Constructors, Destructors & Assignment

Constructors and Destructors

The job of a *constructor* is to initialize a block of memory that the system has just allocated. Conversely, the job of a *destructor* is to do any last-minute cleanup required when the system is just about to deallocate memory.

For local variables, constructors run whenever we enter the scope[1] of local (stack) variables, and destructors run whenever we are about to leave the scope of these variables. So, if a program contains code

```
{
    string s;
    vector<string> v{10};

    ... use s and v ...

}
```

what really happens is more like:[2],[3]

```
{
    string s;                   // Space for s is allocated on the stack, and string's
                                // default constructor runs (initializing s
                                // empty string).
    vector<string> v{10};       // Space for v is allocated on the stack, and the
                                // vector<string>(int size) constructor runs.
                                // (This constructor then creates a primitive array
                                // of 10 strings on the heap, which also involves
                                // the two steps of memory allocation [on the heap]
                                // and initialization [using the string class's default
                                //  constructor].)

    ... use s and v ...

    // The destructor for v runs automatically here (the destructor has to clean
    // up the strings it allocated); once the destructor has run, the stack space
    // for v itself can be re-used or popped.
    // The destructor for s runs automatically here (the destructor has to clean
    // up any heap memory used to store the characters in the string), so the
    // stack space for s itself can be re-used or popped.
}
```

[1] The *scope* of a variable is the section of the program where we can refer to it by name.

[2] The code in this handout does *not* follow good commenting practices—most comments are obvious to anyone with experience in C++.

[3] In fact compilers typically generate code that allocates space on the stack for all the local variables when a function begins, that runs the constructors and destructors on this space exactly as shown, and finally that deallocates all the function's stack space at once when it returns.

C++ destroys stack objects in opposite order from their creation, so v's destructor is first.

For objects on the heap, constructors and destructors run only when the programmer says `new` or `delete` respectively. Thus, when we evaluate the expression

```
new vector<int> {10}
```

the job of `new` here is

1. First, allocate space for a `Vector<int>` object in an unused portion of the heap.
2. Only then, run the appropriate `vector<int>` constructor (the one taking a single integer argument) to initialize that space—in this case, it will allocate more heap space suitable for containing 10 integers.

If `p` points to a single object on the heap, the statement

```
delete p;
```

will (1) run the destructor for the object being pointed to, and then (2) mark that heap space as available for future uses of `new`.

If `p` points to an *array* on the heap, the statement

```
delete [] p;
```

will (1) run the destructor for each object in the array, and then (2) mark that heap space as available for future uses of `new []`.

Constructors are *not* responsible for allocating memory for the object itself being constructed—the constructor code always starts out with `this` pointing to a freshly-allocated block of memory. Conversely, the destructor is *not* responsible for deallocating memory occupied by the object being destroyed—you never ever say `delete this;` because the destructor only runs when the system is already about to reclaim the object's memory.[4]

Default Initialization

For local variables, if we don't specify any initialization, it is *default initialized*. Thus, all the examples below are default initialization:

[4] In addition, because the object whose destructor is running might be on the stack (in which case passing its address to `delete` could completely confuse the memory manager), and because the first thing `delete` does is invoke the destructor (so you'd go into an infinite loop), ...

```
string s;
vector v<int>;
int n;
char* cp;
int scores[24];
```

"Default initialization" means different things for primitive types and objects. For primitive types, it actually means *do nothing at all*, whereas for object types, default initialization invokes the default constructor.

Thus `n`, the pointer `cp`, and all the values in the `scores` array contain whatever bits were in those memory locations before (thus, `cp` is pointing who-knows-where). In contrast, `s` and *v* are properly initialized with well-defined contents (an empty *string* and empty *vector* of *int*s, respectively).

The behavior of primitive types is different from Java, where such variables are guaranteed to be initialized to zero or null!

In contrast, the definitions below all specify how to initialize the variables they declare and define:

```
int x = 0;
int y { 0 };
int z { };

int scores[5]  = { 71, 85, 76, 68, 83 };

int* p         = new int;
vector<int>* q = new vector<int>;
```

In this example, `x`, `y`, and `z` are all initialized to zero. In the case of `z`, this is known as *value initialization*.

"Value initialization" means different things for primitive types and objects. For primitive types, it actually means "set it to zero" (`nullptr` in the case of pointers), whereas for object types, value initialization invokes the default constructor (no differently from default initialization).

When you write constructors for a class that contains data members that are primitive types, you can choose how to initialize those data members. If you do nothing, it'll be default initialized and have an arbitrary value. Generally, you want to properly initialize things.

For C++ *objects*, the default constructor usually does something useful (though it depends on the author of the class). The default constructor for the `string` class initializes the object to an empty string. The default constructor for the `vector<T>` class (for any type `T`) initializes the object to an empty vector.

Not every class is guaranteed to have a default constructor. If *Cow* has no default constructor, then none of the following lines will compile

```
Cow mycow;                  // A default cow
Cow mycows[10];             // 10 default cows
vector<Cow> mycows10;       // Starts out with 10 default cows
```

By contrast, `Cow* p;` would compile, because default initialization of pointers does nothing.

The Copy Constructor

The copy constructor takes a single argument, of the same type as the object being constructed, and makes a copy. Thus, if `s` is some `string`, we can say

```
string  s2{s};
int     n{42};
int*    p{new int{7}};
```

which makes the string `s2` contain a copy of the characters in the string `s`, and makes the integer `n` a copy of the number 42. The third line allocates a fresh memory area on the heap (large enough to hold an *int*), initializes it with a copy of the number 7, and copies the address of that location into `p`. An alternate syntax for doing exactly the same thing is

```
string  s2 = s;
int     n = 42;
int*    p = new int {7};
```

but this alternative is essentially just "syntactic sugar"—a nicer way of writing the same thing. As a matter of style, we will prefer this notation. But it's important to realize that, whenever you declare a local variable and specify an initial value of the same type, what really happens is a use of the copy constructor!

The copy constructor for primitive types just copies the bits in memory: thus numbers get copied, and copying a pointer just makes a copy of the address. (This behavior is sometimes referred to as a *shallow* copy.)

Copy constructors for objects often do more. The copy constructor for *string* makes a full copy of all the characters; if you modify the copy, the original remains unchanged, and vice-versa. The copy constructor for *vector* makes a brand new vector of the same length, using the appropriate copy constructor to copy each element. Thus, if you copy a *vector<char*>* you get a new vector containing exactly the same sequence of addresses. But, if you copy a *vector<string>*, you get a new vector of new (copied) strings.

The other major use for the copy constructor is in function calls and returns. Function arguments are normally initialized by using the appropriate copy constructor. Thus every call to the following function:

```
int sumvector(int initial, vector<int> v)
{
    int total = initial;
    for (unsigned int i = 0; i < v.size(); ++i)
        total += v[i];
    return total;
}
```

initializes a local variable `initial` as a copy of the supplied integer, and a local variable `v` as a (full) copy of the vector of integers being summed.

Functions that return values also use the copy constructor by default; the caller of the function gets a copy of the value returned by the callee.

These copies can sometimes be avoided by using reference types (or `const` reference types). In some cases, the compiler can optimize away copies on its own.

Built-In Destructors

The classes in the C++ library typically have destructors that do sensible cleanup. The *string* destructor deallocates any heap memory it was using to store characters. The *vector* destructor deallocates any heap memory it was using to store its elements (which recursively invokes the appropriate destructor for each element, since their memory is being reclaimed as well). The same behavior occurs when primitive arrays are deallocated; a destructor runs for each element of the array.

All primitive types do nothing in their destruction phase (technically, because they are not instaces of a class, they do not even have destructors). Integers and pointers (regardless of the type of object they point to) are also primitives! So, their destructino phase does nothing. There are very good reasons for this behavior, but it may surprise you initially.

When a pointer variable on the stack goes out of scope, or a pointer value on the heap has its memory reclaimed, the value it points to is *not* automatically deallocated. Similarly, when an array or vector of pointers is deallocated, running the destructor for each pointer in the array does nothing, and any data being pointed to is unaffected.

The Assignment Operator

The *assignment operator* (named `operator=` in C++) is often confused with the copy constructor[5], but they are really separate and used at different times. If `s1` and `s2` are strings, then the definition

```
string s = s1;
```

makes use of the copy constructor (initializing an uninitialized piece of memory for `s`), whereas the assignment

```
s2 = s1;
```

is a use of the assignment operator (resetting the contents of a *previously-initialized* piece of memory).

Don't be confused by the fact that `=` appears in both lines! If a new variable is being initialized (constructed), the copy constructor runs. If a pre-existing variable is being assigned to, the assignment operator runs. In general, you have to write these as two separate pieces of code.

The assignment operator overwrites the bits for primitive data (so that assignment of integers, floating-point values, and pointers acts exactly as you would expected). For strings and vectors, the contents of the left-hand-side are replaced by a copy of the right-hand-side.

By convention, the assignment operator returns the object that was just assigned to, by reference (to avoid extra copying). Returning the target of the assignment allows code such as

```
a = b = c = 0;
```

to work, since it parenthesizes as

```
a = (b = (c = 0));
```

In CS 70, we don't consider either of these forms very readable, so you should avoid them in your own code. But when you write assignment operators, you nevertheless need to give your assignment operators the proper return type so that "old school" programmers can write their one-liners.

Writing Your Own Constructors

The constructors are declared in the class declaration using the same name as the class but with no return type[6], e.g.,

[5] The version of the assignment operator described here is even called the *copy-assignment operator* in the C++ Standard.

[6] When declaring or implementing the default constructor, you *always* need the empty parentheses. Sigh…

```
class Cow {
public:
    Cow();                          // I plan to write a default constructor
    Cow(const Cow& other);          // I plan to write a copy constructor
    Cow(std::string name);          // I plan to write another constructor

    ... declare other member functions ...

private:
    std::string name_;
    Cow* bestFriend_;

};
```

You then can write code for the constructors you have declared, e.g.,

```
Cow::Cow()
{
    name_       = "Bessie";
    bestFriend_ = nullpr;
}
```

but more is going on than it appears. *Before* the code you write for any constructor runs, all the data members (fields) of the object first have their constructor run (by default, the default constructor). The constructors always run in the order the data members were listed in the class declaration.

Thus, the full effect of the *Cow* default constructor above is to default-initialize `name_` (to an empty string), to default-initialize the pointer `bestFriend_` (which does nothing and leaves the pointer undefined), and only then to assign the string `"Bessie"` to `name_`, and then to assign the value `nullptr` to `bestFriend_`.

In this case, not only is it not good C++ style, it's also a little bit wasteful—first we initialize `name_` to the empty string, and then we overwrite this empty string with `"Bessie"`. We can request that a data member use a non-default constructor by specifying *colon-initializers* to supply constructor arguments:

```
Cow::Cow()
        : name_("Bessie"), bestFriend_(nullptr)
{
    // Nothing else to do
}
```

In this revised code, all data members have been colon-initialized (e.g., `bestFriend_` is being initialized using the copy constructor for pointers), and so by the time we get to the actual "code", there's no more work for the constructor to do. You don't need to list all data members; any that you don't mention will be default-initialized. But it's considered good style to always use colon initialization to initialize data members

unless the constructor arguments would be unreasonably complicated. (It's also good to put a comment as above when there's no code, so you know you didn't accidentally forget to write the body of the constructor.)

Data members are initialized in the order they appear in the class declaration. Listing them in a different order in the colon-initialization list has no effect, except to be confusing and/or misleading. (The compiler will warn you about this.)

Writing Your Own Destructors

Destructors are declared with no return type and an empty argument list, using the class name preceded by a tilde (~).

```
class Cow {
    ... same as above ...
    ~Cow();                          // I intend to write a destructor
};
```

The implementation code then would be

```
Cow::~Cow()
{
    ... your cleanup code ...
}
```

Destructors are backwards from constructors: *first* the code you write runs (e.g., closing files, deallocating heap memory, etc.), and then the destructor runs for each data member, from last to first.

Writing Your Own Assignment Operator

If you want to write your own assignment operator, you need to declare it in the class declaration:

```
class Cow
{
    ... same as above ...
    Cow& operator=(const Cow& rightHandSide);
};
```

and then write code, e.g.,

```
Cow& Cow::operator=(const Cow& rightHandSide)
{
    if (this != &rightHandSide) {
        .. .clear out LHS object (i.e., the one that this refers to) if needed ...
        ... copy rightHandSide into LHS object ...
    }
    return *this;                    // Return the left-hand-side of
                                     // the assignment by reference.
}
```

The test `this != &rightHandSide` tests for the case of *self-assignment* (specifically, it compares the addresses of the right-hand-side object and left-hand-side objects, to see if an object it being assigned to itself). Otherwise, statements like

```
mycow = mycow;
```

could break, because we would first clear out the left-hand-side, and then try to copy the contents of the (now cleared) right-hand-side.[7]

Also, the convention is that the assignment operator returns the *left-hand-side* of the assignment. Thus, by convention any assignment operator you write should finish by returning `*this`, as shown above.

Let the Compiler Do It?

If you don't want to write some of the things above, you can just let the compiler do it! However, the code the compiler writes may or may not do what you need.

- If you declare no constructors *at all* in the class declaration, then the compiler will write a default constructor that does nothing except default-initialize all the data members in order.
- If you declare no copy constructor (a constructor that takes an object of the same class by reference) in the class declaration, the compiler will write a copy constructor; it copies (copy-constructs) each of the data members in order.
- If you declare no copy-assignment operator (defining `operator=` to take an object of the same class) in the class declaration, the compiler will write an assignment operator; it just assigns to each of the data members in order.
- If you declare no destructor in the class declaration, the compiler will write a destructor that invokes the destructors for each of the data members (in reverse order).

The criterion for whether or not a default constructor is provided by the compiler is *different* than the criteria for the copy constructor, the assignment operator, and the destructor.

[7] An alternate technique to avoid the problem, preferred by some writers because it interacts better with exceptions (not discussed in CS 70), is the "swap technique" used in some future homework assignments.

Thus, the class declaration:

```
class MyString {
public:
    std::string get();
    void set(const string&);

private:
    std::string s_;
}
```

is treated by the compiler as if you had written:

```
class MyString {
public:
    MyString();
    MyString(const MyString&);
    MyString& operator=(const MyString&);
    ~MyString();

    std::string get();
    void set(const std::string&);

private:
    std::string s_;
};
```

and written the following implementation code

```
MyString::MyString()
: s_{}
{
    // Nothing else to do
}

MyString::MyString(const MyString& original)
: s_{original.s_}
{
    // Nothing else to do
}

MyString& MyString::operator=(const MyString& rightHandSide)
{
    if (this != &rightHandSide) {
        s_ = rightHandSide.s_;
    }
```

```
    return *this;
}

MyString::~MyString()
{
    // Nothing to do (before data members are automatically destroyed)
}
```

Disable It?

If it doesn't make sense for your class to be default constructed, copyable, or assignable, you should remove that functionality from the interface of your class. Because the C++ compiler will usually create this functionality for you, you have to explicitly remove it. The following code declares a class *Person* and disables its default constructor.

```
class Person {
public:
    Person() = delete;          // A Person is cannot be default constructed

    Person(const std::string& name, int age);
private:
    std::string name_;
    int age_;
};
```

Because the default constructor is disabled, users of the *Person* class must provide sensible default values for the class's data. If a user tried to write `Person p;` the compiler would give an error.[8]

It's up to the designer of a class to decide whether it should support default construction, copying, or assignment. Typically, the class should not be default-constructible if there is not a default instance of the class that all users would agree is sensible. Typically, the class should not be copyable or assignable if the class doesn't need to be copied, e.g., if there should only ever be one version of a particular instance at a time. For example, we might decide to disable copying and assignment in the *Person* class because each person is unique.

[8] If we had just omitted the definition of *Person*'s default constructor, the compiler would also give an error, but the error message would probably be less helpful: rather than telling the user that the default constructor had been explicitly disabled, it would tell the user that the class has no matching constructor. It's good style to be explicit about the behavior we're disabling.