Xtal 初心者入門と実践

2I 魯鈍ジョン

Xtal(くりすたる)とは

Xtal というのは、主にゲーム製作用途で C++と連携して使用する目的で作られたスクリプト言語のことです。同じような目的を持つスクリプト言語に、「Lua」「Squirrel」「AngelScript」などが挙げられますが、Xtal の場合そもそも C++で使用されることを前提としているので、導入も比較的簡単で、構文もそれなりに扱いやすいものとなっています。

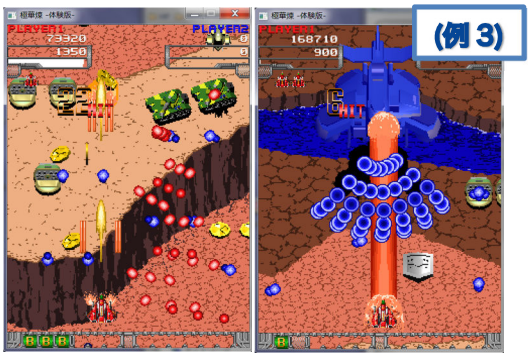
Xtal が使用されているゲーム

商業ゲームではあまり使用されておらず、主に同人ゲームで使用されています。もちろんですが、私も制作に役立てています。

（例１）諏訪子の弾幕ぴょんぴょん大散策
(制作サークル：UTG Software)
立体弾幕シューティングゲーム
執筆者が制作した STG です。
このゲームはちょっと特殊だけど。

（例２）双子魔法組曲
(制作サークル：永久る～ぶ)
ロールプレイングゲーム
RPG 制作のときは超便利です。ただやることが多いので覚悟を決めたほうがいいかもしれません。

（例３）極華煉
(制作サークル：極彩レヴェリ)
硬派なシューティングゲーム
友人のサークルの期待の新作です。



この記事の対象読者について

この記事は、C++がある程度（C 言語の知識、クラスやクラスの継承、STL の基礎的な使い方）が習得できていて、ゲーム制作やそれ以外でも手軽にスクリプト言語を使用してみたいという人を対象にしています。特に使用する開発環境は指定しませんが、Visual Studio を例に進めていきます。それ以外で開発している人でもわかるように基礎的な操作方法を記して説明しますし、C++以外の知識を前提としないように、必要なところは結構詳しく書いたつもりです。この記事に書いてあるとおりに操作すれば大丈夫かと思われます。

この記事の表記について

この記事にはソースコードが大量に掲載されています。そこで、コードの見やすさの観点から枠線の種類で区別をしています。点線枠内のコードが C++、二重線枠内のコードが Xtal のスクリプトです。コード枠線の上にどの種類のコードかは一応書いてありますが、そのように枠線で区別しています。また、DVD に収録されているカラー版では、C++のコードは、Visual Studio の初期設定の基準に沿ってある程度色分けされています。

筆者の環境について

使用している PC は、VAIO Z (2010 年秋冬モデル)です。構成としては、

OS　Windows 7 Professional 64bit
CPU　Intel Core i7 M640 (4 コアの 2.80GHz)
メモリ　8GB
開発　Visual Studio 2010 Professional

となっています。一応これを基準にして説明していますが、Visual Studio 2008 でもできると思います。Xtal はもともと Visual Studio 2008 を想定しているらしく、Visual Studio 2010 だとソースコードの修正が必要になりますが後で解説をします。

これから Xtal 導入を考えている人に言いたいこと

現在、Xtal のコミュニティ自体が存在しません。そう、あまり使用者がいないのです。そこで、みなさんに Xtal を使用して実績を作ってもらいたいのです。そのための記事です。筆者も極力わかりづらい部分がないように頑張って書いたつもりですので頑張ってください。

さぁ、導入も簡単で開発効率 UP も期待できる、すばらしい Xtal ワールドにご招待しましょう。(持ち上げすぎ)

Xtal の導入方法

まず何をすればいいのか。それは、Xtal のソースコートをダウンロードするところから始まります。公式サイトに上がっている最新版らしきものがありますが、実は結構古いバージョンなのです。使っても得しません。そこで最新版を落とす準備をします。Subversion(さぶばーじょん)というものを使用しますが、一昔前まではパスの登録など全部手作業で登録して面倒だったのですが最近は GUI で操作でき自動化されているのでどうということはありません。

SubVersion を導入する

いわゆるバージョン管理ツールなのですが、今回はこれで Xtal の任意のリビジョン(バージョンのこと)を落とせるようにします。

今回は、「TortoiseSVN」を利用します。インストール時に勝手に設定を行ってくれるので結構楽です。

まずは、公式ページ http://tortoisesvn.net/downloads.html へアクセスし、自分の CPU に合わせて 32bit/64bit を選択し、ダウンロードします。

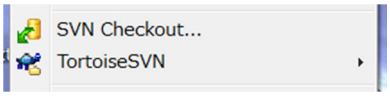
インストーラを起動し、適当にインストールをすれば大丈夫です。完了したら再起動しましょう。再起動しないと後で困ります。

最新版のチェックアウト（ダウンロード）をする

http://code.google.com/p/xtal-language/source/checkout へアクセスすると、
「svn checkout　http://xtal-language.googlecode.com/svn/trunk/ xtal-language-read-only」
という文字列を見つけられると思います。これは、コマンドプロンプトでこのコマンドを入力すれば、最新

版のチェックアウトができるということです。
しかし、TortoiseSVN は GUI プログラムなので、GUI で操作します。

まずは、適当にデスクトップで右クリックして、「SVN Checkout」をクリックします。



つぎに、チェックアウトの指定などを行うウィンドウが出てきますので、次のとおりに入力しましょう。

URL of repository のボックスに、「http://code.google.com/p/xtal-language/source/checkout」と入力し、Checkout directory のボックスに任意のインストール先を入力、Revision にラジオボタンを選択し、429 とボックスに打ちます。そして OK を押すとダウンロードが開始されます。
今回は、**C:\Users\(ユーザー名)\xtal-language-read-only** にインストールします。
これで完了です。ちなみに、Revision の代わりに HEAD revision を選択して OK 押すと最新版(執筆時はリビジョン 436)を落とすことができます。

ライブラリファイル(.lib)を作成する

VisualStudio2008 を使っている人と VisualStudio2010 を使っている人でやり方が違います。違いはおそらくここだけかと思われます。

VisualStudio2008 の場合

xtal-language-read-only フォルダ内の vc9 フォルダにある xtalproj.sln を開き、まずはビルド設定を Debug にしてそのままビルドします。すると、lib フォルダ内に xtallib.lib が生成されますので、xtallibd.lib とリネームしておきます。つぎに、ビルド設定を Release にしてビルドし、xtallib.lib を生成します。これで終了です。

VisualStudio2010 の場合

xtal-language-read-only フォルダ内の vc9 フォルダにある xtalproj.sln を開き、変換ウィザードを適当に完了させます。次にビルド設定を Debug にしてそのままビルドしようとすると、なんとコンパイルエラーを吐きます。ちょっとだけソースコードをいじらなければいけないのです。

そのソースコードのいじり方ですが、うまく説明できないので Xtal のコミッターである Sukai さんのコードを拝借します。※許可はいただいています。

xtal_base.h

```
// 100 行付近から
template<class T>
struct VirtualMembersOverwriteChecker{
        typedef char (&yes)[2];
        typedef char (&no)[1];
        // 他のは省略
        // 頭に check_ってつけただけ
        static no check_visit_members(void (RefCountingBase::*)(Visitor&));
        template<class U> static yes check_visit_members(void (U::*)(Visitor&));

        enum{
        // 他のは省略
        // こいつも check_ってつける
        visit_members_overwrite=sizeof(check_visit_members(&T::on_visit_members))!=sizeof(no),
        }
};
```

xtal_utility.h

```
// 284 行付近から
template<class T, class U>
struct Convertible{
        typedef char (&yes)[2];
        typedef char (&no)[1];
        static yes check_convertible(U);
        static no check_convertible(...);
        static T makeT();

        enum{ value = sizeof(check_convertible(makeT()))==sizeof(yes) };
};

template<class T, class U>
struct IsInherited{
        typedef char (&yes)[2];
        typedef char (&no)[1];
        static yes check_is_inherited(const U*);
        static no check_is_inherited(...);
        static T* makeT();

        enum{
                // 完全型チェック
                CHECK = sizeof(T) + sizeof(U),

                value = sizeof(check_is_inherited(makeT()))==sizeof(yes)
        };
};
```

このように書き換えれば OK です。指定した行番号に関しては、リビジョンごとに違ってくるので一応目安としてみてください。

これでビルドが通るはずです。あとは VisualStudio2008 の場合と同じように xtallibd.lib と xtallib.lib を生成すれば完了です。ちょっと面倒だったと思いますが、他のスクリプト言語よりはマシなのでなんとかがんばってください。

実際にプロジェクトで使ってみる

いよいよ実際に使ってみる感じになってきました。とりあえず今回はコンソールのプログラムで適当に動作確認をしてみたいと思います。一応 Visual Studio を使用したことがない人にも扱えるよう詳しくやっていきたいと思います。

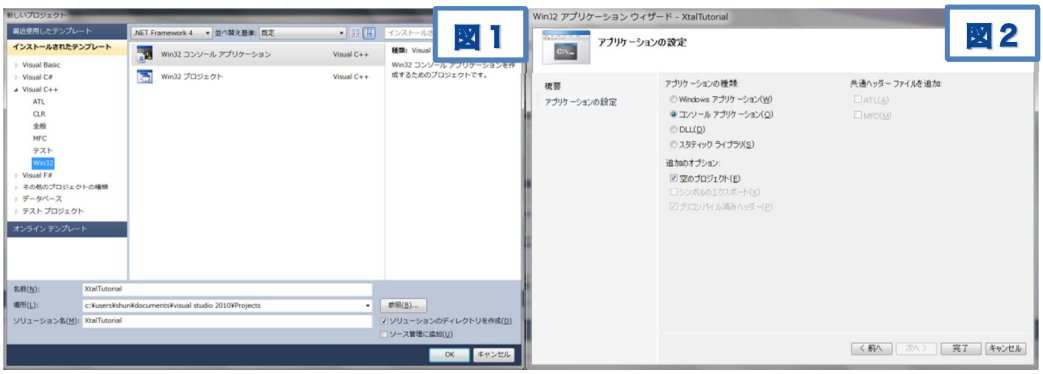
画面はすべて VisualStudio2010 ですが、VisualStudio2008 でもおなじようにやればできると思います。ただ VisualStudio2008 のほうがもう数年使ってないので覚えていません。すみません。

空の C++プロジェクトを作る

VisualStudio を起動し、ファイル→新規作成→プロジェクトを押します。
VisualC++ → Win32 → Win32 コンソールアプリケーションを選択します。

名前とソリューション名を適当に決めて(今回は XtalTutorial)OK を押します。(図 1)

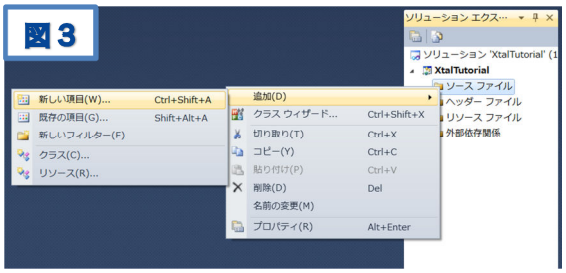
次に、Win32 アプリケーションウィザードが出てきたら、「追加のオプション」の「空のプロジェクト」にチェックを入れて、「完了」をクリックします。(図 2)これで、プロジェクトが作れるはずです。



ソースコードにファイルを追加する

画面の中に「ソリューションエクスプローラ」というウィンドウがあります。その「ソース ファイル」フォルダ（正式にはフィルタ）を右クリックし、追加→新しい項目を選択します。(図 3)

何を作成するか聞いてくるので、C++ ファイル(.cpp)を選択して、ファイル名を決め(今回は main.cpp)、追加をクリックします。これでプログラムが書けます。



一応テストコードを書いて確認してみましょう。

```
#include <iostream>

int main()
{
	std::cout<<"すわこちゃん"<<std::endl;

	//画面が一瞬で消えるのを防止
	fflush(stdin);
	getchar();

	return 0;
}
```

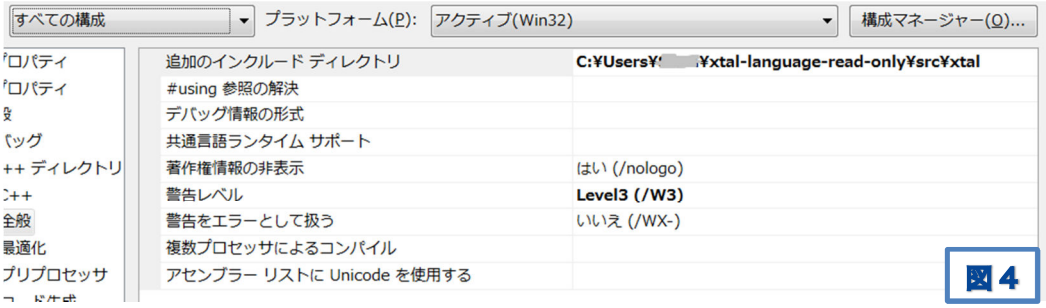
とりあえず実行してコンソール画面に「すわこちゃん」と表示されれば大丈夫です。
次はプロジェクトの設定をしましょう。

プロジェクトの設定をする

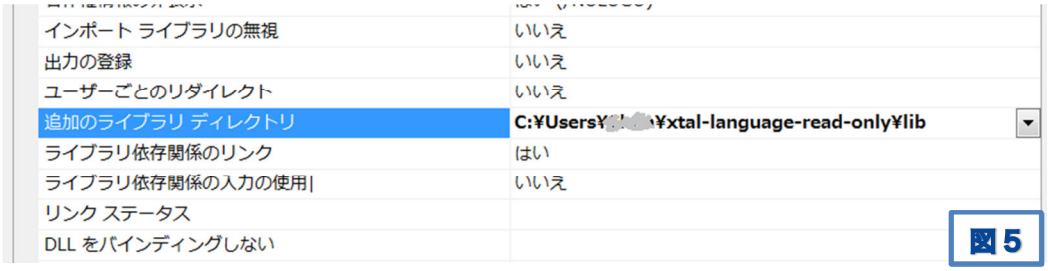
これぱかりはパターンなのでどうしようもないです。

ツールバーの　プロジェクト　→（プロジェクト名）のプロパティをクリックします。するとプロパティ画面が開きますので、「構成」の項目で、「すべての構成」を選択します。

構成プロパティ→C++→全般の、「追加のインクルードディレクトリ」に Xtal のソースコードのディレクトリを入力します。ここでは、さきほどインストールしたディレクトリのソースコードフォルダの C:\Users\(ユーザー名)\xtal-language-read-only\src\xtal　とします。(図 4)



次に、構成プロパティ → リンカー → 全般の、「追加のライブラリディレクトリ」に、Xtal のライブラリファイルのディレクトリを入力します。ここでは、C:\Users\(ユーザー名)\ xtal-language-read-only\lib とします。(図 5)



それ以外の設定として、
構成プロパティ→全般の「文字セット」を「マルチバイト文字セットを使用する」に選択しておきましょう。
どうしても Unicode で使用したい場合、先ほどの lib ファイル生成のビルド時に、プロパティから同様の設定項目を「Unicode 文字セットを使用する」にしてビルドしてください

これで Xtal のプログラミング環境が整いました。
次は、Xtal を使用する際の定石みたいなものを組みます。これもパターンしかないので覚えるというより使いまわせるようにしてください。

Xtal の定石コード

ここでは Xtal を扱う上で基本となるコードを掲載します。
Xtal を使用するうえで本当に基礎となるコードですが、ここに掲載したコードはあくまで例であり、そのまま実践で扱うと非常に使いづらいコードになると思われます。ある程度 Xtal に慣れてきたら自分で使いやすいようにライブラリを作ってみると良いかもしれません。

基本的に Xtal は、スクリプトを実行可能なバイトコードに変換し、そのバイトコードを実行して動作するようになっています。いたって単純です。
流れとしては、まずその Xtal の初期化をし、コードを動かすための準備をします。そしてスクリプトをコンパイルしてバイトコードにし、実行します。そして、Xtal がプログラム上で不要になったときは解放を行います。基本はそのステップしか無いのでわかりやすいとは思います。

それでは次のページからそのコードを掲載します。ちょっと長いです。初期化処理と終了（解放）処理を行うコードです。

main.cpp

```
#ifndef _DEBUG
#pragma comment (lib,"xtallib.lib")
#else
#pragma comment (lib,"xtallibd.lib")
#endif

#include <iostream>
#include <xtal.h>
#include <xtal_macro.h>                 // Xid など便利なマクロが定義されている
#include <xtal_utility.h>
#include <xtal_lib/xtal_cstdiostream.h> // CStdioStdStreamLib のため
#include <xtal_lib/xtal_winthread.h>    // WinThreadLib のため
#include <xtal_lib/xtal_winfilesystem.h>// WinFilesystemLib のため
#include <xtal_lib/xtal_chcode.h>       // SJISChCodeLib のため
#include <xtal_lib/xtal_errormessage.h> // bind_error_message()のため

//これらのクラスは、xtal::uninitialize()が呼ばれるまで残っていなければならない。
//そのためグローバルなスコープに書いておく。
xtal::CStdioStdStreamLib cstdStdStreamLib;
xtal::WinThreadLib threadLib;
xtal::WinFilesystemLib filesystemLib;
xtal::SJISChCodeLib chCodeLib;          // SJIS
//xtal::UTF16ChCodeLib chCodeLib;       //Unicode

int main()
{
        //Xtal の初期化
        xtal::Setting setting;
        setting.std_stream_lib = &cstdStdStreamLib;
        setting.thread_lib = &threadLib;
        setting.filesystem_lib = &filesystemLib;
        setting.ch_code_lib = &chCodeLib;

        xtal::initialize(setting);
        xtal::bind_error_message();

        //終了処理
        xtal::uninitialize();

        //画面が一瞬で消えるのを防止
        fflush(stdin);
        getchar();

        return 0;
}
```

とりあえずコードです。実行しても何も起きません。Xtal の初期化と終了処理だけしてプログラムを終了するだけです。

Unicode で使用する場合、lib ファイルを Unicode 設定でビルドした上で、

xtal::SJISChCodeLib chCodeLib;　の行をコメントアウトし、

xtal::UTF16ChCodeLib chCodeLib;　の行のコメントアウトを外してください。

とりあえず関数名からして、どういう処理がなされているのかは容易に想像ができると思います。直感的に理解するのも必要です。

コードをだらーっと書いてしまいましたが、このコードが Xtal の基礎となる部分です。これらのコードを自分で使いやすいように改造していけばいいのです。

Xtal のスクリプトを実行してみよう

実行自体はものすごく簡単です。xtal::compile();にスクリプトの文字列(char*)を渡して、返されたポインタに対して->call();をするだけでいいのです。

Unicode の場合でも、文字列を w_char*で渡せば問題ありません。

紙面の都合上、main 関数部分だけを書きます。

main.cpp (main 関数の部分)

```
char script[] = "100.p;";
int main()
{
	//Xtal の初期化
	xtal::Setting setting;
	setting.std_stream_lib = &cstdStdStreamLib;
	setting.thread_lib = &threadLib;
	setting.filesystem_lib = &filesystemLib;
	setting.ch_code_lib = &chCodeLib;

	xtal::initialize(setting);
	xtal::bind_error_message();

	// 文字列をコンパイル
	xtal::CodePtr pCode =xtal::compile(script);

	//実行
	pCode->call();

	//終了処理
	pCode = xtal::null;		//コードの解放処理
	xtal::uninitialize();

	//画面が一瞬で消えるのを防止
	fflush(stdin);
	getchar();

	return 0;
}
```

ね？簡単でしょう？これだけで Xtal のコードを実行できます。

char script[] の中身はもちろんスクリプトです。この中身である

100.p;

の意味は、コンソール画面に 100(数値)を表示しろということです。一応デバッグプリントと呼ばれるらしいです。また、終了処理時に pCode に xtal::null を代入してコードを開放しておくと、実行時に assert が出なくなります。

このように、Xtal のコードの実行はとても簡単にできます。しかし、これだとスクリプトをプログラムに直接書くので意味がありません。そこで、ファイルから読み込んでみましょう。

ファイルから Xtal のスクリプトを読み込む

xtal::compile_file();にファイル名を渡せば、スクリプトが書かれたテキストファイルをコンパイルできます。もちろん、ポインタを返すので、そのポインタに対して->call();をするだけで実行できます。Unicode の場合、テキストファイルのエンコードも Unicode でなければなりません。当たり前ですが。

紙面の都合上、main 関数部分だけを書きます。

main.cpp (main 関数の部分)

```
int main()
{
	//Xtal の初期化部分は省略

	//外部ファイルをコンパイル
	xtal::CodePtr pCode =xtal:: xtal::compile_file ("script.xtal");

	//実行
	pCode->call();

	//終了処理以降も省略
}
```

カレントディレクトリの script.xtal を読み込んで実行します。カレントディレクトリは、基本的には実行ファイルと同じディレクトリか、ソリューションファイル(.sln)が置かれるディレクトリ内にあるプロジェクト名と同名のフォルダ内かのどちらかになります。今回場合は後者のディレクトリに置きます。main.cpp と同じディレクトリですね。

とりあえず、試しに script.xtal の中身を

script.xtal

```
100.p;
```

と書いて保存し、実行してみましょう。100 と表示されれば大丈夫です。

ちなみに、Xtal の構文は C/C++と似ているため、最後にはセミコロンを必ず付けてください。

さて、これで Xtal のスクリプトファイルをコンパイルできるようになりました。これでようやく次から Xtal の構文解説に入れます。長かったので休憩しましょう。休むことも大事です。

ちょっとブレイク

一応 Xtal はこんなこともできるんだよ！という宣伝の一環として、コミックマーケット 79(当時 X680x0 同好会は落選)にて頒布された『Xtal 合同ミニゲーム集「Xtal プログラマはせつなくてスクリプトを組み込むとすぐゲーム作っちゃうの」』というものがありました。もちろん筆者も参加しています。

ソースコード付きで、いろんな人の Xtal の技法を学べると好評でした。たぶんね。

詳細は以下の URL からどうぞ

http://sukai.sakura.ne.jp/c79.html

Xtal の構文解説

ここからは、Xtal の構文解説をしていきます。
いくつかの構文解説には、同じく Sukai さんのコードを拝借しました。

デバッグプリントについて

100.p にあったこの.p、これがデバッグプリントです。
コンソール画面に出力できれば、ゲームのデバッグに便利に使用できます。

```
100.p;              //100 と表示される
"すわこちゃん".p;   //すわこちゃんと表示される
(123+456).p;        //123+456 の結果、つまり 579 と表示される
```

計算の順序について

基本的に C++と変わらず、括弧で計算順序を指定できます。

```
(3+7*2).p;          //17
((3+7)*2).p;        //20
```

変数の宣言

コロンを使用します。動的型付け言語であるので、int などの型指定子は不要です。

```
a:20;     //変数の宣言
a.p;      //20 と表示される
a=40;     //a に 40 を代入（代入はイコールで）

b,str : 40,"すわこちゃん";    //コンマで分けて宣言も可能
(a+b).p;                      //80 と表示される
str.p;                        //すわこちゃんと表示される
```

関数の宣言と呼び出し

fun というものを使います。もちろん return で値も返せます。
int , float , bool は値渡しで、残りは全て参照渡しとなります。

```
fun add(a,b)
{
        return a+b;
}
add(50,100).p    //150
```

繰り返し構文

C++みたいなものも使用可能ですが、Ruby 的な繰り返しも使用できます。
特に Ruby 風のブロック構文はお手軽に書けるので大変便利です。

```
for (i : 0; i < 10; ++i)
{
    i.p;
}

count : 0;
while (count < 10){
    count.p;
    ++count;
}
```

```
// ブロック構文
// 配列に順番にアクセスできる
array : [1, 2, 3, 5, 8, 13];
array{
    // "it"という変数が勝手に定義されて使えるようになる
    it.p;
    // 代入もできる
    it = it*2;
    it.p;
}
```

条件分岐

if , switch-case , 三項演算子が使用可能です。

switch-case 構文はちょっと特殊です。

```
// if, else if, else は C++式
x : 10;
if (x == 0) {
    "zero".p;
}
else if ((x % 2) == 0){
    "even".p;
}
else {
    "odd".p;
}


// switch-case は C++と比べて少し特殊
y : 3;
z : "";
switch (y){
    case (0){
        z = "zero";
    }
    case (1){
        z = "one";
    }
    case (2){
        z = "two";
    }
    default {
        z = "bigger than two";
    }
}
z.p;

// 三項演算子
w : (10 >= 0)? "positive or zero": "negative";
w.p;
```

クラスの定義

メンバ変数の扱いがちょっと面倒かもしれません。使いこなせれば一番便利な機能だと思います。

```
// クラスを定義
class Foo{
    _bar;    // メンバ変数は最初に"_"をつける
    + _hoge; // 最初に+をつけることで外から***.hoge みたいにアクセスできるようにする

    // クラスインスタンス生成時に呼び出される
    // Foo(10); と生成されたときは bar=10 となる
    initialize(bar){
        _bar = bar;
    }

   // メソッド（メンバ関数）
    print(){
        _bar.p;
        _hoge.p;
    }
}   // C++と違って最後に";"がいりません

// クラスを使う
foo : Foo("bar!");
foo.hoge = "hoge?";            //+がついているため書き換え可能
foo.print();
```

シングルトン

C++でいう namespace みたいなものです。つまり、それ自身しか存在しないクラスのことです。

```
singleton abc{
    ConstValue : 1;            //書き換えできない
    +_Value : 2;               //書き換えできる
}
abc.Value = 5;      //書き換え可能
```

Xtal のデータ型について

Xtal にもやはり int , float , bool というデータ型が存在します。

整数・浮動小数点数

スクリプトの中で数値を使うときはこの型のどちらかになります。

```
i : 10;      // 整数型
f : 10.5;    // 浮動小数点数型

i2 : f.to_i // 整数型へ変換(10)
f2 : i.to_f // 浮動小数点数型へ変換(10.0)
```

真偽値

true/false です。

```
t : true;   // 真
b1 : 10 == 20;
b2 : 10 == 10;
b1.p;
b2.p;
```

文字列

”” で囲まれた部分は文字列型と認識されます。他の型から to_s によって変換することもできます。また、~(チルダ)演算子で連結も可能です。

```
str : "This is string";
str2 : 200.to_s;      //数値を文字列へ変換

str.p;
str2.p;
(str~" m(^o^)m "~str2).p;
```

配列

C++でいう std::vector、std::map みたいなものが存在します。

```
a:[2,4,3];    //配列の宣言(C++の std::vector みたいなもの)
b:[ "name":"すわこ","age":16,"weight":45];    //連想配列の宣言(C++の std::map みたいなもの)
c:[];         //空の配列の宣言
d:[:];        //空の連想配列の宣言

a[1].p;       //4 と表示される
a[2] = 8;     //もちろん代入も可能

b["name"].p;          //すわこ
b["age"].p;           //16
b["weight"].p;        //45
```

配列では push_back や push_front , clear や erase など std::vector にありそうなものはだいたい使用可能です。

```
a:[2,4,3];     //配列の宣言
a.size().p;    //要素数を返すので、3 が表示される
a.push_back(5);    //5 が後ろに追加される
a[3].p;            //5
a.push_front(6);   //6 が頭に追加される
a[1].p;            //2
a.erase(1);        //要素の 1 番が消える(先頭は 0 番)
a[1].p;            //4
a.clear();         //全要素を消す
a.size().p;        //0
```

連想配列では、好き勝手に要素に代入して値を追加することができます。

```
b:[:];     //空の連想配列の宣言
// []のなかにはどんなものでも入れることができます
b[0] = "foo";
b["bar"] = 10;
```

一応これで基本的な構文解説はひと通りできたと思います。もう少し特殊な書き方があったり、別の便利な機能があったりしますが、これから話を勧めていく上で出てきたり、特に必要ないものもあったりするので書いたり書かなかったりします。

次は、地味に使いやすい「グローバル変数」についてです。地味に便利です。
まぁあんまり使わないほうが好ましいのではありますが、やっぱり便利なので使ってしまいますね。

グローバル変数について

global:: をつけることでグローバル変数やグローバル関数を定義、アクセスすることができます。別のスクリプトからもアクセス可能なのでたまに使われます。

```
global::var : "this is global variable";
fun global::function(){
    "global function called".p;
    global::var.p;
}

global::function();
global::var : "this is changed global variable";
```

しかし、値の変更ができないため、そのときは singleton を利用するなどして変数に書き換えられるようにするしかありません。

しかも、この global::は再定義（上書き）が可能なので注意が必要です。私もこの上書きによるミスでバグを引き起こしたことがあるので特に注意してください。

数学関数の使い方

スクリプトの最初に inherit(math); と書いておけば、三角関数などの数学関数を使用することが可能になります。

```
inherit(math);
sin(1.0).p;
cos(1.0).p;
atan2(2.0,1.0).p;
abs(-5).p;
pow(4,2.5).p;
```

なんだかんだ、ゲームプログラミングに必須な数学関数であるので、知っておくと便利かもしれません。

こうして Xtal の基本的な構文を紹介して行きましたが、C++に似ているのでかなり書きやすいのではないかなと思います。慣れてくれば本当に素早く製作可能です。

次は、エラー処理を実装します。このエラー処理とは、コンパイルエラーや実行時エラーを検出し、スクリプトの構文ミスを修正したりデバッグしたりします。これがないとどうしようもないですからね。それではまた休憩。もうかれこれ十数時間は連続で書いているのでしんどいです。

Xtal のエラー処理

この機能があるかないかで相当変わってくるこの「エラー処理」。Xtal では謎のマクロを使用します。

C++のコード

```
XTAL_CATCH_EXCEPT(e){
std::cout << e->to_s()->c_str() << std::endl;
}
```

このようにすれば、コンソール画面にエラー内容が吐き出されます。私はメッセージボックスに表示しちゃいますが、せっかくコンソール画面があるのでそっちに出力しちゃいましょう。

このコードを具体的にどこに書けばいいのかというと、コンパイルや実行の直後に置きます。

念のため、エラー処理をもう一つつけましょう。コンパイルに失敗すると pCode に NULL が代入されるようになります。それを利用して次のように組みます。

main.cpp (main 関数の部分)

```
int main()
{
        //Xtal の初期化部分は省略

        // 文字列をコンパイル
        xtal::CodePtr pCode =xtal:: xtal::compile_file ("script.xtal");
        XTAL_CATCH_EXCEPT(e){
        std::cout << e->to_s()->c_str() << std::endl;
        }

        //実行
        if(pCode)
        {
            pCode->call();
            XTAL_CATCH_EXCEPT(e){
            std::cout << e->to_s()->c_str() << std::endl;
            }
        }

        //終了処理以降も省略
}
```

これでエラー処理はどうにかなるはずです。
試しにエラーを起こしてみましょう。

script.xtal

```
"すわこちゃん".p
"ひゃっほう".p;
```

セミコロンがありません。よくやるミスですね。この状態で実行してみるとどんなエラーが出るでしょうか。

```
lib::builtin::CompileError: XRE1016:ファイル 'script.xtal' のコンパイル中、コンパイルエラーが発生しました。
script.xtal:2:XCE1002:';'を期待しましたが'"'が検出されました。
```

このように出ました。ファイル名：行数：エラーID：エラーの内容 が吐き出されています。どこを修正すればいいかがわかるようになります。

紙面の都合上、中途半端になるのでエラー処理はここでおしまいにします。次はいよいよ C++の関数を Xtal で使用する方法について説明します。

C++の関数を Xtal で使用するには

C++の関数を Xtal で使用するには、C++の関数を Xtal 上での名前をつけて関連付けをします。これを「関数をバインドする」といいます。これも謎のマクロを使用します。Xtal の便利な機能はこのバインドの機能であって、C++に特化したためか超簡単に関数のバインドが行えます。

使い方ですが、

xtal::lib()->def(Xid(Xtal 側の名前), xtal::fun(バインドする関数のポインタ));

という感じになっています。Xtal 側の名前と関数名を一致させる必要はありませんが、同じにしておいたほうがいいでしょう。

main.cpp (main 関数の部分)

```
//バインドする関数
int func1()
{
	std::cout<<" func1 を実行しました。"<<std::endl;
	return 1;
}

int main()
{
	//Xtal の初期化部分は省略

	//関数のバインド
	//関数名の先頭に&をつければポインタになるよね
	xtal::lib()->def(Xid(func1), xtal::fun(&func1));

	// 文字列をコンパイル
	xtal::CodePtr pCode =xtal:: xtal::compile_file ("script.xtal");
	XTAL_CATCH_EXCEPT(e){
	std::cout << e->to_s()->c_str() << std::endl;
	}

	//実行
	if(pCode)
	{
	    pCode->call();
	    XTAL_CATCH_EXCEPT(e){
	    std::cout << e->to_s()->c_str() << std::endl;
	    }
	}

	//終了処理以降も省略
}
```

script.xtal

```
lib::func1().p;
```

実行結果は、

```
func1 を実行しました。
1
```

となるはずです。

Xtal のバインドでは、関数ポインタだけ渡せばあとは勝手に自動で引数の型や数を判別してくれるのでかなり便利です。しかし、Xtal 側で呼ぶ場合、lib::を付けなければなりません。よって lib::まみれなスクリプトが出来上がる場合がありますがまぁそれくらい慣れてください（何

それでは、練習問題として１題出題します。
渡された数の二乗を返す関数 int square(int x)を作ってバインドし、Xtal 側で５を渡して結果を表示させてみましょう。

main.cpp (main 関数の部分)

```
//バインドする関数
int square(int x)
{
	return x*x;
}

int main()
{
	//Xtal の初期化部分は省略

	//関数のバインド
	xtal::lib()->def(Xid(square), xtal::fun(&square));

	//以後省略
}
```

script.xtal

```
lib::square(5).p;
```

当然結果は 25 になります。

もちろん、文字列も渡すことができます。引数を const char* 型にすればいいのです。また、Xtal 独特の機能として別の方法もあり、これを使用すれば文字列を返すことも可能です。これを実現するには xtal::StringPtr 型を使用します。この型自体はポインタのようなものですので、引数にも返り値にも xtal::StringPtr 型を使用します。

この xtal::StringPtr (実体は xtal::String)の詳しい使用法はマニュアルを見てください。マニュアルは、この Xtal の記事の最後の参考文献のほうに書きます。

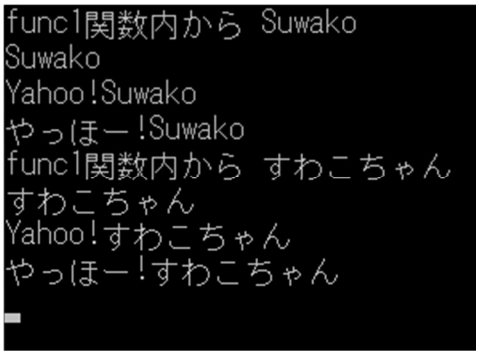
main.cpp (バインドする関数のみ)

```
xtal::StringPtr func1(const char* val) //文字列を引数として渡す
{
	xtal::StringPtr str(val);  //文字列を作成する

	//.c_str()を使えば C++でも読める
	std::cout << "func1 関数内から " << str->c_str() << std::endl;

	return str;  //文字列を返す
}
xtal::StringPtr func2(const char* val)
{
	xtal::StringPtr str("Yahoo!");  //文字列を作成する
	str=str->op_cat(val); //連結する
	return str;
}
xtal::StringPtr func3(xtal::StringPtr val) //この方法でも文字列を渡すことができる
{
	xtal::StringPtr str("やっほー!");  //文字列を作成する
	str=str->op_cat(val); //連結する
	return str;
}
```

script.xtal

```
lib::func1("Suwako").p;
lib::func2("Suwako").p;
lib::func3("Suwako").p;

lib::func1("すわこちゃん").p;
lib::func2("すわこちゃん").p;
lib::func3("すわこちゃん").p;
```

実行結果



文字列も自由自在ですね。すばらしい。

これで、C++の関数を Xtal 側から呼ぶことができるようになりました。これを使えば、Xtal 側から、C++で用意された描画関数の呼び出しや BGM・効果音再生、コントローラ入力情報の取得などができます。

次のお話は、逆に Xtal 側の関数を C++から呼び出す方法についての説明です。これさえできればもう必要な機能はほぼ揃っているといっても過言ではないと思います。それでは休憩しましょう。うどん食べます。

そろそろゲーム制作やってきている人だと、使い方のビジョンが思い浮かべていると思います。そう、使い方はあなた次第なのです。Xtal はコンパイルの時間が短いので、C++で何度も繰り返しビルドするよりはるかに効率がいいのです。そのことに気づいた瞬間、あなたはもう Xtal を使いこなせます。頑張っていきましょう。

Xtal の関数を C++側で呼び出すには

実は、今までのに比べればちょっと厄介ですが、そこまで難しいものでもありません。手順としては、コードをコンパイルし、それを実行してから呼び出すことになります。呼び出し方は次のコードを見てください。

main.cpp (main 関数の部分)

```
int main()
{
        //Xtal の初期化部分・バインド部分は省略

        // 文字列をコンパイル
        xtal::CodePtr pCode =xtal:: xtal::compile_file ("script.xtal");
        XTAL_CATCH_EXCEPT(e){
        std::cout << e->to_s()->c_str() << std::endl;
        }
        //実行
          if(pCode)
```

```
        {
            pCode->call();
            XTAL_CATCH_EXCEPT(e){
            std::cout << e->to_s()->c_str() << std::endl;
            }
        }

        //Xtal の関数の呼び出し
        //Xtal の PrintString を呼び出す
        const xtal::AnyPtr xtalFunc1(pCode->member(Xid(PrintString)));
        xtalFunc1 -> call("すわこちゃん");

        //Xtal の lib::PrintString2 を呼び出す
        const xtal::AnyPtr xtalFunc2(xtal::lib()->member(Xid(PrintString2)));
        xtalFunc2 -> call("きよーか");

        //Xtal の global::PrintScreen3 を呼び出す
        const xtal::AnyPtr xtalFunc3(xtal::global()->member(Xid(PrintString3)));
        xtalFunc3 -> call("いえーい");

        //終了処理以降も省略
        //xtalFunc1 , xtalFunc2 , xtalFunc3 それぞれに xtal::null を代入しておくこと(解放処理)
}
```

script.xtal

```
fun PrintString(str)
{
    str.p;
}

fun lib::PrintString2(str)
{
    str.p;
}

fun global::PrintString3(str)
{
    str.p;
}
```

実行結果はわかると思うので省略。

このように C++から Xtal の関数を呼び出すことができます。lib::はさきほど関数のバインドをしたのと同じものです。global::もいわゆるグローバル変数みたいなものと解説をしました。lib，global のそれぞれの呼び出し方が微妙に違っているので注意してください。

Xtal の関数を呼び出すときに、lib::や global::がついてないものを filelocal といい、そのスクリプトからでしかアクセス出来ないことをいいます（C++からは例の通り呼び出せます）。lib::や global::が付いているものはどこからでもアクセスが可能です。詳しくはのちほどやります。

Xtal の関数を呼び出す際、引数は１３個までらしいので、それに収まるようにしてください。注意はそれくらいです。

それとここでは省略しましたが、実は関数実行ごとにエラー処理を書いたほうがいいです。ちょっと面倒かもしれませんが、デバッグが簡単になるのでできる限り書いたほうがいいです。エラー処理を書くのが面倒ならば、そのエラー処理自体を関数化するのもアリだと思います。

クラスの使い方あれこれ

ここからはちょっと実践的な Xtal のクラスの使用法について書きます。基本的に動的型付け言語であるので継承が不要にもかかわらず継承っぽいことができます。

インスタンスの生成

クラスの実体を生成します。といっても C++側はスクリプトのコンパイル・実行を行うだけで、スクリプト側だけで完結する場合はこう書きます。

main.cpp (main 関数の部分)

```
int main()
{
        //Xtal の初期化部分・バインド部分・コンパイル・実行を省略
        //終了処理以降も省略
}
```

script.xtal

```
// クラスを定義します
class lib::TestClass{
    _var : 0;    // メンバ変数

    initialize(x){    // コンストラクタ。Class::call ではこれが呼ばれる
        _var = x;
    }

    // method
    testMethod(x){
        x.p();
        println(%f[testMethod call: _var = %s](_var));
    }
}

instance : lib::TestClass(100);    //クラスのインスタンスの生成
instance.testMethod("testMethod called");    //クラスのメソッドを実行
```

実行結果

```
testMethod called
testMethod call: _var = 100
```

これで使い方がわかると思います。わからない場合、「クラスの定義」のページを探して読み返してください。

次に、この Xtal のクラスを C++側でインスタンスを生成して呼び出してみましょう。全く同じ結果になります。

main.cpp (main 関数の部分)

```
int main()
{
        //Xtal の初期化部分・バインド部分・コンパイル・実行を省略
        //Xtal の TestClass を取得する
        const xtal::AnyPtr TestClass(xtal::lib()->member(Xid(TestClass)));

        // lib::TestClass(100)と同じ　インスタンスの生成
        xtal::AnyPtr instance(TestClass->call(100));
```

```
        //メソッドの呼び出しには、->call ではなく、->send を使う。
        instance->send(Xid(testMethod), "testMethod called");

        //終了処理以降も省略　instance に xtal::null を代入しておくこと
}
```

script.xtal

```
// クラスを定義します
class lib::TestClass{
    _var : 0;    // メンバ変数

    initialize(x){    // コンストラクタ。Class::call ではこれが呼ばれる
        _var = x;
    }

    // method
    testMethod(x){
        x.p();
        println(%f[testMethod call: _var = %s](_var));
    }

    // class Func
    fun classFunc(){
        println("classFunc is called");
    }
}
```

補足ですが、Xtal 側の println 関数は、コンソール画面に文字列を出力する関数です。C++側でも xtal::stdout_stream()->println で呼び出せます。

実行結果はさっきと同じになるはずです。ちょっと C++でのインスタンスの生成がわかりづらいと思いますが、私は現在のゲームエンジンで C++側からのインスタンス生成を使った記憶がないので特に気にする必要はないと思います。ただクラスのメンバにアクセスするのはよく使います。

メンバ変数へのアクセス

public な(先頭に+をつけた)メンバ変数_foo を定義すると、foo()という getter と set_foo(x)という setter（どちらもメンバ関数）が定義されます。これらを send で呼び出すことでクラスのメンバ変数にアクセスすることが可能となります。

main.cpp(実行と終了処理の間のコード)

```
    AnyPtr bar(lib()->member(Xid(bar)));

    xtal::stdout_stream()->println(bar->send(Xid(foo)));
    //xtal::stdout_stream()->println(bar->member(Xid(_foo)));  //できません

    bar->send(Xid(set_foo), "changed foo");
    xtal::stdout_stream()->println(bar->Xid(foo));
```

script.xtal

```
class Bar{
	+ _foo : "foo";
}

lib::bar : Bar();
```

「_foo はインスタンス lib::bar のメンバだから……」って member でやろうとしても無理なので、面倒でも send で set/get する必要があります。

クラスを利用してシーン遷移を実装してみよう

ようやくゲーム制作らしいことをします。シーン遷移プログラムを書いてみましょう。一応この記事では具体的なコードは書きますが、具体的に実行するには実際にゲーム作らなければならないため、あくまでも紹介を目的として書きます。

まずはシーン遷移の仕様を決めます。

- ●Xtal のクラスを利用する
- ●シーンを std::map で、番号で管理したい
- ●Update()メソッドに毎フレーム呼び出す処理を書く
- ●Draw()メソッドに描画処理を書く

この方針でやってみようと思います。

まず、シーン遷移で必要そうな機能といえば、最低限

- ○シーンの登録
- ○シーンの切り替え
- ○毎フレーム呼び出す関数

このくらいは必要そうです。そこで、この２つの機能を実装しようと考えて関数を書きます。

C++のコード

```
#include <map>

std::map<int,xtal::AnyPtr> sc;
int nowScene=0;

//ゲームシーンの登録
int RegistGameScene(int num , xtal::AnyPtr scene)
{
	sc.insert(std::make_pair(num, scene));
	return 1;
}

//ゲームシーンを変える
int ChangeGameScene(int num)
{

	nowScene = num;
	return 1;
}
```

```
//毎フレーム実行する関数
int ExecGameScene()
{
	if(sc.count(nowScene)==0)return 0;
	sc[nowScene]->send(Xid(Update));

	//エラー処理をここに書いておく

	return 1;
}

//描画する関数
int DrawGameScene()
{
	if(sc.count(nowScene)==0)return 0;
	sc[nowScene]->send(Xid(Draw));

	//エラー処理をここに書いておく

	return 1;
}
```

ざっとこんな感じで、あとは C++側で ExecGameScene()と DrawGameScene()を毎フレーム呼び出せば簡単シーン遷移のできあがりです。

この中で、Xtal 側から呼び出す必要がありそうな関数を２つ RegistGameScene と ChangeGameScene をバインドしておきましょう。

次にシーンの登録です。このクラスのインスタンスを C++側に直接渡すような感じでシーンの登録をしていきます。

それでは、こんな感じでクラスを定義してみましょう。

xtal のスクリプト

```
class GameScene{
	Update()
	  {
	    //ゲームのメイン処理を書く
	  }

	Draw()
	  {
	    //ゲームの描画処理を書く
	  }
}

lib::RegistGameScene( 0 , GameScene() ); //インスタンスを生成して渡し、0 番に登録する。
lib::ChangeGameScene(0);                 //ゲームシーンを 0 番に切り替える
```

このような感じになります。もちろん、シーン番号を変えれば、Update と Draw メソッドがあるクラスならメモリの許す限り登録できます。

シーン遷移を簡潔に書くことができるようになりました。これで登録もシーン遷移も楽になりました。便利になりましたねー　はい。

これらのコードはあくまで一例で最低限の機能しか入れてないので、自分で実際にゲームに使用する際にはもっと便利な機能を入れてみてください。

今回はクラスの利用法をゲームのシーン遷移だけの説明で終わってしまいましたが、同じ事を応用すれば複数のオブジェクトの管理もかなり楽になります（ただ弾幕とか物量勝負の時は、オブジェクトの管理を C++にするなど Xtal の利用は控えたほうがいいかもしれないですが)。

さて、これにて Xtal でゲーム制作に必要そうな機能はだいたい揃ったと思います。実は C++のクラスを Xtal から呼び出すのもあるのですが、これはちょっとややこしいし執筆者自身も使ったことがないため、参考文献から巡ってみて自分で調べてみてください。大丈夫ですきみならできます。

それでは、次はセーブデータの実装について紹介します。そのまえに休憩しましょう。オレンジジュースがおすすめです。

セーブデータの実装

ここでは、Xtal 側でセーブファイルのデータ（中身）を作成し、それを C++側に渡して、それをセーブすることにします。

今回使用するのは、予め用意されている MemoryStream クラスです。

C++のコード（バインドする関数部分）

```
#include <fstream>

int Save(const char* filename , xtal::StreamPtr& stream)
{
	std::ofstream ofs(filename,std::ios::out | std::ios::binary);
	stream->seek(0);
	while(!stream->eos())ofs.put(stream->get_u8());
	return 1;
}

int Load(const char* filename, xtal::StreamPtr& stream)
{
	std::ifstream ifs(filename);
	while(!ifs.eof()){stream->put_u8(ifs.get());}
	stream->seek(0);
	return 1;
}
```

xtal のスクリプト

```
ms:MemoryStream();	//メモリストリームを定義
ms.put_i32le(123);	//数値 123 を、符号付き 32bit 整数リトルエンディアンで書き込む
ms.put_i32le(-456);	//数値-456 を、符号付き 32bit 整数リトルエンディアンで書き込む
ms.put_i32le(65536);	//数値 65536 を、符号付き 32bit 整数リトルエンディアンで書き込む
lib::Save("savedata.sav",ms);	//メモリストリームの中身をセーブする

ml:MemoryStream();	//メモリストリームを定義
lib::Load("savedata.sav",ml);	//セーブしたものを読み込んでメモリストリームの中に入れる
ml.get_i32le().p;	//読み出し
ml.get_i32le().p;	//読み出し
ml.get_i32le().p;	//読み出し
```

実行結果

セーブして同じのを読み込んでいるだけなのでそりゃそうなのですが、実行結果はこうなります。しかし、この C++のコードはあまりスタイリッシュなものではなく、いろいろ妥協した書き方なので実際はもっとスタイリッシュでスマートな方法はあります。（正直言うとうまく行かなかったから妥協しました。はい。）

この MemoryStream の get_i32le や put_i32le の意味ですが、
○get は取得、put は書込
○ i は符号あり整数、u は符号なし整数、f は浮動小数点数
○その次の数字はビット数(8,16,32,64 に対応)
○その後ろの le と be はリトルエンディアンかビッグエンディアン
　ただしビット数が 8 の場合は不要
ということになっています。例えば、16 ビットの符号なし整数をビッグエンディアンで取得する場合は「get_u16be()」となり、32 ビットの浮動小数点数の 23.4 をリトルエンディアンで書き込む場合「put_f32le(23.4)」となり、8 ビットの符号あり整数を取得する場合は「get_i8()」となります。ただし、浮動小数点数は 32 か 64 ビットしか対応していません。
また、文字列を書きだす場合は put_s (文字列)、読み込む場合は get_s (文字の長さ)を使用します。しかし文字列のサイズは可変であるため、予め文字列のサイズを MemoryStream に書きこんでから、それから文字列を書きこむようにすれば、文字列の読み込みのときにサイズを読み込んでそれから文字列を指定サイズ分読み込むということが可能になります。
また、MemoryStream は seek で位置の移動、tell で現在位置の取得、size で全体のサイズが取得できるようになっています。
ここまでで Xtal でのセーブデータの実装方法を紹介しました。ちとめんどくさいかなぁと思いますが、こういう作業も楽しめるようになってくれば立派なゲーム製作者になれる　とおもったら大間違いでした。ドンマイ。

それでは、だいたいやりたいことは説明し終わったので、次は C++やっている人にはピンとこない fiber(ファイバー)についてです。マイクロスレッドとも呼ばれる機能で、手順の記述を直感的にできるので RPG 制作でのイベントの記述やアクションゲームにおけるボスの動きの手順を簡潔に記述できるようになる便利な機能です。それでは休憩しましょう。文字ばっかり打っているからかなり疲れています。フランクフルト食べましょう。

fiber(ファイバー)について

fiber というのは、途中で中断・再開できる関数のようなものです。ピンとこないのは C++脳ですから仕方が無いのです。ちょっとずつやっていきましょう。
fiber は、yield で一旦処理を中断し、また呼び出されると yield から再開します。

Xtal のスクリプト

```
fiber fib()
{
	"初回の実行".p;
	yield;
	"二回目の実行".p;
	yield;
	"三回目の実行".p;
	yield;
	"四回目の実行".p;
}

fib();
```

```
fib();
fib();
fib();
```

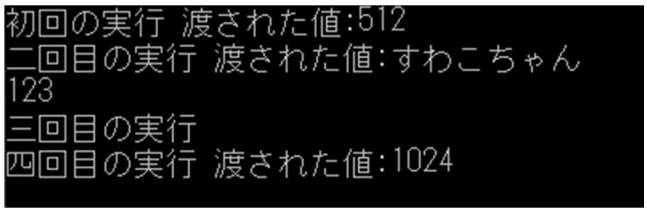
実行結果

また引数を渡したり、値を返したりもできます。
ちょっとややこしいかもしれません。

Xtal のスクリプト

```
fiber fib(x)  //初回の引数受け取り
{
        ("初回の実行 渡された値:"~x.to_s).p;
        str:yield;  //引数を受け取れる
        ("二回目の実行 渡された値:"~str).p;
        yield 123;  //値も返せる
        "三回目の実行".p;
        n:yield;
        ("四回目の実行 渡された値:"~n.to_s).p;
}

fib(512);
fib("すわこちゃん").p;
fib();
fib(1024);
```

実行結果

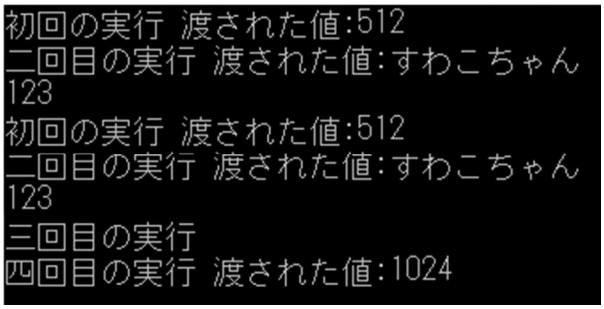


Xtal のスクリプト

```
fiber fib(x)  //初回の引数受け取り
{
        ("初回の実行 渡された値:"~x.to_s).p;
        str:yield;  //引数を受け取れる
        ("二回目の実行 渡された値:"~str).p;
        yield 123;  //値も返せる
        "三回目の実行".p;
        n:yield;
        ("四回目の実行 渡された値:"~n.to_s).p;
}

fib(512);
```

```
fib("すわこちゃん").p;
fib.reset();
fib(512);
fib("すわこちゃん").p;
fib();
fib(1024);
```

実行結果



また、複数の返り値を返したり、複数の引数を指定したりすることもできます。
これは関数(fun)でも使用可能です。

Xtal のスクリプト

```
fiber fib(x,y)
{
        x.p;
        y.p;
        xx,yy : yield x+y , x-y;
        xx.p;
        yy.p;
}

a,b:fib(512,256);
a.p;
b.p;
fib(a*b,a/b);
```

実行結果

fiber の説明はこれにて終了です。使い方がピンとこないかもしれませんが、実際使い方を思いつけば驚くほど便利なものだということに気づくでしょう。実際、筆者は RPG 制作の時におおいに活用した機能でもあります。おどろくほど便利でした。

それでは休憩しましょう。納豆ごはんたべます。

Xtal の機能と小技集と注意すること

逆引き形式で載せていきます。
ここで掲載するコードは特に断りがなければ Xtal のスクリプトのコードです。

文字の番号（アスキーコード）を取得する

```
str : "A";
str.ascii.p;

"F".ascii.p;
```

変数を未定義のまま宣言する

```
val : undefined;  //undefined の代わりに null も OK

if(val){
      //ここは通らない
}else{
      //ここは通る
    val = 20; //なんでも代入可能
}
```

global::と lib::の違い

```
lib::Value1 : 123;
global::Value2 : 456;

//Value1.p;    //エラー
lib::Value1.p;  //OK

Value2.p;          //OK
global::Value2.p;  //OK

//lib::Value1 : 113;     //上書き不可能
global::Value2 : 789;  //上書き可能
Value2.p;
```

lib::で宣言すると、必ず lib::を付けなければならず、上書き不可能になります。
global::で宣言すると、global::を省略可能で、上書きも可能になります。
ただし、両方とも定数扱いであるので代入は不可能です。

複数の Xtal スクリプトをコンパイルしたい

pCode を配列で持って、それぞれ順番にコンパイルと実行を行えば問題ないと思います。私は、unordered_map を利用して管理しています。

複数のスプリクト間では、lib や global で宣言された変数やクラスなどのオブジェクトは共通して使用することができますが、ファイル内のローカル変数にはアクセスが不可能になっています。

もちろん、解放処理を行う場合、すべての pCode に xtal::null を代入しておくことをお忘れなく。

実際、同人ゲームレベルだとスクリプトファイルが最低でも３０～４０くらいは必要になってきますので、複数のスクリプトをコンパイルできるような仕組みは作っておくべきだと思います。

Xtal のバイトコード（コンパイル済みのデータ）を吐き出し・読み込みをする

ここは Sukai さんのコードを見ちゃったほうが早いので引用します。

ファイルにバイトコードを吐き出す(C++)

```
void DumpByteCode(const xtal::CodePtr& code, const xtal::StringPtr& filename)
{
    xtal::SmartPtr<xtal::FileStream> fs = xtal::xnew xtal::FileStream>(filename, "w");
    fs->serialize(code);
}
```

ファイルからバイトコードを読み込む（C++）

```
xtal::CodePtr LoadBytecodeFromFile(const xtal::StringPtr& filename)
{
     xtal::SmartPtr xtal::FileStream> fs = xtal::xnew<xtal::FileStream>(filename, "r");
     return xtal::ptr_cast<xtal::Code>(fs->deserialize());
}
```

メモリからバイトコードを読み込む（C++）

```
xtal::CodePtr LoadBytecodeFromMemory(const void* ptr, xtal::uint_t size)
{
    xtal::SmartPtr<xtal::PointerStream> ps = xtal::xnew<xtal::PointerStream>(ptr, size);
    return xtal::ptr_cast<xtal::Code>(ps->deserialize());
}
```

別のスレッドでの扱いについて

Xtal ではスレッドごとに VM（バーチャルマシン）を持っています。同じスレッドなら lib::や global::はどこからでもアクセス可能ではありますが、別スレッドだとアクセスができなくなります。注意しましょう

メモリ管理について

Xtal では、参照カウンタと GC（ガベージコレクション）を用いてメモリ管理を行っていますので、メモリについてはあまり気にしなくてもいいとされています。

一応ですが、ゲーム制作する際は毎フレームごとに xtal::gc()、シーン遷移時に xtal::full_gc()を呼び出したほうがいいと思います。

しかし、コード例で出てきたような pCode など、C++側で作成したオブジェクトについては、終了処理時に残っていると assert が出てしまうので、必ず xtal::null を代入しておく必要があります。

Xtal のライセンスについて

MIT ライセンスです。要約すると「このソフトウェアを誰でも無償で無制限に扱って良い。ただし、著作権表示および本許諾表示をソフトウェアのすべての複製または重要な部分に記載しなければならない。作者または著作権者は、ソフトウェアに関してなんら責任を負わない。」(Wikipedia より)のような感じであるので、商用や同人ゲームでも使用して問題ないということです。そうじゃなかったら同人ゲームで使われているということはありえませんからね。

リロード機能について

ちょっと今回は触れられなかった機能のひとつに、リロード機能があります。この機能を詳しく説明すると結構ページ数を食うので今回は諦めます。参考文献のページからめぐればやり方が載っているので、興味があれば探してみてください。

参考文献

ひと通り Xtal について説明しましたが、まだ足りない部分もいくつかあります。
そこでもっと Xtal を知りたい人へのリンクを掲載します。

●Xtal の Google Code のページ
http://code.google.com/p/xtal-language/
Xtal のリファレンスをダウンロードすることができます。

●新言語 Xtal を作る日記
http://d.hatena.ne.jp/xtalco/
Xtal の開発日記ですが、更新止まっていますね。

●Xtal Unofficial Wiki
http://sukai.sakura.ne.jp/xtal/
Sukai さんの「石ころ Games」内の 1 コンテンツである Xtal のサイトです。
現状で、Xtal について一番詳しくかつわかりやすく書かれているサイトをここ以外に知りません。今回の記事作成には、このページを大いに参考に(引用)しました。

あとがき

今回は、44 ページにもわたって Xtal の記事を書きました。長いマジで長い。と思ったら会誌にホチキス留まらないとか言われて削らなきゃいけなくなったので、削れるものがそもそも少なかったということで、文字のサイズ小さくしたり行間詰めたりして 30 ページに抑えました。これ以上は無理です勘弁して下さい。

あとはこれで Xtal を使用してくれる人が増えてくれることを祈るしかないですね。Xtal に限らずスクリプト言語を利用すると開発効率が結構上がるので導入してみましょう。その時はぜひ Xtal を活用してみてください。
記事を書くにあたって、Sukai さんが記事の引用を快く承諾してくださいましたので本当に感謝しています。ありがとうございました。

これ書いている時点ではコミケの当落が出るまであと二週間くらいですね。受かっているといいなぁ……そもそも落ちたこと無いけど。そのときはダイエットサイドビューアクションゲームが出ると思います。

それでは、最後になりましたが、どうせ最後なのですわこちゃん載せます。それではまたどこかで会える時が来たり私のゲームをプレイするときが来たりしたら、そのときはよろしくお願い致します。受かっていたらコミケで会いましょう。