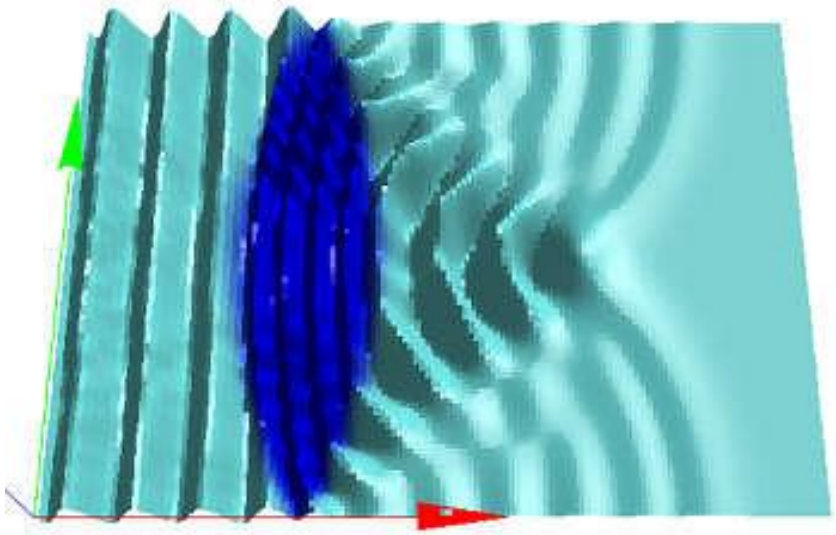
C 系言語ユーザー向け
VCSSL 即席ガイド



はじめに ー このガイドについて

VCSSL ―― Visualization/Computation/Simulation Script Language ―― は、データの可視化や、小規模な数値計算・科学技術シミュレーションのコードを、手軽に記述する事を主目的としたプログラミング言語（スクリプト言語）です。VCSSL は、開発コンセプトの 1 つとして、C/C++などの既存の C 系言語で数値計算を行っているユーザーが、可能な限り少ない学習コストで扱える事を目指しています。

このガイドは、C/C++などの各種 C 系言語ユーザーの方が、最小限の学習コストで VCSSL を扱うための、即席的なガイドです。読者はすでに何らかの C 系言語を習得している事を前提としており、C 系言語に共通するような内容の説明量を軽く抑える事で、簡潔にまとめています。内容の目安としては、前半（概要/文法）15 ページ + 後半（応用）12 ページで、1 日～1 週間程度での読了を想定しています。

VCSSL の概要

最初に、開発者視点で率直に目指したものなども交えて、VCSSL の概要を短くまとめてみます。

電卓ソフトや現実の電卓（ポケコン含む）には、関数定義や自動処理の機能として、何らかのプログラミング言語をサポートしているものが結構あります。あくまで筆者個人の勝手な考察ですが、電卓で動作するプログラミング言語には、以下のような点が要求されると考えています：

・少ない学習コスト	→ 電卓のために多くの学習コストを費やす人は恐らく少ない
・短い計算コードを手軽に書ける	→ 大規模よりも小規模重視、計算重視
・コンパクトな実装と仕様	→ 処理系を開発しやすい、搭載しやすい、移植しやすい
・数値計算の処理速度	→ 少なくともある程度の幅の数値計算には耐えないといけない

実は VCSSL も、もともと電卓ソフト用に開発したものです。つまり、何らかの斬新な言語仕様やパラダイムから作ったというよりは、最初に具体的な用途がまずあって、上のような要求点があり、そこから仕様が決まったという傾向の言語です。実際のVCSSLの開発コンセプトは以下の通りです：

・少ない学習コスト
- 数値計算分野の C 系言語ユーザーを主眼に置いた、C 系の標準的な文法
- 独特の文法は可能な限り避ける

・短い計算コードを手軽に書ける
- 汎用性を削り、C 系の中でも簡素な文法に
- 静的型付けでありつつ、やや緩めに
- 簡易用途の補助を重視した標準ライブラリ

・コンパクトな実装と仕様
- 制御構造の構文は if (if-else) / for / while の 3 種
- 基本的なプリミティブ型は int / float (=double) / complex / string / bool の 5 種
- 非オブジェクト指向、大規模向けのパラダイムは特に追わない
- 参照型やポインタを排除して GC を省略、配列を含む全ての型を値型に

・数値計算の処理速度
- 倍精度浮動小数点数演算で数十 MFLOPS ～ 100MFLOPS（※開発当初の目標）
- 高速な eval

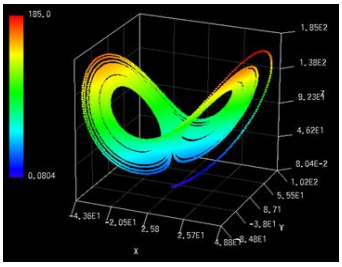
最初に「**少ない学習コスト**」を挙げていますが、やはり開発当初はあくまで電卓用だったので、この点は最も重要でした。実際、既存の電卓ではシンプルな BASIC 系を採用しているものをよく見

かけます。しかし VCSSL では、数値計算分野の C/C++ユーザーを特に重視したかったので、C 系の文法を採用しました。その代わりに、C 系の中でも標準的な、なるべく独特でない見た目になるようにしました。とは言っても、とっつきやすい BASIC と比べて、C は汎用重視の色が濃いので、「**短い計算コードを手短に書ける**」ために、どれだけ（他の C 系スクリプト言語のように）小回りを確保するかという配慮も必要です。結局は妥協点ですが、以下のような感じになりました(FizzBuzz)：

```
int n = input("上限値(整数)");     // 入力（戻り値は文字列だが数値に変換される）
for(int i=1; i<=n; i++) {          // for 文
   if(i%3 == 0 && i%5 == 0) {    // if 文
      println("FizzBuzz,");        // 出力
   } else if(i%3 == 0) {
      println("Fizz,");
   } else if(i%5 == 0) {
      println("Buzz,");
   } else {
      println(i + ",");           // 文字列と数値の結合
   }
}
```

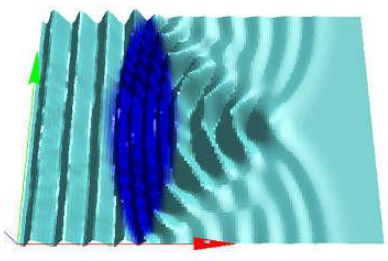
上の通り、VCSSL は C 系スクリプト言語の中でも保守的で、相応しい形容はまさに「電卓とかで動く C っぽいやつ」という感じです。実装をコンパクトに抑えるためのサブセット的な面も強いです。

一転して、標準ライブラリについてはある程度の汎用性を確保しています。これは筆者自身が数値計算をやっていて、「シミュレーションの可視化などをもっと手軽にできたらいいのに」と思っていた事に由来しています。つまり計算メインでもアニメーション描画などを行いたい場面は結構あるわけで、C/C++は速い代わりに、そのあたりを面倒に感じていました。そして VCSSL はせっかく C 系の文法で手軽さを意識したので、そのあたりをカバーできれば、（重い計算はやはり C/C++ですが）ライトなシミュレーションなどで活きる言語になるかもしれないと思いました。実際、VCSSL 2.0 以降は単体のスクリプト言語としても公開しており、標準ライブラリも（簡易用途向けであまり高度な事はできませんが）GUI や 2D/3D グラフィックスなどをサポートし、手軽に可視化などを行えます。



▲ローレンツ方程式の解曲線

https://www.vcssl.org/ja-jp/code/archive/0001/0100-lorenz-attractor/



▲凸レンズ状の高密度領域を通過する力学波

https://www.vcssl.org/ja-jp/code/archive/0001/1800-convex-lens/

■ 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

冒頭で VCSSL の開発コンセプトを列挙しましたが、その中には「**・コンパクトな実装と仕様 → 処理系を開発しやすい、搭載しやすい、移植しやすい**」というものがありました。実際に、VCSSL の開発初期には、スクリプトエンジンをライブラリとしてまとめて、ユーザーが自作のアプリケーション（ソフト）等に手軽に搭載できるようにする事も計画にありました。

一方、それが先延ばしになってしまい、なかなか実現しないまま、機能が拡充されていった結果、現在の VCSSL のスクリプトエンジンは、それなりに複雑化 & 規模が肥大化してしまい、ユーザーが手軽に搭載するのはあまり現実的ではなくなってしまいました。

そこで現在、**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、初期のコンセプト通り、**ユーザーが手軽に自作のアプリケーション等に搭載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。

プログラムの実行と基本的な入出力

それでは、早速 VCSSL のプログラムを作成して実行し、基本的な入出力を行ってみましょう。

プログラムの作成と実行 ー Hello World!

まずは各プログラミング言語で恒例の、Hello World! を出力するコードです：

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

必須では無いですが、1 行目で文字コード（ Shift_JIS / UTF-8 / EUC_JP ）を指定できます：

```
coding UTF-8;              // 文字コードを指定する場合は必ず 1 行目の先頭限定
print("Hello, World!");
```

続いてこの VCSSL のコードを実行するわけですが、以下の通り複数の方法があります：

・VCSSL ランタイムで実行する

まず適当なテキストエディタで、上記の Hello World!のコードを拡張子「 .vcssl 」をつけて保存します。続いて VCSSL ランタイムを https://www.vcssl.org/ja-jp/download/ から入手し、保存したコードを実行します。なお、VCSSL ランタイムはインストール不要で使用でき、USB メモリーからも使用できます。

・電卓ソフト「リニアンプロセッサー」で実行する

リニアンプロセッサーという電卓ソフト（フリー）を https://www.rinearn.com/ja-jp/processor/ から入手して起動し、「 INPUT 」項目に上記の Hello World!のコードを記述して、「 = 」ボタンを押して実行する事もできます（F11 キーでも OK）。これもインストール不要で、USB メモリーからも使用できます。

基本的な入出力

続いて、文法などの解説に入る前に、よく使用する基本的な入出力についてまとめておきます。

・コンソールへのメッセージ出力

print 関数（改行なし）または println 関数（改行あり）で、コンソールへメッセージを出力します：

```
println(123);              //代わりに print 関数を使うと行末で改行されなくなる
println("Hello");
```

実行すると改行を挟んで「123」「Hello」と出力されます。print / println 関数に複数の引数を指定すると、空白区切りで出力されます。配列を指定した場合も、全要素が空白区切りで出力されます。なお、任意の位置で改行したい場合は、以下のように改行コード定数 EOL を挟みます：

```
print("aaa" + EOL + "bbb" + EOL + "ccc");
```

・任意の内容の入力

文字列メッセージや数値など、任意の内容の入力には input 関数を用います：

```
int a = input("整数を入力してください");
print(a);
```

input 関数の戻り値は string（文字列）型で、上ではそれを int 型変数 a で受け取っていますが、代入時に自動で変換されます。変換できない場合は実行時エラーとなります。VCSSL は静的型付けですが、このあたりは小規模用途での小回りを優先して、やや緩い感じになっています。もし入力された値が数値に変換可能か確認したい場合は、先頭で「import system.Int;」した上で isInt(string) 関数で確認できます。float も system.Float の isFloat(string)があります。

なお、「はい」/「いいえ」を問う confirm 関数や、ファイル選択用の choose 関数などもあります：

```
if( confirm("ファイルを選択しますか?") ) {
        string filePath = choose("ファイルを選んでください", "./");
        print("選択されたファイル = " + filePath);
}
```

VCSSL の主な文法・仕様

それでは、VCSSL の主な文法と仕様について、まとめて見てみましょう。ただし、即席ガイドという趣旨から、あまり深く長い解説はしません。読者がすでに C 系の言語（C/C++など）に関する予備知識があるものとの前提に立って、簡潔に駆け足でまとめていきます。

変数と型、演算

・変数の宣言

変数の宣言は以下のようにします：

```
int a;        //宣言のみ
int b = 1;    //宣言と同時に初期化
const int N = 100;    //  変更できない変数（定数）
```

変数はグローバル領域でもローカル領域でも宣言できます。ただしグローバル変数は同じ型なら重複宣言が許されますが、ローカル変数では許されません。また、グローバル変数は宣言行の前であっても参照できますが、ローカル変数は宣言行の後でしか参照できません。なお、const の付いたグローバル定数は、全ての行の実行よりも先に初期化されます。

・プリミティブ型

基本的なプリミティブ型は int / float / complex / string / bool の 5 種です。この他にも多倍長拡張型などがありますが、この 5 種だけ覚えておけば普通に使うには十分です：

```
int a = 1;    //整数型
float f = 2.3;    //浮動小数点数型
complex c = 4.0 + 5.0*I;    //複素数型（I は虚数単位定数）
string s = "Hello";    //文字列型
bool b = true;    //論理型
print(a, f, c, s, b);
```

実行すると「1　2.3　(4.0,5.0)　Hello　true」と表示されます。

float と int の精度は処理系定義で、普通は float が倍精度（double）です。int も符合付き64bit の精度です。一応 double / long も使えますが、float / int と同一視されます。complex の実部と虚部（re 関数と im 関数で取得可能）も float と同じものです。string は値型ですが NULL 許容型で、NULL との比較や代入ができます。

・プリミティブ型の演算

int、float、complex 型については、各種の四則演算や比較演算が行えます。int と float を混ぜた四則演算結果は float 型に、int と complex または float と complex を混ぜた四則演算結果は complex 型になります。なお、complex 型では大小比較は行えません。

string 型は文字列を扱う型で、参照型ではなく値型です。四則演算では加算のみが可能で、文字列の結合となります。string 型と他の型との加算結果は string 型になります。

bool 型は論理型（真偽型）で、論理演算が行えます。VCSSL では比較演算の結果は bool 型で、if 文などの条件式は、必ず bool 型でなければいけません。

・配列

続いて配列ですが、最初に一つ注意が必要な点があります。VCSSL の配列は、多くの C 系言語と異なり、参照型ではなく値型として振る舞います（そもそも VCSSL では、GC を不要にして実装をコンパクトに抑えるため、参照型が存在しません）。つまり、配列同士の代入演算は、参照の代入ではなく、全要素の値のコピーとなります。同じデータ領域を参照するようにはなりません。

さて、VCSSL 配列は以下のような形式になっています：

```
int a[3];   // int[3] a; も可能
a[0] = 0;
a[1] = 1;
a[2] = 2;
print(a);
```

実行結果は「0 1 2」です。多次元配列では以下のように宣言します：

```
int a[2][3];
```

配列は宣言と同時に初期化も可能です：

```
int a[ ] = {0, 1, 2};
```

上のように宣言時に要素数を省略した場合、初期化しなければ要素数 0 の配列となります。

配列の要素数を取得するには length 関数を使用します：

```
int a[3];
int n = length(a, 0);  // a の要素数を取得
print(a);
```

実行結果は「3」です。length 関数の第 2 引数は次元インデックスで、例えば 2 次元なら：

```
int a[2][3];
int n0 = length(a, 0);
int n1 = length(a, 1);
print(n0, n1);
```

のように、要素数を知りたい次元（左から 0, 1, 2, …）を指定します（実行結果は「2　3」）。

配列の要素数は、alloc 演算子で動的に変更できます：

```
int a[2];    // a の要素数は 2
a[0] = 0;
a[1] = 1;
alloc[3] a;    // a の要素数を 3 に変更
a[2] = 2;
print(a);
```

実行結果は「0 1 2」です。要素数変更後も、もともとあった要素の値は保たれます。

・構造体

型の紹介の最後として、構造体です：

```
//構造体の宣言
struct Box {
    int width;
    int height;
}
```

```
//構造体変数の宣言と使用
Box box;   //struct は不要、というより付けてはいけない
box.width = 100;
box.height = 200;
println(box.width, box.height);
```

実行すると「100　200」と出力されます。

構造体変数の宣言時に、struct キーワードは付けてはいけません。なお、構造体も値型で、参照型にすることはできません。構造体変数の代入は、全メンバの値のコピーとなります。また、構造型は NULL 許容型で、NULL との比較と代入ができます。禁止する機能は今のところありません。

ところで、現在の VCSSL の実装では、構造体はメモリ・処理速度共にハイコストです。要素数の多い配列や、高速に回る箇所では、あまり積極的に使うことはできません。

制御構文

・if(if-else)文

VCSSL では、制御構文は if(if-else)/ for / while の 3 種のみです。まずは if 文の例です:

```
int a = input("整数を入力してください",100);
if(a >= 10) {
    print("10 以上です");
} else if(a >= 5) {
    print("5 以上です");
} else {
    print("4 以下です");
}
```

実行すると入力値で分岐してメッセージが出力されます。条件式は bool 型限定です。

なお、VCSSL では、if / else / for / while のぶら下がり構文を禁止しているため、ブロックスコープの { } は省略できません。以下のように省略するとエラーとなります:

```
if(a >= 10) print("10 以上です");   //これはエラーとなる
```

ところで、「ぶら下がり構文が禁止ならば、else if というのはおかしいのでは? elif のような別の

構文が必要では…」という指摘があるかもしれません。これはその通りで、else の後に if がぶら下がっているのではなく、特例的に「else if」で一つの構文キーワードです。

・for 文

続いて for 文の例です：

```
int a = input("整数を入力してください",10);
for(int i=1; i<=a; i++) {
	println(i);
}
```

実行すると、入力値以下の正の整数を出力します。ただしコンソールへの出力処理は、演算処理よりも時間がかかるため、ループ回数を増やすと、出力完了までに時間がかかります。

・while 文

続いて while 文の例です。for 文の例と同じ処理を記述すると以下のようになります：

```
int a = input("整数を入力してください",10);
int i=1;
while(i<=a) {
	println(i);
	i++;
}
```

関数

VCSSL では、関数の宣言は基本的に C 系言語の標準的な形式を踏襲しています：

```
int add(int a, int b) {
	return a+b;
}
int c = add(1, 2);
print(c);
```

実行すると「 3 」と出力されます。
配列を引数や戻り値とするには、次のように記述します：

```
int[ ] add(int a[ ], int b[ ]) {
    …
```

なお、関数はオーバーロードが可能です。つまり、引数の型や個数が異なる関数は、別のものとして扱われるため、識別子（関数名）が競合しても問題ありません。なお、シグネチャが完全に競合してしまっている場合は、後に宣言されたほうの関数が有効になります。

動的解釈

インタプリタ形式の処理系では、eval 関数により、実行時に動的な式の解釈が行えます：

```
string expr = "1+2";
int a = eval(expr);   //文字列を式として動的評価
print(a);
```

実行すると「3」と出力されます。レキシカルスコープで、代入を行う事も可能です：

```
string expr = "a=1+2";
int a;
eval(expr);
print(a);
```

eval の戻り値は string 配列で、int や float の非配列に代入するには変換コストがかかるため、ループ時などでは後者のほうが高速です。なお、ループ時の eval の構文解析コストについては、処理系側で色々とキャッシュされるため、ほとんどありません。処理系の実装にも依存しますが、概ね静的解釈と大差ないパフォーマンスが得られます。

ファイル入出力

数値計算などでは必須となるファイル入出力についても、簡潔にまとめておきます。

・簡易ファイル入出力

まずは最も単純な、簡易ファイル入出力です。あまり複雑な事はできませんが、簡単です：

```
save("test.txt", "Hello World"); //書き込み
string s = load("test.txt");  //読み込み
println(s);
```

実行すると、「test.txt」というファイルが生成されて「Hello World」と書き込まれ、さらにそれが読み込まれてコンソールに表示されます。

なお、適当な箇所で改行したければ、書き込み内容に定数 EOL を挟んでください：

```
save("test.txt", "Hello" + EOL + "World"); //書き込み
```

実行すると、改行を挟んで「Hello」「World」と書き出されます。EOL は環境依存の改行コードを表す文字列定数です。もし、いわゆる LF(0x0A)を書き出したければ定数 LF を、CR(0x0D)を書き出したければ定数 CR を使用してください。

・基本的なファイル入出力

もう少し高度なファイル出力の基本形は、以下のようになります：

```
int file = open("test.txt", "w");  //書き込みモード「w」でファイルを開く
writeln(file, "Hello");  //Hello と書き込んで改行
writeln(file, "World");  //Hello と書き込んで改行
close(file);  //ファイルを閉じる
```

実行すると、改行を挟み「Hello」「World」書かれたファイル「test.txt」が生成されます。

open 関数はファイルを開き、複数のファイルを識別するための番号を割り振って返します。第 2 引数の "w" は書き込みモードを意味します。追記モードなら "a" とします。

writeln 関数は、ファイルに書き込んで改行する関数で、改行を行わない write 関数も存在します。使い方は同じで、第 1 引数が対象ファイルの番号、第 2 引数以降（可変長）が書き込み内容です。

続いてファイル入力です。基本形は以下の通りです：

```
int file = open("test.txt", "r");  //読み込みモード「r」でファイルを開く
int n = countln(file);  //行数をカウント
for(int i=0; i<n; i++){
    string line = readln(file);  //ファイルから 1 行読み込む
    println(line);  //コンソールに表示
}
close(file);  //ファイルを閉じる
```

「test.txt」と同じ場所で実行すると、その内容がコンソールに出力されます。

open 関数の第 2 引数の "r" は読み込みモードを意味します。readln 関数は、ファイルから 1 行の内容を読み込む関数です。実は readln 関数が返すのは配列なのですが、"r" モードの時は行内容をそのまま要素数 1 で返し、このように非配列変数(line)で受け取れます。

・数値 CSV 形式や数値 TSV 形式のファイル入出力

open 関数は、数値 CSV/TSV 形式のファイルを扱えます。まずは書き込みです:

```
int file = open("test.csv", "wcsv");  //CSV 書き込みモードでファイルを開く
writeln(file, 0, 1, 2);  //数値をコンマ区切りで書き込んで改行
writeln(file, 3, 4, 5);  //数値をコンマ区切りで書き込んで改行
close(file);  //ファイルを閉じる
```

実行すると、ファイル「test.csv」に、改行を挟んで「0,1,2」「3,4,5」と書き出されます。

続いて読み込みです:

```
int file = open("test.csv", "rcsv");  //CSV 読み込みモードでファイルを開く
int n = countln(file);  //行数をカウント
for(int i=0; i<n; i++){
    string line[ ] = readln(file);  //ファイルからコンマ区切りで 1 行読み込む
    println(line);
}
close(file);  //ファイルを閉じる
```

実行すると先ほどの「test.txt」を読み込み、改行を挟んで「0　1　2」「3　4　5」と出力します。このように、rcsv モードでの readln 関数は、1 行をコンマ区切りで読み込み、配列で返します。

ここで扱った wcsv / rcsv モードでは数値 CSV 形式ですが、同様に wtsv / rtsv モードで数値 TSV 形式も扱えます。ただし、これらの機能は、あくまで「数値」を対象とする、単純なものであるという点に注意が必要です。文字列を CSV 形式や TSV 形式で正しく扱うには、file.TextFile ライブラリ(https://www.vcssl.org/ja-jp/lib/file/TextFile)を使用してください。

ライブラリの読み込み

文法面は大体まとめ終えたので、この節の最後に、ライブラリの扱いについて説明しておきます。

・最初にライブラリを用意

まずは、以下の内容のファイルを「 TestLib.vcssl 」という名前で、「 testdir 」というディレクトリ(フォルダ)の中に作ってください:

― ./testdir/TestLib.vcssl ―

```
void fun( ) {
    println("Hello");
}
```

・include

上で作ったライブラリを読み込みます。C/C++などでは include がありますが、これは VCSSL でも使えます。先ほどの testdir ディレクトリ(フォルダ)と同じ場所のプログラムから読み込むには:

```
include "./testdir/TestLib.vcssl" ;
fun( );
```

実行すると「Hello」と出力されます。このように、include の後には、「そのファイルから見た相対パス」を " " で囲って記述します。include は C/C++と同じく、ライブラリのコードがその箇所にそのまま展開されます。しかしよく知られる通り、識別子の競合や多重 include などの面倒もあります。

・import

include で生じる面倒を回避した、より高度なライブラリ読み込み方法として、import も使用できます。VCSSL では基本的に include よりも import が推奨です:

```
import testdir.TestLib ;
fun( );
```

実行すると「Hello」と出力されます。このように、import の後には、「実行するファイル（ライブラリではない）から見た相対パス」を、スラッシュ区切り（ / ）の代わりにドット区切り（ . ）で指定します。

・モジュールと名前空間

import で読み込んだライブラリは、include と違ってその場に展開はされず、独立な形で読み込まれます。この独立なプログラム単位をモジュールと呼び、多重 import しても唯一性が確保されます。全てのモジュールは互いにグローバル領域を参照できますが、暗黙的な名前空間も持ちます。複数モジュールで識別子（変数名や関数名）が競合している場合、自身のモジュールのものが優先的に参照されますが、所属名前空間を指定する事により、別モジュールのものも参照できます：

```
import testdir.TestLib ;

void fun( ) {
      println("World");
}
TestLib.fun( );   // 名前空間を明示すれば TestLib の fun が呼ばれる
fun( );    // 名前空間を明示しないとこのモジュールの fun が呼ばれる
```

実行すると改行区切りで「Hello」「World」と出力されます。

・標準ライブラリ

VCSSL の標準ライブラリは、以下の Web ページで参照できます：

```
― VCSSL ライブラリ リファレンス ―
    https://www.vcssl.org/ja-jp/lib/
```

なお、実はこれまでの解説でも使用していた標準ライブラリがあります。それは「System」ライブラリで、例えば print / println 関数なども System ライブラリが提供しています。System ライブラリは、特例的に全モジュールよりも先に自動で読み込まれるため、import 不要で使用できます。

GUI と 2D/3D グラフィックス

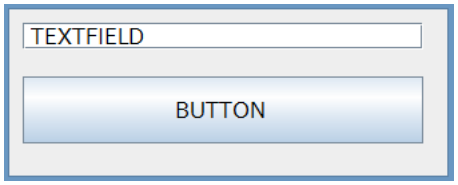
前節で、VCSSL の文法・仕様に関する説明はほぼ終わりました。即席ガイドとしてはここで終わるべきかもしれません。しかしここで切るとまさに「C でいいではないか」という内容だけになってしまうので、標準ライブラリから、GUI と 2D/3D グラフィックスを選んで簡単にまとめておきます。

GUI

まず、VCSSL で GUI を扱うサンプルコードは以下のようになります：

```
import GUI;

// ウィンドウ、テキストフィールド、ボタンを生成
int window = newWindow(0, 0, 350, 180, "WINDOW");
int textField = newTextField(10, 10, 300, 20, "TEXTFIELD");
int button = newButton(10, 50, 300, 50, "BUTTON");

// テキストフィールドとボタンを、ウィンドウ上に配置
mountComponent(textField, window);
mountComponent(button, window);

// ボタンが押された際に呼ばれるイベントハンドラ関数
void onButtonClick(int id, string label) {
    string text = getComponentText(textField);    // テキストフィールドのテキストを取得
    alert(text);    // ダイアログで表示
}
```

▼ 実行結果



実行すると、テキストフィールドとボタンが並ぶウィンドウが表示され、ボタンをクリックすると、テキストフィールドの入力内容を表示します。

上のコードで、new～関数は GUI 部品を生成する関数です。戻り値は int ですが、これは各 GUI 部品を区別するための ID 番号が返されます。それを int 型変数に格納しています：

```
int window = newWindow(0, 0, 350, 180, "WINDOW");
```

ID 番号は、早い話が処理系内部でのリソースの参照みたいなものですが、VCSSL では参照型やポインタの仕組みが無いので、3D などでもシステムリソースの参照は基本的に “むき出し” の int で扱います。（これについては静的型付けの利点を放棄しているわけで、まずかったと思っています。将来的には互換を保ちつつ、型検査が効くような、何らかの対策を取るかもしれません。）

ところで実装面の話ですが、実はこういった new～関数の実装には、現状であまり好ましくない挙動が残っています。それは ID 番号が「 0 から順に割り振られる 」という事です。そして int 型のデフォルト値も「 0 」です。つまり初期化を忘れた ID 格納変数でも、0 番の GUI 部品を参照している状態と見なせるため、初期化を忘れるとデバッグの難しいバグを招きます。これを防ぐためには、かなり後付けで強引な仕様ですが、int を NULL で初期化する事によって対応できます：

```
int window = NULL;    // NULL で初期化すれば、ID の格納を忘れても大丈夫
…
window = newWindow(0, 0, 350, 180, "WINDOW");
```

int に NULL を代入すると、「 システムリソースの ID 番号で絶対に割り振られない値 」が代入されます。ID を格納するのを忘れて使っても実行時エラーで落ちます。普通に new～関数が 1 以上を返すように実装を変えれば済むのですが、現状は互換問題から上のような解決策になっています。

続いてイベントハンドラを見てみます：

```
//  ボタンが押された際に呼ばれるイベントハンドラ関数
void onButtonClick(int id, string label) {
   string text = getComponentText(textField);    //  テキストフィールドのテキストを取得
   alert(text);    //  ダイアログで表示
}
```

イベントハンドラとは、ボタンのクリックなど、外部からの操作があった際に、それに対応した処理を行わせるためのもので、VCSSLでは上のような普通の関数です。しかし上の関数をボタンに紐づけるような記述は、コード中のどこにもありません。実は VCSSL では、「onButtonClick」という名称（と適切なシグネチャ）の関数は、自動でボタンのイベントハンドラと見なされ、全てのボタンに紐づけられます。つまり、どのボタンを押しても上の関数が呼ばれます。では複数のボタンがある場合はどう区別するのかというと、引数 id に、押されたボタンの ID が渡されるので、それで区別します：

```
void onButtonClick(int id, string label) {
   if(id == buttonA) {
      (ボタン A が押された場合の処理)
   }else if(id == buttonB) {
      (ボタン B が押された場合の処理)
   }
}
```

この方式は、ボタンが多いと面倒ですが、少ない場合は手短に書けます。VCSSL は後者優先です。

さて、ここで扱ったのはテキストフィールドとボタンだけですが、概ね VCSSL で GUI をどう扱うかという雰囲気はまとめられたと思います。より詳しくは、以下の資料をご参照ください。

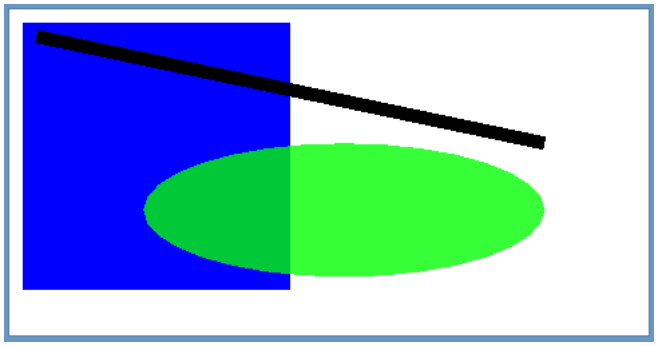
VCSSL GUI 仕様書 - https://www.vcssl.org/ja-jp/lib/GUI
VCSSL GUI ガイド - https://www.vcssl.org/ja-jp/doc/gui/

2D グラフィックス

続いて 2D グラフィックスです。まずはサンプルコードです：

```
import GUI;
import Graphics;
import Graphics2D;
import graphics2d.Graphics2DFramework;

void onStart(int rend){   // ここで初期化を行う
   setWindowSize(500, 300);      // ウィンドウサイズを設定
}
void onPaint(int rend){   // ここで描画を行う
   setDrawColor(rend, 0, 0, 255, 255);          //描画色の設定
   drawRectangle(rend, 10, 10, 200, 200, true);    //四角形描画
   setDrawColor(rend, 0, 255, 0, 200);          //描画色の設定
   drawEllipse(rend, 100, 100, 300, 100, true); //楕円描画
   setDrawColor(rend, 0, 0, 0, 255);            //描画色の設定
   drawLine(rend, 20, 20, 400, 100, 10);       //線描画
}
```

▼実行結果



上のコードは説明を手短に済ませるために 2DCG 用フレームワークを使っています。「 import graphics2d.Graphics2DFramework; 」で読み込んでいるのがそれで、標準で使えます。何をやってくれるかというと、ウィンドウや描画エンジンを生成したり、アニメーション用のループを回したり、フレームレートの調整を行ったりしてくれます。このフレームワークは全て VCSSL で書いてあり、処理系の lib/graphics2d ディレクトリの中にあるので、実装を見たい場合はそちらを参照してください。フレームワークを使わず、これらの処理をゼロから実装する事も可能です。

上のコードで onStart 関数は、起動後に 1 度だけフレームワークから呼ばれ、初期化処理を記述します。onPaint 関数は毎秒数十回呼ばれ、画面の描画処理などを記述します。時刻のカウンタ変数を作っておいて、毎回変化する内容を記述すれば、そのままアニメーション描画もできます。

さて、肝心の描画部分を説明します：

```
void onPaint(int rend){    //  ここで描画を行う
   setDrawColor(rend, 0, 0, 255, 255);          //描画色の設定
   drawRectangle(rend, 10, 10, 200, 200, true);     //四角形描画
   …
```

まず onPaint 関数の引数 rend ですが、これは描画エンジンの ID が渡されます。GUI 部品を生成する new～関数を思い出してください。あれと同じで、描画エンジンにも int 型の ID が割り振られます（本来は描画エンジンも newGraphics2DRenderer 関数で、必要なら自分で何個でも生成できますが、今回はフレームワーク裏でやってくれていて、それが引数で渡されています）。

続いて描画を行う関数です。これらは第一引数に、描画エンジンの ID を指定します。これは、例えば描画エンジンを 2 個生成して合成するような場合などに、どちらの描画エンジンへの命令かを区別するためです。setDrawColor 関数は描画色を設定する関数で、0～255 の範囲で RGBA（赤,緑,青,不透明度）値を指定します。drawRectangle 関数は四角形を描画する関数で、引数は「描画エンジン ID, X, Y, 幅, 高さ, 塗りつぶしの有無」です。

最後に、「S」キーを押すと画像を保存するようにしてみましょう。上のコードの末尾に追記します：

```
//「S」キーが押されたら画像を保存する
void onKeyDown(int id, string key){
  if(key == "S"){
    exportGraphics(getGraphics(), "test.jpg", "JPEG", 100.0);  // JPEG(100%)で保存
  }
}
```

onKeyDown はキー入力のイベントハンドラです。画像の保存には Graphics ライブラリ exportGraphics 関数を使用しています。この関数は第一引数に「グラフィックスデータ ID」という、つまるところ描画用バッファ（オフスクリーンバッファ）の参照を渡す必要があるのですが、フレームワークが getGraphics という関数の戻り値で返してくれるので、それをそのまま渡します。

2D グラフィックスの使い方は大体こんな調子です。より詳しくは、以下の資料をご参照ください。

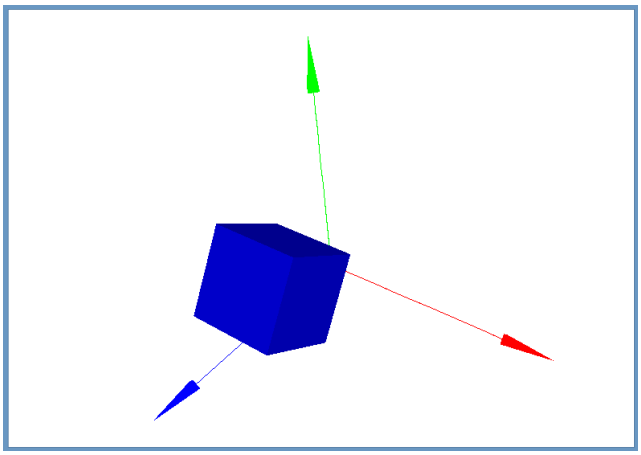
VCSSL Graphics2D ライブラリ 仕様書 - https://www.vcssl.org/ja-jp/lib/Graphics2D
VCSSL 2DCG ガイド - https://www.vcssl.org/ja-jp/doc/2d/

3D グラフィックス

最後に 3D グラフィックスです。サンプルコードは以下の通りです：

```
import GUI;
import Graphics;
import Graphics3D;
import graphics3d.Graphics3DFramework;

void onStart(int rend){  // ここで初期化を行う
  setWindowSize(800, 600);  // ウィンドウサイズを設定
  int axis = newAxisModel(3.0, 3.0, 3.0);     // 座標軸モデルを生成
  mountModel(axis, rend);                     // 描画エンジンに配置登録
  int box = newBoxModel(1.0, 1.0, 1.0);  // 箱型モデルを生成
  setModelColor(box, 0, 0, 255, 255);     // 色設定
  rotModelX(box, 1.0);                    // 回転
  moveModel(box, 0.5, 1.0, 1.5);          // 平行移動
  mountModel(box, rend);                   // 描画エンジンに配置登録
}
```

▼実行結果



2D の場合同様、3D でも簡潔に済ませるためにフレームワークを使用しました。このフレームワークも標準で使えるもので、基本的に 2D/3D で共通設計になっています。なので、概ね同じような感覚で使用できます。

onStart 関数も 2D の場合と同様、起動時にフレームワーク側から 1 度だけ呼ばれる関数で、ここで初期化処理を行います。引数には描画エンジンの ID が渡されます。この ID は立体の配置登録に使います。

```
void onStart(int rend){    //  ここで初期化を行う
  setWindowSize(800, 600);    //  ウィンドウサイズを設定
  int axis = newAxisModel(3.0, 3.0, 3.0);       //  座標軸モデルを生成
  mountModel(axis, rend);                       //  描画エンジンに配置登録
  …
```

newAxisModel 関数は、座標軸の形をしたモデルを生成する関数で、戻り値はモデルに固有の ID が返されます。自分でポリゴンを定義してモデルを自作する事もできますが、単純なものは最初から用意されています。続いて mountModel 関数で、それを描画エンジンに配置登録しています。

3D では 2D のように直接的に描画を行っていくのではなく、上の通りモデルやポリゴンなどの立体オブジェクトを生成し、それを描画エンジンに配置登録するという形が基本になります。あとは描画エンジンが勝手に座標変換やシェーディングをした上でレンダリングしてくれます。ウィンドウへの表示などやアニメーションループ、フレームレート制御などもフレームワークがやってくれます。

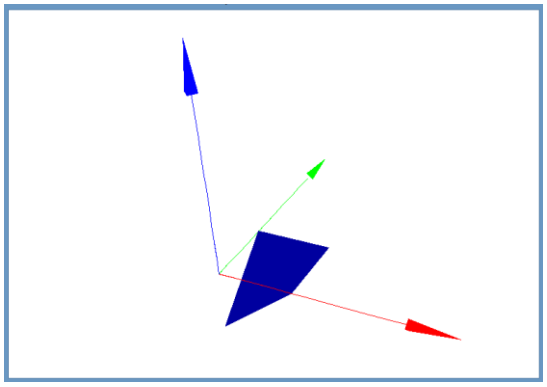
モデルの色設定や回転・移動などを行う関数は、第一引数にモデルの ID を渡します：

```
  setModelColor(box, 0, 0, 255, 255);      //  色設定
  rotModelX(box, 1.0);                     //  回転
```

場のシミュレーションなどでは、箱型のような立体モデルよりも、四角ポリゴンを直接描いて動かしたい場合もあるでしょう。実際に四角ポリゴンをアニメーションさせるコードを例示しておきます。

```
import GUI;
import Graphics;
import Graphics3D;
import graphics3d.Graphics3DFramework;
import Math;

int polygon = NULL; //  ポリゴンの ID 格納変数
double t = 0.0;         //  時刻変数
double dt = 0.01;     //  時刻変数の加算単位

void onStart(int rend){    //  ここで初期化を行う
   setWindowSize(800, 600); //  ウィンドウサイズを設定
   mountModel(newAxisModel(3.0,3.0,3.0), rend); //座標軸モデルの生成と配置

   //  四角形ポリゴンの生成（引数は頂点 XYZ 座標×4 頂点、しかしすぐ書き換えるので適当）
   polygon =newQuadranglePolygon(0.0,0.0,0.0, 0.0,0.0,0.0, 0.0,0.0,0.0, 0.0,0.0,0.0);
   setPolygonColor(polygon, 0, 0, 255, 255);     //  色設定
   mountPolygon(polygon, rend);                        //  描画エンジンに配置登録
}

void onUpdate(int rend){    //  ここで毎秒数十回のアニメーション更新処理を行う
   t += dt; //時刻変数を加算

   //  ポリゴンの座標値（正方形）
   float x0=0.0, y0=0.0, z0=0.0;
   float x1=1.0, y1=0.0, z1=0.0;
   float x2=1.0, y2=1.0, z2=0.0;
   float x3=0.0, y3=1.0, z3=0.0;
   x0=sin(t); y0=cos(t);     //  時間によって変形させる

   //  ポリゴンの座標値を更新（引数は頂点 XYZ 座標×4 頂点）
   setPolygonVertex(polygon, x0,y0,z0, x1,y1,z1, x2,y2,z2, x3,y3,z3);
}
```

▼実行結果



実行すると、四角ポリゴンの頂点の 1 つ (x0,y0,z0) が、X-Y 平面内で円を描くようにアニメーションします。上のコードでは、四角ポリゴンの頂点を setPolygonVertex 関数で書き換えた後、描画エンジンへの再登録などは行っていません。この通り、描画エンジンに一度登録した立体オブジェクトは、移動や変形などが自動で反映されます。

なお、VCSSL の 3D 機能の実装は、現時点で Z ソート形式/フラットシェーディングのみです。Z バッファ形式や、各種の補完シェーディング、テクスチャマッピングなどには対応していません。早い話がベタ塗りのカクカクで、ポリゴン同士が重なるような場合には前後関係を正しく描画できません。これはプラットフォームやハードウェアへの依存性を下げるため、GPU を使わないソフトウェアレンダリング式の 3D 描画エンジンを VCSSL 処理系に内蔵しているためで、あまり大がかりなものを載せられないという点があります。基本的に CPU 処理が大きなウェイトを占めるため、GPU を強化してもあまり効果はありません。パフォーマンスとしては概ね数十万ポリゴン/秒程度です。シミュレーションの可視化などはある程度こなせると思いますが、ポリゴン数の多い美麗なグラフィックの 3D ゲームを作ろうと思っても困難です。このあたりの限界については、あらかじめご了承ください。

さて最後ですが、画像の保存は 2D と同様、末尾に以下のようなコードを追記してください：

```
//「S」キーが押されたら画像を保存する
void onKeyDown(int id, string key){
  if(key == "S"){
    exportGraphics(getGraphics(), "test.jpg", "JPEG", 100.0);   // JPEG(100%)で保存
  }
}
```

3D グラフィックスの使い方は大体こんな調子です。より詳しくは、以下の資料をご参照ください。

VCSSL Graphics3D ライブラリ 仕様書 - https://www.vcssl.org/ja-jp/lib/Graphics3D
VCSSL 3DCG ガイド - https://www.vcssl.org/ja-jp/doc/3d/

外部言語連携

最後に、VCSSL から他言語のコードを実行する方法についてまとめます。

実行形式のプログラムを実行する（標準入出力のやり取りも可能）

単体で実行できるプログラムについては、exec 関数を使って VCSSL からも実行できます：

```
import File;    //  ファイルパス処理などを提供する標準ライブラリ

string path = getFilePath("test.exe"); //  実行したいプログラムの絶対パスを取得
exec(path, "aaa", "bbb", "ccc");            //  実行（非同期）
```

上の例では、VCSSL プログラムと同じ場所にある、何らかの言語で開発されたプログラム「test.exe」を、コマンドライン引数「aaa」「bbb」「ccc」を渡して実行します。

exec 関数は、基本的に処理をそのままオペレーティングシステムに依頼します。第一引数には実行対象のプログラムを指定しますが、実行時にシステムのカレントディレクトリがどこになっているかは、環境や処理系に依存するため、絶対パスで指定するのが無難です。第二引数以降（可変長）は、実行するプログラムに渡すコマンドライン引数を指定します。なお、exec の実行はバックグラウンドであり、非同期です。処理が完了するまで待機したい場合は system 関数を使用します：

```
import File;    //  ファイルパス処理などを提供する標準ライブラリ

string path = getFilePath("test.exe");      //  実行したいプログラムの絶対パスを取得
system(path, "aaa", "bbb", "ccc");           //  実行（完了するまで待機）
print("処理が完了しました。");
```

さらに、標準ライブラリの Process ライブラリを使用すれば、実行したプログラムと標準入出力でやり取りするなど、より高度なプロセス制御を行う事もできます。上手く活用すれば、C/C++などで開発した標準入出力ベースのプログラムに、VCSSL で GUI 部分や可視化部分を作るなどの使い道もあるかもしれません。

```
VCSSL Process ライブラリ 仕様書 - https://www.vcssl.org/ja-jp/lib/Process
```

Java の処理を呼ぶ（GPCI インターフェイス）

現在の VCSSL 処理系は Java 言語で開発しているため、Java 言語用のプラグイン・インターフェイスがあります。現状、Java から VCSSL を呼ぶことはできません。VCSSL から Java を呼ぶのみです。まずはプラグイン接続用インターフェイス「GeneralProcessConnectionInterface.java」を以下のように記述します：

― GeneralProcessConnectionInterface.java ―

```
public interface GeneralProcessConnectionInterface{
        public void init();
        public void dispose();
        public boolean isProcessable(String functionName);
        public String[] process(String functionName, String[] args);
}
```

これをコンパイルした上で、以下の例のように実装してプラグインを作成します：

― TestPlugin.java ―

```
public class TestPlugin implements GeneralProcessConnectionInterface {
  @Override
  public void init() { }  //初期化処理
  @Override
  public void dispose() { }  //終了時処理

  // 関数検索時に呼ばれる。このプラグインで処理する関数名なら true を返す。
  @Override
  public boolean isProcessable(String functionName) {
    return functionName.equals("___add");
  }

  // 関数コール時に呼ばれる。実際の処理を行う。
  @Override
  public String[] process(String functionName, String[] args) {
    // 引数 args を double として加算した数値を返す
    if(functionName.equals("___add") && args.length==2) {
```

```
        try {
            double a = Double.parseDouble(args[0]);
            double b = Double.parseDouble(args[1]);
            double c = a + b;
            return new String[]{ Double.toString(c) };
        } catch(NumberFormatException e) {
            return null;
        }
    }
    return null;
  }
}
```

これをコンパイルし、プラグイン TestPlugin.class を作成します。プラグインは複数併用できますが、それらを読み込むクラスローダの唯一性は保証されず、Singleton パターンなどの使用には注意が必要です。同一インスタンスを共有する処理は、同一プラグイン内にまとめる必要があります。

さて、上のプラグインは、引数を double として加算して返す「___add」関数を提供します（ただしプラグイン側は引数・戻り値共に String 配列として扱います）。この関数を VCSSL のコードから呼び出すには、TestPlugin.class を同じディレクトリに置き、以下のように記述します：

```
connect TestPlugin;          // プラグインの接続（TestPlugin の箇所は Java クラス名）
float a = ___add(1.2, 3.4);  // プラグインの関数をコール（戻り値の型は string 配列）
print(a);                    // 結果を出力
```

実行結果は「4.6」です。ところで GPCI 経由の関数は、VCSSL からは任意型の可変長引数を持つと見なされます。これはある意味強力ですが、引数の型・個数検査が効かないため、そのまま使うのは推奨できません。例えば上の関数を bool 型の引数 1 個でコールする事は、設計側の意図に明らかに反します。適切なシグネチャを持つ VCSSL の関数でラッピングして使うのが無難です：

```
connect TestPlugin;

float add(float x, float y) {   // プラグイン関数を、仕様に適切なシグネチャでラッピングする
  return ___add(x, y);
}
float a = add(1.2, 3.4);   // ラッピングした VCSSL の関数をコール
print(a);                  // 結果を出力
```

商標などに関して

[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] 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。

著者　松井文宏

このガイドの内容は、以下の Web ページでも公開されています：
https://www.vcssl.org/ja-jp/doc/cprogrammer/