C++ Programming

Handling Pointers

Mostafa S. Ibrahim
Teaching, Training and Coaching since more than a decade!

Artificial Intelligence & Computer Vision Researcher
PhD from Simon Fraser University - Canada
Bachelor / Msc from Cairo University - Egypt
Ex-(Software Engineer / ICPC World Finalist)

In Practice

- If your struct wants to use objects from another structs, **pointer** is the way
  - vector<Employee> deps;
    - The same employee object will be **copied in memory**
  - vector<**Employee***> deps;
    - Same object and several guys see it
  - Another way: Use object in one place only + ID for whoever need it
- The troubles come from 2 things:
  - Correctly deleting the created pointers
  - Avoiding the loss of pointers
  - In OOP: We will learn how to reduce the pain
  - In Modern C++: Smart pointers make our life much easier!

Struct memory leak

```
1  #include <iostream>
2  using namespace std;
3
4  struct Emplyee {
5      int *xPtr = nullptr;
6
7      Emplyee() {
8          xPtr =  new int[5]{ 1, 2, 3, 4, 5 };
9      }
10
11     void print() {
12         cout<<*xPtr<<" "<<xPtr<<" "<<&xPtr<<"\n";
13     }
14 };
15
16 int main() {
17     int *x = new int {10};
18     delete x;
19     // I created and deleted
20
21     // But this struct creates internally who deletes!
22     Emplyee e;  // memory leak!
23     e.print();
24
```

Destructor

- **Destructor** is a function called once the object is out of scope
  - ~ is the **tilde** character
- We force logic before we lose it
- It is called before complete destruction
- We then free memory
- Use if if you CREATE pointers INSIDE struct

```
4  struct Emplyee {
5      int *xPtr = nullptr;
6
7      Emplyee() {
8          xPtr =  new int[5]{ 1, 2, 3, 4, 5 };
9      }
10
11     ~Emplyee() {    // destroctur
12         cout<<"Bye\n";
13         delete[] xPtr;
14         xPtr = nullptr;
15     }
16
17     void print() {
18         cout<<*xPtr<<" "<<xPtr<<" "<<&xPtr<<"\n";
19     }
20 };
21
22 int main() {
23     Emplyee e;
24     e.print();
25     // 1 0x13dbc20 0x7ffe28f16900
26     // Bye
```

Memory Leak / Dangling Pointer

```
21 int main() {
22     Emplyee e1, e2;
23     e1.xPtr[0] = 20;
24
25     // 20 0x19c 0x7fff
26     e1.print();
27     // 1 0xaaa 0xbb
28     e2.print();
29
30     e2 = e1;
31
32     // 20 0x19c 0xbb
33     e2.print();
34     // Crash!
35     // e2 lost its xPtr value of the created memory
36     // the value of e1 copied
37
38     // problem 1): memory leak as we lost e2 created memory
39     // problem 2): e1 ptr will be deleted twice = dangling = crash
40
41     Emplyee e3 = e1;    // same issue but more implicit
```

Preventing Mistakes (temporarily)

```
 4  struct Emplyee {
 5      int *xPtr = nullptr;
 6      Emplyee() {..
 9      ~Emplyee() {..
13      void print() {..
16
17      // For now: Add these 2 lines when you have internal pointers
18      // They will inform you about soon problems
19      // Later in OOP: Full understanding and proper treatment
20      Emplyee(const Emplyee& e) = delete;
21      void operator = (const Emplyee& e) = delete;
22  };
23
24  int main() {
25      Emplyee e1, e2;
26
27      // NOW CE = saves us from our mistakes
28      e2 = e1;
29      Emplyee e3 = e1;
30
```

"Acquire knowledge and impart it to the people."

"Seek knowledge from the Cradle to the Grave."