**DATA STRUCTURE**

**What is Data Structure:**

A data structure is a storage that is used to store and organize data. It is a way of arranging data on a computer so that it can be accessed and updated efficiently.

A diagram of data structure

Description automatically generated

\*ARRAY

**1.Array Searching Operation:**

#include<iostream>

using namespace std;

int main(){

    int array[] = {12,45,22,34,11,56};

    int size = sizeof(array)/sizeof(array[0]);

    cout<<"Enter the element you want to search : ";

    int element,c=0;

    cin>>element;

    for(int i=0;i<size;i++){

        if(array[i] == element){

                cout<<"Element Found at "<<(i+1)<<" Position"<<endl;

        }

        else{

c++;

        }

    }

    if(c==size)cout<<"Not Find"<<endl;

}

**2.Array Delete:**

#include<iostream>

using namespace std;

int main(){

    int a[5] = {23,12,11,90,22};

    int index=3;

    for(int i=index;i<5;i++){

        a[i] = a[i+1];

    }

    cout<<"After Delete : ";

    for(int i=0;i<4;i++){

cout<<" "<<a[i];

    }

}

**3.Array Insert:**

#include <iostream>

using namespace std;

int main() {

    int array[5] = {34, 12, 55, 11, 10};

    int n = 5;

    cout << "Enter the index at which you want to insert: ";

    int pos, element;

    cin >> pos;

    if (pos == 5) {

        cout << "Enter the element: ";

        cin >> element;

        array[pos] = element;

        n++;

    }

    else if (pos < 5) {

        cout << "Enter the element you want to insert: ";

        cin >> element;

        for (int i = n; i > pos; i--) {

            array[i] = array[i - 1];

        }

        array[pos] = element;

        n++;

    }

    else {

        cout << "\nArray index out of bounds";

    }

    cout << "\nNew Array: ";

    for (int i = 0; i < n; i++) {

        cout << " " << array[i];

    }

}

**Memory Access :**

A black background with a black square

Description automatically generated with medium confidence

**&array[index] = startlocation + index\*(size\_of\_data)**

**In the photo start location should be 567**

Here size of data defines the size of the DATA \_TYPE

If the array is int the size should be : 4 bytes

If the array is boolean type then the size should be 2bytes.

The name of the array always holds the memory address of the first element of the array.

A black rectangular object with white text

Description automatically generated

**A black background with white text

Description automatically generated**

**STRING**

A **string** is a sequence of characters, typically used to represent text. In most programming languages, a string is treated as a data type that can hold letters, numbers, symbols, and even spaces.

**Why we use getline(cin,string)?**

* It is used to take input of multiple words. In normal cases of cin>>string, we cant take input more than 1 word. So that’s why to take multiple words as input we use getline function in cpp.

**String Handling Functions:**

All the functions of string is stored in cstring header file.

1. Strlen() = returns the length of the string.
2. Strcat(string1,string2) – It actually concats two strings together. The string2 will be added at the end of string1. The value of string1 will be string1+string2.
3. #include<iostream>
4. #include<cstring>
5. #include<string.h>
6. using namespace std;
7. int main(){
8. char  s1[] = "sadman";
9. char s2[] ="sakib";
10. cout<<"After strcat(s1,s2) "<<strcat(s1,s2)<<endl;
11. cout<<"New Value of s1  = "<<s1<<endl;
12. }

3. strcmp(string1,string2) :

Returns **0** if the strings are **equal**.

Returns a **negative value** if str1 is lexicographically **less than** str2.

Returns a **positive value** if str1 is lexicographically **greater than** str2.

**What is lexicographically :**

The strcmp function in C uses lexicographical order based on ASCII values to compare two strings. It checks characters from left to right and returns

* **0** if both strings are equa
* a **negative value** if the first string is smaller,
* a **positive value** if the first string is larger.

**Lexicographical order** is a generalization of the way words are ordered in a dictionary. It is based on comparing sequences of elements (such as characters in strings) one by one, starting from the first position and moving to the next if they are equal. The comparison is typically based on the **alphabetical order** of characters or their **ASCII values** in programming.

4.strcpy(string1,string2) : Copies string 2 into string1. Then the value of string1 is auto updated by string2.

**All functions of string:**

**A screen shot of a computer screen

Description automatically generated**

A black screen with white text

Description automatically generated

**2D ARRAY**

2D Array is called the array of arrays. Which represents row and columns to store values into this.

Here is the declaration process of the array :

int minu[3][5] = {1,2,3,4,5,2,4,6,8,10,3,6,9,12,15};

int minu[3][5] = {{1,2,3,4,5},{2,4,6,8,10},{3,6,9,12,15}};

The curly brackets aren’t important to write but it helps to catch the values for a specific row and col.

**Some Important Notes :**

* If there are not enough elements in the curly braces to account for every single element in an array, the remaining elements will be filled out with garbage/zeros.
* Static and global variables are always guaranteed to be initialized to zero anyway, whereas auto or local variables are guaranteed to be garbage.
* **&array[i][j]=start\_location + (i \* (C \* size\_of\_data)) + (j \* size\_of\_data)**

**Declaration :**

**A screen shot of a computer code

Description automatically generated**

**Memory Access of 2D Array:**

* For an array int array[R][C]; and 0≤i<R; 0≤j<C.
  + array[i] = &array[i][0] represents the starting address of ith row.
  + array[i] skips i number of rows each with C number of elements from the start\_location of the array.
  + **So, &array[i][j] = start\_location + (i \*(c\*size\_of\_data) + (j \* size\_of\_data)**
  + **where size\_of\_data is counted in bytes here int=4bytes.**

**POINTER**

Pointer is a variable that stores the memory address of another variable.

Pointer is declared by specifying the data type it points to.

Example:

int variable = 45;

int \*pointer = &variable;

cout<<”Value of variable : ”<<variable<<endl; -> 45

cout<<”Address of variable : ”<<variable<<endl;->address

If we declare like this :

Int a=89;

Int \*p = &a;

\*p = holds the value

P = holds the address

A screen shot of a computer program

Description automatically generated

* A screen shot of a computer screen

  Description automatically generated

**Pointer in array :**

We all know the array name holds the memory address of the first element.

Int array[9] = {78,34,32,11,12};

Int \*p = array;

Here p = address , \*p = value

Here the \*p holds the first elements value and if you want to print the other elements value then you should use \*(p+i)

And to print the address you should use &p[i]

Here &p holds the memory location of variable p itself. Not related to array.

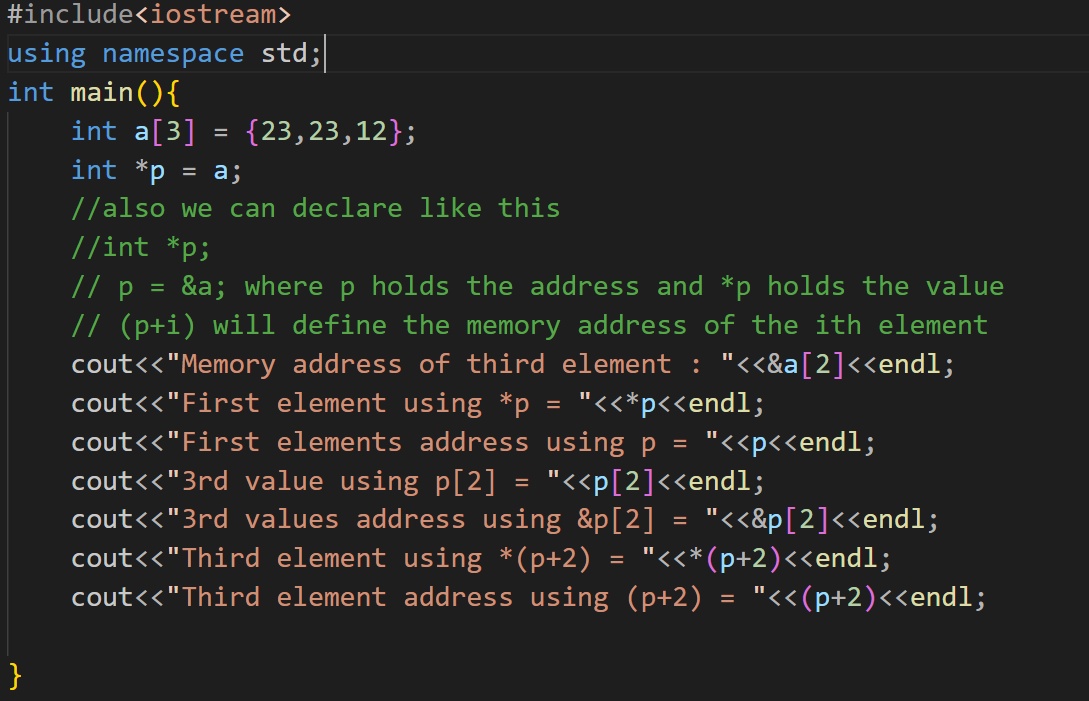
Example :

  also we can declare like this

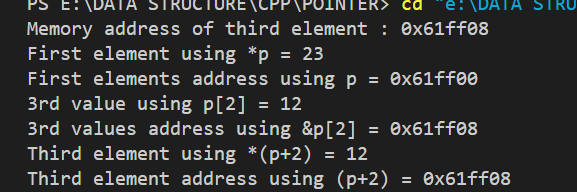
    int \*p;

  p = &a; where p holds the address and \*p holds the b value.

If a is the array then p[i] will hold the value of ith element. And &p[i] will hold the memory address of ith element.



OUTPUT :



**VOID POINTER**

Void pointer is a pointer with no data type.

A *void pointer* is a special type of pointer in C++ that can point to any data type. It’s declared with void\*, and since it doesn’t have a specific data type, it doesn’t know what kind of data it points to. This makes it very flexible, but before you can use it (like reading or modifying the data), **you must cast it to a specific pointer type.**

A screen shot of a computer program

Description automatically generated

**OUTPUT :**

A black background with white text

Description automatically generated

**NULL POINTER**

A *null pointer* is a special pointer in C and C++ that doesn’t point to any valid memory location. It’s used to indicate that the pointer intentionally doesn’t refer to any object or variable. You create a null pointer by assigning NULL to it.

int\* ptr = NULL;

* Do not confuse null pointers with void pointers. A null pointer is a value that any pointer may take to represent that it is pointing to "nowhere", while a void pointer is a special type of pointer that can point to somewhere without a specific type.

**DYNAMIC MEMORY ALLOCATION**

Dynamic memory allocation is when a program asks for memory while it's running, instead of before it starts running. This is useful because sometimes we don’t know exactly how much memory we’ll need until the program is already running.

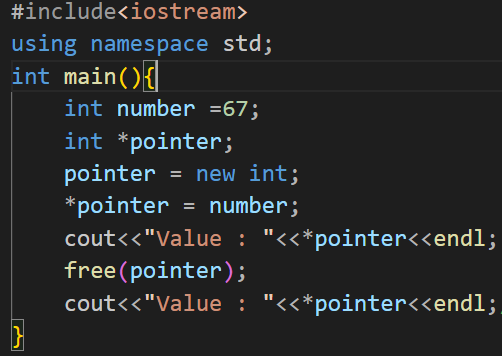
Imagine you're at a party and you don’t know how many people will come, so you wait and make space for each new guest as they arrive. Dynamic memory allocation works similarly—memory is given as the program needs it, rather than setting aside a fixed amount from the start.

**Declaration:**

int\* x = new int; // Allocated on the heap

\*x = 5; // Set the value

* **Memory Location:** The memory for \*x is allocated on the **heap**.
* **Lifetime:** The memory stays allocated until you explicitly free it using delete. You can keep this
* memory even after the function where it was created has ended.



The last cout will print garbage value because we have freed the value of number by using free function.

A screen shot of a computer code

Description automatically generated

A screen shot of a computer

Description automatically generated

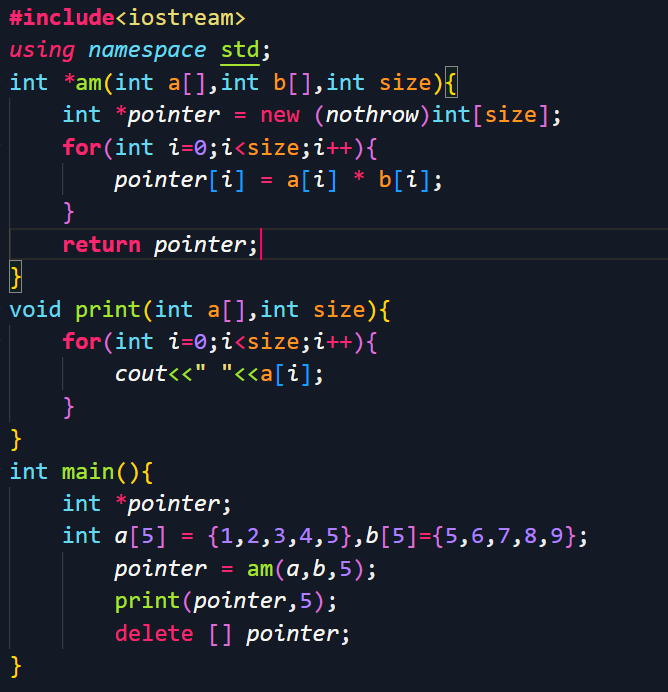
nothrow -> it is used not to throw errors if the dynamic memory allocation is failed.

A computer screen shot of a code

Description automatically generated

Here we have declared \*pointer .

Pointer = new(nothrow)int[number] means the pointer will point to an int type array with size of “number” which will be declared by the user. We used nothrow functions to prevent errors if something went wrong while creating dynamic memory allocation using new operator.

****

It generally multiplies to arrays a and b then shows the result.

Explanation: first we have declared array a and b then declared a pointer named \*pointer.

**Int \*am** : it receives two arrays. Inside the function we have declared another pointer \*pointer with new keyword. And stored the multiplication value. Then returns it to main function.

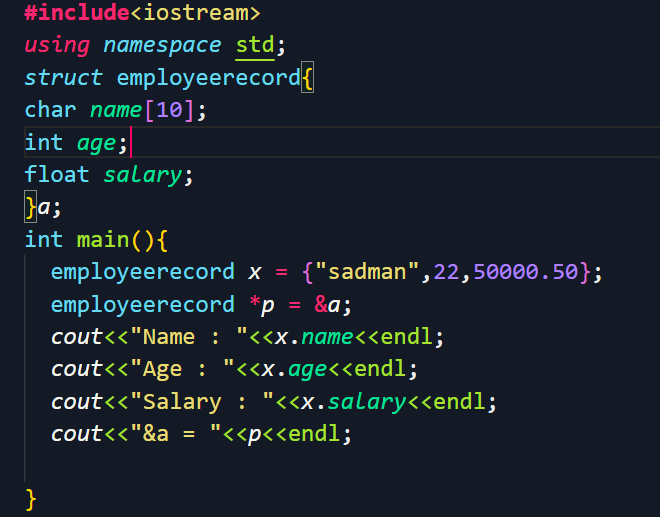
**Main function**: insidethe main function we have declared to arrays and one pointer. The pointer receives the value returned by the function “int \*am”.

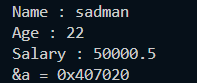
Print function: inside the print function we have received the pointer passed from the main function and the size. Then printed the array.

We used the \* sign in function name because the function returns pointer to an integer.

***STRUCTURE***

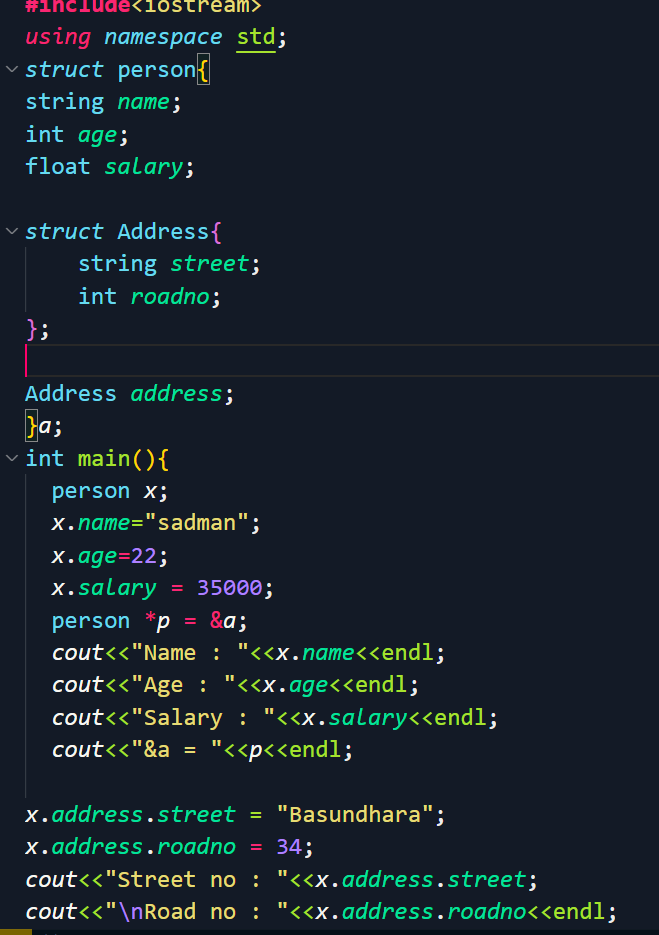
STRUCTURE is an aggregate data type built using element of other types. Its basically is a collection of data.





\***Nested Structure:**

* As any number and type of variables declared inside a structure, another structure can also be declared/defined inside another structure.



A screenshot of a computer

Description automatically generated

**SELF REFERENTIAL STRUCTURE:**

A **self-referential structure** is a structure that includes a pointer to an instance of the same type as itself.

A screen shot of a computer code

Description automatically generated

This defines a structure Person with two members:

* name – a character array (C-style string) that can store up to 29 characters (plus a null terminator).
* child – a pointer to another Person structure, which makes this structure **self-referential**.

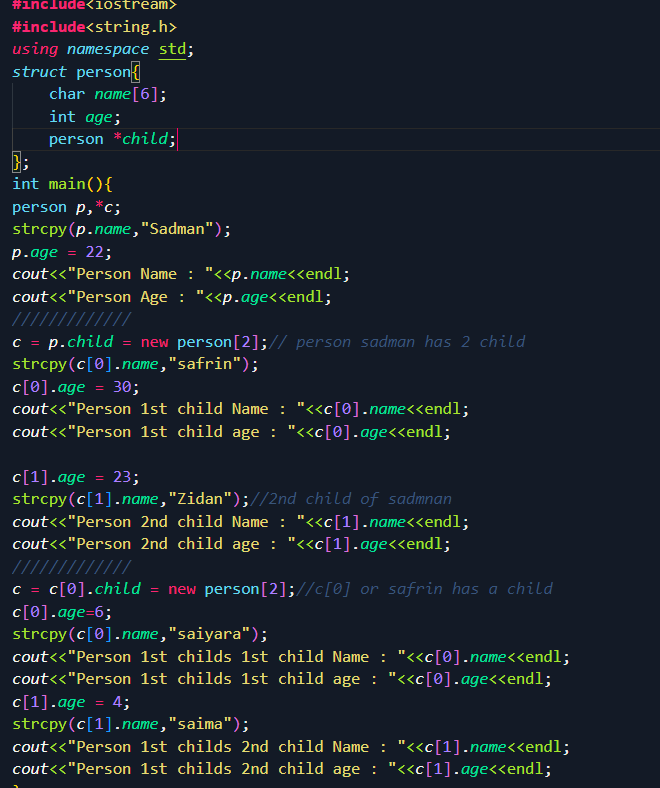
***int main() {***

***Person p, \*c; }***

* Here, p is an instance of Person.
* c is a pointer to a Person structure.

c = p.child = new Person[2];

* new Person[2] dynamically allocates an array of 2 Person objects and assigns this array's address to p.child, meaning p.child now points to the first element of this new Person array.
* c is also set to point to this array, so c and p.child now refer to the same memory location, which is the start of this array of two Person objects.

.A screenshot of a computer

Description automatically generated

Instead of creating many instances of person with nested child structure inside we use pointer that points to person structure and use functions as well.

**Linked List**

This is a list in which every items are linked with each other by their address.

We declare pointer inside struct/class to name the variable which will hold the address of next variable.

A computer code on a dark background

Description automatically generated

**Explanation**:

We have declared the class named Node

Variable data int type

“->” use this sign when you have a pointer to an object and want to access a member of the object.

A screen shot of a computer program

Description automatically generated

Explanation:

**Node\* temp = first;**

* temp is a pointer of type Node\*.
* It is initially set to point to the first node in the linked list.
* This allows us to start traversing the list from the first node.

**while (temp != nullptr)**

* The while loop runs as long as temp is not nullptr.
* nullptr indicates the end of the linked list (there are no more nodes to process).
* The loop ensures that we visit every node in the list.

**cout << temp->data << " -> ";**

* temp->data accesses the data stored in the current node that temp is pointing to.
* This prints the value in the node, followed by " -> " to indicate a link to the next node.

**. temp = temp->next;**

* This moves the pointer temp to the next node in the linked list.
* temp->next holds the address of the next node.
* After updating temp, the loop processes the next node.