Memory Management: Overloading Operator new and delete 2

In this lesson, we will refine the strategy for overloading operators new and delete.

WE'LL COVER THE FOLLOWING

What were the not-so-nice properties of the previous lesson?

Firstly, we only get a hint of which memory was lost. Secondly, we had to prepare the whole bookkeeping of memory management at compile time. In this lesson, we aim to overcome these shortcomings.

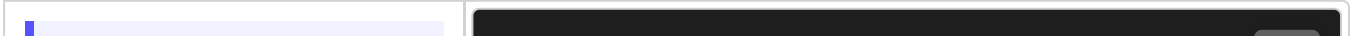
Who is the Bad Guy?

Special tasks call for special strategies. We must use a small macro for debugging purposes.

Let's take a look at this macro. `#define new new(__FILE__, __LINE__)`

The macro causes each `new` call to be mapped onto the overloaded new call. This overloaded `new` call also receives the name of the file and the line number respectively. That is exactly the information we need to solve this problem.

So, what will happen if we use the macro in line 6?



main.cpp

myNew4.hpp
myNew5.hpp

```
// overloadOperatorNewAndDelete2.cpp

//#include "myNew4.hpp"

//#include "myNew5.hpp"

#define new new(__FILE__, __LINE__)

#include <iostream>
#include <new>
#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;

    dummyFunction();

    //getInfo();

}
```




The preprocessor substitutes all `new` calls, showing exactly the modified main function.

modified_main

```
class MyClass{
  float* p= new("main.cpp", 14) float[100];
};

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

```
int main(){


    int* myInt= new("main.cpp", 24) int(1998);
    double* myDouble= new("main.cpp", 25) double(3.14);
    double* myDoubleArray= new("main.cpp", 26) double[2]{1.1,1.2};

    MyClass* myClass= new("main.cpp", 28) MyClass;
    MyClass2* myClass2= new("main.cpp", 29) MyClass2;

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

    dummyFunction();

    getInfo();

}
```

Lines 2 and 12 demonstrate how the preprocessor substitutes the constants `__FILE__` and `__LINE__` in the macro. So, how does this technique work? The header `myNew4.hpp` solves the problem.

First Try

main.cpp

myNew4.hpp

myNew5.hpp

```
// myNew4.hpp

#ifndef MY_NEW4
#define MY_NEW4

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

int const MY_SIZE= 10;

int counter= 0;

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

void* newImpl(std::size_t sz,char const* file, int line){
    void* ptr= std::malloc(sz);
    std::cerr << file << ": " << line << " " <<  ptr << std::endl;
    myAlloc.at(counter++)= ptr;
```

```
    return ptr;
}


void* operator new(std::size_t sz,char const* file, int line){
    return newImpl(sz,file,line);
}

void* operator new [](std::size_t sz,char const* file, int line){
    return newImpl(sz,file,line);
}

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);
}

#define new new(__FILE__, __LINE__)

void dummyFunction(){
    int* dummy= new int;
}

void getInfo(){

    std::cout << std::endl;

    std::cout << "Allocation: " << std::endl;
    for (auto i: myAlloc){
        if (i != nullptr ) std::cout << " " << i << std::endl;
    }

    std::cout << std::endl;

}

#endif // MY_NEW4
```




Theory

In lines 25 and 29, we implement special operators `new` and `new[]` to delegate their functionality to the helper function `newImpl` (line 18 - 23). The function completes two important jobs:

1. It displays the name of the source file and the line number (line 20) to each `new` call of the function.

2. In the static array `myAlloc`, it keeps track of each used memory address (line 21).

This fits the behavior of the overloaded operator `delete` which sets all

This fits the behavior of the overloaded operator `delete` which sets all memory addresses to the null pointer `nullptr` (line 35). The memory addresses stand for the deallocated memory areas. In the end, the function `getInfo` displays the memory addresses that were not deallocated. We can directly see them together with the file name and line number.

Of course, we can also directly apply the macro in the file `myNew4.hpp`. Now that we've gone over the theory, what is the output of the program?

Understanding the Output

The memory areas to three memory addresses were not deallocated. The problems, therefore, are

- `new` calls in line 23 and line 13 in `main.cpp` and
- the `new` call in line 42 in `myNew4.hpp`

Impressive, isn't it? It's important to note that the presented technique has two significant drawbacks.

1. We must overload the simple operator `new` and the operator `new []` for arrays, due to the fact that the overloaded operator `new` is not a fallback for the three remaining operators `new`.
2. We cannot use the special operator `new` that returns a null pointer in the error case since it will be explicitly called by the operator new with the argument `std::nothrow: int* myInt= new (std::nothrow) int(1998);`

Now, we must solve the first issue. We must use a data structure for the array `myAlloc` that manages its memory at run time. Therefore, it is no longer necessary to eagerly allocate the memory at compile time.

All at Run Time

Why did we not not allocate memory in the `operator new`? The `operator new` was globally overloaded. Therefore, a call of `new` would end in never-ending recursion. That will occur if we use a container such as `std::vector` which dynamically allocates its memory.

This restriction no longer holds since we did not overload the global `operator`

`new`, which is a fallback for the three remaining `new` operators. Due to the

macro, our own variant of the operator `new` is now used. Therefore, we can use `std::vector` in our `operator new`.

You can see this operation in the program below while using the header `myNew5.hpp`

main.cpp

myNew4.hpp

myNew5.hpp

```
// myNew5.hpp

#ifndef MY_NEW5
#define MY_NEW5

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

std::vector<void*> myAlloc;

void* newImpl(std::size_t sz,char const* file, int line){
    static int counter{};
    void* ptr= std::malloc(sz);
    std::cerr << file << ": " << line << " " <<  ptr << std::endl;
    myAlloc.push_back(ptr);
    return ptr;
}

void* operator new(std::size_t sz,char const* file, int line){
    return newImpl(sz,file,line);
}

void* operator new [](std::size_t sz,char const* file, int line){
    return newImpl(sz,file,line);
}

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);
}

#define new new(__FILE__, __LINE__)

void dummyFunction(){
    int* dummy= new int;
}
```

```
void getInfo(){

    std::cout << std::endl;

    std::cout << "Allocation: " << std::endl;
    for (auto i: myAlloc){
        if (i != nullptr ) std::cout << " " << i << std::endl;
    }

    std::cout << std::endl;

}

#endif // MY_NEW5
```




In lines 13, 19, and 33, we use `std::vector`.

Let’s take a look at an example of memory management in the next lesson.