C++ & Bitcoin Core
注意点、まとめ

@DG Lab - Karl-Johan Alm

© 2017 Digital Garage. All rights reserved. Redistribution or public display not permitted without written permission from Digital Garage.

逃げられる前に言っておこう

これはC++を教えるセッションというより、C++にあまり慣れていない人がBitcoin Coreのコードを開くと混乱しそうな機能の注意点を並べていくセッションである！

Agenda

- const
- Ref(&)とpointer(*)
- Ref(&)とby-ref (&)
- constとby-ref (&)の関係
- operators
- friends
- virtual
- テンプレート

const（定数）

Objective-Cの(im)mutability（不変的・変動的）と似ている

例：

```
class A {
  int a;
  const int b;
  void m();
  const int n();
  void o() const;
};
```

種類：	A myA; 変更実行可能？	const A myCnstA; 変更実行可能？
変数	✓	×
const変数	×	×
関数	✓	×
	✓	×
const関数	✓	✓

const(定数)

```
class A {
public:
  int a = 1;
  const int b = 2;
  int m() {return 3;};
  const int n() {return 4;};
  int o() const {return 5;};
};
```

	A			const A		
	R	W	X	R	W	X
`int a = 1;`	✓	✓		✓	×	
`const int b = 2;`	✓	×		✓	×	
`int m() {return 3;};`			✓			×
`const int n() {return 4;};`			✓			×
`int o() const {return 5;};`			✓			✓

const（定数）の実行例

A myA;の場合

```
A myA;
printf("a:%d\n", myA.a);
myA.a = 10;
printf("a:%d\n", myA.a);
// Compile Error
// myA.b = 20;
printf("b:%d\n", myA.b);
printf("m:%d\n", myA.m());
printf("n:%d\n", myA.n());
printf("o:%d\n", myA.o());
```

```
a:1
a:10
b:2
m:3
n:4
o:5
```

const A myConstA;の場合

```
const A myConstA;
printf("a:%d\n", myConstA.a);
// Compile Error
// myConstA.a = 10;
// myA.b = 20;
printf("b:%d\n", myConstA.b);
// Compile Error
// printf("m:%d\n", myConstA.m());
// printf("n:%d\n", myConstA.n());
printf("o:%d\n", myConstA.o());
```

```
a:1
b:2
o:5
```

const(定数)

const <type> myFun()は定数の<type>を返す。

例:

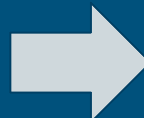
```
class B {
private:  A myA;
Public:   const A getA() { return myA; }
};
int main(void) {                      someA.m();   // できない
  B myB;                              someA.o();   // できる
  const A someA = myB.getA();         someA.a = 38;// できない
```

const（実行例）

A myA;の場合

```
class B {
private:
  A myA;
public:
  const A getA() {
    return myA;
  };
};
```

```
B myB;
const A someA = myB.getA();
printf("a:%d\n", someA.a);
printf("b:%d\n", someA.b);
// Compile Error
// someA.a = 10;
// someA.b = 10;
// printf("m:%d\n",
someA.m());
// printf("n:%d\n",
someA.n());
printf("o:%d\n", someA.o());
```



```
a:1
b:2



o:5
```

Reference(&)とpointer(*)

C言語：

Code	
`int a = 123;`	a = 123, &a = 0x7fa..b (アドレス); *a = <爆発>
`int* b = NULL;`	b = NULL; &b = 0x7ef..c（アドレス）; *b = ＜爆発＞
`b = &a;`	a = ???; &a = ???; b = ????; &b = ???; *b = ???
`*b = 456;`	a = ???; &a = ???; b = ????; &b = ???; *b = ???
`int** c = &b;`	c = ???; *c = ???; **c = ???; a, &a, b, &b, *b?

`void myfun(int *x)` を実行するときに、`int a`をパラメーターに入れたい時はどう書けば？

Reference(&)とpointer(*)＜実行例＞

```
int a = 123;
printf("int a = 123;\n");
printf("a is %-20d, &a is %-20p, *a is error\n", a, &a);
int *b = NULL;
printf("int *b = NULL;\n");
printf("b is %-20p, &b is %-20p, *b is error\n", b, &b);
b = &a;
printf("b = &a;\n");
printf("a is %-20d, &a is %-20p, *a is error\n", a, &a);
printf("b is %-20p, &b is %-20p, *b is %d\n", b, &b, *b);
*b = 456;
printf("*b = 456;\n");
printf("a is %-20d, &a is %-20p, *a is error\n", a, &a);
printf("b is %-20p, &b is %-20p, *b is %d\n", b, &b, *b);
int **c = &b;
printf("int **c = &b;\n");
printf("a is %-20d, &a is %-20p, *a is error\n", a, &a);
printf("b is %-20p, &b is %-20p, *b is %d\n", b, &b, *b);
printf("c is %-20p, *c is %-20p, **c = %d\n", c, *c, **c);
```

```
int a = 123;
a is 123, &a is   0x7ffe146f9154, *a is error

int *b = NULL;
b is (nil), &b is 0x7ffe146f9158, *b is error

b = &a;
a is 123, &a is   0x7ffe146f9154, *a is error
b is 0x7ffe146f9154, &b is 0x7ffe146f9158, *b is 123

*b = 456;
a is 456, &a is 0x7ffe146f9154, *a is error
b is 0x7ffe146f9154, &b is 0x7ffe146f9158, *b is 456

int **c = &b;
a is 456, &a is 0x7ffe146f9154, *a is error
b is 0x7ffe146f9154, &b is 0x7ffe146f9158, *b is 456
c is 0x7ffe146f9158, *c is 0x7ffe146f9154, **c = 456
```

Reference(&)とpointer(*)

一つ、重要なポイントがある。

```
void myfun1a(type v)
```

vの変更は、呼出元には反映されない！

```
void myfun1b(type* vptr)
```

vptrの指すオブジェクトの変更が、呼出元でも反映される！

Reference(&)とpointer(*)

```
void myfun2a(int i)  {  i =  i * 2; }
void myfun2b(int* i) { *i = *i * 2; }

int a = 3;
myfun2a(a);     // a == 3
myfun2b(&a);    // a == 6
```

Reference(&)とby-reference (&)

```
class Elephant {
private:
  char hugeChar[1000000]; // 1MBの変数
public:
  Elephant() {
      printf("New elephant %p created\n", this);
  }
  void greet(Elephant other) {
    printf("%p greets %p\n", this, &other);
    printf("%p vs %p\n", hugeChar, other.hugeChar);
  }
};
```

Reference(&)とby-reference (&)

さてと。

```
int main() {
  Elephant myElephant;
  myElephant.greet(myElephant); // 自分自身を渡してみる
}
```

どうなるこれ？

Reference(&)とby-reference (&)

こうなる:

```
New elephant 0x7fff545fe148 created
0x7fff545fe148 greets 0x7fff54415cc0
0x7fff545fe148 vs 0x7fff54415cc0
```

つまり、呼ぶ時にコピーしてから引き渡す。
あまりに大きい引数は、一時的にメモリが大量に必要になったり、コピー処理自体のオーバーヘッドも無視できなくなるため、望ましくない。

Reference(&)とby-reference (&)

`0x7fff545fe148`

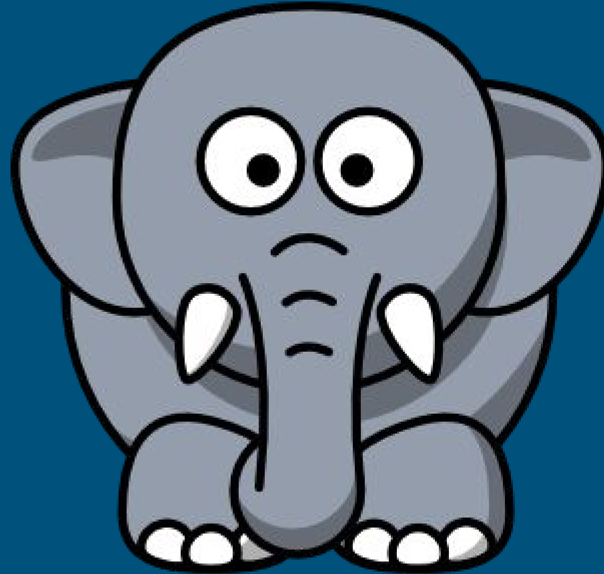
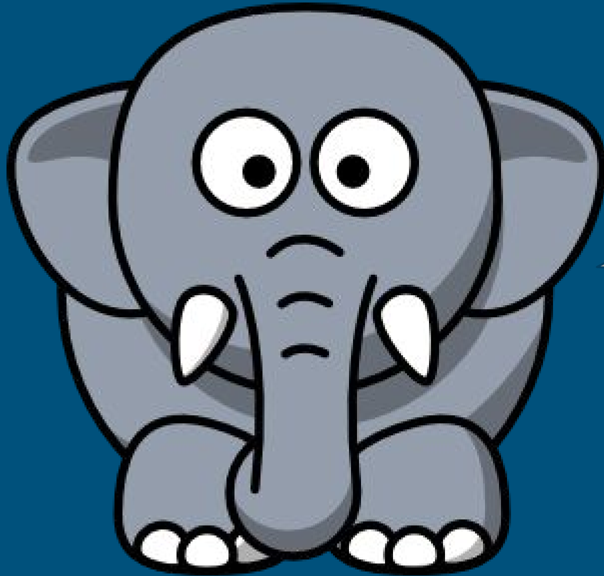


Image by AnimalsClipart

Reference(&)とby-reference (&)

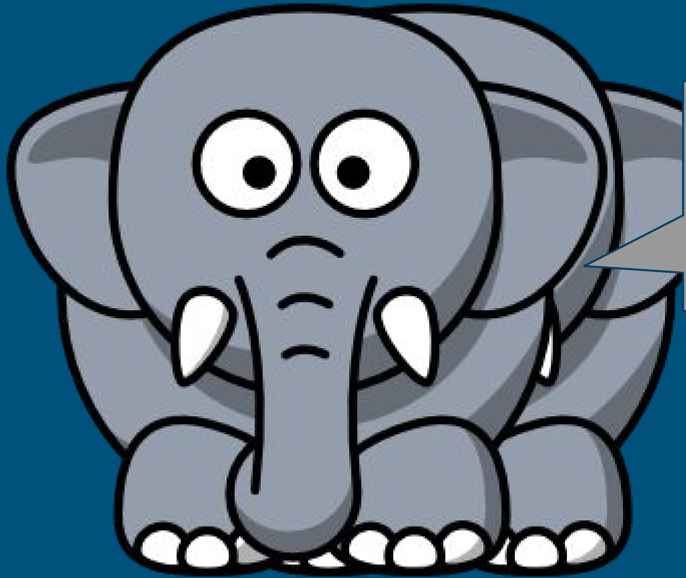
0x7fff545fe148



greet(0x7fff
545fe148)

Reference(&)とby-reference (&)

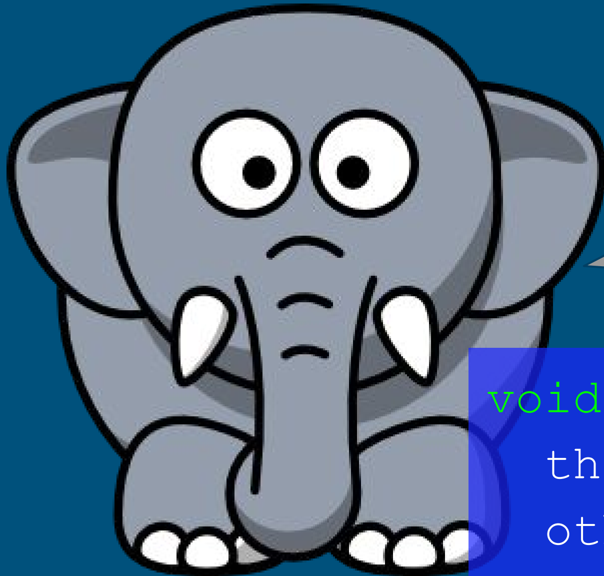
0x7ff0x545fe148.5cc0



greet(0x7fff
545fe148)

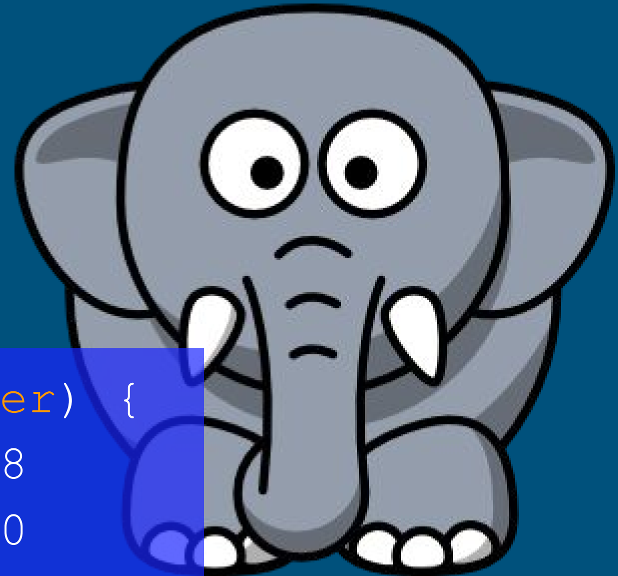
Reference(&)とby-reference (&)

0x7fff545fe148



greet(0x7fff545fe148)

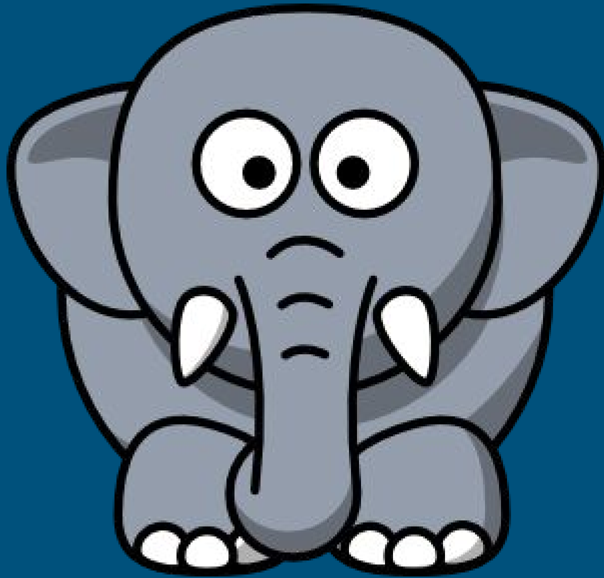
0x7fff54415cc0



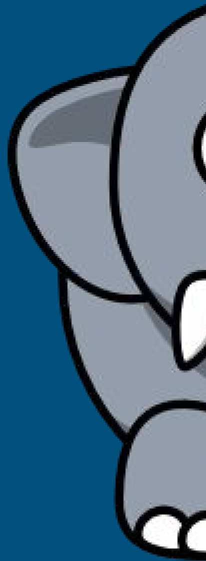
```
void greet(Elephant other) {
  this  → 0x7fff545fe148
  other → 0x7fff54415cc0
```

Reference(&)とby-reference (&)

0x7fff545fe148

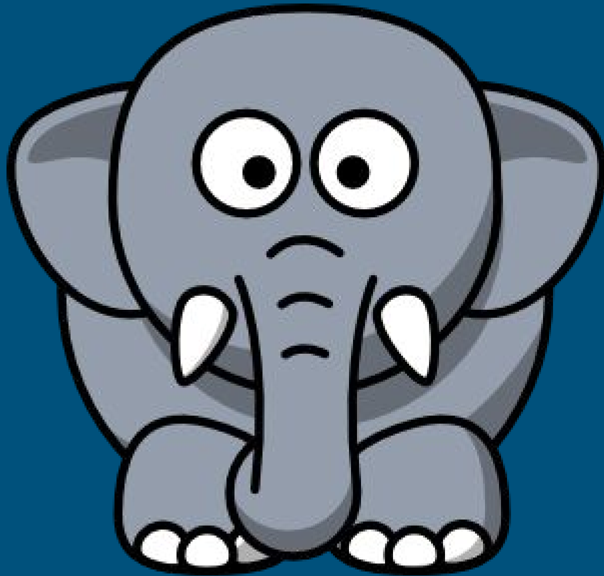


0x7f



Reference(&)とby-reference (&)

0x7fff545fe148



Reference(&)とby-reference (&)

C言語での解決方法（分かりやすくするためにクラスを使う）：

```
// class Elephant内
  void greet2(Elephant* other) {
    printf(“%p greets %p\n”, this, other);
    printf("%p vs %p\n", hugeChar, other->hugeChar);
  }
// そして, mainでは
  myElephant.greet2(&myElephant);
```

Reference(&)とby-reference (&)

C++ではもっと分かりやすい方法がある。これはby-referenceと言って自動的に

① コピーしないでオブジェクトをそのまま引き渡す（速い）

② そのまま引き渡す＝オブジェクトの中身は変えられる（mutability機能）

となる。

```
void doNotCopy(Elephant& other)   // by-reference パラメータ
```

、、、②は変えられると困る場合もある。

constとby-referenceの関係

constな変数やインスタンスを持っている時には、
・const関数しか呼べない
・publicの変数は変えられない
・constのままでしか使えない
という制限がある。

```
void foo(const Elephant& myElephant)
```

という関数の呼び出し方であれば、myElephantは絶対に変わらない(fooが違う関数を呼び出しても)。このため、コピーして欲しくない(遅いから)し、変更もして欲しくない(セーフティの為)という問題が解決する。

constとby-referenceの関係

```
void bar(Elephant& e)
void zed(const Elephant& e)

void foo(const Elephant& e) {
    bar(e);         // コンパイル時にエラーが出る
    zed(e);         // エラーは出ない。zedもeを変えられない
}
```

Reference, by-reference, pointer

	言語	アクセス	コピーされる	変えられる
<T> x	C/C++	x.var	はい	いいえ
<T>& x	C++	x.var	いいえ	はい
<T>* x	C/C++	x->var	いいえ	はい

operator(演算子)

クラスで演算子を多重定義することができる。
ただし演算子の元々の機能とかけ離れた定義にすると可読性が下がる。

```
class A {
public:
    int i = 5;
    A& operator=(int x) {
        i = x;
    }
};
```

```
int main() {
    A myA;
    int foo = 38;
    myA = foo;
    printf("myA.i=%d\n", myA.i);
    // myA.i = 38
}
```

friend（友達）機能

C++ではfriendという特殊な機能が付いている。
friendに指定されると指定元クラスのprivate変数・関数が使えるようになる。

```
class A {
private:
    int privInt;
public:
    friend class B;
};
```

```
class B {
public:
    void poke(A& anA) {
        anA.privInt = 38;
    }
};
```

クラスをfriendにすることもできるし、

friend（友達）機能

メソッドもfriendにできる。

```
class A {
private:
    int privInt = 0;
public:
    friend bool operator==(const A& first, const A& second);
};
bool operator==(const A& first, const A& second) {
    return first.privInt == second.privInt;
} // Aの外なのにprivIntにアクセスできる
```

friend(友達)機能

```
int main() {
    A a1;
    A a2;          // a1.privInt = ??? a2.privInt = ???
    B myB;
    myB.poke(a2);// a1.privInt = ??? a2.privInt = ???
    printf("same? %d\n", a1 == a2);   // 出力=???
    myB.poke(a1);
    printf("same? %d\n", a1 == a2);   // 出力=???
}
B.poke → void poke(A& anA) { anA.privInt = 38; }
```

friend(友達)機能

よく使う場面:さまざまなoperator。

```
class A {
private: int i = 0;
friend A& operator+(const A& one, const A& other);
}; // myA + myOtherA -> new A where i = myA.i + myOtherA.i

friend A& operator+(const A& one, const A& other) {
    A *tmp = new A;
    tmp->i = one.i + other.i;
    return *tmp;
}
```

const, friend (読(r)書(w)呼(x)可、r可wx不可、rwx不可)

（注意： <t> <名>（…） const という関数は呼(x)可）	Non-const	Const
public	自分 Friend 他人	自分 Friend 他人
private	自分 Friend 他人	自分 Friend 他人

virtual

```
class A {
public:
    void greet() {
        printf("Hello!\n");
    }
};
```

```
class B : public A {
public:
    void greet() {
        printf("Hi there!\n");
    }
};
```

```
void greetFun(A& a) { a.greet(); }
int main() {
    A myA; B myB;
    greetFun(myA); greetFun(myB); // 出力は?
}
```

virtual

```cpp
class A {
public:
    virtual void greet() {
        printf("Hello!\n");
    }
};
```

```cpp
class B : public A {
public:
    void greet() {
        printf("Hi there!\n");
    }
};
```

```cpp
void greetFun(A& a) { a.greet(); }
int main() {
    A myA; B myB;
    greetFun(myA); greetFun(myB); // 出力は？
}
```

virtual＜実行例＞

```
class A {
public:
  virtual void greet() {
    printf("Hello!\n");
  }
};

class B: public A {
public:
  void greet() {
    printf("Hi there!\n");
  }
};

void greetFun(A& a) { a.greet(); }
```

```
A myA;
B myB;
greetFun(myA);
greetFun(myB);
```

virtualなし →

```
Hello!
Hello!
```

virtualあり →

```
Hello!
Hi there!
```

テンプレート

C++にはテンプレートの機能がある。
テンプレートによって使用する場面に合わせた関数やクラスが生成される。

例: 以下のような関数があるとする。

```
template<typename T>
void print(T myT) {
  std::cout << myT.toString() << std::endl;
}
```

テンプレート

二つの、全く関係ないクラスに同じシグネチャのメソッドを作る。

```
class A {
    std::string toString() {
        return “myA”;
    }
};
```

```
class B {
    std::string toString() {
        return “myB”;
    }
};
```

テンプレート

するとこういう書き方ができる:

```
int main() {
    A anA;
    B aB;
    print(anA);   // “myA” - print(A myT);
    print(aB);    // “myB” - print(B myT);
}
```

クラスもテンプレートとして作ることができる。

テンプレート

例:

```
template<typename T>
class MyClass {
public:
    void print(T someT) {
        std::cout << someT << std::endl;
    }
};
```

```
MyClass<int> myIntClass;
int a = 5;
myIntClass.print(a);
```

さて、テンプレートで何ができるか考えよう！

テンプレート

C++では標準ライブラリ(Standard Template Library)としてテンプレートベースのクラスが多数存在し、よく使われている。

その中でも、

```
std::vector
std::map
```

がおそらく一番よく使われている。

テンプレート

タスク：printVecという関数（クラスは不要）を作って下さい。例：

```
std::vector<A> aVec;
std::vector<B> bVec;
aVec.resize(2); bVec.resize(3);
printVec(aVec); // 出力（二行）：myA  myA
printVec(bVec); // 出力（三行）：myB  myB  myB
```

テンプレート＜解答例＞

```
class A {
public:
  std::string toString() {
     return "myA";
  }
};

class B {
public:
  std::string toString() {
     return "myB";
  }
};

template<typename T>
void printVec(T vect) {
  for (auto item : vect) {
     printf("%s\n", item.toString().c_str());
  }
}
```

```
std::vector<A> aVec;
std::vector<B> bVec;
aVec.resize(2);
bVec.resize(3);
printVec(aVec);
printVec(bVec);
```

```
myA
myA
myB
myB
myB
```

Bitcoin Coreで使われているテンプレート

Serialize/de-Serializeという機能でテンプレートが非常によく使われている。

・ネット上でのやり取り(block、transactionの送信等)
・ディスクとの書込読込(chain、undoデータ、utxoDB等)
・RPC環境でのやり取り
・検証、ハッシュの作成

等等！

Bitcoin Coreで使われているテンプレート

主にこのように使われている:

・serialize.hの中にあらゆるタイプの基本serialize関数がある。

Bitcoin Coreで使われているテンプレート

・各クラスに ADD_SERIALIZE_METHODS;というマクロが入っている。そして、

```
template <typename Stream, typename Operation>
inline void SerializationOp(Stream& s,
Operation ser_action, int nType, int nVersion)
```

で読み書き対象のデータをREADWRITEメソッドに引き渡す。

Bitcoin Coreで使われているテンプレート

例(class COutPoint):

```
ADD_SERIALIZE_METHODS;

template <typename Stream, typename Operation>
inline void SerializationOp(Stream& s, Operation
ser_action, int nType, int nVersion) {
    READWRITE(hash);
    READWRITE(n);
}
```

Bitcoin Coreで使われているテンプレート

読み込み時と書き込み時で特有の処理が必要な場合の切り分けには`ser_action.ForRead()`を参照する。

例) `SerializeTransaction`:

```
if (ser_action.ForRead()) {    // 読込
    const_cast<std::vector<CTxIn>*>(&tx.vin)->clear();
    [...]
} else {  // 書込
    [...]
}
```

@DG Lab - Karl-Johan Alm