1 Constructors and Destructors

1.1 Copy Constructor

The copy constructor should copy data. However, it’s not this simple, and we need to make a distinction here between a shallow copy and a deep copy. A shallow copy simply copies over stack variables from one object to another, whereas a deep copy actually copies heap data. In particular, let A be a class with a pointer member variable. A shallow copy of A will simply copy the pointer, so at the end of the copy there will be 2 objects, each pointing to the same data. On the other hand, a deep copy of A will allocate new memory for the new object, assign the memory to a pointer, and then copy the heap data from the original object into the new memory location of the new object. Examples are below

```cpp
struct S {
    S() : ptr(nullptr) {}
    S(const double num) : ptr(new double(num)), num(num) {}
    S(const S& s) : ptr(new double(*s.ptr)), num(s.num) {}
    S(S&& s) : S() {
        std::swap(ptr, s.ptr);
        std::swap(num, s.num);
    }
    ~S() {delete ptr;}
    double* ptr;
    double num;
};

struct C {
    C(const std::string& str, const double val) : dat(std::make_unique<std::string>(str)), S(val) {}
    C(const C& c) : dat(std::make_unique<std::string>(*c.dat)), S(c.s) {}
    C(C&& c) : s(std::move(c.s)), dat(std::move(c.dat)) {}
    std::unique_ptr<std::string> dat;
    S s;
};
```

The struct S above contains a stack variable num and a pointer ptr to heap data. Consider the copy constructor of S on line 4. A shallow copy constructor would look like

```cpp
S(const S& s) : ptr(s.ptr), num(s.num) {}
```

where we simply copy the ptr, which is a stack variable, to the new object. This is very rarely what is wanted in a copy constructor, which is why on line 4 we use a deep copy instead, allocating new memory and assigning to that memory the value from the old memory. The stack variable num is copied normally as it is a stack variable. Consider the copy constructor of C on line 15 now. Even though the member variable s contains dynamically allocated memory, to copy s we do not need to worry about allocation, etc., because the copy constructor of S does this for us. Accordingly, all that is required to construct S in the copy constructor of S is a call to the copy constructor of S. Also on line 15, we must do a deep copy of the unique_ptr member variable dat by allocating new memory: in fact, we are required to do this because unique_ptrs do not have copy constructors.
To summarize:

1. To create a deep copy of an object with heap member data assigned to pointers, allocate new memory and copy the data over.

2. Member variables that are objects should be copied using the copy constructors of the objects (if those objects have copy constructors).

3. Stack data should be copied normally.

1.2 Move constructor

A move constructor should change the ownership of data from one object to another object. Consider the code in the previous section. For the struct S, the move constructor on line 5 should swap the ownership of the double data pointed at by ptr as well as the double num from the old object to the new object. To do this, the swap function is used, which simply swaps the values that two variables hold. It is safe to use the swap function here because we do not care about what happens to the moved-from object since the moved-from object is an r-value. For pointers, the swap function swaps the memory locations pointed at by the varibles, which is exactly what is desired in the move constructor. On line 16, the move constructors in S and in the unique_ptr class are called in the move constructor of C. We are forced to use the std::move() function here because the expressions c.s and c.dat are l-values even though they reference an r-value expression; i.e. we must convert the l-value expressions c.s and c.dat to r-value expressions. One should always call move constructors when available to move object member variables.

Notice the parameter of the move constructors above is an r-value reference. An r-value reference can hold either a pr-value or an x-value; a pr-value is a variable that does not have an identity (a location in memory) and an x-value is a variable that does have an identity but is temporary. In either case, it is safe to take the data in the old object and move it to the new object, since the old object either cannot be used by us (pr-value: it has no name so we can’t use it) or is temporary and should be deleted immediately (x-value). Of course, what has just been said depends on whether the programmer is writing code correctly. Consider the following main method:

```cpp
int main() {
    C c("hello", 10);
    C d(std::move(c));
    // c is in an unspecified state!
}
```

In the code above, on line 4 we call the move constructor of d by passing in c. After move construction, c is referred to as a “moved-from” object and is in an unspecified but valid state, meaning that there are no requirements in C++ on what the moved-from object should look like after move construction. In the move constructors above, we use std::swap in the implementation, but other valid move constructors for S could look like

```cpp
S::S(S&& s) : ptr(s.ptr), num(s.num) {}
S::S(S&& s) : ptr(s.ptr), num(s.num) {s.ptr = nullptr;}
```

In the three constructors above, the pointer in the move-from object is one of: a hanging pointer, a nullptr, or a pointer to the data. All three constructors are valid move constructors, and all
three can be used in valid C++ code. This is why the state of the moved-from object is considered unspecified: it could look like almost anything depending on the implementation of the move constructor. Now, going back to our conversation before, notice that by casting the l-value to an r-value using std::move, we are telling the compiler that the old object is temporary. We need to keep our end of the bargain by not allowing situations like the in the main method above, where c actually is not temporary and can be used later in the code.

### 1.3 Destructor

For classes composed solely of stack data, it is not necessary to write a constructor because the compiler automatically generates one that (shallowly) deletes all stack data in the class. However, if one has pointers as member variables, one has to write one’s own destructor, because otherwise the pointers will be deleted but the data they point to will not be deleted. In other words, there will be a memory leak. This is why on line 9 inside the S class, we manually must delete the data pointed to by ptr. Notice in the C class, we do not have to provide a destructor because of our use of a smart pointer. The smart pointer’s destructor will automatically delete the dynamically allocated memory without our having to request it to do so. This is a key advantage of smart pointers.

### 2 Namespaces

Namespaces are organizational tools. They allow us to put a bunch of related functions and classes into the same folder, and then put these folders into a file cabinet, so we don’t just have a function of functions and classes lying on the floor all over our office. Consider the following code

```cpp
namespace Space {
    void fun() {
        // do nothing
    }
    class A {
        // no data
    };
}
```

Here, fun and A are assumed to be related somehow, so we put them in the same namespace Space. To use either the function fun or the class A, we would do

```cpp
int main() {
    Space::fun();
    Space::A a;
}
```

Although the syntax above looks the same as when we are defining member functions of a class, it is different. For member functions, saying Class::func() is specifying which class the member function func belongs to, whereas above we are specifying which namespace the functions/classes belong to. The following code should be understandable

```cpp
using namespace Space;
int main() {
    fun();
}
```
If we declare functions or classes within a namespace in a header file, we must include the namespace specifier for the definitions in the .cpp file. For example, in .h

```cpp
namespace Space {
    void fun();
    class A {
        int stuff();
    }
}
```

and in .cpp

```cpp
void Space::fun() { // blah blah }
Space::A::stuff() { // blah blah }
```

If one knows there will be no namespace collisions in the .cpp file, one can use “using namespace” to get rid of some of the specifiers

```cpp
using namespace Space;
void fun() { // blah blah }
Space::A::stuff() { // blah blah }
```

Another option is given below, which is useful when multiple namespaces need to be used in a .cpp file

```cpp
namespace Space {
    void fun() { // blah blah }
    A::stuff() { // blah blah }
}
```

### 3 Some keywords

#### 3.1 Explicit

Something is done implicitly in C++ if the compiler does something without explicitly being asked by the programmer to do so. For example, consider

```cpp
struct A {
    A() : x(0) {}
    A(int x) : x(x) {}
    int x;
};
int main() {
    double y = 3.14;
    int x = y; // y is implicitly casted to the int
    A a = y; // y is converted to an integer, then
    // the non-default constructor of a is called,
    // then the data is copied/moved (moved if available) from
    // the constructed A to a
}
```

Line 8 is implicit conversion and line 9 is implicit copy initialization. When the compiler sees code like line 9, it will consider the constructors of A along with all of the conversion functions that take
in a parameter with the same type as the expression. If there exists a conversion function (double-\int above) and a constructor (A(int) above) that will allow for construction of the class, they will be used by the constructor. To prevent this from occurring, one can use the explicit keyword:

```cpp
struct A {
    A() : x(0) {}  
    explicit A(int x) : x(x) {}
    int x;
};
int main() {
    double y = 3.14;
    int x = y;  // y is implicitly casted to the int
    A a = y;  // not going to work, must call the constructor explicitly
}
```

### 3.2 Delete

Setting a constructor equal to delete makes it unable to be used. Examples include the copy constructor of the std::ostream and of the std::istream classes as well as the copy constructor of the std::unique_ptr class.

### 4 Copy Elision

The kind of copy elision we will discuss in this class is called “NRVO” or “named return value optimization.” Consider the following (borrowed from en.cppreference.com):

```cpp
struct Noisy {
    Noisy() { std::cout << "constructed\n";  }
    Noisy(const Noisy&) { std::cout << "copy-constructed\n";  }
    Noisy(Noisy&&) { std::cout << "move-constructed\n";  }
    "Noisy"() { std::cout << "destructed\n";  }
};
std::vector<Noisy> f() {
    std::vector<Noisy> v = std::vector<Noisy>(3);  // copy elision when initializing v
    return v;  // NRVO from v to the result object (not guaranteed in C++17)
}
void g(std::vector<Noisy> arg) {
    std::cout << "arg.size() = " << arg.size() << \n;  }
int main() {
    std::vector<Noisy> v = f();  // copy elision in initialization of v
    g(f());  // from the temporary returned by f() (since C++17)
}
```
In this code, the construction of the `std::vector<Noisy>` in main will be elided: `v` will be directly initialized instead of copy initialized from the temporary object returned by `f()`. Thus, instead of copying the `std::vector<Noisy>` from the scope of `f()` to a temporary in `main()` and then copying the data from the temporary in `main()` to `v`, the compiler directly assigns to `v` in `main()` the vector `v` constructed in `f()`. In other words, line 11 directly initializes the `std::vector` in `main()` rather than initializing an `std::vector` in `f()`.

Because of copy elision, we will not see “copy-constructed” printed out in the code above. The print statements in the constructors are called “side effects” of the constructors: a side effect of a function is a change the function creates outside of the scope of the function. Thus, copy elision will cause side effects of avoided constructors to not occur, so one must be careful about depending on side effects of copy/move constructors. Note that some compilers do not automatically copy elide; in particular, Visual Studio does not (the last time I checked).