Ch. 11: References & the Copy-Constructor

- continued -
When a reference is made const, it means that the object it refers cannot be changed through that reference - it may be changed in a different way (unless the object itself is constant)
const references

```cpp
int a = 1;
int &b = a;
const int &c = a;

b = 2; // OK

C = 3; // illegal
cout << C; // OK
```
const references for return values

What is the difference between these two versions of \texttt{g}?

\begin{verbatim}
int& \texttt{g}(\texttt{int}& \texttt{x}) \{ \\
    \texttt{x}++; \\
    \texttt{return} \texttt{x}; \\
\}
\end{verbatim}

\begin{verbatim}
\texttt{const} int& \texttt{g}(\texttt{int}& \texttt{x}) \{ \\
    \texttt{x}++; \\
    \texttt{return} \texttt{x}; \\
\}
\end{verbatim}

Remember ch.8: Using the return value of a function as \texttt{lvalue} ...
int& g(int& x) {
    x++;
    return x;
}

const int& g(int& x) {
    x++;
    return x;
}

int a = 1;
g(a) = 42;
const references for arguments

If you know the function will respect the constness of an object, making the argument a const reference will allow the function to be used in all situations.

This means that:

• for built-in types, the function will not modify the argument

• for user-defined types, the function will call only const member functions, and won’t modify any public data members.
**QUIZ**

One of the function calls is incorrect. Which one and why?
One of the function calls is incorrect. Which one and why?
Remember from C:

pointer-to-pointer, a.k.a. second-level pointer

```c
int i = 42;
int *ip = &i;
int **ipp = &ip;
```
QUIZ: What does this program print?

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

int main () {
    int a;
    int *aptr;
    int **apptr;

    a = 42;
aptr = &a;
apptr = &aptr;

    cout << a << endl;
cout << *aptr << endl;
cout << **apptr << endl;
}
```
#include <iostream>
using namespace std;

int main () {
    int a;
    int *aptr;
    int **apptr;

    a = 42;
    aptr = &a;
    apptr = &aptr;

    cout << a << endl;
    cout << *aptr << endl;
    cout << **apptr << endl;
}

42
42
42
Solution

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

int main () {
    int  a;
    int *aptr;
    int **apptr;

    a = 42;
    aptr = &a;
    apptr = &aptr;

    cout << a << endl;
    cout << *aptr << endl;
    cout << **apptr << endl;
}
```

It is possible to pass a pointer-to-pointer as argument …

... but in C++ we can pass a **pointer reference** (reference to a pointer) instead!

```cpp
void f(int **);
int i = 42;
int *ip = &i;
f(&ip);

void f(int * &);
int i = 42;
int *ip = &i;
f(ip);
```
In C++, we can pass a pointer reference (reference to a pointer):

```cpp
// C11:ReferenceToPointer.cpp
#include <iostream>
using namespace std;

void increment(int*& i) { i++; }

int main() {
    int* i = 0;
    cout << "i = " << i << endl;
    increment(i);
    cout << "i = " << i << endl;
} ///:~
```

Make sure you grok this pointer arithmetic!
The copy-constructor

Passing & returning by value
a.k.a.
A little bit of assembly language and computer architecture
The definition of function f is elsewhere

We’re showing the (simplified) assembly code that implements this line

```
push  b
push  a
call  f()
add   sp,4
mov   g, register a
```

Arguments pushed on stack from right to left

Beginning address of f’s code is loaded in Program Counter.

When f() - code returns, execution continues here.
In cdecl convention, the caller is responsible for “cleaning up”, i.e. restoring the stack. It depends on the sizes for `int` and `char` on this machine!

This means 4 Bytes removed from the stack. It depends on the sizes for `int` and `char` on this machine!

In cdecl convention, the caller is responsible for “cleaning up”, i.e. restoring the stack.

Do not confuse with the variable `a`. Functions return values in a specific register, known to the compiler, e.g. in an Intel CPU this is EAX.
```c
int f(int x, char c);
int g = f(a, b);
```

- push b
- push a
- call f()
- add sp, 4
- mov g, register a

---

**Address** | **Contents**
--- | ---
| Stack before f() is called | b
| a |
The complete picture: Function-call stack frame

```c
int f(int x, char c);
int g = f(a, b);
```

```assembly
push b
push a
call f()
add sp, 4
mov g, register a
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Stack</td>
</tr>
<tr>
<td></td>
<td>before</td>
</tr>
<tr>
<td></td>
<td>f() is</td>
</tr>
<tr>
<td></td>
<td>called</td>
</tr>
<tr>
<td></td>
<td>b</td>
</tr>
<tr>
<td></td>
<td>a</td>
</tr>
<tr>
<td></td>
<td>Return</td>
</tr>
<tr>
<td></td>
<td>address</td>
</tr>
<tr>
<td></td>
<td>Local</td>
</tr>
<tr>
<td></td>
<td>variables</td>
</tr>
</tbody>
</table>

Function arguments
- Return address
- Local variables
How does this change when passing a (large) user-defined object by value?

```c
int f(int x, char c);
int g = f(a, b);
```

```
push b
push a
call f()
add sp, 4
mov g, register a
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Stack before f() is called</td>
</tr>
<tr>
<td></td>
<td>b</td>
</tr>
<tr>
<td></td>
<td>a</td>
</tr>
<tr>
<td></td>
<td>Return address</td>
</tr>
<tr>
<td></td>
<td>Local variables</td>
</tr>
</tbody>
</table>

- Function arguments
- Return address
- Local variables
struct Big {
  char buf[100];
  int i;
  long d;
} B, B2;

Big bigfun(Big b) {
  b.i = 100;  // Do something to the argument
  return b;
}

int main() {
  B2 = bigfun(B);
}  //://:
The question is: How to return the result? (There aren’t enough registers!)

Array of 100 chars!
// C11: PassingBigStructures.cpp

struct Big {
    char buf[100];
    int i;
    long d;
} B, B2;

Big bigfun(Big b) {
    b.i = 100; // Do something to the argument
    return b;
}

int main() {
    B2 = bigfun(B);
} ///:~

One possible solution is to also push the return values on the stack:
To understand why pushing return values on the stack is not good, we need to understand the constraints on the compiler when it’s making a function call.

```cpp
//: C11:PassingBigStructures.cpp
struct Big {
    char buf[100];
    int i;
    long d;
} B, B2;

Big bigfun(Big b) {
    b.i = 100; // Do something to the argument
    return b;
}

int main() {
    B2 = bigfun(B);
} //://:
```
The compiler must use a solution that works with:

- Interrupts and ISRs = Interrupt Service Routines (functions, really!)
- Recursive functions
Interrupts

Consider the following scenario:

• Function pushes return values on stack and returns normally

• Before the calling function has a chance to process the return values, an interrupt occurs, and the ISR overwrites them!

ISRs are “smart” enough to save all registers on the stack, and restore them upon exit, but they have their limits.
Re-entrancy

Would placing the return values on the heap work?

Why not? 😊
Historical note:
In K&R C, it was not possible to return structs, only pointers to structs.
Since C89, C functions can return structs, although many compilers don’t actually place the struct on the stack – they do return values optimization instead (where only a pointer is returned, e.g. in register eax).

This means that the caller is responsible for setting up space for return values, contrary to C custom!
The solution chosen in C++ is this:

- Push the address of the return value’s destination on the stack as a *hidden* argument.
- Let the function copy the return value into the destination before returning.

```cpp
//: C11:PassingBigStructures.cpp
struct Big {
    char buf[100];
    int i;
    long d;
} B, B2;

Big bigfun(Big b) {
    b.i = 100; // Do something to the argument
    return b;
}

int main() {
    B2 = bigfun(B);
} ///:~
```
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 unconstrcted 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!
This function is called automatically by the compiler whenever a new object is created from an existing one.
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;
}
16. Now create:

- a function that is passed 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){};

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

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

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

How many times is the constructor called?
How many times the copy-constructor?
void foo(CCClass){};

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

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

    CCCClass 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);
    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, 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

UML diagram!
/ *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) {}
    void print(const string& msg = ")" const {
        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);
    }
};
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()
Conclusion

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.
```cpp
#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;
}
```
Is this program correct?

#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!

main.cpp:13:11: error: no matching function for call to 'Point::Point()'
    Point p1;
    ^
#include <iostream>
using namespace std;

class Point 
{
    int x, y;
public:
    Point(int i, int j) { x = 10; y = 20; }
    int getX() { return x; }
    int getY() { return y; }
};

int main()
{
    Point p1(10, 20);
    Point p2 = p1;
    cout << "x = " << p2.getX() << " y = " << p2.getY();
    return 0;
}
How about the opposite? Here we have a constructor, but no copy-constructor!

A default copy-constructor was synthesized by the compiler!
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.*
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.
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&)'
    NoCC x2=x1;
        ^
A program that refers to a deleted function implicitly or explicitly, other than to declare it, is ill-formed.
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);
    ^
Read FYI:
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++ \textit{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:

```
int a = 42;
int *aPtr;
aPtr = &a;
```

or

```
int a = 42;
int *aPtr = &a;
```
Let’s say that we have a class with multiple members of the same type ...

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?
How to read this from R to L ...

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

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(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;
}
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;
}
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.
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;
    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 Friday, Apr. 13, at the beginning of class.

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