Memory Management: Memory Allocation

In this lesson, we will learn about a subsection of memory management - memory allocation.

WE'LL COVER THE FOLLOWING

Introduction

Explicit memory management in C++ is highly complex, but it also provides us with great functionality. Unfortunately, this special domain is not as common in C++. For example, you can directly create objects in static memory, in a reserved area, or even in a memory pool. This functionality is often key in the safety-critical applications of the embedded world.

- C++ enables the dynamic allocation and deallocation of memory.
- Dynamic memory (heap) must be explicitly requested and released by the programmer.
- You can use the operators `new` and `new[]` to allocate memory and the operators `delete` and `delete[]` to deallocate the memory.
- The compiler manages its memory automatically on the stack.

> 💡 Smart pointers manages the memory automatically.

Memory Allocation

new

Due to the operator `new`, you 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 the memory allocation and the object initialization.
- The arguments in the brackets go directly to the constructor.
- `new` returns the address of the object that has been given memory.
- If the class of dynamically created objects is part of a type hierarchy, more constructors are invoked.

new[ ]

`new[]` creates a C array of objects. 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.

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

Placement new

Placement new is often used to instantiate an object in a pre-reserved memory area. In addition, you can overload placement new globally or for your own data types. This is a big benefit that C++ offers.

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

- The header `<new>` is necessary
- Placement new can be overload on a class basis or global

Typical use-cases

- Explicit memory allocation
- Avoidance of exceptions
- Debugging

Failed Allocation

If the memory allocation fails, `new` and `new[]` will raise a `std::bad_alloc` exception. This is not the desired response. Rather, you can invoke placement new with the constant `std::nothrow`. This call generates a `nullptr` in the error case

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

New Handler

In the case of a failed allocation, you can use `std::set_new_handler` with your 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 argument and return nothing. We can get the currently-used handler by invoking the function `std::get_new_handler`.

Your own handler enables you to implement special strategies for failed allocations:

- request more memory

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

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