Overloading new and delete

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

WE'LL COVER THE FOLLOWING

Quite often, a C++ application allocates memory but does not deallocate it. This is a job for the `operator new` and `operator delete`. Thanks to both of them, we can explicitly manage the memory of an application.

From time to time, we have to verify that an application correctly releases 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 memory during program shutdown is not an option.

The baseline

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

main.cpp

myNew.hpp

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

myNew2.hpp

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, does there need to be a corresponding `delete` for each `new` call?

The question can be simply answered by overloading the global operators, `new` and `delete`. For each operator, I count how often it was called.

`new` operator

C++ offers the `new` operator 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 an `std::bad_alloc` exception if they can not provide the necessary memory. The last two variations return a null pointer. It's quite convenient and sufficient to overload only version **1** because 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. We can read the details of the global operator, `new`, here.

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

`delete` operator

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

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

In the last two last versions of `delete`, we have the length of the memory block in the variable `sz` at our disposal. Read the details here.

Counting allocations and deallocations

Let's use the header `myNew.hpp` (line 1). Here we invoke the function `getInfo` to get information about our memory management.

main.cpp
myNew.hpp
myNew2.hpp
myNew3.hpp

```
// myNew.hpp

#ifndef MY_NEW

#define MY_NEW

#include <cstdlib>
#include <iostream>
#include <new>

static std::size_t alloc{0};
static std::size_t dealloc{0};

void* operator new(std::size_t sz){
    alloc+= 1;
    return std::malloc(sz);
}

void operator delete(void* ptr) noexcept{
    dealloc+= 1;
    std::free(ptr);
}

void getInfo(){

    std::cout << std::endl;

    std::cout << "Number of allocations: " << alloc << std::endl;
    std::cout << "Number of deallocations: " << dealloc << std::endl;

    std::cout << std::endl;
}

#endif // MY_NEW
```

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

The question is, have we cleaned everything?

Of course not. That's the aim of this lesson and the next one. 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

Naturally, we have to be cleverer with the header `myNew2.hpp`.

main.cpp
myNew.hpp
myNew2.hpp
myNew3.hpp

```cpp
// 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
```

The key idea is to use the `static array myAlloc` (line 15) to keep track of the addresses of all `std::malloc` (line 19) and `std::free` (line 27) invocations. In the function, `operator new`, we cannot use a container that needs dynamic memory. This container would invoke the `operator new`, and use recursion, which would cause the program to crash. Now what can happen, is that our `std::array` becomes too small. Therefore, we invoke `myAlloc.at(counter++)` to check the array boundaries.

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

A simple search for the object that has the address is not a good idea because it is quite probable that a new call of `std::malloc` reuses an old address. That is fine as long as the objects have been deleted in the meantime.

But why are the addresses part of the solution? We only have to compare the memory addresses of the created objects with the memory addresses of the objects that are not deleted in order to ensure that we have properly deallocated our memory.

Comparison of the memory addresses

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

main.cpp

myNew.hpp

myNew2.hpp

myNew3.hpp

```
// myNew3.hpp

#ifndef MY_NEW3
#define MY_NEW3

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

int const MY_SIZE= 10;
```

```
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;
    std::cerr << "Addr.: " << ptr << " size: " << sz << std::endl;
    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_NEW3
```

Now, the allocation and deallocation of the application are 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. Now, it should be quite easy to identify the missing memory releases.

Further information

- global operator, `new`
- operator_new

The program is not beautiful for two reasons. First, we statically allocated the memory for `std::array`. Second, we want to know which object was not released. In the next lesson, we will solve both issues.