7.7. Arrays

Recall that arrays are ordered collections of data elements of the same type that are contiguously stored in memory. Statically allocated single-dimension arrays have the form `Type arr[N]` where `Type` is the data type, `arr` is the identifier associated with the array, and `N` is the number of data elements. Declaring an array statically as `Type arr[N]` or dynamically as `arr = malloc(N * sizeof(Type))` allocates *N* x sizeof(*Type*) total bytes of memory.

To access the element at index *i* in array `arr`, use the syntax `arr[i]`. Compilers commonly convert array references into pointer arithmetic prior to translating to assembly. So, `arr+i` is equivalent to `&arr[i]`, and `*(arr+i)` is equivalent to `arr[i]`. Since each data element in `arr` is of type `Type`, `arr+i` implies that element *i* is stored at address `arr + sizeof(Type) * i`.

Table 1 outlines some common array operations and their corresponding assembly instructions. In the examples that follow, suppose that we declare an `int` array of length 10 (`int arr[10]`). Assume that register `%rdx` stores the address of `arr`, register `%rcx` stores the `int` value `i`, and register `%rax` represents some variable `x` (also of type `int`). Recall that `int` variables take up four bytes of space, whereas `int *` variables take up eight bytes of space.

Table 1. Common Array Operations and Their Corresponding Assembly Representations

Operation	Type	Assembly Representation
`x = arr`	`int *`	`mov %rdx, %rax`
`x = arr[0]`	`int`	`mov (%rdx), %eax`
`x = arr[i]`	`int`	`mov (%rdx, %rcx,4), %eax`
`x = &arr[3]`	`int *`	`lea 0xc(%rdx), %rax`
`x = arr+3`	`int *`	`lea 0xc(%rdx), %rax`
`x = *(arr+5)`	`int`	`mov 0x14(%rdx), %eax`

Pay close attention to the *type* of each expression in Table 1. In general, the compiler uses `mov` instructions to dereference pointers and the `lea` instruction to compute addresses.

Notice that to access element `arr[3]` (or `*(arr+3)` using pointer arithmetic), the compiler performs a memory lookup on address `arr+3*4` instead of `arr+3`. To understand why this is necessary, recall that any element at index *i* in an array is stored at address `arr + sizeof(Type) * i`. The compiler must therefore multiply the index by the size of the data type (in this case four, since `sizeof(int) = 4`) to compute the correct offset. Recall also that memory is byte-addressable; offsetting by the correct number of bytes is the same as computing an address. Lastly, because `int` values require only 4 bytes of space, they are stored in component register `%eax` of register `%rax`.

As an example, consider a sample array (`array`) with 10 integer elements (Figure 1):

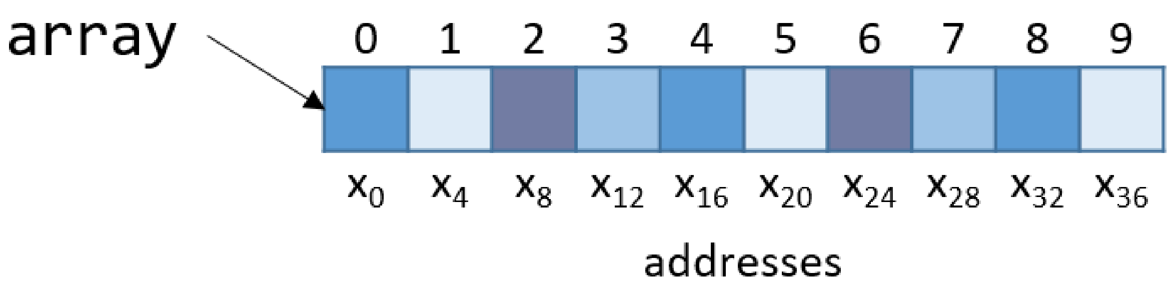

Figure 1. The layout of a 10-integer array in memory. Each x_i-labeled box represents four bytes.

Notice that since `array` is an array of integers, each element takes up exactly four bytes. Thus, an integer array with 10 elements consumes 40 bytes of contiguous memory.

To compute the address of element 3, the compiler multiplies the index 3 by the data size of the integer type (4) to yield an offset of 12 (or 0xc). Sure enough, element 3 in Figure 1 is located at byte offset x_{12}.

Let's take a look at a simple C function called `sumArray` that sums up all the elements in an array:

```c
int sumArray(int *array, int length) {
    int i, total = 0;
    for (i = 0; i < length; i++) {
        total += array[i];
    }
    return total;
}
```

The `sumArray` function takes the address of an array and the array's associated length and sums up all the elements in the array. Now take a look at the corresponding assembly for the `sumArray` function:

```
0x400686 <+0>:  push %rbp                     # save %rbp
0x400687 <+1>:  mov  %rsp,%rbp                # update %rbp (new stack
frame)
0x40068a <+4>:  mov  %rdi,-0x18(%rbp)         # copy array to %rbp-0x18
0x40068e <+8>:  mov  %esi,-0x1c(%rbp)         # copy length to %rbp-0x1c
```

```
0x400691 <+11>: movl $0x0,-0x4(%rbp)        # copy 0 to %rbp-0x4 (total)
0x400698 <+18>: movl $0x0,-0x8(%rbp)        # copy 0 to %rbp-0x8 (i)
0x40069f <+25>: jmp  0x4006be <sumArray+56> # goto <sumArray+56>
0x4006a1 <+27>: mov  -0x8(%rbp),%eax        # copy i to %eax
0x4006a4 <+30>: cltq                        # convert i to a 64-bit
integer
0x4006a6 <+32>: lea  0x0(,%rax,4),%rdx      # copy i*4 to %rdx
0x4006ae <+40>: mov  -0x18(%rbp),%rax       # copy array to %rax
0x4006b2 <+44>: add  %rdx,%rax              # compute array+i*4, store in
%rax
0x4006b5 <+47>: mov  (%rax),%eax            # copy array[i] to %eax
0x4006b7 <+49>: add  %eax,-0x4(%rbp)        # add %eax to total
0x4006ba <+52>: addl $0x1,-0x8(%rbp)        # add 1 to i (i+=1)
0x4006be <+56>: mov  -0x8(%rbp),%eax        # copy i to %eax
0x4006c1 <+59>: cmp  -0x1c(%rbp),%eax       # compare i to length
0x4006c4 <+62>: jl   0x4006a1 <sumArray+27> # if i<length goto
<sumArray+27>
0x4006c6 <+64>: mov  -0x4(%rbp),%eax        # copy total to %eax
0x4006c9 <+67>: pop  %rbp                   # prepare to leave the
function
0x4006ca <+68>: retq                        # return total
```

When tracing this assembly code, consider whether the data being accessed represents an address or a value. For example, the instruction at `<sumArray+11>` results in `%rbp-0x4` containing a variable of type `int`, which is initially set to 0. In contrast, the argument stored at `%rbp-0x18` is the first argument to the function (`array`) which is of type `int *` and corresponds to the base address of the array. A different variable (which we call `i`) is stored at location `%rbp-0x8`. Lastly, note that size suffixes are included at the end of instructions like `add` and `mov` only when necessary. In cases where constant values are involved, the compiler needs to explicitly state how many bytes of the constant are being moved.

The astute reader will notice a previously unseen instruction at line `<sumArray+30>` called `cltq`. The `cltq` instruction stands for "convert long to quad" and converts the 32-bit `int` value stored in `%eax` to a 64-bit integer value that is stored in `%rax`. This operation is necessary because the instructions that follow perform pointer arithmetic. Recall that on 64-bit systems, pointers take up 8 bytes of space. The compiler's use of `cltq` simplifies the process by ensuring that all data are stored in 64-bit registers instead of 32-bit components.

Let's take a closer look at the five instructions between locations `<sumArray+32>` and `<sumArray+49>`:

```
<+32>: lea 0x0(,%rax,4),%rdx        # copy i*4 to %rdx
<+40>: mov -0x18(%rbp),%rax         # copy array to %rax
<+44>: add %rdx,%rax                # add i*4 to array (i.e. array+i) to
%rax
<+47>: mov (%rax),%eax              # dereference array+i*4, place in %eax
<+49>: add %eax,-0x4(%rbp)          # add %eax to total (i.e.
total+=array[i])
```

Recall that the compiler commonly uses `lea` to perform simple arithmetic on operands. The operand `0x0(,%rax,4)` translates to `%rax*4 + 0x0`. Since `%rax` holds the value of `i`, this operation copies the value `i*4` to `%rdx`. At this point, `%rdx` contains the number of bytes to calculate the correct offset of `array[i]` (recall that `sizeof(int)` = 4).

The next instruction (`mov -0x18(%rbp), %rax`) copies the first argument to the function (the base address of `array`) into register `%rax`. Adding `%rdx` to `%rax` in the next instruction causes `%rax` to contain `array+i*4`. Recall that the element at index *i* in `array` is stored at address `array + sizeof(T) * i`. Therefore, `%rax` now contains the assembly-level computation of the address `&array[i]`.

The instruction at `<sumArray+47>` *dereferences* the value located at `%rax`, placing the value of `array[i]` into `%eax`. Notice the use of the component register `%eax`, since `array[i]` contains a 32-bit `int` value! In contrast, the variable `i` was changed to a quad-word on line `<sumArray+30>` since `i` was about to be used for *address computation*. Again, addresses are stored as 64-bit words.

Lastly, `%eax` is added to the value in `%rbp-0x4`, or `total`. Therefore, the five instructions between locations `<sumArray+22>` and `<sumArray+39>` correspond to the line `total += array[i]` in the `sumArray` function.

Contents