

CIS 330

C/C++ and Unix

Lecture 14

Copy Constructors

Return Values

If you say that a function's return value is const

```
const int g() {int a = 1; return a;}
```

You are promising that the original variable (a from inside the function) will not be modified

- Again, this is implicit, since you're returning by value (it's a **copy** of the original variable inside the function, which will be destroyed when the function ends)

For **built-in types**, avoid returning value as const (as it can lead to more confusion and has not real impact)

However, things are different for **objects**

Return Values

`const` becomes more important when using user-defined types (e.g., classes)

If a function returns a class object as `const`, the return value of that function **cannot** be an l-value (i.e., it cannot be assigned or modified)

L-value - something that points to a memory location

- Exists as variables and lives longer

R-value - something that does not point to anything

- Temporary and short lived

```
int x = 666; // x is l-value, 666 is r-value
```

Left operand of an assignment operator must be an l-value,

Temporaries

During the evaluation of an expression, compiler must create temporary objects

- `int x = x * y + z;`
- `x * y` must be stored in a temporary before `z` is added
- They require storage, and are constructed and destroyed (like any other objects)
- They are **automatically** `const`
- Modifying a temporary is most likely a mistake/bug (even though the compiler may allow it)

Summary

Fundamental data types returned by a function

- returned as `const`
- cannot be an l-value
- specifying the function as `const` is unnecessary (e.g., `const int foo(int i)`)

Objects returned by a function

- Not returned as a `const`
- **Can** be an l-value
- Must be specified as `const` if you don't want to modify it (i.e., use it as an l-value)

```
X f5() {  
    return X();  
}  
  
f5() = X(1);  
f5().modify();
```

Temporaries

- `const` by default
- `(f5() + f5()).modify()` - illegal

Summary

What type should a function argument be

- When you want to
 - pass by reference - pass it by reference (e.g., `int& a`)
 - pass by value
 - i. pass it by value (e.g., `int a`)
 - ii. pass it by reference (because it requires less storage) but specify it as `const` so that the original variable cannot be modified

Example

```
void t(int*) {}

void u(const int* cip) {
    //! *cip = 2; // Illegal

    int i = *cip; // OK -- copies
value

    //! int* ip2 = cip; // Illegal:
non-const
}

const char* v() {
    // Returns address of static
character array:

    return "result of function v()"
}

const int* const w() {
    static int i;

    return &i;
}

int main() {
    int x = 0;

    int* ip = &x;

    const int* cip = &x;

    t(ip); // OK

    //! t(cip); // Not OK

    u(ip); // OK

    u(cip); // Also OK

    //! char* cp = v(); // Not OK

    const char* ccp = v(); // OK

    //! int* ip2 = w(); // Not OK

    const int* const ccip = w();
// OK

    const int* cip2 = w(); // OK

    //! *w() = 1; // Not OK

    } ///:~
```

Example

```
void t(int*) {}

void u(const int* cip) {
    //! *cip = 2; // Illegal

    int i = *cip; // OK -- copies
    value

    //! int* ip2 = cip; // Illegal:
    non-const
}

const char* v() {
    // Returns address of static
    character array:

    return "result of function v()"
}

const int* const w() {
    static int i;

    return &i;
}

int main() {
    int x = 0;

    int* ip = &x;

    const int* cip = &x;

    t(ip); // OK

    //! t(cip); // Not OK

    u(ip); // OK

    u(cip); // Also OK

    //! char* cp = v(); // Not OK

    const char* ccp = v(); // OK

    //! int* ip2 = w(); // Not OK

    const int* const ccip = w();
    // OK

    const int* cip2 = w(); // OK

    //! *w() = 1; // Not OK

    } ///:~
```


Example

```
void t(int*) {}

void u(const int* cip) {
    //! *cip = 2; // Illegal

    int i = *cip; // OK -- copies
    value

    //! int* ip2 = cip; // Illegal:
    non-const
}

const char* v() {
    // Returns address of static
    character array:

    return "result of function v()"
}

const int* const w() {
    static int i;

    return &i;
}

int main() {
    int x = 0;

    int* ip = &x;

    const int* cip = &x;

    t(ip); // OK

    //! t(cip); // Not OK

    u(ip); // OK

    u(cip); // Also OK

    //! char* cp = v(); // Not OK

    const char* ccp = v(); // OK

    //! int* ip2 = w(); // Not OK

    const int* const ccip = w();
    // OK

    const int* cip2 = w(); // OK

    //! *w() = 1; // Not OK

    } ///:~
```

Example

```
void t(int*) {}

void u(const int* cip) {
    //! *cip = 2; // Illegal

    int i = *cip; // OK -- copies
    value

    //! int* ip2 = cip; // Illegal:
    non-const
}

const char* v() {
    // Returns address of static
    character array:

    return "result of function v()"
}

const int* const w() {
    static int i;

    return &i;
}

int main() {
    int x = 0;

    int* ip = &x;

    const int* cip = &x;

    t(ip); // OK

    //! t(cip); // Not OK

    u(ip); // OK

    u(cip); // Also OK

    //! char* cp = v(); // Not OK

    const char* ccp = v(); // OK

    //! int* ip2 = w(); // Not OK

    const int* const ccip = w(); // OK

    const int* cip2 = w(); // Why is
    this OK?

    //! *w() = 1; // Not OK

    } ///:~
```

Standard Argument Passing

It is possible to pass a temporary object to a function that takes `const` **reference** (but not a function that accepts a **pointer**)

```
class X {};  
  
X f() { return X(); } // Return by value  
  
void g1(X&) {} // Pass by non-const reference  
  
void g2(const X&) {} // Pass by const reference  
  
int main() {  
    // Error: f() creates a temporary (which is const)  
    //! g1(f()); // g1 accepts non-const, so this is illegal  
  
    // OK: g2 takes a const reference, so it's legal  
    g2(f());  
}
```

Standard Argument Passing

This can be confusing - aren't addresses and references the same thing?

- Yes and no - references are (likely) implemented using addresses
- However, references can be considered a "safe" address - so compiler just does not allow you to access the address of a variable but allows you to pass a variable's reference so its content can still be changed

const and Classes

Meaning of `const` **inside** a class is slightly different

You can make an entire object (of a class) `const` (e.g., a temporary object), but preserving the `const`-ness of an object is more complex

- Remember, the compiler enforces the `constness` of built-in types, but with the complexity of classes, it's more difficult to do so
- To guarantee the `constness` of a class object, `const` member functions are used - only a `const` member function may be called for a `const` object

const in Classes

Let's say you want to `#define` an array size inside a class

- A `const int` inside the class does not produce the desired effect
- It reverts to the C definition - it's a storage allocated inside the object, initialized once, and cannot be changed for the lifetime of the **object**
- However, **each object** may hold a **different** value for this `const int`
- By its definition (i.e., `#define`), it should not change across objects for the same class

const in Classes

When you create a `const` **inside** a class, you cannot give it an initial value (and it's okay, since storage is allocated for it)

Initialization must occur in the constructor (like other variables), but in a special place in the constructor

Because `const` must be initialized when it is **created**, it must already be initialized when you reach the main body of the constructor (remember, storage is allocated for an object, and then the constructor is called to initialize it)

const in Classes

Use the **constructor initializer list**

- Originally put in place for use in inheritance
- Occurs only in the definition of the constructor
- It is a list of constructor calls that occur after the function argument list (and a colon), but **BEFORE** the opening braces of the constructor body

const in Classes

```
#include <iostream>

using namespace std;

class Fred {
    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();
}
```

Variable Initialization

```
int a = 5; /* Regular */  
int b(6); /* constructor init */  
int c{9}; /* uniform init */
```

- Uniform init requires new C++ standard
- This makes it easier to differentiate because {} and () – functional form
- This will make more sense when we start talking about classes and objects (and constructors)

const in Classes

To answer the original question - what is the proper way of **#define** constant value inside a class?

- Use the `static` keyword in addition to `const` - means only **one instance** of this data member, regardless of how many objects are created
- Similar to static variables in functions - they exist across multiple function calls, which can be seen as only **one copy** existing across all function calls
- `static const` of a built-in type is treated as a compile-time constant

const in Classes

```
class StringStack {  
    static const int size = 100;  
    const string* stack[size];  
    int index;  
public:  
    StringStack();  
    void push(const string* s);  
    const string* pop();  
};
```

const Objects and Member Functions

const objects are created similarly to those of built-in types

```
const int i = 1;
```

```
const blob b(2);
```

Since blob is a const, no member of the blob object must change during its lifetime

- How do we know which member functions can be executed safely (i.e., and not change its data)?
- Declare a member function also `const`, and the compiler knows this function can be called safely

const Objects and Member Functions

```
class X {  
    int i;  
public:  
    X(int ii);  
    int f() const;  
};
```

```
X::X(int ii) : i(ii) {}
```

```
int X::f() const { return i; } // reiterate const at def  
// error if you try to change any member of the object  
// or call another non-const member function
```

```
int main() {  
    X x1(10);  
    const X x2(20);  
    x1.f();  
    x2.f();  
}
```

const Objects and Member Functions

Constructors and destructors are not `const` (even for a `const` object), since it almost always changes things

But these are still called, since they are only called at initialization and at end of scope/life

volatile

As before, these are variables that can change by external forces (i.e., the compiler does not know)

Even if you read a variable `x` at time A and not change `x` between time A and time B, you cannot assume the data has not changed at time B.

Therefore, you cannot optimize away the read at point B and reuse the value in `x`

You can make objects `volatile`

You can also create `const volatile` - you can't change its value but it may still change (by external forces)

Only `volatile` functions can be called by a `volatile` object (just as with `const` objects)

Questions

References

- Pointer overview – C allows void pointers (void*)
 1. `bird* b;`
 2. `rock* r;`
 3. `void* v;`
 4. `v = r;`
 5. `b = v;`
- C++ does **NOT** allow this because it is a strongly typed language (i.e., stronger type rules at compile time)

C++ Reference

'&' is like a **constant pointer** that is **automatically dereferenced**

```
1. int y = 13;
2. int& r = y;
3. cout << "y = " << y << endl;
4. cout << "r = " << r << endl;
5. const int& q = 12;
6. int x = 0;
7. int& a = x;
8. cout << "x = " << x << ", a = " << a << endl;
9. a++; x++;
10. cout << "x = " << x << ", a = " << a << endl;
```

← r is a "pointer" to y

but you can use it like a regular variable

References must be initialized at definition

When **references** are created, they **must** be initialized

C++ Reference

```
y = 13  
r = 13  
x = 0, a = 0  
x = 2, a = 2
```

C++ Reference

- A reference **must be initialized** when it is created (pointers can be initialized any time)
- Once initialized, it **cannot** refer to another object (pointers can)
- You cannot have a NULL reference (must be connected to **real storage**)
- References are like fancy pointers that you never have to wonder about initialization (it won't compile if not initialized) and how to deference it (compiler does it automatically).

Example

Will this work?
If so, what would it do?

```
1.  int y = 13;
2.  int &r = y;
3.  int x = 0;
4.  int &a = x;
5.  a++; x++;
6.  a = y;
7.  a++;
8.  cout << "x = " << x << ", a = " << a << endl;
9.  cout << "y = " << y << ", r = " << r << endl;
```

Example

$X = 14, a = 14$
 $Y = 13, r = 13$

References in Functions

- Acts as pass by reference without using pointers

```
1.  int* f(int *x)
2.  {
3.      *x = *x + 1;
4.      return x;
5.  }
6.  int& g(int &x)
7.  {
8.      x++;
9.      return x;
10. }
11. int a = 0;
12. int* b = f(&a);
13. cout << a << endl;
14. a = g(a);
15. cout << a << endl;
```

Will this work?
If so, what would it do?

References in Functions

1
2

Pointer References

- In C, if you want to change the content of a pointer (vs. what it points to), you need to use a double pointer (i.e., `int**`)

```
1.  void dp(int **j)
2.  {
3.      int *x = (int*) malloc(sizeof(int));
4.      x[0] = 100;
5.      *j = x;
6.  }
7.  int i = 47;
8.  int *j = &i;
9.  cout << *j << endl;
10. dp(&j);
11. cout << *j << endl;
```

Pointer References

- However, if you use references

```
1.  void ref(int*& k)
2.  {
3.      int *x = (int*) malloc(sizeof(int));
4.      x[0] = 200;
5.      k = x;
6.  }
7.  int i = 47;
8.  int *k = &i;
9.  cout << *k << endl;
10. ref(k);
11. cout << *k << endl;
```

References

- Provides a **cleaner** method of pass-by-reference
- However, it's **less explicit** than pointers (so it could lead to confusion if you're used to C pointers).
- You also don't know if a function will change the value of a variable passed in (if you don't read the header definition carefully).

Argument Passing Guideline

- In C++, passing argument to a function should be done as **const reference**
 - If not, what would happen if you pass an integer value to a function that accepts int?

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

```
int main() {  
    f(1); // Error  
    g(1);  
} ///:~
```

Argument Passing Guideline

- In C++, passing argument to a function should be done as **const reference**
 - If not, what would happen if you pass an integer value to a function that accepts int?
- Passing by reference is more **efficient**, because you're not making a copy of the argument (which can be expensive for large classes) in terms of storage and construction/destruction (whenever you create an object, you call the constructor and destructor)
 - However, if they are not meant to be changed, they should be passed as **const**
- Only exception might be when you might change the object in a destructive manner within the function, so you want to send just a copy of it

My Advice

Use pointers to make it less ambiguous that you are passing in an address

- But you still have to know this because others will be using references in their code

You can prevent people from modifying your data (that has been passed using a pointer) by declaring the argument as `const`

```
1. void ff(const int *x)
2. {
3.     cout << x[0] << endl;
4.     x[0] = 100;
5. }
6. void testSix()
7. {
8.     int *x = new int[10];
9.     ff(x);
10. }
```

Allowed

Not allowed

Questions?

Copy Constructor

- Constructor for **passing** and returning **user-defined types by value** (during function calls)
- Compiler will automatically create one if you don't provide one yourself (like the default constructor), but it's typically better to create it yourself for efficiency and correctness
- For simple data types, the constructor does bit-by-bit copy

Example

```
1. class HowMany {
2. private:
3.     static int objectCount;
4. public:
5.     HowMany() { objectCount++; }
6.     static void print(const string& msg = "") {
7.         if(msg.size() != 0) {
8.             cout << msg << ": ";
9.         }
10.        cout << "objectCount = "
11.            << objectCount << endl;
12.    }
13.    ~HowMany() {
14.        objectCount--;
15.        print( "~HowMany()" );
16.    }
17. };

18. int HowMany::objectCount = 0;
19.
20. HowMany f(HowMany x) {
21.     x.print("x argument inside
22.         f()");
23.     return x;
24. }
25. void testFour()
26. {
27.     HowMany h1;
28.     HowMany::print( "after construction
29.         of h1" );
29.     HowMany h2 = f(h1);
30.     HowMany::print( "after f()" );
31. }
```

Why?

after construction of h1: `objectCount = 1`

x argument inside `f()`: `objectCount = 1`

`~HowMany(): objectCount = 0`

after `f()`: `objectCount = 0`

`~HowMany(): objectCount = -1`

`~HowMany(): objectCount = -2`

Why

```
18. HowMany f (HowMany x) {  
19.     x.print("x argument inside  
20.         f()");  
21.     return x;  
22. }  
23. void testFour()  
24. {  
25.     HowMany h1;  
26.     HowMany::print("after  
27.         construction of h1");  
28.     HowMany h2 = f(h1);  
29.     HowMany::print("after f()");  
30. }
```

h1 exists
(object count = 1)

same for h2

copy of h1 exists
inside f()

copy of h1 is
destroyed when
function ends (1st
destroy)

h1 and h2 are
destroyed (2nd and 3rd destroy)

Why?

- Copy of `h1` in the function `f()` is a bit-by-bit copy (i.e., **constructor was not called**), which is why `objectCount` was **not** incremented (via default copy-constructor)
- Same goes for `h2` (i.e., `h2` was a bit-by-bit copy of the return value of `f()`, which was a bit-by-bit copy of `h1`)
- Then copy of `h1`, `h1`, and `h2` are destroyed (decremented 3 times)
- This problem happens because the compiler makes certain assumptions about how to create a new object from an existing object (i.e., a copy)

Copy Constructor

```
1. class HowMany2 {
2. private:
3.     string name;
4.     static int objectCount;
5. public:
6.     HowMany2(const string& id = "") : name(id) {
7.         ++objectCount;
8.         print("HowMany2()");
9.     }
10.    ~HowMany2() {
11.        --objectCount;
12.        print("~HowMany2()");
13.    }:
14.    HowMany2(const HowMany2& h) : name(h.name) {
15.        name += " copy";
16.        ++objectCount;
17.        print("HowMany2(const HowMany2&)");
18.    }

19.     void print(const string& msg = "")
20.     const {
21.         if(msg.size() != 0) {
22.             cout << msg << endl;
23.         }
24.         cout << '\t' << name << ": "
25.             << "objectCount = "
26.             << objectCount << endl;
27.     };

28. HowMany2 f(HowMany2 x) {
29.     x.print("x argument inside f()");
30.     out << "Returning from f()" << endl;
31.     return x;
32. }
```

Copy Constructor

- `HowMany2 h1("h1");`
- `cout << "Entering f()" << endl;`
- `HowMany2 h2 = f(h1);`
- `h2.print("h2 after call to f()");`

- `HowMany2()`
- `h1: objectCount = 1`
- Entering `f()`
- `HowMany2(const HowMany2&)`
- `h1 copy: objectCount = 2`
- `x` argument inside `f()`
- `h1 copy: objectCount = 2`
- Returning from `f()`
- `HowMany2(const HowMany2&)`
- `h1 copy copy: objectCount = 3`
- `~HowMany2()`
- `h1 copy: objectCount = 2`
- `h2` after call to `f()`
- `h1 copy copy: objectCount = 2`

Copy Constructor

- `HowMany2 h1("h1");`
- `cout << "Entering f()" << endl;`
- `HowMany2 h2 = f(h1);`
- `h2.print("h2 after call to f()");`

- `HowMany2()`
- `h1: objectCount = 1`
- Entering `f()`
- `HowMany2(const HowMany2&)`
- `h1 copy: objectCount = 2`
- `x` argument inside `f()`
- `h1 copy: objectCount = 2`
- Returning from `f()`
- `HowMany2(const HowMany2&)`
- `h1 copy copy: objectCount = 3`
- `~HowMany2()`
- `h1 copy: objectCount = 2`
- `h2` after call to `f()`
- `h1 copy copy: objectCount = 2`


Copy Constructor

- `HowMany2 h1("h1");`
- `cout << "Entering f()" << endl;`
- `HowMany2 h2 = f(h1);`
- `h2.print("h2 after call to f()");`

- `HowMany2()`
- `h1: objectCount = 1`
- `Entering f()`
- `HowMany2(const HowMany2&)`
- `h1 copy: objectCount = 2`
- `x argument inside f()`
- `h1 copy: objectCount = 2`
- `Returning from f()`
- `HowMany2(const HowMany2&)`
- `h1 copy copy: objectCount = 3`
- `~HowMany2()`
- `h1 copy: objectCount = 2`
- `h2 after call to f()`
- `h1 copy copy: objectCount = 2`

Copy Constructor

Creates a copy using
the
copy-constructor



- `HowMany2 h1("h1");`
- `cout << "Entering f()" << endl;`
- `HowMany2 h2 = f(h1);`
- `h2.print("h2 after call to f()");`

- `HowMany2()`
- `h1: objectCount = 1`
- Entering `f()`
- `HowMany2(const HowMany2&)`
- `h1 copy: objectCount = 2`
- **x argument inside `f()`**
- `h1 copy: objectCount = 2`
- Returning from `f()`
- `HowMany2(const HowMany2&)`
- `h1 copy copy: objectCount = 3`
- `~HowMany2()`
- `h1 copy: objectCount = 2`
- `h2 after call to f()`
- `h1 copy copy: objectCount = 2`

Copy Constructor

- `HowMany2 h1("h1");`
- `cout << "Entering f()" << endl;`
- `HowMany2 h2 = f(h1);`
- `h2.print("h2 after call to f()");`

Return value is
created



- `HowMany2()`
- `h1: objectCount = 1`
- `Entering f()`
- `HowMany2(const HowMany2&)`
- `h1 copy: objectCount = 2`
- `x argument inside f()`
- `h1 copy: objectCount = 2`
- `Returning from f()`
- `HowMany2(const HowMany2&)`
- `h1 copy copy: objectCount = 3`
- `~HowMany2()`
- `h1 copy: objectCount = 2`
- `h2 after call to f()`
- `h1 copy copy: objectCount = 2`

Copy Constructor

- `HowMany2 h1("h1");`
- `cout << "Entering f()" << endl;`
- `HowMany2 h2 = f(h1);`
- `h2.print("h2 after call to f()");`

`h1 copy` is destroyed at the end of the function `f()`

- `HowMany2()`
- `h1: objectCount = 1`
- Entering `f()`
- `HowMany2(const HowMany2&)`
- `h1 copy: objectCount = 2`
- `x` argument inside `f()`
- `h1 copy: objectCount = 2`
- Returning from `f()`
- `HowMany2(const HowMany2&)`
- `h1 copy copy: objectCount = 3`
- `~HowMany2()`
- `h1 copy: objectCount = 2`
- `h2` after call to `f()`
- `h1 copy copy: objectCount = 2`

Copy Constructor

- `HowMany2 h1("h1");`
- `cout << "Entering f()" << endl;`
- `HowMany2 h2 = f(h1);`
- `h2.print("h2 after call to f()");`

- `HowMany2()`
- `h1: objectCount = 1`
- `Entering f()`
- `HowMany2(const HowMany2&)`
- `h1 copy: objectCount = 2`
- `x argument inside f()`
- `h1 copy: objectCount = 2`
- `Returning from f()`
- `HowMany2(const HowMany2&)`
- `h1 copy copy: objectCount = 3`
- `~HowMany2()`
- `h1 copy: objectCount = 2`
- `h2 after call to f()`
- `h1 copy copy: objectCount = 2`

Copy Constructor

h2 is destroyed



- `~HowMany2()`

h1 is destroyed



- `h1 copy copy: objectCount = 1`
- `~HowMany2()`
- `h1: objectCount = 0`

Copy Constructor for Composite Class

- When a new class is created that uses other classes, the compiler creates a default copy constructor using the copy-constructors for those member classes
- If a member class does not have a copy-constructor, it will use the default copy-constructor for that class (i.e., bit-by-bit copy)

Alternatives to Copy Constructors

Do not pass by value (copy-constructor is called only when you pass-by-value, or make a copy)

You can **prevent** pass by value by creating a **private member copy-constructor** – it does not even have to be defined, just declaring one is sufficient

```
1. class NoCC {
2.     int i;
3.     NoCC(const NoCC&) ;
4. public:
5.     NoCC(int ii = 0) : i(ii) {}
6. };
7. void f(NoCC);
8. int main() {
9.     NoCC n;
10.    //! f(n); // Error: copy-constructor called
11.    //! NoCC n2 = n; // Error: c-c called
12.    //! NoCC n3(n); // Error: c-c called
13. }
```


Questions?