1、前置き

2011年1月4日
18:22

LuabindはC++とLuaのバインドを手助けするライブラリで、C++で書かれた関数やクラスをLuaに公開する機能があります。
また、Lua内でクラスを定義する機能も持ち、LuaのクラスまたはC++のクラスを引き出せます。
Luaのクラスは、C++で定義されたクラスベースの仮想関数をオーバーライドすることができます。
これは、lua5.0に向けて書いており、lua4では動作しません。

これはテンプレートメタプログラミングにより実装されます、つまり、プロジェクトのプリプロセッサに余分に時間がかかります。(これはコンパイラによって実行されます)
また、これは通常中身については知る必要性はありませんが、このライブラリはコンパイル時のコードの内容によって生成されるものが変わるので、コンパイルの時間が変わるためcppを同じプロジェクト内に入れることが推奨されます。

luabindはMITライセンス(English Page)です。

私たちは、luabindを使ったプロジェクトに非常に興味があります。是非、フィードバックをお寄せください。

luabindに関するヘルプ・フィードバックはluabind mailing list(English Page)で請け負ってます。

2,特徴

2011年1月4日
19:03

luabindがサポートしているもの：
・フリー関数のオーバーロード
・C++のクラスをLuaで扱うことができる。
・メンバ関数をオーバーロードすることができる。
・演算子（のオーバーロードのことかな?)
・プロパティ
・列挙体
・C++内でのLua関数の利用
・C++内でのLuaクラスの利用
・LuaまたはC++からクラスを引き出すことができる
・C++で定義されたクラスの仮想関数をオーバーライドすることができます。
・登録されている型同士での暗黙の型変換

3,移植可能性

2011年1月4日
19:13

luabindは以下のコンパイラでテストされ、サポートしています。
・Visual Studio 7.1
・Intel C++ 6.0(Windows)
・GCC 2.95(Cygwin)
・GCC 3.04(debian/Linux)
・GCC 3.1(SunOS5.8)
・GCC 3.2(cygwin)
・GCC 3.31(cygwin)
・GCC 3.3(Apple,MacOS X)
・GCC 4.0(Apple,MacOS X)

以下のコンパイラでは動かないことが確認されました。
・GCC 2.95.2 (SunOS 5.8)

MetoroWerks 8.3(Windows)コンパイラでは、テストに失敗しました。
これは、メンバ関数が非constとして扱われるためです。
[
Metrowerks 8.3 (Windows) compiles but fails the const-test. This means that const member functions are treated as non-const member functions.
貼り付け元 <http://www.rasterbar.com/products/luabind/docs.html>
]

もしあなたが、このリストに書いていないコンパイラでテストしたのなら、私たちに結果を教えてください。

4 Luabindのビルド

2011年1月4日
21:29

4-1 必要不可欠のもの

LuabindはBoost 1.34ライブラリに依存します。Boost JamとBoost Build V2

めんどくさい、割愛(´・ω・`)
この辺は日本語の解説サイトがあるようなので、そちらの方を読んでください。

割愛

2011年1月4日
21:36

6,スコープ

2011年1月4日
21:36

Luaに登録されるすべてのものは、グローバルな空間(モジュールと呼びます。)もしくは、名前空間(Luaテーブル)に登録されます。
それぞれの宣言はluabind::moduleで囲まなければなりません。これは以下のように使用されます。

```
module(L)
[
    // declarations
];
```

貼り付け元 <http://www.rasterbar.com/products/luabind/docs.html>

また、名前空間を利用する際はこのようにします。

```
module(L, "my_library")
[
    // declarations
];
```

貼り付け元 <http://www.rasterbar.com/products/luabind/docs.html>

luabindでは、moduleにてlua_State*を渡せば、ほかでは指定する必要がありません。
また、名前空間の中に名前空間を使いたい場合、このように書くこともできます。

```
module(L, "my_library")
[
    // declarations
namespace_("detail")
    [
        // library-private declarations
    ]
];
```

貼り付け元 <http://www.rasterbar.com/products/luabind/docs.html>
また以下の２つは等価なコードとなります。

```
module(L)
[
    namespace_("my_library")
    [
        // declarations
    ]
];
```

貼り付け元 <http://www.rasterbar.com/products/luabind/docs.html>

```
module(L, "my_library")
[
    // declarations
];
```

貼り付け元 <http://www.rasterbar.com/products/luabind/docs.html>
また、それぞれの宣言は、以下のようにカンマで区切ります。

```
module(L)
[
    def("f", &f),
    def("g", &g),
    class_<A>("A")
        .def(constructor<int, int>),
```

```
    def("h", &h)
];
```

貼り付け元 <http://www.rasterbar.com/products/luabind/docs.html>

7,Luaに関数をバインドする

2011年1月4日
21:52

LuaにC++の関数をバインドするには、luabind::def()を使用します。
def()の定義は、以下の通りです。

```
template<class F, class policies>
void def(const char* name, F f, const Policies&);
```

貼り付け元　<http://www.rasterbar.com/products/luabind/docs.html>
＊nameはLua内で使用される関数名です。
＊fはあなたが登録したい関数への関数ポインタです。
＊Policiesパラメーターは、パラメーターと戻り値がどのように機能によって扱われるかを表したパラメーターです。
これはオプションのパラメーターです。詳しくは14章の手段をご覧ください。

これはあなたがfloat std::sin(float)をエクスポートしたいとした場合の例です。

```
module(L)
[
    def("sin", &std::sin)
];
```

貼り付け元　<http://www.rasterbar.com/products/luabind/docs.html>

7-1オーバーロードされた関数

もしあなたが同じ名前で引数が違うオーバーロードされた関数をLuaにエクスポートしたいのならば、次のように書くことができます。
以下例です。int f(const char*),void f(int)が存在するとしてそれをエクスポートするとします。

```
module(L)
[
    def("f", (int(*)(const char*)) &f),
    def("f", (void(*)(int)) &f)
];
```

貼り付け元　<http://www.rasterbar.com/products/luabind/docs.html>

7-2型変換

luabindはスタック内の関数呼び出し時の引数型をチェックします。
不明瞭な宣言の場合以下のように実行されます。

```
struct A
{
    void f();
    void f() const;
};
const A* create_a();
struct B: A {};
struct C: B {};
void g(A*);
void g(B*);
```

貼り付け元　<http://www.rasterbar.com/products/luabind/docs.html>

```
a1 = create_a()
a1:f() -- constのほうが呼ばれます
a2 = A()
a2:f() -- constではない方がよばれます。
a = A()
b = B()
c = C()
g(a) --  g(A*)が呼ばれます
g(b) --  g(B*)が呼ばれます
g(c) --  g(B*)が呼ばれます
```

貼り付け元 <http://www.rasterbar.com/products/luabind/docs.html>

7-3 Luaの関数を呼ぶ

luaの関数を呼ぶ場合は、call_function()かオブジェクトを使用することができます。

```
template<class Ret>
Ret call_function(lua_State* L, const char* name, ...)
template<class Ret>
Ret call_function(object const& obj, ...)
```

貼り付け元 <http://www.rasterbar.com/products/luabind/docs.html>

関数呼び出しに失敗した場合luabind::error()が呼び出されます。
引数への参照をそのまま渡すには、(この場合、Lua側で変数に変更を行うと、C++側にも反映される)Boost::refを使うとよい。

```
int ret = call_function(L, "fun", boost::ref(val));
```

貼り付け元 <http://www.rasterbar.com/products/luabind/docs.html#policies>

もし、エラーの時の処理をカスタマイズしたいならば、set_pcall_callbackを見てください。(16-1)

7-4Luaのスレッドを利用する

luaのスレッドを開始するには、lua_resumeを呼ぶ必要があります。またこれは、前に説明したcall_functionを使うことができないことを意味します。
この二つの関数の定義を以下に示します。

```
template<class Ret>
Ret resume_function(lua_State* L, const char* name, ...)
template<class Ret>
Ret resume_function(object const& obj, ...)
```

貼り付け元 <http://www.rasterbar.com/products/luabind/docs.html#policies>

```
template<class Ret>
Ret resume(lua_State* L, ...)
```

貼り付け元 <http://www.rasterbar.com/products/luabind/docs.html#policies>

あなたは初めてスレッドを利用する際、関数にスレッドを与えなければなりません。
すなわちあなたはresume_functiom()を利用しなくてはなりません。
例えば、このようにしてスレッドを利用します。

```
lua_State* thread = lua_newthread(L);
object fun = get_global(thread)["my_thread_fun"];
resume_function(fun);
```

貼り付け元 <http://www.rasterbar.com/products/luabind/docs.html#policies>

(翻訳者)おそらく以下のコードでも動くかと思われます。

```
lua_State* thread=lua_newthread(L);
resume_function(thread,"my_thread_fun");
```

8,クラスをLuaにバインドさせる

2011年1月4日
23:34

クラスをLua側に登録するには、class_を使用する。また、それぞれの名前はC++のキーワードと似ています。直感的にバインドさせることができます。
def()を使用することで、メンバー、関数、演算子のオーバーロード、コンストラクタ、列挙体そして、クラス内のプロパティをバインドさせることができます。

以下がシンプルな例のC++のクラスです。

```
class testclass
{
public:
    testclass(const std::string& s): m_string(s) {}
    void print_string() { std::cout << m_string << "¥n"; }
private:
    std::string m_string;
};
```

貼り付け元 <http://www.rasterbar.com/products/luabind/docs.html#policies>
このクラスの登録はこのようにします。

```
module(L)
[
    class_<testclass>("testclass")
        .def(constructor<const std::string&>())
        .def("print_string", &testclass::print_string)
];
```

貼り付け元 <http://www.rasterbar.com/products/luabind/docs.html#policies>

このようにした状態で、実験するとこのようになります。

```
Lua 5.0  Copyright (C) 1994-2003 Tecgraf, PUC-Rio
> a = testclass('a string')
> a:print_string()
a string
```

貼り付け元 <http://www.rasterbar.com/products/luabind/docs.html#policies>

これは、フリー関数、もメンバー関数のように登録可能となります。
例えば次の例をご覧ください。

```
struct A
{
    int a;
};
int plus(A* o, int v) { return o->a + v; }
```

貼り付け元 <http://www.rasterbar.com/products/luabind/docs.html#policies>
plusは、登録するクラスのポインターを第一引数に受け取ることができるので、Lua側にはクラスとして登録することができるのです。
このようなコードに対し、以下のように登録できます。

```
class_<A>("A")
    .def("plus", &plus)
```

貼り付け元 <http://www.rasterbar.com/products/luabind/docs.html#policies>

8-1 オーバーロードされたメンバ関数

メンバー関数がオーバーロードされているときは、以下のようにdef()を呼び出します。

メンバー関数ポインタの文法
return-value (*class-name*::*)(*arg1-type*, *arg2-type*, ...)

貼り付け元 <http://www.rasterbar.com/products/luabind/docs.html#policies>

例えば、簡単な例として以下があります。
```
struct A
{
    void f(int);
    void f(int, int);
};
```

貼り付け元 <http://www.rasterbar.com/products/luabind/docs.html#policies>

```
class_<A>()
    .def("f", (void(A::*)(int))&A::f)
```

貼り付け元 <http://www.rasterbar.com/products/luabind/docs.html#policies>

こうして、最初の(void(int))をバインドしています。２つめの(void f(int,int))はバインドしていません。

8-2 プロパティ
簡単にクラスメンバ変数は登録できます。
例えば、以下のC++クラスを登録するとします。
```
struct A
{
    int a;
};
```

貼り付け元 <http://www.rasterbar.com/products/luabind/docs.html#policies>

以下のようにして登録することが可能です。
```
module(L)
[
    class_<A>("A")
        .def_readwrite("a", &A::a)
];
```

貼り付け元 <http://www.rasterbar.com/products/luabind/docs.html#policies>

読み取り専用にするには以下のようにします。
```
module(L)
[
    class_<A>("A")
        .def_readonly("a", &A::a)
];
```

貼り付け元 <http://www.rasterbar.com/products/luabind/docs.html#policies>

また、プロパティとして、安全に簡単に登録することも可能で、set(),get()が簡単に実装できます！！
```
class A
{
public:
    void set_a(int x) { a = x; }
    int get_a() const { return a; }
private:
```

```
    int a;
};
```

貼り付け元 <http://www.rasterbar.com/products/luabind/docs.html#policies>

publicなデータメンバしか登録することはできないのでこのようにします。

```
class_<A>("A")
    .property("a", &A::get_a, &A::set_a)
```

貼り付け元 <http://www.rasterbar.com/products/luabind/docs.html#policies>

また、getのほうにはconstを忘れないように注意してください。エラーが出ます。

8-3列挙体

もしクラスが、列挙された定数ならば(列挙体)、以下のようにして登録することができます。
全ての列挙体は、Luaでは整数型として扱われます。
たとえば、このようにして列挙体を登録します。

```
module(L)
[
    class_<A>("A")
        .enum_("constants")
        [
            value("my_enum", 4),
            value("my_2nd_enum", 7),
            value("another_enum", 6)
        ]
];
```

貼り付け元 <http://www.rasterbar.com/products/luabind/docs.html#policies>

そして以下のように確かめることができます。

```
Lua 5.0  Copyright (C) 1994-2003 Tecgraf, PUC-Rio
> print(A.my_enum)
4
> print(A.another_enum)
6
```

貼り付け元 <http://www.rasterbar.com/products/luabind/docs.html#policies>

8-4演算子

演算子をバインドさせるには、luabind/operator.hppをインクルードしてください。

演算子を登録するには、とてもシンプルです。
luabind::selfを使用して、それ自身のオペレーターを登録します。
例えば以下のクラスのオペレーターを登録します。

```
struct vec
{
    vec operator+(int s);
};
```

貼り付け元 <http://www.rasterbar.com/products/luabind/docs.html#policies>

```
module(L)
[
    class_<vec>("vec")
        .def(self + int())
];
```

貼り付け元 <http://www.rasterbar.com/products/luabind/docs.html#policies>

また、それ自身を指すには、const_selfか前述のselfを使用します。

これは以下の演算子において可能です。
+ - * / == < <=

ただし、==に関しては一部だけとなります。参照に対しては呼ばれないからです。
また、!=,>,>=はサポートされません。

例では、int()でしたが、もし複雑な型(基本形以外ってことかな？)を登録する場合、other<>を使用します。
例えば、以下の例があります。
```
struct vec
{
    vec operator+(std::string);
};
```

貼り付け元 <http://www.rasterbar.com/products/luabind/docs.html#policies>

これを、other<>を用いて以下のように登録します。
```
module(L)
[
    class_<vec>("vec")
        .def(self + other<std::string>())
];
```

貼り付け元 <http://www.rasterbar.com/products/luabind/docs.html#policies>

特別な演算子として、luaには、tostringがあります。しかし、C++にはありませんので、std::ofstreamと、演算子<<を利用することで成り立たせます。
```
class number {};
std::ostream& operator<<(std::ostream&, number&);
...
module(L)
[
    class_<number>("number")
        .def(tostring(self))
];
```

貼り付け元 <http://www.rasterbar.com/products/luabind/docs.html#policies>

8-5階層化されたクラス、静的な関数

階層化(ネスト)されたクラスでも、登録することが可能です。また、これは静的な関数でも同じになります。
```
class_<foo>("foo")
    .def(constructor<>())
    .scope
    [
        class_<inner>("nested"),
        def("f", &f)
    ];
```

貼り付け元 <http://www.rasterbar.com/products/luabind/docs.html#policies>

また、クラス内の名前空間も同じ文法で実現が可能になります。

8-6クラスを継承する

他のクラスから引き継いで、クラスを登録することができます。
例えば以下の2つのクラスを登録する場合
```
struct A {};
struct B : A {};
```

貼り付け元 <http://www.rasterbar.com/products/luabind/docs.html#policies>

```
module(L)
[
    class_<A>("A"),
    class_<B, A>("B")
];
```

貼り付け元 <http://www.rasterbar.com/products/luabind/docs.html#policies>

とすることで、登録できます。しかし、もし、BはCクラスからも継承したい場合。
つまり、継承元が1つ以上である場合
bases<>を使用します。
例えば以下のようにします。

```
module(L)
[
    class_<B, bases<A, C> >("B")
];
```

貼り付け元 <http://www.rasterbar.com/products/luabind/docs.html#policies>

8-7クラスを分けて登録するには

クラスを分けて登録することがこのようにすると可能になります。

```
void register_part1(class_<X>& x)
{
    x.def(/*...*/);
}
void register_part2(class_<X>& x)
{
    x.def(/*...*/);
}
void register_(lua_State* L)
{
    class_<X> x("x");
register_part1(x);
    register_part2(x);
module(L) [ x ];
}
```

貼り付け元 <http://www.rasterbar.com/products/luabind/docs.html#policies>

詳しくは15章の分割登録をご覧ください。