Overloading new and delete: A Few Adjustments

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

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

First, we only got a hint of which memory was lost. Second, we had to prepare the whole bookkeeping process 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 forces. Here, we can benefit from the use of a small macro for debugging purposes.

Let's have a look at the macro `#define new new(__FILE__, __LINE__)`

The macro causes each `new` call to be mapped to the overloaded `new` call. This overloaded `new` call gets, in addition, the name of the file and the line number respectively. That's exactly the information we need.

But what will happen if we use the macro in line 4 below?

main.cpp

myNew4.hpp

```
//#include "myNew4.hpp"
//#include "myNew5.hpp"

#define new new(__FILE__, __LINE__)
```

myNew5.hpp

```
#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. That shows the modified main function exactly.

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;
```

```
    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 show that the preprocessor substitutes the constants `__FILE__` and `__LINE__` in the macro. But how does the magic work? The header `myNew4.hpp` solves the riddle.

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){
```

```
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

We implement, in lines 25 and 28 in `main.cpp`, the special operators `new` and `new[]` that delegate their functionality to the helper function `newImpl` (lines 18 - 23 in `myNew4.hpp`). The function performs two important jobs:

1. It displays, to each `new` call, the name of the source file and the line number (line 20 in `myNew4.hpp`).

2. In the static array `myAlloc`, it keeps track of each used memory address (line 21 in `myNew4.hpp`).

This fits the behavior of the overloaded `delete` operator that sets all memory addresses to the null pointer, `nullptr` (line 35 in `myNew4.hpp`). 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 directly apply the macro in the file `myNew4.hpp` as well. That was a lot of theory. What's the output of the program?

Understanding the output

The memory areas of three memory addresses were not deallocated. The bad guys are the

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

Impressive, isn't it? But the presented technique has two drawbacks: one minor and one major.

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

Now, we have to solve the first issue. We want to use a data structure for the array `myAlloc` that manages its memory at run time. Therefore, it is not necessary anymore to eagerly allocate the memory at compile time.

All at runtime

For what reason could we not allocate memory in the `operator new`? It was globally overloaded. Therefore, a call of `new` would face never-ending recursion. That is exactly what will happen if we use a container like `std::vector` that dynamically allocates its memory.

This restriction does not hold anymore because we didn't overload the global `operator new` that is a fallback to the three remaining `new` operators. Thanks to the macro, our own variant of the `new` operator is now used. Therefore, we can use `std::vector` in our `operator new`.

We can see that in the program below while using the `myNew5.hpp` header:

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 have a short exercise to test the concept of memory in C++.