C++ Programming For Absolute Beginners & Question Set II

Sahas Talasila

Contents

In C++, a **pointer** is a variable that holds the memory address of another variable. Pointers are powerful because they allow you to:

- Directly access and modify memory locations.
- Pass large data structures to functions efficiently by reference (address) rather than by value (copy).
- Dynamically allocate and deallocate memory on the heap as needed, giving more control over how memory is used.

However, with great power comes great responsibility. Incorrect use of pointers can lead to errors such as:

- **Memory leaks:** Forgetting to release dynamically allocated memory.
- **Dangling pointers:** Pointers that still reference memory that has already been deallocated.
- **Segmentation faults (crashes):** Accessing invalid memory.

1.1 Pointers and Memory: A Visual Overview

To better understand pointers, let's visualize how variables and pointers map to memory. Suppose we have a simple scenario:

```cpp
int x = 10;     // x is stored in some memory location
int* ptr = &x; // ptr stores the address of x
```

Conceptually, in memory, it might look like this:

```
Memory Address      Value
     0x100          10    <--- This is x
     0x104          ...
     0x108          ...
     0x10C   -->   0x100  <--- This is ptr (contains address 0x100)
```

Here, `x` resides at address `0x100` (example only), and `ptr` at address `0x10C`. The contents of `ptr` are `0x100`, meaning "pointer `ptr` holds the address of `x`."

1.2 Declaration and Initialization

A pointer is declared by specifying the type of data it will point to, followed by an asterisk (*) and a pointer name. For example:

```cpp
int* ptr;     // Pointer to an integer
double* dptr; // Pointer to a double
```

You can also write `int *ptr;` or `int * ptr;`, which are stylistically the same in C++. To obtain the address of a variable, use the & (address-of) operator:

```cpp
int x = 10;
int* ptr = &x; // ptr now holds the address of x
```

Common Error: Uninitialized Pointer

- **What happens?** If you declare a pointer but never assign it an address, it may point to a random memory location.
- **Error symptom:** Attempting to dereference an uninitialized pointer often causes a *segmentation fault* or undefined behavior.

- **How to fix:** Always initialize pointers, for example:

```
    int* ptr = nullptr;  // or int* ptr = &someVariable;

```

1.3 Dereferencing a Pointer

To access the *value* stored at the address contained in a pointer, use the * (dereference) operator:

```cpp
int x = 10;
int* ptr = &x;
std::cout << *ptr << std::endl; // Prints the value of x (10)
```

Changing the value through a pointer:

```cpp
*ptr = 20;  // Now x = 20
```

Visually:

```
ptr ------> x (value = 10)

*ptr is the same as x, so *ptr = 20 modifies x to 20.
```

Common Error: Null Pointer Dereference

- **What happens?** If `ptr` is `nullptr` (or `NULL`), dereferencing it (`*ptr`) leads to invalid memory access.
- **Error symptom:** Typically results in a crash (segmentation fault).
- **How to fix:** Always check if a pointer is `nullptr` before dereferencing:

```cpp
    if (ptr != nullptr) {
        *ptr = 100;
    }

```

1.4 Dynamic Memory Allocation

Dynamic memory allocation allows you to request memory from the *heap* at runtime. This is useful if the size of an array or the lifespan of an object is not known until the program is running.

In C++, dynamic memory is allocated using the `new` operator and deallocated using the `delete` operator.

1.4.1 Allocating and Deallocating a Single Object

```cpp
int* ptr = new int; // Allocates an integer on the heap
*ptr = 42;          // Store a value
std::cout << *ptr << std::endl; // Output: 42

delete ptr;   // Deallocate memory to prevent memory leak
ptr = nullptr; // Optional, but good practice to avoid dangling pointer
```

Diagrammatically:

```
Stack                        Heap
-----                        ----
ptr  --------------------> [  42  ]
                          (allocated by new)
... other local variables
```

When you do `delete ptr;`:

```
* Freed memory on the heap *
ptr still has the old address,
but that memory is no longer valid
```

Setting `ptr = nullptr;` means `ptr` is no longer pointing to invalid memory.

1.4.2 Allocating and Deallocating Arrays

```cpp
int* arr = new int[5]; // Allocates an array of 5 ints on the heap
for(int i = 0; i < 5; i++) {
    arr[i] = i * 2; // Assign values
}
delete[] arr; // Use delete[] for arrays
arr = nullptr;
```

Visually:

```
Stack                      Heap
-----                      ----
arr ---------------> [ 0   | 2   | 4   | 6   | 8   ]
                      size=5   (allocated by new[])
```

Common Error: Using `delete` instead of `delete[]`

- **What happens?** If you allocated with `new[]` but used `delete` (without brackets) to deallocate, it leads to undefined behavior.
- **Error symptom:** May cause crashes or corrupt your memory.
- **How to fix:** Always match `new[]` with `delete[]`:

```cpp
    int* arr = new int[5];
    delete[] arr;  // correct

```

1.5 Memory Leaks and Dangling Pointers

- A **memory leak** occurs when you lose the address of allocated memory without deallocating it.
- A **dangling pointer** occurs when a pointer points to memory that has already been freed.

```cpp
int* leakPtr = new int(100);
// Some code that never calls delete on leakPtr
// Eventually, leakPtr goes out of scope or is reassigned
// The memory allocated for '100' is never freed => Memory leak
```

```cpp
int* dangPtr = new int(50);
delete dangPtr;
// dangPtr still points to that memory address, but it's invalid now.
std::cout << *dangPtr << std::endl; // undefined behavior
```

How to Avoid:

- Always pair `new` with `delete` and `new[]` with `delete[]`.
- Set pointers to `nullptr` after deleting them.

- Use RAII (Resource Acquisition Is Initialization) techniques, smart pointers (`std::unique_ptr`, `std::shared_ptr`), or other memory management strategies in modern C++.

1.6 Smart Pointers (Brief Mention)

Example: Dangling Pointer In modern C++, you can avoid many pointer pitfalls by using *smart pointers* from the `<memory>` header, such as:

- `std::unique_ptr<T>`: Exclusively owns the resource, automatically deletes it when out of scope.
- `std::shared_ptr<T>`: Allows multiple owners of a resource, deleted when the last reference goes out of scope.

These can prevent memory leaks and dangling pointers because the deletion is handled automatically by RAII.

1.7 Summary of Pointer Errors and Resolutions

Table 1: Common Pointer-Related Errors

Error	**Cause**	**Solution**
Uninitialized Pointer	Declaring a pointer but never assigning an address	Initialize pointer to `nullptr` or a valid address
Null Pointer Dereference	Dereferencing a pointer that is `nullptr`	Check if pointer is `nullptr` before dereferencing
Memory Leak	Not calling `delete`/`delete[]` for dynamically allocated memory	Ensure each `new`/`new[]` has a matching `delete`/`delete[]`
Dangling Pointer	Pointer still references memory that is already freed	Set pointer to `nullptr` after deletion, avoid using it afterwards
Mismatched Delete	Using `delete` on memory allocated with `new[]`	Always use `delete` for single objects, `delete[]` for arrays

1.8 Example: Safe Usage of Dynamic Memory

Below is an example program illustrating safe pointer usage, dynamic allocation, and deallocation:

```cpp
#include <iostream>

int main() {
    // 1. Allocate a single integer
    int* singleInt = new int;
    *singleInt = 42;
    std::cout << "singleInt value: " << *singleInt << std::endl;

    // 2. Allocate an array
    int size;
    std::cout << "Enter the number of elements: ";
    std::cin >> size;

    int* arr = new int[size];
    for(int i = 0; i < size; i++) {
        arr[i] = i * 10;
    }

    std::cout << "Array elements: ";
    for(int i = 0; i < size; i++) {
        std::cout << arr[i] << " ";
    }
    std::cout << std::endl;

    // 3. Deallocate both
    delete singleInt;
    singleInt = nullptr;

    delete[] arr;
    arr = nullptr;

    return 0;
}
```

Key Takeaways:

- We used `new` and `delete` correctly.
- We used `new[]` and `delete[]` correctly.
- Setting pointers to `nullptr` after deletion helps avoid accidental usage later.

Multiple Choice Questions (MCQs)

1. Which operator is used to allocate memory dynamically in C++?
   (a) `malloc()`
   (b) `new`
   (c) `alloc()`
   (d) `malloc_cpp()`
2. Which of the following is the correct way to declare a pointer to an integer?
   (a) `int ptr;`
   (b) `int* ptr;`
   (c) `int ptr*;`

   (d) `pointer<int> ptr;`

3. Which symbol is used to get the address of a variable?
   (a) `*`
   (b) `&`
   (c) `%`
   (d) `#`
4. When using `delete` on an array allocated by `new`, which syntax should be used?
   (a) `delete arr;`
   (b) `delete[] arr;`
   (c) `free(arr);`
   (d) `arr.delete();`
5. What is the result of dereferencing a pointer that has not been initialized?
   (a) It points to the first memory location in the program.
   (b) It always gives 0.
   (c) **Undefined behavior.**
   (d) It produces a compile-time error.
6. Which of the following statements about dynamic memory allocation is true?
   (a) Memory is allocated on the stack.
   (b) You must free it explicitly in Java.
   (c) **Memory is allocated on the heap and must be freed explicitly in C++.**
   (d) It is managed automatically by the C++ runtime.
7. What happens if you use `delete` on a pointer that was not allocated by `new`?
   (a) The program ignores the request.
   (b) **The program exhibits undefined behavior.**
   (c) The pointer is automatically set to `nullptr`.
   (d) Nothing happens.
8. Which of the following is **not** a benefit of using pointers?
   (a) They can allow dynamic memory management.
   (b) They can be used to pass large objects to functions efficiently.
   (c) **They automatically prevent memory leaks.**
   (d) They can help with implementing data structures like linked lists.
9. If `p` is a pointer to an integer, what does the expression `p + 3` mean?
   (a) Add 3 bytes to the address in `p`.
   (b) **Point to the 3rd integer ahead of `p`. (According to the size of `int`.)**
   (c) Add 3 to the value pointed to by `p`.

(d) This is invalid syntax.

10. Which of the following best describes a **dangling pointer**?
    (a) A pointer that has a `nullptr` value.
    (b) A pointer that was declared but never initialized.
    (c) **A pointer that points to memory that has been freed or deallocated.**
    (d) A pointer that points to an array's first element.

Short Form Questions

1. What does the `*` operator do when placed in front of a pointer variable?
2. How do you obtain the address of a variable in C++?
3. Which C++ operator is used to free dynamically allocated memory for a single object?
4. Which C++ operator is used to allocate memory for an array?
5. What is a **memory leak**?
6. Define a **dangling pointer**.
7. Why should we set a pointer to `nullptr` after deleting it?
8. What is the difference between `new` and `new[]`?
9. How do you dynamically allocate a single `double` variable in C++?
10. Which keyword can you use instead of `NULL` in modern C++ to indicate an empty pointer?

Open Ended (Coding) Questions

1. **Dynamic Array Function:** Write a C++ function `createArray` that:
   - Dynamically allocates an integer array of size `n`.
   - Fills the array with values from `1` to `n`.
   - Returns the pointer to the array.

   Then, write a `main` function that calls `createArray`, prints the values, and frees the memory.
2. **Singly Linked List Implementation:** Implement a singly linked list class in C++ with the following operations:
   - `pushFront(int val)`: Inserts a new node at the beginning.
   - `popFront()`: Removes the first node (if it exists).
   - `printList()`: Prints all node values.
   - Proper destructor to free all nodes.

   Demonstrate these operations in `main`.
3. **2D Dynamic Array (Double Pointer):** Write a C++ program that:
   - Dynamically allocates a 2D integer array of size `rows x cols`.
   - Fills it with some sample values (e.g., row index + column index).
   - Prints the entire 2D array.

- Frees the allocated memory.

4. **Reading Unknown Number of Integers:** Create a program that:
   - Continuously reads integer input from `std::cin` until a non-integer or end-of-file is encountered.
   - Stores these integers in a dynamically allocated array that grows as needed (e.g., double the size whenever full).
   - Prints all the integers read and then deallocates the memory.
5. **Smart Pointer Demonstration:** Write a short code snippet that (**EXTRA**):
   - Creates a `std::unique_ptr<int>` and allocates an integer.
   - Assigns a value and prints it.
   - Shows that when the `unique_ptr` goes out of scope, memory is automatically freed.

   Compare this usage to a raw pointer to highlight how smart pointers help prevent memory leaks.

Spot the Error

1. Spot the error in this pointer initialization:

```
int *ptr;
*ptr = 10;
```

2. Spot the error in this dynamic memory allocation:

```
int *arr = malloc(5 * sizeof(int));
arr[0] = 10;
```

3. Spot the error in this deallocation logic:

```
int *p = new int;
delete p[];
```

4. Spot the error in using a pointer after deletion:

```
int *data = new int(42);
delete data;
std::cout << *data;
```

5. Spot the error in freeing stack memory:

```
int x = 100;
free(&x);
```

6. Spot the error in this memory leak scenario:

```
int *ptr = new int(10);
ptr = new int(20);
```

7. Spot the error in this null pointer dereference:

```
int *p = nullptr;
*p = 5;
```

8. Spot the error in this reallocation example:

```
int *arr = (int*) malloc(3 * sizeof(int));
arr = realloc(arr, 6 * sizeof(int));
if (arr == NULL)
    std::cout << "Reallocation failed";
```