




Object Data Structures on Legacy Hardware

One of the most fundamental aspects of programming is finding a way to represent complex objects. An object is any one item that has multiple attributes. For example, a Facebook profile is one item with multiple attributes. Each profile is comprised of the same parts - a name, photos, posts, personal information, and a friend list. Although each profile object is different, they are all structurally the same. Computer scientists and engineers have designed not only programming languages, but also processors around creating digital objects. The field often refers to this as Object Oriented Programming, and it has dominated how we look at data science for the past 30 years. The most useful data structure when programming just about anything, is creating a collection, or list, of these objects and looping through them to perform repetitive tasks. Every time someone logs in to Facebook, their program loops through a list of all the user's friend objects, and performs the repetitive task of displaying their post data on a page. Without the ability to loop through a collection of objects, Facebook would be unable to show their users an up-to-date page of all their friend's photos and text posts. Because engineers and data scientists have spent decades perfecting this data structure, modern programming languages and processors essentially have this feature baked into them, and it is prevalent practice to simply create an array of structs to represent this data. But how can this be implemented on a computer that can only address a 256 byte array? Creating a list of 10 objects would only leave 25 bytes per object, barely enough data to store someone's name and phone number. Using this standard implementation would severely limit any program. This term I made a goal to implement this data structure for the Nintendo Entertainment System or NES. This datatype is absolutely vital when creating any game object that moves, such as the player, as these objects will have a wide variety of attributes such as X and Y coordinates, hit detection, collision detection, and game physics, just to name a few. To further complicate this goal, the NES is a video game console that uses a 40 year old processor, and runs off an all-but-dead computer language lacking all modern features.

In order to create an object datatype for the NES, a programmer must take into account the many ways to accomplish this task. To demonstrate the organization techniques being discussed, all of the examples will use these basic cubes. Like all cubes, these cubes will have a length, a width, and a height. For ease of readability, they will be given unique values for each measurement, although this technically makes them square prisms, and not cubes. Because computers begin counting at 0, these cubes will be named cube0, cube1, and cube2. The 6502 processor found in the NES can only address 256 byte arrays, so how can this data be organized so that it is not only modular, but also produces the fastest and simplest code possible?

	cube0	cube1	cube2
			
Length	01	04	07
Width	02	05	08
Height	03	06	09

The first layout to consider is the simplest, but also the most limited. A developer can easily place the data within three unique arrays, storing the data in order of length, width, and height. Although this is a very fast way to access data, it essentially hardcodes these values in place, and forces the developer to add a new array each time the project is scaled upward. Additionally, this approach can not be treated as a list of items, and the processor is therefore unable to iterate over the entire collection within a loop. Because the items are not grouped in one list, this technique cannot be used for any meaningful work, and will not be used. As a general rule of software design, any hardcoded approach will create messy code and will become a nightmare to change and maintain.

The next organizational method is most akin to the C style struct of modern computer languages. Unfortunately, unlike modern processors which have special registers to address more complex data, the 6502 is still limited to only addressing within one array, so this implementation will group all three objects into one array

	L	W	H	
cubes:	000002FD	00	00	00 ...
	00000300	01	02	03 ...
	00000303	04	05	06 ...
	00000306	07	08	09 ...
	00000309	00	00	00 ...

$x = (\text{SIZE_OF_OBJECT} * \text{objectNumber}) + \text{field}$
finding the length of cube2
 $\text{SIZE_OF_OBJECT} = 3 \text{ (bytes)}$
 $x = (3 * 2) + 0 = 6$
 $\text{cubes}[6] = 07$

called “cubes.” With all the objects in a single array, the computer can now iterate over the data as a list, making code smaller and easier to expand in the future. Although this code is modular, it has some severe drawbacks. Every time that the computer accesses any field, it must first solve an algorithm involving multiplication. Unfortunately, another limitation of the processor is that it cannot perform multiplication natively. To circumvent this, it must perform addition multiple times to find the product, an operation that would occupy the processor for dozens of cycles each time an attribute is fetched. A programmer can reasonably expect to address within their list of objects dozens, even hundreds of times per frame, so this added processing cost would grow exponentially, lowering frame rates, and eventually leading to stripped game features and performance plateaus. Furthermore, because every array has a 256 byte limit, the list of objects can only become so large. If the game allows for 10 moving objects to be on screen simultaneously, this only allows for 25 bytes per object, a limit that is easy to surpass on even the simplest programs, a limit that I hit very quickly. Needless to say, a better solution needs to be found in order to create a robust program for the NES, and this method will also not be used.

The last implementation strategy, and the one this program will use, acknowledges that every object cannot be placed into one list. Instead of creating one big list, this arrangement will create multiple lists, grouped by field. Each object is now represented by a position within the array. Simply put, cube0 will be the 0th byte in each array, cube1 will be the 1st byte, and so on. Even though each field is a separate array, the processor can still iterate through the objects, as programmers will only need access to data as it relates to fields. Not only does this eliminate the need to perform any complex algorithms to access fields, it also makes adding fields and scaling the program incredibly simple. To add an additional cube, the programmer will simply make each array one byte longer. To add a field, the programmer only needs to add an

	000002FD	00	00	00
lengths:	00000300	01	04	07
widths:	00000303	02	05	08
heights:	00000306	03	06	09
	00000309	00	00	00

Finding the length of cube2

lengths[2] is 07

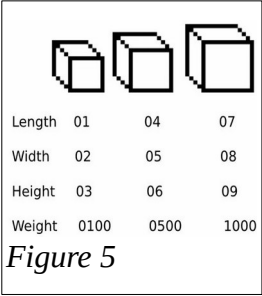


Figure 5

additional array. This will be the definitive method my program will use when storing objects in ROM, as well as unzipping, altering, and accessing those values in RAM when they are needed.



Length	01	04	07
Width	02	05	08
Height	03	06	09
Weight	0100	0500	1000

Adding one byte fields to the objects like the ones mentioned earlier is quite straight forward, but fields can be much longer than one byte. For example, perhaps the cubes have a weight that requires two bytes to store. The processor is limited to an 8 bit data bus, so the data will need to be accessed separately. The length, width, and height were

all stored as one byte, so how can this attribute be stored so that it does not effect our previous standard of having each byte in the array correspond to the object it represents?

The simplest conclusion that can be reached is to store the weights in one array twice the length of the others, and take that into account when addressing the data. Usually this can be done pretty quickly, as most fields will not need to be longer than two bytes long, and the 6502 can take advantage of a base 2 system like binary and double numbers by simply shifting the bits to the left, an instruction the processor can perform quite easily. Although this is easy to implement, it will not be used as it does not work in the same way as our other attributes. The programmer would have to remember every special

weights:	0000030C	00 00 00 00 00 00
	00000312	01 00 05 00 10 00
	00000318	00 00 00 00 00 00


To find the weight of cube2
SIZE_OF_FIELD = 2 (bytes)
x = object * SIZE_OF_FIELD
x = 2 * 2
2 (00000010) becomes 4 (00000100) when shifted left
weights[4] is 10 and weights[4+1] is 00
cube2 weighs 1000

	000002FD	00 00 00
	00000300	01 04 07
	00000303	02 05 08
	00000306	03 06 09
weightsHigh:	00000309	01 05 10
weightsLow:	0000030C	00 00 00
	0000030F	00 00 00

To find the weight of cube2
weightsHigh[2] is 10 and weightsLow[2] is 00
cube2 weighs 1000

attribute that is two bytes long, and address it differently than all the others. If a mistake is made, and the two byte attribute is not handled differently, the program will have unwanted behavior, leading to frustration and bugs. Instead, every attribute that is two bytes long will be broken up into a high and low address. The computer can only access one byte at a time, so splitting these attributes in two will have no effect on mathematics performed on them. Now the object number still corresponds to its byte position within the array, and no mathematics need to be performed in order to access the data.

This approach should also be used to store fields with arbitrary lengths. Perhaps the cubes have colors now, but each cube has a different amount of colors. Cube0 has one color, cube1 has two, and cube2 has three. When creating the arrays, each object will be treated as though they all have the maximum number of fields, in this case each cube will have enough room to store all three colors. However, because each cube has an arbitrary amount of colors, there is no way for the computer to tell that cube0 only has one color and that the other two bytes are unused. Reading from values that are uninitialized will produce



	31	32	33	34	35	36	37	38	39
Length	01			04				07	
Width	02			05				08	
Height	03			06				09	
color0	31			31				31	
color1					37			37	
color2									39

garbage data. Therefore, when storing data with arbitrary lengths, the programmer must also store the amount of colors each object has, and use that variable to ensure that no uninitialized values are used. When storing arbitrary lengths in ROM, this program will store objects with more data first to avoid “burner bytes,” useless bytes of data used to pad out unused space. In this example we would store cube2 first, followed by cube1, and ending with cube0, and they would be renamed accordingly. This technique will save almost a page of ROM per field.

	000002FD	00	00	00
	00000300	01	04	07
	00000303	02	05	08
	00000306	03	06	09
color0:	00000309	37	37	37
color1:	0000030C	00	31	31
color2:	0000030F	00	00	39
numberOfColors:	00000312	01	02	03
	00000315	00	00	00