Introduction to Programming

Week – *6*, Lecture – *1*

Pointers in C – Part 1

SAURABH SRIVASTAVA

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

IIT KANPUR

A recap of some basics

What happens when we declare a variable?

- Some space is allocated “in the memory” for storing a value

A recap of some basics

What happens when we declare a variable?

- Some space is allocated “in the memory” for storing a value

From there on, the name of the variable becomes an alias for that memory location

- All future assignments and manipulations, affect the value stored at this location

A recap of some basics

What happens when we declare a variable?

- Some space is allocated “in the memory” for storing a value

From there on, the name of the variable becomes an alias for that memory location

- All future assignments and manipulations, affect the value stored at this location

Also, arrays are basically a collection of variables…

- … which are allocated memory contiguously, i.e. one after another

A recap of some basics

What happens when we declare a variable?

- Some space is allocated “in the memory” for storing a value

From there on, the name of the variable becomes an alias for that memory location

- All future assignments and manipulations, affect the value stored at this location

Also, arrays are basically a collection of variables…

- … which are allocated memory contiguously, i.e. one after another

We can access different variables in the array, by using an index, that starts from 0…

- … and goes up to one less than the declared capacity of the array

Creating Pointer variables

Some languages – like C and C++ – also allow you know more about this memory location

Creating Pointer variables

Some languages – like C and C++ – also allow you know more about this memory location

A *pointer* is a variable that stores *address* of another variable

Creating Pointer variables

Some languages – like C and C++ – also allow you know more about this memory location

A *pointer* is a variable that stores *address* of another variable

In such a case, the pointer variable can also be used to manipulate the value of the variable…

- … because the value of the variable is just a value stored at a location in the memory

Creating Pointer variables

Some languages – like C and C++ – also allow you know more about this memory location

A *pointer* is a variable that stores *address* of another variable

In such a case, the pointer variable can also be used to manipulate the value of the variable…

- … because the value of the variable is just a value stored at a location in the memory

We declare pointer variables, by adding a `*` before their names like

```
int *p;
```

- … where `p` is a ”pointer variable, that points to a variable of type `int`”

Creating Pointer variables

Some languages – like C and C++ – also allow you know more about this memory location

A *pointer* is a variable that stores *address* of another variable

In such a case, the pointer variable can also be used to manipulate the value of the variable…

- … because the value of the variable is just a value stored at a location in the memory

We declare pointer variables, by adding a `*` before their names like

```
int *p;
```

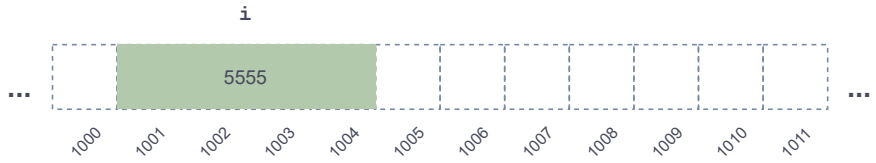
- … where `p` is a ”pointer variable, that points to a variable of type `int`”

To assign a value to a pointer variable, we use the `&` operator like

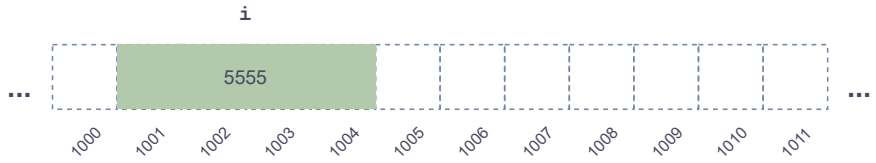
```
int *p; int i = 5555; p = &i;
```

- … where `&i` represents the “address of `i`” – the memory location that stores the value associated with `i`

Pointers in memory

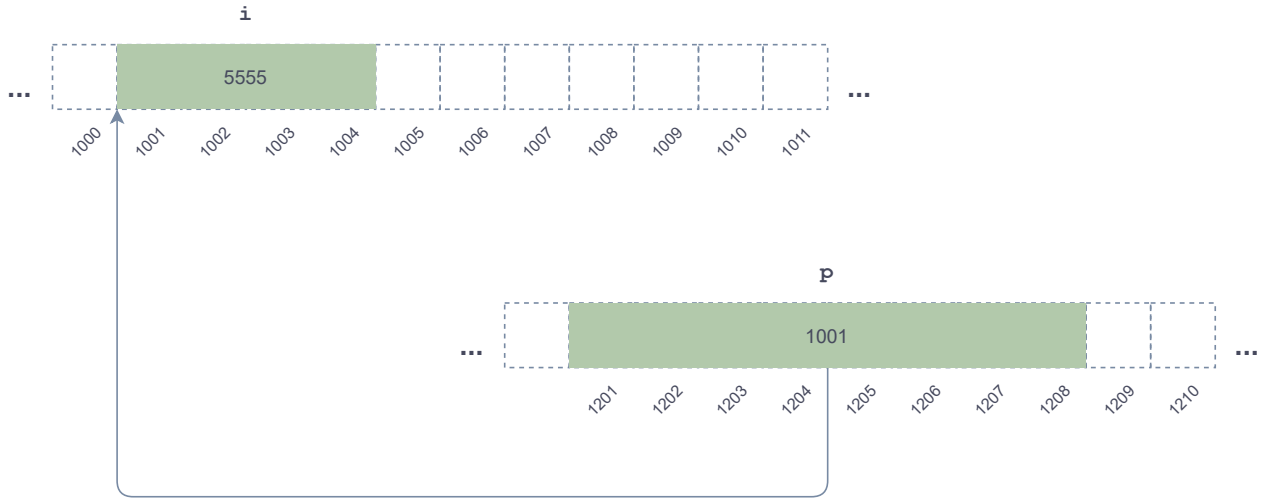


Pointers in memory

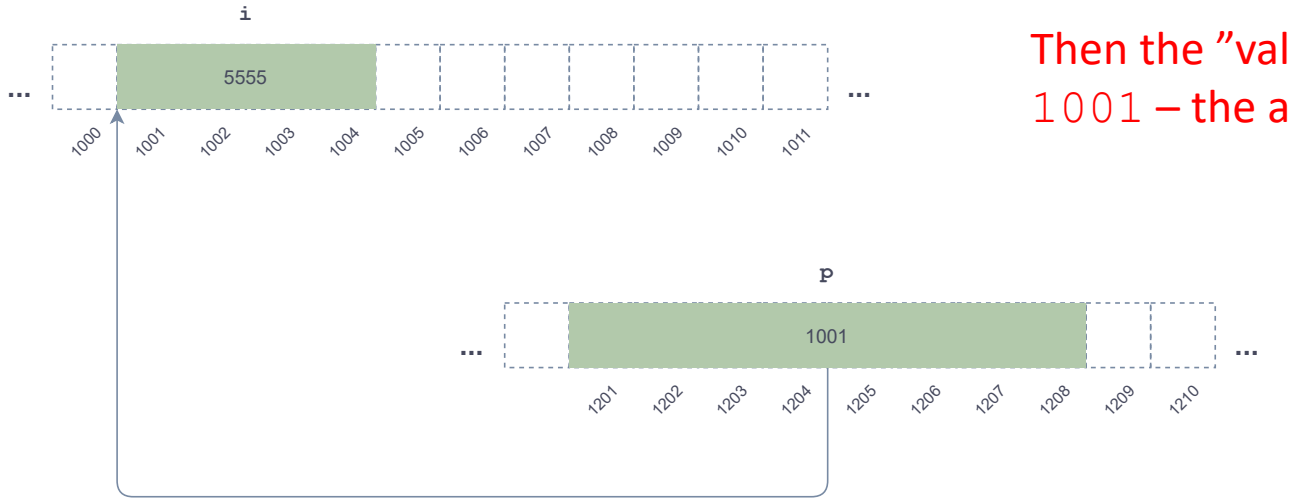


For example, assume that i has been allocated `4` bytes of memory starting at address `1001`

Pointers in memory



Pointers in memory



Then the ”value of” the pointer variable `p` is `1001` – the address of `i`

Using Pointer variables – I

The * operator can be used to ”dereference” an address…

- … meaning, getting the value stored at the address, stored in the pointer variable

Using Pointer variables – I

The `*` operator can be used to ”dereference” an address…

- … meaning, getting the value stored at the address, stored in the pointer variable, e.g.

```
int *p; int i = 5555; p = &i; printf(“The value of i is %d”, *p);
```

- … prints the current value of `i`, i.e. `5555`

Using Pointer variables – I

The `*` operator can be used to ”dereference” an address…

- … meaning, getting the value stored at the address, stored in the pointer variable, e.g.

```
int *p; int i = 5555; p = &i; printf("The value of i is %d", *p);
```

- … prints the current value of `i`, i.e. `5555`

Thus, `&` and `*` are kind of complementary to each other…

- … while `&` applies to any C variable, returning its address…

Using Pointer variables – I

The `*` operator can be used to ”dereference” an address…

- … meaning, getting the value stored at the address, stored in the pointer variable, e.g.

```
int *p; int i = 5555; p = &i; printf("The value of i is %d", *p);
```

- … prints the current value of `i`, i.e. `5555`

Thus, `&` and `*` are kind of complementary to each other…

- … while `&` applies to any C variable, returning its address…
- … `*` applies to a pointer variable, returning the value stored at the “pointed address”

Using Pointer variables – I

The `*` operator can be used to ”dereference” an address…

- … meaning, getting the value stored at the address, stored in the pointer variable, e.g.

```
int *p; int i = 5555; p = &i; printf("The value of i is %d", *p);
```

- … prints the current value of `i`, i.e. `5555`

Thus, `&` and `*` are kind of complementary to each other…

- … while `&` applies to any C variable, returning its address…
- … `*` applies to a pointer variable, returning the value stored at the “pointed address”
- To print addresses of variables using `printf()`, you can use the `%p` format specifier

Using Pointer variables – I

The `*` operator can be used to ”dereference” an address…

- … meaning, getting the value stored at the address, stored in the pointer variable, e.g.

```
int *p; int i = 5555; p = &i; printf("The value of i is %d", *p);
```

- … prints the current value of `i`, i.e. `5555`

Thus, `&` and `*` are kind of complementary to each other…

- … while `&` applies to any C variable, returning its address…
- … `*` applies to a pointer variable, returning the value stored at the “pointed address”
- To print addresses of variables using `printf()`, you can use the `%p` format specifier

The & operator can be used with ”any” type of variables – including a pointer variable

Using Pointer variables – I

The * operator can be used to ”dereference” an address…

- … meaning, getting the value stored at the address, stored in the pointer variable, e.g.

```
int *p; int i = 5555; p = &i; printf("The value of i is %d", *p);
```

- … prints the current value of `i`, i.e. `5555`

Thus, `&` and `*` are kind of complementary to each other…

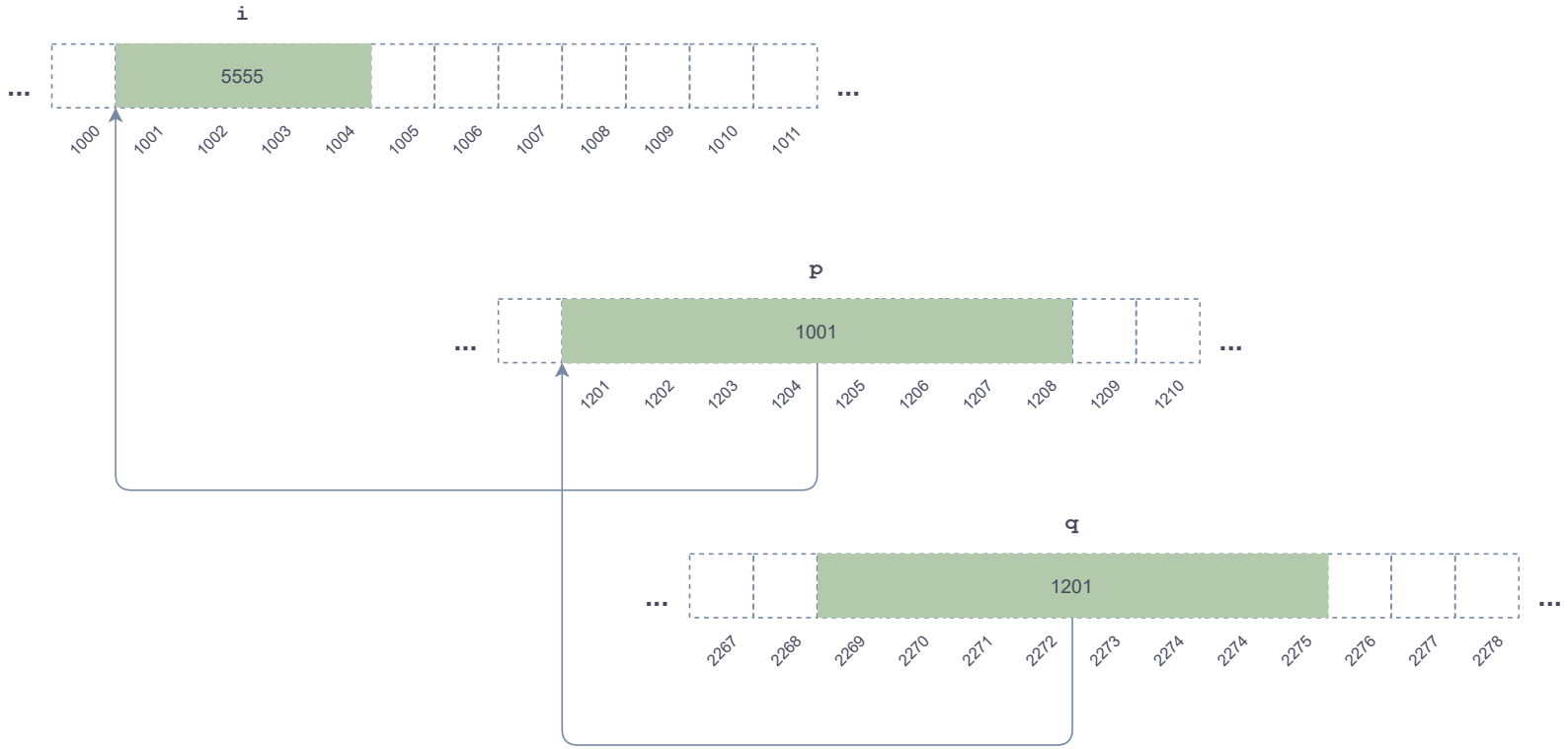
- … while `&` applies to any C variable, returning its address…
- … `*` applies to a pointer variable, returning the value stored at the “pointed address”
- To print addresses of variables using `printf()`, you can use the `%p` format specifier

The & operator can be used with ”any” type of variables – including a pointer variable

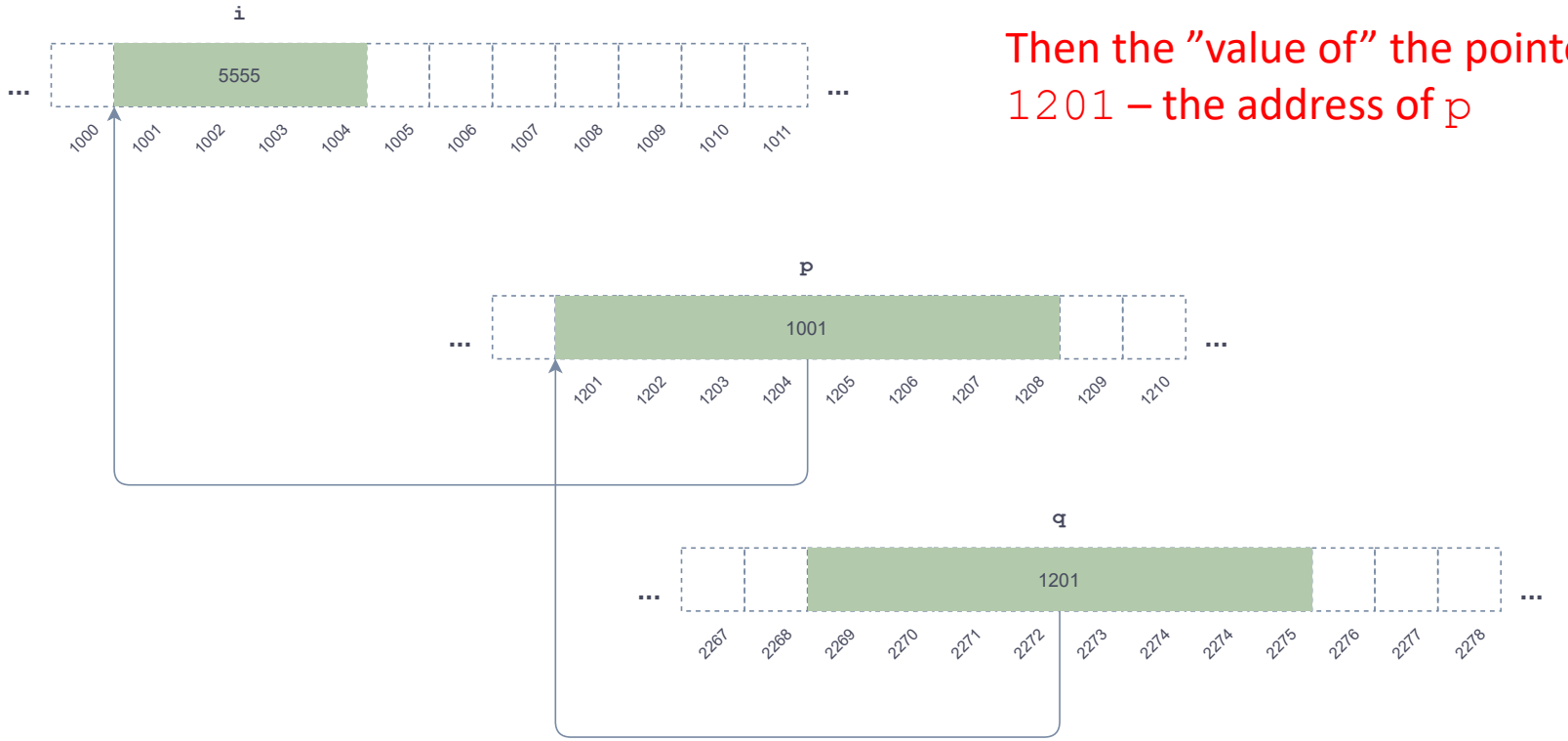
- Thus, this is perfectly valid

```
int *p; int i = 5555; p = &i; int **q = &p;
```

Pointers in memory



Pointers in memory



Then the ”value of” the pointer variable `q` is `1201` – the address of `p`

Using Pointer variables – I

The * operator can be used to ”dereference” an address…

- … meaning, getting the value stored at the address, stored in the pointer variable, e.g.

```
int *p; int i = 5555; p = &i; printf("The value of i is %d", *p);
```

- … prints the current value of `i`, i.e. `5555`

Thus, `&` and `*` are kind of complementary to each other…

- … while `&` applies to any C variable, returning its address…
- … `*` applies to a pointer variable, returning the value stored at the “pointed address”
- To print addresses of variables using `printf()`, you can use the `%p` format specifier

The & operator can be used with ”any” type of variables – including a pointer variable

- Thus, this is perfectly valid

```
int *p; int i = 5555; p = &i; int **q = &p;
```

- It means `q` stores to the *address* of `p` (not to be confused with the *value* of `p`, which is the address of `i`)

Using Pointer variables – I

The * operator can be used to ”dereference” an address…

- … meaning, getting the value stored at the address, stored in the pointer variable, e.g.

```
int *p; int i = 5555; p = &i; printf("The value of i is %d", *p);
```

- … prints the current value of `i`, i.e. `5555`

Thus, `&` and `*` are kind of complementary to each other…

- … while `&` applies to any C variable, returning its address…
- … `*` applies to a pointer variable, returning the value stored at the “pointed address”
- To print addresses of variables using `printf()`, you can use the `%p` format specifier

The & operator can be used with ”any” type of variables – including a pointer variable

- Thus, this is perfectly valid

```
int *p; int i = 5555; p = &i; int **q = &p;
```

- It means `q` stores to the *address* of `p` (not to be confused with the *value* of `p`, which is the address of `i`)
- The two `*`s in `q`’s declaration here mean that `q` is a “pointer to a pointer to an `int`” variable

Manipulating variables through pointers

```c
#include<stdio.h>

int main()
{
        int i = 5555;
        int *p = &i;
        int **q = &p;
        printf("The value of i is %d.\n", i);
        printf("The address of i is %p. ", p);
        printf("The updated value of i (via p) is %d.\n", ++*p);
        printf("The address of p is %p. ", q);
        printf("The value of p (via q) is %p. ", *q);
        printf("The updated value of i (via q) is %d.\n", ++**q);
        printf("The final value of i is %d.\n", i);
        printf("The sizes of all the variables here are:\n");
        printf("sizeof(i) = %zd\n", sizeof(i));
        printf("sizeof(p) = %zd\n", sizeof(p));
        printf("sizeof(q) = %zd\n", sizeof(q));
        return 0;
}
```

Manipulating variables through pointers

```c
#include<stdio.h>

int main()
{
        int i = 5555;
        int *p = &i;
        int **q = &p;
        printf("The value of i is %d.\n", i);
        printf("The address of i is %p. ", p);
        printf("The updated value of i (via p) is %d.\n", ++*p);
        printf("The address of p is %p. ", q);
        printf("The value of p (via q) is %p. ", *q);
        printf("The updated value of i (via q) is %d.\n", ++**q);
        printf("The final value of i is %d.\n", i);
        printf("The sizes of all the variables here are:\n");
        printf("sizeof(i) = %zd\n", sizeof(i));
        printf("sizeof(p) = %zd\n", sizeof(p));
        printf("sizeof(q) = %zd\n", sizeof(q));
        return 0;
}
```

The same memory location can be manipulated by any number of redirections through pointers

Manipulating variables through pointers

```
saurabh@saurabh-VirtualBox:~/C/examples/Week 6$ gcc Redirections.c
saurabh@saurabh-VirtualBox:~/C/examples/Week 6$ ./a.out
The value of i is 5555.
The address of i is 0x7ffdfc8e31e4. The updated value of i (via p) is 5556.
The address of p is 0x7ffdfc8e31e8. The value of p (via q) is 0x7ffdfc8e31e4. The updated value of i (via q) is 5557.
The final value of i is 5557.
The sizes of all the variables here are:
sizeof(i) = 4
sizeof(p) = 8
sizeof(q) = 8
saurabh@saurabh-VirtualBox:~/C/examples/Week 6$
```

Manipulating variables through pointers

```
saurabh@saurabh-VirtualBox:~/C/examples/Week 6$ gcc Redirections.c
saurabh@saurabh-VirtualBox:~/C/examples/Week 6$ ./a.out
The value of i is 5555.
The address of i is 0x7ffdfc8e31e4. The updated value of i (via p) is 5556.
The address of p is 0x7ffdfc8e31e8. The value of p (via q) is 0x7ffdfc8e31e4. The updated value of i (via q) is 5557.
The final value of i is 5557.
The sizes of all the variables here are:
sizeof(i) = 4
sizeof(p) = 8
sizeof(q) = 8
saurabh@saurabh-VirtualBox:~/C/examples/Week 6$
```

You can reach the same memory location via multiple redirections

Using Pointer variables – II

The amount of space that a pointer variable takes, depends on the platform

Using Pointer variables – II

The amount of space that a pointer variable takes, depends on the platform

- This is because the value that a pointer variable contains is an address
- If the machine has “shorter addresses”, the space required by pointer variables is also smaller

Using Pointer variables – II

The amount of space that a pointer variable takes, depends on the platform

- This is because the value that a pointer variable contains is an address
- If the machine has “shorter addresses”, the space required by pointer variables is also smaller
- For 32-bit systems, pointers take 4 bytes of space, and on 64-bit systems, they take 8 bytes of space

Using Pointer variables – II

The amount of space that a pointer variable takes, depends on the platform

- This is because the value that a pointer variable contains is an address
- If the machine has “shorter addresses”, the space required by pointer variables is also smaller
- For 32-bit systems, pointers take 4 bytes of space, and on 64-bit systems, they take 8 bytes of space

In any case, “all” pointer variables take the same amount of space

- Irrespective of whether they are pointing to a pointer or “regular” variable

Size of pointer variables

```c
#include<stdio.h>

int main()
{
        int i = 5555;
        int *p = &i;
        int **q = &p;
        printf("The value of i is %d.\n", i);
        printf("The address of i is %p. ", p);
        printf("The updated value of i (via p) is %d.\n", ++*p);
        printf("The address of p is %p. ", q);
        printf("The value of p (via q) is %p. ", *q);
        printf("The updated value of i (via q) is %d.\n", ++**q);
        printf("The final value of i is %d.\n", i);
        printf("The sizes of all the variables here are:\n");
        printf("sizeof(i) = %zd\n", sizeof(i));
        printf("sizeof(p) = %zd\n", sizeof(p));
        printf("sizeof(q) = %zd\n", sizeof(q));
        return 0;
}
```

Size of pointer variables

```c
#include<stdio.h>

int main()
{
        int i = 5555;
        int *p = &i;
        int **q = &p;
        printf("The value of i is %d.\n", i);
        printf("The address of i is %p. ", p);
        printf("The updated value of i (via p) is %d.\n", ++*p);
        printf("The address of p is %p. ", q);
        printf("The value of p (via q) is %p. ", *q);
        printf("The updated value of i (via q) is %d.\n", ++**q);
        printf("The final value of i is %d.\n", i);
        printf("The sizes of all the variables here are:\n");
        printf("sizeof(i) = %zd\n", sizeof(i));
        printf("sizeof(p) = %zd\n", sizeof(p));
        printf("sizeof(q) = %zd\n", sizeof(q));
        return 0;
}
```

Size of pointer variables

```
saurabh@saurabh-VirtualBox:~/C/examples/Week 6$ gcc Redirections.c
saurabh@saurabh-VirtualBox:~/C/examples/Week 6$ ./a.out
The value of i is 5555.
The address of i is 0x7ffdfc8e31e4. The updated value of i (via p) is 5556.
The address of p is 0x7ffdfc8e31e8. The value of p (via q) is 0x7ffdfc8e31e4. The updated value of i (via q) is 5557.
The final value of i is 5557.
The sizes of all the variables here are:
sizeof(i) = 4
sizeof(p) = 8
sizeof(q) = 8
saurabh@saurabh-VirtualBox:~/C/examples/Week 6$
```

Size of pointer variables

```
saurabh@saurabh-VirtualBox:~/C/examples/Week 6$ gcc Redirections.c
saurabh@saurabh-VirtualBox:~/C/examples/Week 6$ ./a.out
The value of i is 5555.
The address of i is 0x7ffdfc8e31e4. The updated value of i (via p) is 5556.
The address of p is 0x7ffdfc8e31e8. The value of p (via q) is 0x7ffdfc8e31e4. The updated value of i (via q) is 5557.
The final value of i is 5557.
The sizes of all the variables here are:
sizeof(i) = 4
sizeof(p) = 8
sizeof(q) = 8
saurabh@saurabh-VirtualBox:~/C/examples/Week 6$
```

The size of any pointer variable is the same, irrespective of the number of redirections

Using Pointer variables – II

The amount of space that a pointer variable takes, depends on the platform

- This is because the value that a pointer variable contains is an address
- If the machine has “shorter addresses”, the space required by pointer variables is also smaller
- For 32-bit systems, pointers take 4 bytes of space, and on 64-bit systems, they take 8 bytes of space

In any case, “all” pointer variables take the same amount of space

- Irrespective of whether they are pointing to a pointer or “regular” variable

You can *add* or *subtract* integers to and from pointer variables

Using Pointer variables – II

The amount of space that a pointer variable takes, depends on the platform

- This is because the value that a pointer variable contains is an address
- If the machine has “shorter addresses”, the space required by pointer variables is also smaller
- For 32-bit systems, pointers take 4 bytes of space, and on 64-bit systems, they take 8 bytes of space

In any case, “all” pointer variables take the same amount of space

- Irrespective of whether they are pointing to a pointer or “regular” variable

You can *add* or *subtract* integers to and from pointer variables

- It is equivalent to pointing to the “next” or “previous” variables of the *respective type*

Using Pointer variables – II

The amount of space that a pointer variable takes, depends on the platform

- This is because the value that a pointer variable contains is an address
- If the machine has “shorter addresses”, the space required by pointer variables is also smaller
- For 32-bit systems, pointers take 4 bytes of space, and on 64-bit systems, they take 8 bytes of space

In any case, “all” pointer variables take the same amount of space

- Irrespective of whether they are pointing to a pointer or “regular” variable

You can *add* or *subtract* integers to and from pointer variables

- It is equivalent to pointing to the “next” or “previous” variables of the *respective type*
- For instance, if you add 1 to a pointer variable which points to a `char`, it will point to the next address …
- … but if you add 1 to a pointer variable which points to an `int`, it will jump ahead by 4 addresses !!

Incrementing pointers

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

int main()
{
        int dummy;
        int *ptr = &dummy;

        printf("I am going to show you how Segmentation Faults occur\n");
        printf("I'll do that by accessing a memory location that I am not allowed to !!\n");
        printf("I am starting with an int* variable, pointing to memory location %p\n", ptr);
        do
        {
                printf("I am attempting to write at memory location %p\n", ptr);
                *ptr = 5;
                printf("Successful, press Enter to continue...");
                scanf("%*c");
                ptr++;
        }
        while(1);
}
```

Incrementing pointers

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

int main()
{
        int dummy;
        int *ptr = &dummy;

        printf("I am going to show you how Segmentation Faults occur\n");
        printf("I'll do that by accessing a memory location that I am not allowed to !!\n");
        printf("I am starting with an int* variable, pointing to memory location %p\n", ptr);
        do
        {
                printf("I am attempting to write at memory location %p\n", ptr);
                *ptr = 5;
                printf("Successful, press Enter to continue...");
                scanf("%*c");
                ptr++;
        }
        while(1);
}
```

We can increment a pointer variable like an integer

Incrementing pointers

```
saurabh@saurabh-VirtualBox:~/C/examples/Week 6$ gcc SegmentationFaulter.c
saurabh@saurabh-VirtualBox:~/C/examples/Week 6$ ./a.out
I am going to show you how Segmentation Faults occur
I'll do that by accessing a memory location that I am not allowed to !!
I am starting with an int* variable, pointing to memory location 0x7ffc3b4b1c7c
I am attempting to write at memory location 0x7ffc3b4b1c7c
Successful, press Enter to continue...
I am attempting to write at memory location 0x7ffc3b4b1c80
Successful, press Enter to continue...
```

Incrementing pointers

```
saurabh@saurabh-VirtualBox:~/C/examples/Week 6$ gcc SegmentationFaulter.c
saurabh@saurabh-VirtualBox:~/C/examples/Week 6$ ./a.out
I am going to show you how Segmentation Faults occur
I'll do that by accessing a memory location that I am not allowed to !!
I am starting with an int* variable, pointing to memory location 0x7ffc3b4b1c7c
I am attempting to write at memory location 0x7ffc3b4b1c7c
Successful, press Enter to continue...
I am attempting to write at memory location 0x7ffc3b4b1c80
Successful, press Enter to continue...
```

The increment advanced the pointer by 4 addresses

Using Pointer variables – II

The amount of space that a pointer variable takes, depends on the platform

- This is because the value that a pointer variable contains is an address
- If the machine has “shorter addresses”, the space required by pointer variables is also smaller
- For 32-bit systems, pointers take 4 bytes of space, and on 64-bit systems, they take 8 bytes of space

In any case, “all” pointer variables take the same amount of space

- Irrespective of whether they are pointing to a pointer or “regular” variable

You can *add* or *subtract* integers to and from pointer variables

- It is equivalent to pointing to the “next” or “previous” variables of the *respective type*
- For instance, if you add 1 to a pointer variable which points to a `char`, it will point to the next address …
- … but if you add 1 to a pointer variable which points to an `int`, it will jump ahead by 4 addresses !!

Be careful while you do pointer arithmetic…

Using Pointer variables – II

The amount of space that a pointer variable takes, depends on the platform

- This is because the value that a pointer variable contains is an address
- If the machine has “shorter addresses”, the space required by pointer variables is also smaller
- For 32-bit systems, pointers take 4 bytes of space, and on 64-bit systems, they take 8 bytes of space

In any case, “all” pointer variables take the same amount of space

- Irrespective of whether they are pointing to a pointer or “regular” variable

You can *add* or *subtract* integers to and from pointer variables

- It is equivalent to pointing to the “next” or “previous” variables of the *respective type*
- For instance, if you add 1 to a pointer variable which points to a `char`, it will point to the next address …
- … but if you add 1 to a pointer variable which points to an `int`, it will jump ahead by 4 addresses !!

Be careful while you do pointer arithmetic…

- … it may result in illegal (or unexpected) memory manipulations

Caution – Illegal memory operations

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

int main()
{
        int dummy;
        int *ptr = &dummy;

        printf("I am going to show you how Segmentation Faults occur\n");
        printf("I'll do that by accessing a memory location that I am not allowed to !!\n");
        printf("I am starting with an int* variable, pointing to memory location %p\n", ptr);
        do
        {
                printf("I am attempting to write at memory location %p\n", ptr);
                *ptr = 5;
                printf("Successful, press Enter to continue...");
                scanf("%*c");
                ptr++;
        }
        while(1);
}
```

Caution – Illegal memory operations

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

int main()
{
        int dummy;
        int *ptr = &dummy;

        printf("I am going to show you how Segmentation Faults occur\n");
        printf("I'll do that by accessing a memory location that I am not allowed to !!\n");
        printf("I am starting with an int* variable, pointing to memory location %p\n", ptr);
        do
        {
                printf("I am attempting to write at memory location %p\n", ptr);
                *ptr = 5;
                printf("Successful, press Enter to continue...");
                scanf("%*c");
                ptr++;
        }
        while(1);
}
```

If you are not careful, you may end up accessing an illegal memory location

Caution – Illegal memory operations

```
saurabh@saurabh-VirtualBox:~/C/examples/Week 6$ gcc SegmentationFaulter.c
saurabh@saurabh-VirtualBox:~/C/examples/Week 6$ ./a.out
I am going to show you how Segmentation Faults occur
I'll do that by accessing a memory location that I am not allowed to !!
I am starting with an int* variable, pointing to memory location 0x7ffc3b4b1c7c
I am attempting to write at memory location 0x7ffc3b4b1c7c
Successful, press Enter to continue...
I am attempting to write at memory location 0x7ffc3b4b1c80
Successful, press Enter to continue...
I am attempting to write at memory location 0x7ffc00000009
Segmentation fault (core dumped)
saurabh@saurabh-VirtualBox:~/C/examples/Week 6$
```

Caution – Illegal memory operations

```
saurabh@saurabh-VirtualBox:~/C/examples/Week 6$ gcc SegmentationFaulter.c
saurabh@saurabh-VirtualBox:~/C/examples/Week 6$ ./a.out
I am going to show you how Segmentation Faults occur
I'll do that by accessing a memory location that I am not allowed to !!
I am starting with an int* variable, pointing to memory location 0x7ffc3b4b1c7c
I am attempting to write at memory location 0x7ffc3b4b1c7c
Successful, press Enter to continue...
I am attempting to write at memory location 0x7ffc3b4b1c80
Successful, press Enter to continue...
I am attempting to write at memory location 0x7ffc00000009
Segmentation fault (core dumped)
saurabh@saurabh-VirtualBox:~/C/examples/Week 6$
```

… and that’s what you’ll see, when you do so !!

Homework !!

There is another type of pointer, which is used under certain conditions…

- … it is of type `void*` …
- Find out what it is, and under what circumstances it may be useful

Find the address of the variable `ptr` in the program written in `SegmentationFaulter.c`

- Can you now explain the address which, when accessed, shot the Segmentation fault?
- Can you also guess, as to how variables are usually assigned space in the memory in C?