Introduction to Memory Management Scheme for Array in Programming Languages

CS2163 Java Additional Class Notes

This document needs to be studied carefully and be understood along with materials in textbook. Before you take Test 2, you need to fully understand this document.

Along the years when I am teaching programming languages courses, I find out that those students who had good understanding of memory management scheme turned out to have easier journey when sailing through the course material, compared to the harsh time that other students experienced, who did not have such understanding. It then becomes a motivation for me to write down this illustration document to help student achieve a pleasant tour in the programming course. Let’s begin with two basic questions about memory in the computer system.

*Question 1: what does memory do in computer system?*

Answer: store data.

*Question 2: what are two basic issues that the memory must handle when storing data?*

Answer: the first one is **what value** to read from or write into the memory, and the second is **where** to read or write the value.

The two questions above catch the core functionalities of memory. The software programs we write have direct connections with these two core functionalities. Let’s first take a look at the pin layout of a memory chip.

Before we continue, you might have this question for me: wait a minute, isn’t this memory chip layout supposed to be in the material of a computer hardware class? My answer to you doubts is: yes, it is hardware class material, but, if you don’t have this hardware knowledge, it will be difficult for you to understand a variety of important concepts in software programming class. Now let’s continue.

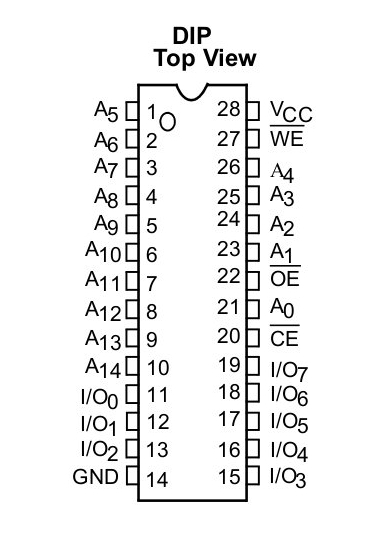


Figure 1: A typical memory chip layout

Figure 1 illustrates a typical memory layout, and it’s a top view. The acronym DIP means “Dual In-line Package”, which is obvious, since this chip package has double lines of pins in parallel: with one column of pins at each side of the chip. Each pin has a name shown in the figure. Among the pins, let’s focus on these two set of pins:

**Data pins:** they consist of 8 pins, with pin names as **I/O7, I/O6, I/O5, …, I/O1,** and **I/O0**. The abbreviation I/O means these data pins can be used as Input and Output bus, which implies that user can write data into (input) or read data from (output) the memory chip. Each data pin can have a high voltage or a low voltage, corresponding to the binary value of 1 or 0. These eight pins together represent 8 bits of data, which is 1 byte.

**Summary of data pins**: they solve the issue of **what value** to read from or write into memory.

**Address pins:** in figure 1, the memory chip consists of 15 pins, with pin name as **A14, A13, A12, …, A1,** and **A0.** Each pin also can have a high voltage or low value, corresponding to binary value of 1 or 0. The total number of addresses these 15 pins can provide is 215 , which is 32K, because 215 = 25 X 210 , and 25 = 32, 210 = 1024, which is called 1K when measuring memory size. The lowest address provided by these 15 address pins are 0x0000 when all the address pins are in binary value 0, and the highest address provided by them is 0x7FFF when all the address pins are in binary value 1. The unique starting character of 0x indicates this is a hexadecimal value, and in computer science, address value is always expressed in hexadecimal format.

**Summary of address pins**: they solve the issue of **where** to read from or write into the memory.

An analogy of **memory** is a hotel, with the **memory address** compared to **hotel room number**, and the **data in memory** being compared to the **guest staying in a hotel room**. A hotel manager, by accessing the hotel room number, can find out which guest stays in this particular hotel room; similarly, the software program, by accessing the memory address, can retrieve the data dwelling in this particular memory address.

Attention: I ASSUME you have already gained enough knowledge about binary and hexadecimal number system, if not, you need to learn binary and hexadecimal by yourself, for example, you can read the first two pages of file “Test1StudyGuide.docx” in Moodle folder “Test 1 Review Lesson”. This is a computer science class, knowing binary and hexadecimal is like knowing the English alphabet for an English writing class, or knowing the keyboard notes for a piano class. **You can’t say you are good at writing without knowing the alphabet; similarly, you can’t say that you are a good computer science student without knowing binary and hexadecimal.** You need to know how to:

1, convert between binary and hexadecimal

2, convert between binary and decimal

3, convert between hexadecimal and decimal

4, calculate how may address pins are needed for a certain memory size (for example, how many address pins are needed for a memory of 4GB?), while assuming the data pins are 8 bits (1 byte).

If you still cannot figure out the questions above after studying binary and hexadecimal, **you should contact me for help**, and I am willing to help you achieve a good understanding of these number systems. Also, there is no ambiguity between understanding and not-understanding binary and hexadecimal. You either understand it or you do not, thus you need to make sure you are crystal clear about binary and hexadecimal.

Java virtual machine JVM has 32bit mode and 64 bit mode. Without the loss in generality, we use a 32-bit JVM system as example in this document, in other words, we assume there are 32 address bits in the JVM address space. Thus the total number of addresses that can be provided is 232, which is 4GB (why 4GB? You need to figure this out. Hint: 1KB = 210 bytes , and 1MB = 220 bytes, and 1GB = 230 bytes). Here is one way of how to allocate the 4GB address spaces, as illustrated by Figure 2 below.

The figure 2 below is a description of the memory allocation scheme for a general programming language, such as java or C++. I draw the lower address 0x00000000 on the top, and the higher address 0xFFFFFFFF at the bottom, which corresponds to the memory figure in the textbook, such as figure 5.3 in sec 5.3.1. In other textbooks, you may see the same memory allocation map with lower address 0x 00000000 being put on the bottom, and the higher address 0xFFFFFFFF being put on the top. Both drawing preference are good, and I choose this preference, because I want to be consistent with the drawing style in textbook.

|  |  |  |
| --- | --- | --- |
| **memory address** | **memory data: space usage** |  |
| 0x00000000 | executable code of a program |  |
|  | static data |  |
|  | heap | <---Memory allocated by the “new” keyword locates in heap. In java, this is where all the objects or array bodies locate (we cover array in chap 6, and object reference in chap 3 & 4).  Boundary of heap and stack |
|  |
| address increases from low to high |
| …heap keeps moving down… |
|  | …stack keeps moving up… |  |
|  | calling stack of method m2 | <--assume m2 is called by m1; |
|  | calling stack of method m1 | <--assume m1 is called by main; |
|  | calling stack of main method | <--formal parameters and local variables of main() method; |
|  |  |
| 0xFFFFFFFF |  |

Figure 2 Memory Management Scheme or Memory Space Usage Scheme

In textbook, Figure 6.4, the heap is drawn side by side with stack, which is just another way of drawing stack and heap. In the above figure, heap is drawn right above stack, to demonstrate the idea that, heap and stack share continuous memory addresses.

The above figure also conveys the idea that, stack size is limited, and if method calling is not handled correctly, stack can be overflowed, which is a serious mistake but not uncommon in programming. You should avoid stack overflow by eliminating the unlimited built up of the calling stacks.

Figure 2 above illustrates the memory allocation scheme for a program. As we can see from this figure, the entire memory is divided into 4 different partitions: executable code, static data, stack, and heap, with address increasing from low to high.

**1, executable code:** this is where the binary code of a program is stored, and CPU read the instructions from this area to execute commands of a program.

**2, static data:** this is where static data are stored. We can see that the static data are independent of any method’s stack, and also independent of any object or array in heap. Thus static data can exist outside any method or object or array, and it exists in a class, and that is why it is referred to as class variable.

**3, heap:** this is where free trunks of memory locates. These memory trunks are allocated by using the “new” keyword in Java. Java treats class type data and array name type (not variable of primitive data type) as object or array, thus the body of the object or array always locates in heap. That is the reason why each array must be created with the “new” keyword, which allocates a block of memory in heap to hold the actual data in array. In comparison, variable of primitive (built-in) data type are not treated as array in Java, thus primitive type variable hold its actual value, e.g., when we define a variable of int type in the program:

int myIntVar = 10;

, then the variable named myIntVar actually holds the value of 10. However, for variable such as an array name, since its body is stored in the heap, the variable name does not hold its actual data, instead, the variable name holds only the address (or reference) of the data block in the heap. For example, when we declare and create an array like this:

double[] scores = new double[3];

, then the variable named scores is only the address of a block of memory in the heap, and the content of this array in the heap has not been initialized yet, until the following lines are executed:

scores[0] = 301.5;

scores[1] = 72.6;

scores[2] = 196.68;

**4, stack:** this is where all the methods store their parameters and local variables. It is also called **run-time stack** or **call stack**. The content of this partition is dynamically dependent on what methods are being called when program executes. For example, main method may call method m1, which again calls another method named m2, thus the stack looks like what is shown in the above figure. After m2 finish executing, the stack of m2 will be released and free for used by the next invoked method. The same happens for m1, i.e., after m1 finish execution, the stack space occupied by m1 will be released and free for used by the next invoked method in main, e.g., if main method invokes m3 after invoked m1, then the calling stack of m3 will be on top of the calling stack of main when m3 is invoked. In each method’s calling stack, usually we are concerned with **two types of data**: **formal** **parameters** (if there is any) of this method passed by from the invoking method, and **local variables** declared inside this method. Source code of “*StackHeapExampleForArray.java*” is shown below:

// This is an example file to explain memory allocation schemes,

// for variable of primitive (built-in) data type and array type.

public class StackHeapExampleForArray{

public static void main(String[] args){

int myIntVar = 10; // int is one of the 8 primitive (built-in) types

double[] scores = new double[3]; // declaration and creation of array

scores[0] = 301.59; // now is the initialization of the array.

scores[1] = 72.6;

scores[2] = 196.68;

// the above four lines can be re-written as one line of code below:

// double[] scores = {301.59, 72.68, 196.68};

//, which follows the same process of declaration, creation, and

// initialization of an array. Even this one of code does not have

// the keywork new, but this one line of code does exactly the same

// thing as the four lines of code does.

// the rest of the method is omitted

}

}

The figure below indicates the memory allocation for the example program above.

|  |  |
| --- | --- |
| **memory address** | **memory data** |
|  |  |
| 0x003D042C | 301.59 : scores[0]  72.6 : scores[1]  196.68: scores[2] |
|  |  |
|  | refers to |
|  |
|  |
|  |  |
|  | 0x003D042C |
| refe | 10 |

int myIntVar is in stack

array named scores is in stack

this is the boundary between stack and heap, above is heap, below is stack.

below is the stack of main()

This block of memory in heap is allocated by the keyword new, and the body of this array is stored here. This memory block is referenced by the reference variable scores in main method, with the first content accessed as scores[0], etc.

myIntVar, variable of built-in data type int, it holds the actually value, which is 10 in this case.

scores, variable of array name, is a reference (its value is an address) variable, and it does not hold the actual data of the array.

Figure 3 Difference between variable of primitive (built-in) data type and variable of array

From Figure 3 above, we can see the difference between variable of built-in data type and and an array variable. **Variable named myIntVar is int type,** which is a primitive or built-in data type in Java, actually holds the data, in other words, the value of myIntVar is the data value, which is 10 in this example.

**Variable named scores is an array**, and it **does not** holds the array content, instead, **it only holds the reference** (the address) to the block of memory in heap, where the actual body the array is stored, i.e, the array content. In other words, the value of array name ***scores*** is only a reference to a block of memory in the heap. The allocation of this block of memory to the heap is done by using the keyword new, and a suitable memory block’s address is returned by the new statement, and assigned to the array name, as show in this statement:

double[] scores = new double[3];

Let’s go into more detail and nail down the events that happen inside this new statement. The major events that happen in sequential order behind the scene for the new statement are listed below:

1. a memory space to hold a reference variable is allocated in the calling stack of the method where this statement locates (in our example, it is the main method), and this reference variable is named ***scores***, and the value of ***scores*** has not been determined yet, which means, this reference value does **NOT** refer to any place in the heap yet.
2. the new statement allocates a block of memory in the heap, and the size is big enough to hold the entire content of the array ***scores***, and in our example, it needs to hold three double type variables, since the array size is 3, the each array element is a double type. In Figure 3, we use a random number 0x003D042C to represent the address being allocated. As programmer, we don’t need to know what is the exact address of the allocated memory block, but we need to know that this memory block is in the heap, and the memory size is big enough to hold the body of an entire array.
3. the address (reference) of the block of memory allocated in heap in step 2 is assigned to the value of the reference variable allocated in step 1, in our example, it is the ***scores*** variable in the left hand side of the assignment operator. For the scenario of address allocation illustrated in figure 3, the address value of 0x003D042C is written into the value filed of variable ***scores***.

You can view the above steps as the screen shots of a movie and you need remember what exactly happen in each step. The entire scenes of this movie should impress into your mind permanently.

Are we done with the array initialization yet? Not yet, because the content of the array in the heap has not be determined yet, so we need to the following statements to initialize the array content in the heap.

scores[0] = 301.59;

scores[1] = 72.6;

scores[2] = 196.68;

After the above initialization statements have been executed, the heap has the corresponding data value being set, as indicated in Figure 3 above. If the array size is big, we may need to use a loop to initialize each element, instead of hardcoding for each element in separate line.

**Attention:** about the **“flat”** property of reference variable. When people use the word “flat” to describe reference variable, it means that all reference variable have the same size: 4 types (32 bits) in a 32-bit system (of course it will be 8 bytes (64 bits) in a 64-bit system), and the size does not vary with the actual size of the array being referred to. This is obvious, since the memory block in heap has the actual size of the array content, but the reference variable located in stack only records the starting address (reference) of the memory block in heap. Even the actual array size varies from array to array, but the reference variables of different arrays are the same in size, and they are all “flat” down to 32 bits (4 bytes).

Having mentioned the difference between variable of built-in data type and variable of array, let’s talk about parameter passing in Java. This is the rule you need to remember for passing parameters in Java: **Parameter passing in java is pass-by-value for both built-in data type and array variable.**

However, there is difference between parameter passing for variable of built-in data type and array type, as explained below:

**For variable of built-in data, its value is the actual data value**, thus the actual value is passed, and a copy of the value is created in the invoked method’s calling stack. As a result, no matter what action the invoked method performs to the passed value (the formal parameter), the action will not affect the original variable in the invoking method (the actual parameter).

**For array variable, its value is the address (reference)** of the block of memory in the heap where the body of the array is stored, thus the actual value of the variable (the address, i.e., reference) is copied from the stack of the invoking method to the stack of the invoked method. The consequence is: the invoked method can changed the content of the original array in heap, since in the invoked method, the passed parameter refers to the same block of the memory in the heap where the array body locates, and this passed reference variable has the same power of changing the heap content as the original reference variable in the invoking method. Here is an example source code for method invoking and parameter passing in Java:

public class PassingParameterExampleForArray{

public static void main(String[] args){

int myIntVar = 10;

double[] scores = new double[3]; // declaration and creation of array

scores[0] = 301.59; // now is the initialization of the array.

scores[1] = 72.6;

scores[2] = 196.68;

System.out.println("in main, before processing:");

System.out.printf("myIntVar:%5d; array: %6.2f %6.2f %6.2f\n\n",

myIntVar, scores[0], scores[1], scores[2]);

process(myIntVar, scores);

System.out.println("in main, after processing:");

System.out.printf("myIntVar:%5d; array: %6.2f %6.2f %6.2f\n\n",

myIntVar, scores[0], scores[1], scores[2]);

}

public static void process(int myInt, double[] list){

myInt = 20; // change the value of parameter of built-in data type

for(int i=0; i<list.length; i++){

list[i] = list[i]/2; // change the array content

}

System.out.println("inside processing:");

System.out.printf("myIntVar:%5d; array: ", myInt);

for(int i=0; i<list.length; i++){

System.out.printf("%6.2f ", list[i]);

}

System.out.print("\n\n");

}

}

The stack and heap map for the above program is illustrated below:

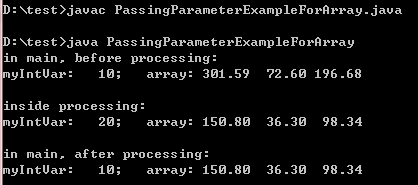
memory data/content

|  |  |
| --- | --- |
| memory address |  |
|  |  |
| 0x 03A8EC20 | 301.59 to 150.80 : scores[0]  72.6 to 36.30 : scores[1]  196.68 to 98.34 : scores[2] |
|  |
|  | above is heap; below is stack |
|  | below is the stack of process() |
|  |  |
| refers to | list : 0x03A8EC20  value copy of parameter of array name |
|  | myInt : 10, then changed to 20  below is the stack of main() |
|  |  |
| refers to | scores : 0x03A8EC20  value copy of parameter of built-in type |
|  | myIntVar : 10 |

Figure 4 the parameters are pass-by-value when invoking method in Java

Notice that in Figure 4, I used a different address value than the address value that I used in Figure 3, because each new statement will return an address (reference) determined by the memory management scheme of the virtual machine, and the program source code has no control over what address value that is returned.

After compiling and running the above program in command line (or you can use Eclipse to run it), we can see the following output:



Observations for the above output include:

1. **For the built-in data type parameter myIntVar,** its value was changed inside the process() method, but the change **DOES NOT carry back** to the main method. The reason is that, the changed made inside the process() method happened to the parameter of myInt variable of the process() method, which occupied a different memory space in the stack than the myIntVar variable of the main() method in the stack, and you can easily verify this by looking at the stack and heap map in Figure 4, or the output screen shot above. A value change in one memory place will NOT affect the memory date value in another memory place.
2. **For array name, a reference type parameter,** its reference value is copied from actual parameter ***scores*** to formal parameter ***list***, thus they point to the same memory location in the heap. The change made to the array body inside the invoked method **DOES carry back** to the invoking method, because both reference variables refers to (points to) the same block of memory in the heap. In other words, the changed made in the process() method happens in the heap, thus the changes carried back.

In this semester, we are going to use the stack and heap map frequently to help illustrate the source code, and you are required to know how to draw memory map of stack and heap, similar to the above figures. Be prepared.