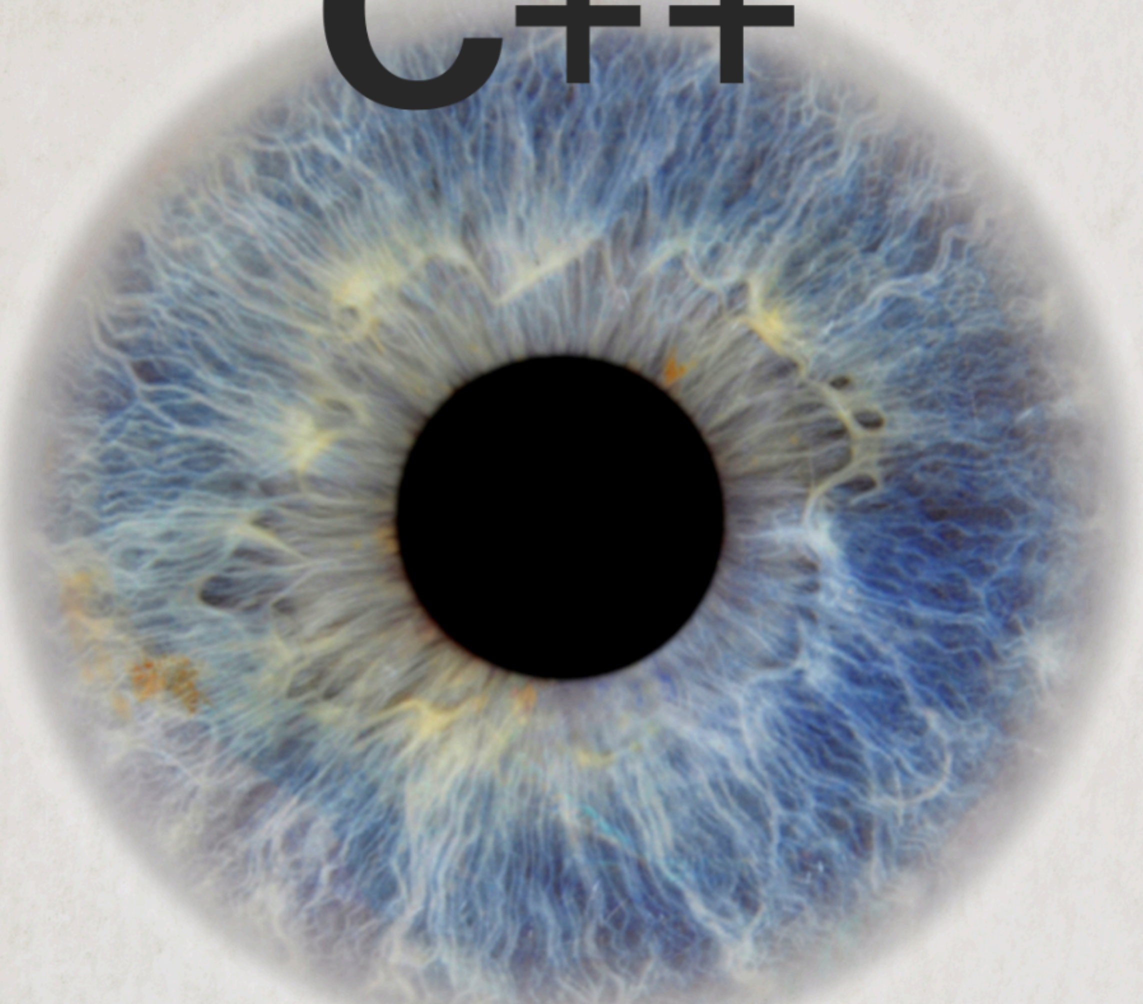
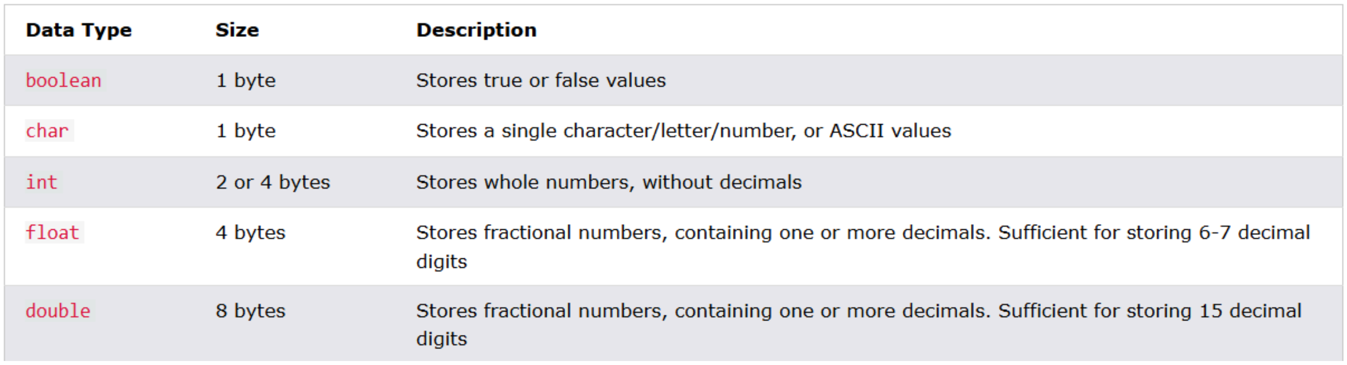
dark sunshine: C++



S, TANMAY

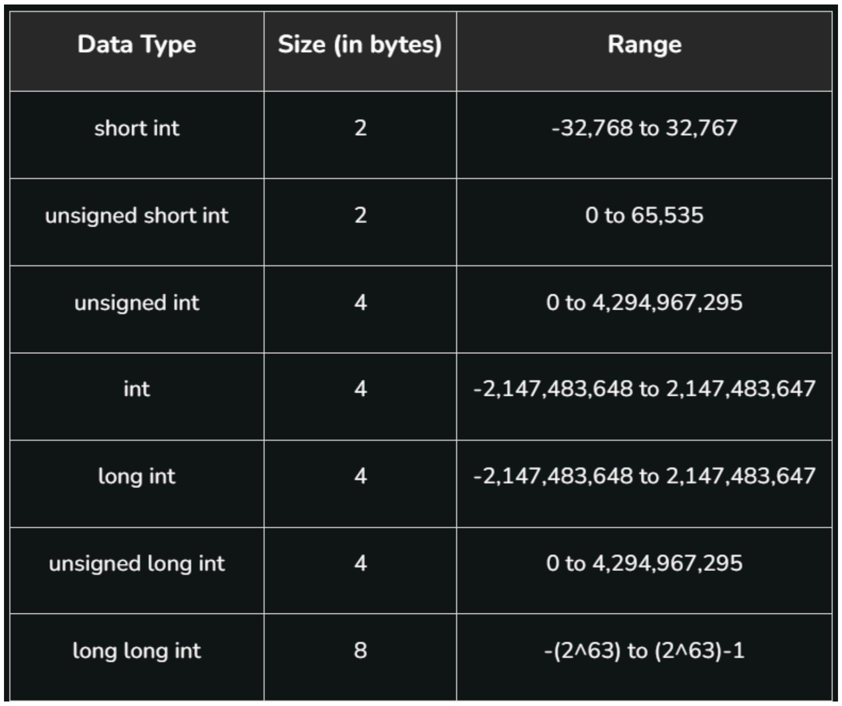
CONTENTS

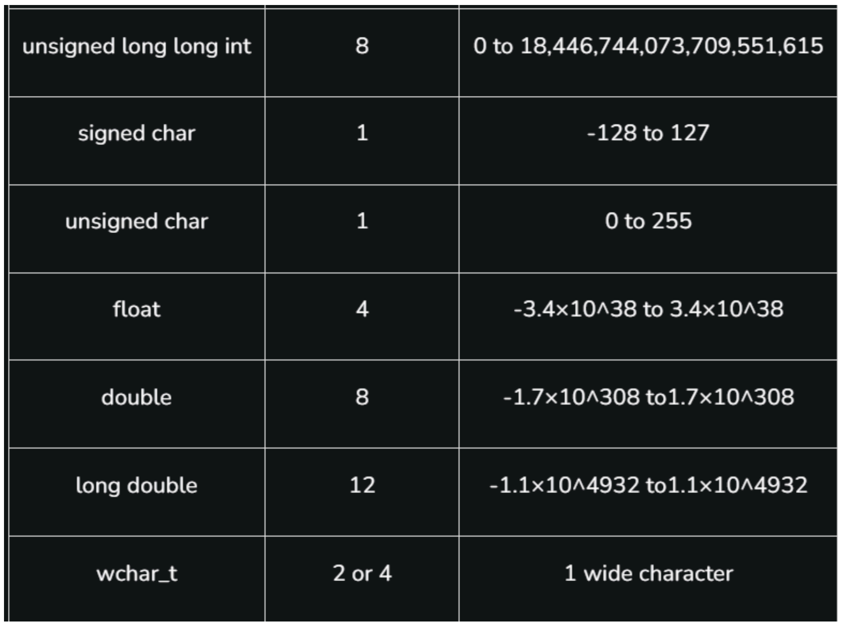
C++: Data Types



Data Type	Size	Description
`boolean`	1 byte	Stores true or false values
`char`	1 byte	Stores a single character/letter/number, or ASCII values
`int`	2 or 4 bytes	Stores whole numbers, without decimals
`float`	4 bytes	Stores fractional numbers, containing one or more decimals. Sufficient for storing 6-7 decimal digits
`double`	8 bytes	Stores fractional numbers, containing one or more decimals. Sufficient for storing 15 decimal digits

Modifiers:





Data Type	Size (in bytes)	Range
short int	2	-32,768 to 32,767
unsigned short int	2	0 to 65,535
unsigned int	4	0 to 4,294,967,295
int	4	-2,147,483,648 to 2,147,483,647
long int	4	-2,147,483,648 to 2,147,483,647
unsigned long int	4	0 to 4,294,967,295
long long int	8	-(2^63) to (2^63)-1
unsigned long long int	8	0 to 18,446,744,073,709,551,615
signed char	1	-128 to 127
unsigned char	1	0 to 255
float	4	-3.4×10^38 to 3.4×10^38
double	8	-1.7×10^308 to1.7×10^308
long double	12	-1.1×10^4932 to1.1×10^4932
wchar_t	2 or 4	1 wide character

Struct:

- Structures (also called structs) are a way to group several related variables into one place. Each variable in the structure is known as a member of the structure.
- Unlike an array, a structure can contain many different data types (int, string, bool, etc.).
- Code

```
struct myDataType { // This structure is named "myDataType"
  int myNum;
  string myString;
};
```

- **Usecases:**
    - Structs can be used as user-defined datatypes
    - We can create variables of type struct
    - Encapsulation of data
    - Structs can behave like classes in C++, they can have their own constructors and destructors
    - We can define methods in structs in C++, NOT in C language

Enum:

- An enum is a special type that represents a group of constants (unchangeable values).
- Default start from 0, but you can assign start value to starting constant.
- Code

```
#include <bits/stdc++.h>
using namespace std;
enum week {
    monday = 1,
    tuesday,
    wednesday,
    thursday,
    friday,
    saturday,
    sunday
};
int main() {
    week today = wednesday;
    cout << today << endl;
    return 0;
}
```

- **Usecases:**
    - Heard of READ, WRITE, EXECUTE? That's where these are used
    - Code:

```cpp
#include <bits/stdc++.h>
using namespace std;
enum permission {
    EXECUTE = 1,
    WRITE = 2,
    READ = 4
};

int main() {

    int permission = EXECUTE | READ | WRITE;
    cout << permission << endl; // output 7 (gave all permissions)
    // chmod u=rwx,g=rx,o=r myfile
    // example "sudo chmod +777 etc/file.network"
    return 0;
}
```

Union:

- union is a user-defined datatype in which we can define members of different types of data types just like structures.
- But one thing that makes it different from structures is that the member variables in a union share the same memory location, unlike a structure that allocates memory separately for each member variable.
- The size of the union is equal to the size of the largest data type.
- Like structs unions can have members, constructors and destructors in C++, NOT in C.
- **Usecases:**
    - When the available memory is limited, it can be used to achieve memory efficiency.
    - It is used to encapsulate different types of data members.
    - It helps in optimizing the performance of applications.
- If you allocate values to all variables in a union, only the last one retains and others give garbage values
- Code:

```cpp
union myunion {
public:
    int data;
    int age;
    char grade;

    myunion(int d, int a, char g) {
        this->age = a;
        this->data = d;
        this->grade = g;
        }
    };
```

We can define anonymous struct, enum and union in a class,

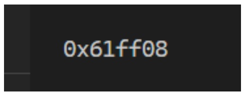
```
class student {
public:
    string name;

    struct {
        int standard;
        char grade;
    };

    enum {
        EXEC = 1,
        READ = 2,
        WRITE = 4
    };

    union {
        int enroll;
        int rank;
    };
};
```

C++ Pointers

Pointers are symbolic representations of addresses. They enable programs to simulate call-by-reference as well as to create and manipulate dynamic data structures. Iterating over elements in arrays or other data structures is one of the main use of pointers.

- Definition in C++:

```
int a = 12;
int *addr = &a;
```

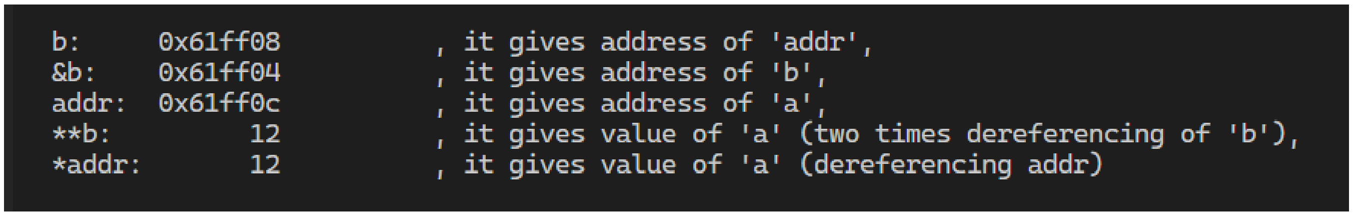


```
0x61ff08
```

- Why is there a need to specify data type for pointers as well? Since, it just stores an address then why specify data type? Why can't use int * instead of double * for a double data type?
    - The reason we associate data type with a pointer is that it knows how many bytes the data is stored in. When we increment a pointer, we increase the pointer by the size of the data type to which it points.

- Pointer to a Pointer?

```
int a = 12;
int *addr = &a;
int **b = &addr;
```



```
b:      0x61ff08          , it gives address of 'addr',
&b:     0x61ff04          , it gives address of 'b',
addr:   0x61ff0c          , it gives address of 'a',
**b:          12          , it gives value of 'a' (two times dereferencing of 'b'),
*addr:        12          , it gives value of 'a' (dereferencing addr)
```

References and Pointers:

- There are 3 ways to pass C++ arguments to a function:
    - Call-By-Value

```
int find_val1(int n) {
    cout << "pass by value: " << n << endl;

    return 0;
}

find_val1(n);
```

- Call-By-Reference with a Pointer Argument

```
int find_val2(int *n) {
    cout << "pass by reference, pointer: " << "address of n: " << n
<< " value of dereferenced n: " << *n << endl;

    return 0;
}

find_val2(&n);
```

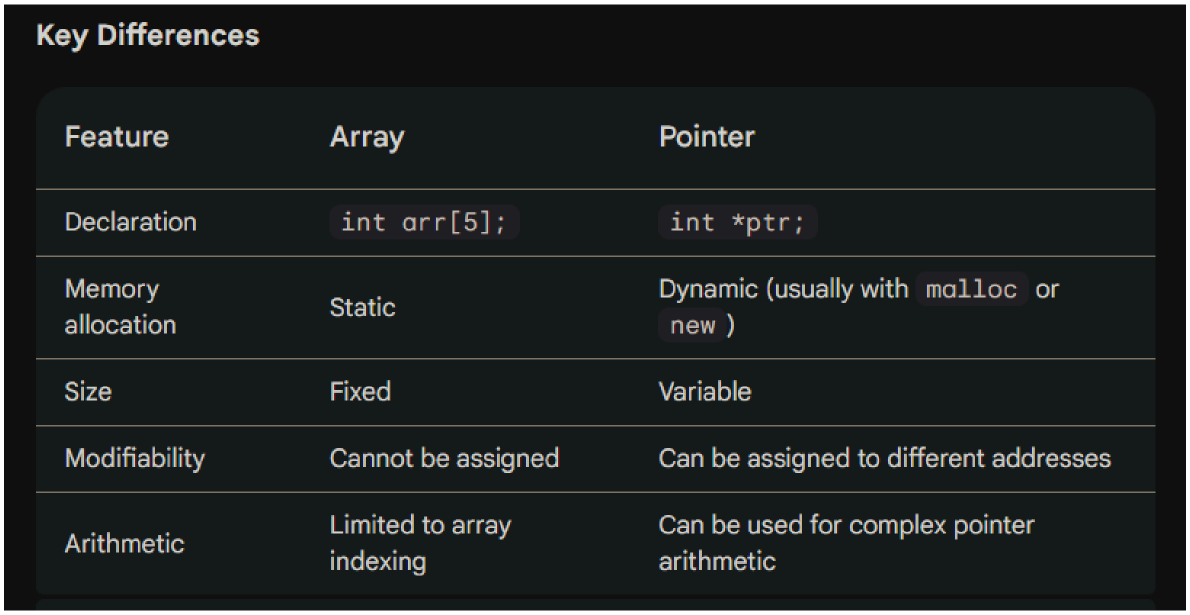
- Call-By-Reference with a Reference Argument

```
int find_val3(int &n) {
    cout << "pass by reference, reference argument: " << "address of
n: " << &n << " value of n: " << n << endl;
}

find_val3(n);
```

- Array names are also pointers in C++
  - An array name can often be treated as a constant pointer to the first element of the array
  - Are vectors too pointer names?
    - No, they aren't. C++ internally manages vectors by allocating memory dynamically. Vectors are dynamically resizable class, while arrays aren't class.

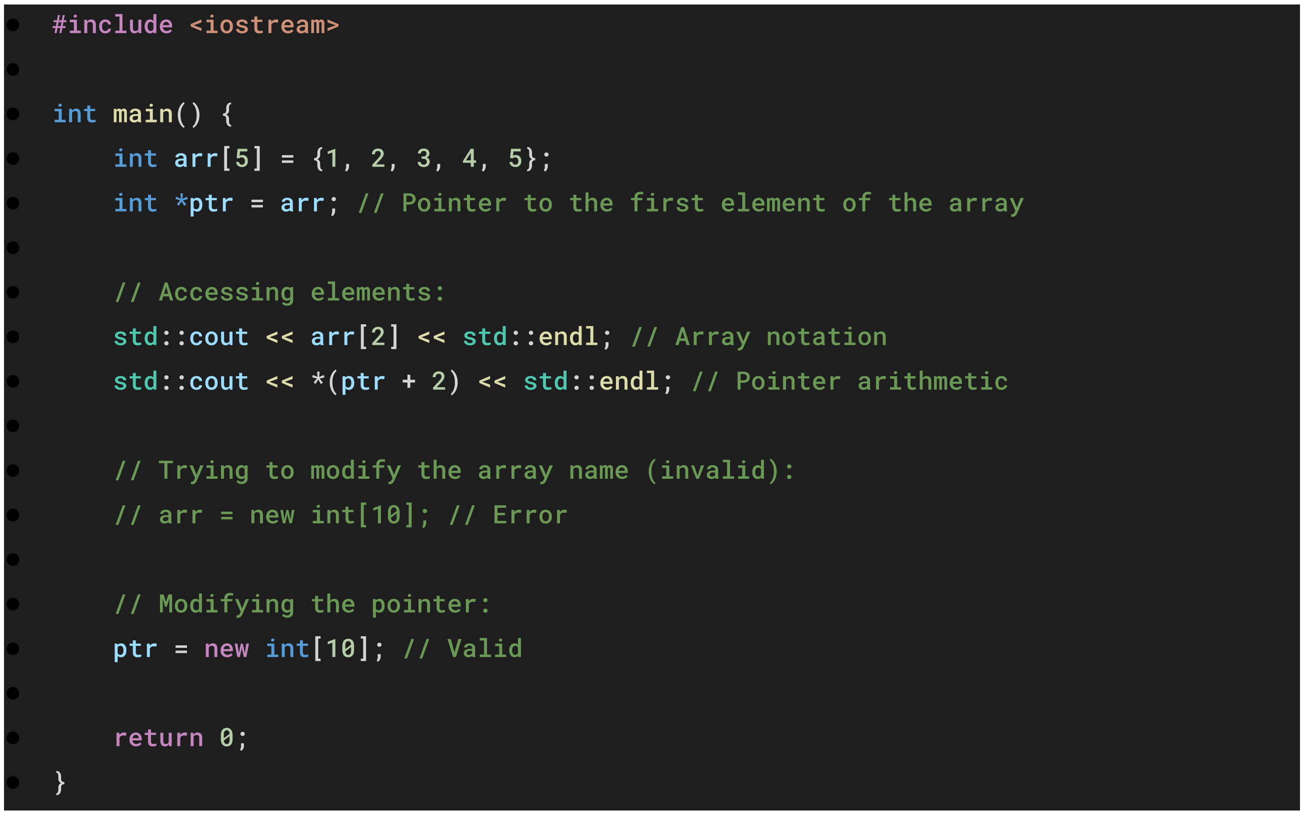
```
void tanmay() {
    int arr[] = {1, 2, 3, 4, 5};
    cout << "accessing using normal method: " << endl;
    cout << arr[0] << " " << arr[1] << " " << arr[2] << endl
         << endl;

    // pointer to arr
    int *ptr = arr;

    cout << "accessing using pointer name dereferencing: " << endl;
    cout << *ptr << " " << *(ptr + 1) << " " << *(ptr + 2) << endl
         << endl;

    cout << "accessing using normal method for pointer name: " << endl;
    cout << ptr[0] << " " << ptr[1] << " " << ptr[2] << endl
         << endl;
    cout << "accessing using array name dereferencing: " << endl;
    cout << *(arr) << " " << *(arr + 1) << " " << *(arr + 2) << endl;
}
```

Array name are pointers to first element, but pointers and arrays are different



Key Differences

Feature	Array	Pointer
Declaration	`int arr[5];`	`int *ptr;`
Memory allocation	Static	Dynamic (usually with `malloc` or `new`)
Size	Fixed	Variable
Modifiability	Cannot be assigned	Can be assigned to different addresses
Arithmetic	Limited to array indexing	Can be used for complex pointer arithmetic



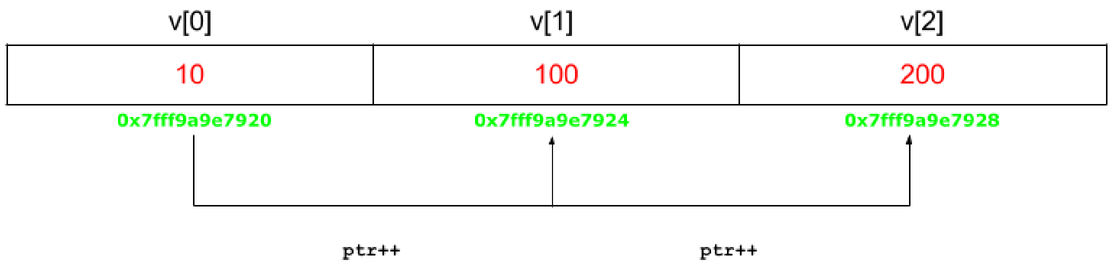
```
#include <iostream>

int main() {
    int arr[5] = {1, 2, 3, 4, 5};
    int *ptr = arr; // Pointer to the first element of the array

    // Accessing elements:
    std::cout << arr[2] << std::endl; // Array notation
    std::cout << *(ptr + 2) << std::endl; // Pointer arithmetic

    // Trying to modify the array name (invalid):
    // arr = new int[10]; // Error

    // Modifying the pointer:
    ptr = new int[10]; // Valid

    return 0;
}
```

Size of pointer remains same for all data types (based on gcc compiler):
Even though ptr_arr points to arr, it’s size is 4 bytes.

```
    int a = 0;
    char b = 'a';
    long c = 1l;
    double d = 0.0;
    long long e = 0L;

    int *ptra = &a;
    char *ptrb = &b;
    long *ptrc = &c;
    double *ptrd = &d;
    long long *ptre = &e;

    cout << sizeof(ptra) << endl
         << sizeof(ptrb) << endl
         << sizeof(ptrc) << endl
         << sizeof(ptrd) << endl
         << sizeof(ptre) << endl;

    int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int *ptr_arr = arr;

    cout << sizeof(arr) << endl
         << sizeof(ptr_arr) << endl;
```

Pointer Expressions and Pointer Arithmetic:

- incremented (++)
- decremented (—)
- an integer may be added to a pointer (+ or +=)
- an integer may be subtracted from a pointer (– or -=)
- difference between two pointers (p1-p2)
- (Note: Pointer arithmetic is meaningless unless performed on an array.)



Advanced Pointer Notation:

```
int nums[2][3]  =  { { 16, 18, 20 }, { 25, 26, 27 } };
```

- In general, nums[i][j] is equivalent to *(*(nums+i)+j)

Pointer Notation	Array Notation	Value
*(*nums)	nums[0] [0]	16
*(*nums+1)	nums[0] [1]	18
*(*nums+2)	nums[0] [2]	20
((nums + 1))	nums[1] [0]	25
((nums + 1)+1)	nums[1] [1]	26
((nums + 1)+2)	nums[1] [2]	27

Types:

- **String literals pointers:**

```
const char *str = "Hello World!";
cout << str << endl;
```

- **Void Pointers:**
  - Void pointers have great flexibility as they can point to any data type. There is a payoff for this flexibility.
  - These pointers cannot be directly dereferenced.
  - They have to be first transformed into some other pointer type that points to a concrete data type before being dereferenced.

```
int a = 12;
void *ptr = &a;
int *next = (int *)ptr;

cout << ptr << " " << &ptr << " " << *next << endl;
```

- **Invalid pointers:**
  - A pointer should point to a valid address but not necessarily to valid elements (like for arrays). These are called invalid pointers.
  - Uninitialized pointers are also invalid pointers.
  - `int *ptr1;`
  - `int arr[10];`
  - `int *ptr2 = arr+20;`

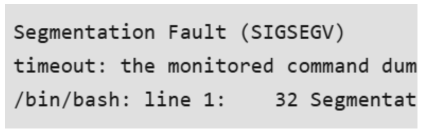
  - Here, ptr1 is uninitialized so it becomes an invalid pointer and ptr2 is out of bounds of arr so it also becomes an invalid pointer. (Note: invalid pointers do not necessarily raise compile errors)
- **NULL Pointers:**
  - A null pointer is a pointer that point nowhere and not just an invalid address. Following are 2 methods to assign a pointer as NULL:
  - `int *ptr1 = 0;`
  - `int *ptr2 = NULL;`

- **Dangling Pointer:**
  - A pointer pointing to a memory location that has been deleted (or freed) is called a dangling pointer.
  - Such a situation can lead to unexpected behavior in the program and also serve as a source of bugs in C programs.

```
    int* ptr = (int*)malloc(sizeof(int));

    // After below free call, ptr becomes a dangling pointer
    free(ptr);
    printf("Memory freed\n");

    // removing Dangling Pointer
    ptr = NULL;
```

  - Reasons for dangling pointer:
    - Memory deleted
    - Function returning pointer of local variable
    - When variable goes out of scope, eg, defining pointer within small scope

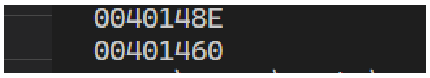
- **Wild Pointer:**
  - A pointer that has not been initialized to anything (not even NULL) is known as a wild pointer. The pointer may be initialized to a non-NULL garbage value that may not be a valid address.
  - `dataType *pointerName;`



```
Segmentation Fault (SIGSEGV)
timeout: the monitored command dum
/bin/bash: line 1:    32 Segmentat
```

Advantages of Pointers:
- Pointers reduce the code and improve performance. They are used to retrieve strings, trees, arrays, structures, and functions.
- Pointers allow us to return multiple values from functions.
- In addition to this, pointers allow us to access a memory location in the computer’s memory.
- Functions also have addresses in memory
- Function names too acts as pointers to their addresses in memory

```
#include <bits/stdc++.h>
using namespace std;

void tanmay() {
    cout << "Tanmay S." << endl;
}

int main() {

    printf("%p\n", main);
    printf("%p\n", tanmay);

    return 0;
}
```



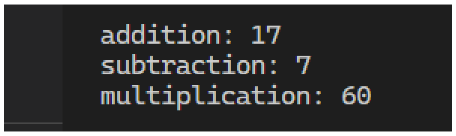
```
0040148E
00401460
```

- **Function Pointers:**
  - Unlike normal pointers, a function pointer points to code, not data. Typically a function pointer stores the start of executable code.
  - Unlike normal pointers, we do not allocate de-allocate memory using function pointers.
  - A function's name can also be used to get functions' address. For example, in the below program, we have removed address operator '&' in assignment. We have also changed function call by removing *, the program still works.

```
void computer(int n) {
    cout << "value of n: " << n << endl;
}


void (*ptr)(int) = &computer;

ptr(10);    // either this way
(*ptr)(11); // or this way both ways it works
```

  - Like normal pointers, we can have an array of function pointers. Below example in point 5 shows syntax for array of pointers.
  - These can be used in place of switch case statements.
  - You can pass pointers to a function as well, and pointer functions in another function as well

```
void add(int a, int b) {
    cout << "addition: " << (a + b) << endl;
}

void subtract(int a, int b) {
    cout << "subtraction: " << (a - b) << endl;
}

void multiply(int a, int b) {
    cout << "multiplication: " << (a * b) << endl;
}

void (*fun_ptr[])(int, int) = {&add, &subtract, &multiply};
int a = 12, b = 5;
(*fun_ptr[0])(a, b);
(*fun_ptr[1])(a, b);
(*fun_ptr[2])(a, b);
```



```
addition: 17
subtraction: 7
multiplication: 60
```

- **Functions returning pointers:**

```
#include <bits/stdc++.h>
using namespace std;

// instead make x static or global
int *func() {
    int x = 100;
    int *ptr = &x;

    return ptr;
}
int main() {
    int *addr = func();
    fflush(stdin);

    printf("%p\n", addr);

    return 0;
}
```

Difference between pointer to constant, constant pointer and constant pointer to constants:

- **Pointer to Constant:**
  - Here, the pointed variable is constant, it can't change, but the pointer can change and point to another const variable

```
    const int high = 100;
    const int *ptr = &high;
    cout << high << " " << *ptr << endl;

    const int low = 0;
    ptr = &low;
    cout << low << " " << *ptr << endl;
```

- **Constant Pointer:**
  - This pointer can only hold single address, when tried to change to another memory, it gives read-only variable error

```
    int high = 100;
    int *const ptr = &high;

    cout << high << " " << *ptr << endl;

    int low = 0;
    ptr = &low; // gives error, read-only variable 'ptr'
    cout << low << " " << *ptr << endl;
```

  - The value residing in *ptr can be changed, but not memory

```
    *ptr = 1000;
    cout << *ptr << endl
         << ptr << endl;
```

- **Constant pointer to constant:**
  - Constant value and constant memory in pointer

```
    const int high = 100;
    const int low = 0;

    const int *const ptr = &high;
    cout << high << " " << *ptr << endl;

    ptr = &low; // gives error, read-only variable 'ptr'
    *ptr = 1000; // assignment of read-only location error
```

Malloc, Calloc, Realloc and Free
Header file: `<stdlib.h>`

- **Malloc**: memory allocation is used to assign a large chunk of memory and returns a void pointer to it's base address. It just gives us a large chunk of memory, do whatever you want with it.
    - It takes only size of memory as argument

```
// gives me a memory block of 4 * 4 = 16 size (as per gcc)
int *ptr_malloc = (int *)malloc(4 * sizeof(int));

for (int i = 0; i < 4; i++) {
    ptr_malloc[i] = i + 1;
}

for (int i = 0; i < 4; i++) {
    cout << ptr_malloc[i] << " ";
}
cout << endl;
```

- **Calloc**: does the same but in a contiguous manner and asks for number of memory blocks with size of each block.
    - It takes number of blocks and size of each blocks as arguments
    - In following case, I've used memory using int but assigned to a char pointer

```
// gives me 5 block of size 4 each = 20 bytes (as per gcc)
// since I'm storing character here, of 1 byte each
// it only uses 1 byte
char *ptr_calloc = (char *)calloc(5, sizeof(int));

for (int i = 0; i < 5; i++) {
    ptr_calloc[i] = 'a' + i;
}

for (int i = 0; i < 5; i++) {
    cout << ptr_calloc[i] << " ";
}
cout << endl;

for (int i = 0; i < 5; i++) {
    printf("%p %d\n", (ptr_calloc + i), sizeof(ptr_calloc[i]));
}
cout << endl;
```

- **Free**: frees up the space or deallocates the space given using malloc or calloc.

```
free(ptr_calloc);
```

- **Realloc**: this one is used to change the size of memory issued using malloc or calloc at runtime, it can increase the size or decrease the size.

```
ptr_malloc = (int *)realloc(ptr_malloc, 10 * sizeof(int));

for (int i = 4; i < 10; i++) {
    ptr_malloc[i] = i + 1;
}

for (int i = 0; i < 10; i++) {
    cout << ptr_malloc[i] << " ";
}
cout << endl;
```

Pointer Usecases:

- Dynamic memory allocation
- Used in data structures like linked lists, trees and graphs where we don't know the final size of data
- Used in low level memory operations for writing device drivers
- Function pointers are used in interrupt handling and callback mechanisms (in real world you don't see all interrupts coming up at once, they're executed sequentially one after the other and OS doesn't create instances of them, it simply uses pointers and addresses to call those interrupt handlers by reference)
- Used in accessing array elements, array of any dimension

Are Pointers Integers?

- No, they're not. Pointer is an address and a positive hexadecimal number.

Ways to initialize pointers?

- Pointer variables are initialized by one of the following ways.
    - Static Memory Allocation

```
int data = 10;
int *ptr = &data; // Pointer to a statically allocated variable
```

    - Dynamic Memory Allocation

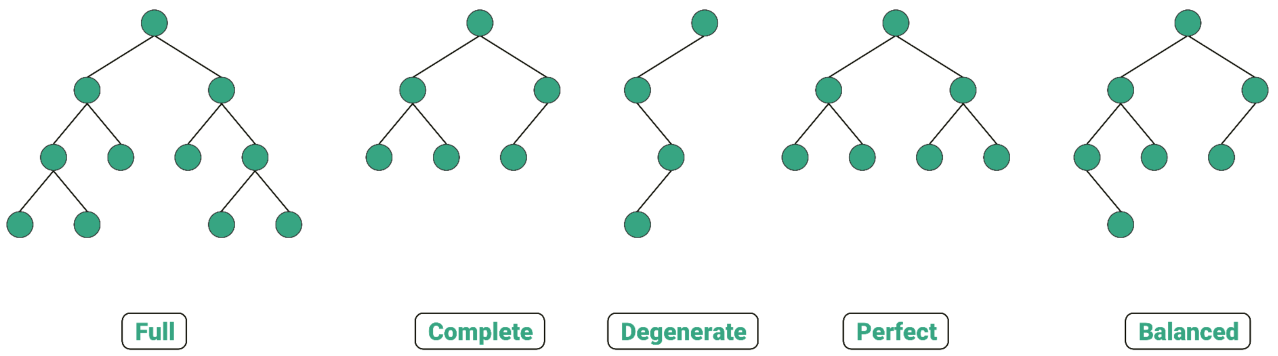
```
int *ptr = (int*)malloc(sizeof(int)); // Allocate memory for an integer
if (ptr != nullptr) {
    *ptr = 20; // Assign a value to the allocated memory
}
```

Binary Trees

Binary Trees are non linear data structures comprising of nodes. Each node has it's own data and pointers to left and right child nodes.

Types:

- **Basis of Number of Children:**
    - Full BT: each node has either 0 or 2 children.
    - Degenerate BT: each parent node has 1 child only. Like a linked list.
    - Skewed BT: a type of Degenerate BT, it has either 1 or no child such that all child are of same type (left skewed or right skewed).



- **Basis of Completion of Level:**
    - Complete BT: all the levels of BT are completely filled except the last which also fills from left.
        - **Root:** which has no parent node or no incoming edge.
        - **Child:** which has some incoming edge to it.
        - **Siblings**: sharing same parent node.
        - **Degree of Node**: number of direct childrens.
        - **External and Internal Nodes**: Leaf are external and rest internal nodes.
        - **Level**: count nodes from root to destination node (start from root as 0).
        - **Height**: count edges from root to destination node.
        - **Depth**: defined for whole tree, max height of any node.
        - In complete BT all leaves are at same depth.
        - Nodes in CBT at depth 'd' are $\mathbf{2^d}$.
        - In a CBT having n nodes, height of tree is **log (n + 1)**.
    - **Perfect BT:** all levels are completely filled, each node has two children.
    - **Balanced BT:** BT where the absolute difference between heights of children is not more than 1. And same applies to it's subtrees.
- **Basis of Node Value:**
    - **BST**: binary search Trees follow the pattern such that the value of left children is strictly less than root and the value of right children is strictly more than root. Same applies to it's subtrees.

- **AVL Trees:**
    - Self-balancing binary search tree.
    - Height difference between left and right subtrees of any node is at most 1.
    - Guarantees efficient search, insertion, and deletion operations.

- **Red-Black Trees**:
    - Self-balancing binary search tree.
    - Uses colors (red and black) to maintain balance.
    - Provides good performance guarantees, but slightly less strict balancing than AVL trees.
    - **Usecases:**
        - These are quite fast and so used in cache lookup for frequent data access
        - Used in databases for fast indexing
        - Also used in routing algorithms
- **B Trees**:
    - Balanced tree that allows multiple keys and children in a node.
    - Optimized for disk-based storage.
    - Used in databases and file systems for efficient data retrieval.
    - **Usecases:**
        - Same as AVL trees
- **B+ Trees:**
    - Variation of B-trees where all data is stored in leaf nodes.
    - Internal nodes only store keys for directing searches.
    - Excellent for indexing and sequential access.
    - **Usecases:**
        - Same as AVL trees
- **Segment Trees:**
    - Used for efficient range query operations.
    - Each node represents a range of data.
    - Supports operations like finding the sum, minimum, maximum, etc., within a given range.
    - **Usecases:**
        - Range based statistics like, percentile, variance, standard deviation
        - Image processing for dividing an image based on color, texture
- **B* trees:**
    - Variation of B-trees with higher minimum and maximum number of keys per node.
    - Aims to improve space utilization and reduce the number of disk accesses.
    - **Usecases:**
        - Same as AVL trees

Implementation of BT:

- Code

```
class Node {
public:
    int data;
    Node *left;
    Node *right;

    Node(int value) {
        this->data = value;
        this->left = this->right = NULL;
    }
};
```

Insertion in Binary Tree:

- No fixed approach:
    - Check if left child exists, if not insert new node there.
    - Check if right child exists, if not insert new node there.
    - If not found on either left or right, then iterate to left subtree or right subtree (based on your choice).
- Fixed approach (Level Order):
    - To insert node at first empty found place.
    - Use a queue and insert root.
    - Check if left subtree exists, if yes insert into queue. If not then place the node there.
    - Similarly check for right subtree.

```
    static Node *build_tree(Node *&root, int value) {
        if (!root) {
            return new Node(value);
        }
        if (!root->left) {
            root->left = new Node(value);
            return root;
        }
        if (!root->right) {
            root->right = new Node(value);
            return root;
        }
        if (root->left) {
            root->left = build_tree(root->left, value);
        } else {
            root->right = build_tree(root->right, value);
        }
        return root;
    }
```

Traversal Techniques:

- Depth First Search (DFS):
    - **Preorder**:
        - Traversal is like Node-Left-Right, visit node then go to left subtree and then to right subtree.
        - Code

```
    static void preorder(Node *&root) {
        if (!root) {
            return;
        }
        cout << root->data << " ";
        preorder(root->left);
        preorder(root->right);
    }
```

    - **Postorder**:
        - Traversal is like Left-Right-Node
        - Code

```
    static void postorder (Node *&root) {
        if (!root) {
            return;
        }
        postorder (root->left);
        postorder (root->right);
        cout<< root->data<< " ";
    }
```

    - **Inorder**:
        - Traversal is like Left-Node-Right
        - Code

```
    static void inorder (Node *&root) {
        if (!root) {
            return;
        }
        inorder (root->left);
        cout<< root->data<< " ";
        inorder (root->right);
    }
```

- Breadth First Search (BFS):
    - **Level Order traversal**:
        - We traverse each level from left to right or vice versa
        - Code

```
    static void level_order(Node *&root) {
        if (!root) {
            cout << "Empty tree!";
            return;
        }

        queue<Node *> q;
        q.push(root);
        while (!q.empty()) {
            int size = q.size();
            for (int i = 0; i < size; i++) {
                Node *front = q.front();
                q.pop();
                cout << front->data << " ";
                if (front->left) {
                    q.push(front->left);
                }
                if (front->right) {
                    q.push(front->right);
                }
            }
            cout << endl;
        }
    }
```

Deletion from BT:

- Deletion of leaf nodes (external nodes) is very simple it can be done without shifting any node.
- But deletion of nodes in between the tree, the internal nodes is quite tricky. You gotta find the right most node and replace the data with to_delete node and delete the right one node.
- Code

```
static void find_deepest(Node *&root, Node *&deepest, Node *&parent) {
        queue<Node *> q;
        q.push(root);
        while (!q.empty()) {
            Node *front = q.front();
            q.pop();
            if (!front->left && !front->right) {
                deepest = front;
            }
            if (front->left) {
                q.push(front->left);
                parent = front;
            } else if (front->right) {
                q.push(front->right);
                parent = front;
            }
        }
    }

    static void delete_node(Node *&root, int value) {
        if (!root) {
            return;
        }
        if (root->data == value) {
            if (!root->left && !root->right) {
                root = NULL;
                delete root;
                return;
            }
            Node *parent = NULL;
            Node *deepest = NULL;
            find_deepest(root, deepest, parent);
            root->data = deepest->data;
            if (parent->left == deepest) {
                parent->left = NULL;
                delete deepest;
            } else if (parent->right == deepest) {
                parent->right = NULL;
                delete deepest;
            }
            return;
        }
        delete_node(root->left, value);
        delete_node(root->right, value);
    }
```

Searching in a BT:

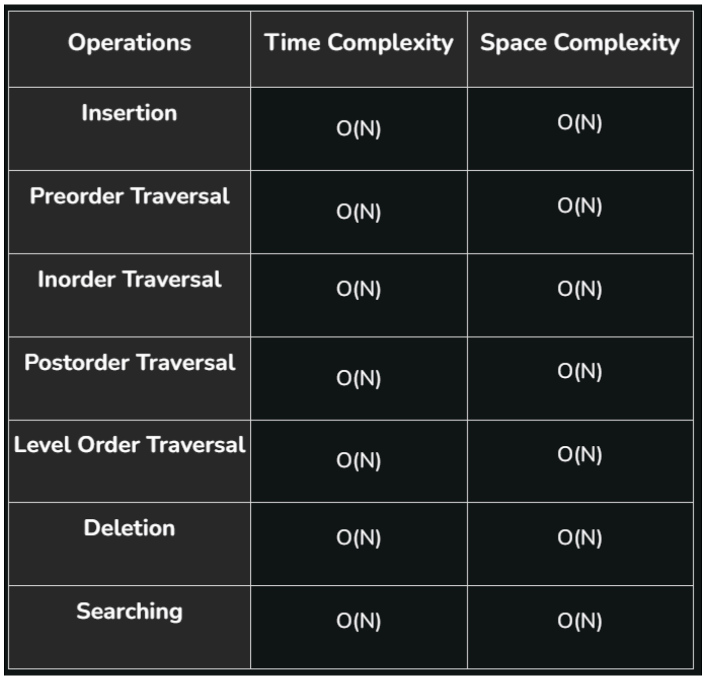
- Normal search operation, can use any of traversals we studied above.

Morris Traversal:

Questions:

1. Height of BT:
   - Going to use recursion
   - Find height of left subtree
   - Find height of right subtree
   - Find max of left and right, add 1 of current root node

    - Return final value
2. Level of a target node:
    - We'll use level order traversal, set initial level = 1 and increase after each for loop
    - If we find front->data == target, return level
    - Else return 0
3. Get size of BT:
    - We'll use recursion, find size of left subtree, then right subtree, add 1 to answer and return
    - If null node, return 0
    - If leaf node, return 1



Operations	Time Complexity	Space Complexity
Insertion	O(N)	O(N)
Preorder Traversal	O(N)	O(N)
Inorder Traversal	O(N)	O(N)
Postorder Traversal	O(N)	O(N)
Level Order Traversal	O(N)	O(N)
Deletion	O(N)	O(N)
Searching	O(N)	O(N)

Applications/ Advantages and Disadvantages of BTs:
- Applications (most used BT is a BST):
    - File Systems are stored in a tree structure and root folder at root node
    - Database Systems also use BSTs for searching and sorting records
    - Sorting algorithms also utilize binary search trees as insertion in O(log n)
    - DOM in HTML, it's a tree structure with childrens arranged (it used BT)
    - Routing tables utilze variant of BT called trie to link routers in a network
    - File explorers in almost every OS
    - BTs are also utilized in ranking of web pages
    - Google Servers also utilize a variation of BTs called trie for providing fast results
    - Data Compression
        - Huffman encoding, leaves represent characters and their frequency of occurence
- Advantages:
    - Efficient searching
    - Ordered traversals

    - Fast insertion and deletion
    - Easy to implement
    - Useful in places that require sorting
- Disadvantages:
    - A slight unbalance in the tree can make sub-structure worthless
    - Balancing algorithms in AVL, red-black trees are complex to implement

Binary Search Trees

These are a variant of Binary Trees but with the property of holding a sorted structure in their values. The value of left node < root node < right node, this exists for each node except for leaf nodes, i.e., both left and right subtrees are also BSTs.

Generally there aren't any duplicates allowed in BSTs, but if allowed then should be kept either to left or to right (fixed fashion to avoid confusion).

Structure:

```
class Node {
public:
    int data;
    Node *left;
    Node *right;

    Node(int value) {
        this->data = value;
        this->left = this->right = NULL;
    }
};
```

Build height balanced BST from array:

- Code:

```
static Node *build_tree(Node *&root, vector<int> &arr, int start, int end) {
    if (start > end) {
        return NULL;
    }

    int mid = (start + end) / 2;
    root = new Node(arr[mid]);
    root->left = build_tree(root->left, arr, start, mid - 1);
    root->right = build_tree(root->right, arr, mid + 1, end);
    return root;
}
```

Deletion in a BST:

- There are 3 possibilities when it comes to delete a node in a BST.
- 0 child case, 1 child case and 2 child case.
- If 0 children, then delete node and return NULL
- If 1 children, store whichever left node or right one child, delete node and return child
- If 2 children, find next inorder predecessor of node, make node->data = pred->data, then recursively delete predecessor

- Code:

```
static Node *deleteNode(Node *root, int x) {
    if (!root) {
        return NULL;
    }

    if (root->data == x) {

        // 0 child case
        if (!root->left && !root->right) {
            delete root;
            return NULL;
        }

        // 1 child case
        if (!root->right) {
            Node *temp = root->left;
            delete root;
            return temp;
        }

        if (!root->left) {
            Node *temp = root->right;
            delete root;
            return temp;
        }

        // 2 child case
        Node *temp = root->left;
        while (temp->right) {
            temp = temp->right;
        }

        root->data = temp->data;
        root->left = deleteNode (root->left, temp->data);
        return root;
    }

    else if (root->data > x) {
        root->left = deleteNode (root->left, x);
        }
        else {
            root->right = deleteNode (root->right, x);
        }

        return root;
    }
```

Applications/ Advantages/ Disadvantages:

- Applications same as AVL trees (databases, file systems, etc)
- Advantages:
    - Efficient Searching
    - Dynamic insertion and deletion
- Disadvantages:
    - If the structure becomes large, it becomes difficult to manage, so dividing the structure into subproblems is considered good