Memory Management: Memory Deallocation

In this lesson, we will learn about the second subsection of memory management - memory deallocation.

WE'LL COVER THE FOLLOWING

Memory Deallocation

delete

A `new` with previously allocated memory will be deallocated with `delete`.

```
Circle* c= new Circle;
...
delete c;
```

The destructors of the object and the destructors of all base classes will be automatically called. If the destructor of the base class is virtual, we can destroy the object with a pointer or reference to the base class.

After the memory of the object is deallocated, the access to the object is undefined. We must initialize the pointer of the object to a point it to a different object.

> STOP The deallocation of `new[]` allocated object with `delete` has undefined behavior.

delete[]

delete[]

You must use the operator `delete[]` for the deallocation of a C array that was allocated with `new[]`.

```
Circle* ca= new Circle[8];
...
delete[] ca;
```

By invoking `delete[]`, all destructors of the objects will automatically be invoked.

>  The deallocation `new` allocated object with `delete[]` is undefined behavior.

Placement Delete

According to placement new, you can implement placement delete. The C++ runtime will not automatically call placement delete. Therefore, it is the programmer's duty to do so.

A commonly used strategy is to invoke the destructor in the first step and to delete it in the second step. The destructor deinitializes the object, and the placement delete deallocates the memory.

```
char* memory= new char[sizeof(Account)];
Account* a= new(memory) Account;  // placement new
...
a->~Account();                    // destructor
operator delete(a, memory);       // placement delete
```

In the next lesson, we will learn how to overload the new and delete operators.