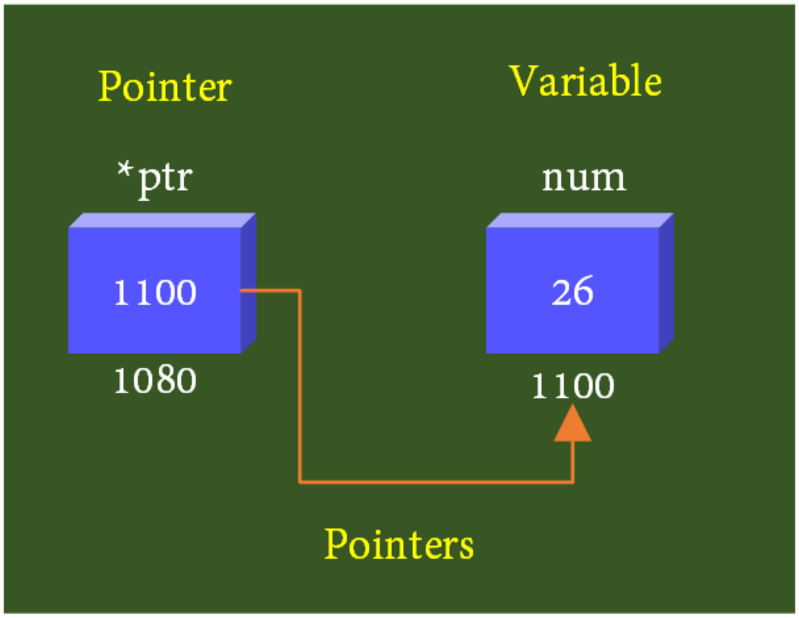
Unit 8

Pointer

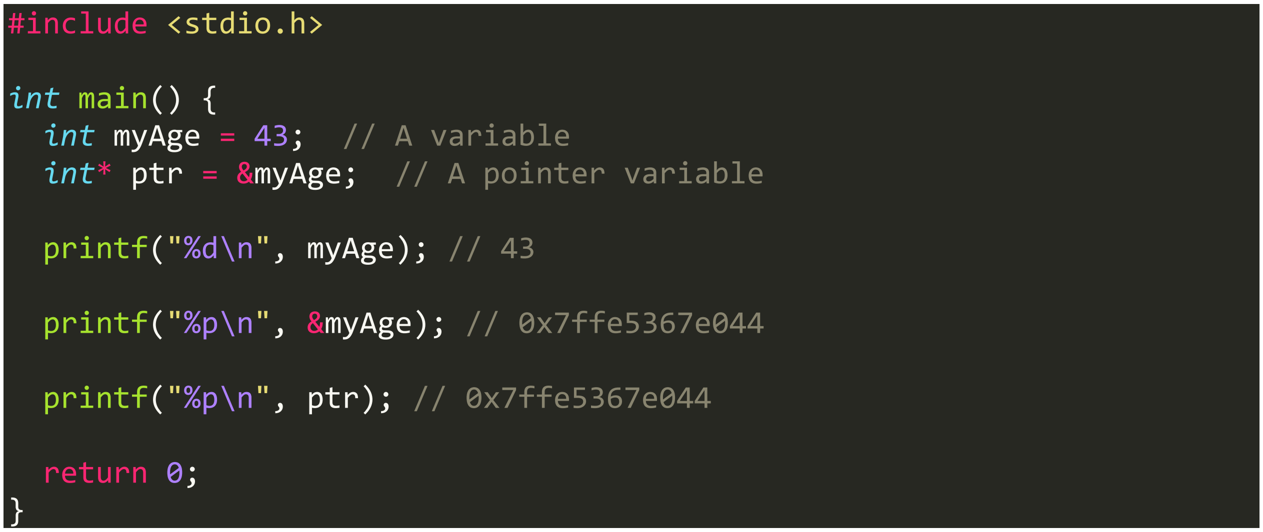
- Introduction to Pointer
- Address (&) and indirection (*) operator
- Pointer Arithmetic Operations
- Pointer to Pointer in C
- Dynamic Memory Allocation (malloc(), calloc(), realloc(), free())

Pointer

- Pointers (pointer variables) are special variables that are used to store addresses rather than values.
- A pointer is a variable that stores the memory address of another variable as its value.



Example program:



```
#include <stdio.h>

int main() {
  int myAge = 43;  // A variable
  int* ptr = &myAge;  // A pointer variable

  printf("%d\n", myAge); // 43

  printf("%p\n", &myAge); // 0x7ffe5367e044

  printf("%p\n", ptr); // 0x7ffe5367e044

  return 0;
}
```

Syntax

Here is how we can declare pointers.

```
int* p;
```

Here, we have declared a pointer p of int type.

We can also declare pointers in these ways.

```
int *p1;
int * p2;
```

Assigning addresses to Pointers

```
int* p, num;
num = 5;
p = &num;
```

Here, 5 is assigned to the c variable. And, the address of c is assigned to the pc pointer.

Address (&) and indirection (*) operator

- The address operator is used to obtain the memory address of a variable. It is denoted by the ampersand (&) symbol. When applied to a variable, it returns the address where that variable is stored in memory.
- The indirection operator is used to access the value stored at a particular memory address. It is denoted by an asterisk (*) symbol. When applied to a pointer, it retrieves the value stored at the memory address pointed to by the pointer.

Example:

```
#include <stdio.h>

int main() {
    int num = 42;
    int* ptr = &num;  // ptr stores the address of num

    printf("Address of num: %p\n", &num); //prints the address using &
opeator
```

```
    printf("Value of num: %d\n", num); // prints the value
    printf("Value stored in ptr: %d\n", *ptr); // prints the value
using * operator

    return 0;
}
```

Pointer Arithmetic Operations

We can perform arithmetic operations on the pointers like addition, subtraction, etc.

Following arithmetic operations are possible on the pointer in C language:

- Increment & Decrement
- Addition & Subtraction
- Comparison

Incrementing and Decrementing pointers in C

- If we increment a pointer by 1, the pointer will start pointing to the immediate next location.
- If we decrement a pointer by 1, the pointer will start pointing to the previous location.

For example: If we define 64-bit int variable, it will be incremented by 4 bytes.

```
#include <stdio.h>
int main()
{
    int num1 = 50, num2 = 30;
    int *p1 = &num1;
    int *p2 = &num2;
    printf("Before Inc. : Address of p1 variable is %u \n", p1);
    p1++;
    printf("After Inc. : Address of p1 variable is %u \n\n", p1);

    printf("Before Dec. :Address of p2 variable is %u \n", p2);
    p2--;
    printf("After Dec. : Address of p2 variable is %u \n\n", p2);
    return 0;
}
```

Addition and Subtraction pointers in C

- We can add and subtract a value to the pointer.
- The given value points to the next/previous location multiplied by the size of the datatype.
- For example, if we add 4 to the 64-int pointer it will add 4*4=16(bytes)

Example:

```
#include <stdio.h>
int main()
{
    int num1 = 50, num2 = 30;
    int *p1 = &num1;
    int *p2 = &num2;
    printf("Before Inc. : Address of p1 variable is %u \n", p1);
    p1 += 3;
    printf("After Inc. : Address of p1 variable is %u \n\n", p1);

    printf("Before Dec. :Address of p2 variable is %u \n", p2);
    p2 -= 5;
    printf("After Dec. : Address of p2 variable is %u \n\n", p2);
    return 0;
}
```

Pointer comparison in C

We can compare pointers if the addresses are greater, lesser, equal or not equal.

```
#include <stdio.h>
int main()
{
    int num1 = 50, num2 = 30;
    int *p1 = &num1;
    int *p2 = &num2;

    if (p1<p2) printf("p2 is greater than p1");
    if(p1>p2) printf("p1 is greater than p2");
    if(p1==p2) printf("p1 is equal to p2");
    if(p1 != p2) printf("p1 is not equal to p2");

    return 0;
}
```

Illegal Arithmetic Operations

- Address + Address = illegal
- Address * Address = illegal
- Address % Address = illegal
- Address / Address = illegal
- Address & Address = illegal
- Address ^ Address = illegal
- Address | Address = illegal
- ~Address = illegal

Pointer to a Pointer in C (Double Pointer)

In C, a pointer to pointer is a pointer that holds the memory address of another pointer. It is also known as a double pointer.

```c
int *p; // a pointer to an integer
int **pp; // a double pointer to an integer
```

Example

```c
#include <stdio.h>
void main()
{
    int a = 10;
    int *p = &a;
    int **pp = &p;                     // double pointer
    printf("address of a: %x\n", p);   // Address of a will be printed
    printf("address of p: %x\n", pp);  // Address of p will be printed
    printf("value stored at p: %d\n", *p); // value at address p i.e. 10 will be printed
    printf("value stored at pp: %d\n", **pp); // value at address *p i.e 10 will be printed
}
```

Dynamic Memory Allocation

- As we know, an array is a collection of a fixed number of values. Once the size of an array is declared, we cannot change it.
- Sometimes the size of the array we declared may be insufficient. To solve this issue, we can allocate memory manually during run-time. This is known as dynamic memory allocation in C programming.
- To allocate memory dynamically, library functions are malloc(), calloc(), realloc() and free() are used. These functions are defined in the <stdlib.h> header file.

malloc()

- The name "malloc" stands for memory allocation.
- The malloc() function reserves a block of memory of the specified number of bytes. And, it returns a pointer of void which can be casted into pointers of any form.
- Syntax of malloc()

```
ptr = (castType*) malloc(size);
```

Example

```
ptr = (float*) malloc(100 * sizeof(float));
```

- The above statement allocates 400 bytes of memory. It's because the size of float is 4 bytes. And, the pointer ptr holds the address of the first byte in the allocated memory.
- The expression results in a NULL pointer if the memory cannot be allocated.

calloc()

- The name "calloc" stands for contiguous allocation.
- The malloc() function allocates memory and leaves the memory uninitialized, whereas the calloc() function allocates memory and initializes all bits to zero.

Syntax of calloc()

```
ptr = (castType*)calloc(n, size);
```

Example:

```
ptr = (float*) calloc(25, sizeof(float));
```

The above statement allocates contiguous space in memory for 25 elements of type float.

Realloc()

- If the dynamically allocated memory is insufficient or more than required, you can change the size of previously allocated memory using the realloc() function.

Syntax of realloc()

ptr = realloc(ptr, x);

Here, ptr is reallocated with a new size x.

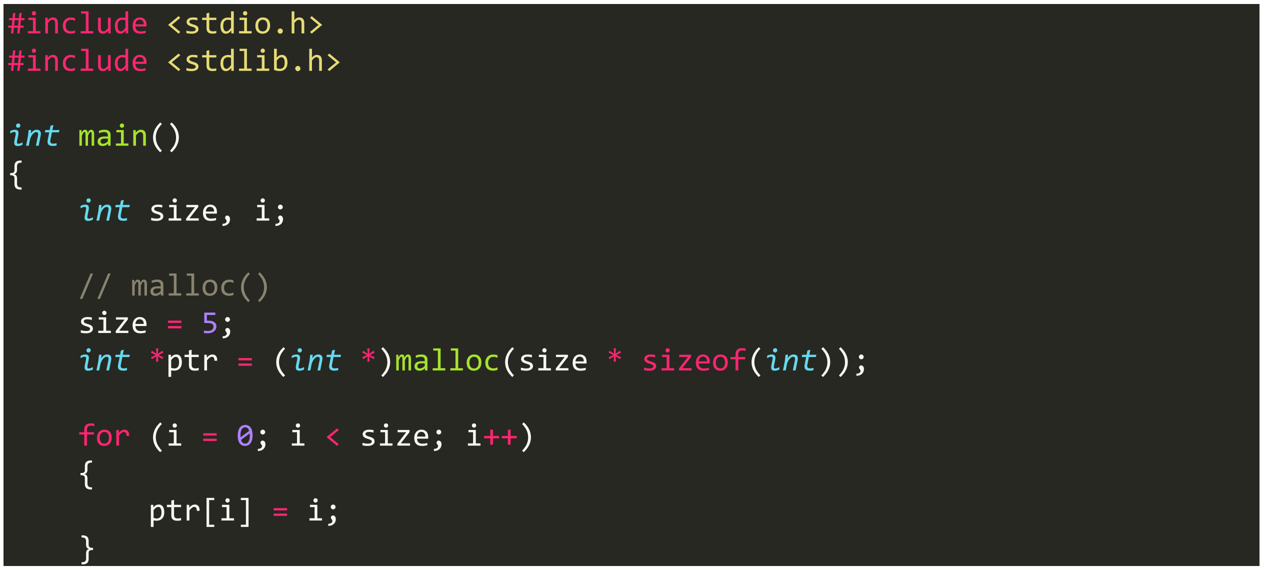
free()

- Dynamically allocated memory created with either calloc() or malloc() doesn't get freed on their own. You must explicitly use free() to release the space.

Syntax of free()

```
free(ptr);
```

This statement frees the space allocated in the memory pointed by ptr.



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

int main()
{
    int size, i;

    // malloc()
    size = 5;
    int *ptr = (int *)malloc(size * sizeof(int));

    for (i = 0; i < size; i++)
    {
        ptr[i] = i;
    }
```

```
    for (i = 0; i < size; i++)
    {
        printf("%d ", i, ptr[i]);
    }

    printf("\n");
    // calloc()
    int *array = calloc(10, sizeof(int));
    for (int i = 0; i < 10; i++)
    {
        array[i] = i;
    }
    for (int i = 0; i < 10; i++)
    {
        printf("%d ", i, array[i]);
    }

    printf("\n");

    // realloc()
    array = realloc(array, 20 * sizeof(int));
    for (int i = 10; i < 20; i++)
    {
        array[i] = i;
    }
    for (int i = 0; i < 20; i++)
    {
        printf("%d ", i, array[i]);
    }

    // free()
    free(ptr);
    free(array);

    return 0;
}
```

Advantages of pointers

1. Pointers enable flexible memory usage by allocating and releasing memory during program execution.
2. Pointers reduce memory consumption by sharing and referencing data instead of making unnecessary copies.
3. Pointers allow efficient manipulation of complex data structures through direct access and modification of memory elements.

4. Pointers enable efficient passing of large data structures or arrays to functions without copying the entire data.
5. Pointers provide direct memory access and greater control over program execution for handling complex tasks efficiently.

Disadvantages of Pointers:

1. Pointers require careful memory management to prevent memory leaks and undefined behavior.
2. Using pointers introduces complex syntax and concepts, increasing the likelihood of errors in the code.
3. Null and dangling pointers can cause crashes or unexpected behavior if not handled properly.
4. Pointers are generally more challenging to use compared to other data types.
5. Pointers are more prone to errors than other data types and can lead to memory leaks if not used and freed correctly.

Applications of Pointers:

1. Pointers are commonly used for dynamic data structures like linked lists and trees, allowing them to grow or shrink as needed.
2. Manipulating strings, such as copying, concatenation, and searching, relies on pointers.
3. Function pointers enable dynamic function invocation and are used in advanced techniques like callbacks and event handling.
4. Pointers provide direct access to memory, allowing low-level operations like hardware manipulation and bit-level operations.
5. Dynamic memory allocation is achieved using pointers, which is useful for creating linked lists, trees, and other data structures.