Memory Management: Overloading Operator new and delete 1

In this lesson, we will learn how to overload the operators new and delete so we can manage memory in a better way.

WE'LL COVER THE FOLLOWING

It happens quite often that a C++ application allocates memory but does not deallocate it. This is the job for the operators `new` and `delete`. Due to both features, you can explicitly manage the memory management of an application.

Occasionally, we must verify that an application has correctly released its memory. In particular, for programs running for long periods of time, it is a challenge to allocate and deallocate memory from a memory management perspective. Of course, the automatic release of the memory during the shutdown of the program is not an option.

The Baseline

As a baseline for our analysis, we use a simple program that often allocates and deallocates memory.

main.cpp

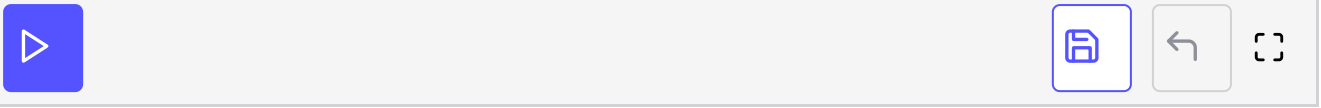
```
// overloadOperatorNewAndDelete.cpp
```

myNew.hpp

myNew2.hpp

myNew3.hpp

```
// #include "myNew.hpp"
// #include "myNew2.hpp"
// #include "myNew3.hpp"

#include <iostream>
#include <string>

class MyClass{
  float* p= new float[100];
};

class MyClass2{
  int five= 5;
  std::string s= "hello";
};


int main(){

    int* myInt= new int(1998);
    double* myDouble= new double(3.14);
    double* myDoubleArray= new double[2]{1.1,1.2};

    MyClass* myClass= new MyClass;
    MyClass2* myClass2= new MyClass2;

    delete myDouble;
    delete [] myDoubleArray;
    delete myClass;
    delete myClass2;

//  getInfo();

}
```



The key question is as follow:, is there a corresponding `delete` to each `new` call?

Operator new

C++ offers the operator new in four variations:

```
void* operator new  (std::size_t count );
void* operator new[](std::size_t count );
void* operator new  (std::size_t count, const std::nothrow_t& tag);
void* operator new[](std::size_t count, const std::nothrow_t& tag);
```

The first two variations will throw a `std::bad_alloc` exception if they can not provide the memory. The last two variations return a null pointer. It's

convenient and sufficient to overload only version 1 since the versions **2 - 4** use version **1**:

```
void* operator new(std::size_t count)
```

This statement also holds for the variants **2** and **4**, which are designed for C arrays. You can read the details of the global `operator new` here.

> 💡 The statements also hold for `operator delete`.

Operator delete

C++ offers six variations for `operator delete` :

```
void operator delete  (void* ptr);
void operator delete[](void* ptr);
void operator delete  (void* ptr, const std::nothrow_t& tag);
void operator delete[](void* ptr, const std::nothrow_t& tag);
void operator delete  (void* ptr, std::size_t sz);
void operator delete[](void* ptr, std::size_t sz);
```

According to the properties of `operator new`, it is sufficient to overload `operator delete` for the first variant since the remaining 5 use `void operator delete(void* ptr)` as a fallback.

Only a word about the two last versions of `operator delete`. In this version, you have the length of the memory block in the variable `sz` at your disposal. Read the details here.

Counting Allocations and Deallocations

Let's use the header `myNew.hpp` (line 3). The same holds for the lines 34. Here we invoke the function `getInfo` to get information about our memory management.

main.cpp

myNew.hpp

myNew2.hpp

```
// overloadOperatorNewAndDelete.cpp

#include "myNew.hpp"
// #include "myNew2.hpp"
// #include "myNew3.hpp"

#include <iostream>
```

myNew3.hpp

```
#include <string>

class MyClass{
  float* p= new float[100];
};

class MyClass2{
  int five= 5;
  std::string s= "hello";
};


int main(){

    int* myInt= new int(1998);
    double* myDouble= new double(3.14);
    double* myDoubleArray= new double[2]{1.1,1.2};

    MyClass* myClass= new MyClass;
    MyClass2* myClass2= new MyClass2;

    delete myDouble;
    delete [] myDoubleArray;
    delete myClass;
    delete myClass2;

  getInfo();

}
```

In the header file `myNew.hpp`, we created two static variables `alloc` and `dealloc` (line 10 and 11). They keep track of how often we have used the overloaded operator new (line 13) and operator delete (line 18). In the functions, we delegate the memory allocation to `std::malloc` and the memory deallocation to `std::free`. The function `getInfo` (lines 23 - 31) provides us the numbers and displays them.

The question is as follows: have we cleaned everything properly?

Of course, not! That was the intention of this and the next lesson. Now, we know that we have leaks. Maybe it will be helpful to determine the addresses of the objects which we have forgotten to clean up.

Addresses of the Memory Leaks

So, we have to put more cleverness into the header `myNew2.hpp`.

main.cpp

myNew.hpp

myNew2.hpp

myNew3.hpp

```
// myNew2.hpp

#ifndef MY_NEW2
#define MY_NEW2

#include <algorithm>
#include <cstdlib>
#include <iostream>
#include <new>
#include <string>
#include <array>

int const MY_SIZE= 10;

std::array<void* ,MY_SIZE> myAlloc{nullptr,};

void* operator new(std::size_t sz){
    static int counter{};
    void* ptr= std::malloc(sz);
    myAlloc.at(counter++)= ptr;
    return ptr;
}

void operator delete(void* ptr) noexcept{
    auto ind= std::distance(myAlloc.begin(),std::find(myAlloc.begin(),myAlloc.end(),ptr));
    myAlloc[ind]= nullptr;
    std::free(ptr);
}

void getInfo(){

    std::cout << std::endl;

    std::cout << "Not deallocated: " << std::endl;
    for (auto i: myAlloc){
        if (i != nullptr ) std::cout << " " << i << std::endl;
    }

    std::cout << std::endl;

}

#endif // MY_NEW2
```

Therefore, we must be clever about the header `myNew2.hpp`.

The key idea is to use the `static array myAlloc` (line 15) to track the addresses of all `std::malloc` (line 19) and `std::free` (line 27) invocations. On the function `operator new`, we cannot use a container that needs dynamic memory. This container would invoke the `operator new` — a recursion that would cause the program to crash. Therefore, we used an std::array in line 15, since `std::array` gets its memory at compile time. `std::array` can become too small in this process. Therefore, we invoke `myAlloc.at(counter++)` in order to check the array boundaries.

Which memory address we have forgotten to release? The output gives the answer.

A simple search for the object having the address is the best approach since it is probable that a new call of `std::malloc` reuses an already-used address. This suffices, so long as the objects have been deleted in the meantime. But why are the addresses parts of the solution? We must only compare the memory address of the created objects with the memory address of the objects that are not yet deleted.

Comparison of the Memory Addresses

In addition to the memory address, we also have the size of the reserved memory at our disposal. We will use this information in `operator new`.

main.cpp

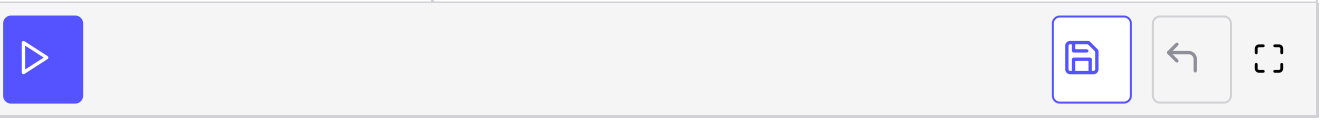
myNew.hpp

myNew2.hpp

myNew3.hpp

```
// overloadOperatorNewAndDelete.cpp

//#include "myNew.hpp"
//#include "myNew2.hpp"
#include "myNew3.hpp"

#include <iostream>
#include <string>

class MyClass{
  float* p= new float[100];
};

class MyClass2{
  int five= 5;
  std::string s= "hello";
};


int main(){
```

```
int* myInt= new int(1998);
double* myDouble= new double(3.14);
double* myDoubleArray= new double[2]{1.1,1.2};

MyClass* myClass= new MyClass;
MyClass2* myClass2= new MyClass2;

delete myDouble;
delete [] myDoubleArray;
delete myClass;
delete myClass2;

getInfo();

}
```



The allocation and deallocation of the application are clearly more transparent.

A simple comparison shows that we forgot to release an object with 4 bytes and an object with 400 bytes. In addition, the sequence of allocation in the source code corresponds to the sequence of outputs in the program. Thus, it should be easy to identify the missing memory release

---

The program has two main issues. Firstly, we statically allocate the memory for `std::array`. Secondly, we want to know which object was not released. In the next lesson, we will solve both issues.