GSSoC '24 Mentor | 7K+ Family @Linkedin | Mentor At @topmate.io |
Building @CryptoMinds | MERN Stack Web Developer | Web 3.0 |
Blockchain Developer | Career Counselor | Graphic Designer
Darshan Vasani
Helping Business
vasanidk30@gmail.com
https://linktr.ee/dpvasani56

Pointer

Darshan Vasani

Software Engineer | Software Developer

Connect Me

🌐 **Personal Websites:**

- **Personal Website**: https://dpvasani-58.netlify.app/
- **GitHub**: https://github.com/dpvasani
- **LinkedIn**: https://www.linkedin.com/in/dpvasani56/
- **Linktree**: https://linktr.ee/dpvasani56
- **Topmate**: https://topmate.io/dpvasani_56
- **Credly**: https://www.credly.com/users/dpvasani55/
- **CryptoMinds Developer Community**: https://chat.whatsapp.com/HNccLA2tBx52BoNajBngz7
- **Twitter**: https://x.com/vasanidarshan56

---

https://replit.com/@DarshanVasani/Pointer-Level-I#main.cpp

https://replit.com/@DarshanVasani/Pointer-Level-II#main.cpp https://replit.com/@DarshanVasani/Pointer-Level-III#main.cpp

https://github.com/dpvasani/Pointer-In-Cpp

Understanding Pointers in C++ Level I

Pointer Basics

Pointers are a powerful feature in C++ that allow you to directly manage memory. Here is a detailed breakdown based on the provided notes.

What is a Pointer?

- **Pointer Definition**: A pointer is a variable that stores the address of another variable.
- **Symbol Table**: This is a hidden data structure that keeps track of variable names and their corresponding addresses.
- **Memory Management**: Managed by the OS, and pointers allow direct access to memory locations.

Declaring and Using Pointers

- **Declaration**:

  ```
  int *ptr; // ptr is a pointer to an integer
  ```

- **Initialization**:

  ```
  int a = 5;
  int *ptr = &a; // ptr holds the address of variable a
  ```

Here, `ptr` is a pointer to the integer variable `a`. The `&` operator is used to get the address of `a`.

- **Accessing Value and Address**:

  ```
  cout << *ptr << endl; // Outputs the value stored at the
  address ptr is pointing to
  cout << ptr << endl;  // Outputs the address stored in p
  tr (address of a)
  cout << &ptr << endl; // Outputs the address of ptr itse
  lf
  ```

Summary of Pointer Operators

- `ptr`: Value stored at the location `ptr` is pointing to.
- `&ptr`: Address of the pointer variable `ptr`.
- `&a`: Address of variable `a`.
- `ptr`: Value of `ptr` (which is the address of `a`).

Pointer Example

Here's a practical example demonstrating the concepts discussed:

```
#include <iostream>
using namespace std;

int main() {
  int a = 10;
  int *p = &a;
  int *q = p;
  int *r = q;

  cout << a << endl;                   // 10
  cout << &a << endl;                  // Address of a
  cout << p << endl;                   // Address of a (sam
e as &a)
  cout << &p << endl;                  // Address of p
  cout << *p << endl;                  // 10
  cout << q << endl;                   // Address of a (sam
```

```
e as p)
  cout << &q << endl;                  // Address of q
  cout << *q << endl;                  // 10
  cout << r << endl;                   // Address of a (sam
e as q)
  cout << &r << endl;                  // Address of r
  cout << *r << endl;                  // 10
  cout << (*p + *q + *r) << endl;      // 30
  cout << (*p) * 2 + (*r) * 3 << endl; // 50
  cout << (*p) / 2 - (*q) / 2 << endl; // 0

  return 0;
}
```

Output Explanation

- `cout << a << endl;` prints the value of `a`.
- `cout << &a << endl;` prints the address of `a`.
- `cout << p << endl;` prints the address stored in `p` (which is the address of `a`).
- `cout << &p << endl;` prints the address of `p`.
- `cout << *p << endl;` prints the value stored at the address `p` is pointing to (value of `a`).
- Similarly, for `q` and `r`.

Pointer Size

- The size of a pointer is architecture-dependent. On a 64-bit system, the size of any pointer is typically 8 bytes.

  ```
  cout << sizeof(ptr) << endl; // Outputs 8 on a 64-bit sy
  stem
  ```

Why Use Pointers?

1. **Dynamic Memory Allocation**: Allocate memory during runtime.
2. **Memory Management**: Efficiently manage memory resources.

3. **Pointer Arithmetic**: Move from one memory location to another.
4. **Pass by Reference**: Efficiently pass arrays and other structures to functions.
5. **Function Pointers**: Pass functions as arguments to other functions.

Good and Bad Practices

- **Bad Practice**:

  ```
  int *ptr;
  cout << ptr << endl; // Uninitialized pointer, may cause
  segmentation fault
  ```

- **Good Practice**:

  ```
  int *p = 0;          // NULL pointer
  int *ptr2 = NULL;    // NULL pointer
  int *ptr3 = nullptr; // NULL pointer (C++11 and later)
  ```

Pointer Arithmetic

- Incrementing a pointer moves it to the next memory location of the type it points to.

  ```
  int a = 5;
  int *ptr = &a;
  ptr = ptr + 1; // Moves to the next integer memory locat
  ion
  ```

Understanding pointers is crucial for effective memory management and optimization in C++. The concepts discussed here provide a foundational understanding necessary for more advanced topics like dynamic memory allocation and complex data structures.

Step 1: Simple Variable

Consider the code:

```
int a = 5;
```

1. **Memory Block Representation**: In memory, there is a block allocated for the integer variable `a`.
2. **Address and Value**:
   - Let's say the address of this memory block is `104`.
   - The value stored in this block is `5`.
3. **Symbol Table**: The symbol table maps the variable `a` to the address `104`.

Visual Representation:

```
Memory:

   Address: 104
   +-------+
   |   5   |  <- Value of `a`
   +-------+

Symbol Table:

   a -> 104
```

Step 2: Pointer to the Variable

Now, let's introduce a pointer:

```
int *p = &a;
```

1. **Memory Block for Pointer**: A pointer `p` is created and it also has its own memory block.
2. **Pointer Address and Value**:
   - Let's assume the address of the pointer `p` is `208`.
   - The value stored in `p` is the address of `a`, which is `104`.

Visual Representation:

```
Memory:

   Address: 104
   +--------+
   |   5    |  <- Value of `a`
   +--------+

   Address: 208
   +--------+
   |  104   |  <- Value of `p` (address of `a`)
   +--------+

Symbol Table:

   a -> 104
   p -> 208
```

Step 3: Annotated Box Diagram

We can now draw a more detailed box diagram:

```
+----------------------+    +-----------------------+
| Memory Address: 104 |    | Memory Address: 208  |
|----------------------|    |-----------------------|
|         5            |    |        104            |
|----------------------|    |-----------------------|
|         a            |    |         p             |
+----------------------+    +-----------------------+

Symbol Table:
a -> 104
p -> 208
```

Here:

- The first box represents the memory block at address `104` containing the value `5` and labeled as `a`.

- The second box represents the memory block at address `208` containing the value `104` (the address of `a`) and labeled as `p`.

Explanation:

- The integer variable `a` is stored at address `104` with the value `5`.
- The pointer `p` is stored at address `208` and contains the address of `a` (`104`).

This visual representation helps in understanding how variables and pointers are stored in memory, with the symbol table linking variable names to their corresponding addresses.

Understanding Pointers in C++ - Level II

Pointer and Arrays

In this section, we'll explore the use of pointers with arrays and functions, diving deeper into how pointers work in these contexts.

Arrays and Pointers

- **Array Definition**:

  ```
  int arr[10];
  ```

  Here, `arr` is an array of 10 integers.

- **Base Address**:
  - The name of the array (`arr`) represents the base address of the array.
  - `arr[0]` is the first element, and `&arr[0]` is its address.
  - `arr` and `&arr` both give the base address, which is the same as `&arr[0]`.

Example with Array Initialization

```
#include <iostream>
using namespace std;

int main() {
```

```
  int arr[4] = {12, 44, 16, 18};
  cout << arr << endl;          // Base address of the array
  cout << arr[0] << endl;       // First element (12)
  cout << &arr << endl;         // Base address of the array
  cout << &arr[0] << endl;      // Address of the first eleme
nt

  int *p = arr;
  cout << p << endl;            // Base address of the array
(same as arr)
  cout << &p << endl;           // Address of pointer p
  cout << *arr << endl;         // First element (12)
  cout << arr[0] << endl;       // First element (12)
  cout << *arr + 1 << endl;     // 13 (12 + 1)
  cout << *(arr + 1) << endl;   // Second element (44)
  cout << arr[1] << endl;       // Second element (44)
  cout << *(arr + 2) << endl;   // Third element (16)
  cout << arr[2] << endl;       // Third element (16)
  cout << *(arr + 3) << endl;   // Fourth element (18)
  cout << arr[3] << endl;       // Fourth element (18)

  return 0;
}
```

How Array Indexing Works

- `arr[i]` is equivalent to `(arr + i)`.
- `i[arr]` is also equivalent to `(i + arr)`, which is a less common but valid way to access array elements in C++.

Array Size and Pointer Size

- **Array Size**:

  ```
  cout << sizeof(arr) << endl; // Outputs size of the arra
  y in bytes (e.g., 40 for 10 integers)
  ```

- **Pointer Size**:

```
int *p = arr;
cout << sizeof(p) << endl;   // Outputs size of the poin
ter (e.g., 8 bytes on a 64-bit system)
cout << sizeof(*p) << endl;  // Outputs size of the inte
ger (4 bytes)
```

Char Array

Char arrays are used to store strings.

```
char ch[10] = "Babbar";
char *c = ch;
cout << ch << endl;     // Babbar
cout << &ch << endl;    // Base address of the array
cout << ch[0] << endl;  // B
cout << &ch[0] << endl; // Base address of the array
cout << c << endl;      // Babbar
cout << *c << endl;     // B
cout << &c << endl;     // Address of the pointer c
```

Pointers in Functions

Passing an array to a function actually passes a pointer to its first element.

```
void solve(int arr[]) {
  cout << "Size Of Array In Function : " << sizeof(arr) <<
endl; // 8 (size of pointer)
  cout << "Arr :" << arr << endl;  // Base address of the a
rray
  cout << "&Arr :" << &arr << endl; // Address of the point
er variable
  arr[0] = 50; // Modifies the original array
}
```

In `main` function:

```
int arr[10] = {1, 2, 3, 4};
cout << "Size Of Arr inside Main Function : " << sizeof(ar
r) << endl; // 40
solve(arr);
cout << arr[0] << endl; // 50 (modified by solve function)
```

Example with Function Pointer

Here's a detailed example of how a pointer to a function can be used to update a variable.

```
#include <iostream>
using namespace std;

void update(int *p) {
  cout << "Address Stored In p is: " << p << endl;
  cout << "Address of p is :" << &p << endl;
  *p = *p + 10;
}

int main() {
  int a = 5;
  cout << "Address of a is : " << &a << endl;
  int *ptr = &a;
  cout << "Address Stored In Ptr Is : " << ptr << endl;
  cout << "Value Stored In Ptr Is : " << *ptr << endl;
  cout << "Address Of Ptr Is : " << &ptr << endl;

  update(ptr);

  cout << "Value Of A :" << a << endl;

  return 0;
}
```

Output Explanation

- `cout << "Address of a is : " << &a << endl;` prints the address of `a`.
- `cout << "Address Stored In Ptr Is : " << ptr << endl;` prints the address stored in `ptr` (address of `a`).

- `cout << "Value Stored In Ptr Is : " << *ptr << endl;` prints the value at the address `ptr` is pointing to (value of `a`).
- `cout << "Address Of Ptr Is : " << &ptr << endl;` prints the address of `ptr`.

In the `update` function:

- `p = *p + 10;` modifies the value at the address `p` is pointing to, which is the value of `a`.

After calling `update(ptr)`:

- `cout << "Value Of A :" << a << endl;` prints the updated value of `a`.

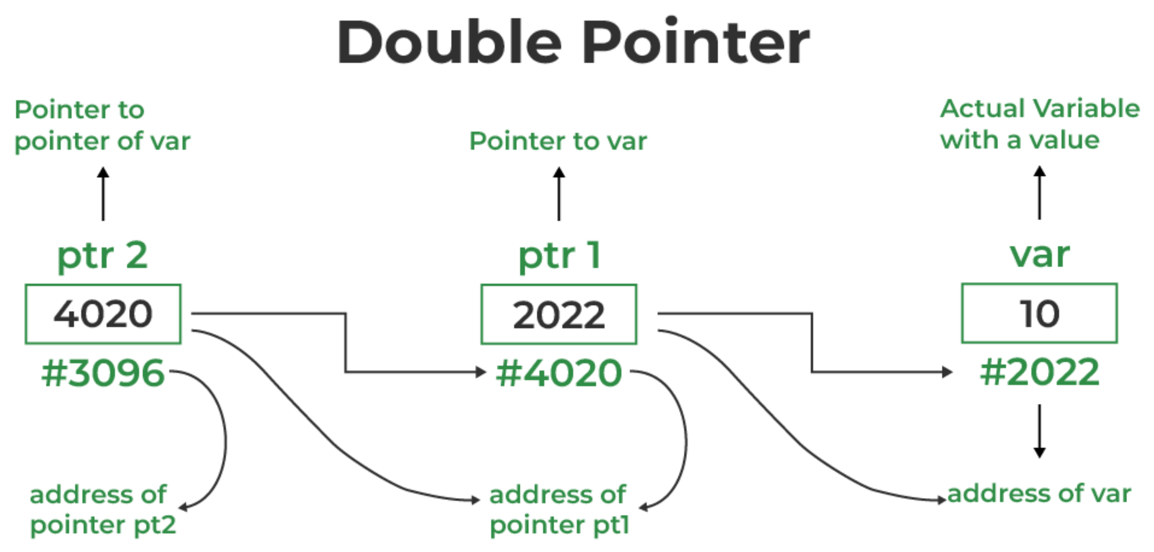
Understanding pointers at this level is crucial for effective memory management and optimization in C++. The concepts discussed here provide a foundational understanding necessary for more advanced topics like dynamic memory allocation and complex data structures.

Understanding Pointers in C++ - Level III

Advanced Pointer Concepts

In this section, we explore more advanced pointer concepts, including pointers to functions, double pointers, triple pointers, and passing pointers by reference.

Double and Triple Pointers



Double and triple pointers are pointers that point to other pointers.

```
#include <iostream>
using namespace std;

int main() {
  int a = 5;
  int *p = &a;
  int **q = &p;  // double pointer
  int ***r = &q; // triple pointer

  cout << a << endl;   // 5
  cout << &a << endl;  // address of a
  cout << p << endl;   // address of a
  cout << &p << endl;  // address of p
  cout << *p << endl;  // value of a (5)
  cout << q << endl;   // address of p
  cout << &q << endl;  // address of q
  cout << *q << endl;  // address of a
  cout << **q << endl; // value of a (5)
  cout << r << endl;   // address of q
  cout << &r << endl;  // address of r
  cout << *r << endl;  // address of p
  cout << **r << endl; // address of a
  cout << ***r << endl; // value of a (5)

  return 0;
}
```

Passing Pointers by Reference

Passing a pointer to a function by reference allows the function to modify the pointer itself, not just the value it points to.

```
#include <iostream>
using namespace std;
```

```
void solve(int *&p) {
  // Passed By Reference
  p = p + 1; // Changes the address stored in p
}

int main() {
  int a = 5;
  int *p = &a;
  cout << "Before: " << p << endl;
  solve(p);
  cout << "After: " << p << endl;

  return 0;
}
```

Output Explanation

- `Before: 0x7fffdf39602c` prints the address of `a`.
- After calling `solve(p)`, `p` is incremented by the size of an integer (4 bytes), resulting in a new address.
- `After: 0x7fffdf396030` prints the new address stored in `p`.

Return by Reference

Returning a pointer to a local variable can lead to undefined behavior because the local variable's scope ends when the function returns.

```
#include <iostream>
using namespace std;

int* solve() {
  int a = 5; // Local variable
  int *ans = &a;
  return ans; // Warning: returns address of local variable
}

int main() {
  int *p = solve();
  cout << "Returned pointer: " << p << endl;
  cout << "Dereferenced value: " << *p << endl; // Undefine
d behavior

  return 0;
}
```

Pointers to Functions

Pointers to functions are used to store the address of a function, allowing for function pointers to be passed as arguments, stored in arrays, etc.

```
#include <iostream>
using namespace std;

void func() {
  cout << "Hello, World!" << endl;
}

int main() {
  void (*funcPtr)() = func; // Declare a function pointer a
nd assign it the address of func
  funcPtr(); // Call the function through the function poin
ter

  return 0;
}
```

Example with Pointers in Functions

Here's an example demonstrating pointers passed by value and by reference:

```
#include <iostream>
using namespace std;

void util(int *p) {
  cout << "Before In Function" << endl;
  cout << p << endl;
  cout << *p << endl;
```

```cpp
  p = p + 1; // Increment address
  cout << "After In Function" << endl;
  cout << p << endl;
  cout << *p << endl; // Garbage value
}

void solve(int *&p) {
  // Passed By Reference
  p = p + 1;
}

int main() {
  int a = 5;
  int *p = &a;

  cout << "Before:" << endl;
  cout << a << endl;
  cout << p << endl;
  cout << *p << endl;
  cout << "Function Call" << endl;

  util(p);

  cout << "Again In Main Function" << endl;
  cout << "After" << endl;
  cout << a << endl;
  cout << p << endl;
  cout << *p << endl;

  cout << "Before solve by reference:" << p << endl;
  solve(p);
  cout << "After solve by reference:" << p << endl;

  return 0;
}
```

Output Explanation

- In `util`, `p` is incremented inside the function, but this change does not reflect outside the function because `p` is passed by value.
- In `solve`, `p` is passed by reference, so the change in address is reflected outside the function.

Understanding these concepts of pointers and references in C++ helps in efficient memory management and deepens comprehension of how data is accessed and manipulated in various contexts.

Got it! Let's visually represent the memory and symbol table for the following code:

```
int a = 5;
int *p = &a;
int **q = &p;  // double pointer
int ***r = &q; // triple pointer
int ****s = &r; // quadruple pointer
```

We'll use the box analogy for each pointer level.

Step 1: Simple Variable

```
int a = 5;
```

1. **Memory Block Representation**: A memory block is allocated for `a`.
2. **Address and Value**:
   - Let's say the address of `a` is `104`.
   - The value stored in `a` is `5`.
3. **Symbol Table**: The symbol table maps `a` to the address `104`.

```
Memory:

   Address: 104
   +-------+
   |   5   |  <- Value of `a`
   +-------+
Symbol Table:

    a -> 104
```

Step 2: Pointer to `a`

```
int *p = &a;
```

1. **Memory Block for Pointer**: A memory block is allocated for `p`.
2. **Pointer Address and Value**:
   - Let's assume the address of `p` is `208`.
   - The value stored in `p` is the address of `a` (`104`).

```
Memory:

    Address: 104
    +--------+
    |   5    |  <- Value of `a`
    +--------+

    Address: 208
    +--------+
    |  104   |  <- Value of `p` (address of `a`)
    +--------+

Symbol Table:

    a -> 104
    p -> 208
```

Step 3: Pointer to Pointer (`q`)

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

1. **Memory Block for Double Pointer**: A memory block is allocated for `q`.

2. **Double Pointer Address and Value**:
    - Let's assume the address of `q` is `312`.
    - The value stored in `q` is the address of `p` (`208`).

```
Memory:

    Address: 104
    +-------+
    |   5   |  <- Value of `a`
    +-------+

    Address: 208
    +-------+
    |  104  |  <- Value of `p` (address of `a`)
    +-------+

    Address: 312
    +-------+
    |  208  |  <- Value of `q` (address of `p`)
    +-------+

Symbol Table:

    a -> 104
    p -> 208
    q -> 312
```

Step 4: Triple Pointer (`r`)

```
int ***r = &q;
```

1. **Memory Block for Triple Pointer**: A memory block is allocated for `r`.
2. **Triple Pointer Address and Value**:
    - Let's assume the address of `r` is `416`.
    - The value stored in `r` is the address of `q` (`312`).

```
Memory:

   Address: 104
   +--------+
   |   5    |  <- Value of `a`
   +--------+

   Address: 208
   +--------+
   |  104   |  <- Value of `p` (address of `a`)
   +--------+

   Address: 312
   +--------+
   |  208   |  <- Value of `q` (address of `p`)
   +--------+

   Address: 416
   +--------+
   |  312   |  <- Value of `r` (address of `q`)
   +--------+

Symbol Table:

   a -> 104
   p -> 208
   q -> 312
   r -> 416
```

Step 5: Quadruple Pointer (`s`)

```
int ****s = &r;
```

1. **Memory Block for Quadruple Pointer**: A memory block is allocated for `s`.
2. **Quadruple Pointer Address and Value**:
   - Let's assume the address of `s` is `520`.

- The value stored in `s` is the address of `r` (`416`).

```
Memory:

   Address: 104
   +--------+
   |   5    |  <- Value of `a`
   +--------+

   Address: 208
   +--------+
   |  104   |  <- Value of `p` (address of `a`)
   +--------+

   Address: 312
   +--------+
   |  208   |  <- Value of `q` (address of `p`)
   +--------+

   Address: 416
   +--------+
   |  312   |  <- Value of `r` (address of `q`)
   +--------+

   Address: 520
   +--------+
   |  416   |  <- Value of `s` (address of `r`)
   +--------+

Symbol Table:

   a -> 104
   p -> 208
   q -> 312
   r -> 416
   s -> 520
```

Annotated Box Diagram

```
+-------------+   +-------------+   +-------------+   +----
---------+   +-------------+
| Addr: 104   |   | Addr: 208   |   | Addr: 312   |   | Add
r: 416   |   | Addr: 520   |
|-------------|   |-------------|   |-------------|   |----
---------|   |-------------|
|      5      |   |     104     |   |     208     |   |
312     |   |     416     |
|-------------|   |-------------|   |-------------|   |----
---------|   |-------------|
|      a      |   |      p      |   |      q      |   |
r       |   |      s      |
+-------------+   +-------------+   +-------------+   +----
---------+   +-------------+

Symbol Table:
a -> 104
p -> 208
q -> 312
r -> 416
s -> 520
```

Pointers in C++ Level I

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

int main() {
// Pointer Level I
// Storage Location -> Address
// Hiden Data Structure -> Symbol Table
// int a = 5;
// Symbol Table -> a -> Address
// a -> 104 // At Address 104 There is data is 5
// Symbol Table Stores Mapping
// Memory Management Is Done By OS
// We Can Access Memory Using Pointers
// Address Of Operator -> &

// Pointer
// int a = 5;
// inside of a you can stoe integer type data
// int *ptr;
// ptr is a pointer to integer data
// Poiinter is a data type which holds the address of other data type
// Pointer is a data type which store only address
// ptr is variable name
// Explain Through Example
// int a = 5;
// int *ptr = &a; -> ptr is a pointer to a  which contain integer data
// int is datatype
// ptr is pointer to integer data
// * is syntex for pointer creation or dereference Oprator
// p variable name
// & address of operator
// a is variable name
// (int *) -> Collectively is a pointer to integer data
// Data_Type *Variable_Name;
// variable_Name is a pointer to Data_Type
// int a = 5;
// // Poniter Creation
// int *ptr = &a;
// // Access The value ptr is pointing to
// // Dereference Operator
// cout << *ptr << endl;
// Above Mentioned Is For Understanding puposes
// Pointer Is Not Data Type
// Pointer Is Variable Name
// Pointer In Cpp Is Variable That Store Address Of Another Variable
// Pointer Through Two Thing You Can Acess
// 1. Value    cout<<*ptr<<endl;
// 2. Address  cout<<ptr<<endl;
// cout<<ptr<<endl; -> ptr Vale Dabbe Me Jo Pada He Uski Bat Ho Rahi He
// cout<<&ptr<<endl; -> ptr Vale Dabbe ka Address He Yeh
```

```
 // Summery
// *ptr -> Value Stored At Location In Ptr
// &ptr -> Address Of Ptr
// &a -> Address Of a
// ptr -> Value of ptr -> Which Is Addrress Of a
```

```
 // Example
// int a = 5;
// int *ptr = &a;
// a[5] -> Address is 104
// ptr[104] -> Address is 302
// cout<<a; -> 5
// cout<<*a; -> Error
// cout<<&a; -> 104
// cout<<ptr; -> 104
// cout<<&ptr; -> 302
// cout<<*ptr; -> 5
```

```
 // Size Of Pointer Will Be Always 8 -> Architecture Dependent
// System Always Take 8 bite Memory For Pointer
// 64 Bit Architecture -> 8 Byte
// int a = 8;
// int *ptr = &a;
// cout << sizeof(ptr) << endl;
// Why Need Of Pointer
// 1. Dynamic Memory Allocation
// 2. Memory Management
// 3. Pointer Arithmetic -> Go From One Location To Another
// 4. Passed By Reference In Array
// 5. To Create Pointer To Function -> Passing a Function Inside Function As
// An Argument
```

```
// Bad Practice
// int *ptr;            // It Has Some Random Grabag Value
// cout << ptr << endl; // -> Grabage Value -> Segmenation Fault
```

```
// // Good Practice
// // NULL Pointer
// int *p = 0;
// int *ptr2 = NULL;
// int *ptr3 = nullptr;
// All Three Are Same
// cout << p << endl; // -> Segmentation Fault
// Segmenation Fault -> When You Access Memory Location Which Is Not Available
// Or Memory Of Other Which Is Not Allocated To Your Program
```

```
// Pointer Arithmetic
// int a = 5;
// int *ptr = &a;
// a[5] -> Address is 104
// ptr[104] -> Address is 208
// a= a+1;
// ptr = ptr+1; -> 108
// a1 to a1 + 3 -> Taken By Integer So Next Address Will Be a1+ 4
// *p= *p+1; -> Value Stored In P(not Address ) Will Be Incremented
// So if a = 5
// *p = *p+1; -> 6
// So Now Value Of a = 6
```

```
// Revision
// a -> a vala dabba
// ptr -> ptr vala dabba
// &a -> a vale dabbe ka address
// &ptr -> ptr vale dabbe ka address
// *ptr -> ptr vale dabbe ka value -> ptr vale dabbe me jo location he us
// location pe jao vaha daba milga us dabbe me jo valu padi he
```

```
// Copy pointer
// int a = 5;
// int *ptr = &a;
// int *dusraPtr = ptr;
```

```
int a = 10;
int *p = &a;
int *q = p;
int *r = q;
```

```
cout << a << endl;                     // 10
cout << &a << endl;                    // Address Of a
cout << p << endl;                     // Addre Of a
cout << &p << endl;                    // Addre Of p
cout << *p << endl;                    // 10
cout << q << endl;                     // Addre Of a
cout << &q << endl;                  // Addre Of q
cout << *q << endl;                  // 10
cout << r << endl;                   // Addre Of a
cout << &r << endl;                  // Addre Of r
cout << *r << endl;                  // 10
cout << (*p + *q + *r) << endl;      // 30
cout << (*p) * 2 + (*r) * 3 << endl; // 50
cout << (*p) / 2 - (*q) / 2 << endl; // 0
```

```
// Output
// 10
// 0x7ffc2482ae5c
// 0x7ffc2482ae5c
// 0x7ffc2482ae50
// 10
// 0x7ffc2482ae5c
// 0x7ffc2482ae48
// 10
// 0x7ffc2482ae5c
// 0x7ffc2482ae40
// 10
// 30
// 50
// 0
}
```

Pointers in C++ Level II

```
#include <bits/stdc++.h>
#include <iostream>
using namespace std;
// void solve(int arr[]) {
//   cout << "Size Of Array In Function : " << sizeof(arr) << endl;
//   cout << "Arr :" << arr << endl;
//   cout << "&Arr :" << &arr << endl;
//   arr[0] = 50;
// }
```

```
void update(int *p) {
cout << "Address Stored In p is: " << p << endl;
cout << "Address of p is :" << &p << endl;
*p = *p + 10;
}
```

```
int main() {
// Pointer Level II
// int arr[10];
// arr[0] arr[1] arr[2] arr[3] arr[4] arr[5] arr[6] arr[7] arr[8] arr[9]
// arr[0] = 5;
// cout << &arr[0] << endl;
// arr[0] -> 5
// &arr[0] -> 0x7ffcc7e6e3c0 or 104 (For Understanding 104 Has Taken)
// arr -> Base Address -> 0x7ffcc7e6e3c0 or 104
// cout<<&arr<<endl; -> Base Address -> Because Of Symbol Table
// cout << arr << endl;
// int arr[4] = {12, 44, 16, 18};
// cout << arr << endl;
// cout << arr[0] << endl;
// cout << &arr << endl;
// cout << &arr[0] << endl;
```

```
 // int *p = arr;
// cout << p << endl;
// cout << &p << endl;
// cout << *arr << endl;        // 12
// cout << arr[0] << endl;      // 12
// cout << *arr + 1 << endl;    // 13
// cout << *(arr + 1) << endl; // 44
// cout << arr[1] << endl;      // 44
// cout << *(arr + 2) << endl; // 16
// cout << arr[2] << endl;      // 16
// cout << *(arr + 3) << endl; // 18
// cout << arr[3] << endl;      // 18
```

```
 // How Arr[i] Resolves
// arr[i] = *(arr + i)
// i[arr] = *(i + arr)
```

```
 // int i = 0;
// cout << arr[i] << endl;
// cout << i[arr] << endl;
// arr = arr + 2; -> Error
// int *p = arr;
// p = p + 2; -> Works
// Through Pointer I Can Show Any Subpart Of Array
// int arr[10];
// cout<<sizeof(arr)<<endl; //40
// int *p = arr;
// cout<<sizeof(p)<<endl; // 8
// cout<<sizeof(*p)<<endl; // 4
```

```
 // Char Array
// char ch[10] = "Babbar";
// char *c = ch;
// // cout << c << endl; // Babbar
// // Lets Work
// cout << ch << endl;     // Babbar
// cout << &ch << endl;    // 0x7ffcc7e6e3c0
// cout << ch[0] << endl;  // B
// cout << &ch[0] << endl; // Babbar
```

```
 // cout << &c << endl; // 0x7ffcc7e6e3c0
// cout << *c << endl; // B
// cout << c << endl;  // Babbar
```

```
 // *c = *(c + 0) -> c[0] -> B
// char name[9] = "SherBano";
// char *cptr = &name[0];
```

```
 // cout << name << endl;         // SherBano
// cout << &name << endl;        // 0x7ffcc7e6e3c0
// cout << *(name + 3) << endl; // r
// cout << cptr << endl;         // SherBano
// cout << *cptr << endl;        // S
// cout << &cptr << endl;        // 0x7ffcc7e6e3c9
// cout << *(cptr + 3) << endl; // r
// cout << cptr + 2 << endl;    // erBano
// cout << *cptr + 2 << endl;   // erBano
// cout << cptr + 9 << endl;    // Garbage Value
// cout << *cptr + 9 << endl;   // Garbage Value
```

```
 // char ch = 'k';
// char *cptr2 = &ch;
// cout << cptr2 << endl; // kGarbage Value
```

```
 // char name[10] = "Babbar";
// cout << name << endl;
// char *ch = "Babbar";
// cout << ch << endl; // Babbar
```

```
 // Pointer In Function
// int arr[10] = {1, 2, 3, 4};
// cout << "Size Of Arr inside Main Function : " << sizeof(arr) << endl;
// cout << "Arr :" << arr << endl;
// cout << "&Arr :" << &arr << endl;
// // Printing Array inside Main
// for (int i = 0; i < 10; i++) {
//   cout << arr[i] << " ";
// }
// cout << endl;
// cout << endl << endl << "Now Calling To Solve Function" << endl;
// solve(arr);
// cout << "Wapis Main Function Me" << endl;
// // Printing Array inside Main
// for (int i = 0; i < 10; i++) {
//   cout << arr[i] << " ";
// }
// Output
// Size Of Arr inside Main Function : 40
// Arr :0x7ffc0b76f290
// &Arr :0x7ffc0b76f290
// 1 2 3 4 0 0 0 0 0 0
```

```
 // Now Calling To Solve Function
// Size Of Array In Function : 8
// Arr :0x7ffc0b76f290
// &Arr :0x7ffc0b76f228 -> New Pointer
// Wapis Main Function Me
// 50 2 3 4 0 0 0 0 0 0
int a = 5;
cout << "Address of a is : " << &a << endl;
int *ptr = &a;
cout << "Address Stored In Ptr Is : " << ptr << endl;
cout << "Value Stored In Ptr Is : " << *ptr << endl;
cout << "Address Of Ptr Is : " << &ptr << endl;
update(ptr);
cout << "Value Of A :" << a << endl;
```

```
 // Output
// Address of a is : 0x7ffd1a0ec2bc
// Address Stored In Ptr Is : 0x7ffd1a0ec2bc
// Value Stored In Ptr Is : 5
// Address Of Ptr Is : 0x7ffd1a0ec2b0
// Inside Update Function
// Address Stored In p is: 0x7ffd1a0ec2bc
// Address of p is :0x7ffd1a0ec268
// Inside Main Function
// Value Of A :15

// Point to Note :
// Address Of ptr != Addres Of p
}
```

Pointers in C++ Level III

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

// void util(int *p) {
//    cout << "Before In Function" << endl;
//    cout << p << endl;
//    cout <<
p << endl;
//    p = p + 1; // Address + 1 -> Next Address -> Here 0x7fffab387cf4 + (1
4) =
//                // 0x7fffab387cf8
//    // (**ptr = **ptr + 1) != (p = p + 1) != (*p = *p + 1)
//    cout << "After In Function" << endl;
//    cout << p << endl;
//    cout << *p << endl; // Garbage Value
// }

// void solve(int *val) { *val = *val + 1; }
void solve(int *&p) {
// Passed By References
p = p + 1;
}

int main() {
// Pointer III
// Pointer To Function
// Function Pointer

// int a = 5;
// int *p = &a;
// int **q = &p;  // double pointer
// int ***r = &q; // triple pointer

// cout << a << endl;   // 5
// cout << &a << endl;  // 0x7ffc7fdc21ec
// cout << p << endl;   // 0x7ffc7fdc21ec
// cout << &p << endl;  // 0x7ffc7fdc21e0
// cout << *p << endl;  // 5
// cout << q << endl;   // 0x7ffc7fdc21e0
// cout << &q << endl;  // 0x7ffc7fdc21d8
// cout << *q << endl;  // 0x7ffc7fdc21ec
// cout << **q << endl; // 5
// cout << r << endl;   // 0x7ffc7fdc21d8
// cout << &r << endl;  // 0x7ffc7fdc21d0
// cout << *r << endl;  // 0x7ffc7fdc21e0
```

```
// Output
// 5
// 0x7ffc7fdc21ec
// 0x7ffc7fdc21ec
// 0x7ffc7fdc21e0
// 5
// 0x7ffc7fdc21e0
// 0x7ffc7fdc21d8
// 0x7ffc7fdc21ec
// 5
// 0x7ffc7fdc21d8
// 0x7ffc7fdc21d0
// 0x7ffc7fdc21e0
```

```
// int a = 5;
// int *p = &a;
```

```
// cout << "Before" << endl;
// cout << a << endl;
// cout << p << endl;
// cout << *p << endl;
// cout << "Function Call" << endl;
```

```
// util(p);
```

```
// cout << "Again In Main Function" << endl;
// cout << "After" << endl;
// cout << a << endl;
// cout << p << endl;
// cout << *p << endl;
```

```
// Output :
// Before
// 5
// 0x7fffab387cf4
// 5
// Function Call
// Before In Function
// 0x7fffab387cf4
// 5
// After In Function
// 0x7fffab387cf8
// -2090870528 -> Garbage Value
// Again In Main Function
// After
// 5
// 0x7fffab387cf4
// 5
```

```
// ****ptr ->  Value Present At Location Stored in ***ptr
// **ptr -> Value Present At Location Stored in **ptr
// *ptr -> Value Present At Location Stored in *ptr
// Star Star  -> Is Pointer To Pointer
```

```
// For Replacing Concept Of Pointer -> New Concept -> Reference Variable
// Refereance Variable
// int a = 5;
// int &b = a;
// cout << a << endl; // 5
// cout << b << endl; // 5
// Same Memory Location Diffrent Name
// Same Address
// Only Entry In Symbol Table
// Genrally Used In Function Call And To Impliment Pass By Reference Concept
// Concept Come For Increase Readability and Reducing Complexity
// Why To Use Refereance VariableRefer Rather Than Pointer
// 1. First You can Set Any Refereance Variable As NULL -> Safety More -> You
// Can Set Pointer To NULL
// 2. Ease Of Use While Pointer Difficult To Understand And Use
```

```
// solve(int a) {
//   // Passed By Value
// }
// solve(int &a) {
//   // Passed By Refereance
// }
```

```
// int a = 12;
// solve(&a);          // Pointer Has Been Created
// cout << a << endl; // 13
// return 0;
```

```
// Pointer Passed By Value -> By Default
// Pointer Passed By Refereance
int a = 5;
int *p = &a;
cout << "Before :" << p << endl;
solve(p);
cout << "After :" << p << endl;
```

```
// Output :
// Before :0x7fffdf39602c
// After :0x7fffdf396030
```

```
// Return By Reference
// Famous Example
int *solve() {
// Local Varible Block Scope -> Dead After Function End
int a = 5;
int *ans = &a;
return ans;
}
}
```