Memory Allocation

In this lesson, we will learn about an important subsection of memory management, i.e., memory allocation.

WE'LL COVER THE FOLLOWING

Introduction

Explicit memory management in C++ has a high complexity but also provides us with great functionality. Sadly, this special domain of C++ is not so well known.

For example, we can directly create objects in static memory, in a reserved area, or even in a memory pool. This functionality is often key in safety-critical applications in the embedded world.

- C++ enables the dynamic allocation and deallocation of memory.
- Dynamic memory, or the heap, has to be explicitly requested and released by the programmer.
- We can use the operators `new` and `new[]` to allocate memory and the operators `delete` and `delete[]` to deallocate memory.

- The compiler manages its memory automatically on the stack.

> 💡 Smart pointers manage memory automatically.

Memory allocation

`new`

Thanks to the `new` operator, we can dynamically allocate memory for the instance of a type.

```
int* i = new int;
double* d = new double(10.0);
Point* p = new Point(1.0, 2.0);
```

- `new` causes memory allocation and object initialization.
- The arguments in the brackets go directly to the constructor.
- `new` returns a pointer to the corresponding object.
- If the class of dynamically created objects is part of a type hierarchy, more constructors are invoked.

`new[]`

`new[]` allows us to allocate memory to a C array. The newly created objects need a default constructor.

```
double* d = new double[5];
Point* p = new Point[10];
```

- The class of the allocated object must have a default constructor.
- The default constructor will be invoked for each element of the C array.

> 💡 The STL Containers and the C++ String automatically manage their memory.

Placement `new`

Placement `new` is often used to instantiate an object or a C array in a specific area of memory. In addition, we can overload placement `new` globally or for our own data types. This is a big benefit offered by C++.

```
char* memory = new char[sizeof(Account)]; // allocate std::size_t
Account* acc = new(memory) Account; // instantiate acc in memory
```

- The header, `<new>`, is necessary.
- Can be overloaded on a class basis or globally.

Typical use-cases

- Explicit memory allocation
- Avoidance of exceptions
- Debugging

Failed allocation

If the memory allocation operation fails, `new` and `new[]` will raise a `std::bad_alloc` exception. But that is not the behavior we want. Therefore, we can invoke placement `new` with the constant `std::nothrow`. This call will return a `nullptr` in the case of failure.

```
char* c = new(std::nothrow) char[10];
if (c){
  delete c;
}
else{
// an error occured
}
```

New handler

In the case of a failed allocation, we can use `std::set_new_handler` with our own handler. `std::set_new_handler` returns the older handler and needs a callable unit. A callable unit is typically a function, a function object, or a lambda-function. The callable unit should take no arguments and return

nothing. We can get the handler currently being used by invoking the function `std::get_new_handler`.

Our own handler allows us to implement special strategies for failed allocations:

- request more memory
- terminate the program with `std::terminate`
- throw an exception of type `std::bad_alloc`

Further information

- Smart pointers

In the next lesson, we will study how to deallocate memory.