📌 C++ Pointers & Arrays – Deep Dive Notes

📃 Pointer Level II – Arrays & Pointers

📍 Array Initialization

```
int arr[10];
// arr[0] to arr[9]
arr[0] = 5;
```

- `&arr[0]` 👉 gives memory address (e.g., `0x104`)
- `arr` 👉 base address of the array (points to `arr[0]`)
- `&arr` 👉 also gives base address due to symbol table

```
cout << arr << endl;      // Base address
cout << &arr << endl;     // Base address
cout << &arr[0] << endl;  // Same as above
```

🔍 Accessing Array Elements

```
int arr[4] = {12, 44, 16, 18};
int *p = arr;
```

☑️ Access Patterns

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

👉 `arr[i] = *(arr + i)` 👉 `i[arr] = *(i + arr)`

```
int i = 0;
cout << arr[i] << endl;   // 12
cout << i[arr] << endl;   // 12
```

> 🚫 `arr = arr + 2;` ❌ (Error, because array is constant pointer) ☑️ But `int *p = arr; p = p + 2;` works perfectly.

📦 Sizeof Behavior

```
int arr[10];
cout << sizeof(arr) << endl; // 40 (10 ints, each 4 bytes)

int *p = arr;
cout << sizeof(p) << endl;   // 8 (pointer size on 64-bit)
cout << sizeof(*p) << endl;  // 4 (int size)
```

🔤 Character Arrays & Pointers

```
char ch[10] = "Babbar";
char *c = ch;
```

```
cout << ch << endl;     // Babbar
cout << &ch << endl;    // Prints address
cout << &ch[0] << endl; // Babbar
cout << *c << endl;     // B
cout << c << endl;      // Babbar
cout << &c << endl;     // Address of pointer variable
```

🧪 String Pointer Behavior

```
char name[9] = "SherBano";
char *cptr = &name[0];

cout << name << endl;          // SherBano
cout << *(name + 3) << endl;   // r
cout << cptr + 2 << endl;      // erBano
cout << *cptr + 2 << endl;     // S + 2 = 'U' (ASCII math)
cout << cptr + 9 << endl;      // Garbage beyond '\0'
```

> ⚠️ `*cptr + 9` applies ASCII math and can result in garbage output.

🧪 Char Pointer Quirk

```
char ch = 'k';
char *cptr2 = &ch;
cout << cptr2 << endl;  // kGarbageValue (because it keeps reading till it hits
'\0')
```

> ☑ Safe for null-terminated strings ✖ Not for single char

🔄 String Literals

```
char name[10] = "Babbar"; // Array
char *ch = "Babbar";      // String literal (const memory)
```

- `cout << name;` → Babbar
- `cout << ch;` → Babbar

🎯 Pointer in Function

```
void solve(int arr[]) {
  cout << "Size Of Array In Function : " << sizeof(arr) << endl;  // 8 (pointer)
  cout << "Arr :" << arr << endl;        // Address
  cout << "&Arr :" << &arr << endl;      // Address of pointer (new memory)
  arr[0] = 50;                            // Mutates original array
}
```

🔎 Calling from main:

```
int arr[10] = {1, 2, 3, 4};
cout << "Size Of Arr inside Main : " << sizeof(arr) << endl; // 40
solve(arr);
```

Output:

```
Size Of Arr inside Main Function : 40
Arr :0x7ffc...
&Arr :0x7ffc...
1 2 3 4 ...
Now Calling To Solve Function
Size Of Array In Function : 8
Arr :0x7ffc...
&Arr :0x7ffc... (Different from &arr in main)
```

```
Wapis Main Function Me
50 2 3 4 ... (arr[0] changed!)
```

☑️ Shows that arrays are passed by reference (via pointer), and mutations persist ✖️ But `sizeof()` gives misleading result inside function (`int*` not `int[]`)

🔧 Function Pointer Update Example

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

🔍 Calling `update(ptr)` from `main()`:

```
int a = 5;
int *ptr = &a;
update(ptr);
cout << "Value Of A :" << a << endl; // 15
```

📃 Output Trace:

```
Address of a is : 0x7ffd1...
Address Stored In Ptr Is : 0x7ffd1...
Value Stored In Ptr Is : 5
Address Of Ptr Is : 0x7ffd1...

Inside Update Function:
Address Stored In p is: 0x7ffd1... (same as ptr)
Address of p is : 0x7ffd1... (different from &ptr)
Value Of A :15
```

> 🎯 `*p = *p + 10;` modifies `a` directly 🔄 Even though `ptr` and `p` are different variables, they point to same address

---

📝 Key Takeaways

🔸 **Arrays decay into pointers** when passed to functions 🔸 `sizeof(array)` ≠ `sizeof(pointer)`, be careful 🔸 `arr[i] = *(arr + i)` is how indexing works internally 🔸 `char *` behaves differently – can print till `\0` 🔸 Use care when printing single characters via pointer 🔸 You **can modify original array values** via pointer inside functions 🔸 Address of pointers inside function is different (`&p` vs `&ptr`)

🔚 Summary Table

Concept	Behavior
`arr` vs `&arr`	Both give base address (symbol table)
`sizeof(arr)`	Actual size of array (e.g. 40 bytes for 10 ints)
`sizeof(p)`	Size of pointer (typically 8 bytes on 64-bit systems)
Character Array Printing	Prints till `\0`
`*cptr + n`	Performs ASCII arithmetic, not pointer movement
Array as Function Arg	Passed as pointer, allows mutation
Address of pointer param	Differs from address of pointer in caller

---

🧾 Understanding `int* ptr = &arr;` in C++

🔑 Core Idea:

In C++, the statement `int* ptr = &arr;` **may or may not be valid**, depending on **what `arr` actually is**.

Let's explore different cases step by step.

🧪 Case 1: `arr` is a Single Integer

```
int arr = 10;
int* ptr = &arr;
```

☑️ Valid

🧠 Explanation:

- Here, `arr` is just a single integer variable.
- `&arr` gives the **address of an `int`**.
- `ptr` is declared as a pointer to `int` → `int* ptr`.

📦 Analogy:

Think of `arr` as a **locker** holding one item, and `ptr` as a **key** to that locker.

🧪 Case 2: `arr` is an Array

```
int arr[5] = {1, 2, 3, 4, 5};
int* ptr = &arr;
```

❌ Invalid ❗

🧠 Why?

- `arr` is an **array of 5 integers** → type: `int[5]`.
- `&arr` gives the **address of the whole array**, so its type is `int (*)[5]` — a **pointer to array of 5 ints**.
- But `ptr` is `int*`, which expects a pointer to just **one int**.

😬 **Type mismatch!** `int*` ≠ `int (*)[5]`

✅ How to Fix It

```
int* ptr = arr;       // 👍 Valid
int* ptr = &arr[0];   // 👍 Valid
```

🧠 Why?

- In most expressions, `arr` **decays** to a pointer to its first element (`&arr[0]`).
- So `ptr = arr` is equivalent to `ptr = &arr[0]` → both are of type `int*`.

📦 Analogy: Array as a Building

Imagine:

- `arr[5]` is a **5-room building** 🏢.
- `&arr` gives the **address of the whole building** (like GPS to the building).
- `arr` or `&arr[0]` gives the **address of Room 0**.

🔑 If you want a **room key (int)**, use `arr` or `&arr[0]`.

But `&arr` is a **building key (int (*)[5])** — doesn't fit in a room lock.

🔍 Summary Table

Code Example	Type of `arr`	What `&arr` returns	`ptr` Type	✅ Valid?	Reason
`int arr = 5; int* ptr = &arr;`	`int`	`int*`	`int*`	✅ Yes	Types match: pointer to int
`int arr[5]; int* ptr = arr;`	`int[5]`	N/A (decays to `int*`)	`int*`	✅ Yes	Points to first element
`int arr[5]; int* ptr = &arr[0];`	`int[5]`	`int*`	`int*`	✅ Yes	Address of first element

Code Example	Type of `arr`	What `&arr` returns	`ptr` Type	☑️ Valid?	Reason
`int arr[5]; int* ptr = &arr;`	`int[5]`	`int (*)[5]`	`int*`	❌ No	Mismatch: pointer to array ≠ pointer to int
`int arr[5]; auto ptr = &arr;`	`int[5]`	`int (*)[5]`	`int (*)[5]`	☑️ Yes	Using `auto` deduces correct pointer type

---

🧠 Bonus: What if You Want the Address of the Whole Array?

If your goal **is** to get the address of the entire array, then:

```
int arr[5] = {1, 2, 3, 4, 5};
int (*ptr)[5] = &arr;  // ☑️ Valid
```

☑️ Why?

- Now `ptr` is a **pointer to an array of 5 ints** → type matches.

🧠 Golden Rule: Pointer Types Must Match

🧬 Think of pointer types as DNA. Even if the memory address is similar, the **type must match exactly** for safe and valid access.

📚 Final Takeaway

> Use `int* ptr = arr;` or `int* ptr = &arr[0];` when you want a pointer to the **first element of an array**. Avoid `int* ptr = &arr;` — it's invalid because you're mixing types: `int*` vs `int (*)[N]`.

📚 C++ Arrays: Passed by Reference or Pointer?

🧠 Concept Summary

In C++, **arrays seem like they're passed by reference**, but technically, **they decay into pointers** when passed to functions. This means that functions receive the memory address of the array's first element — allowing them to modify the original array.

📦 Analogy: Parcel vs. Address 📫

Imagine your array is a **parcel box** 📦.

- If you **send the whole box**, that's **pass-by-value** (a copy).

- If you just **give the delivery address**, that's **pass-by-pointer** (which is what happens with arrays in C++).
- If you send a **direct link to the box itself**, that's **pass-by-reference**.

🔑 **In C++, arrays usually pass the "address" (pointer), not the actual box (copy).**

🔍 Let's Break It Down

✅ 1. Default Behavior: **Decay to Pointer**

```
void modifyArray(int arr[]) {
    arr[0] = 100;
}
```

This function is **actually** equivalent to:

```
void modifyArray(int* arr) {
    arr[0] = 100;
}
```

🧠 **Explanation:** `arr[]` is syntactic sugar. Under the hood, it becomes a pointer `int* arr`, pointing to the first element of the array.

✅ Example:

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

void modifyArray(int arr[]) {
    arr[0] = 100;
}

int main() {
    int numbers[3] = {1, 2, 3};
    modifyArray(numbers);
    cout << numbers[0] << endl; // Output: 100 ✅
}
```

✅ **Original array is modified** because we passed a pointer to it.

❗ But What If We Want to Keep Size Info?

✅ 2. Pass by Reference (True Reference)

You can pass an array **by reference** (along with its size):

```
void modifyArray(int (&arr)[3]) {
    arr[0] = 200;
}
```

- `int (&arr)[3]` means: "a reference to an array of 3 integers".

✅ Example:

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

void modifyArray(int (&arr)[3]) {
    arr[0] = 200;
}

int main() {
    int numbers[3] = {1, 2, 3};
    modifyArray(numbers);
    cout << numbers[0] << endl; // Output: 200 ✅
}
```

📌 **Now the array doesn't decay to a pointer.** We pass the actual reference, keeping size info!

💥 What's the Difference?

Method	Type Inside Function	Can Modify?	Size Info Retained?
`void func(int arr[])`	`int*` (pointer)	✅ Yes	❌ No
`void func(int* arr)`	Pointer	✅ Yes	❌ No
`void func(int (&arr)[N])`	Reference to array	✅ Yes	✅ Yes (at compile time)

🎯 Why It Matters

- 🧮 **Losing size info** can cause bugs in loops.
- 📦 Use references (`int (&arr)[N]`) for safety and clarity.
- 🔁 For dynamic arrays, consider `std::vector<int>`.

✅ Bonus: Pass Array by Value (Copy)

If you use `std::array`, it gets passed **by value** unless explicitly passed by reference:

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

void modifyArray(array<int, 3> arr) {
    arr[0] = 500;
}

int main() {
    array<int, 3> nums = {1, 2, 3};
    modifyArray(nums);
    cout << nums[0] << endl; // Output: 1 ❌ (original unchanged)
}
```

🧠 TL;DR

- ◇ Arrays in C++ **decay into pointers** when passed to functions.
- ◇ That's why changes inside the function affect the original array.
- ◇ To keep size and pass by reference, use: `void func(int (&arr)[N])`.
- ◇ Use `std::array` or `std::vector` for modern, safer handling.

🧪 Try It Yourself

```cpp
// Task: Try changing the array inside a function using both pointer and
reference.
// Then try passing by value with std::array.
```

🚨 What Happens When You Access an Array Out of Bounds Using a Pointer in C++?

✅ First, Quick Recap

When you pass an array in C++, it decays into a pointer. You can then use pointer arithmetic:

```cpp
int arr[3] = {10, 20, 30};
int* ptr = arr;
cout << *(ptr + 1); // valid → 20
```

But what if you go beyond the allocated size?

```cpp
cout << *(ptr + 10); // ❗ Dangerous!
```

💣 Out-of-Bounds Access: What Happens?

❗ Undefined Behavior (UB)

Accessing memory beyond the bounds of an array (whether with pointers or indices) results in **undefined behavior**.

🧪 What Can Go Wrong?

Here are common outcomes of **undefined behavior**:

Scenario	Result
Accessing nearby memory	You get garbage/unintended data
Accessing protected memory	Segmentation fault (crash) 💥
Overwriting adjacent variables	Data corruption 🧟
Writing into code/data segment	Crash or program compromise 🔓
Nothing apparent happens	Still dangerous, just silent

🧪 Example: Reading Out of Bounds

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

int main() {
    int arr[3] = {1, 2, 3};
    cout << arr[5] << endl; // ❗ UB: may print garbage or crash
    return 0;
}
```

☑ Compiles Fine

Because C++ **doesn't do bounds checking** on raw arrays.

❌ May:

- Print garbage
- Crash
- Corrupt memory

⚔ Analogy: Walking Off a Cliff

Imagine your array is a balcony with a railing (3 steps long). Walking within bounds is safe:

🧍 → Step 1 🧍 → Step 2 🧍 → Step 3

Now try stepping off...

😵 → Step 10 🪦 → You're off the cliff → **Undefined Behavior** 💀

💡 How to Prevent This?

☑ 1. Use `std::array` or `std::vector`

They **offer** `.at(index)` which throws an exception if index is out of bounds:

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

int main() {
    array<int, 3> arr = {1, 2, 3};
    cout << arr.at(5); // Throws std::out_of_range exception
}
```

☑ 2. Manual Bounds Checking

Always check before access:

```
if (i >= 0 && i < size) {
    cout << arr[i];
}
```

☑ Summary

Action	Result
`ptr + index` out-of-bounds	❗ Undefined Behavior
`arr[index]` out-of-bounds	❗ Undefined Behavior
`std::vector::at(index)`	☑ Exception (safe)

🧩 Why `char` Arrays Behave Differently in `cin` / `cout` (Compared to Other Types)

🔍 The Observation

```
int arr[] = {1, 2, 3};
char cArr[] = "Hello";

cout << arr << endl;     // prints a memory address (like 0x61ff08)
cout << cArr << endl;    // prints: Hello ✅
```

Why? Let's deep dive 👇

🎭 What's Happening Behind the Scenes?

🟣 For Non-`char` Arrays (e.g., `int[]`):

```
int arr[] = {1, 2, 3};
cout << arr;
```

This prints the **pointer** (address) of the first element — like `0x7ffeab2d`.

Because:

- `cout` sees `int*` (a pointer).
- There's **no overload** of `operator<<` for `int*` that prints array contents.
- So it prints the pointer's value (address).

🟢 For `char` Arrays:

```
char cArr[] = "Hello";
cout << cArr;
```

This prints the actual string: `Hello`.

Because:

- `cout` **has a special overload** for `char*`.
- It **treats `char*` as a C-style null-terminated string (`\0`)**.
- It prints characters until it hits a `\0`.

📌 `char[]` decays to `char*`, and `cout` is smart enough to handle this case.

🔍 Behind the Curtain: `cout <<` Overloads

```
ostream& operator<<(ostream&, const char*);     // Special for strings ✅
ostream& operator<<(ostream&, const int*);      // ❌ Not defined — prints
address
```

This is why:

Type	Result
`char*`	Printed as string
`int*`	Printed as address
`double*`	Printed as address

📥 What About `cin >>`?

☑️ Works for `char` arrays (as C-strings):

```
char name[100];
cin >> name; // Reads until first space, adds '\0' at end
```

❌ But for `int[]`, it doesn't work like this:

```
int numbers[3];
cin >> numbers; // ❌ Invalid: `cin` doesn't know how to read into array
```

Why?

- `cin >> name` works because of overload for `char*`
- No such overload for `int[]` or `int*`
- You must use a loop: `cin >> numbers[i];`

🧠 Analogy: Special Guest Treatment 🎤

Imagine `cout` is a host.

- When `char*` walks in, the host goes: 👉 "Ah! A guest with a NAME!" and prints the full name (`Hello`).
- But when `int*` comes in: 👉 "Uhh... just an address? I'll show your seat number (0x7ff...)."

💡 Only `char*` gets this **string interpretation**. Others don't.

🔐 TL;DR Summary

Expression	Behavior	Why?
`cout << char*`	Prints the string	Special overload for `char*`
`cout << int*`	Prints the address	No overload for `int*`

Expression	Behavior	Why?
`cin >> char[]`	Reads a word, adds `\0`	Uses `istream >> char*`
`cin >> int[]`	❌ Not allowed directly	No overload; use a loop

🔧 Best Practice

✅ Use `std::string` instead of `char[]` for better safety, usability, and flexibility.

```
string name;
cin >> name;
cout << name;
```

🧠 Understanding Character Arrays and Pointers in C++ (with Emojis & Analogies) 🚀

📃 Code

```
char ch[] = "Babbar";
char* cptr = ch;
```

🧩 What's Happening Here?

- 📦 `ch` is a **character array** holding the string `"Babbar"` (`'B'`, `'a'`, `'b'`, `'b'`, `'a'`, `'r'`, `'\0'`)
- 📍 `cptr` is a pointer pointing to the **first element** of that array — the character `'B'`.

✅ 1. `cout << ch << endl;`

🔎 **Analogy:** You show the full message starting from the mailbox 📫 📤 **Output:**

```
Babbar
```

🧠 `ch` decays into a `char*`, and `cout` prints the null-terminated string.

✅ 2. `cout << &ch << endl;`

🔎 **Analogy:** Showing the full mailbox's location 🏠 📤 **Output (likely):**

```
Babbar
```

🧠 Even though `&ch` is technically `char (*)[7]`, most compilers treat it like a pointer to the string.

✅ 3. `cout << &cptr << endl;`

🔎 **Analogy:** Showing the location of the bookmark 📌 📤 **Output:**

```
0x61fefc (some memory address)
```

🧠 Prints the address where the pointer `cptr` is stored — not what it points to.

✅ 4. `cout << *cptr << endl;`

🔎 **Analogy:** Reads the letter pointed to by the bookmark 💌 📤 **Output:**

```
B
```

🧠 Dereferences the pointer and gives the first character.

✅ 5. `cout << *(cptr + 3) << endl;`

🔎 **Analogy:** Peek 3 steps ahead in the message 📤 **Output:**

```
b
```

🧠 `cptr + 3` points to the fourth character (`'b'` at index 3).

✅ 6. `cout << cptr << endl;`

🔎 **Analogy:** Read the full message from the bookmark 📖 📤 **Output:**

```
Babbar
```

🧠 `cptr` points to the start of the string, so it prints from there.

✅ 7. `cout << ch[0] << endl;`

🔎 **Analogy:** Directly grabbing the first letter in the array 🫴 📤 **Output:**

```
B
```

🧠 Accesses the first element of the character array using indexing.

☑️ 8. `cout << &ch[0] << endl;`

🔎 **Analogy:** Asking for the address of the first letter, and then reading from it 📦 📤 **Output:**

```
Babbar
```

🧠 `&ch[0]` is equivalent to `ch` and `cptr` — points to the start of the string.

🧪 Full Polished Code:

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

int main() {
    char ch[] = "Babbar";
    char* cptr = ch;

    cout << "ch: " << ch << endl;
    cout << "&ch: " << &ch << endl;
    cout << "&cptr: " << &cptr << endl;
    cout << "*cptr: " << *cptr << endl;
    cout << "*(cptr + 3): " << *(cptr + 3) << endl;
    cout << "cptr: " << cptr << endl;
    cout << "ch[0]: " << ch[0] << endl;
    cout << "&ch[0]: " << &ch[0] << endl;

    return 0;
}
```

📊 Summary Table

💡 Expression	🔍 Output Example	🧠 Meaning
`cout << ch`	Babbar	Entire string from the array
`cout << &ch`	Babbar	Address of array, interpreted as string
`cout << &cptr`	0x...	Address of pointer variable itself
`cout << *cptr`	B	First character pointed by the pointer
`cout << *(cptr+3)`	b	4th character (index 3) in the string

💡 Expression	🔍 Output Example	🧠 Meaning
`cout << cptr`	Babbar	Pointer to the string, prints from start
`cout << ch[0]`	B	First character using array index
`cout << &ch[0]`	Babbar	Address of first char, interpreted as string

🧠 TL;DR

- 🧵 `ch`, `&ch[0]`, and `cptr` all point to the same location — beginning of `"Babbar"` 🎯
- 💥 `&ch` might look different but acts like `ch` when printed.
- 🧭 `&cptr` gives the pointer's own location.
- 🎲 `*cptr`, `ch[0]` both give the first character — `'B'`.
- 🎯 `cptr + n` and `*(cptr + n)` let you step through the string character by character.

Final Code

```
#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
}
```