Introduction to Programming

Week – *6*, Lecture – *2*

Pointers in C – Part 2

SAURABH SRIVASTAVA

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

IIT KANPUR

What we know till now…

Pointers are a special type of variables, which store the address of other variables…

- … in essence, “pointing” to them

What we know till now…

Pointers are a special type of variables, which store the address of other variables…

- … in essence, “pointing” to them

A pointer variable have an associated data type – the type of variables, it can point to

What we know till now…

Pointers are a special type of variables, which store the address of other variables…

- … in essence, “pointing” to them

A pointer variable have an associated data type – the type of variables, it can point to

We can assign the address of a compatible variable to a pointer variable using the `&` operator

- … e.g. `ptr = &var;`

What we know till now…

Pointers are a special type of variables, which store the address of other variables…

- … in essence, “pointing” to them

A pointer variable have an associated data type – the type of variables, it can point to

We can assign the address of a compatible variable to a pointer variable using the `&` operator

- … e.g. `ptr = &var;`

Similarly, to access the value stored at the address stored in a pointer variable, `*` can be used

- … e.g. `*ptr = value;`

What we know till now…

Pointers are a special type of variables, which store the address of other variables…

- … in essence, “pointing” to them

A pointer variable have an associated data type – the type of variables, it can point to

We can assign the address of a compatible variable to a pointer variable using the `&` operator

- … e.g. `ptr = &var;`

Similarly, to access the value stored at the address stored in a pointer variable, `*` can be used

- … e.g. `*ptr = value;`

We can add or subtract integers, to and from a pointer variable…

- … which advances or backtracks the stored address in the pointer, in fixed denominations…
- … which is equal to the size of the associated data type for the pointer variable

Pointers and Arrays

Remember that arrays are a collection of variables, residing contiguously in the memory

- … you can access the i^{th} element of the array, using the expression `arr[i]`

Pointers and Arrays

Remember that arrays are a collection of variables, residing contiguously in the memory

- … you can access the i^{th} element of the array, using the expression `arr[i]`

However, whenever you write `arr[i]`, the compiler converts it into `*(arr + i)`

Pointers and Arrays

Remember that arrays are a collection of variables, residing contiguously in the memory

- … you can access the i^{th} element of the array, using the expression `arr[i]`

However, whenever you write `arr[i]`, the compiler converts it into `*(arr + i)`

You may have guessed it by now – here, `arr` is nothing but a pointer …

- … which points to the starting address of the allocated space

Pointers and Arrays

Remember that arrays are a collection of variables, residing contiguously in the memory

- … you can access the `i`th element of the array, using the expression `arr[i]`

However, whenever you write `arr[i]`, the compiler converts it into `*(arr + i)`

You may have guessed it by now – here, `arr` is nothing but a pointer …

- … which points to the starting address of the allocated space
- Therefore, for a given array, you can always interchange `arr[i]` with `*(arr + i)`

Pointers and Arrays

Remember that arrays are a collection of variables, residing contiguously in the memory

- … you can access the i^{th} element of the array, using the expression `arr[i]`

However, whenever you write `arr[i]`, the compiler converts it into `*(arr + i)`

You may have guessed it by now – here, `arr` is nothing but a pointer …

- … which points to the starting address of the allocated space
- Therefore, for a given array, you can always interchange `arr[i]` with `*(arr + i)`

Also, `arr` is just an alias for `&arr[0]`…

- … thus, if `p` is a compatible pointer variable, the statements `p = &arr[0]` is equivalent to `p = arr`
- Also, it is perfectly legal, in this case, to write `p[i]` in place of `arr[i]`

Pointers and Arrays

Remember that arrays are a collection of variables, residing contiguously in the memory

- … you can access the i^{th} element of the array, using the expression `arr[i]`

However, whenever you write `arr[i]`, the compiler converts it into `*(arr + i)`

You may have guessed it by now – here, `arr` is nothing but a pointer …

- … which points to the starting address of the allocated space
- Therefore, for a given array, you can always interchange `arr[i]` with `*(arr + i)`

Also, `arr` is just an alias for `&arr[0]`…

- … thus, if `p` is a compatible pointer variable, the statements `p = &arr[0]` is equivalent to `p = arr`
- Also, it is perfectly legal, in this case, to write `p[i]` in place of `arr[i]`
- However, while writing `p++` is perfectly fine, `arr++` is not a valid statement…
- … basically, while the array name is definitely a pointer, it is *not a variable*, but a constant

Equivalence of Pointers and Arrays

```c
#include<stdio.h>

int main()
{
        char hello[] = "Hello World !!";
        char *hptr = hello; // points to the starting address of hello
        for(;*hptr != '\0'; hptr++)
                printf("%c", *hptr);
        printf("\n");
        return 0;
}
```

Equivalence of Pointers and Arrays

```c
#include<stdio.h>

int main()
{
        char hello[] = "Hello World !!";
        char *hptr = hello; // points to the starting address of hello
        for(;*hptr != '\0'; hptr++)
                printf("%c", *hptr);
        printf("\n");
        return 0;
}
```

This allows us to move through the elements of `hello`, using `hptr`

Equivalence of Pointers and Arrays

```c
#include<stdio.h>

int main()
{
        char hello[] = "Hello World !!";
        char *hptr = hello; // points to the starting address of hello
        for(;*hptr != '\0'; hptr++)
                printf("%c", *hptr);
        printf("\n");
        return 0;
}
```

This is equivalent to writing:

```c
char *hptr = &hello[0]; or
char *hptr; hptr = &hello[0];
```

Equivalence of Pointers and Arrays

```c
#include<stdio.h>

int main()
{
        char hello[] = "Hello World !!";
        char *hptr = hello; // points to the starting address of hello
        for(;*hptr != '\0'; hptr++)
                printf("%c", *hptr);
        printf("\n");
        return 0;
}
```

… and, this loop is equivalent to writing:

```c
for(i = 0; arr[i] != '\0'; i++)
        printf("%c", arr[i]);
```

Equivalence of Pointers and Arrays

```c
#include<stdio.h>

int main()
{
        char hello[] = "Hello World !!";
        char *hptr = hello; // points to the starting address of hello
        for(;*hptr != '\0'; hptr++)
                printf("%c", *hptr);
        printf("\n");
        return 0;
}
```

… and, this loop is equivalent to writing:

```c
for(i = 0; arr[i] != '\0'; i++)
        printf("%c", arr[i]);
```

The pointer arithmetic becomes handy here !!

Equivalence of Pointers and Arrays

```
saurabh@saurabh-VirtualBox:~/C/examples/Week 6$ gcc HelloPointers.c
saurabh@saurabh-VirtualBox:~/C/examples/Week 6$ ./a.out
Hello World !!
saurabh@saurabh-VirtualBox:~/C/examples/Week 6$
```

Equivalence of Pointers and Arrays

```
saurabh@saurabh-VirtualBox:~/C/examples/Week 6$ gcc HelloPointers.c
saurabh@saurabh-VirtualBox:~/C/examples/Week 6$ ./a.out
Hello World !!
saurabh@saurabh-VirtualBox:~/C/examples/Week 6$
```

… and this is our "`Hello World !!`" message, printed with the help of pointers !!

Using dynamically sized arrays

We haven’t discussed a single reason to use pointers as yet

- After all, everything that we did with pointers, we could do that by other means anyhow
- So what is the need of pointers then? We’ll now see one good reason for using them…

Using dynamically sized arrays

We haven't discussed a single reason to use pointers as yet

- After all, everything that we did with pointers, we could do that by other means anyhow
- So what is the need of pointers then? We'll now see one good reason for using them…

Remember that we discussed that the declared capacity of an array has to be a constant

- … something like `int arr[10];` – here, `10` is the constant
- We cannot, however, declare an array like `int arr[n];` – where `n` is a variable

Using dynamically sized arrays

We haven't discussed a single reason to use pointers as yet

- After all, everything that we did with pointers, we could do that by other means anyhow
- So what is the need of pointers then? We'll now see one good reason for using them…

Remember that we discussed that the declared capacity of an array has to be a constant

- … something like `int arr[10];` – here, `10` is the constant
- We cannot, however, declare an array like `int arr[n];` – where `n` is a variable

The `malloc()` function allocates an arbitrary amount of memory, and returns a pointer to it

- … e.g., if we require an `int` array of size `n`, we can allocate `sizeof(int) * n` bytes of space in memory
- We can then use the returned pointer, similar to an array

Using dynamically sized arrays

We haven't discussed a single reason to use pointers as yet

- After all, everything that we did with pointers, we could do that by other means anyhow
- So what is the need of pointers then? We'll now see one good reason for using them…

Remember that we discussed that the declared capacity of an array has to be a constant

- … something like `int arr[10];` – here, `10` is the constant
- We cannot, however, declare an array like `int arr[n];` – where `n` is a variable

The `malloc()` function allocates an arbitrary amount of memory, and returns a pointer to it

- … e.g., if we require an `int` array of size `n`, we can allocate `sizeof(int) * n` bytes of space in memory
- We can then use the returned pointer, similar to an array

There is a drawback of using `malloc()` though – the allocated memory must be "freed" explicitly

- … unlike spaces for regular variables, which are freed "automatically" when the control leaves their block

Using dynamically sized arrays

We haven't discussed a single reason to use pointers as yet

- After all, everything that we did with pointers, we could do that by other means anyhow
- So what is the need of pointers then? We'll now see one good reason for using them…

Remember that we discussed that the declared capacity of an array has to be a constant

- … something like `int arr[10];` – here, `10` is the constant
- We cannot, however, declare an array like `int arr[n];` – where `n` is a variable

The `malloc()` function allocates an arbitrary amount of memory, and returns a pointer to it

- … e.g., if we require an `int` array of size `n`, we can allocate `sizeof(int) * n` bytes of space in memory
- We can then use the returned pointer, similar to an array

There is a drawback of using `malloc()` though – the allocated memory must be "freed" explicitly

- … unlike spaces for regular variables, which are freed "automatically" when the control leaves their block
- To free this memory, you must pass the returned pointer to the `free()` function, when you are done

Allocating and freeing dynamic memory

```c
#include<stdio.h>
#include<stdlib.h>
#include<ctype.h>

#define MAX_CHARS 21
int main()
{
        char word[MAX_CHARS];
        int characters_count = 0, i = 0;
        char *characters = NULL;
        printf("Enter a word (up to 20 characters):\n");
        fgets(word, MAX_CHARS, stdin);
        // Find out the number of characters
        while(word[i++] != '\n')
                characters_count++;
        characters = (char*)malloc(sizeof(char) * (characters_count+1));
        for(i = 0; i < characters_count; i++)
                characters[i] = toupper(word[i]);
        characters[characters_count] = word[characters_count] = '\0';
        printf("%s, when converted to uppercase, is %s\n", word, characters);
        free(characters);
        return 0;
}
```

Allocating and freeing dynamic memory

```c
#include<stdio.h>
#include<stdlib.h>
#include<ctype.h>

#define MAX_CHARS 21
int main()
{
        char word[MAX_CHARS];
        int characters_count = 0, i = 0;
        char *characters = NULL;
        printf("Enter a word (up to 20 characters):\n");
        fgets(word, MAX_CHARS, stdin);
        // Find out the number of characters
        while(word[i++] != '\n')
                characters_count++;
        characters = (char*)malloc(sizeof(char) * (characters_count+1));
        for(i = 0; i < characters_count; i++)
                characters[i] = toupper(word[i]);
        characters[characters_count] = word[characters_count] = '\0';
        printf("%s, when converted to uppercase, is %s\n", word, characters);
        free(characters);
        return 0;
}
```

To create a one-dimensional char array dynamically, we need a pointer variable of type `char*`

Allocating and freeing dynamic memory

```c
#include<stdio.h>
#include<stdlib.h>
#include<ctype.h>

#define MAX_CHARS 21
int main()
{
        char word[MAX_CHARS];
        int characters_count = 0, i = 0;
        char *characters = NULL;
        printf("Enter a word (up to 20 characters):\n");
        fgets(word, MAX_CHARS, stdin);
        // Find out the number of characters
        while(word[i++] != '\n')
                characters_count++;
        characters = (char*)malloc(sizeof(char) * (characters_count+1));
        for(i = 0; i < characters_count; i++)
                characters[i] = toupper(word[i]);
        characters[characters_count] = word[characters_count] = '\0';
        printf("%s, when converted to uppercase, is %s\n", word, characters);
        free(characters);
        return 0;
}
```

To create a one-dimensional char array dynamically, we need a pointer variable of type `char*`

`NULL` is a special constant that can be used with pointer variables to signify that this pointer variable does not point to any location

Allocating and freeing dynamic memory

```c
#include<stdio.h>
#include<stdlib.h>
#include<ctype.h>

#define MAX_CHARS 21
int main()
{
        char word[MAX_CHARS];
        int characters_count = 0, i = 0;
        char *characters = NULL;
        printf("Enter a word (up to 20 characters):\n");
        fgets(word, MAX_CHARS, stdin);
        // Find out the number of characters
        while(word[i++] != '\n')
                characters_count++;
        characters = (char*)malloc(sizeof(char) * (characters_count+1));
        for(i = 0; i < characters_count; i++)
                characters[i] = toupper(word[i]);
        characters[characters_count] = word[characters_count] = '\0';
        printf("%s, when converted to uppercase, is %s\n", word, characters);
        free(characters);
        return 0;
}
```

The `malloc()` function takes a single argument – the number of bytes required

Allocating and freeing dynamic memory

```c
#include<stdio.h>
#include<stdlib.h>
#include<ctype.h>

#define MAX_CHARS 21
int main()
{
        char word[MAX_CHARS];
        int characters_count = 0, i = 0;
        char *characters = NULL;
        printf("Enter a word (up to 20 characters):\n");
        fgets(word, MAX_CHARS, stdin);
        // Find out the number of characters
        while(word[i++] != '\n')
                characters_count++;
        characters = (char*)malloc(sizeof(char) * (characters_count+1));
        for(i = 0; i < characters_count; i++)
                characters[i] = toupper(word[i]);
        characters[characters_count] = word[characters_count] = '\0';
        printf("%s, when converted to uppercase, is %s\n", word, characters);
        free(characters);
        return 0;
}
```

The `malloc()` function takes a single argument – the number of bytes required

It is a good practice to use `sizeof()`, instead of putting absolute numbers like `1`, `4` or `8`

Allocating and freeing dynamic memory

```c
#include<stdio.h>
#include<stdlib.h>
#include<ctype.h>

#define MAX_CHARS 21
int main()
{
        char word[MAX_CHARS];
        int characters_count = 0, i = 0;
        char *characters = NULL;
        printf("Enter a word (up to 20 characters):\n");
        fgets(word, MAX_CHARS, stdin);
        // Find out the number of characters
        while(word[i++] != '\n')
                characters_count++;
        characters = (char*)malloc(sizeof(char) * (characters_count+1));
        for(i = 0; i < characters_count; i++)
                characters[i] = toupper(word[i]);
        characters[characters_count] = word[characters_count] = '\0';
        printf("%s, when converted to uppercase, is %s\n", word, characters);
        free(characters);
        return 0;
}
```

The `malloc()` function takes a single argument – the number of bytes required

It is a good practice to use `sizeof()`, instead of putting absolute numbers like `1`, `4` or `8`

`malloc()` just returns a block of space – you should typecast the returned pointer before using it in your code

Allocating and freeing dynamic memory

```c
#include<stdio.h>
#include<stdlib.h>
#include<ctype.h>

#define MAX_CHARS 21
int main()
{
        char word[MAX_CHARS];
        int characters_count = 0, i = 0;
        char *characters = NULL;
        printf("Enter a word (up to 20 characters):\n");
        fgets(word, MAX_CHARS, stdin);
        // Find out the number of characters
        while(word[i++] != '\n')
                characters_count++;
        characters = (char*)malloc(sizeof(char) * (characters_count+1));
        for(i = 0; i < characters_count; i++)
                characters[i] = toupper(word[i]);
        characters[characters_count] = word[characters_count] = '\0';
        printf("%s, when converted to uppercase, is %s\n", word, characters);
        free(characters);
        return 0;
}
```

Once allocated, we can use the pointer variable in the same fashion, as we use an array

Allocating and freeing dynamic memory

```c
#include<stdio.h>
#include<stdlib.h>
#include<ctype.h>

#define MAX_CHARS 21
int main()
{
        char word[MAX_CHARS];
        int characters_count = 0, i = 0;
        char *characters = NULL;
        printf("Enter a word (up to 20 characters):\n");
        fgets(word, MAX_CHARS, stdin);
        // Find out the number of characters
        while(word[i++] != '\n')
                characters_count++;
        characters = (char*)malloc(sizeof(char) * (characters_count+1));
        for(i = 0; i < characters_count; i++)
                characters[i] = toupper(word[i]);
        characters[characters_count] = word[characters_count] = '\0';
        printf("%s, when converted to uppercase, is %s\n", word, characters);
        free(characters);
        return 0;
}
```

At the end, we should free the memory that we allocated, by calling the `free()` function, and passing It the pointer to the allocated block

Digression – the right to left operators

Remember the discussion on post increment and pre increment operators?

- We saw that pre increment operator has right to left associativity

Digression – the right to left operators

Remember the discussion on post increment and pre increment operators?

- We saw that pre increment operator has right to left associativity
- The `*` and `&` operators are also right to left associative

Digression – the right to left operators

Remember the discussion on post increment and pre increment operators?

- We saw that pre increment operator has right to left associativity
- The `*` and `&` operators are also right to left associative

Also, the pre increment operator has the same precedence as `*`

- … while the post increment operator has a higher precedence then both

Digression – the right to left operators

Remember the discussion on post increment and pre increment operators?

- We saw that pre increment operator has right to left associativity
- The `*` and `&` operators are also right to left associative

Also, the pre increment operator has the same precedence as `*`

- … while the post increment operator has a higher precedence then both

In a nutshell,

- The expression `++*ptr` and `*ptr++` are *not the same*

Digression – the right to left operators

Remember the discussion on post increment and pre increment operators?

- We saw that pre increment operator has right to left associativity
- The `*` and `&` operators are also right to left associative

Also, the pre increment operator has the same precedence as `*`

- … while the post increment operator has a higher precedence then both

In a nutshell,

- The expression `++*ptr` and `*ptr++` are *not the same*
- `++*ptr` is evaluated as `++(*ptr)`, i.e. “increment the value, stored at the address pointed by `ptr`”

Digression – the right to left operators

Remember the discussion on post increment and pre increment operators?

- We saw that pre increment operator has right to left associativity
- The `*` and `&` operators are also right to left associative

Also, the pre increment operator has the same precedence as `*`

- … while the post increment operator has a higher precedence then both

In a nutshell,

- The expression `++*ptr` and `*ptr++` are *not the same*
- `++*ptr` is evaluated as `++(*ptr)`, i.e. “increment the value, stored at the address pointed by `ptr`”
- `*ptr++` is evaluated as `*(ptr++)`, i.e. “add one to `ptr` and get the value at the previously pointed address”

Digression – the right to left operators

Remember the discussion on post increment and pre increment operators?

- We saw that pre increment operator has right to left associativity
- The `*` and `&` operators are also right to left associative

Also, the pre increment operator has the same precedence as `*`

- … while the post increment operator has a higher precedence then both

In a nutshell,

- The expression `++*ptr` and `*ptr++` are *not the same*
- `++*ptr` is evaluated as `++(*ptr)`, i.e. “increment the value, stored at the address pointed by `ptr`”
- `*ptr++` is evaluated as `*(ptr++)`, i.e. “add one to `ptr` and get the value at the previously pointed address”

A reminder – all operators with the same precedence, also have the same associativity

Homework !!

There are two more library functions, which which are used for dynamic memory allocation…

- `calloc()`, and
- `alloca()`
- Find out how they are different from `malloc()`

Find out the other C operators, which have right to left associativity

- This link has a table: https://www.tutorialspoint.com/cprogramming/c_operators_precedence.htm