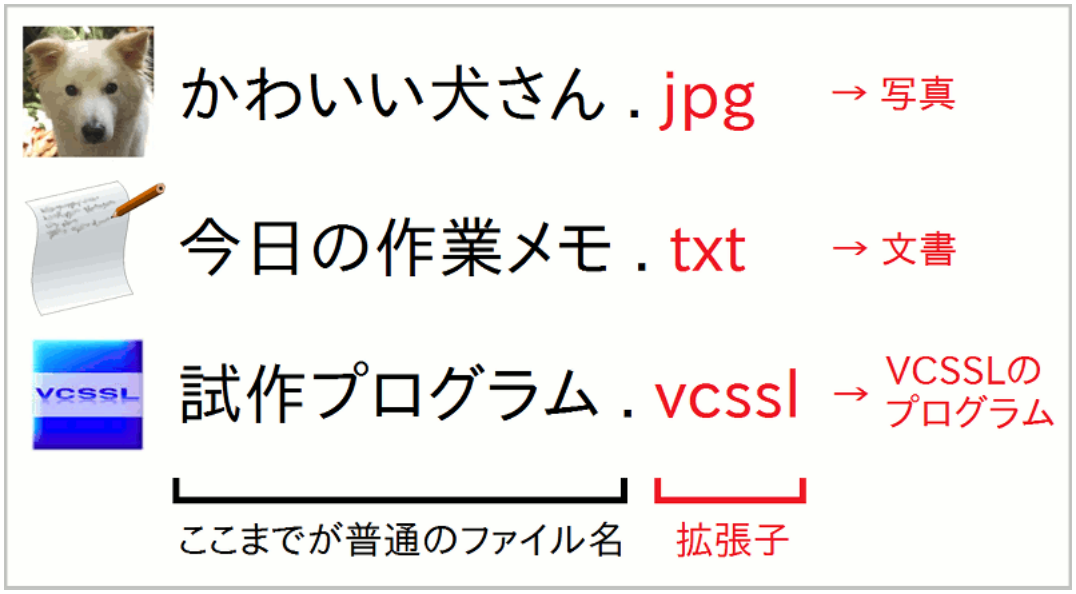
たとえば「 ストレージの 10111010 番地から 11101011 番地までにデータを書き込みます 」といったやり取りは、つらいですよね。ファイルは、こうした「 ストレージ内でのデータの場所 」を、人間にとってわかりやすい「 名前 」という形でおきかえてくれます。それによって、「『 今日の作業内容 』という名前のファイルにデータを書き込みます 」というように簡単にストレージを使えるわけです。これは、メモリーのアドレスと、変数の名前との関係に似ていますね。

■ 1 と 0 の列でしかないデータの「 意味 」を表すための「 拡張子 」

また、「 ストレージに記録されている 1 と 0 のデータが、何を意味しているものなのか 」というのも、大切な情報です。たとえば、テキストエディタで作ったファイルなら、「 テキストを 1 と 0 になおしたもの 」です。カメラで撮った写真なら、「 画像を 1 と 0 になおしたもの 」です。いや、もっとなにか別のものだったかもしれません。それを忘れると、どう解釈するべきか不明な、ただの 1 と 0 の暗号のようなものになってしまい、とても困ります。



そこで、ファイル名の末尾に、テキストなら「 .txt 」、カメラの写真の画像なら「 .jpg 」といったように、 中身を区別するためのキーワードを「 . 」記号区切りで付ける方式がよく使われます。 このキーワード部分は「 拡張子 」と呼ばれます。



この拡張子の部分を見れば、そのファイルの中身（1 と 0 の列）が何だったのかを判断できるわけです。

ここで、「 え?ファイル名にそんなもの付いていないぞ? 」と思った方も多いでしょう。 それもそのはず、拡張子を表示しない設定になっているコンピューターも多いのです。 それは恐らく、ファイル名を書きかえる際に、ミスして拡張子を消してしまって、何のファイルか不明になってしまう事を防ぐためでしょう。 でも、プログラミングなどでコンピューターを深く使いこなすには、拡張子が見えない事が逆に不便な場面もあります。 そこで、表示させるように設定を変える事もできます。「 拡張子 表示 」などのキーワードで Web 検索してみてください。

なお、OS の種類によっては拡張子に機能的な役割がなく、ただの目じるしで、本当に拡張子が付いていないファイルが普通に存在する場合もあります（ 拡張子以外にも、ファイルの中身を判断する方法があるためです ）。

■ プログラムからのファイルの読み書き

■ まずは書き込み

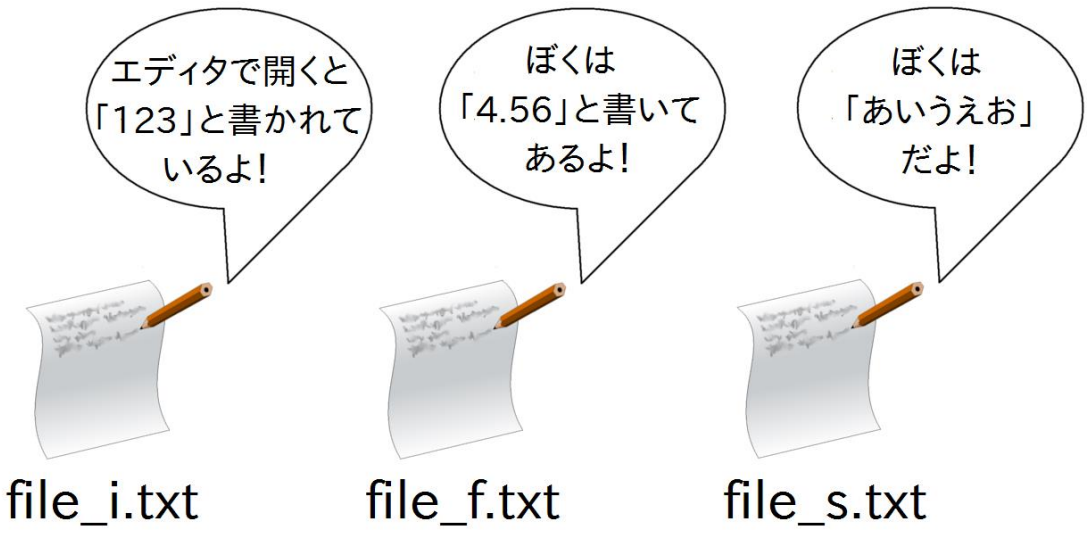
さて、いよいよ本題です。プログラムからファイルを読み書きしてみましょう。 そうすれば、データはストレージに保存されるので、プログラムの実行終了後も、コンピューターの電源を切った後も、ずっとデータを残しておく事ができます。

まずは、いちばん簡単な方法から試してみましょう。 変数の数だけファイルを作り、それぞれの変数の値を、それぞれのファイルに書き込んでみます ：

```
// 整数、小数（実数）、文字列型の変数を用意
int i = 123 ;
float f = 4.56 ;
string s = ”あいうえお” ;

// 変数の値を、それぞれ別のファイルに書き込む
save ( ”file_i.txt”, i ) ;    // 「 .txt 」は拡張子
save ( ”file_f.txt”, f ) ;
save ( ”file_s.txt”, s ) ;
```

ここで使っている「 save（ セーブ ）」関数は、データをファイルに書き込むための関数です。カッコ内には「 , 」区切りで、ファイル名と書き込み内容を指定します。上のように、書き込み内容を変数で指定する事も可能です。 その場合、変数の値が書き込まれます。

このプログラムを実行すると、「 file_i.txt 」と「 file_f.txt 」および「 file_s.txt 」という名前の 3 つのファイルが、プログラムと同じ場所に作られます。（ ※ ただし「 .txt 」の部分はテキストファイルを意味する拡張子なので、コンピューターの設定によっては表示されません。）



これらのファイルはただのテキストファイルなので、マウスでダブルクリックすると、テキストエディタなどで開けるはずです。 中には、それぞれの変数の値が、ちゃんと書き込まれています。 コンピューターの電源を切って入れなおしても、ちゃんと中身はそのまま残っています。

■ 続いて読み込み

今度は、ファイルから変数に、データを読み込んでみましょう ：

```
// ファイルからデータを読み込み、変数に代入する
int i = load ( ”file_i.txt” ) ;     //「 .txt 」は拡張子
float f = load ( ”file_f.txt” ) ;
string s = load ( ”file_s.txt” ) ;

// 変数の値を表示
println ( i ) ;
println ( f ) ;
println ( s ) ;
```

ここで使った「 load（ロード）」関数は、データをファイルから読み込むための関数です。このようにカッコ内にファイル名を書いて、「 = 」記号の右で使えば、左の変数にファイルの内容が代入されます。ただし、ファイル名はちゃんと拡張子まで含めて書く必要があります。ここではちゃんと拡張子「 .txt 」を付けています。
（ ※ コンピューターで拡張子が表示されない設定になっていて、読み込みたいファイルの拡張子がわからない場合は、 ファイルを右クリックして「 プロパティ 」などから調べる必要があります。詳しい方法は環境によります。）

このプログラムの実行結果は以下の通りです：

```
123
4.56
あいうえお
```

このように、3 つのファイルから 3 つの変数に、それぞれ値を読み込めていますね。
（※ ファイルの内容は文字列型のデータとして読み込まれ、変数に代入する時点で、その変数の型に変換されます。変換できない内容だった場合はエラーになります。）

■ 行単位でのファイルの読み書き

■ 1 行ごとの書き込み

上で扱った save 関数や load 関数は、ファイルの中身全体をまとめて読み書きします。でも、1 行ごとに読み書きしたいという場合もよくあります。たとえば、上で行ったように変数の値をファイルにメモしたい場合、1 行につき 1 つの値を書くようにすれば、ファイルを 1 個ですませる事もできますね。

その代わり、行単位の読み書きでは、以下のように少し手順が増えます：

(1) ファイルを開く
(2) 1 行ずつ内容を書いていく / 読んでいく
(3) ファイルを閉じる

ちょっとめんどうですが、便利なので試してみましょう。まずは書き込みです：

```
// 整数、小数（実数）、文字列型の変数を用意
int i = 123 ;
float f = 4.56 ;
string s = ”あいうえお” ;

// ファイルを書き込みモードで開き、ID 番号を変数に代入
int id = open ( ”file.txt”, ”w” );

// 開いたファイルに、行ごとに書き込む
writeln ( id, i ) ;     // 最初の行に i の値
writeln ( id, f ) ;     // 次の行に f の値
writeln ( id, s ) ;     // その次の行に s の値

// ファイルを閉じる
close ( id ) ;
```

詳しい説明の前に、まずは実行してみましょう。すると「 file.txt 」というファイルが作られ、それをテキストエディタで開くと：

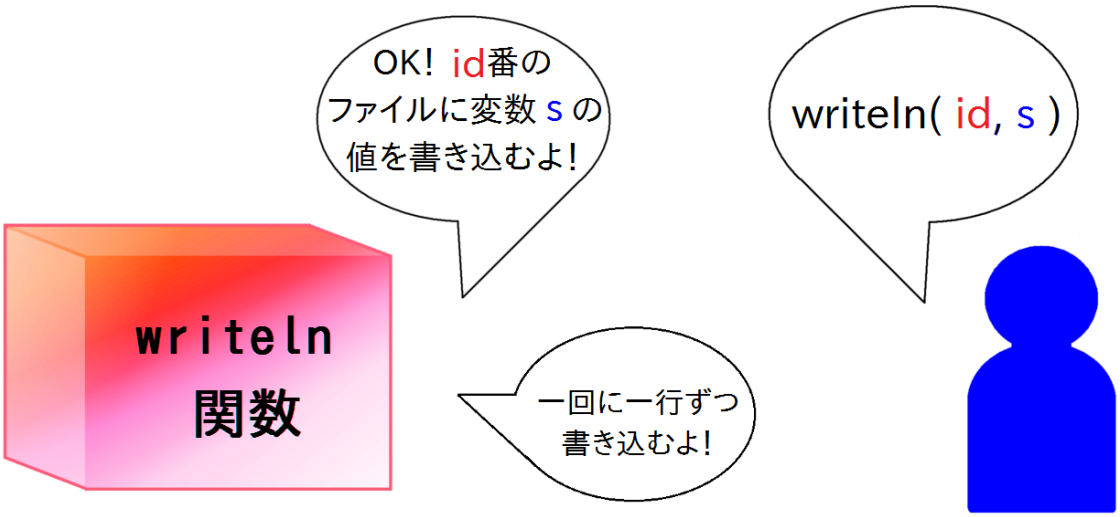
```
123
4.56
あいうえお
```

と書かれています。ちゃんと行ごとに書き込めていますね。

さて、順を追って説明しましょう。まず「 open(オープン) 」関数は、ファイルを開いてくれる関数です。「 開く 」と言っても、コンピューターの内部で開かれるだけなので、画面上で内容が表示されたりはしません。でも、開いたファイルに ID 番号を割りふって、「 = 」の左の変数(ここでは id)に代入してくれます。これは複数のファイルを区別するための、名簿(めいぼ)の番号のようなものです。

open 関数のカッコ内の最初はファイル名を書きます。その後に「 , 」記号で区切って「 "w" 」とありますが、これは日本語で「 書く 」を意味する英語「 write (ライト) 」の 1 文字目です。これにより、「 このファイルは書くために開きます 」とコンピューターに知らせています。書くために開いたファイルを読む事はできませんし、その逆もできないので、注意が必要です。

さて、いよいよ書き込みです。「 writeln (ライトライン) 」関数は、ファイル内容を 1 行だけ書き込んでくれる関数です。カッコ内の最初には、open 関数のところで変数に代入しておいた ID 番号を使います。これは、仮にファイルが複数あっても、どのファイルに書き込むのかわかるように知らせているわけです。その後は、書き込む行の内容を指定しています。ここでは変数の値を書き込んでもらっています。



最後の「 close（クローズ）」関数では、ファイルを閉じています。 この処理は、見た目では何も起こらないように感じますが、忘れるとファイルが最後まで書き込まれなかったり、 他にもいろいろな点でよくないので、忘れないようにしましょう。

■ 1 行ごとの読み込み

続いて、上で作ったファイルを 1 行ずつ読み込んでみましょう:

```
// ファイルを読み込みモードで開き、ID 番号を変数に代入
int id = open ( "file.txt", "r" );

// 開いたファイルから、行ごとに読み込む
int i = readln ( id ) ;        // 最初の行を i に代入
float f = readln ( id ) ;     // 次の行を f に代入
string s = readln ( id ) ;   // その次を s に代入

// ファイルを閉じる
close ( id ) ;

// 変数の値を表示
println ( i ) ;
println ( f ) ;
println ( s ) ;
```

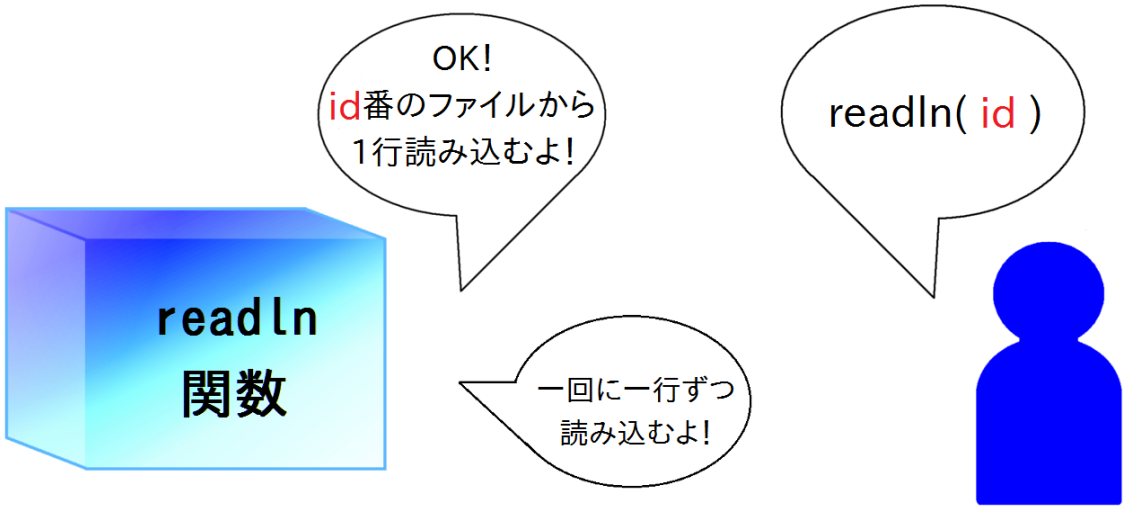
このプログラムの実行結果は以下の通りです:

```
123
4.56
あいうえお
```

このように、ファイル内容を 1 行ずつ変数に代入できている事がわかります。

最初に open 関数でファイルを開いている点は、先ほどの writeln 関数による書き込みと同じですね。でも、open 関数のカッコ内の 2 つめの内容が「 "r" 」になっている事に注意してください。これは日本語で「 読む 」を意味する英語「 read （ リード ）」の 1 文字目です。これにより、「 このファイルは読むために開きます 」とコンピューターに知らせています。 書き込みで使う「 "w" 」の逆バージョンです。

続いて読み込み処理です。load 関数はファイルの中身をまるごと変数に代入してくれましたが、上の例で使っている「 readln （ リードライン ）」関数は、 ファイルの内容を 1 行だけ読んで、それを「 = 」の左の変数に代入してくれます。 このように何度も続けて使うと、毎回 1 行ずつ、次の行へと読み進んでくれます。なのでここでは、変数「 i 」にファイルの 1 行目、「 f 」に 2 行目、「 s 」に 3 行目の内容が代入されるわけです。



最後に先ほどと同様 close 関数でファイルを閉じています。忘れずに閉じましょう。

■「 文字化け 」と文字コード

■ テキストファイルであっても、ストレージには 1 と 0 で記録

さて、最後におまけです。ファイルの読み書きでは、しばしば「 文字化け 」に出くわす事があります。文字化けというのは、「 あれ、ファイルを開いて読んだら、書かれているはずの内容じゃなくて、意味不明な内容になってる？ 」という現象です。Web ページなどでたまに見かけますね。特にあるコンピューターで書いたファイルを、OS の種類が違う別のコンピューターで読み込んだ場合などに、よく発生します。

文字化けは、なぜ発生するのでしょうか。それを詳しく理解するために、まず今回の序盤で説明した、ストレージとファイルの話をふり返ってみましょう。ストレージはあくまでも膨大な 1 と 0 の列を覚えておく電子部品で、それを人間にとって扱いやすい形で見せているのが、ファイルでしたね。つまり日本語で書かれたテキストファイルであろうと、ストレージには 1 と 0 の列になおして記録されているのです。

論より証拠、たとえば先ほど書き込んだファイル「 file.txt 」の中身は：

```
123
4.56
あいうえお
```

でしたが、これを 1 と 0 の列のまま読み込んで、表示してみましょう：

```
// ファイルをバイナリ（1 と 0 の）読み込みモードで開く
int id = open ( "file.txt", "rb" ) ;
// 1 バイト(8 桁)ずつの配列として読み込む
int byte[ ] = read( id ) ;
// ファイルを閉じる
close( id ) ;

// 読み込んだ内容を 1 と 0 の列で表示
int n = length(byte, 0);     // 配列の長さ（バイト数）
for (int i=0; i<n; i++) {
      println( bin( byte[i] ) );   // bin は 2 進数表示
     //println( hex( byte[i] ) );  // 16 進数はこちら
}
```

すこし難しくなってしまうので、プログラム内容の細かい説明はやめましょう。 詳しく読む必要はありません。重要なのは実行結果で、以下の内容が表示されます：

```
0b110001
0b110010
0b110011
0b1101
0b1010
0b110100
0b101110
0b110101
0b110110
0b1101
0b1010
0b10000010
0b10100000
0b10000010
0b10100010
0b10000010
0b10100100
0b10000010
0b10100110
0b10000010
0b10101000
0b1101
0b1010
```

（ ※ **色を付けてある部分は「 あ 」の文字を表しているデータです。**ただし内容は環境によって異なります。 なお、各行の頭の「 0b 」の部分は、1 と 0 のデータに付く目じるしのようなものなので、無視してください。 ）

これが、ファイル「 file.txt 」の中身を、 1 と 0 のまま読んだ中身です。つまりストレージに記録されているデータそのものです。 みなさんが、ふつうにコンピューター上でファイルを開くと、画面にテキストが表示されますね。 それは、内部でこのような 1 と 0 の列から、今度はテキストへと逆向きにおきかえて、なおされるからです。

■ 文字コード

そして、このようなテキストから 1 と 0 へのおきかえ方に、「 文字コード 」と呼ばれる複数の方式があるのです。たとえば「 あ 」という文字は、「 UTF-8 」という文字コードでは「 11100011 10000001 10000010 」におきかえられます。 一方で、「 Shift_JIS 」という文字コードでは「 10000010 10100000 」になります。そう、同じ文字でも、文字コードによって別の 1 と 0 の列になってしまうのです。

では、たとえば UTF-8 の文字コードを使って書き込まれたファイルを、もし Shift_JIS が使われていると思って読み込んだらどうなるでしょうか？ なんとなく想像できるでしょうが、やってみましょう:

```
// UTF-8 でファイルに「 あ 」と書き込む
int id1 = open ( "a.txt", "w", "UTF-8" ) ;
writeln ( id1, "あ" ) ;
close ( id1 ) ;

// それを Shift_JIS で読み込む
int id2 = open ( "a.txt", "r", "Shift_JIS" ) ;
string s = readln ( id2 );
close ( id2 );

// 結果を表示
println ( s ) ;
```

このように、実は open 関数のカッコ内の最後に、読み書きに使う文字コードを指定する事ができます。実行すると:

```
縺?
```

このように、書き込んだ「 あ 」とは全然違う文字として読み込まれた事がわかります。これが文字化けです。そこで、open 関数のカッコ内で指定している文字コードを、 読み書きで両方とも "UTF-8" か "Shift_JIS" でそろえれば:

```
あ
```

このように、書き込んだ通り「 あ 」として読み込めます。 つまり、書き込みと読み込みの文字コード

は、同じものを使う必要があるわけです。

■ 標準的な文字コードや改行コードは、環境ごとに違う！

さて、open 関数のカッコ内に文字コードを指定しなかった場合、 標準的な文字コードが自動で使われます。 これは、他のプログラミング言語や、ソフトウェアなどでもよくある事です。 でも、実は**この「 標準的な文字コード 」というのが、OS の種類や時代、日本語・英語バージョンなどの環境によって、違ったりする**のです。 これが、ファイルを別のコンピューターで開くと、よく文字化けしてしまう原因です。 そんなときは、ファイルを書き込むのに使われた**「 文字コード探し 」**をする事になります。 ここで扱った UTF-8 と Shift_JIS は特によく使われるものなので、この 2 つはぜひ覚えておきましょう。

また、文字コードと並んでやっかいなのが、**改行（ 行のおりかえし ）**を意味する 1 と 0 の列です。 先ほど 1 と 0 で表示してみたファイル内容で、3 回ほど「 0b1101 0b1010 」という部分がありましたが、 この部分がそうで、**「 改行コード 」**などと呼ばれます。 実はこれも OS の種類によって異なり、 テキストファイルの見た目がおかしくなったりする原因です。 readln 関数では、いろいろな環境の改行コードをふまえた上で、1 行ごとに読み込んでくれます。 読み書きの際に自分で細かく調整する事もできますが、ちょっとややこしいので、ここでは省略しましょう。

CSV ファイルとグラフの描画

ここでは、CSV ファイルとグラフの描画について説明します。

■ CSV ファイル

■ プログラムは、別のソフトなどと組み合わせて使う事も多い

これまでは、式や変数をはじめ、条件分岐や繰り返し、ファイルの読み書きなど、 プログラムにおける基本的な処理について説明してきました。 でも、前回までで基本はだいたい出そろいました。これまでの内容を組み合わせれば、もう色々な事ができるはずです。 そこで最終回となる今回は、より実践的な内容に挑戦してみましょう。

これまでに扱ってきたのは、いわば「 プログラムの書き方を学ぶためのプログラム 」でした。 画面に表示する内容や、ファイルに書き込む内容なども、プログラムの動作を確認するためのものばかりでした。 変数の値や入力値などについてもそうで、いわば「 とりあえずの値 」を入れていましたね。 でも、実際のプログラムは、もっと具体的な目的があって使われるものです。 読み込んだり書き込んだりする内容も、ちゃんと意味や使い道があるものになるでしょう。 たとえば、なにか別のソフトで使うためのデータを作成したり、 逆に別のソフトで作ったデータを読み込んで処理したり、といったパターンはよくあります。

別のソフトやプログラムとデータをやり取りする方法はいくつかありますが、 単純にファイルの読み書きで行うのも、一般によく使われる方法です。 たとえばプログラムからデータをファイルに書き込んで、そのファイルを別のソフトで開くわけです。 人間同士が、手紙やメールでやり取りする感じに似ていますね。

■ ファイルの「 書き方 」によく使われる CSV 形式

このようにファイル経由で別のソフトとやり取りする場合、「 ファイル内に、データをどういう書き方で書くか 」というルール を決めておく必要があります。 これが書く側と読む側でちぐはぐだと、データを正しく伝えられません。

さて、「 ファイルの読み書き 」の節では、 以下の 2 通りの書き方を使いました:

(1) 値の数だけファイルを作って、各ファイルに 1 つずつ値を書く
(2) ファイルは 1 個だけで、各行に 1 つずつ値を書く

(1) は究極的にシンプルですが、値の個数が多い場合にファイルだらけになって管理が大変です。その点で (2) のほうが実用的でしょう。でも、もう一歩すすんで、1 行に複数の値を書けると、さらに便利です。そこで実際には、

(3) 各行の中に、何かの文字(「 , 」など)で区切って、複数の値を書く

のような書き方がよく使われます。こうすると、たとえば以下のような「 表 」の形のデータを書くのに便利です:

	A	B
1	0	0
2	0.1	0.01
3	0.2	0.04
4	0.3	0.09
5	0.4	0.16
6	0.5	0.25
7	0.6	0.36
8	0.7	0.49
9	0.8	0.64
10	0.9	0.81
11	1	1

このような表の形のデータは、よく表計算ソフトなどで登場します。実際に表計算ソフトで上の図のデータを打ち込み、「 CSV 」という形式で保存(「 エクスポート 」メニューなどから)すると、以下のようなファイルが作られます:

[表計算ソフトで保存したファイル「 hyou.csv(ひょう.csv)」の中身]

```
0,0
0.1,0.01
0.2,0.04
0.3,0.09
0.4,0.16
0.5,0.25
0.6,0.36
0.7,0.49
0.8,0.64
0.9,0.81
1,1
```

(※ CSV ファイルをふつうにダブルクリックすると、表計算ソフトで開かれてしまう事が多いでしょう。その場合はファイルを右クリックし、「 プログラムから開く 」メニューからテキストエディタ(メモ帳など)を選択して開いてください。)

上のように、表の行(よこ方向の並び)はそのままファイル内の行に対応しています。そして、**表の列(たて方向の並び)は、ファイル内で「 , 」記号で区切って表されています。** 表の左側の列にある値は「 , 」記号の左に、表の右側の列にある値は「 , 」記号の右に、という具合です。

「 , 」記号はコンマやカンマと呼ぶので、このようなファイルの書き方は、「 カンマ区切りの値 」を意味する英語 「 Comma Separated Values 」を略して**「 CSV(シーエスブイ) 」**形式と呼ばれています。そして CSV 形式で書かれたファイルは **CSV ファイル**と呼ばれます。CSV ファイルは、いろいろなソフトで幅広くサポートされているため、プログラムでも読み書きする場面がよくあります。よって、今回も CSV 形式を用いる事にしましょう。

■ グラフソフト

■ CSV ファイルに描かれたデータから、グラフを描くソフト

さて、CSV ファイルを使って何をするかですが、今回は絵的にわかりやすい **「 グラフを描く 」**という事を行ってみましょう。グラフを描くソフトは、CSV ファイルを使う典型的なソフトの一つです。高機能なものもありますが、今回はシンプルな**「 1 行ごとに 1 つの点を描いてくれるタイプのグラフソフト 」**を用意します。たとえば:

・リニアングラフ 2D (※ 2 列の CSV ファイルから、平面のグラフを描く)
https://www.rinearn.com/ja-jp/graph2d/

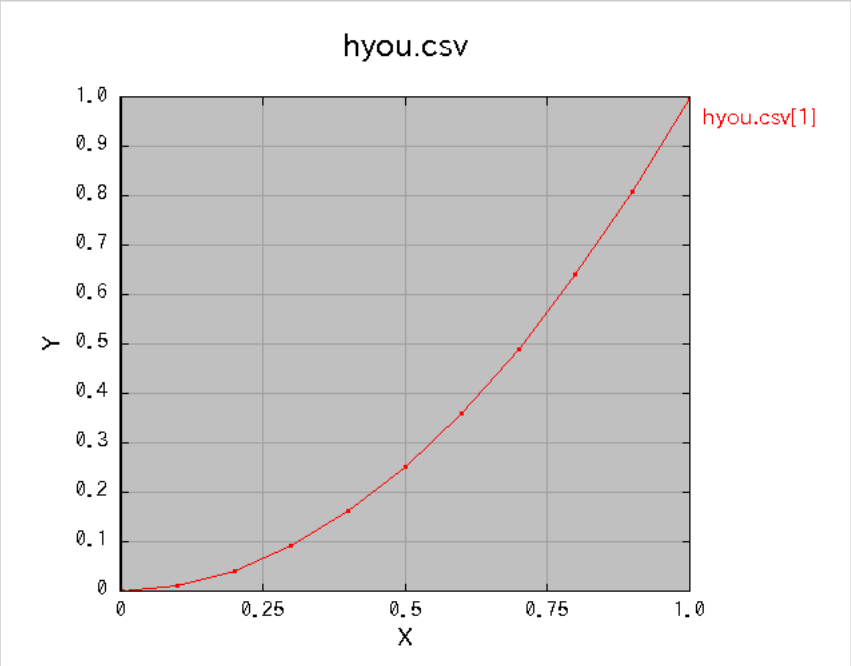
・リニアングラフ 3D (※ 3 列の CSV ファイルから、立体のグラフを描く)
https://www.rinearn.com/ja-jp/graph3d/

などがあります。これらは PC 用の各種 OS で動作し、無料で使用できます。

試しにリニアングラフ 2D(※ 3D ではない)を使って、先ほどの CSV ファイルからグラフを描いてみましょう。まず上のリンクからソフトをダウンロード・展開し、中の**「 RinearnGraph2D.jar (JAR ファイル)」をダブルクリック**して起動してください。

そしてメニューバーの「 File(ファイル)」メニューから「 Open File(ファイルを開く)」を選び、**先ほど表計算ソフトで保存した CSV ファイル「 hyou.csv 」を読み込んでプロット**してください。
(※ 表計算ソフトが無い場合などは、上で掲載してある「 hyou.csv 」の内容をメモ帳などのテキストエディタで書いて保存してください。保存の際は、「 ファイルの種類 」の項目を「 すべてのファイル 」にしてください。)

実際にプロットした様子は下図の通りです:



このように画面上に、いくつかの点が線でつながれたグラフが表示されます。この中の 1 つの点が、ファイル内の 1 行に対応していて、**左の列(「 , 」記号の左側)の値を x 座標、右の列の値を y 座標とみなした位置**に描かれています。実は、さっきの CSV ファイルにおいて、右の列の値は、左の列の値を 2 乗したものになっていました。よってこのグラフは、中学校で習う**「 $y = x^2$ 」のグラフ**になっています。きれいな形ですね。

■ プログラムからグラフソフトを起動させる

ところで、上で使ってみたリニアングラフは、実は VCSSL と同じ開発元のソフトです。なので、VCSSL には標準で、**プログラム内からグラフソフト(※ リニアングラフが使われます)を起動させたり、操作したりする機能**がサポートされています:

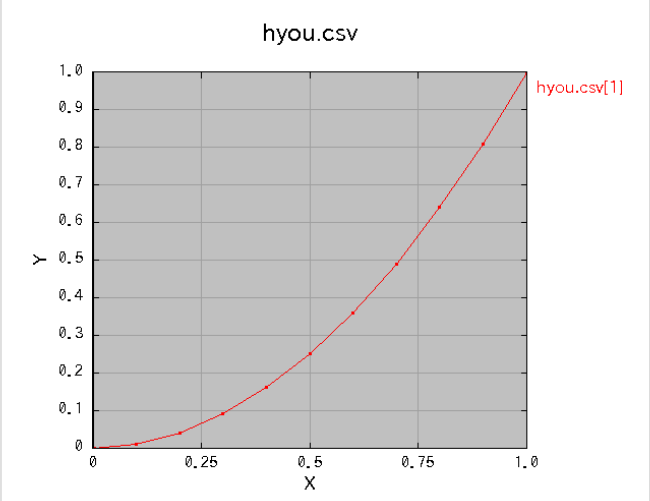
```
// グラフソフトを操作するために必要な行
import tool.Graph2D ;

// グラフソフトを起動し、ID 番号を変数に代入
int graphID = newGraph2D () ;

// グラフソフトに CSV ファイルを読み込ませる
setGraph2DFile ( graphID, "hyou.csv" ) ;
```

最初の行「 import ～ 」は、グラフソフトを操作するために必要な「 ライブラリ 」というものを読み込んでいる行です。ライブラリとは、プログラム内で使う機能を増やすためのものです。この行により、グラフ関連の関数が使えるようになります。

続いて使っている「 newGraph2D (ニューグラフ・ツーディー)」関数は、グラフソフトを起動する関数です。その際、グラフが複数あっても区別できるよう、ID 番号が割りふられます。それを「 = 」の左の変数「 graphID 」に代入しています。

最後の「 setGraph2DFile(セットグラフ・ツーディー・ファイル)」関数は、グラフソフトに CSV ファイルなどを読み込ませてくれる関数です。カッコ内の最初にグラフソフトの ID 番号を指定し、その次に、読み込ませるファイルの名前を書きます。

このプログラムを、先ほどの CSV ファイル「 hyou.csv 」と同じフォルダ内に置いて実行すると、グラフが表示されます:



このように、**結果は手動でグラフソフトを使った場合と同じ**です。ある意味、手動でプロットしたほうが「プログラムを別のソフトと連携させてる感」が出るかもしれないですが、せっかくなので、今回は上のようにプログラム内からグラフソフトを起動し、CSV ファイルを読ませる事にしましょう。もちろん、なにか別のグラフソフトを使いたい場合など、あえて手動でグラフ表示を行っても OK です。

■ プログラムでの CSV ファイルの読み書きとグラフ描画

■ プログラムで CSV を書き、グラフソフトに描画してもらう

さて、CSV ファイルとグラフソフトがどんなものか大体わかったところで、本題です。プログラムでデータを CSV ファイルに書きだして、そのデータをグラフソフトに描画してもらいましょう。これは、自作のプログラムを、別のソフトやプログラムと組み合わせて使う、いい例になります。

CSV ファイルを読み書きするには、まず**書き込む内容に「，」を付けたり、読み込んだ行を「，」で区切ったりする**処理が必要になりますね。また、実際の CSV ファイルにはいろいろな書き方があり、値が「"」で囲まれていたりして、もっと複雑な処理が必要になる事もあります。

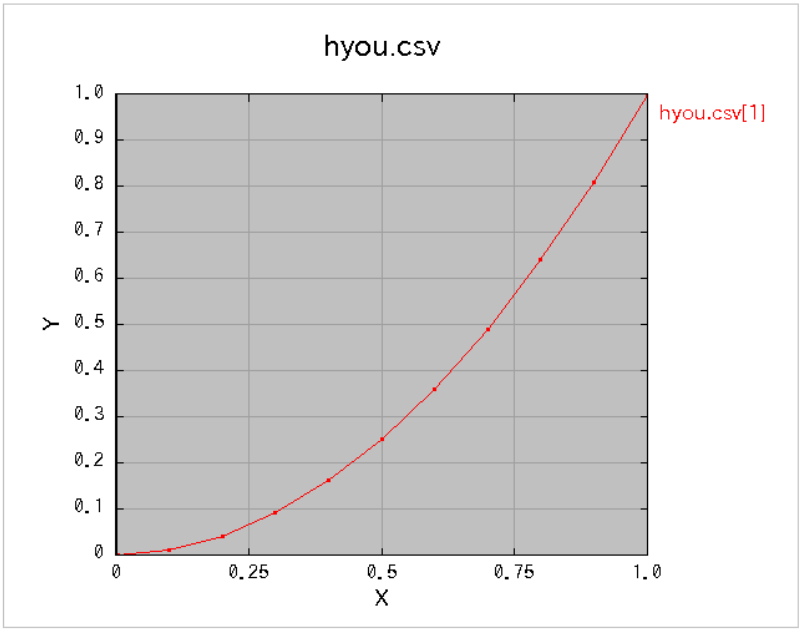
もちろん、そういった処理を自力でプログラムに書く事もできるのですが、VCSSL には CSV ファイルを手軽に読み書きするための機能が、標準で用意されています。今回はそれを使いましょう。一番単純なのは、**ファイルを開く open 関数で、CSV 読み書き用のモードを使う**事です。たとえば書き込みなら、以下のようにします：

```
// グラフソフトを操作するために必要な行
import tool.Graph2D ;

// ファイルを CSV 書き込みモードで開き、ID を変数に代入
int fileID = open ( "file.csv", WRITE_CSV ) ;

 // くりかえし処理で、1 行ごとに x, y の値を書き込む
for ( int i=0; i<=10; i++ ) {
    // ※ カウンタ変数 i は毎回 1 ずつ増える

    // x と y の値を変数に用意
    float x = i * 0.1 ;  // x は 0.1 ずつ増える
    float y = x * x ;    // y は x の 2 乗

    // x と y の値をファイルに「 , 」区切りで書き込む
    writeln ( fileID, x, y ) ;
}

// ファイルを閉じる
close ( fileID ) ;

// グラフソフトを起動し、CSV ファイルを読み込ませて描画
int graphID = newGraph2D ( ) ;
setGraph2DFile ( graphID, "file.csv" ) ;
```

このように、open 関数の中で「 WRITE_CSV（ライト・シーエスブイ）」と書いていますが、これはファイルに CSV 形式でデータを書き込むためのモードです（ 実はここは "wcsv" と書いても同様に動きます ）。このモードを使った場合、上のように **write 関数のカッコ内に複数の値を書くと、それらは「 , 」記号で区切ってファイルに書き込まれる**ようになります。

上のプログラムでは、for 文を使う事で、ファイルの各行を自動で書き込むようにしています。for 文の詳細は「 くりかえし処理 」の回を読みかえしてみてください。ここでは、カウンタ変数「 i 」の値が 0 からスタートし、1 ずつ増えながら、10 までの間ずっと { … } の中の処理がくりかえされます。その中で毎回 x と y の値を求め、writeln 関数でファイルに 1 行書き込んでいます。

このプログラムを実行すると、**「 file.csv 」という名前の CSV ファイルが作られ、その中に x と y の値が書き込まれます。**そして、それがグラフソフトに読み込まれて、以下のようなグラフが表示されます：



また先ほどと同じグラフですね。それもそのはず、**上のプログラムで求めている x は、for 文のカウンタ変数 i に 0.1 をかけたものなので、0 から 1 まで 0.1 ずつ変化**します。y はその 2 乗としています。これらは先ほど表計算ソフトで作った「 hyou.csv 」と同じなので、同じグラフになったわけです。

実際にプログラムで書き込んだ CSV ファイル「 file.csv 」の中身を見てみましょう:

```
0.0,0.0
0.1,0.010000000000000002
0.2,0.04000000000000001
0.30000000000000004,0.09000000000000002
0.4,0.16000000000000003
0.5,0.25
0.6000000000000001,0.3600000000000001
0.7000000000000001,0.4900000000000001
0.8,0.6400000000000001
0.9,0.81
1.0,1.0
```

このように、たしかに x の値(「 , 」の左の列)は 0.1 ずつ増えていて、y の値(左の列)は x の 2 乗になっています。ちゃんと計算して書き込めているようです。

でも上の内容には、たとえば「 0.010000000000000002 」のように、**いくつかの値に小さなゴミのようなものが付いています。**これは「 式と計算 」の回 で説明した、小数(浮動小数点数)

の誤差によるものです。この誤差は非常に小さいので、ふつうはほとんど問題にならないですが、このようにファイルに書き出すとじゃまになる事もあります。その場合、書き込み前に round 関数で適当な桁数に丸めれば、きれいになります ：

```
...
// x と y の値を変数に用意
float x = i * 0.1 ;  // x は 0.1 ずつ増える
float y = x * x ;    // y は x の 2 乗

// 誤差を丸めるため、小数点以下 5 桁までに四捨五入
x = round ( x, 5, HALF_UP ) ;
y = round ( y, 5, HALF_UP ) ;

// x と y の値をファイルに「 , 」区切りで書き込む
writeln ( fileID, x, y ) ;
...
```

この例では x と y の値を、小数点以下が 5 ケタまでになるよう四捨五入しています。「 HALF_UP（ハーフアップ）」のかわりに「 HALF_UP_SIGNIF（ハーフアップ・シグニフィカンド）」と書けば、有効数字のケタ数指定もできます。さて結果は：

```
0.0,0.0
0.1,0.01
0.2,0.04
0.3,0.09
0.4,0.16
0.5,0.25
0.6,0.36
0.7,0.49
0.8,0.64
0.9,0.81
1.0,1.0
```

誤差が丸め込まれて、きれいになりましたね。「 00...0 」と続くケタは省略されてしまっていますが、ちゃんと内部で小数点以下 5 ケタまでに丸められています。

■ CSV ファイルに書かれたデータを、配列として読み込む

続いて、CSV ファイルからデータを読み込んでみましょう。これも、別のソフトで作ったデータを、プログラムで加工したい場合などによくあるパターンです。ファイルを開く open 関数には、CSV 形式の読み込みのためのモードが用意されています:

```
// ファイルを CSV 読み込みモードで開き、ID を変数に代入
int fileID = open ( "file.csv", READ_CSV ) ;

// ファイルの行数を数えて、変数に代入
int n = countln ( fileID ) ;

// x と y の値を格納する配列を宣言 ( 要素数は行数 )
float x [ n ] ;
float y [ n ] ;

// くりかえし 1 行ずつ読んでいく
for ( int i=0; i<n; i++ ) {

    // 1 行の各列の値を格納する配列を宣言
    float line [ 2 ] ;      // 2 列のデータなので [ 2 ]

    // 1 行読んだ内容を、「 , 」区切りで配列に代入
    line = readln ( fileID ) ;

    // 左の列の値は x に、右の列の値は y に代入
    x [ i ] = line [ 0 ] ;    // [ 0 ] 番目は左の列
    y [ i ] = line [ 1 ] ;    // [ 1 ] 番目は右の列
}
// ファイルを閉じる
close ( fileID ) ;

// 読み込んだ内容を表示
for ( int i=0; i<n; i++ ) {
     println ( x [ i ], y [ i ] ) ;
}
```

基本的には、「 ファイルの読み書き 」の回で説明した、ループを使った読み込み と同様です。ファイルを開いた後に countln 関数で行数を数えて、for 文 でくりかえし 1 行ずつ、readln 関数を使って読み込んでいます。

違う点は、まず open 関数のカッコ内で「 READ_CSV （ リード・シーエスブイ ）」と書いている事です。これは CSV 形式のファイルを読み込むためのモードで、書き込みで使っていた「 WRITE_CSV 」の逆バージョンです。

このモードでは、readln 関数が、読み込んだ行の内容を「 , 」記号で区切ってくれるようになります。その結果は、「 = 」記号の左にある配列（ ここでは「 line 」 ）に代入されます。その配列の [0] 番には一番左の列の値が入り、[1] 番にはその右の列、[2] 番には（ あれば ）さらにその右の列 ... という順番で値が入ります。

読み込んだ値をその場で println 関数で表示してもよいのですが、それでは実用的な例になりませんね。実際には、読み込んだデータにいろいろな処理を加える事が多いでしょう。そこで、ここではあらかじめ宣言しておいた x と y の配列に、各行ごとの値を入れておく処理にしています。それを、あとでまとめてで表示しています。

実行結果は以下の通りです：

```
0.0	0.0
0.1	0.01
0.2	0.04
0.3	0.09
0.4	0.16
0.5	0.25
0.6	0.36
0.7	0.49
0.8	0.64
0.9	0.81
1.0	1.0
```

この通り、ちゃんと CSV ファイルの値を正しく読み込めている事がわかりますね。

（ ※ println 関数のカッコ内に複数の値を書いた場合、このように空白（ タブ ）で区切って表示されます。）

■ 3D グラフの描画

■ 3D の線グラフを描画する

プログラム内での CSV ファイルとグラフソフトの扱いにもなれてきたところで、一歩進んで、3D のグラフを描いてみましょう。まずは、これまでの 2D グラフの延長線上で簡単に描ける、「線グラフ」からです。この場合、CSV ファイルを 3 列にして、各行に「x,y,z」のように点の座標を書けば OK です:

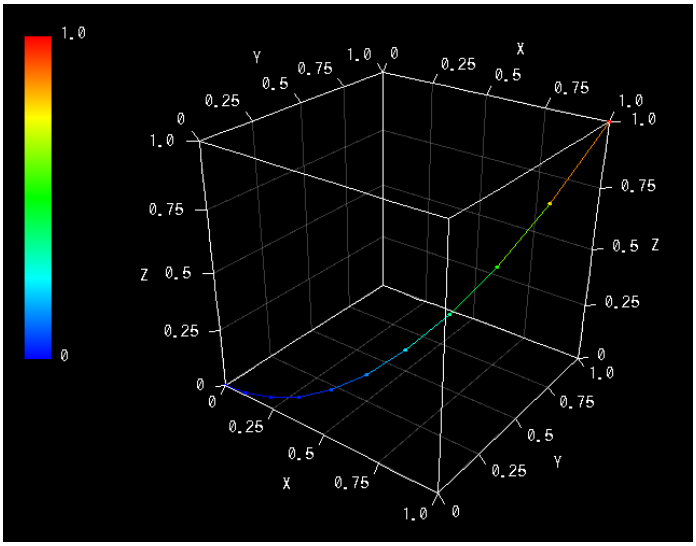
```
// グラフソフトを操作するために必要な行
import tool.Graph3D ;

// ファイルを CSV 書き込みモードで開き、ID を変数に代入
int fileID = open ( "file.csv", WRITE_CSV ) ;

// くりかえし処理で、1 行ごとに x, y の値を書き込む
for ( int i=0; i<=10; i++ ) {

    // ※ カウンタ変数 i は毎回 1 ずつ増える

    // x, y, z の値を変数に用意
    float x = i * 0.1 ;       // x は 0.1 ずつ増える
    float y = x * x ;         // y は x の 2 乗
    float z = x * x * x ;  // z は x の 3 乗

    // 誤差を丸めるため、小数点以下 5 桁までに四捨五入
    x = round ( x, 5, HALF_UP ) ;
    y = round ( y, 5, HALF_UP ) ;
    z = round ( z, 5, HALF_UP ) ;

    // x, y, z の値をファイルに「 , 」区切りで書き込む
    writeln ( fileID, x, y, z ) ;
}
// ファイルを閉じる
close ( fileID ) ;
```

```
// グラフソフトを起動し、CSV ファイルを読み込ませて描画
int graphID = newGraph3D ( ) ;
setGraph3DFile ( graphID, ”file.csv” ) ;

// 点を線でつなぐオプションを有効化
setGraph3DOption ( graphID, ”WITH_LINES”, true ) ;
```

ちょっと長いですが、基本的には先ほどの 2D グラフの場合とほぼ同じです。違うのは、ところどころ「 Graph2D 」のかわりに「 Graph3D 」になっている事と、**z の値の処理が追加されている**事くらいです。z の値は x の 3 乗にして、writeln 関数で y の右の列に書き込むようにしています。このプログラムを実行すると:

```
0.0,0.0,0.0
0.1,0.01,0.001
0.2,0.04,0.008
0.3,0.09,0.027
0.4,0.16,0.064
0.5,0.25,0.125
0.6,0.36,0.216
0.7,0.49,0.343
0.8,0.64,0.512
0.9,0.81,0.729
1.0,1.0,1.0
```

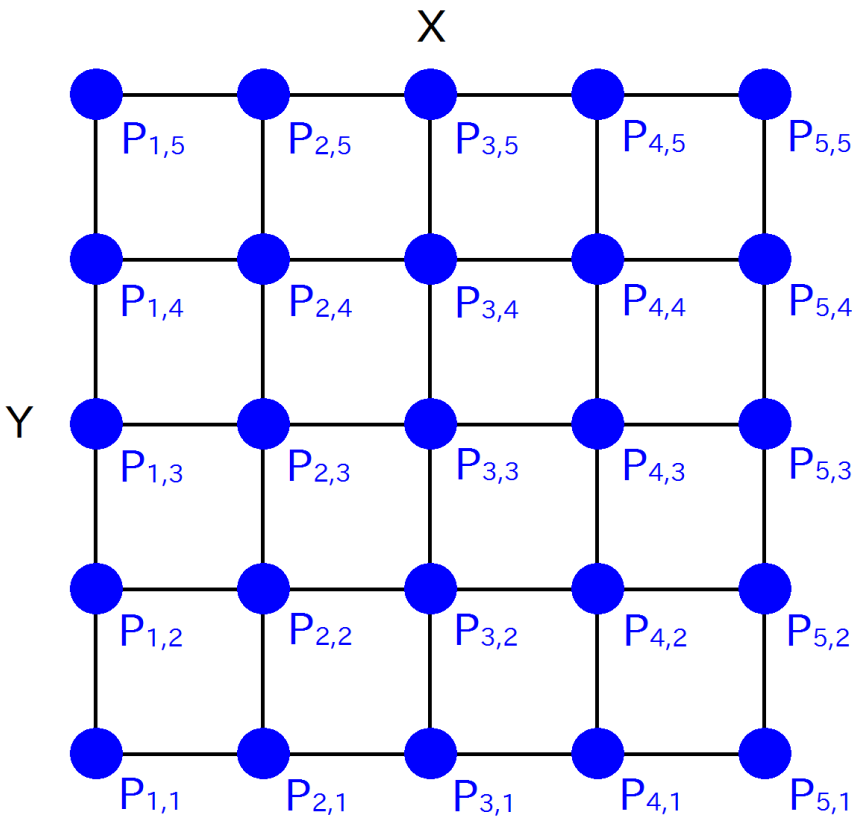
このように、CSV ファイル「 file.csv 」に「 , 」区切りで 3 列のデータが書き込まれます。一番右の列に z の値が書かれています。たしかに x の値(一番左の列)の 3 乗ですね。続いて、以下のような 3D グラフが描画されます。ちゃんと立体的に曲がった曲線になっていますね。



■ 3D のメッシュグラフや曲面グラフを描画する

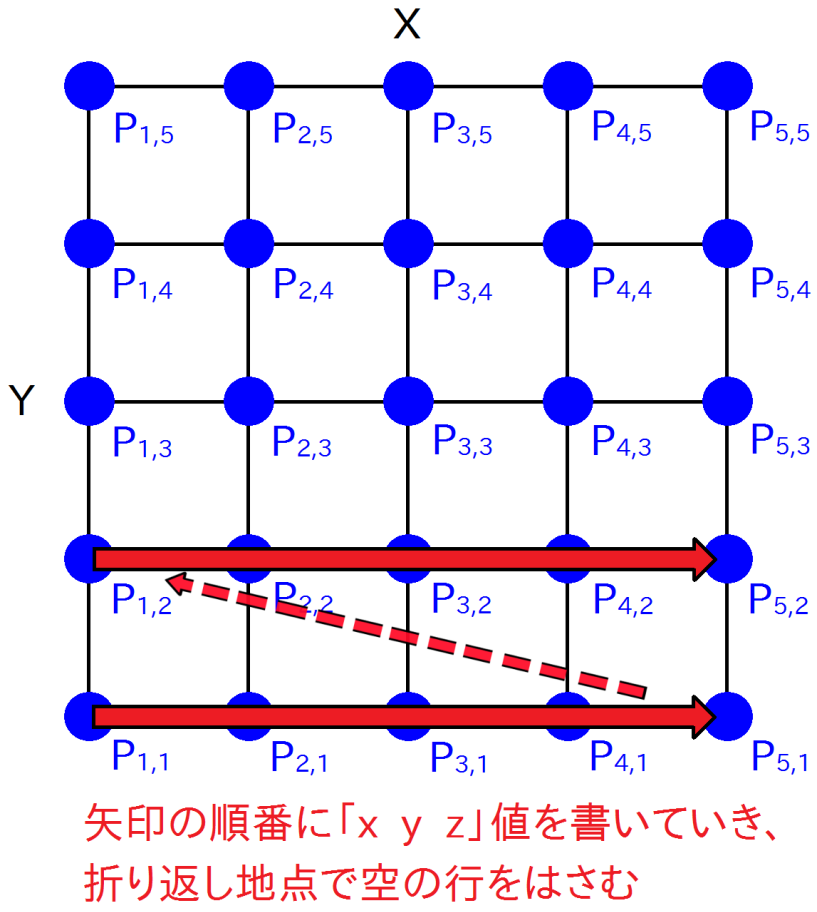
さて、最後です。少しむずかしくなってしまいますが、3D のメッシュグラフを描いてみましょう。メッシュグラフというのは、SF 映画などであみの目のように立体形状が描かれている、あれです。3D グラフと言えば、やはりあれでしょう!

メッシュグラフを描くためには、CSV ファイルに、独特の書き方でデータを書き込む必要があります。1 行につき 1 点の座標を「 x,y,z 」と書く事は同じですが、その点を書いていく順番が決まっています。たとえば、メッシュ上の点(格子点)が、X-Y 平面において下図のように並んでいるものとしましょう:



(※ もちろん、それぞれの点は z の値も持っていますが、それはこの図では画面と垂直な方向です。)

さて、この図のような場合、まず y は端(はし)の値に固定して、x 軸の方向に点の座標を書いていきます。この図では P1,1 から P5,1 までです。**そして x 軸の端まできたら、空(から)の行を 1 行だけ書き込みます。**続いて、y 軸方向に 1 個だけ進んで、また x 軸の方向に点の座標を書いていきます。この図では P1,2 から P5,2 までです。その後はまた空の行を 1 行だけはさんで、P1.3 から P5.3 まで書きます。これを繰り返して、最終的に P5,5 まで書き込めば終わりです。



書き込む内容

x1,1 y1,1 z1,1
x2,1 y2,1 z2,1
x3,1 y3,1 z3,1
x4,1 y4,1 z4,1
x5,1 y5,1 z5,1
(空の行をはさむ)
x1,2 y1,2 z1,2
x2,2 y2,2 z2,2
x3,2 y3,2 z3,2
x4,2 y4,2 z4,2
x5,2 y5,2 z5,2
(空の行をはさむ)
…

なんだかややこしい書き方ですね。でも実は、プログラムだと意外と簡単なのです:

```
// グラフソフトを操作するために必要な行
import tool.Graph3D ;

// ファイルを CSV 書き込みモードで開き、ID を変数に代入
int fileID = open ( "file.csv", WRITE_CSV ) ;
```

```
// y 方向のくりかえし
for ( int j=0; j<=10; j++ ) {

    // x 方向のくりかえし
    for ( int i=0; i<=10; i++ ) {

        // ※ y 方向（外側）のくりかえしで j が 1 ずつ増加
        // ※ x 方向（内側）のくりかえしで i が 1 ずつ増加

        // x, y, z の値を変数に用意
        float x = i * 0.1 ;         // x は i に伴い増加
        float y = j * 0.1 ;         // y は j に伴い増加
        float z = y*y - x*x + x ;   // z はちょっと複雑に

        // 小数点以下 5 桁までに四捨五入（ 誤差の丸め ）
        x = round ( x, 5, HALF_UP ) ;
        y = round ( y, 5, HALF_UP ) ;
        z = round ( z, 5, HALF_UP ) ;

        // x, y, z の値を「 , 」区切りで書き込む
        writeln ( fileID, x, y, z ) ;
    }

    // x 方向（内側）のくりかえしが終わると空行をはさむ
    writeln ( fileID, "" ) ;
}

// ファイルを閉じる
close ( fileID ) ;

// グラフソフトを起動し、CSV ファイルを読み込ませて描画
int graphID = newGraph3D ( ) ;
setGraph3DFile ( graphID, "file.csv" ) ;

// 点をメッシュでつなぐオプションを有効化
setGraph3DOption ( graphID, "WITH_MESHES", true ) ;
```

このように、基本的には **for 文が 2 重になった事と、y の値の計算式が少し違うだけ**ですね。内側にある(上から 2 つめの)for 文は、これまでと同様に、x の値を 0.1 ずつ増やすくりかえしです。いまの場合は、これは先ほどの例の P1,1 から P5,1 までのように、書き込む点を x 方向に動かしていく事に対応しています。x が端(はし)にくると、この for 文は一旦終わりです。

すると、外側にある(上から 1 つめの)for 文が 1 周回りますね。これによって、**カウンタ変数 j の値に 1 が足されて、y の値が 0.1 だけ増えます。**そして再び内側の for 文がくりかえされ、先ほどの例の P1,2 から P5,2 までのように、書き込む点が x 方向に動いていきます。後も同様です。確かに先ほどの例の通りになりますね。

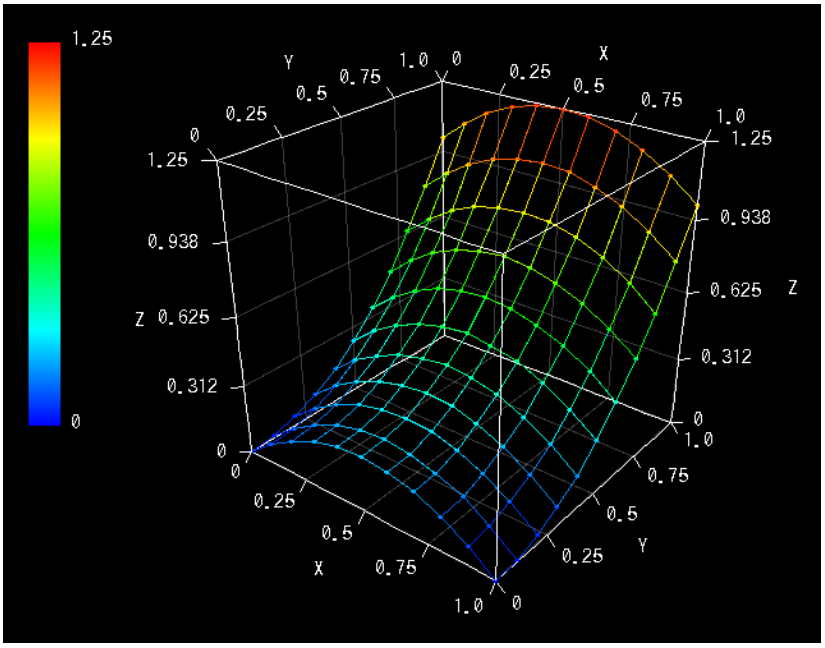
さて、上のプログラムでは、z については少し複雑に「 $z = y^2 - x^2 + x$ 」という値にしています。グラフがどんな形になるのか、ぱっとイメージするのはちょっと難しいですね。でも、実はそれが狙いです。**結果が簡単にイメージできないからこそ、プログラムを書いてコンピューターに計算させる意味がある**と思いませんか?

それでは実行してみましょう。実行すると、まず以下の内容が CSV ファイル「 file.csv 」に書き込まれます:

```
0.0,0.0,0.0
0.1,0,0.09
0.2,0,0.16
0.3,0,0.21
0.4,0,0.24
0.5,0,0.25
0.6,0,0.24
0.7,0,0.21
0.8,0,0.16
0.9,0,0.09
1.0,0.0,0.0

0.0,0.1,0.01
0.1,0.1,0.1
0.2,0.1,0.17
0.3,0.1,0.22
0.4,0.1,0.25
0.5,0.1,0.26
0.6,0.1,0.25
0.7,0.1,0.22
```

```
0.8,0.1,0.17
0.9,0.1,0.1
1.0,0.1,0.01

0.0,0.2,0.04
...
```

ちゃんと想定通りの内容ですね。そして、以下のようなグラフが表示されます：



この通りです。思いもよらない形だったのではないでしょうか？ という事は、 いま私たちはコンピューターに「 答えを教えてもらった 」わけです。 このコーナーでのこれまでのプログラムは、結果がわかりきったものばかりでしたが、今回は違います。「 難しいグラフの形を求める 」という、れっきとした問題を、コンピューターとプログラミングによって解いたわけです！

■ 3D のメッシュグラフや曲面グラフを描画する

さて、もしかすると上のグラフの形が、予想どおりだったという人もいるかもしれません。**そこで最後におまけとして、もっと複雑なグラフを描いておしまいにしましょう。** 内容が数学的で少し難しくなってしまうため、詳しい説明は省略し、プログラムとグラフだけ掲載します：

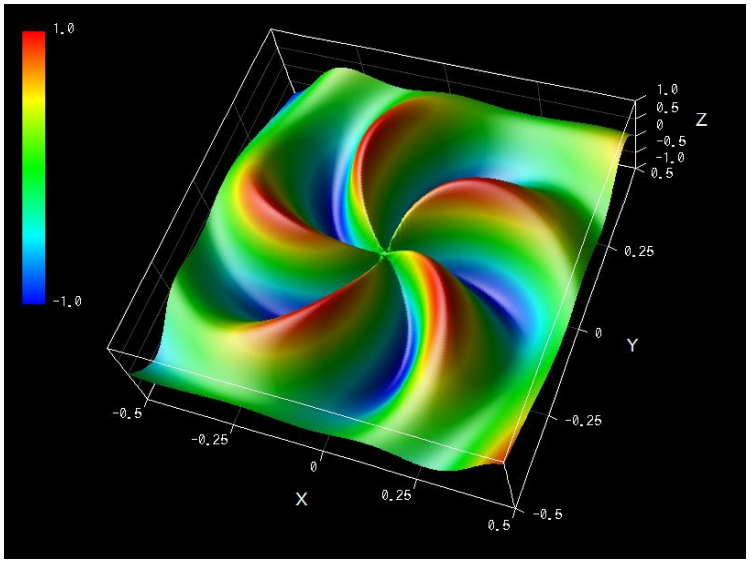
```
// グラフソフトを操作するために必要な行
import tool.Graph3D ;
```

```
// 数学関数を使うために必要な行
import Math ;

// ファイルを CSV 書き込みモードで開き、ID を変数に代入
int fileID = open ( "file.csv", WRITE_CSV ) ;

for ( int j=-50; j<=50; j++ ) {
    for ( int i=-50; i<=50; i++ ) {
        // ※ 外側のくりかえしで j が 1 ずつ増える
        // ※ 内側のくりかえしで i が 1 ずつ増える

        // x, y の値を変数に用意
        float x = i * 0.01 ; // x は i に伴い増加
        float y = j * 0.01 ; // y は j に伴い増加

        // X-Y 平面での点と原点との距離 r を求める
        float r = sqrt ( x * x + y * y ) ;

        // 点が X 軸となす角度θを求める
        float theta = 0.0 ;
        if ( x == 0.0 ) {
            if ( y > 0 ) {
                theta = PI ;
            } else {
                theta = PI * 2.0;
            }
        }
        if ( x > 0.0 ) {
             theta = atan ( y / x ) + PI / 2.0 ;
        }
        if ( x < 0 ) {
             theta = atan ( y / x ) + PI * 1.5 ;
        }

        // θ と r の値を使って、z の値を計算する
        float z = sin(5.0*theta+10.0*r) * sin(6.0*r) ;
```

```
        // 小数点以下 5 桁までに四捨五入 ( 誤差の丸め )
        x = round ( x, 5, HALF_UP ) ;
        y = round ( y, 5, HALF_UP ) ;
        z = round ( z, 5, HALF_UP ) ;

        // x, y, z の値を「 , 」区切りで書き込む
        writeln ( fileID, x, y, z ) ;
    }

    // 内側のくりかえしの最後に、空の行をはさむ
    writeln ( fileID, "" ) ;
}

// ファイルを閉じる
close ( fileID ) ;


// グラフソフトを起動し、CSV ファイルを読み込ませて描画
int graphID = newGraph3D ( ) ;
setGraph3DFile ( graphID, "file.csv" ) ;

// 点を描かずに、曲面を描くようにオプションを設定
setGraph3DOption ( graphID, "WITH_POINTS", false ) ;
setGraph3DOption ( graphID, "WITH_MEMBRANES", true ) ;
```

このプログラムで描かれるのは、座標の原点との距離を r 、X 軸とのなす角度を θ として、「 $z = \sin(5\theta + 10r)\sin(6r)$ 」のグラフです。ちなみに、このような r と θ による表し方を「 極座標 」といいます。さあ、実行してみましょう:

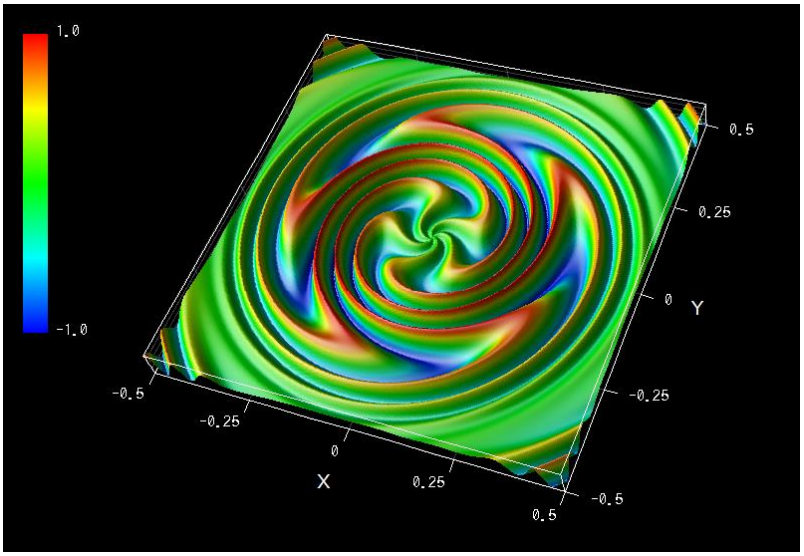
(※ グラフの「 Option(オプション)」メニューから「 Tesselation(曲面高画質化)」を選択すると綺麗になります。)



三角関数が入り混じった複雑な式からすると、意外なほどきれいなグラフですね！ さらに、z の値を計算している行を、以下のように 1 行だけ書き換えると、もっと複雑なグラフになります：

```
// θ と r の値を使って、z の値を計算する
float z = sin(5.0*theta + 10.0*sin(14.0*r)) * sin(6.0*r) ;
```

実行結果は以下の通りです。 このグラフをすぐに、そして正確に手で描けるという人は、ほぼいないのではないでしょうか。



コンピューターは、言われた事を忠実に行うだけのやや不器用な存在ですが、 そのかわり処理の正確さとスピードは、人間よりもはるかに優れています。 そんなコンピューターの長所を理解して活用すると、上のグラフのように、人間だけでは見えなかった新しい世界を見せてくれます。 それでは、楽しいプログラミング生活を！

本文書内の商標について

[１] Oacle と Java は、Oracle Corporation 及びその子会社、関連会社の米国及びその他の国における登録商標です。文中の社名、商品名等は各社の商標または登録商標である場合があります。

[２] Windows、Visual Studio、C# は米国 Microsoft Corporation の米国およびその他の国における登録商標です。この文書は独立著作物であり、Microsoft Corporation と関連のある、もしくはスポンサーを受けるものではありません。

[３] Linux は、Linus Torvalds 氏の米国およびその他の国における商標または登録商標です。

[４] その他、文中に使用されている商標は、その商標を保持する各社の各国における商標または登録商標です。

プログラミング言語 VCSSL スタートアップガイド 第 3 版

著　　者　　松井文宏

このガイドの内容は、以下の Web サイトにおいても公開されています。
https://www.vcssl.org/ja-jp/doc/start/