C++ Pointers

Setenay Tutucu

January 2025

0.1 Introduction

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.

0.2 Pointers in C++

A pointer in C++ is a variable that stores the memory address of another variable. Instead of holding a data value, a pointer holds the location in memory where the data is stored.

0.2.1 Address-of Operator (&)

The address of a variable can be obtained by preceding the name of a variable with an ampersand sign (&), known as address-of operator. For example:

```
&variable
```

Here, variable is any valid variable, and & variable will give the memory address of that variable.

0.2.2 Dereference Operator (*)

An interesting property of pointers is that they can be used to access the variable they point to directly. This is done by preceding the pointer name with the dereference operator (*). The operator itself can be read as "value pointed to by".

0.2.3 Declaring Pointers

Due to the ability of a pointer to directly refer to the value that it points to, a pointer has different properties when it points to a char than when it points to an int or a float. Once dereferenced, the type needs to be known. And for that, the declaration of a pointer needs to include the data type the pointer is going to point to.
Here is the general syntax for declaring a pointer:

```
type* pointer_name;
```

- **type:** The data type of the variable that the pointer will point to (e.g., int, float, char).

- **pointer_name:** The name of the pointer variable.

For Example:

```
    int * number;
    char * character;
    double * decimals;
```

These are three declarations of pointers. Each one is intended to point to a different data type, but, in fact, all of them are pointers and all of them are likely going to occupy the same amount of space in memory (the size in memory of a pointer depends on the platform where the program runs).

Nevertheless, the data to which they point to do not occupy the same amount of space nor are of the same type: the first one points to an int, the second one to a char, and the last one to a double. Therefore, although these three example variables are all of them pointers, they actually have different types: int*, char*, and double* respectively, depending on the type they point to.

Note: Note that the asterisk (*) used when declaring a pointer only means that it is a pointer (it is part of its type compound specifier), and should not be confused with the dereference operator seen a bit earlier, but which is also written with an asterisk (*). They are simply two different things represented with the same sign.
For example:

```
    int num = 10;
    int* ptr = &num;   // Assigning the address of 'num' to
       the pointer 'ptr'
    int value = *ptr;  // Dereferencing the pointer to access
        the value stored at the address
```

0.2.4 Pointer Arithmetic

To conduct arithmetical operations on pointers is a little different than to conduct them on regular integer types. To begin with, only addition and subtraction operations are allowed; the others make no sense in the world of pointers.

You can increment or decrement a pointer, and the pointer will move by the size of the data type it points to.

For Example:

```cpp
    int arr[] = {1, 2, 3, 4, 5};
    int* ptr = arr;    // 'ptr' points to the first element of the array

    ptr++;  // Moves the pointer to the next element in the array
    cout << *ptr;  // Output: 2, as ptr now points to arr[1]
```

In this example, ptr++ moves the pointer to the next integer element in the array. The size of the pointer moves by the size of the type it points to, i.e., sizeof(int).

0.2.5 Pointers and Const

Pointers can be used to access a variable by its address, and this access may include modifying the value pointed. But it is also possible to declare pointers that can access the pointed value to read it, but not to modify it. For this, it is enough with qualifying the type pointed to by the pointer as const. For example:

```cpp
    int x;
    int y = 10;
    const int * p = &y;
    x = *p;           // ok: reading p
    *p = x;           // error: modifying p, which is const-qualified
```

0.2.6 Pointers to Arrays

Arrays and pointers are closely related in C++. The name of an array can be used as a pointer to the first element of the array. This makes arrays and pointers interchangeable in many contexts.
Example:

```
    int arr[3] = {1, 2, 3};
    int *ptr = arr;    // ptr points to the first element of the array
    cout << *(ptr + 1);  // Outputs 2 (second element)
```

0.2.7 Dynamic Memory Allocation in C++

C++ allows for dynamic memory allocation and deallocation through the new and delete operators. The new operator allocates memory on the heap, while delete is used to free it. Proper management of dynamic memory is crucial to prevent memory leaks.
Example:

```
    int *ptr = new int(10);  // Dynamically allocate memory for one integer
    cout << *ptr;            // Outputs 10
    delete ptr;              // Free the dynamically allocated memory
```

0.2.8 Types of Pointers

Pointers come in various types, each suited to different use cases.

Void Pointers

The void type of pointer is a special type of pointer. In C++, void represents the absence of type. Therefore, void pointers are pointers that point to a value that has no type (and thus also an undetermined length and undetermined dereferencing properties).
This gives void pointers a great flexibility, by being able to point to any data type, from an integer value or a float to a string of characters.
Example:

```
    void *ptr;
    int num = 5;
    float pi = 3.14;

    ptr = &num;  // ptr points to an integer
    ptr = &pi;   // ptr now points to a float
```

In exchange, they have a great limitation: the data pointed to by them cannot be directly dereferenced (which is logical, since we have no type to

dereference to), and for that reason, any address in a void pointer needs to be transformed into some other pointer type that points to a concrete data type before being dereferenced.
Example:

```
void *ptr;
int a = 10;
ptr = &a;

// You need to cast it to the appropriate type before dereferencing
std::cout << *(static_cast<int *>(ptr)) << std::endl;  // Output: 10
```

Null Pointers

A null pointer is a pointer that doesn't point to any valid memory location. In C++, the nullptr keyword is used to represent a null pointer.
Example:

```
int *ptr = nullptr;  // A null pointer
```

Note: Do not confuse null pointers with void pointers! A null pointer is a value that any pointer can take to represent that it is pointing to **"nowhere"**, while a void pointer is a type of pointer that can point to somewhere without a specific type. One refers to the value stored in the pointer, and the other to the type of data it points to.

Dangling Pointer

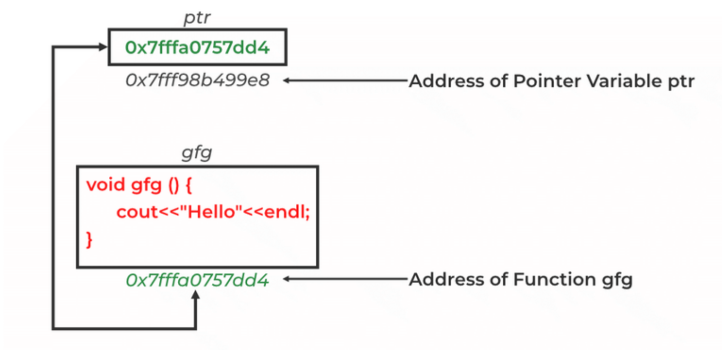
A dangling pointer is a pointer that points to a memory location that has been deallocated or freed. Using a dangling pointer is dangerous and can cause undefined behavior.
Example:

```
int *ptr = new int(10);  // Dynamically allocated memory
delete ptr;              // Memory is deallocated
// Now, ptr is a dangling pointer.
cout << *ptr;            // Undefined behavior, accessing dangling pointer
```

To avoid dangling pointers, always set pointers to nullptr after deleting the memory they point to.

Function Pointer

In C++, a function pointer is a pointer that points to the address of a function instead of pointing to data. Just like a normal pointer holds the memory address of a variable, a function pointer holds the memory address of a function.



This allows you to call functions dynamically or pass functions as arguments to other functions.
The syntax for declaring a function pointer is as follows:

```
return_type (*pointer_name)(parameter_types);
```

- **return_type:** The return type of the function that the pointer will point to.
- **pointer_name:** The name of the pointer variable.
- **parameter_type:** The types of parameters that the function takes.

For Example:

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

// A simple function that adds two integers
int add(int a, int b) {
return a + b;
}

int main() {
// Declare a function pointer
int (*func_ptr)(int, int);

```

```
    // Point the pointer to the 'add' function
    func_ptr = add;

    // Use the function pointer to call the 'add' function
    int result = func_ptr(3, 4);

    // Print the result
    cout << "The result is: " << result << endl;

    return 0;
}
```

Output:

```
    The result is: 7
```

0.2.9 Smart Pointers

Pointers are a fundamental feature in C++ that provide the ability to directly access and manipulate memory addresses.
However, improper use of pointers can lead to severe memory management issues like memory leaks, dangling pointers, and buffer overflows. In response to these problems, C++ introduced **smart pointers**, a feature that helps automate memory management and improve the safety and efficiency of programs.

Garbage Collection Mechanism

In C++, memory is primarily managed manually through the use of pointers and the new/delete operators. Unlike languages such as Java or Python, C++ does not have an automatic garbage collection mechanism. Garbage collection refers to the automatic process of reclaiming memory that is no longer needed by the program, thus preventing memory leaks.

Since C++ does not provide built-in garbage collection, developers must manually ensure that memory allocated with new is freed using delete. This gives C++ more control over memory management but places a greater burden on the developer to avoid common memory.

C++ introduced smart pointers in the Standard Library (since C++11) to mitigate the dangers of raw pointers and improve memory safety. Smart pointers automate memory management, freeing developers from the responsibility of explicitly calling delete.

auto_ptr

The `auto_ptr` was introduced in earlier versions of C++ but has since been deprecated in C++11 and removed in C++17. It provided automatic memory management by deleting the managed object when the `auto_ptr` goes out of scope. However, had major flaws, particularly when it comes to ownership transfer: copying an `auto_ptr` caused ownership transfer instead of a copy, which could lead to confusion and unintentional memory deallocation.

It is recommended to avoid using `auto_ptr` in modern C++ and instead use other smart pointers like std::`unique_ptr`.

unique_ptr

The `std::unique_ptr` is a smart pointer that provides exclusive ownership of a dynamically allocated object. It automatically deallocates the memory when it goes out of scope and ensures that no other `unique_ptr` can share ownership of the same object.
`unique_ptr` supports move semantics, meaning it allows the transfer of ownership but does not allow copying.

- **When to Use:** Use `unique_ptr` when you need sole ownership of a dynamically allocated object. It is ideal for managing resources like memory, file handles, or network connections in cases where only one entity should own the resource at a time.

Example:

```
    std::unique_ptr<int> ptr = std::make_unique<int>(5);
```

shared_ptr

`std::shared_ptr` allows shared ownership of an object. Multiple `shared_ptr` instances can point to the same object, and the object is only destroyed when the last `shared_ptr` goes out of scope or is reset. This is achieved through reference counting.

- **Reference counting:** Reference counting is a technique where each object being managed (in this case, the object pointed to by `shared_ptr`) keeps track of how many `shared_ptr` instances (owners) currently point to it.
  Each time a new `shared_ptr` is created to point to the same object, the reference count is increased.
  When a `shared_ptr` is destroyed (or goes out of scope), the reference count is decreased.
  Once the reference count reaches zero (i.e., there are no more `shared_ptr` instances pointing to the object), the resource (the object) is automatically deleted, and memory is freed.

- **When to Use:** Use `shared_ptr` when multiple parts of a program need shared access to the same object, and you want to ensure automatic memory management. Be cautious of cyclic references, which can cause memory leaks unless addressed with `weak_ptr`.

Example:

```
    std::shared_ptr<int> ptr1 = std::make_shared<int>(10);
    std::shared_ptr<int> ptr2 = ptr1; // Both ptr1 and ptr2 share ownership
```

weak_ptr

A `std::weak_ptr` is used to prevent cyclic references when using `shared_ptr`. Unlike `shared_ptr`, `weak_ptr` does not increment the reference count, thus preventing circular references that would otherwise prevent memory from being deallocated. A `weak_ptr` can be converted into a `shared_ptr` using the lock() method, which returns a valid `shared_ptr` if the object is still alive.

- **When to Use:** Use `weak_ptr` when you need to observe an object managed by a `shared_ptr` without affecting its lifetime. It is particularly useful for breaking reference cycles in graphs or other circular data structures.

Example:

```
    std::weak_ptr<int> weakPtr = ptr1; // weak_ptr does not affect reference count
    std::shared_ptr<int> lockedPtr = weakPtr.lock(); // Converts to shared_ptr if object is still valid
```

0.2.10 Cons of Raw Pointers in C++

Raw pointers provide powerful mechanisms for direct memory access but come with several disadvantages that lead to issues in memory safety and application stability:

- **Memory Leaks:** A memory leak occurs when dynamically allocated memory is not freed before the pointer goes out of scope or is reassigned. Over time, memory leaks can accumulate, consuming system resources and leading to application crashes or poor performance.

- **Dangling Pointers:** A dangling pointer is a pointer that references memory that has already been deallocated. Dereferencing such a pointer leads to undefined behavior and potential crashes.

- **Wild Pointers:** A wild pointer is an uninitialized pointer that points to an arbitrary memory location. Dereferencing a wild pointer can cause unpredictable results, including crashes or data corruption.

- **Data Inconsistency:** When multiple raw pointers access the same memory without proper synchronization, data inconsistency can arise. This is particularly problematic in multithreaded environments, where race conditions may cause inconsistent state or corrupt data.

- **Buffer Overflow:** A buffer overflow occurs when data is written beyond the allocated memory boundaries, potentially overwriting adjacent memory. This can lead to application crashes, security vulnerabilities, and undefined behavior. Raw pointers are often involved in buffer overflows because the programmer must manually manage memory bounds.

Example usage of pointers is given below:

```cpp
#include<iostream>
#include<memory>
#include <utility>
template <typename T>
class MyPointer{
private:
    T* ptr;
public:
    explicit MyPointer(T* p = nullptr) : ptr(p) {}

```

```
    ~MyPointer() {
        delete ptr;
    }

    T& operator*() {
        return *ptr;
    }

    T* operator -> () {
        return ptr;
    }
};
class Test {
    public:
    void printMessage () {
        std::cout << "Smart Pointer is working." << std::endl;
    }
};


int main () {
    MyPointer <int> sp1(new int (10));
    std::cout << "Value: " << *sp1 << std::endl;
    MyPointer<Test> sp2(new Test());
    sp2->printMessage();


    std::auto_ptr<int> auto_p1 (new int);
    *auto_p1.get()=10;
    std:: cout << "auto_p1 points to: " << *auto_p1 << std::endl;
    std::cout << "auto_p1's location is: " << &auto_p1 << std::endl;
    std::auto_ptr<int> auto_p2 = std::move(auto_p1);
    std:: cout << "auto_p2 is points to: " << *auto_p2 << std::endl;
    std:: cout << "auto_p2's location is: " << &auto_p2 << std::endl;
    std:: cout << "After move , auto_p1' location is: " << &auto_p1 << std::endl;

/* An auto_ptr owns the object it holds a pointer to. Copying an auto_ptr copies the pointer and transfers ownership to the destination.
```

```
If more than one auto_ptr owns the same object at the same time the behavior of the program is undefined.
The uses of auto_ptr include providing temporary exception-safety for dynamically allocated memory,
passing ownership of dynamically allocated memory to a function, and returning dynamically allocated memory from a function. std::unique_ptr should be used instead*/

    std::unique_ptr<int> unique_p1 (new int);
    *unique_p1.get() = 20;
    std:: cout << "unique_p1 points to: " << *unique_p1 << std::endl;
    std:: cout << "unique_p1's location is: " << &unique_p1 << std::endl;
    std::unique_ptr<int> unique_p2 = std::move(unique_p1);
    std::cout << "unique_p2 is points to: " << *unique_p2 << std::endl;
    std:: cout << "unique_p2's location is: " << &unique_p2 << std::endl;
    std:: cout << "After move, unique_p1's location is: " << &unique_p1 << std::endl;



    std::shared_ptr<int> shared_p1 (new int);
    *shared_p1.get() = 30;
    std::cout << "shared_p1 points to:" << *shared_p1 << std::endl;
    std::cout << "Use cout of shared_p1: " << shared_p1.use_count() <<std::endl;
    std::shared_ptr<int> shared_p2 = shared_p1;
    std:: cout << "shared_p2 points to: " << *shared_p2 << std::endl;
    std:: cout << "shared_p1's location is: " << &shared_p1 << std::endl;
    std:: cout << "shared_p2's location is: " << &shared_p2 << std::endl;
    std::cout << "Use cout shared_p1: " << shared_p1.use_count() << std::endl;
    std::cout << "Use cout shared_p2: " << shared_p2.use_count() << std:: endl;
    shared_p1.reset(); //reset shared_p1
    // Output the use_count of shared_p2 after resetting shared_p1
    std::cout << "After reseting shared_p1 use cout of
```

```
        shared_p2: " <<shared_p2.use_count() << std::endl;


    std::shared_ptr<int> ptr1 (new int);
    *ptr1.get() = 40;
    std::shared_ptr<int> ptr2 = ptr1;
    std::cout << "Use cout ptr1: " << ptr1.use_count() << std
        ::endl;
    std::cout << "Use cout ptr2: " << ptr2.use_count() << std
        :: endl;
    std::weak_ptr<int> wptr1 = ptr1;
    std::weak_ptr<int> wptr2 = ptr2;
    std::cout << "After wptr1 and wptr2 use cout pt1: " <<
        ptr1.use_count() << std::endl;
    std::shared_ptr<int> loced_ptr1 = wptr1.lock();
    std::cout << "After lock function use cout ptr1: " <<
        ptr1.use_count() << std::endl;
    std::shared_ptr<int> moved_ptr2 = std::move(ptr2);
    std:: cout<< "moved_ptr2 use_cout: " << moved_ptr2.
        use_count() << std::endl;
    std::cout<< "ptr2 use_cout: " << ptr2.use_count() << std
        ::endl;


    return 0;
}
```

Output:

```
Value: 10
Smart Pointer is working.
auto_p1 points to: 10
auto_p1's location is: 0x7fffffffd4f0
auto_p2 is points to: 10
auto_p2's location is: 0x7fffffffd4f8
After move , auto_p1' location is: 0x7fffffffd4f0
unique_p1 points to: 20
unique_p1's location is: 0x7fffffffd500
unique_p2 is points to: 20
unique_p2's location is: 0x7fffffffd508
After move, unique_p1's location is: 0x7fffffffd500
shared_p1 points to:30
Use cout of shared_p1: 1
shared_p2 points to: 30
shared_p1's location is: 0x7fffffffd510
shared_p2's location is: 0x7fffffffd520
Use cout shared_p1: 2
Use cout shared_p2: 2
After reseting shared_p1 use cout of shared_p2: 1
Use cout ptr1: 2
Use cout ptr2: 2
After wptr1 and wptr2 use cout pt1: 2
After lock function use cout ptr1: 3
moved_ptr2 use_cout: 3
ptr2 use_cout: 0
```