

[ホーム](#) < [ゲームつくるー！](#) < [デバッグ技術編](#)

その6 CPPUnitを使ってテスト駆動型開発

前章からほぼ3年、気の長い話です(^-^;)。

C++に限らず何かプログラムを作る時、特に意識しなければ、

1. 作りたい物（設計）を考える
2. クラス（機能）を作る → 一塊のモジュール
3. 動かしてみる（プロジェクトに加える）
4. うまく動かない時は修正

というプロセスを経てプログラムを大きくしていくと思います。しかし、この方法はそのうち変更に対する保守が難しくなってきます。それはクラスのメソッドを追加したり変更した時、その変更がどこでどう影響するかが分かりにくくなってしまうためです。

何か変更追加した後も、すでに実装されている箇所がちゃんと意図した結果を返しているか？それを判断する簡単な理屈は「何かテストする入力を与えてみて、その出力が予定通りかどうか判断する」です。以前は合っていた答えが、変更した事で違ってしまったら、その変更が影響を与えたと判断できるので、その影響箇所を整合性が合うように変更すればOKですよね。ただ、ここで重要なのが「**以前は合っていた**」という所。その以前の状態はコードに残っているのでしょうか？それとも場当たり的にテストしたものを単に記憶しているだけでしょうか？もし後者なら、実装が込み入ってくれば来るほど再テストを要するため「以前合っていた」と確認する作業量が増えてしまう事になります。そこでこう考えます、「**最初からテストコードを書いておこう**」と。すると、開発のプロセスは

1. クラス設計を考える
2. **テストコードを書く**
3. クラスを作る
4. テストコードを実行してパスするかテスト
5. パスしない時はパスするまで修正

という流れに変わります。クラスを実際にコードに落とす前にテストコードを先に書くのがポイントです。クラスを修正・機能追加した時には、

1. **追加部分をチェックするテストコードを書く**
2. クラスの機能を追加する
3. テストコードを実行してパスするかテスト
4. パスしない時はパスするまで修正

となります。やはり先にテストコードから書き、そのテストをパスするようにクラスを実装する。このように、先にチェックすべきテストコードを書き、その後にそのテストを通る機能を実装するという開発方法を「**テスト駆動型開発**」と言います。

これは自前でももちろんできます。ただ、世の中にはもうこのテスト環境を整えてくれるライブラリが沢山公開されています。IDEが提供してくれている事もあります。コードレベルでフレームワークを提供してくれている中で有名なのが「○○Unit」という名前が付いたライブラリです。○○にはプログラム言語の名前が入ります。例えばPHPなら「PHPUnit」、C言語なら「CUnit」など。その中でC++でのテスト駆動型開発をサ

ポートしてくれるのが「**CPPUnit**」です。テスト駆動型開発はそのテスト環境を作る事が割と面倒なのですが、CPPUnitはそれをとても簡単にしてくれます。

すっごく前置きが長くなってしまったのですが(^-^);、この章ではそんなCPPUnitの導入から基本的な使い方までを見て行く事にします。

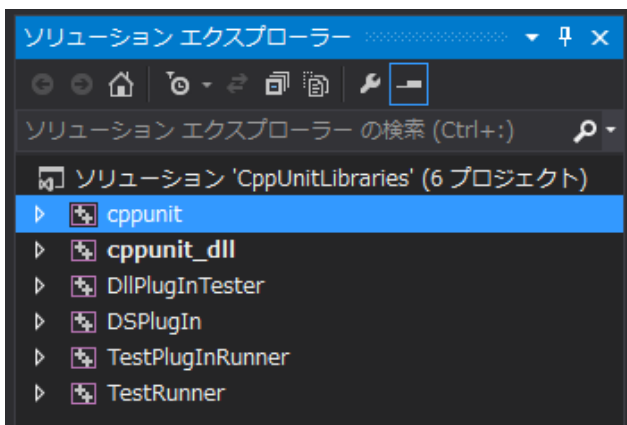
① CPPUnitをダウンロード

CPPUnitはSource Forgeで公開されています ([CppUnit - C++ port of JUnit](#))。2014.7時点で最新バージョンは1.12.1ですが2008年公開になっています。もちろん十分に使えます。最新版のリンク先に飛ぶと「cppunit-1.12.1.tar.gz」がありますのでDLして解凍して下さい。

② CPPUnitのライブラリcppunit.libを作成

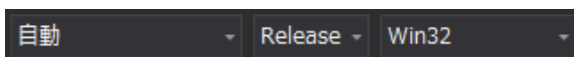
解凍するとcppunit-1.12.1というフォルダの下に沢山のC++コードが作られます。CPPUnitを使うには、このコードを直接プロジェクトに加えるのではなくて、cppunit.libを作ります。**次の目的はcppunit.libをビルドする事**です。

そのためには[解凍フォルダ]/src/CppUnitLibraries.dswをVisual Studioで開きます。.dswは結構昔のVSのプロジェクトファイルなので、多くの場合プロジェクトの変換作業が始まると思います。変換すると6つのプロジェクトが入ったソリューションができるはずです：

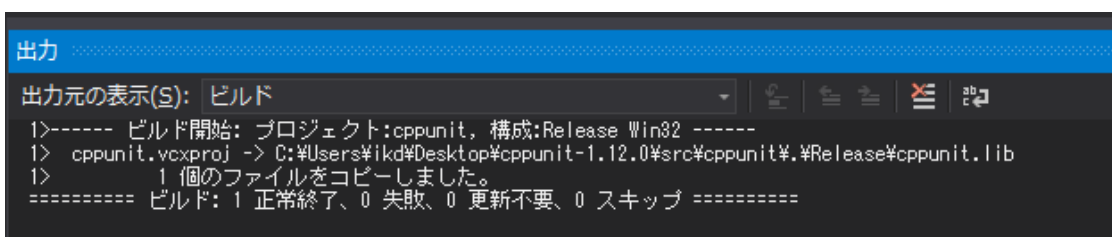


VS2013です

cppunit.libを作ってくれるプロジェクトは一番上の「cppunit」です。ただ、ソリューションの構成がDebugの場合はcppunitd.libが出来てしまうので、ここをReleaseに変えましょう：



この状態でメニューから[ビルド]→[cppunitのビルド]を選択するとこのプロジェクトだけがビルドされます。ビルド結果はこんな感じ：



cppunit.libが出来たのがわかりますね。そして、どこかにコピーされたとあります。それがどこかというところ[解凍フォルダ]/libフォルダ下です（プロジェクトのプロパティのビルド後イベントに書いてあります）。これでcppunit.libが出来ました(^-^)

③ テスト環境を作る

cppunit.libはヘッダーファイルと共にテスト用のプロジェクトに組み込んで使用します。早速テスト用のプロジェクトを作ってみましょう。

cppunit.libは基本的に「コンソールアプリケーション」として使うのが簡単です。新規でコンソールアプリケーションを作成したら、プロジェクトのプロパティを開きます。先程の**cppunit.libはRelease用なので、構成をRelease**にして下さい。Debugにするとビルドでいっぱいエラーが出ます。リンカー/全般内の「追加のライブラリディレクトリ」に[解凍フォルダ]/libを、リンカー/入力内の「追加の依存ファイル」にcppunit.libを追加します。ヘッダーファイルは[解凍フォルダ]/includeにありますので、C/C++/全般の[追加のインクルードディレクトリ]にそのパスを通します。これでcppunit.libが使えるようになります。

④ テストコードを書いてみよう

では、テストコードを書いてみましょう。まず何か作りたい物が必要です。今回はとってもしょぼい電卓クラス（Calculator）を作ってみましょう。

CPPUnitによるテスト駆動型開発では、作りたいクラスを決めた後、すぐにテスト用のクラスを作ります。慣習的にテスト用のクラスはテスト対象のクラス名+Testと付けるようです（CalcuratorTest）。で、テストクラスはCPPUnitが用意してくれている「CPPUnit::TestFixture」クラスを継承します：

```
#include "cppunit/extensions/HelperMacros.h"

class CalcuratorTest : public CppUnit::TestFixture {
public:
    ...
};
```

TestFixtureクラスはテストが開始された時及び終了時に呼ばれる2つのメソッド（setUp, tearDown）を定義してくれています。virtualなので上のクラスに定義すれば何らかの初期化と終了処理ができるわけです。それは追々(^-^;。インクルードしているHelperMacrosヘッダーには、テストに使うマクロが登録されています。

下地となるクラスを作ったら、テストとなるコードを書きます。Calcuratorクラスはしょぼいので、電卓のボタンに対応したメソッドを持っているとしましょう。電卓には色々ボタンがありますが、まずは数字ボタンであるpushNumberメソッドが「**クラスにあると想定したテストコード**」を書きます。ここポイントです。Calcuratorクラスにはまだ無いメソッドをテストにいきなり書くんです。

もう一つ大切な事。「何をテストするか？」を明確にします。そのためにはpushNumberメソッドの機能を明確にする必要があります。ありますが、まずは小さい事を決めるだけに留めます。これが大切なコツです。とりあえず「**pushNumberメソッドは引数に0～9までの数字を取る**」とだけ決めましょう。で、これらの数字が来たらtrue、それ以外はfalseを返すようにします。CalcuratorTestクラスには、これをテストするコードを次のように記載します：

```
#include "cppunit/ui/text/TestRunner.h"
#include "cppunit/extensions/HelperMacros.h"
#include "cppunit/TextOutputter.h"
```

```

class CalculatorTest : public CppUnit::TestFixture {

    CPPUNIT_TEST_SUITE( CalculatorTest );
    CPPUNIT_TEST( pushNumberTest_push0 );
    CPPUNIT_TEST_SUITE_END();

public:
    void pushNumberTest_push0() {

        Calculator calc;
        CPPUNIT_ASSERT( calc.pushNumber( 0 ) == true );

    };

int _tmain(int argc, _TCHAR* argv[])
{
    CPPUNIT_TEST_SUITE_REGISTRATION( CalculatorTest );

    CppUnit::TextUi::TestRunner runner;
    runner.addTest( CppUnit::TestFactoryRegistry::getRegistry().makeTest() );

    CppUnit::Outputter* outputter = new CppUnit::TextOutputter( &runner.result(), std::cout );
    runner.setOutputter( outputter );

    return runner.run() ? 0 : 1;
}

```

少しコードが増えました。まずCalculatorTestクラス内に「CPPUNIT_TEST_SUITE」というマクロを追加します。このマクロの引数にはクラス名をそのまま渡します。このマクロでこのクラスがテスト用のクラスである事をCppUnitに伝えているわけです。次の「CPPUNIT_TEST」にはテストとして呼び出すメソッド名を引数に渡します。CPPUNIT_TESTマクロは呼び出すメソッドが増える度にどんどん追加していきます。そして最後に「CPPUNIT_TEST_SUITE_END」マクロを置く事で呼び出しメソッドが裏で列挙されます。

続いて、実際にテストで呼び出したいメソッドをpublicで定義します。pushNumberTest_push0メソッドの中では、電卓であるCalcurator型のcalcオブジェクトを定義しています。その下が正にテスト部分。

「CPPUNIT_ASSERT」マクロは沢山あるテスト用マクロの一つで、引数の中が偽の場合に「テスト失敗」と判断して診断結果を出力してくれます。

そのテストの書き方がとても大切になります。上の例ではcalc.pushnumber(0)と電卓の「0」を押した場合のテストを想定しています。正しい挙動をしているならメソッドはtrueを返します。しかし、テスト駆動型開発では、まず「**テストに失敗する実装**」を書く事が鉄則になっています。これにより「対象としているテストコードが通っていて機能している」事を確認できます。つまり、折角テストコードを書いても、何かの手違いでそこが通っていないければ全く意味がありません。また最初から真となるテストコードを書くと、本当に真なのか、たまたま通っていないのかを診断結果から判断できなくなります。ですから「テストに失敗するコードをまず書く」というのが大切なんです。

メイン関数の中は、実際にテストを動かすためのコードです。CPPUNIT_TEST_SUITE_REGISTRATIONマクロは、その名の通りテストクラスを登録します。引数にCalculatorTestが登録されているので、このテストクラス内のテストメソッドが実行されます。

続くTestRunnerクラスはテストを実行する本体となるクラスです。このaddTestメソッドに上のコードのように登録されたテストオブジェクトを渡します（ここはこう書けば問題ありません）。Outputterクラスはテスト出力用のクラスで、テスト結果を整えて出力してくれます。上の場合TextOutputterという派生クラスを渡しています。コンストラクタの引数を見ると、第1引数にTestRunnerの結果を、第2引数に出カストリームを渡しています。このoutputterをTestRunner::setOutputterメソッドに渡して関連付けをします。

最後にrunner.run()でテストを開始してくれます。結果はboolで、何か一つでもテストにパスしていない部分があればfalseを、全部合格していればtrueが返されます。上のようにメイン関数の戻り値を変えることで、このテストの実行ファイルの戻り値で合否を判断できるようになります。

メイン関数の所はCPPUNIT_TEST_SUITE_REGISTRATIONマクロに渡すクラス名以外はだいたいいつも一緒になりますので、共通項として分離しておくを使い勝手が良くなりますね。

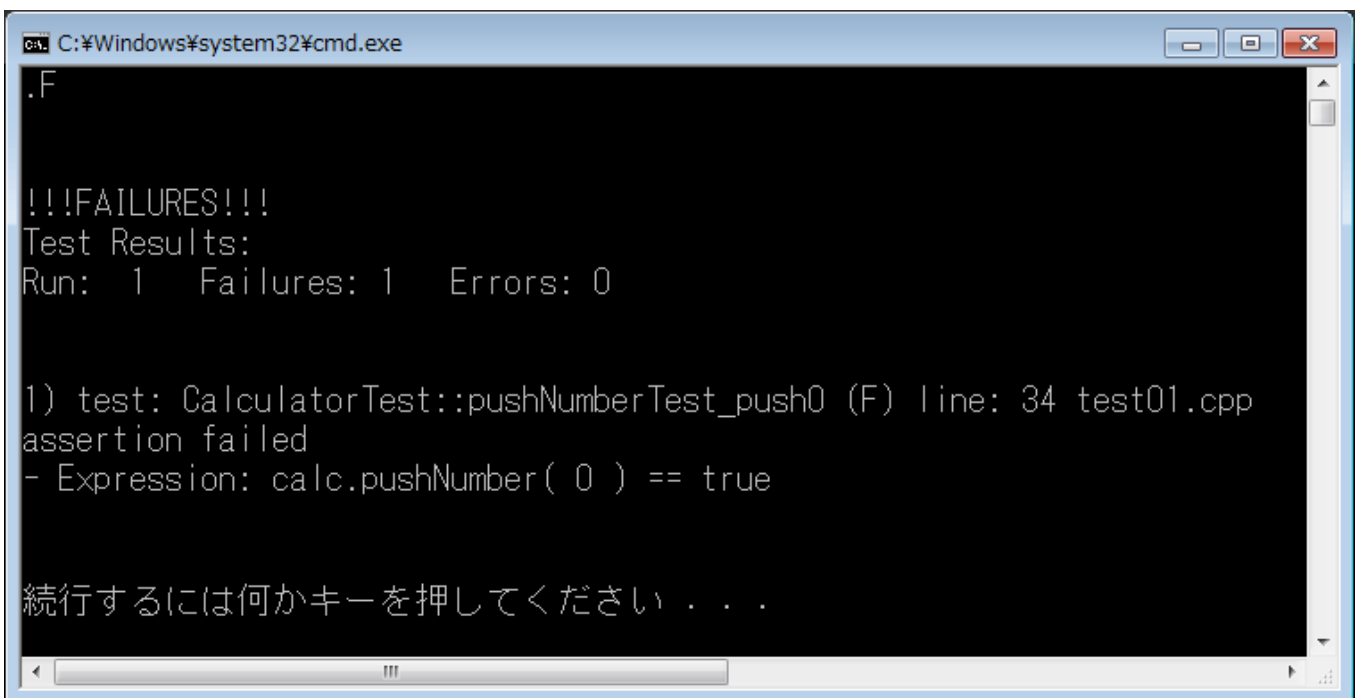
さて、上のコードをコンパイルすると、当然ですがコンパイルエラーが出ます。Calculatorクラスが無いからです。このコンパイルエラーを確認して、初めてCalcLatorクラスを作り始めます。テスト駆動型開発は徹底して「テストから」なんです。

⑤ Calculator::pushNumberメソッドのテスト

テスト対象であるCalculator::pushNumberメソッドを作ります。今は「引数に0～9が入ってきたらtrue、それ以外はfalseを返す」という仕様だけを満たすように作ります。この数字をどう保存するかなどはまだ考えません。ただし、最初はテストに失敗する実装を書きます：

```
class Calculator {
public:
    bool pushNumber( unsigned number ) {
        return false;
    }
};
```

これで再度コンパイルしてみましょう。もしソリューション構成がDebugになっていたらビルドエラーが沢山出てしまうので（cppunit.libがRelease用だから）Releaseに変更してからコンパイルしましょう。ビルドに成功したらテスト実行です。Ctrl + F5でデバッグ無し実行しましょう。すると：

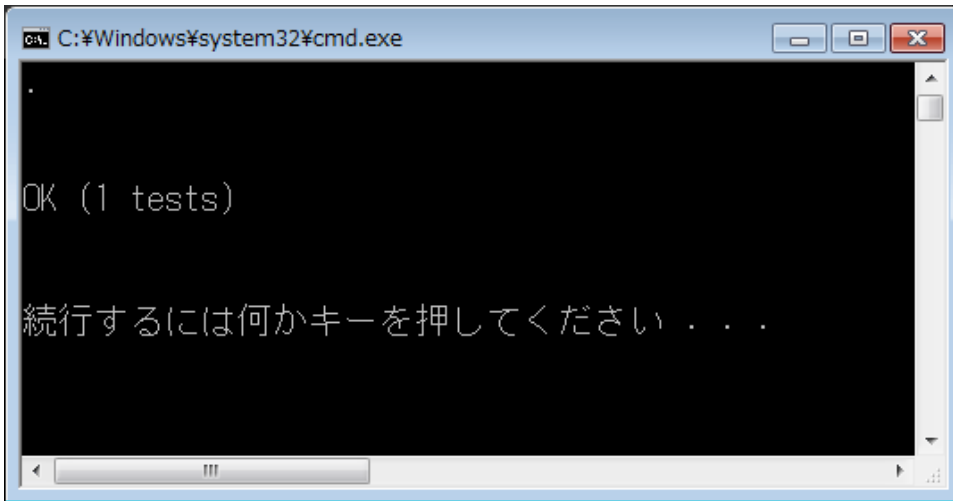


こんな出力結果が得られます。「!!!FAILURES!!!」と強調されているように、意図した通りテストに失敗しています。実行したテストは1つで、失敗が1つある事を教えてくれています。その下にはどのテストが失敗したのかログが出力されます。見ての通り、先程作ったpushNumberTest_push0テストメソッドの34行目にあるテストに失敗した事がわかります。

このように、テストに失敗した場合に詳細なログをぶわーっと出力してくれるのがテスト駆動型開発の魅力です。これにより、「クラスの実装に何か問題があるんだな」とはっきり認識できます。ただ、今回の場合はここでわざとエラーが出るように仕込みました。そこで次にテスト対象のメソッド内を次のように正します：

```
class Calculator {
public:
    bool pushNumber( unsigned number ) {
        return number < 10;
    }
};
```

これで正しいテストコードとなりました。数字の0を入れるテストを再度動かしてみましょう：



すべてのテストに合格した場合はこんな感じでシンプルな結果が返ります。これで一つのテストコードが書け、そのテストも完了しました。このように、

- ① テストコードを書く
- ② テストコードの対象となるメソッドを実装する。ただし失敗するコード。
- ③ 動かして失敗する事を確認する
- ④ 実装部を正しいコードにする
- ⑤ 再度動かして成功する事を確認する

これがテスト駆動型開発での一つのテストの過程になります。これは必ず守ってください。

⑥ 2つ目以降のテスト

1つ目のテストが完全に終わったら、2つ目のテストを作れます。先のテストではpushNumberメソッドに0を入れた場合のテストでした。このメソッドには後1～9までの正しい数値と、それ以外の不正な数値を入れられます。続けてそういうテストを書いてみましょう：

```
class CalculatorTest : public CppUnit::TestFixture {
    CPPUNIT_TEST_SUITE( CalculatorTest );
    CPPUNIT_TEST( pushNumberTest_push_minus1 );
    CPPUNIT_TEST( pushNumberTest_push0 );
    CPPUNIT_TEST( pushNumberTest_push9 );
    CPPUNIT_TEST( pushNumberTest_push10 );
    CPPUNIT_TEST_SUITE_END();

public:
    void pushNumberTest_push_minus1() {
        Calculator calc;
        CPPUNIT_ASSERT( calc.pushNumber( -1 ) == false );
    }
    void pushNumberTest_push0() {
        Calculator calc;
        CPPUNIT_ASSERT( calc.pushNumber( 0 ) == true );
    }
    void pushNumberTest_push9() {
        Calculator calc;
```

```

        CPPUNIT_ASSERT( calc.pushNumber( 9 ) == true );
    }
    void pushNumberTest_push10() {
        Calculator calc;
        CPPUNIT_ASSERT( calc.pushNumber( 10 ) == false );
    }
};

```

テストコードが4つに増えていますが、実際はこのように**複数のテストをいっぺんに書いてはいけません**。テストは原則一つずつ行うのがテスト駆動型開発です。上のはその手順を追って4つ目のテストをしていると思って下さい。

このテストは先のpushNumberメソッドの実装で合格します。良かった良かったです(^-^)。所で、なぜ-1,0,9,10という4つの数値なのか？これはいわゆる「境界テスト」という考え方に従っています。境界テストとは、条件の境目を挟む数値をテストする考え方です。一般にバグはこういう境界で起こる物であるため、そこをテストすると効果的というわけです。電卓の数字ボタンは0～9までの数字を受け付けるのでした。と言う事はtrueとfalseの境界は-1～0と9～10です。unsignedなので-1は実際は0xffffffffですが、今回はこれも境界としました。

上のテストコードでは1～8までの数値を入れて正しく動くかはテストできていません。詳しくテストしたいのであればそういうテストコードを書けばOKです。

⑦ 後は1テスト1実装の繰り返し

テスト駆動型開発を続ければ、Calculatorクラスは少しずつ機能が増えて行くはずで。そして、1つの機能を入れる度にそのテストコードが書けているはずなので、クラスの機能を確認しながら開発出来ます。確認中にその前までは合格していたはずのテストコードが不合格になる事があります。その時はその不合格テストを見て、そこが合格するようにCalculatorクラス内を変更します。すべて合格するまで新しい機能を入れてはいけません。

こうしてCalculatorクラスのテストが終了すると同時に、Calculatorクラスは機能も実装し終わっている事になります。素晴らしいのは、その機能がテストの範囲で確実である事が保障されるという事です。極めてバグの少ないクラスをこうして作る事が出来る訳です。テスト駆動型開発は単体クラスを作る強力な開発手法なのです。

⑧ テスト駆動型開発の注意

CppUnitを使ったテスト駆動型開発は、単体のクラスに対しては抜群の効果を発揮します。ただ、複数のクラスの組み合わせをテストする「結合テスト」のフェーズになるとテストコードを書くのが難しくなってきます。もちろん出来ない事はありませんが、組み合わせが非常に多くなるため網羅出来なくなってくるわけです。一般に、CppUnitは単体テスト（ユニットテスト）の為のフレームワークだと捉えておいた方が良いでしょう。

また、一般にテスト駆動型開発はGUIの部分のテストを苦手とします。これもMVCパターンで対処出来ない事も無いのですが、スマートではありません。

このようにいくつか不得手とする場面はありますが、テスト用のクラスがあって正誤判定ができれば複雑な物でもテストは可能です。出来る限りテストコードを書いて、コードの保守性をうんと高め、それを維持して開発していきましょう～

