CS-1201 Object-Oriented Programming

Destructor

Arbish Akram

Department of Computer Science
Government College University

What Is a Destructor?

- A destructor is a special member function that is automatically called when an object is destroyed.
- Just as a constructor sets an object up when it's created, a destructor performs *shutdown housekeeping* when the object goes out of existence, its complement.
- Common use: freeing memory that was dynamically allocated by the constructor.

Naming and Rules

- Named *~ClassName*: for class `Foo`, the destructor is `~Foo`.
- Takes no arguments and has no return type (not even `void`).
- Cannot be overloaded, a class has exactly one destructor, whether you write it or not.

```
class Foo {
    private:
        int data;
    public:
        Foo() : data(0) {}      // constructor
        ~Foo() {}                // destructor
};
```

The Default Destructor

- If you don't write a destructor, the compiler generates an *empty* one automatically.
- It does nothing extra, it simply lets each member's own destructor (if any) run.
- Fine for classes holding only plain data, but not enough for classes that manage a resource themselves, such as memory obtained with `new`.

When Is a Destructor Called?

- Destructors are called implicitly by the compiler, you never call one directly.
- *Automatic (local) object*: destroyed when execution leaves the scope in which it was created.
- *Dynamically allocated* object (`new`): destroyed, destructor called, when you explicitly `delete` it.
- *Global object*: destructor runs when `main` terminates.

Example 1: Destructor

```
1  // This program demonstrates a destructor.
2  #include <iostream>
3  using namespace std;
4
5  class Demo {
6      public:
7          Demo();   // Constructor
8          ~Demo();  // Destructor
9  };
10
11 Demo::Demo() {
12     cout << "Welcome to the constructor!" << endl;
13 }
14 Demo::~Demo() {
15     cout << "The destructor is now running." << endl;
16 }
17
18 int main() {
19     Demo demoObject; // Define a Demo object
20     cout << "This program demonstrates an object" << endl;
21     cout << "with a constructor and destructor." << endl;
22     return 0;
23 }
```

Example 2: ContactInfo

```
1   class ContactInfo {
2       private:
3           char *name;   // pointer to the name string
4           char *phone;  // pointer to the phone number string
5       public:
6           // Parameters n and p are pointers to char. They receive the address
7           // of the string literals the caller passes in, e.g. "Ali".
8           ContactInfo(char *n, char *p) {
9               // 'new char[...]' allocates a block of memory and returns a pointer
10              name  = new char[strlen(n) + 1]; // +1 for '\0'
11              phone = new char[strlen(p) + 1]; // +1 for '\0'
12              strcpy(name, n);   // copy characters into new memory
13              strcpy(phone, p);  // copy characters into new memory
14          }
15          // Destructor: releases the memory 'new' allocated above.
16          ~ContactInfo() {
17              delete [] name;   // free array pointed to by name
18              delete [] phone;  // free array pointed to by phone
19          }
20          const char *getName() const { return name; }
21          const char *getPhoneNumber() const { return phone; }
22  };
```

Why `ContactInfo` Needs a Destructor

- `name` and `phone` are pointers to memory the constructor allocates with `new char[...]`, sized exactly to fit the given strings.
- That memory does not free itself when the object goes out of scope, it must be released explicitly.
- The destructor calls `delete[] name;` and `delete[] phone;`, so this happens automatically no matter how or where the object is destroyed.
- Without it, every `ContactInfo` object would *leak memory*.

Using ContactInfo

```
1  // This program demonstrates a class with a destructor.
2  #include <iostream>
3  #include "ContactInfo.h"
4  using namespace std;
5
6  int main() {
7      ContactInfo entry("Kristen Lee", "555-2021");
8
9      cout << "Name: "  << entry.getName() << endl;
10     cout << "Phone Number: " << entry.getPhoneNumber() << endl;
11
12     return 0;
13 }
```

- When `entry` goes out of scope at the end of `main`, `~ContactInfo` runs automatically and frees both allocations, no explicit cleanup code needed at the call site.

Order of Construction and Destruction

- Construction and destruction are tied to scope: entering a scope constructs, leaving it destroys.
- In general, *destructors run in the reverse order* of the corresponding constructor calls.
- `static` local objects are the exception to the usual timing: constructed only once, the first time execution reaches their definition, and destroyed only when the program ends (not at block exit).

Summary

- `~ClassName`: no parameters, no return type, cannot be overloaded, exactly one per class.
- Runs automatically: scope exit, `delete`, or program termination, in reverse order of construction.
- Main job: release resources the object acquired, most commonly memory obtained with `new`.
- Looking ahead: a class that allocates memory in its constructor and frees it in its destructor needs special care when an object of that class is copied, that's exactly the problem the copy constructor solves next.