Overloaded and Customized new/delete

GKxx

July 31, 2022

`new` Expressions

The execution of a `new` expression takes two steps:

1. Allocate a block of memory.
2. Construct the object(s) on the allocated memory.

What we can control is the first step.

operator new

Memory allocation is done by a group of functions:

```
// Not inlined, not in any namespace
void *operator new(std::size_t size);
void *operator new[](std::size_t size);
```

- For `new Type(args)`, the memory is allocated by calling `operator new(sizeof(Type))`.
- For `new Type[n]{initializers}`, the memory is allocated by calling `operator new[](sizeof(Type) * n)`.

* C++17 alignment-aware allocation? Talk later.

operator new

```
void *operator new(std::size_t size);
void *operator new[](std::size_t size);
```

- These two functions **do not know** the type of object(s) to be created.
- `operator new[]` **does not know** the number of objects to be created.

delete Expressions

The execution of a `delete` expression takes two steps:

1. Destroy the object. (Not executed by C++20 destroying-delete)
2. Deallocate the memory.

What we can control is the second step.

operator delete

Memory deallocation is done by a group of functions:

```
// Not inlined, not in any namespace
void operator delete(void *ptr) noexcept;
void operator delete[](void *ptr) noexcept;
```

- `delete ptr` deallocates the memory by calling `operator delete(ptr)`.
- `delete[] ptr` deallocates the memory by calling `operator delete[](ptr)`.

* C++14 sized-deallocation? Talk later.

`operator delete`

Exceptions are not welcomed!

- All deallocation functions are `noexcept`, unless specified otherwise in the declaration.
- If a deallocation function terminates by throwing an exception, the behavior is **undefined**, even if it is declared with `noexcept(false)`.
  - Such exception is not expected to be caught. Stack is possibly not unwound in this case.

Contents

Standard operator new

The following functions are *replacable*:

```
void *operator new(std::size_t size);
void *operator new[](std::size_t size);
void operator delete(void *ptr) noexcept;
void operator delete[](void *ptr) noexcept;
```

- Standard versions (normal versions) are defined in standard library file `<new>`.
- But the compiler will choose the user-defined version if there exists one.
- In this case, they do not constitute redefinition.

Standard operator new

Difference between `operator new` and `malloc`?

Standard operator new

Difference between `operator new` and `malloc`?

Basic:

- `operator new` allocates some memory when `size == 0`, while the behavior of `malloc(0)` is implementation-defined.
- On failure, `operator new` throws `std::bad_alloc`, while `malloc` returns null pointer.

Standard operator new

A simple operator new that uses malloc for allocation:

```
void *operator new(std::size_t size) {
  if (size == 0)
    size = 1;
  if (auto ptr = std::malloc(size))
    return ptr;
  throw std::bad_alloc{};
}
```

(Similar for operator new[]...)

Standard operator new

In fact, operator new keeps trying to allocate memory and, on failure, does some possible adjustment by calling a **new-handler**, until the allocation succeeds or no new-handler is available.

```
void *operator new(std::size_t size) {
  if (size == 0)
    size = 1;
  while (true) {
    if (auto ptr = std::malloc(size))
      return ptr;
    auto handler = std::get_new_handler();
    if (handler)
      handler();
    else
      throw std::bad_alloc{};
  }
}
```

Standard operator delete

Possible implementation of operator delete that uses std::free to deallocate memory:

```
void operator delete(void *ptr) noexcept {
  std::free(ptr);
}
```

- Make sure it is safe to delete a null pointer.
- Similar for operator delete[].

Contents

Why Overload them?

Effective C++ Item 50 talks about the following most common reasons:

- To detect usage errors.
- To improve efficiency.
- To collect usage statistics.

Record Allocations

```cpp
void *operator new(std::size_t size) {
  if (size == 0)
    size = 1;
  while (true) {
    if (auto ptr = std::malloc(size)) {
      recorder.add_record(ptr);
      return ptr;
    }
    auto handler = std::get_new_handler();
    if (handler)
      handler();
    else
      throw std::bad_alloc{};
  }
}
```

Record Allocations

```
void operator delete(void *ptr) noexcept {
  if (!recorder.find(ptr))
    throw std::invalid_argument
        {"Invalid pointer passed to operator delete"};
  recorder.remove_record(ptr);
  std::free(ptr);
}
```

Class-specific Versions

```
struct Widget {
  static void *operator new(std::size_t size);
  static void *operator new[](std::size_t size);
  static void operator delete(void *ptr);
  static void operator delete[](void *ptr);
};
```

- When we use `new`/`new[]` to create class-type objects, the lookup for `operator new`/`operator new[]` begins in the class scope.
- If the `new`-expression uses the form `::new`, the class-scope lookup is **bypassed** and the global version `::operator new`/`::operator new[]` will be called.

Class-specific Versions

```
struct Widget {
  static void *operator new(std::size_t size);
  static void *operator new[](std::size_t size);
  static void operator delete(void *ptr);
  static void operator delete[](void *ptr);
};
```

- The keyword `static` is optional: these functions are always static members.
- Deallocation functions are implicitly `noexcept`.

Example: Heap_tracked

This example is from *More Effective C++* Item 27: Requiring or prohibiting heap-based objects.

- `dynamic_cast<const void *>(ptr)` yields the beginning address of the object. (Casting it to `void *`, `volatile void *` or `const volatile void *` also work.)
- Track whether an object is heap-based by inheriting `Heap_tracked` in a **mixin** style.

Contents

new/delete with Extra Arguments

```
void *operator new(std::size_t size, const std::nothrow_t &) noexcept;
void *operator new[](std::size_t size, const std::nothrow_t &) noexcept;
void *operator new(std::size_t size, void *place) noexcept;
void *operator new[](std::size_t size, void *place) noexcept;

void operator delete(void *ptr, const std::nothrow_t &) noexcept;
void operator delete[](void *ptr, const std::nothrow_t &) noexcept;
void operator delete(void *ptr, void *place) noexcept;
void operator delete[](void *ptr, void *place) noexcept;
```

Non-throwing operator new

```
auto ptr = new (std::nothrow) Type(args);
auto arr = new (std::nothrow) Type[n];
```

- std::nothrow is a tag of type std::nothrow_t defined in <new>.

```
namespace std {
  struct nothrow_t {
    explicit nothrow_t() = default;
  };
  extern const nothrow_t nothrow;
}
```

Non-throwing operator new

```
auto ptr = new (std::nothrow) Type(args);
auto arr = new (std::nothrow) Type[n];
```

- `new (std::nothrow) Type(args)` calls `operator new(sizeof(Type), std::nothrow)` for memory allocation.
- `new (std::nothrow) Type[n]{initializers}` calls `operator new[](sizeof(Type) * n, std::nothrow)` for memory allocation.
- Returns null pointer on failure. No exception would be thrown.

Non-throwing operator new

Possible implementation:

```
void *operator new(std::size_t size,
                   const std::nothrow_t &) noexcept {
  void *ptr = nullptr;
  try {
    ptr = ::operator new(size);
  } catch (...) {}
  return ptr;
}
```

Placement-new

The “real” placement-new:

```
Type *pos1 = somewhere();
new (pos1) Type(args);
Type *pos2 = somewhere_else();
new (pos2) Type[n]{a,b,c,...};
```

- No allocation is performed.
- Placement-new is used for construct object(s) on given place.

Placement-new

Possible implementation:

```
void *operator new(std::size_t, void *place) noexcept {
  return place;
}
void *operator new[](std::size_t, void *place) noexcept {
  return place;
}
```

Notice

These two functions (as well as the corresponding `operator delete`s) **cannot** be replaced.

Contents

Customized Arguments

```
void *operator new(std::size_t size,
                   long line, const char *file) {
  auto ptr = ::operator new(size);
  log_allocation(ptr, size, line, file);
  return ptr;
}
auto ptr = new (__LINE__, __FILE__) Type(args);
```

Placement-delete

Recall the two steps for a `new` expression:

1. Allocate enough memory.
2. Construct the object(s).

For a `new` expression `new (args...) Type(ctor_args...)`, if an exception is thrown during the **second** step:

- The corresponding `operator delete` is called with `ptr, args...` passed to it, where `ptr` is the beginning location of memory allocated in the first step.
- The `operator delete` deallocates the memory allocated by `operator new` to ensure memory-safety and exception-safety.

Placement-delete

Possible implementation for non-throwing `new`:

```
void operator delete(void *ptr,
                     const std::nothrow_t &) noexcept {
  ::operator delete(ptr);
}
```

Placement-delete

Possible implementation for non-throwing new:

```
void operator delete(void *ptr,
                     const std::nothrow_t &) noexcept {
  ::operator delete(ptr);
}
```

Possible implementation of placement-delete for our customized placement-new:

```
void operator delete(void *ptr,
           long line, const char *file) noexcept {
  log_failure(ptr, line, file);
  ::operator delete(ptr);
}
```

Placement-delete

Possible implementation for the real "placement-new"?

Placement-delete

Possible implementation for the real "placement-new"?

```
void operator delete(void *, void *) noexcept {}
void operator delete[](void *, void *) noexcept {}
```

Placement-delete

Possible implementation for the real "placement-new"?

```
void operator delete(void *, void *) noexcept {}
void operator delete[](void *, void *) noexcept {}
```

Notice

If no suitable placement-delete is found, no deallocation function would be called, which possibly results in memory leak.

Placement-delete

Which `operator delete` is called?

```
auto ptr = new (std::nothrow) Type(args);
delete ptr;
```

Placement-delete

Which operator delete is called?

```
auto ptr = new (std::nothrow) Type(args);
delete ptr;
```

Answer: **the normal version with no extra arguments.** A placement-delete is called only when constructors throw an exception.

Contents

Sized-delete

For some kinds of deallocation functions, it might be necessary to know the **size** of the block of memory to be deallocated.
A deallocation function which receives an extra `std::size_t` parameter is a **sized-deallocation function**.

Notice
Sized-deallocation function is a **usual** deallocation function. It is not a placement version.

Sized-delete

Before C++14, sized-delete could only be class-scoped static member:

```
struct Widget {
  int x;
  static void operator delete(void *p, std::size_t sz) {
    std::cout << "size == " << sz << '\n';
    ::operator delete(p);
  }
};
auto ptr = new Widget;
delete ptr;
```

Output:

```
size == 4
```

Sized-delete

Before C++14, sized-delete could only be class-scoped static member:

```
struct Widget {
  int x;
  static void operator delete[](void *p, std::size_t sz) {
    std::cout << "size == " << sz << '\n';
    ::operator delete[](p);
  }
};
auto ptr = new Widget[100];
delete []ptr;
```

Possible output:

```
size == 408
```

Sized-delete

If a sized-deallocation function is defined and its corresponding unsized version is not, the sized version is called for a `delete`-expression to deallocate the memory.

- The `std::size_t` argument is passed by the compiler automatically.

Sized-delete

If a sized-deallocation function is defined and its corresponding unsized version is not, the sized version is called for a delete-expression to deallocate the memory.

- The `std::size_t` argument is passed by the compiler automatically.

Since C++14, global sized-deallocation functions are also allowed:

```
void operator delete(void *ptr, std::size_t size) noexcept;
void operator delete[](void *ptr, std::size_t size) noexcept;
```

Sized-delete

The compiler may choose to call the sized version **or** the unsized one.

- Clang-14 calls the unsized version by default, even when the sized version is provided.
- Calls to one version must be effectively equivalent to the other version, otherwise the program has undefined behavior.
- The standard library implementations of sized-deallocation functions directly call the unsized versions.

Contents

Alignment of Objects

Every object type has the **alignment requirement** property, representing the number of bytes between successive addresses at which objects of this type can be allocated.

- Alignment requirement of an object is an integer value of type `std::size_t`, and is always a power of 2.
- Alignment requirement could be queried with `alignof` or `std::alignment_of`.

Alignment of Objects

On 64-bit Ubuntu 20.04:

- `alignof(int)`: 4
- `alignof(long)`: 8
- `alignof(char)`: 1

Alignment of Objects

On 64-bit Ubuntu 20.04:

- `alignof(int)`: 4
- `alignof(long)`: 8
- `alignof(char)`: 1

```
struct Widget {
  int x;
  char y;
};
```

`alignof(Widget)` is 4 because `x` must be placed at 4-byte boundaries.

Alignment of Objects

We may use alignas to set a special alignment requirement:

```
struct alignas(32) Widget {
  // ...
};
```

Alignment of Objects

We may use `alignas` to set a special alignment requirement:

```
struct alignas(32) Widget {
  // ...
};
```

Some types may have special alignment requirements: Intel intrinsic type `__m256` is a 256-bit type and is aligned at 32-byte boundaries.

Alignment-aware Allocation

Since C++17, a group of alignment-aware allocation functions are introduced:

```
void *operator new(std::size_t size, std::align_val_t al);
void *operator new[](std::size_t size, std::align_val_t al);
void *operator new(std::size_t size,
      std::align_val_t al, const std::nothrow_t &) noexcept;
void *operator new[](std::size_t size,
      std::align_val_t al, const std::nothrow_t &) noexcept;
```

(Together with their deallocation functions and sized-deallocation functions.)

- `<cstdlib>` also introduces `std::aligned_alloc`.

Alignment-aware Allocation

```
namespace std {
  enum class align_val_t : size_t {};
}
```

- Normal allocation functions allocate objects aligned at `__STDCPP_DEFAULT_NEW_ALIGNMENT__` (might be 16).
- If `alignof(Type)` exceeds the default `new` alignment, the `new`-expression calls the alignment-aware `operator new` and passes `alignof(Type)` as the second argument. (Similar for array-`new`.)

Contents

Destroying-delete

```
namespace std {
  struct destroying_delete_t {
    explicit destroying_delete_t() = default;
  };
  inline constexpr destroying_delete_t
      destroying_delete{}; // a tag
}
struct T {
  void operator delete(T *ptr, std::destroying_delete_t);
  // Together with its sized and alignment-aware versions.
};
```

Destroying-delete

If a destroying-`delete` is defined:

- `delete`-expressions do not execute the destructor before a call to `operator delete`.
- The destroying-`delete` is chosen in preference to the normal `operator delete`.
- It becomes the responsibility of the destroying-`delete` to destroy the object correctly.

Destroying-delete

If a destroying-`delete` is defined:

- `delete`-expressions do not execute the destructor before a call to `operator delete`.
- The destroying-`delete` is chosen in preference to the normal `operator delete`.
- It becomes the responsibility of the destroying-`delete` to destroy the object correctly.

What for? See `https://open-std.org/JTC1/SC22/WG21/docs/papers/2017/p0722r1.html`.