

CSCI-3081: Program Design and Development

C++ NAME CONTROL, CONSTANTS, AND COPY CONSTRUCTOR (ECKEL CHPS. 8, 10, 11)

CSCI-3081: Program Design and Development

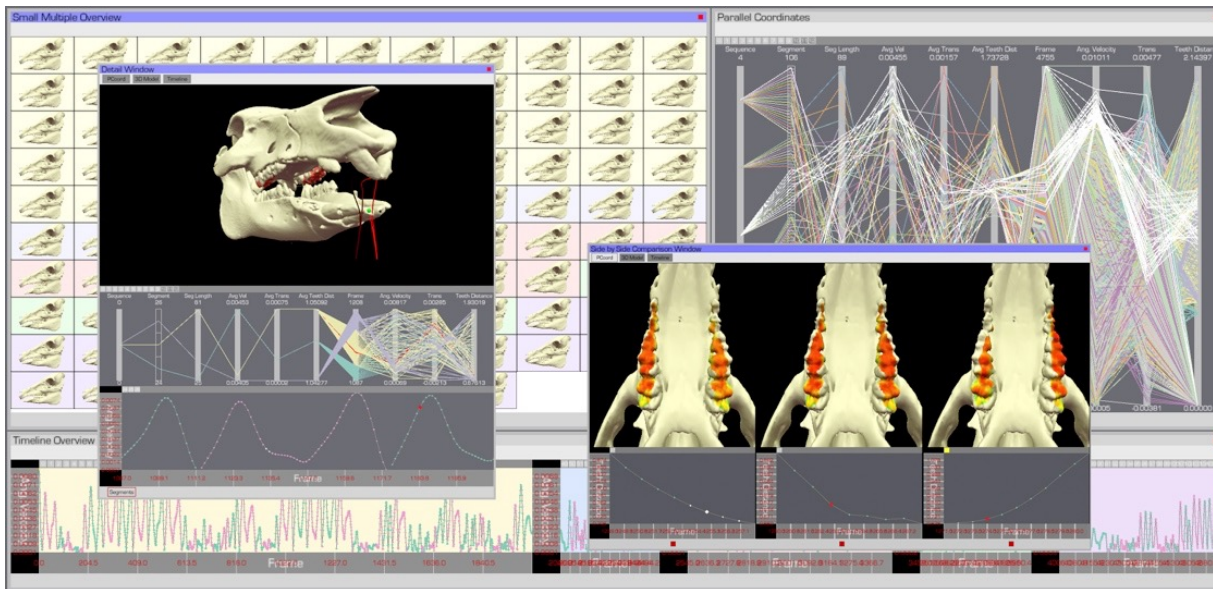
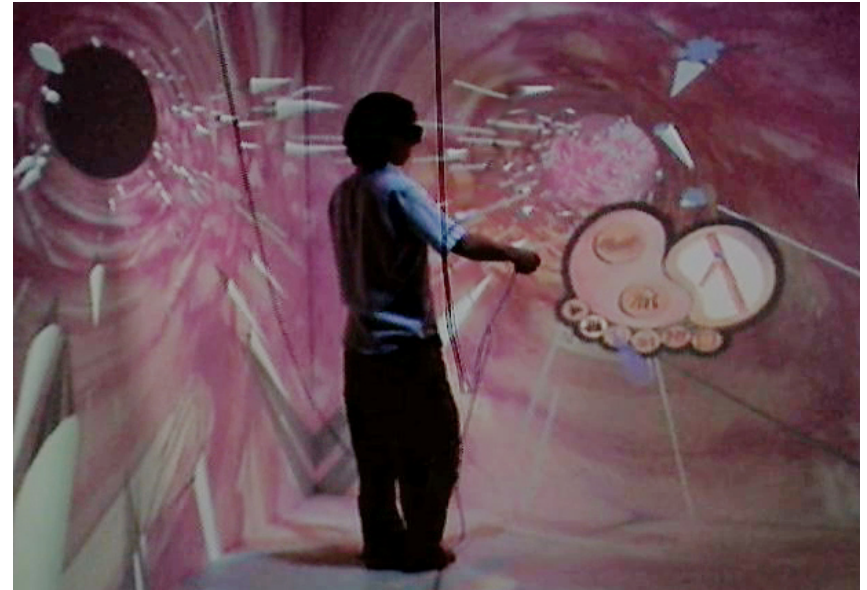
C++ NAME CONTROL

A “Real” Exercise

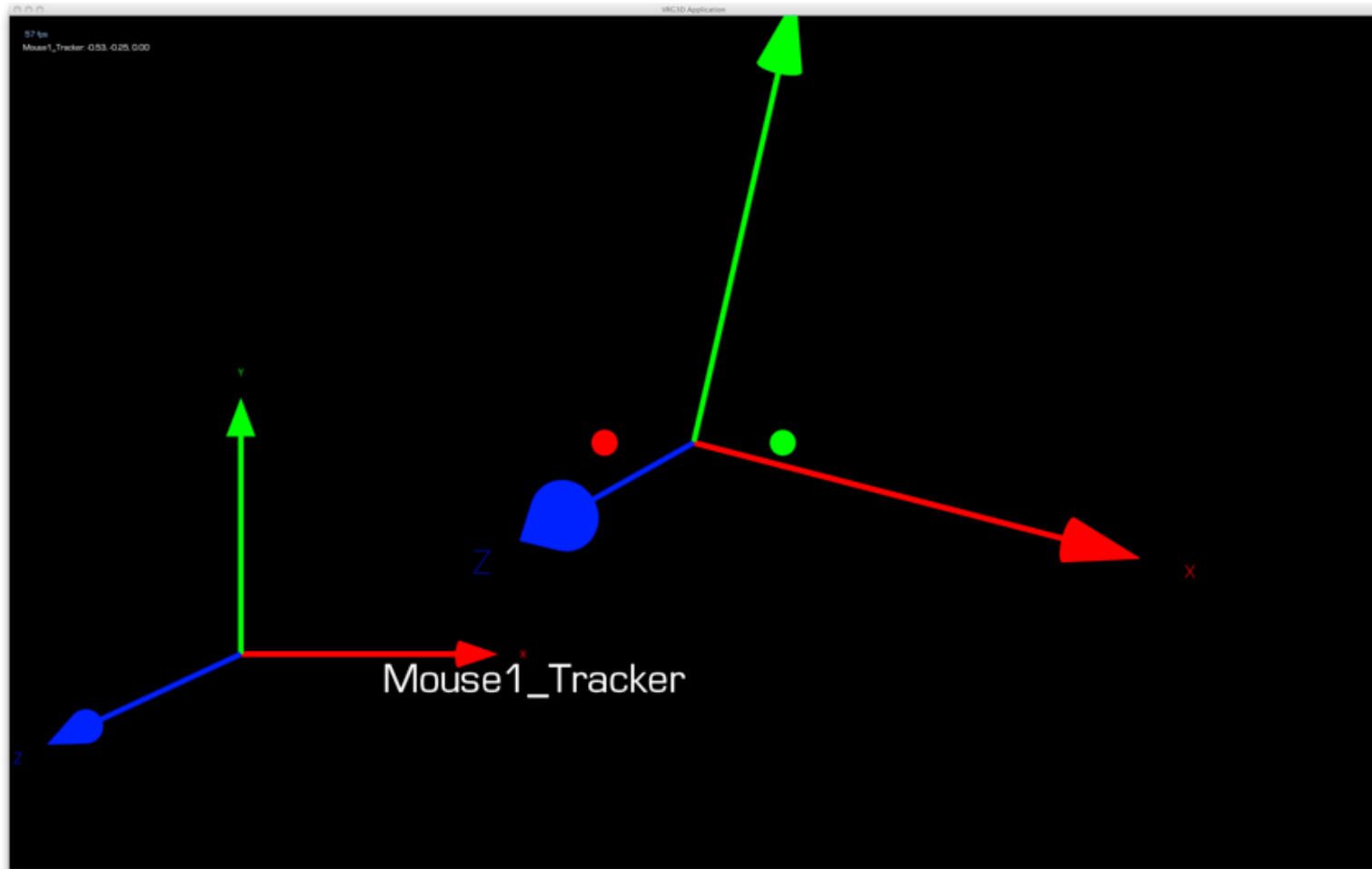
- We’ll look at a simple demo program provided with the VRG3D library.
- VRG3D is an open source library for virtual reality applications.
 - Builds on a great C++ 3D computer graphics / games library called G3D. (You can try G3D out, it’s available at <http://g3d.sourceforge.com>)
 - Extends G3D to support virtual reality devices and displays.

Example VRG3D Applications

These have all been built on top of the code you're about to work with.



Demo Application (Hello World for VRG3D)



- Draws some 3D axes.
- You can rotate these around using keyboard and mouse input.
- Uses a font to draw the “frames per second” and other debug info in the top left corner of the screen.

Activity: Name Control in the VRG3D Demo

- Goal: Identify and understand specific uses of name control (namespaces, static members and functions) that show up in this real example.

```
#include <VRG3D.H>
using namespace VRG3D; ← A

/** A simple virtual reality application built using the VRG3D library. VRApp is a
    base class defined within the VRG3D namespace. Subclasses of VRApp should
    re-implement two key virtual methods: doUserInput() and doGraphics(). */
class MyVRApp : public VRApp {
public:
    MyVRApp(const std::string &mySetup);

    ~MyVRApp();

    void doUserInput(Array<VRG3D::Event*> &events);
    void doGraphics(RenderDevice *rd);

private:
    static GFont*      m_font; ← B
    CoordinateFrame    m_worldTransform;
};
```

GFont* MyVRApp::m_font = NULL; ← **B**

```
MyVRApp::MyVRApp(const std::string &mySetup) : VRApp(mySetup) {
    if (m_font == NULL) {
        std::string fontfile = VRApp::findVRG3DDataFile("eurostyle.fnt"); ← C
        if (fileExists(fontfile)) {
            m_font = GFont::fromFile(fontfile); ← C
        }
    }
}
```

MyVRApp::~~MyVRApp() {}

A

```
void MyVRApp::doUserInput(Array<VRG3D::Event*> &events) {
    for (int i=0;i<events.size();i++) {
        // Rotate the 3D world when the user presses the left arrow key.
        if (events[i]->getName() == "kbd_LEFT_down") {
            Matrix3 rotation = Matrix3::fromAxisAngle(Vector3(0,1,0), toRadians(5.0));
            m_worldTransform = CoordinateFrame(rotation) * m_worldTransform;
        }
    }
}
```

```
void MyVRApp::doGraphics(RenderDevice *rd) {
    if (m_font != NULL) {
        std::string msg = format("%3d fps", iRound(rd->frameRate()));
        m_font->draw2D(rd, msg, Vector2(25,25), 12, Color3(0.61, 0.72, 0.92));
    }
}
```


Namespaces

- The **using directive** (e.g. `using namespace VRG3D;`) imports the entire namespace at once.
- The **using declaration** (e.g. `using VRG3D::VRApp;`) allows you to inject specific names into the current scope one at a time.
- You can resolve name ambiguity by fully specifying the name of a class, function, etc. using the namespace.
 - Writing **`VRG3D::Event`** rather than just `Event` tells the compiler we want the specific `Event` class that is defined in the `VRG3D` namespace, not some other `Event` defined elsewhere.

Static Data Members

- Like global variables, but stored within a class.
- Occupy a single piece of storage regardless of how many objects of the class are created.
- Because of this, we need to define the storage in the implementation file:
 - Example, in .cpp file we see: `GFont* MyVRApp::m_font = NULL;`
 - **The linker will exit with an error if you forget to define this storage.**

Static Member Functions

- Like global functions, but defined within a class.
- Like static data members, these work for the class as a whole rather than for a particular object of the class.
- You can call these functions in the “ordinary way” using `obj.function()` or `objPtr->function()` but since they aren’t associated with a particular object, usually we call them using `Class::function()` like this:
 - `VRAApp::findVRG3DDataFile(..)`
- **IMPORTANT POINT: Here’s an interesting case. It creates a new GFont.**
 - `m_font = GFont::fromFile(..);`
- **Why not just use a constructor, like this:**
 - `m_font = new GFont(..);`

Note: Order of Static Initialization

- Static variables are initialized before your “normal” code runs:
 - Global static objects (not inside classes) are initialized before `main()` is called.
 - Static data members in classes are initialized before any other code in the translation unit (.o file) is executed.
- But, there are no guarantees about the order of initialization across translation units.
- Be very, very careful with (or just avoid) using static variables that depend upon each other.
- Additional reading in Eckel about strategies for controlling the order of initialization -- these are advanced topics. Usually, a good design avoids the need for these.

CSCI-3081: Program Design and Development

C++ CONSTANTS

Eckel Chapter 8

- “The concept of constant (expressed by the `const` keyword) was created to allow the programmer to draw a line between what changes and what doesn't. This provides safety and control in a C++ programming project.”
- How?

How can `const` be used?

- Depending how you count, Eckel describes perhaps 4 to 8 different ways that **`const`** can be used and goes into great detail about how to use `const` in several situations.

Ways that const can be used

- As a replacement for preprocessor `#define`'s
- At runtime: “safety const”
- With pointers (requires some special discussion)
- In function arguments
- In function return types (including for temporaries)
- With classes (static const, const objects, member func, mutable)

1. As a replacement for preprocessor `#define`'s

- Review: What's the **preprocessor** anyway?
- Can define constant values using the preprocessor like this:
 - `#define BUFSIZE 100`
 - ...
 - `int str_buf[BUFSIZE];`
- This is better than sprinkling 100 throughout your code.
- But, using `const` is a much better practice:
 - `const int bufsize = 100;`
 - ...
 - `int str_buf[bufsize];`
- Why? What are the advantages?

Advantages of const vs. preprocessor

- Because the value of bufsize is known at compile time and the compiler knows it is constant -- it will never change....
- In most cases, the compiler doesn't need to allocate storage for bufsize, it can be "compiled away".
- The compiler can also do "constant folding" -- simplifying constant expressions at compile time.
 - `i = 320 * 200 * 32;`
 - becomes `i = 2,048,000;`
- With the const modifier we also have type information (we know bufsize is an int), so the compiler can do type checking to prevent bugs.

2. At runtime “safety const”

- We can also set constants at runtime.
- Again, const means that the value of the variable cannot change, but in this case we don't know the value until the program runs.
- In this case, the compiler will have to allocate storage for the variable.
 - This means sometimes when you use const the compiler doesn't need to allocate storage for the variable, but sometimes it does.
 - It can be helpful to understand the different cases, in order to be able to analyze the performance of your programs.
- Why do we call this use of const “safety const”?

Example: Eckel's Safecons.cpp

```
#include <iostream>
using namespace std;

const int i = 100;           // Typical constant
const int j = i + 10;        // Value assigned from a const expression
long address = (long)&j;     // Forces storage of j
char buf[j + 10];           // Still a const expression, does folding

int main() {
    cout << "type a character & return:";
    const char c = cin.get(); // Can't change, only avail at runtime
    const char c2 = c + 'a';
    cout << c2;
}
```

3. Special Considerations with Pointers

- What do we want to make constant for pointers?
- Key Idea #1: There are two (actually three) answers:
 - The pointer itself
 - The thing that it points to
 - Both the pointer and the thing that it points to
- How?
- Key Idea #2: The syntax can be confusing.
 - `const int * u; // const modifies int here.. so u is a pointer to a const int.`
 - `int const * u; // strange, but this is completely equivalent to the line above.`
 - `int * const u; // this is a const pointer to an int`

Const Pointers Example

```
int d = 1;  
int * const w = &d;
```

- Is `*w = 2;` legal?
- Yes. The value of the pointer is constant, but not the value of the int that it points to.
- If we wanted to make both const, we would do this:

```
const int * const x = &d;  
// Or equivalently..  
int const * const x2 = &d;
```

Const Type Checking: A Key Concept for the Remaining Uses of Const

- You can assign the address of a non-const object to a const pointer.
- But, you **cannot** assign the address of a const object to a non-const pointer -- this breaks the promise that it cannot change.


```
int d = 1;
const int e = 2;
int* u = &d;           // ok because d is not const
int* v = &e;           // illegal, e is const, but v is not
```

4. Const in Function Arguments

```
void foo(const int i) {  
    i++; // Illegal, i is const, so we can't modify it  
}
```

- So, if a function parameter is declared as const, then it cannot be modified inside the function.
- What about the impact on the code that calls this function?
 - The programmer knows that the function will not change i.
 - But, in the case above we know this anyway because i is passed by value, so the use above can be a bit confusing.

Preferred: Pass by Constant Reference



```
void foo(const std::string &s) {  
    cout << s << endl; // Ok, prints, but doesn't change s  
    s = "Hello";        // Illegal, s cannot be changed  
}
```

- Whenever you can, “pass by constant reference” is the preferred way to pass arguments to functions in C++.
- To the client, it behaves the same way as pass by value.
 - The parameter is declared as a const, so its value cannot change within the function.. same effect as if we passed-by-value.
- But, it is more efficient
 - Do not need to copy the the value of the parameter to a new variable, we just use the reference to the original.
- Same effect as pass by value, but more efficient.

5. Const in Function Return Types and Temporaries

- What happens if you call functions like this: `g(f());`
- The result of `f()` will get passed to `g()`.
- The result of `f()` is “temporary”, we can’t access it, but the compiler needs to store it somewhere so it can pass it on to `g()`.
- In C++ all temporaries like this are automatically `const`.

Example with Temporaries

```
class Box { // Stores a 3D box with a width, height, depth
};

Box cube() { // returns a Box that is a standard cube
}

Box smallerBox(Box &origBox) {
    // returns a Box that is smaller than the original
}

Box largerBox(const Box &origBox) {
    // returns a Box that is larger than the original
}

int main() {
    // Error const temporary returned by cube() can't be passed by reference to the
    // non-const parameter of the smallerBox function.
    Box b1 = smallerBox(cube());

    // This works because the parameter for largerBox is const.
    Box b2 = largerBox(cube());
}
```

Returning Pointers to a Const



```
const char* v() {  
    // Returns address of static character array:  
    return "result of function v()";  
}
```

- You can also have a const return type for a function. (e.g. Returning a pointer to a const char array.)
- This is useful in the situation above to return the address of a static character array.
- Additional examples in Eckel.


6. Const with Classes

- Const can be used in similar ways within classes, but the syntax changes somewhat.
- Constants for each instance of a class.
- Compile-time constants for all instances of a class.
- Const member functions.

Const Fields in Classes

```
class Fred {  
    // This value of size cannot change for each instance  
    // of Fred, but each instance might have a different value.  
    const int size;  
public:  
    Fred(int sz);  
    void print();  
};  
  
Fred::Fred(int sz) : size(sz) {  
}  
  
void Fred::print() {  
    cout << size << endl;  
}  
  
int main() {  
    Fred a(1), b(2), c(3);  
    a.print();  
    b.print();  
    c.print();  
}
```

important: the constructor initializer list



Compile Time (Static) Const Fields

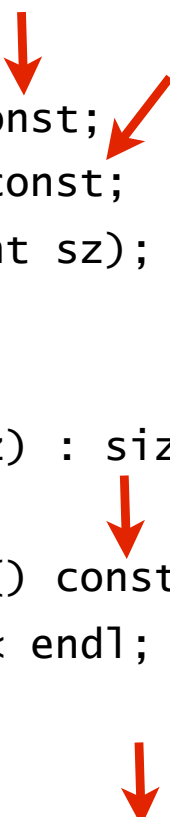
```
class Fred {  
    // This value of size will be treated as a compile-time constant within  
    // the Fred class.  
    static const int size = 100;  
    ...  
};
```

- Similar to the use of const as a replacement for #defines.
- One constant for **all** instances of the class and the constant is only available within the class.
- **static const int bufsize = 100;** Add the static keyword and initialize right where it is defined.

Const Member Functions

- Declare class member functions const when calling them would not result in a change to the data stored by the class.

```
class Fred {  
    int size;  
public:  
    Fred(int sz);  
    void print() const;  
    int getSize() const;  
    void setSize(int sz);  
};  
  
Fred::Fred(int sz) : size(sz) {}  
  
void Fred::print() const {  
    cout << size << endl;  
}  
  
int Fred::getSize() const {  
    return size;  
}
```



```
int main() {  
    // Make a constant Fred object  
    const Fred myFred(1);  
  
    // Even though myFred is const, it's  
    // ok to call print and getSize  
    // because these methods are  
    // declared const -- they don't  
    // change myFred.  
    myFred.print();  
    int mySize = myFred.getSize();  
  
    // Not ok to call myFred.setSize(2);  
}
```


CSCI-3081: Program Design and Development

C++ COPY CONSTRUCTOR

References Warm Up (1)

```
void f(int *x) {  
    (*x)++;  
}
```

```
void g(int &x) {  
    x++;      // Same effect as in f()  
}
```

```
int main() {  
    int a = 0;  
    f(&a);    // Ugly, but explicit  
    g(a);     // Clean, but hidden (do we expect “a” to be modified by “g” or not?)  
}
```

References Warm Up (2)

```
int& h() {  
    int q;  
    return q; // Error: q is only valid inside this function so can't return a  
              // reference to it.  
}
```

References Warm Up (3)

```
void f(int &i) {}
```

```
void g(const int &i) {}
```

```
int main() {  
    f(1); // Error, int &i requires an i that can be modified  
    g(1); // The const means the i cannot be modified, so this is ok.  
}
```

Copy Constructors

- A special constructor that is called to create a new object as a copy of an existing one.
- Objects are copied a lot (e.g. passing arguments by value) so this is an important constructor.
- It's so important that a copy constructor is automatically created for each class behind the scenes for you.
 - But, watch out... the automatic version might not always do what you want, so you may need to override it to make it work correctly -- we'll see examples.
- Copy constructors can be confusing.
 - Objects need to be copied a lot. You're probably writing code now that calls a copy constructor without realizing it.
 - You don't always need to create a copy constructor, the default one often works, but there are cases when it doesn't. So you need to really understand both when copy constructors are called and when you'll need to provide your own constructor rather than using the default version.

The Default Copy Constructor

- The default copy constructor just does a bitcopy of the object.
- Often, this is fine... sometimes it doesn't do what we expect.
- See the `howmany.cpp` example from Eckel.

```

ofstream out("HowMany.out");

// A class that counts its objects
class HowMany {
    static int objectCount;
public:
    HowMany() { objectCount++; }
    static void print(const string& msg = "") {
        if(msg.size() != 0) out << msg << ": ";
        out << "objectCount = " << objectCount << endl;
    }
    ~HowMany() {
        objectCount--;
        print("~HowMany()");
    }
};

int HowMany::objectCount = 0;

// Pass and return BY VALUE:
HowMany f(HowMany x) {
    x.print("x argument inside f()");
    return x;
}

int main() {
    HowMany h;
    HowMany::print("after construction of h");
    HowMany h2 = f(h);
    HowMany::print("after call to f()");
} ///:~

```

OUTPUT:

```

after construction of h: objectCount = 1
x argument inside f(): objectCount = 1
~HowMany(): objectCount = 0
after call to f(): objectCount = 0
~HowMany(): objectCount = -1
~HowMany(): objectCount = -2

```

The Problem in One Sentence

- The compiler's assumption is that copying an object can be done using a bitcopy, and in many cases this may work fine, but in HowMany it doesn't fly because the meaning of initialization goes beyond simply copying the bits.

User-Defined Copy Constructors

- We can fix this with a user-defined copy constructor.

The Form of the Copy Constructor

- The copy constructor takes 1 parameter -- a reference (preferably const) to an object of the same type.
- `MyClass(const MyClass &c);` // In .h file
- `MyClass::MyClass(const MyClass &c) { } // In .cpp file`

Which is the Copy Constructor for a Class Named X?

- `X(X copyFromMe);`
- `X(const X copyFromMe)`
- `X(X ©FromMe)`
- `X(const X ©FromMe)`

- Why do we need the reference in `X(const X& copyFromMe)`?

HowMany Version 2

- Add a copy constructor that does the right thing.
 - increments objectCount rather than just doing a bitcopy.
 - some additional printouts added as well.

```
// Regular constructors  
HowMany2() {}
```

```
HowMany2(const std::string &myName) {  
    name = myName;  
}
```

```
// The copy-constructor:  
HowMany2(const HowMany2& h) : name(h.name) {  
    name += " copy";  
    ++objectCount;  
    print("HowMany2(const HowMany2&)");  
}
```

```

ofstream out("HowMany2.out");

class HowMany2 {
    string name; // Object identifier
    static int objectCount;
public:
    HowMany2(const string& id = "") : name(id) {
        ++objectCount;
        print("HowMany2()");
    }

    ~HowMany2() {
        --objectCount;
        print("~HowMany2()");
    }

    // The copy-constructor:
    HowMany2(const HowMany2& h) : name(h.name) {
        name += " copy";
        ++objectCount;
        print("HowMany2(const HowMany2&)");
    }

    void print(const string& msg = "") const {
        if(msg.size() != 0)
            out << msg << endl;
        out << '\t' << name << ": "
            << "objectCount = "
            << objectCount << endl;
    }
};

```

```

int HowMany2::objectCount = 0;

// Pass and return BY VALUE:
HowMany2 f(HowMany2 x) {
    x.print("x argument inside f()");
    out << "Returning from f()" << endl;
    return x;
}

int main() {
    HowMany2 h("h");
    out << "Entering f()" << endl;
    HowMany2 h2 = f(h);
    h2.print("h2 after call to f()");
    out << "Call f(), no return value"
        << endl;
    f(h);
    out << "After call to f()" << endl;
} ///:~

```

OUTPUT:

```

1) HowMany2()
2)   h: objectCount = 1
3) Entering f()
4) HowMany2(const HowMany2&)
5)   h copy: objectCount = 2
6) x argument inside f()
7)   h copy: objectCount = 2
8) Returning from f()
9) HowMany2(const HowMany2&)
10)  h copy copy: objectCount = 3
11) ~HowMany2()
12)  h copy: objectCount = 2
13) h2 after call to f()

```

When Are Copy Constructors Called?

When an object is created from another object during initialization:

- `Class a = b; // Copy constructor called, a = copy of b.`
- `a = b; // Note, this is different, the assignment operator is called here`

When an object is created from another object as a parameter to a constructor: `Class a(b);`

When an object is passed-by-value as an argument to a function: `function(Class a);`

When an object is returned from a function: `Class a; ... return a;`

When an exception is thrown using this object type: `Class a; throw a;`

When to Define Your Own Copy Constructor

- When an object uses it's own pointers or non-shareable references, such as to a file.
 - A bitwise copy will just copy the address of the pointer, not allocate new memory for whatever that pointer points to.
 - After the copy the original and the copy objects would have pointers pointing to the same memory.
- When you need to control how many instances of a class exist.
 - Howmany example from earlier.
 - Next example: Singletons -- enforce that only a single instance of a class exists.

“Singletons” via **static** and a **Copy Constructor**

- Make sure that only one Egg object can ever be created.

```
class Egg {
public:
    // Return the single instance of Egg.
    static Egg* instance() { return &e; }

    // Returns the data for this egg
    int getData() const { return data; }

private:
    static Egg e; // Static member variable
    int data;     // Regular member variable

    Egg(int d) : data(d) {} // Private constructor, can't call from outside the class
    Egg(const Egg&);        // Private copy constructor, prevents copies
};

// Define storage and initialize the static member variable e.
Egg Egg::e(47);

int main() {
    //! Egg x(1); <--- error, can't do this.

    // Instead, this is how you access the single instance of Egg.
    cout << Egg::instance()->getData() << endl;
}
```