Jonathan Parlett

May 30, 2022

Memory and Pointers

What were going to talk about

- Simplified model of memory as a table.

What were going to talk about

- Simplified model of memory as a table.
- Pointers and the operations performed on them.

What were going to talk about

- Simplified model of memory as a table.
- Pointers and the operations performed on them.
    1. Address of (&)

What were going to talk about

- Simplified model of memory as a table.
- Pointers and the operations performed on them.
    1. Address of (&)
    2. Dereference ()

What were going to talk about

- Simplified model of memory as a table.
- Pointers and the operations performed on them.
    1. Address of (&)
    2. Dereference ()
- Arrays and pointer addition

What were going to talk about

- Simplified model of memory as a table.
- Pointers and the operations performed on them.
    1. Address of (&)
    2. Dereference ()
- Arrays and pointer addition
- The type system and the sizes of types

What were going to talk about

- Simplified model of memory as a table.
- Pointers and the operations performed on them.
    1. Address of (&)
    2. Dereference ()
- Arrays and pointer addition
- The type system and the sizes of types
- Pointer Addition (+)

- We can think of the memory allocated to a program as a collection of cells that hold data items. Such as the integer $i = 55$, or the character $c =$'a'.

Memory as a Table

- We can think of the memory allocated to a program as a collection of cells that hold data items. Such as the integer $i = 55$, or the character $c =$'a'.
-

Data
0
0
55
0
'a'

Memory as a Table

- These memory locations are in some physical order. So there is a lowest location and a highest location. We will label them in this order from lowest to highest taking 0 to be the lowest and n to be the highest.

Memory as a Table

- These memory locations are in some physical order. So there is a lowest location and a highest location. We will label them in this order from lowest to highest taking 0 to be the lowest and *n* to be the highest.
Address	Data
0	0
1	0
2	55
3	0
4	'a'
⋮	⋮
n	13

Memory as a Table

- Finally we know the stack starts at the lowest address and the heap starts at the highest.

- Finally we know the stack starts at the lowest address and the heap starts at the highest.
-

Stack	
Address	Data
0	0
1	0
2	55
3	0
4	'a'
⋮	⋮
n	13
Heap	

- We will say a row in our table can contain a single variable of any type. So it can contain a single int, char, unsigned long int, etc.

Memory as a Table

- We will say a row in our table can contain a single variable of any type. So it can contain a single int, char, unsigned long int, etc.
- Those of you who are more familiar with C and memory will notice that we are ignoring the size of the type in our table model of memory. If you don't understand this don't be concerned we will go into more detail later.

Memory as a Table

- We will say a row in our table can contain a single variable of any type. So it can contain a single int, char, unsigned long int, etc.
- Those of you who are more familiar with C and memory will notice that we are ignoring the size of the type in our table model of memory. If you don't understand this don't be concerned we will go into more detail later.
- We will sometimes omit labels such the Stack and Heap to save space on a slide.

Memory as a Table

- We will say a row in our table can contain a single variable of any type. So it can contain a single int, char, unsigned long int, etc.
- Those of you who are more familiar with C and memory will notice that we are ignoring the size of the type in our table model of memory. If you don't understand this don't be concerned we will go into more detail later.
- We will sometimes omit labels such the Stack and Heap to save space on a slide.
- This is a fairly precise model of memory in a computer. Each variable or data item is stored in a memory location. That location has an address which is just a index in our table.

Memory as a Table

- We will say a row in our table can contain a single variable of any type. So it can contain a single int, char, unsigned long int, etc.
- Those of you who are more familiar with C and memory will notice that we are ignoring the size of the type in our table model of memory. If you don’t understand this don’t be concerned we will go into more detail later.
- We will sometimes omit labels such the Stack and Heap to save space on a slide.
- This is a fairly precise model of memory in a computer. Each variable or data item is stored in a memory location. That location has an address which is just a index in our table.
- It should be noted you will probably never work with memory addresses this small. Most of the addresses you will work with will be large numbers best represented with hexadecimal notation.

Pointers

- The basic idea of a pointer is something that tells you where something else is.

Pointers

- The basic idea of a pointer is something that tells you where something else is.
- Consider Bob. Bob points to McDonalds.

- The basic idea of a pointer is something that tells you where something else is.
- Consider Bob. Bob points to McDonalds.
- In effect if you know where Bob is you can look where he is pointing to find McDonalds.

Pointers

- The basic idea of a pointer is something that tells you where something else is.
- Consider Bob. Bob points to McDonalds.
- In effect if you know where Bob is you can look where he is pointing to find McDonalds.
- Then it would be reasonable to call Bob a pointer to McDonalds.

Consider our memory table and the variable

```c
int i = 49 //located at address 1
```

Stack	
Address	Data
0	0
1	49
⋮	⋮
n	13
Heap	

- A pointer in C tells us where a variable or data item is located. To locate *i* we just need its address 1.

Consider our memory table and the variable

```c
int i = 49 //located at address 1
```

Stack	
Address	Data
0	0
1	49
⋮	⋮
n	13
Heap	

- A pointer in C tells us where a variable or data item is located. To locate *i* we just need its address 1.
- So thats all a pointer is. Just another variable that holds an address (an index to our table).

Consider our memory table and the variable

```c
int i = 49 //located at address 1
```

Stack	
Address	Data
0	0
1	49
⋮	⋮
n	13
Heap	

- A pointer in C tells us where a variable or data item is located. To locate *i* we just need its address 1.
- So thats all a pointer is. Just another variable that holds an address (an index to our table).
- Since an address is just a number, a pointer is just a number. A pointer to *i* would have the value 1.

```
int i = 49 //located at address 1
int *j = 1 //pointer to i, located at address 2
```

Stack	
Address	Data
0	0
1	49
2	1
⋮	⋮
n	13
Heap	

- Since pointers are also variables they to are stored somewhere in our table.

Pointers

```
int i = 49 //located at address 1
int *j = 1 //pointer to i, located at address 2
```

Stack	
Address	Data
0	0
1	49
2	1
⋮	⋮
n	13
Heap	

- Since pointers are also variables they to are stored somewhere in our table.
- When we declare them they are statically allocated integers so they are stored on the stack.

Pointers

```
int i = 49 //located at address 1
int *j = 1 //pointer to i, located at address 2
```

Stack	
Address	Data
0	0
1	49
2	1
⋮	⋮
n	13
Heap	

- Since pointers are also variables they to are stored somewhere in our table.
- When we declare them they are statically allocated integers so they are stored on the stack.
- *j* is stored at location 2. It is a pointer to *i* so its value is the address of *i* namely 1.

Pointer Operations: Address Of (&)

- The address of operator is applicable to any type in C not just pointers. It returns the memory address of the data item you use it on.

- The address of operator is applicable to any type in C not just pointers. It returns the memory address of the data item you use it on.
- ```c
  int i = 49 //located at address 1
  int *j = 1 //pointer to i, located at address 2
  ```

Stack	
Address	Data
0	0
1	49
2	1
⋮	⋮
  ```

- The address of operator is applicable to any type in C not just pointers. It returns the memory address of the data item you use it on.
- ```c
  int i = 49 //located at address 1
  int *j = 1 //pointer to i, located at address 2
  ```

Stack	
Address	Data
0	0
1	49
2	1
⋮	⋮

- Given the table from before

```c
printf("%d\n", &i); //prints 1
printf("%d\n", &j); //prints 2
```
```

Pointer Operations: Address Of (&)

Since a pointer is just a memory address we can say (&) returns a pointer to the item you use it on. This allows us to do things like.

```
int i = 49;
int *j = &i;
```

Pointer Operations: Dereference (*)

- When you dereference a pointer you are accessing the value at the location specified by the pointer.

Pointer Operations: Dereference (*)

- When you dereference a pointer you are accessing the value at the location specified by the pointer.
- ```
  int i = 49 //located at address 1
  int *j = 1 //pointer to i, located at address 2
  ```

Address	Data
0	0
1	49
2	1
⋮	⋮
  ```

Pointer Operations: Dereference (*)

- When you dereference a pointer you are accessing the value at the location specified by the pointer.
- ```
  int i = 49 //located at address 1
  int *j = 1 //pointer to i, located at address 2
  ```

Address	Data
0	0
1	49
2	1
⋮	⋮

- Considering the table from before and the pointer *j* which points to *i* when we dereference *j* we can essentially replace $*j$ with *i*.

```
printf("value at j: %d\n", *j); //prints 49
*j = 69;
printf("value at j: %d\n", *j); //prints 69
```
```

Arrays are Pointers

- Pointer addition is closely related to how arrays function, so we'll explain arrays before we define it.

Arrays are Pointers

- Pointer addition is closely related to how arrays function, so we'll explain arrays before we define it.
- Arrays are one of the simplest data structures. They are simply a collection of contiguous memory locations that contain items of the same type.

- A statically declared array of integers
  `int A[3] = {4,5,6}`
  would be stored on the stack like so.

- A statically declared array of integers
  ```
  int A[3] = {4,5,6}
  ```
  would be stored on the stack like so.

Stack	
Address	Data
0	0
1	4
2	5
3	6
⋮	⋮

-

- A statically declared array of integers
  ```
  int A[3] = {4,5,6}
  ```
  would be stored on the stack like so.
Stack	
Address	Data
0	0
1	4
2	5
3	6
⋮	⋮
- The variable *A* would be called the *base pointer* of the array, and its value is the address of the first element, in this case address 1.

Arrays are Pointers: Pointer Addition

- If we dereference *A* we get the first element namely 4.

```
*A == 4;
```

Arrays are Pointers: Pointer Addition

- If we dereference *A* we get the first element namely 4.

```
*A == 4;
```

- This is equivalent to array indexing using the zero index.

```
*A == A[0]; //both equal 4
```

Arrays are Pointers: Pointer Addition

- If we dereference A we get the first element namely 4.

  ```
  *A == 4;
  ```

- This is equivalent to array indexing using the zero index.

  ```
  *A == A[0]; //both equal 4
  ```

- To access more elements we can add to A. $A + 1$ references the second item at location 2 namely 5. Dereferencing gives us the item at location 2, namely 5.

  ```
  *(A+1) == A[1]; //both equal 5
  ```

Arrays are Pointers: Pointer Addition

- If we dereference A we get the first element namely 4.

```
*A == 4;
```

- This is equivalent to array indexing using the zero index.

```
*A == A[0]; //both equal 4
```

- To access more elements we can add to A. $A + 1$ references the second item at location 2 namely 5. Dereferencing gives us the item at location 2, namely 5.

```
*(A+1) == A[1]; //both equal 5
```

- So array indexing is really just shorthand for pointer addition + dereference.

```
*(A+i) == A[i];
```

Lets go over this again with a code example.

```
int A[3] = {4,5,6};

printf("Val of base pointer A = 0x%x\n", A);
printf("____________________________________");
for(int i=0; i < 3; i++){
        printf("*(A+%d) = %d | ", i, *(A+i));
        printf("A[%d] = %d | ", i, A[i]);
        printf("Address of value at (A+%d) = 0x%x\n", i, (A+i));
}
```

Arrays are Pointers: Pointer Addition

Lets go over this again with a code example.

- ```
  int A[3] = {4,5,6};

  printf("Val of base pointer A = 0x%x\n", A);
  printf("_____________________________________");
  for(int i=0; i < 3; i++){
          printf("*(A+%d) = %d | ", i, *(A+i));
          printf("A[%d] = %d | ", i, A[i]);
          printf("Address of value at (A+%d) = 0x%x\n", i, (A+i));
  }
  ```
- **Output**

  ```
  Val of base pointer A = 0x8848ab1c

  ____________________________________
  *(A+0) = 4 | A[0] = 4 | Address of value at (A+0) = 0x8848ab1c
  *(A+1) = 5 | A[1] = 5 | Address of value at (A+1) = 0x8848ab20
  *(A+2) = 6 | A[2] = 6 | Address of value at (A+2) = 0x8848ab24
  ```
  ```

Lets go over this again with a code example.

- ```c
  int A[3] = {4,5,6};

  printf("Val of base pointer A = 0x%x\n", A);
  printf("____________________________________");
  for(int i=0; i < 3; i++){
          printf("*(A+%d) = %d | ", i, *(A+i));
          printf("A[%d] = %d | ", i, A[i]);
          printf("Address of value at (A+%d) = 0x%x\n", i, (A+i));
  }
  ```
- **Output**

  ```
  Val of base pointer A = 0x8848ab1c

  ____________________________________
  *(A+0) = 4 | A[0] = 4 | Address of value at (A+0) = 0x8848ab1c
  *(A+1) = 5 | A[1] = 5 | Address of value at (A+1) = 0x8848ab20
  *(A+2) = 6 | A[2] = 6 | Address of value at (A+2) = 0x8848ab24
  ```
- Everything does what you would expect it to, except the value of the base pointer increases in increments of 4 instead of 1 each time.
  ```

The Type System

To understand why this happens we need to understand a little bit about the type system and the sizes of different types.

- The atomic unit of memory in C is the **byte** (8 bits).

To understand why this happens we need to understand a little bit about the type system and the sizes of different types.

- The atomic unit of memory in C is the **byte** (8 bits).
- Each type in C takes up a certain amount of **bytes** of memory.

The Type System

To understand why this happens we need to understand a little bit about the type system and the sizes of different types.

- The atomic unit of memory in C is the **byte** (8 bits).
- Each type in C takes up a certain amount of **bytes** of memory.
- Each of the cells in our current table actually correspond to a certain amount of bytes, based on the type stored there.

The sizeof function

- The function *sizeof()* takes either a type or variable as an argument.

The sizeof function

- The function *sizeof()* takes either a type or variable as an argument.
- Given a type *sizeof()* returns the size of the type in bytes

The sizeof function

- The function *sizeof()* takes either a type or variable as an argument.
- Given a type *sizeof()* returns the size of the type in bytes
- Given a variable it returns the size of the type of the variable in bytes.

The sizeof function

- The function *sizeof()* takes either a type or variable as an argument.
- Given a type *sizeof()* returns the size of the type in bytes
- Given a variable it returns the size of the type of the variable in bytes.
- It does **not** return the length of an array.

Lets look at the size of some types. Please be aware that sizes may be different on different hardware platforms, so these may not be same on your system.

- Well check the sizes of types using the code below.

Lets look at the size of some types. Please be aware that sizes may be different on different hardware platforms, so these may not be same on your system.

- Well check the sizes of types using the code below.
- The size of an *int* is 4 bytes

```
printf("Size of integer = %lu\n", sizeof(int));
```

Lets look at the size of some types. Please be aware that sizes may be different on different hardware platforms, so these may not be same on your system.

- Well check the sizes of types using the code below.
- The size of an *int* is 4 bytes
  ```
  printf("Size of integer = %lu\n", sizeof(int));
  ```
- The size of a *char* is 1 byte
  ```
  printf("Size of char = %lu\n", sizeof(char));
  ```

Lets look at the size of some types. Please be aware that sizes may be different on different hardware platforms, so these may not be same on your system.

- Well check the sizes of types using the code below.
- The size of an *int* is 4 bytes
  ```
  printf("Size of integer = %lu\n", sizeof(int));
  ```
- The size of a *char* is 1 byte
  ```
  printf("Size of char = %lu\n", sizeof(char));
  ```
- The size of any pointer is 8 bytes, no matter the type it points to.
  ```
  printf("Size of int* = %lu\n", sizeof(int*));
  printf("Size of char* = %lu\n", sizeof(char*));
  ```

Pointer Addition (+)

Now lets look more closely at adding to a pointer with a code example.

```
int* i=0;
int j=0;

printf("Value of Int ptr: 0x%x\n",i);
printf("Value of Int: 0x%x\n",j);

i += 2; //increment ptr
j += 2*sizeof(int); //increment integer

printf("Value of Int ptr: 0x%x\n",i);
printf("Value of Int: 0x%x\n",j);
```

Output

```
Value of Int ptr: 0
Value of Int: 0
Value of Int ptr: 8
Value of Int: 8
```

Pointer Addition (+)

Now lets be clear about how pointer addition is actually defined. If a pointer *i* is equivalent to a number *j* then adding to *i* is equivalent to adding the same thing to *j* times the size of the type that *i* points to.

- $int\ j = x,\ x \in \mathbb{N}$

Pointer Addition (+)

Now lets be clear about how pointer addition is actually defined. If a pointer i is equivalent to a number j then adding to i is equivalent to adding the same thing to j times the size of the type that i points to.

- $int\ j = x,\ x \in \mathbb{N}$
- $int * i = j$

Pointer Addition (+)

Now lets be clear about how pointer addition is actually defined. If a pointer *i* is equivalent to a number *j* then adding to *i* is equivalent to adding the same thing to *j* times the size of the type that *i* points to.

- $int\ j = x,\ x \in \mathbb{N}$
- $int * i = j$
- $i + 1 \iff j + 1 * sizeof(int)$

Pointer Addition (+)

Now lets be clear about how pointer addition is actually defined. If a pointer i is equivalent to a number j then adding to i is equivalent to adding the same thing to j times the size of the type that i points to.

- $int\ j = x,\ x \in \mathbb{N}$
- $int * i = j$
- $i + 1 \iff j + 1 * sizeof(int)$
- More generally $i + x \iff j + x * sizeof(int)$

Pointer Addition (+)

Now lets be clear about how pointer addition is actually defined. If a pointer i is equivalent to a number j then adding to i is equivalent to adding the same thing to j times the size of the type that i points to.

- $int\ j = x,\ x \in \mathbb{N}$
- $int * i = j$
- $i + 1 \iff j + 1 * sizeof(int)$
- More generally $i + x \iff j + x * sizeof(int)$
- This is why the array pointer A increased in increments of 4 since the size of the **int** data type is 4.

Pointer Addition (+)

Now that we are no longer ignoring the sizes of types lets take our original table representing the array [3,4,6] and redraw it accurately.

Stack	
Address	Data
0	0
1	4
2	5
3	6
⋮	⋮

Pointer Addition (+)

Now that we are no longer ignoring the sizes of types lets take our original table representing the array [3,4,6] and redraw it accurately.

Stack	
Address	Data
0	0
1	4
2	5
3	6
⋮	⋮

- Each row actually corresponds to 4 bytes for each number in our array so well need 12 cells to represent our 3 numbers.

Pointer Addition (+)

```
int A[3] = {4,5,6}
```

<table>
<tr><th colspan="2">Stack</th></tr>
<tr><th>Address</th><th>Data</th></tr>
<tr><td>0</td><td>0</td></tr>
<tr><td>1</td><td rowspan="4">4</td></tr>
<tr><td>2</td></tr>
<tr><td>3</td></tr>
<tr><td>4</td></tr>
<tr><td>5</td><td rowspan="4">5</td></tr>
<tr><td>6</td></tr>
<tr><td>7</td></tr>
<tr><td>8</td></tr>
<tr><td>9</td><td rowspan="4">6</td></tr>
<tr><td>10</td></tr>
<tr><td>11</td></tr>
<tr><td>12</td></tr>
<tr><td>⋮</td><td>⋮</td></tr>
</table>

Pointer Addition (+)

- The table is almost the same *A* still equals 1, but what are the implications for addition and dereference.

Pointer Addition (+)

- The table is almost the same *A* still equals 1, but what are the implications for addition and dereference.
- When you are dereferencing *A* you are actually referencing the 4 bytes starting at the address of *A*.

Pointer Addition (+)

- The table is almost the same *A* still equals 1, but what are the implications for addition and dereference.
- When you are dereferencing *A* you are actually referencing the 4 bytes starting at the address of *A*.
- When you are adding 1 to *A* you are actually adding 4, so *A* points to the next integer in the array.

Pointer Addition (+)

- The table is almost the same *A* still equals 1, but what are the implications for addition and dereference.
- When you are dereferencing *A* you are actually referencing the 4 bytes starting at the address of *A*.
- When you are adding 1 to *A* you are actually adding 4, so *A* points to the next integer in the array.
- Lets draw this out in more detail.

Thanks for watching

Hopefully you have gained a solid understanding of pointers and their operations. Thanks for watching!