- Example

Here is a working demonstration on the behavior of destructors.

Various scopes

```
#include <iostream>

class HelloGoodBye{
public:
  HelloGoodBye(int i):numb(i){
    std::cout << "Hello from " << numb << ": "<< std::endl;
  }
  ~HelloGoodBye();
private:
  int numb;
};

HelloGoodBye::~HelloGoodBye(){
  std::cout << "Good Bye from : " <<  numb << std::endl;
}

void func(){
  HelloGoodBye helloGoodBye(5);
}

HelloGoodBye helloGoodBye(1);

int main(){

  std::cout << std::endl;

  HelloGoodBye helloGoodBye(2);

  std::cout << std::endl;

  HelloGoodBye* helloGoodByePtr = new HelloGoodBye(3);

  std::cout << std::endl;

  {
    HelloGoodBye helloGoodBye(4);
  }

  std::cout << std::endl;

  delete helloGoodByePtr;

  func();

  std::cout << "--------- End of main ----------"  << std::endl;

}
```

Explanation

The example shows how the destructors of different `HelloGoodbye` instances are called at different times depending on their scopes.

- In line 14, we have explicitly told the destructor to print a statement indicating that the object is about to get deleted.
- The first instance appears at line 21 in which the `numb` attribute has a value of `1`. This object exists outside the scope of `main()`. Hence, it will be destroyed after `main()` ends.
- `helloGoodBye(2)` on line 27 remains active through the scope of `main()`. Its destructor is called once `main()` ends.
- `helloGoodByePtr` presents a case where we need to call `delete` to avoid a memory leak from the pointer. As soon as `delete` is called on line 41, the destructor is invoked.
- `helloGoodBye(4)` on line 36 is inside a nested scope inside `main()`. Since its scope ends the earliest, its destructor is the first to be called.
- `helloGoodBye(5)` on line 18 exists inside the scope of the `func()` function. It is created when `func()` is called and gets destroyed when the function ends.

In the next lesson, we will study the different features of class methods.