All the flavours of operator new and delete

Some ways to allocate and initialize objects in C++

Mateusz Nowak

16.09.2020

4Developers

Who am I

Team Manager

Located in Gdańsk, Poland

Who am I

Team Manager

Located in Gdańsk, Poland

Slides writing machine

cppcast listener

/r/cpp reader

Who am I

cppcast listener

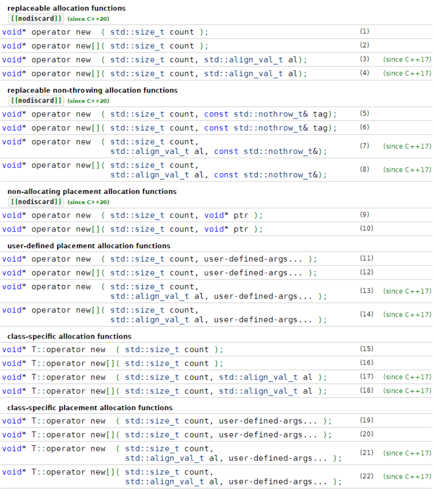
All views expressed herein are those of my own and do not represent the opinions of any entity whatsoever with which I have been, am now, or will be affiliated.

Table of contents

1. Flavours of operator new and delete
2. Usual allocation and deallocation functions
3. Placement allocation and deallocation functions
4. Overriding new and delete
5. Allocating with smart pointers
6. Aligned allocation and deallocation functions

Flavours of operator new and delete

Flavours of operator new



replaceable allocation functions
`[[nodiscard]]` (since C++20)

```
void* operator new  ( std::size_t count );                                  (1)
void* operator new[]( std::size_t count );                                  (2)
void* operator new  ( std::size_t count, std::align_val_t al);             (3)   (since C++17)
void* operator new[]( std::size_t count, std::align_val_t al);             (4)   (since C++17)
```

replaceable non-throwing allocation functions
`[[nodiscard]]` (since C++20)

```
void* operator new  ( std::size_t count, const std::nothrow_t& tag);       (5)
void* operator new[]( std::size_t count, const std::nothrow_t& tag);       (6)
void* operator new  ( std::size_t count,
                      std::align_val_t al, const std::nothrow_t&);         (7)   (since C++17)
void* operator new[]( std::size_t count,
                      std::align_val_t al, const std::nothrow_t&);         (8)   (since C++17)
```

non-allocating placement allocation functions
`[[nodiscard]]` (since C++20)

```
void* operator new  ( std::size_t count, void* ptr );                       (9)
void* operator new[]( std::size_t count, void* ptr );                       (10)
```

user-defined placement allocation functions

```
void* operator new  ( std::size_t count, user-defined-args... );            (11)
void* operator new[]( std::size_t count, user-defined-args... );            (12)
void* operator new  ( std::size_t count,
                      std::align_val_t al, user-defined-args... );          (13)  (since C++17)
void* operator new[]( std::size_t count,
                      std::align_val_t al, user-defined-args... );          (14)  (since C++17)
```

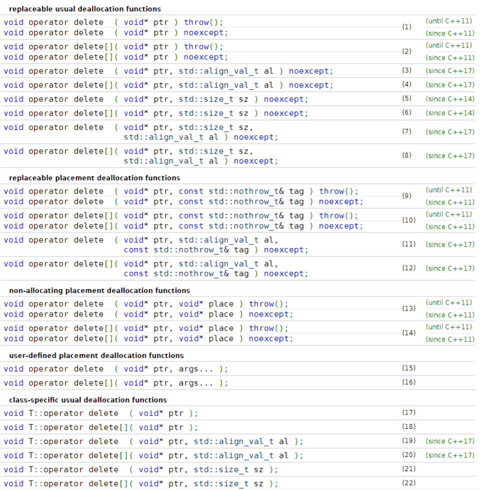
class-specific allocation functions

```
void* T::operator new  ( std::size_t count );                               (15)
void* T::operator new[]( std::size_t count );                               (16)
void* T::operator new  ( std::size_t count, std::align_val_t al );          (17)  (since C++17)
void* T::operator new[]( std::size_t count, std::align_val_t al );          (18)  (since C++17)
```

class-specific placement allocation functions

```
void* T::operator new  ( std::size_t count, user-defined-args... );         (19)
void* T::operator new[]( std::size_t count, user-defined-args... );         (20)
void* T::operator new  ( std::size_t count,
                         std::align_val_t al, user-defined-args... );       (21)  (since C++17)
void* T::operator new[]( std::size_t count,
                         std::align_val_t al, user-defined-args... );       (22)  (since C++17)
```

Flavours of operator delete



replaceable usual deallocation functions		

`void operator delete  ( void* ptr ) throw();`
`void operator delete  ( void* ptr ) noexcept;`
	(1)	(until C++11)
		(since C++11)

`void operator delete[]( void* ptr ) throw();`
`void operator delete[]( void* ptr ) noexcept;`
	(2)	(until C++11)
		(since C++11)

`void operator delete  ( void* ptr, std::align_val_t al ) noexcept;`	(3)	(since C++17)
`void operator delete[]( void* ptr, std::align_val_t al ) noexcept;`	(4)	(since C++17)
`void operator delete  ( void* ptr, std::size_t sz ) noexcept;`	(5)	(since C++14)
`void operator delete[]( void* ptr, std::size_t sz ) noexcept;`	(6)	(since C++14)

`void operator delete  ( void* ptr, std::size_t sz,`
`std::align_val_t al ) noexcept;`
	(7)	(since C++17)

`void operator delete[]( void* ptr, std::size_t sz,`
`std::align_val_t al ) noexcept;`
	(8)	(since C++17)

replaceable placement deallocation functions		

`void operator delete  ( void* ptr, const std::nothrow_t& tag ) throw();`
`void operator delete  ( void* ptr, const std::nothrow_t& tag ) noexcept;`
	(9)	(until C++11)
		(since C++11)

`void operator delete[]( void* ptr, const std::nothrow_t& tag ) throw();`
`void operator delete[]( void* ptr, const std::nothrow_t& tag ) noexcept;`
	(10)	(until C++11)
		(since C++11)

`void operator delete  ( void* ptr, std::align_val_t al,`
`const std::nothrow_t& tag ) noexcept;`
	(11)	(since C++17)

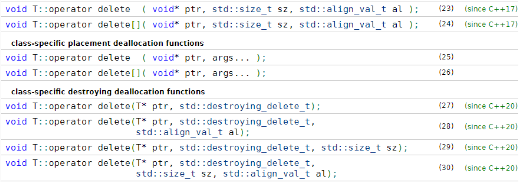
`void operator delete[]( void* ptr, std::align_val_t al,`
`const std::nothrow_t& tag ) noexcept;`
	(12)	(since C++17)

non-allocating placement deallocation functions		

`void operator delete  ( void* ptr, void* place ) throw();`
`void operator delete  ( void* ptr, void* place ) noexcept;`
	(13)	(until C++11)
		(since C++11)

`void operator delete[]( void* ptr, void* place ) throw();`
`void operator delete[]( void* ptr, void* place ) noexcept;`
	(14)	(until C++11)
		(since C++11)

user-defined placement deallocation functions		
`void operator delete  ( void* ptr, args... );`	(15)	
`void operator delete[]( void* ptr, args... );`	(16)	
class-specific usual deallocation functions		
`void T::operator delete  ( void* ptr );`	(17)	
`void T::operator delete[]( void* ptr );`	(18)	
`void T::operator delete  ( void* ptr, std::align_val_t al );`	(19)	(since C++17)
`void T::operator delete[]( void* ptr, std::align_val_t al );`	(20)	(since C++17)
`void T::operator delete  ( void* ptr, std::size_t sz );`	(21)	
`void T::operator delete[]( void* ptr, std::size_t sz );`	(22)	



`void T::operator delete  ( void* ptr, std::size_t sz, std::align_val_t al );`	(23)	(since C++17)
`void T::operator delete[]( void* ptr, std::size_t sz, std::align_val_t al );`	(24)	(since C++17)
class-specific placement deallocation functions		
`void T::operator delete  ( void* ptr, args... );`	(25)	
`void T::operator delete[]( void* ptr, args... );`	(26)	
class-specific destroying deallocation functions		
`void T::operator delete(T* ptr, std::destroying_delete_t);`	(27)	(since C++20)

`void T::operator delete(T* ptr, std::destroying_delete_t,`
`std::align_val_t al);`
	(28)	(since C++20)

`void T::operator delete(T* ptr, std::destroying_delete_t, std::size_t sz);`	(29)	(since C++20)

`void T::operator delete(T* ptr, std::destroying_delete_t,`
`std::size_t sz, std::align_val_t al);`
	(30)	(since C++20)

Usual allocation and deallocation functions

Usual allocation functions

```
// Throwing functions
[[nodiscard]] void* operator new(std::size_t count);
[[nodiscard]] void* operator new[](std::size_t count);
// Non-throwing
[[nodiscard]] void* operator new(std::size_t count, const std::nothrow_t& tag) noexcept;
[[nodiscard]] void* operator new[](std::size_t count, const std::nothrow_t& tag) noexcept;
// Class-specific
void* T::operator new(std::size_t count);
void* T::operator new[](std::size_t count);
```

Usual deallocation functions

```
// Usual
void operator delete(void* ptr) noexcept;
void operator delete[](void* ptr) noexcept;
void operator delete(void* ptr, std::size_t sz) noexcept;
void operator delete[](void* ptr, std::size_t sz) noexcept;
// Class-specific
void T::operator delete(void* ptr);
void T::operator delete[](void* ptr);
void T::operator delete(void* ptr, std::size_t sz);
void T::operator delete[](void* ptr, std::size_t sz);
// Class-specific destroying deallocation
void T::operator delete(T* ptr, std::destroying_delete_t);
void T::operator delete(T* ptr, std::destroying_delete_t, std::size_t sz);
```

Usual allocation example

```
#include <new>
#include <string>

struct t {
  int a;
  std::string b;
};

auto v1 = new t{1, "asdqwe"};  // Calls operator new(unsigned long) and initialize fields
auto v2 = new (std::nothrow) int[20];  // Calls operator new[](unsigned long,
                                       //                     std::nothrow_t const&)
delete v1;    // Calls t::~t() and operator delete(void*)
delete[] v2;  // Calls operator delete[](void*)
```

Default and value initialization

Default and value initialization

Default initialization

Default and value initialization

Default initialization

- Performed when variable is constructed with no initializer

Default and value initialization

- Default initialization
  - Performed when variable is constructed with no initializer
  - If T is class type default constructor is called

Default and value initialization

Default initialization

- Performed when variable is constructed with no initializer
- If T is class type default constructor is called
- If T is array, every array element is default constructed

Default and value initialization

Default initialization

- Performed when variable is constructed with no initializer
- If T is class type default constructor is called
- If T is array, every array element is default constructed
- Otherwise, nothing is done

Default and value initialization

- Default initialization
  - Performed when variable is constructed with no initializer
  - If T is class type default constructor is called
  - If T is array, every array element is default constructed
  - Otherwise, nothing is done
- Value initialization done when initializing value is passed

operator new value vs default initialization example

```cpp
struct test {
  int t1;
};

// Default initialization
auto* t = new test;
// Value initialization with default values
auto* t = new test{};
// Value initialization
auto* t = new test{1};
```

operator new value vs default initialization example

```cpp
struct test {
  int t1;
};

// Default initialization
auto* t = new test;
// Value initialization with default values
auto* t = new test{};
// Value initialization
auto* t = new test{1};
```

```asm
; Default initialization
mov     edi, 4
call    operator new(unsigned long)
mov     qword ptr [rbp - 8], rax
; Value initialization with default value
mov     edi, 4
call    operator new(unsigned long)
mov     rcx, rax
mov     dword ptr [rax], 0
mov     qword ptr [rbp - 16], rcx
; Value initialization
mov     edi, 4
call    operator new(unsigned long)
xor     ecx, ecx
mov     rdx, rax
mov     dword ptr [rax], 1
mov     qword ptr [rbp - 24], rdx
```

Placement allocation and deallocation functions

Placement allocation functions

```
// Standard placement
[[nodiscard]] void* operator new(std::size_t count, void* ptr) noexcept;
[[nodiscard]] void* operator new[](std::size_t count, void* ptr) noexcept;
// User-defined placement
void* operator new(std::size_t count, user_defined_args...);
void* operator new[](std::size_t count, user_defined_args...);
// Class-specific placement
void* T::operator new(std::size_t count, user_defined_args...);
void* T::operator new[](std::size_t count, user_defined_args...);
```

Placement deallocation functions

```
// Replaceable placement
void operator delete(void* ptr, const std::nothrow_t& tag) noexcept;
void operator delete[](void* ptr, const std::nothrow_t& tag) noexcept;
// Non-allocating placement
void operator delete(void* ptr, void* place) noexcept;
void operator delete[](void* ptr, void* place) noexcept;
// User-defined placement
void operator delete(void* ptr, args...);
void operator delete[](void* ptr, args...);
// Class-specific placement
void T::operator delete(void* ptr, args...);
void T::operator delete[](void* ptr, args...);
```

std::launder

Pointer optimization barrier [ptr.launder]

Prevents compiler from assuming that const object the pointer points to won’t change

std::launder example

```
struct X {
  int n;
};

const X* p = new const X{3};
const int a = p->n;
new (const_cast<X*>(p)) const X{5}; // p does not point to new object (6.7.3)
                                     // because its type is const
const int b = p->n;                  // undefined behavior
const int c = std::launder(p)->n;    // OK
```

std:: allocators

std::allocator

std::allocator_traits

Specialized <memory> algorithms (since C++20)

- std::uninitialized_default_construct
- std::uninitialized_value_construct
- std::uninitialized_fill
- std::construct_at
- std::destroy_at

Overriding new and delete

Overriding new and delete

The library provides default implementation for the global allocation and deallocation functions

Any definition of global allocation / deallocation function in C++ program replaces default one

Allocation and deallocation functions may also be declared and defined for any class

Flavours of allocation and deallocation functions change between standard versions

Placement forms can be overriden only for class

new expression lookup order

If :: is present in new expression class specific overloads are ignored

new expression lookup order

If :: is present in new expression class specific overloads are ignored

Otherwise class specific overloads are preferred over global ones

delete expression lookup order

If :: is present in delete expression, global namespace overload is selected

delete expression lookup order

If :: is present in delete expression, global namespace overload is selected

If there is destroying delete, all non-destroying are ignored

delete expression lookup order

If :: is present in delete expression, global namespace overload is selected

If there is destroying delete, all non-destroying are ignored

If alignment exceeds __STDCPP_DEFAULT_NEW_ALIGNMENT__ alignment aware override is preferred

delete expression lookup order

If :: is present in delete expression, global namespace overload is selected

If there is destroying delete, all non-destroying are ignored

If alignment exceeds __STDCPP_DEFAULT_NEW_ALIGNMENT__ alignment aware override is preferred

If there are class specific overloads, size-unaware is preferred over size-aware

delete expression lookup order

If :: is present in delete expression, global namespace overload is selected

If there is destroying delete, all non-destroying are ignored

If alignment exceeds __STDCPP_DEFAULT_NEW_ALIGNMENT__ alignment aware override is preferred

If there are class specific overloads, size-unaware is preferred over size-aware

Then, if type is complete and, for delete[] only, type has non-trivial destructor, size-aware global overload is selected

delete expression lookup order

If :: is present in delete expression, global namespace overload is selected

If there is destroying delete, all non-destroying are ignored

If alignment exceeds __STDCPP_DEFAULT_NEW_ALIGNMENT__ alignment aware override is preferred

If there are class specific overloads, size-unaware is preferred over size-aware

Then, if type is complete and, for delete[] only, type has non-trivial destructor, size-aware global overload is selected

Otherwise it is unspecified if size aware or unaware version will be called.

Application scope new delete override example

```
#include <fmt/format.h>

void* operator new(std::size_t count) {
  fmt::print(”New\n”);
  return std::malloc(count);
}
void operator delete(void* ptr) noexcept {
  fmt::print(”Delete\n”);
  std::free(ptr);
}
void operator delete[](void* ptr) noexcept {
  fmt::print(”Delete[]\n”);
  std::free(ptr);
}
int main() {
  auto* i = new int{0};
  delete i;
  auto* i_a = new int[10];
  delete[] i_a;
  return 0;
}
```

Application scope new delete override example

```
#include <fmt/format.h>

void* operator new(std::size_t count) {
  fmt::print("New\n");
  return std::malloc(count);
}
void operator delete(void* ptr) noexcept {
  fmt::print("Delete\n");
  std::free(ptr);
}
void operator delete[](void* ptr) noexcept {
  fmt::print("Delete[]\n");
  std::free(ptr);
}
int main() {
  auto* i = new int{0};
  delete i;
  auto* i_a = new int[10];
  delete[] i_a;
  return 0;
}
```

```
New
Delete
New
Delete[]
```

Class scope new delete override example

```cpp
#include <fmt/format.h>

struct t_t {
  void operator delete(t_t* ptr, std::destroying_delete_t) {
    fmt::print("Destroying Delete\n");
    ptr->~t_t();
    std::free(ptr);
  }
  void operator delete(void* ptr) {
    fmt::print("Delete\n");
    std::free(ptr);
  }
};

int main() {
  auto* t = new t_t;
  delete t;
}
```

Class scope new delete override example

```cpp
#include <fmt/format.h>

struct t_t {
  void operator delete(t_t* ptr, std::destroying_delete_t) {
    fmt::print("Destroying Delete\n");
    ptr->~t_t();
    std::free(ptr);
  }
  void operator delete(void* ptr) {
    fmt::print("Delete\n");
    std::free(ptr);
  }
};

int main() {
  auto* t = new t_t;
  delete t;
}
```

```
Destroying Delete
```

Override placement allocation example

```
#include <fmt/format.h>
#include <array>
#include <cstdlib>
struct T { std::array<uint8_t, 128> v; };
// Cannot override void * operator new(std::size_t, void *)
void* operator new(std::size_t count, int* ptr) noexcept {
  fmt::print("Version␣1\n");
  return ptr;
}
void* operator new(std::size_t count, T* ptr) noexcept {
  fmt::print("Version␣2\n");
  return ptr;
}
int main() {
  int t1[128];
  new (t1) int;
  T t2;
  new (&t2) float;
  new ((void*)t1) int;
}
```

Override placement allocation example

```cpp
#include <fmt/format.h>
#include <array>
#include <cstdlib>
struct T { std::array<uint8_t, 128> v; };
// Cannot override void * operator new(std::size_t, void *)
void* operator new(std::size_t count, int* ptr) noexcept {
  fmt::print("Version 1\n");
  return ptr;
}
void* operator new(std::size_t count, T* ptr) noexcept {
  fmt::print("Version 2\n");
  return ptr;
}
int main() {
  int t1[128];
  new (t1) int;
  T t2;
  new (&t2) float;
  new ((void*)t1) int;
}
```

```
Version 1
Version 2
```

Allocating with smart pointers

Smart pointers

Explicitly showing ownership and lifetime

Automatically deleted when all owners go out of scope

std::unique_ptr - single owner, lives as long as owner scope

std::shared_ptr - multiple owners, lives as long as all owners scope

std::weak_ptr - non owning, reference to std::shared_ptr with validity check

std::make_unique and std::make_shared

std::make_unique<T>(args...)

std::make_unique<T[]>(size)

std::make_shared<T>(args...)

std::allocate_shared<T>(alloc, args...)

std::make_unique and std::make_shared (added in C++20)

```
std::make_shared<T[]>(size)
std::make_shared<T[N]>()
std::make_shared<T[]>(size, T val)
std::make_shared<T[N]>(T val)
std::make_unique_for_overwrite<T>()
std::make_unique_for_overwrite<T[]>(size)
std::make_shared_for_overwrite<T>()
std::make_shared_for_overwrite<T[]>(size)
std::allocate_shared<T[]>(alloc, size)
std::allocate_shared<T[]>(alloc)
std::allocate_shared<T[]>(alloc, size, T val)
std::allocate_shared<T[N]>(alloc, T val)
std::allocate_shared_for_overwrite<T>(alloc)
std::allocate_shared_for_overwrite<T[]>(alloc, size)
```

Aligned allocation and deallocation functions

Memory accesses and alignment

Data can be read and write from memory in different size chunks

- Depending on architecture / memory type
- Memory accessess are always word size aligned
  - Shorter accesses are translated to longer ones under the hood
    - E.g. *first / last byte enable* in PCI TLBs
- Currently word size is almost never less then 4 bytes, and always power of 2

Aligned allocation functions

```
[[nodiscard]] void *operator new(std::size_t count, std::align_val_t al);
[[nodiscard]] void *operator new[](std::size_t count, std::align_val_t al);
// Non-throwing
[[nodiscard]] void *operator new(std::size_t count, std::align_val_t al,
                                 const std::nothrow_t &) noexcept;
[[nodiscard]] void *operator new[](std::size_t count, std::align_val_t al,
                                   const std::nothrow_t &) noexcept;
// Placement
[[nodiscard]] void *operator new(std::size_t count, std::align_val_t al, user_defined_args...);
void *operator new[](std::size_t count, std::align_val_t al, user_defined_args...);
// Class-specific
[[nodiscard]] void *T::operator new(std::size_t count, std::align_val_t al);
[[nodiscard]] void *T::operator new[](std::size_t count, std::align_val_t al);
// Class-specific placement
[[nodiscard]] void *T::operator new(std::size_t count, std::align_val_t al,
                                    user_defined_args...);
[[nodiscard]] void *T::operator new[](std::size_t count, std::align_val_t al,
                                      user_defined_args...);
```

Aligned deallocation functions

```
void operator delete(void *ptr, std::align_val_t al) noexcept;
void operator delete[](void *ptr, std::align_val_t al) noexcept;
void operator delete(void *ptr, std::size_t sz, std::align_val_t al) noexcept;
void operator delete[](void *ptr, std::size_t sz, std::align_val_t al) noexcept;
// Placement deallocation
void operator delete(void *ptr, std::align_val_t al, const std::nothrow_t &tag) noexcept;
void operator delete[](void *ptr, std::align_val_t al, const std::nothrow_t &tag) noexcept;
// Class-specific
void T::operator delete(void *ptr, std::size_t sz, std::align_val_t al);
void T::operator delete[](void *ptr, std::size_t sz, std::align_val_t al);
// Class-specific destroying
void T::operator delete(T *ptr, std::destroying_delete_t, std::align_val_t al);
void T::operator delete(T *ptr, std::destroying_delete_t, std::size_t sz, std::align_val_t al);
```

Allocating with alignment example (1)

```
struct ex {
  void *operator new(std::size_t c) {
    fmt::print("New␣example␣{}\n", c);
    return ::new ex;
  }
  void *operator new(std::size_t c, std::align_val_t al) {
    fmt::print("New␣aligned␣example␣{}␣{}\n", c, al);
    return ::new (al) ex;
  }
  void operator delete(void *ptr) {
    fmt::print("Delete␣example␣{}\n", fmt::ptr(ptr));
    ::operator delete(ptr);
  }
  void operator delete(void *ptr, std::align_val_t al) {
    fmt::print("Delete␣aligned␣example␣{}␣{}\n", fmt::ptr(ptr), al);
    ::operator delete(ptr, al);
  }
};
```

Allocating with alignment example (2)

```
auto *x1 = new ex;
delete x1;
```

Allocating with alignment example (2)

```
auto *x1 = new ex;
delete x1;
```

```
New example 1
Delete example 0x1eb3e80
```

Allocating with alignment example (3)

```
auto *x2 = new (std::align_val_t{128}) ex;
delete x2;
```

Allocating with alignment example (3)

```cpp
auto *x2 = new (std::align_val_t{128}) ex;
delete x2;
```

```
New aligned example 1 128
Delete example 0x1eb3f00
```

Allocating with alignment example (4)

```
auto *x3 = new (std::align_val_t{128}) ex;
operator delete(x3);
```

Allocating with alignment example (4)

```
auto *x3 = new (std::align_val_t{128}) ex;
operator delete(x3);
```

```
New aligned example 1 128
```

Allocating with alignment example (5)

```
auto *x4 = new (std::align_val_t{128}) ex;
operator delete (x4, std::align_val_t{128});
```

Allocating with alignment example (5)

```
auto *x4 = new (std::align_val_t{128}) ex;
operator delete (x4, std::align_val_t{128});
```

```
New aligned example 1 128
```

Allocating with alignment example (6)

```cpp
auto *x5 = new (std::align_val_t{128}) ex;
x5->operator delete (x5, std::align_val_t{128});
```

Allocating with alignment example (6)

```
auto *x5 = new (std::align_val_t{128}) ex;
x5->operator delete (x5, std::align_val_t{128});
```

```
New aligned example 1 128
Delete aligned example 0x1eb4280 128
```

When you hit "Error: memory is not aligned." - Mateusz Nowak - code::dive 2018

Allocating with pmr::

Another story. . .

Thanks for your attention!

Mateusz Nowak

https://www.linkedin.com/in/mateusz-nowak-qq/

https://github.com/noqqaqq

@noqqaqq

Feedback is appreciated ☺