Overloaded and Customized new/delete

GKxx

July 11, 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]`, the memory is allocated by calling `operator new[](sizeof(Type) * n)`.

* C++17 alignment-aware `operator new`? 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)`.

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 == 1`, 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

```
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 Replacements

```
class Widget {
 public:
  static void *operator new(std::size_t size);
  static void *operator new[](std::size_t size);
  // Does not have to be noexcept.
  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.

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(std::size_t size, const std::nothrow_t &) noexcept;
void operator delete[](std::size_t size, const std::nothrow_t &)
    noexcept;
void operator delete(std::size_t size, void *place) noexcept;
void operator delete[](std::size_t size, 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](args)` 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 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 deletes`) **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.