Copy Ctor

Copying

An Interesting Problem

- For the code below

```
void f() {
    Stash students();
    …
}
```

想要用 dtor 的话，去掉（ ）.

Which statement is RIGHT for the line in function f()?

1. This is a variable definition, while students is an object of Stash, initialized w/ default ctor.
2. This is a function prototype, while students is a function returns an object of Stash.
3. This is a function call.
4. This is illegal in C++.

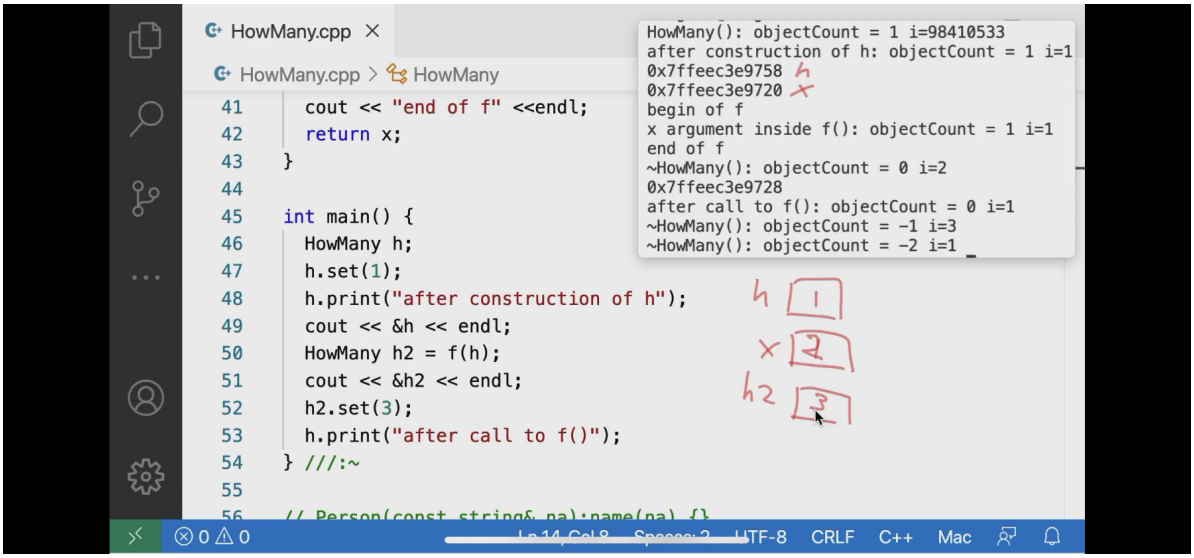
1. 要让 1 成立的话，即想要让**在函数中的**这行代码调用 `Stash` 的 default constructor 来创建一个对象的话，必须要去掉 `()`；
2. 这是古老的 C 语言延续下来的传统，可以**在函数的内部再声明一个函数**，作用域只在这个函数中；

Chase the Objects

```
1  static int objectCount = 0;      // 每次制造一个 HowMany 对象的时候加一
2  class HowMany {
3  public:
4      HowMany() { objectCount++; print("HowMany()"); }
5      // HowMany( const HowMany& r ) {
6      //     objectCount++;
7      //     this->i = r.i;
8      //     print("HowMany(HowMany)");
9      // }
10     void set( int i ) { this->i = i; }
11     void print( const string& msg = "") {
12         if ( msg.size() != 0 ) cout << msg << ": ";
13         cout << "objectCount = " << objectCount << ", i=" << i << endl;
14     }
15     ~HowMany() {
16         objectCount--;
17         print("~HowMany()");
18     }
19 private:
20     int i;
21 };
22
23 HowMany f( HowMany x ) {
24     cout << &x << endl;
```

```
    cout << "begin of f" << endl;
    x.print("x argument inside f()");
    x.set(2);
    cout << "end of f" << endl;
    return x;
}

int main() {
    HowMany h; h.set(1);
    h.print("after construction of h");
    cout << &h << endl;
    HowMany h2 = f(h);
    cout << &h2 << endl;
    h2.set(3);
    h.print("after call to f()");
    return 0;
}
```



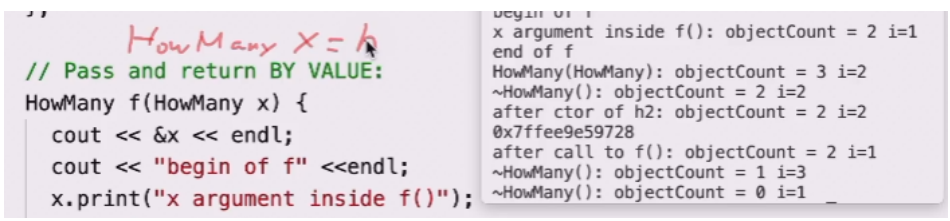
- `h, x, h2`：这三个对象的地址都是不一样的，而最后的 `objectCount = -2` 说明我们漏掉了两次对象的构造，或者说是漏掉了**不是 default constructor 构造来的对象**；

```
static int objectCount = 0;      // 每次制造一个 HowMany 对象的时候加一
class HowMany {
public:
    HowMany() { objectCount++; print("HowMany()"); }
    HowMany( const HowMany& r ) {
        objectCount++;
        this->i = r.i;
        print("HowMany(HowMany)");
    }
    void set( int i ) { this->i = i; }
    void print( const string& msg = "") {
        if ( msg.size() != 0 ) cout << msg << ": ";
        cout << "objectCount = " << objectCount << ", i=" << i << endl;
    }
    ~HowMany() {
        objectCount--;
        print("~HowMany()");
    }
private:
    int i;
```

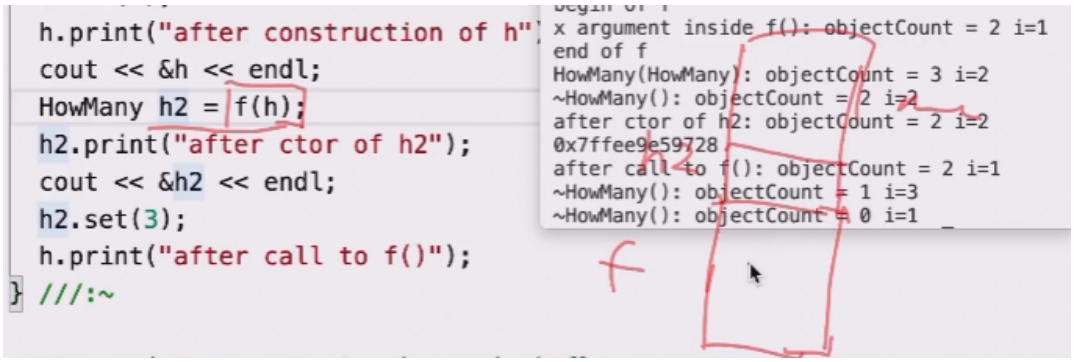
```
};

HowMany f( HowMany x ) {
    cout << &x << endl;
    cout << "begin of f" << endl;
    x.print("x argument inside f()");
    x.set(2);
    cout << "end of f" << endl;
    return x;
}

int main() {
    HowMany h;
    h.set(1);
    h.print("after construction of h");
    cout << &h << endl;
    HowMany h2 = f(h);
    h2.print( "after ctor of h2" );
    cout << &h2 << endl;
    h2.set(3);
    h.print("after call to f()");
    return 0;
}
--------------------------------------------------------------------------------
HowMany(): objectCount = 1, i=13066080
after construction of h: objectCount = 1, i=1
0x72fd60    // h
HowMany(HowMany): objectCount = 2, i=1
0x72fda0    // x
begin of f
x argument inside f(): objectCount = 2, i=1
end of f
HowMany(HowMany): objectCount = 3, i=2
~HowMany(): objectCount = 2, i=2
after ctor of h2: objectCount = 2, i=2  // h2 没有在 main 函数里面构造，而是在 f() 中
0x72fd50    // h2
after call to f(): objectCount = 2, i=1
~HowMany(): objectCount = 1, i=3
~HowMany(): objectCount = 0, i=1
```

- 我们通过 `HowMany( const HowMany& r )` 这个构造函数捕捉到了之前漏掉的两个构造：
  - **Copy Constructor(拷贝构造函数)**：拷贝构造函数的参数是自己这个类型的 reference；
  - Copy Ctor 会在传递给它的**参数是自己这个类型的一个对象**的时候被调用(函数重载嘛)
    - 

- `h2` 没有在 `main` 函数里面构造，而是在 `f()` 中进行构造的——**这是被编译器优化了**：
  - 我们 C 语言在返回 `int` 类型的对象和一个结构体对象的时候会有所区别：
    - `return int`：这个 `int` 是在 Reigister 里面的；

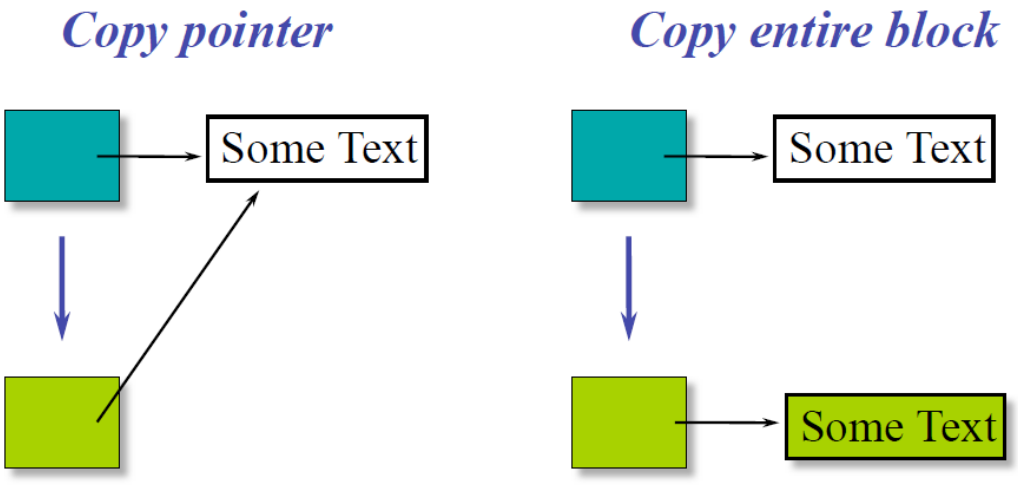
- `return structure`：这个 `structure` 里面的值是在 stack(caller 知道它的地址) 里面的；
  - 

    - `h2` 的地址是在 `main` 函数内存段的，`main` 是 caller，它的对象的赋值是在 callee `f()` 里面进行的

- [x] **进入所有函数的时候，这个函数里面所有对象的空间全部都有，只是它们的值要到执行到它们的构造时才填。**
- [x] 如果我们没有提供拷贝构造，它会执行一个默认的拷贝构造，member-wise copy(和书写顺序一样)；如果我们自己提供了拷贝构造的话，那么它做的就是我们写的东西；

- Has the unique signature
  `T::T(const T&);`
  - Call-by-reference is used for the explicit argument
- C++ builds a copy ctor for you if you don't provide one!
  - Copies each member variable
    - Good for numbers, objects, arrays
  - Copies each pointer
    - Data may become shared!

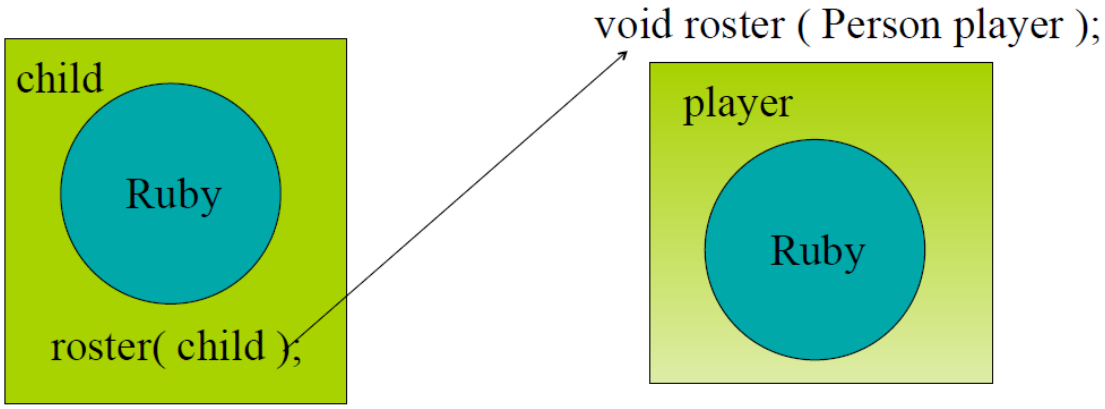
  - 要是拷贝构造里面有指针的话，就可能出现问题，**data 可能被两个对象所 share**，于是在析构的时候就可能出现问题；



When are copy ctors called?

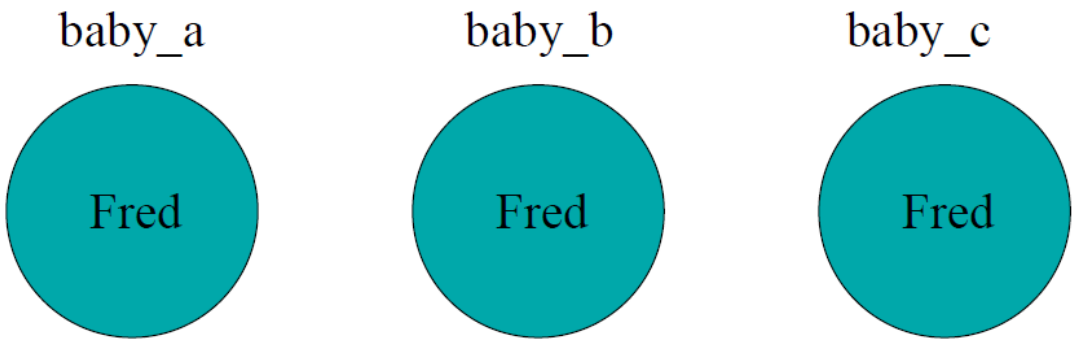
1. During call by value:

```
void roster( Person );  // declare function
Person child( "Ruby" ); // create object
roster( child );        // call function
```



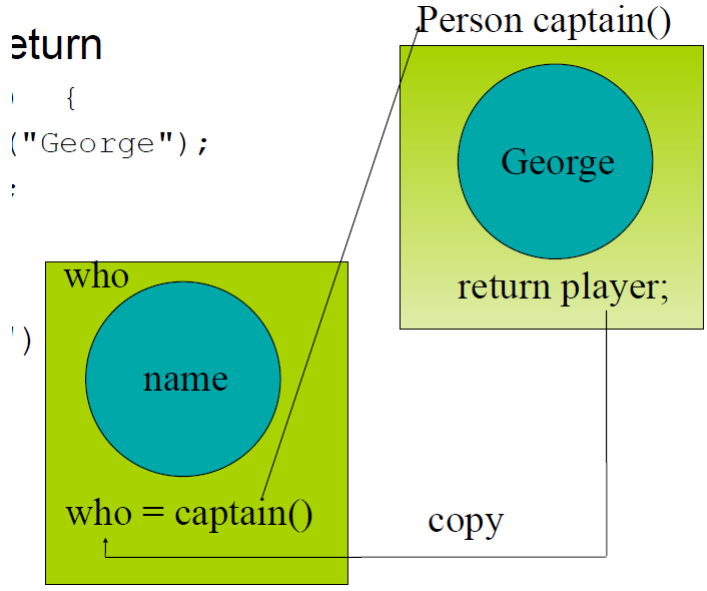
2. During initialization:

```
Person baby_a("Fred");      // these use the copy ctor
Person baby_b = baby_a;     // not an assignment
Person baby_c( baby_a );    // not an assignment
```



3. During function return:

```
Person captain() {
    Person player("George");
    return player;
}
...
Person who("");
...
```



Constructions vs. assignment

- Every object is **constructed once**.

```
HowMany h2 = h; // 构造新对象，调用拷贝构造
h2 = h;         // 对象之间的赋值
```

- Every object should be destroyed once.
  - Failure to invoke `delete()`;
  - Invoking `delete()` more than once;
- Once an object is constructed, it can be the target of many assignment operations.

Copy ctor guidelines

- In general, be explicit
  - Create your own copy ctor -- don't rely on the default
- If you don't need one declare a private copy ctor
  - prevents creation of a default copy constructor
  - generates a compiler error if try to pass-by-value
  - don't need a defintion