Copy Semantics

Sandra Batista, Mark Redekopp, and David Kempe
SHALLOW COPY VS. DEEP COPY
Old School Memory: Shallow Copy

Old School Memory: Deep Copy

```cpp
int main()
{
    int *p = new int{77};
    int *q = new int{*p};
    *p = 88;
}
```

Unique Pointer: NO Shallow Copy!

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

int main()
{
    unique_ptr<int> p = make_unique<int>(77);
    unique_ptr<int> q = p; // Will not compile
    *p = 88;
}
```

No copying or assigning unique pointers:

Only one unique pointer may manage one physical address!
#include <memory>
using namespace std;

int main()
{
    unique_ptr<int> p = make_unique<int>(77);
    unique_ptr<int> q = make_unique<int>(*p);
    *p = 88;
}
Struct/Class Assignment

By default assigning or copying a struct or class object to another of the same type performs **shallow copy**.

This is element by element copy of the source struct/class to the destination struct/class.

```cpp
#include<iostream>
using namespace std;
enum {CS, CECS};
struct student {
    char name[80];
    int id;
    int major;
};
int main(int argc, char *argv[])
{
    student s1, s2;
    strncpy(s1.name,"Bill",80);
    s1.id = 5; s1.major = CS;
    s2 = s1;
    return 0;
}
```
COPY CONSTRUCTORS
Copy Constructors

A **copy constructor** is a constructor that makes a new object from an object of the same type.

Most common prototype: 
ClassName(const ClassName& c);

Default copy constructor makes shallow copy.

If deep copy necessary, such a copy constructor must be defined.

class Complex
{
    public:
    Complex();
    Complex(double r, double i);
    Complex(const Complex& c);

    private:
    double real, imag;
};

int main()
{
    Complex c1(2,3), c2(4,5)
    Complex c3(c1);
    Complex c4 = c2;
}

When an object is **passed by value**, a copy of the object is made by the copy constructor.

When an object is **returned by value**, a copy is made by the copy constructor.
Default Copy Constructors

C++ compiler automatically generates a **default copy constructor**

- **Simply performs an element by element copy**
- **Provides shallow copy**

```cpp
class Complex
{
public:
    Complex(double r, double i);
    // compiler will provide by default:
    // Complex(const Complex& );
    // Complex& operator=(const Complex&);
    ~Complex()
private:
    double real, imag;
};

int main()
{
    Complex c1(2,3), c2(4,5)
    Complex c3(c1); // copy constructor
    Complex c4 = c1; // copy constructor
}
```
Defining Copy Constructors

Let’s examine the deep copy constructor for Str.

```cpp
#include <memory>
#include <string.h>

class Str {
public:
    Str();
    Str(const Str& other);
    Str(const char* s);
    size_t size() const;
    // other member functions

private:
    std::unique_ptr<char[]> buffer;
    size_t len;
};

Str::Str(const Str& other){
    buffer = std::make_unique<char[]>(other.size()+1);
    len = other.size();
    strcpy(this->buffer.get(), other.buffer.get());
}

int main()
{
    Str s1("hello");
    Str s2(s1); // Str s2 = s1;
}
```
COPY ASSIGNMENT
Copy Assignment

The copy assignment operator, `operator=()`, is called when an object already exists and then another object of the same type is assigned to it.

Prototype:

```
ClassName& operator=(const ClassName& c);
```

C++ compiler automatically generates a default copy assignment operator

- **Simply performs an element by element copy**
- **Only shallow copy**

If deep copy necessary, such a copy assignment operator must be defined.

```cpp
class Complex {
    public:
        Complex(double r, double i);
        // compiler will provide by default:
        // Complex(const Complex& );
        // Complex& operator=(const Complex&);
        ~Complex();
    private:
        double real, imag;
};

int main() {
    Complex c1(2,3), c2(4,5)
    c1 = c2;  // default assignment oper.
    // c1.operator=(c2)
}
```
Copy Assignment Operator Details

RHS should be a const reference

Return value should be a reference

• Allows for chained assignments
• Should return (*this)

```cpp
class Complex {
    public:
        Complex(int r, int i);
        ~Complex();
        Complex operator+(Complex right_op);
    Complex& operator=(const Complex &rhs);
    private:
        int real, imag;
};

Complex& Complex::operator=(const Complex &rhs) {
    real = rhs.real;
    imag = rhs.imag;
    return *this;
}

int main() {
    Complex c1(2,3), c2(4,5);
    Complex c3, c4;
    c4 = c3 = c2;
    // same as c4.operator=( c3.operator=(c2) );
}
```
Assignment Operator Overloading

The = operator can be overloaded with different types

```cpp
class Complex
{
public:
    Complex(int r, int i);
    ~Complex();
    Complex operator+(const Complex &rhs);
    Complex &operator=(const Complex &r);
    Complex &operator=(const int r);
    int real, imag;
};

Complex& Complex::operator=(const int& r)
{
    real = r; imag = 0;
    return *this;
}

int main()
{
    Complex c1(3, 5);
    Complex c2, c3, c4;
    c2 = c4 = 5;
    // c2 = (c4 = 5);
    // c4.operator=(5);  // Complex::operator=(int&)
    // c2.operator=(c4);  // Complex::operator=(Complex&)
    return 0;
}
```
Let's examine the deep copy assignment operator for Str.

```cpp
#include <memory>
#include <string.h>

class Str {
public:
    Str();
    Str(const Str& other);
    Str(const char* s);
    size_t size() const;
    Str& operator=(const Str& rhs);
    // other member functions
private:
    std::unique_ptr<char[]> buffer;
    size_t len;
};

Str& Str::operator=(const Str& rhs){
    if (&rhs == this) return *this;
    buffer = std::make_unique<char[]>(rhs.size()+1);
    len = rhs.size();
    strcpy(this->buffer.get(), rhs.buffer.get());
    return *this;
}

int main()
{
    Str s1("hello");
    Str s2("world");
    s2 = s1;
}
```
Assignment Operator Tips

-*Copy Assignment operator input is const reference of the same object type*

-*Copy Assignment operators should check for initialized members and check for self-assignment*

-*Assignment operators should return a reference type and return *this*
When to Manage Copy Semantics

Default copy constructor and assignment operator ONLY perform SHALLOW copies

• SHALLOW COPY (data members only)
• DEEP copy (data members + what they point at)

You SHOULD define your own copy constructor and assignment operator when a DEEP copy is needed

• When data members are pointers to data that should be copied when a new object is made
• Often if your data members are pointing to dynamically allocated data, you need a DEEP copy

If a Shallow copy is acceptable, you do NOT need to define a copy constructor
Managing Copy Semantics

For shallow copies, a default copy constructor and default assignment operator are sufficient.

**Rule of Three:**
- If memory is dynamically allocated (deep copy needed), implement a *copy constructor*, an *assignment operator*, and a *destructor*.

**Rule of Zero:**
- Whenever possible design your classes so that they do not need to define the default constructor, destructor, copy constructor, and copy assignment.