Jonathan Parlett

June 11, 2022

Memory and Pointers

- 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 (+)

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 (+)
- Memory management

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 (+)
- Memory management
    1. Malloc (How we request memory)

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 (+)
- Memory management
    1. Malloc (How we request memory)
    2. Free (How we return memory)

- Lets start with how we will represent memory visually as table.

Memory as a Table

- Lets start with how we will represent memory visually as 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'.

Memory as a Table

- Lets start with how we will represent memory visually as 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
n	0
⋮	⋮
4	0
3	55
2	0
1	'a'
0	13

Memory as a Table

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

Memory as a Table

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

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

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

Stack	
Address	Data
n	0
$\vdots$	$\vdots$
4	0
3	55
2	0
1	'a'
0	13
Heap	

- We will take $n = 64$ so we can work with a manageable memory space for the rest of the presentation.

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.

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

- Moving to pointers lets first explain the basic concept.

Pointers

- Moving to pointers lets first explain the basic concept.
- The basic idea of a pointer is something that tells you where something else is.

Pointers

- Moving to pointers lets first explain the basic concept.
- The basic idea of a pointer is something that tells you where something else is.
- Consider Bob. Bob points to McDonalds.

Pointers

- Moving to pointers lets first explain the basic concept.
- 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

- Moving to pointers lets first explain the basic concept.
- 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 63
```

Stack	
Address	Data
64	0
63	49
⋮	⋮
0	13
Heap	

- A pointer in C tells us where a variable or data item is located. To locate *i* we just need its address 63.

Consider our memory table and the variable

```c
int i = 49 //located at address 63
```

Stack	
Address	Data
64	0
63	49
⋮	⋮
0	13
Heap	

- A pointer in C tells us where a variable or data item is located. To locate *i* we just need its address 63.
- 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 63
```

Stack	
Address	Data
64	0
63	49
⋮	⋮
0	13
Heap	

- A pointer in C tells us where a variable or data item is located. To locate *i* we just need its address 63.
- 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 63.

```
int i = 49 //located at address 63
int *j = 63 //pointer to i, located at address 62
```

Stack	
Address	Data
64	0
63	49
62	63
⋮	⋮
0	13
Heap	

- In this instance *j* is Bob and *i* is McDonalds.

```
int i = 49 //located at address 63
int *j = 63 //pointer to i, located at address 62
```

Stack	
Address	Data
64	0
63	49
62	63
⋮	⋮
0	13
Heap	

- In this instance *j* is Bob and *i* is McDonalds.
- Since pointers are also variables they to are stored somewhere in our table.

```
int i = 49 //located at address 63
int *j = 63 //pointer to i, located at address 62
```

Stack	
Address	Data
64	0
63	49
62	63
⋮	⋮
0	13
Heap	

- In this instance *j* is Bob and *i* is McDonalds.
- 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 63
int *j = 63 //pointer to i, located at address 62
```

Stack	
Address	Data
64	0
63	49
62	63
⋮	⋮
0	13
Heap	

- In this instance *j* is Bob and *i* is McDonalds.
- 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 62. It is a pointer to *i* so its value is the address of *i* namely 63.

Pointer Operations: Address Of (&)

- Moving to pointer operations lets start with the address of operator (&).

Pointer Operations: Address Of (&)

- Moving to pointer operations lets start with the address of operator (&).
- 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.

Pointer Operations: Address Of (&)

- Moving to pointer operations lets start with the address of operator (&).
- 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 63
  int *j = 63 //pointer to i, located at address 62
  ```

Stack	
Address	Data
64	0
63	49
62	63
⋮	⋮
0	13
  ```

- Moving to pointer operations lets start with the address of operator (&).
- 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 63
  int *j = 63 //pointer to i, located at address 62
  ```

Stack	
Address	Data
64	0
63	49
62	63
⋮	⋮
0	13

- Given the table from before
  ```c
  printf("%d\n", &i); //prints 63
  printf("%d\n", &j); //prints 62
  ```
  ```

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; //j = 63 since index of i is 63
```

This is a more practical way to get a pointer to a statically declared variable.

Pointer Operations: Dereference (*)

- Continuing with pointer operations lets talk about dereference (*).

Pointer Operations: Dereference (*)

- Continuing with pointer operations lets talk about dereference (*).
- When you dereference a pointer you are accessing the value at the location specified by the pointer.

Pointer Operations: Dereference (*)

- Continuing with pointer operations lets talk about dereference (*).
- When you dereference a pointer you are accessing the value at the location specified by the pointer.
- ```
  int i = 49 //located at address 63
  int *j = 63 //pointer to i, located at address 62
  ```

Stack	
Address	Data
64	0
63	49
62	63
⋮	⋮
  ```

Pointer Operations: Dereference (*)

- Continuing with pointer operations lets talk about dereference (*).
- When you dereference a pointer you are accessing the value at the location specified by the pointer.
- ```
  int i = 49 //located at address 63
  int *j = 63 //pointer to i, located at address 62
  ```

Stack	
Address	Data
64	0
63	49
62	63
⋮	⋮

- 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
```
```

- Continuing our discussion of pointer operations we would like to discuss pointer addition (+), but there are some other topics we must cover first to fully explain it.

Arrays are Pointers

- Continuing our discussion of pointer operations we would like to discuss pointer addition (+), but there are some other topics we must cover first to fully explain it.
- The functionality of Arrays is closely related to pointer addition so well explore them first.

Arrays are Pointers

- Continuing our discussion of pointer operations we would like to discuss pointer addition (+), but there are some other topics we must cover first to fully explain it.
- The functionality of Arrays is closely related to pointer addition so well explore them first.
- Arrays are one of the simplest data structures. They are simply a collection of contiguous memory locations that contain items of the same type.

Arrays are Pointers

- 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
64	0
63	6
62	5
61	4
⋮	⋮

-

- A statically declared array of integers
  ```
  int A[3] = {4,5,6}
  ```
  would be stored on the stack like so.
Stack	
Address	Data
64	0
63	6
62	5
61	4
⋮	⋮
- 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 61.

Arrays are Pointers: Pointer Addition

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

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

- 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 62 namely 5. Dereferencing gives us the item at location 62, namely 5.

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

- 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 62 namely 5. Dereferencing gives us the item at location 62, namely 5.

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

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

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

Arrays are Pointers: Pointer Addition

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));
  }
  ```
  ```

Arrays are Pointers: Pointer Addition

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
  ```
  ```

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.
  ```

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).

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.

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));
```

The type system

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.

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

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.

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

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.

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

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.

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

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
64	0
63	6
62	5
61	4
⋮	⋮

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
64	0
63	6
62	5
61	4
⋮	⋮

- 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}
```

Stack	
Address	Data
64	0
63 62 61 60	6
59 58 57 56	5
55 54 53 52	4
⋮	⋮

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.

Now were going to discuss how we manage memory.

- We request it from the operating system (allocate it) using
  `void *malloc(size_t size)`

Memory Managment

Now were going to discuss how we manage memory.

- We request it from the operating system (allocate it) using `void *malloc(size_t size)`
- We specify that memory is no longer needed (deallocate) it using `void free(void *ptr)`

Malloc

- Beginning with *malloc()* lets review its summary from the linux manual pages.

Malloc

- Beginning with *malloc()* lets review its summary from the linux manual pages.
- *The malloc() function allocates size bytes and returns a pointer to the allocated memory.* -linux manual pages

Malloc

- Beginning with *malloc()* lets review its summary from the linux manual pages.
- *The malloc() function allocates size bytes and returns a pointer to the allocated memory.* -linux manual pages
- Relating it back to our memory table the malloc function accepts a number of bytes as an argument and returns a memory address, (index of our table), to memory on the Heap.

- Beginning with *malloc()* lets review its summary from the linux manual pages.
- *The malloc() function allocates size bytes and returns a pointer to the allocated memory.* -linux manual pages
- Relating it back to our memory table the malloc function accepts a number of bytes as an argument and returns a memory address, (index of our table), to memory on the Heap.
- Consider the table of bytes below, notice we start from the bottom as we now reference the Heap.

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

Malloc

- A call to malloc will return a heap address. We can use it to request memory for an int pointer like so.

Malloc

- A call to malloc will return a heap address. We can use it to request memory for an int pointer like so.
- `int *i = malloc(sizeof(int));`

Malloc

- A call to malloc will return a heap address. We can use it to request memory for an int pointer like so.
- `int *i = malloc(sizeof(int));`
-

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

Malloc

- A call to malloc will return a heap address. We can use it to request memory for an int pointer like so.
- `int *i = malloc(sizeof(int));`
-

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

- In this instance malloc may return address 0, so rows 0-3 would be used to store our integer.

- A call to malloc will return a heap address. We can use it to request memory for an int pointer like so.
- `int *i = malloc(sizeof(int));`
-

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

- In this instance malloc may return address 0, so rows 0-3 would be used to store our integer.
- Notice the use of the *sizeof(int)* to ensure we request the appropriate amount of storage for the type were pointing to.

Malloc

- A call to malloc will return a heap address. We can use it to request memory for an int pointer like so.
- `int *i = malloc(sizeof(int));`
-

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

- In this instance malloc may return address 0, so rows 0-3 would be used to store our integer.
- Notice the use of the *sizeof(int)* to ensure we request the appropriate amount of storage for the type were pointing to.
- The size of these types may vary from hardware to hardware so it is important to use sizeof to keep our code as hardware independent as possible.

Malloc an array

- Lets see how we can create our own arrays with malloc using a code example.

Malloc an array

- Lets see how we can create our own arrays with malloc using a code example.
- ```c
  int *A = malloc(sizeof(int) * 4);
  //store the first 4 multiples of 5
  for(int i = 0; i < 4; i++) A[i] = 5*i;
  // print the elements of A
  for(int i = 0; i < 4; i++) printf("A[%d] = %d\n", i, A[i]);
  ```
  ```

Malloc an array

- Lets see how we can create our own arrays with malloc using a code example.
- ```c
  int *A = malloc(sizeof(int) * 4);
  //store the first 4 multiples of 5
  for(int i = 0; i < 4; i++) A[i] = 5*i;
  // print the elements of A
  for(int i = 0; i < 4; i++) printf("A[%d] = %d\n", i, A[i]);
  ```
- Were going to modify our table a bit. Its going to index every fourth byte so we can more compactly represent our array in memory.
  ```

Malloc an array

- Lets see how we can create our own arrays with malloc using a code example.
-
```c
int *A = malloc(sizeof(int) * 4);
//store the first 4 multiples of 5
for(int i = 0; i < 4; i++) A[i] = 5*i;
// print the elements of A
for(int i = 0; i < 4; i++) printf("A[%d] = %d\n", i, A[i]);
```
- Were going to modify our table a bit. Its going to index every fourth byte so we can more compactly represent our array in memory.
-

Address	Data
⋮	⋮
12	15
8	10
4	5
0	0
Heap	

Free

- Dynamic memory allows us to allocate memory programmatically. Say we allocate memory for a new game session every time a new user connects to our server.

Free

- Dynamic memory allows us to allocate memory programmatically. Say we allocate memory for a new game session every time a new user connects to our server.
- But obviously we could have a lot of people connecting to our server, we will eventually run out of memory.

Free

- Dynamic memory allows us to allocate memory programmatically. Say we allocate memory for a new game session every time a new user connects to our server.
- But obviously we could have a lot of people connecting to our server, we will eventually run out of memory.
- But eventually some users will stop playing and disconnect from our server. So now we should be able to allow someone else to play. We can do this by reusing the memory allocated for the disconnected player for a new session. We can do this with *free*

Free

- Dynamic memory allows us to allocate memory programmatically. Say we allocate memory for a new game session every time a new user connects to our server.
- But obviously we could have a lot of people connecting to our server, we will eventually run out of memory.
- But eventually some users will stop playing and disconnect from our server. So now we should be able to allow someone else to play. We can do this by reusing the memory allocated for the disconnected player for a new session. We can do this with *free*
- `void free(void *ptr);`

Free

- Dynamic memory allows us to allocate memory programmatically. Say we allocate memory for a new game session every time a new user connects to our server.
- But obviously we could have a lot of people connecting to our server, we will eventually run out of memory.
- But eventually some users will stop playing and disconnect from our server. So now we should be able to allow someone else to play. We can do this by reusing the memory allocated for the disconnected player for a new session. We can do this with *free*
- `void free(void *ptr);`
- Free is the other half of memory management. Whatever memory you request with malloc you must eventually return with free.

Free

- Dynamic memory allows us to allocate memory programmatically. Say we allocate memory for a new game session every time a new user connects to our server.
- But obviously we could have a lot of people connecting to our server, we will eventually run out of memory.
- But eventually some users will stop playing and disconnect from our server. So now we should be able to allow someone else to play. We can do this by reusing the memory allocated for the disconnected player for a new session. We can do this with *free*
- `void free(void *ptr);`
- Free is the other half of memory management. Whatever memory you request with malloc you must eventually return with free.
- If you fail to free memory you allocate it is called a memory leak. Memory leaks are sources of some mysterious bugs as they can occur at seemingly random (to the programmer) times.

Free

- Dynamic memory allows us to allocate memory programmatically. Say we allocate memory for a new game session every time a new user connects to our server.
- But obviously we could have a lot of people connecting to our server, we will eventually run out of memory.
- But eventually some users will stop playing and disconnect from our server. So now we should be able to allow someone else to play. We can do this by reusing the memory allocated for the disconnected player for a new session. We can do this with *free*
- `void free(void *ptr);`
- Free is the other half of memory management. Whatever memory you request with malloc you must eventually return with free.
- If you fail to free memory you allocate it is called a memory leak. Memory leaks are sources of some mysterious bugs as they can occur at seemingly random (to the programmer) times.
- Consider the example of the above gamer server. If it never reclaimed memory it would eventually not allow anyone to play.

Free

- Lets go over another programming example allocating memory for a struct pointer and using free appropriately.

Free

- Lets go over another programming example allocating memory for a struct pointer and using free appropriately.

- ```
typedef struct {
    float* vals;
    int length;
} data;

int main(){
    //we can use sizeof like we would for any other data type
    data* D = malloc(sizeof(data));

    //the data is also a pointer so we must allocate memory for it
    D->vals = malloc(sizeof(float) * 5);
    D->length = 5; //we have space for 5 data points

    for (int i = 0; i < D->length; i++) {
        D->vals[i] = 0.25 * (i+1);
    }
    for(int i = 0; i < D->length; i++){
        printf("data point %d: %f\n", i, D->vals[i]);
    }

    //if we fail to free vals before D, then we have created a memory leak.
    free(D->vals);
    free(D);
}
```
```

Thanks for watching

- Hopefully you have gained a solid understanding of pointers and the basics of memory management.

Thanks for watching

- Hopefully you have gained a solid understanding of pointers and the basics of memory management.
- Next we are going put what we've learned into practice by implementing a dynamic data structure in C.

- Hopefully you have gained a solid understanding of pointers and the basics of memory management.
- Next we are going put what we've learned into practice by implementing a dynamic data structure in C.
- Thanks for watching!