# Copy Functions

In general, a standard copy operation, which might also be called simple assignment, occurs when you use the = operator to assign the value of one variable to another. The expression `myInt2 = myInt1`, for example, causes the integer contents of `myInt1` to be copied from the memory used by `myInt1` into the memory used by `myInt2`. Following the copy operation, two separate areas of memory contain the same value. However, if you attempt to copy a Core Foundation object in this way, be aware that you will not duplicate the object itself, only the *reference* to the object.

For example, someone new to Core Foundation might think that to make a copy of a CFString object she would use the expression `myCFString2 = myCFString1`. Again, this expression does not actually copy the string data. Because both `myCFString1` and `myCFString2` must have the CFStringRef type, this expression only copies the reference to the object. Following the copy operation, you have two copies of the reference to the CFString. This type of copy is very fast because only the reference is duplicated, but it is important to remember that copying a mutable object in this way is dangerous. As with programs that use global variables, if one part of your application changes an object using a copy of the reference, there is no way for other parts of the program which have copies of that reference to know that the data has changed.

If you want to duplicate an object, you must use one of the functions provided by Core Foundation specifically for this purpose. Continuing with the CFString example, you would use `CFStringCreateCopy` to create an entirely new CFString object containing the same data as the original. Core Foundation types which have “CreateCopy” functions also provide the variant “CreateMutableCopy” which returns a copy of an object that can be modified.

## Shallow Copy

Copying *compound objects*, objects such as collection objects that can contain other objects, must also be done with care. As you would expect, using the = operator to perform a copy on these objects results in a duplication of the object reference. In contrast to simple objects like CFString and CFData, the “CreateCopy” functions provided for compound objects such as CFArray and CFSet actually perform a *shallow copy*. In the case of these objects, a shallow copy means that a new collection object is created, but the contents of the original collection are not duplicated—only the object references are copied to the new container. This type of copy is useful if, for example, you have an array that's immutable and you want to reorder it. In this case, you don't want to duplicate all of the contained objects because there’s no need to change them—and why use up that extra memory? You just want the set of included objects to be changed. The same risks apply here as with copying object references with simple types.

## Deep Copy

When you want to create an entirely new compound object, you must perform a *deep copy*. A deep copy duplicates the compound object as well as the contents of all of its contained objects. The current release of Core Foundation includes a function that performs deep copying of a property list (see `CFPropertyListCreateDeepCopy`). If you want to create deep copies of other structures, you could perform the deep copy yourself by recursively descending into the compound object and copying all of its contents one by one. Take care in implementing this functionality as compound objects can be recursive—they may directly or indirectly contain a reference to themselves—which can cause a recursive loop.