Ch. 11: References & the Copy-Constructor
- continued -
Remember returning objects by value from a function
The address of B2 is pushed before making the call.
The function copies the returned object at that address before returning.
What can go wrong when copying?
```cpp
class HowMany {
    static int objectCount;

public:
    HowMany() { objectCount++; }
    static void print(const string& msg = "") {
        if(msg.size() != 0)
            cout << msg << " : ";
        cout << "objectCount = " << objectCount << endl;
    }
    ~HowMany() {
        objectCount--;
        print("~HowMany()");
    }
};

int HowMany::objectCount = 0;
```
When the value was returned in h2, the bit-copy mechanism did not increment objectCount, but when h2 went out of scope, the destructor did decrement it!
Extra-credit
Conclusion

• The low-level, bit-by-bit copying doesn’t always work correctly when combined with C++ constructors.

• We call this **bitcopy**, to distinguish it from **copy-construction** (next).
Copy-construction

The problem occurs because the compiler makes an assumption about how to create a new object from an existing object. When you pass an object by value, you create a new object, the passed object inside the function frame, from an existing object, the original object outside the function frame. This is also often true when returning an object from a function. In the expression

    HowMany h2 = f(h);

h2, a previously unconstructured object, is created from the return value of f(), so again a new object is created from an existing one.
[...] intervene in this process and prevent the compiler from doing a bitcopy. You do this by defining your own function to be used whenever the compiler needs to make a new object from an existing object:

- We’re making a new object, so this function is a **constructor**.
- The constructor has a single argument: the object we’re constructing from.
- The argument is passed **by (constant) reference**!
In general: \( X(\text{const } X&) \)

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

This function is called whenever a new object is created from an existing one.

```cpp
HowMany2 h2 = f(h);
```
16. Write a class `CCClass` with:

- An integer data member
- A constructor that initializes the data member
- A copy-constructor that simply announces itself to `cout`.
class CCClass{
    int a;
public:
    CCClass(int);
    CCClass(const CCClass &);
};

CCClass::CCClass(int x) : a(x) {}
CCClass::CCClass(const CCClass &c) {
    std::cout << "Copy-constructor called!" << std::endl;
}
QUIZ

16. Now create:

• a function that passes a **CCClass** object in by value and does not return anything

• another function that does not take any arguments, creates a local **CCClass** object, and returns it by value.
void foo(CCClass){};

CCCClass bar(){
    CCClass c(43);
    return c;
};
16. Call these functions in the main program.
void foo(CCClass){};

CCClass bar(){
    CCClass c(43);
    return c;
};

int main(){
    CCClass c2(42);
    foo(c2);
    CCClass c3 = bar();
}
void foo(CCClass){}

CCClass bar(){
    CCClass c(43);
    return c;
}

int main(){
    CCClass c2(42);
    foo(c2);

    CCClass c3 = bar();
}

Copy-constructor called!
Copy-constructor called!
Press any key to continue...
What if we call the function, but ignore the return value? A: The compiler creates a temporary, so the C.C. is still called twice!

```cpp
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);  // Highlighted line
    out << "After call to f()" << endl;
} ///:~
```
Default copy-constructor

Because the copy-constructor implements pass and return by value, it’s important that the compiler creates one for you in the case of simple structures – effectively, the same thing it does in C. However, all you’ve seen so far is the default primitive behavior: a **bitcopy**.

When more complex types are involved, the C++ compiler will still automatically create a copy-constructor if you don’t make one. Again, however, a **bitcopy** doesn’t make sense, because it doesn’t necessarily implement the proper meaning.
Default copy-constructor

Suppose you create a new class composed of objects of several existing classes. This is called, appropriately enough, *composition*, and it’s one of the ways you can make new classes from existing classes.

Remember *composition* from Ch.1!
Ch. 1: Introduction to Objects
Reusing the implementation

Composition

Owner class

Component class

Car

Engine

0..1

1..1

UML diagram!
Composition example (not in text)

```java
class Engine{
    char manufacturerID[20];
    unsigned year;
};

class Frame{
    int color;
};

class Car{
    Engine myCarsEngine;
    Frame myCarsFrame;
};
```
//: C11: DefaultCopyConstructor.cpp
// Automatic creation of the copy-constructor
#include <iostream>
#include <string>
using namespace std;

class WithCC {  // With copy-constructor
    public:
    // Explicit default constructor required:
    WithCC() {}  
    WithCC(const WithCC&) {
        cout << "WithCC(WithCC&)" << endl;
    }
};
class WoCC { // Without copy-constructor
  string id;
public:
  WoCC(const string& ident = "") : id(ident) {}  // const
  void print(const string& msg = "") {
    if(msg.size() != 0) cout << msg << ": ";
    cout << id << endl;
  }
};

Do you remember what this means?
class Composite {
    WithCC withcc; // Embedded objects
    WoCC wocc;

public:
    Composite() : wocc("Composite()") {}  
    void print(const string& msg = "") const {
        wocc.print(msg);
    }
};

Draw the UML class diagram for this program!
class Composite {
    WithCC withcc; // Embedded objects
    WoCC wocc;
public:
    Composite() : wocc("Composite()") {} 
    void print(const string& msg = "") const {
        wocc.print(msg);
    }
};

Diagram:

- **Composite**
  - default constr.
  - print()

- **With CC**
  - default constr.
  - copy constr.

- **WoCC**
  - constr.
  - print()
```cpp
int main() {
    Composite c;
    c.print("Contents of c");
    cout << "Calling Composite copy-constructor"
         << endl;
    Composite c2 = c;  // Calls copy-constructor
    c2.print("Contents of c2");
} ///:~
```

```
Contents of c: Composite()
Calling Composite copy-constructor
WithCC(WithCC&)
Contents of c2: Composite()
```
To create a copy-constructor for a class that uses composition (and inheritance, which is introduced in Chapter 14), the compiler recursively calls the copy-constructors for all the member objects (and base classes, for inheritance).

The process the compiler goes through to synthesize a copy-constructor is called memberwise initialization.
To do for next time:

• Read the subsections:
  – Preventing pass-by-value
  – Functions that modify outside objects

• Re-read the entire section **Copy-constructor**.

• Solve end-of-chapter ex. 17.
QUIZ: What happens when the code passes or returns an object **by value** and its class does not have a copy-constructor?
To create a copy-constructor for a class that uses composition (and inheritance, which is introduced in Chapter 14), the compiler recursively calls the copy-constructors for all the member objects (and base classes, for inheritance).
#include <iostream>
using namespace std;

class Point
{
    int x, y;
public:
    Point(const Point &p) { x = p.x; y = p.y; }
};

int main()
{
    Point p1;
    Point p2 = p1;
    return 0;
}
#include <iostream>
using namespace std;

class Point
{
    int x, y;
public:
    Point(const Point &p) { x = p.x; y = p.y; }
};

int main()
{
    Point p1;
    Point p2 = p1;
    return 0;
}

No – if any constructor exists (even only a copy-constructor), the compiler does not create a default constructor!
How about the opposite? Here we have a constructor, but no copy-constructor!
How about the opposite? Here we have a constructor, but no copy-constructor!

What happened here, exactly?
QUIZ: What is memberwise initialization in C++?
The (recursive) process the compiler goes through to synthesize a copy-constructor is called **memberwise initialization**.
Preventing pass-by-value

There’s a simple technique for preventing pass-by-value: declare a private copy-constructor.

If the user tries to pass or return the object by value, the compiler will produce an error message because the copy-constructor is private. It can no longer create a default copy-constructor because you’ve explicitly stated that you’re taking over that job.
There’s a simple technique for preventing pass-by-value: declare a private copy-constructor.

If the user tries to pass or return the object by value, the compiler will produce an error message because the copy-constructor is private. It can no longer create a default copy-constructor because you’ve explicitly stated that you’re taking over that job.
"Deleted" functions were introduced in C++11

```cpp
class NoCC{
    int a;
public:
    NoCC(int x){a=x;};
    NoCC(const NoCC&) = delete;
};

int main(){
    NoCC x1(42);
    NoCC x2=x1;
}
```

prog.cpp:12:11: error: use of deleted function 'NoCC::NoCC(const NoCC&)'
```cpp
    NoCC x2=x1;
    ^
```
A program that refers to a deleted function implicitly or explicitly, other than to declare it, is ill-formed.
QUIZ: Does this program compile?

class NoCC{
  int a;
  public:
    NoCC(int x){a=x;};
    NoCC(const NoCC&) = delete;
};

void foo(NoCC) {cout <<"42!";}

int main(){
  NoCC x1(42);
  foo(x1);
}
QUIZ: Does this program compile?

class NoCC{
    int a;
    public:
        NoCC(int x){a=x;};
        NoCC(const NoCC&) = delete;
    
    void foo(NoCC) {cout <<"42!";}

    int main(){
        NoCC x1(42);
        foo(x1);
        foo(x1);
    }

    prog.cpp:15:9: error: use of deleted function 'NoCC::NoCC(const NoCC&)'
    foo(x1);
    
    ^
Pointers to members

A pointer is a variable that holds the address of some location. You can change what a pointer selects at runtime, and the destination of the pointer can be either data or a function.

The C++ *pointer-to-member* follows this same concept, except that what it selects is a location inside a class.
Pointers to members

The dilemma here is that a pointer needs an address, but there is no “address” inside a class; selecting a member of a class means **offsetting into that class**.

You can’t produce an actual address until you combine that offset with the starting address of a particular object.
Remember how we access members of classes and structs:

```cpp
//: C11:SimpleStructure.cpp
struct Simple { int a; };  
int main() {
    Simple so, *sp = &so;
    sp->a;
    so.a;
} ///:~
```
Remember how we declare and initialize ordinary pointers:

\[
\text{int } a = 42; \\
\text{int } *aPtr; \\
aPtr = \&a;
\]

or

\[
\text{int } a = 42; \\
\text{int } *aPtr = \&a;
\]
Let’s say that we have a class with multiple members of the same type ... 

```c++
class Data {
public:
    int a, b, c;
    void print() const {
        cout << "a = " << a << " , b = " << b << " , c = " << c << endl;
    }
};
```

... How do we create a pointer that can point to those members?
```cpp
int main() {
    Data d, *dp = &d;
    int Data::*pmInt = &Data::a;
    dp->*pmInt = 47;
    pmInt = &Data::b;
    d.*pmInt = 48;
    pmInt = &Data::c;
    dp->*pmInt = 49;
    dp->print();
} ///:~
```
24. Create a class containing a `double` and a `print()` function that prints the `double`. Don’t forget the constructor!
class Foo {
public:
    const double d;
    Foo::Foo(double x):d(x){}
    void print() { cout <<d <<endl; }
};
24. In **main()**, create a pointer to the data member.
class Foo {
public:
    const double d;
    Foo::Foo(double x):d(x){}
    void print() { cout <<d <<endl; }
};

int main() {
    const double Foo::*dPtr = &Foo::d;
}
QUIZ

24. In main, create an object of your class and a pointer to that object.
class Foo {
public:
    const double d;
    Foo::Foo(double x):d(x){}
    void print() { cout <<d <<endl; }
};

int main() {
    const double Foo::*dPtr = &Foo::d;
    Foo myFoo(42);
24. Manipulate the data member of the object, using the pointer inside the object.
class Foo {
public:
    const double d;
    Foo::Foo(double x):d(x){}
    void print() { cout << d << endl; }
};

int main() {
    const double Foo::*dPtr = &Foo::*d;
    Foo myFoo(42);
    cout << myFoo.d << endl;
    cout << myFoo.*dPtr << endl;
}

QUIZ

24. Create a pointer to the object, and the manipulate the data member of the object, using the pointer to the object and the pointer inside the object.
```cpp
#include <iostream>

class Foo {
public:
    const double d;
    Foo(double x) : d(x) {}
    void print() { cout << d << endl; }
};

int main() {
    const double Foo::*dPtr = &Foo::d;
    Foo myFoo(42);

    cout << myFoo.d << endl;
    cout << myFoo.*dPtr << endl;
    cout << myFooPtr->d << endl;
    cout << myFooPtr->*dPtr << endl;
}
```
Pointers to members are quite limited: they can be assigned only to a specific location inside a class.

We cannot increment or compare them like ordinary pointers.
SKIP the last section of ch.11:

Pointers to member functions
Homework for ch. 11

Provided as separate handout (also available on our webpage --\> agapie.net)

Due Monday, April 12, at the beginning of class.

Please hand in a hard-copy, do not email!