Type-aware allocation and deallocation functions

Document #:	D2719R0
Date:	2022-11-11
Project:	Programming Language C++
Audience:	Evolution

Reply-to:	Louis Dionne <ldionne.2@gmail.com>

1 Introduction

C++ currently provides two ways of customizing the creation of new objects in new expressions. First, `operator new` can be provided as a static member function of a class, like `void* T::operator new`. If such a declaration is provided, an expression like `new T(...)` will use that allocation function. Otherwise, the global version of `operator new` can be replaced by users in a type-agnostic way, by implementing `void* operator new(size_t)` and its variants. A similar mechanism exists for *delete-expressions*.

This paper proposes an extension to the way *new-expressions* and *delete-expressions* select their allocation and deallocation function to provide the power of the in-class operator variant, which retains the knowledge of the type being allocated/deallocated, without requiring the intrusive definition of an in-class function. This is achieved by introducing new `std::typed_alloc_t` and `std::typed_dealloc_t` tag types and tweaking the overload resolution already performed by *new-expressions* and *delete-expressions* to find a typed allocation or deallocation function if one exists. The following describes (roughly) the search process performed by the compiler before and after this paper:

Before

```
// user writes:
new (args...) T(...)

// compiler checks:
T::operator new(sizeof(T), args...) // (1)
::operator new(sizeof(T), args...)  // (2)
```

After

```
// user writes:
new (args...) T(...)

// compiler checks:
T::operator new(sizeof(T), args...) // (1)
::operatornew< T>(std::typed_alloc,sizeof(T),args…)//(2)
::operator new(sizeof(T), args...)  // (3)
```

Before

```
// user writes:
delete obj

// compiler checks:
T::operator delete(void-ptr) // (1)
::operator delete(void-ptr)  // (2)
```

After

```
// user writes:
delete ptr

// compiler checks:
T::operator delete(void-ptr) // (1)
::operatordelete< T>(std::typed_dealloc, ptr) // (2)
::operator delete(void-ptr)  // (3)
```

2 Motivation

Knowledge of the type being allocated in a *new-expression* is necessary in order to achieve certain levels of flexibility when defining a custom allocation function. However, requiring an intrusive in-class definition to achieve this is not realistic in various circumstances, for example when wanting to customize allocation for types that are controlled by a third-party, or when customizing allocation for a very large number of types.

To illustrate this, Apple recently published a blog post explaining a technique used inside XNU to mitigate various exploits. At its core, the technique consists in allocating objects of each type in a different bucket. By collocating all objects of the same type into the same region of memory, it becomes much harder for an attacker to exploit a type confusion vulnerability. Since its introduction in the kernel, this technique alone has been by far the most effective at mitigating type confusion vulnerabilities.

In a world where security is increasingly important, it may make sense for some code bases to adopt mitigation techniques such as the above. However, these techniques require a large-scale and almost system-wide customization of how allocation is performed while retaining type information, which is not supported by C++ today. In a world where C++ has been criticized for lagging behind security-wise, we believe that C++ would gain a lot by providing more flexibility for customizing how allocation is performed while retaining type information.

3 Current behavior recap

Today, the compiler performs a lookup in the allocated type's class scope (for `T::operator new`), and then a lookup in the global scope (for `::operator new`) if the previous one failed.

The compiler then performs overload resolution on the name it found in the previous step (let's call it `NEW`) by assembling an argument list that depends on whether `T` has a new-extended alignment or not. For the sake of simplicity, assume that `T` does not have a new-extended alignment. The compiler starts by performing overload resolution as-if the following expression were used:

```
NEW(sizeof(T), args...)
```

If that succeeds, the compiler selects the overload that won. If it does not, the compiler performs overload resolution again as-if the following expression were used:

```
NEW(sizeof(T), std::align_val_t(alignof(T)), args...)
```

If that succeeds, the compiler selects the overload that won. If it does not, the program is ill-formed. For a type `T` that has new-extended alignment, the order of the two overload resolutions performed above is simply reversed.

Delete-expressions behave similarly, but lookup is performed in the context where the virtual destructor is defined, if there is a virtual destructor. The overload resolution process then works by preferring a destroying delete, followed by an aligned delete (if the type has new-extended alignment), followed by the usual operator delete (with or without a `size_t` parameter depending on whether the considered `operator delete` is a member function or not).

4 Proposal

This proposal does not affect the name lookup step explained above. It essentially adds an additional step of overload resolution before each existing overload resolution step described above, passing the allocated type `T` as an explicit template argument and using a new tag type `std::typed_alloc_t` for disambiguation. Hence, the compiler would first perform overload resolution as-if the following call were issued:

```
NEW<T>(std::typed_alloc, sizeof(T), args...)
```

If that fails, it would then perform overload resolution as usual:

```
NEW(sizeof(T), args...)
```

If that fails, it would then proceed to check for the presence of a typed allocation function that takes `align_val_t`:

```
NEW<T>(std::typed_alloc, sizeof(T), std::align_val_t(alignof(T)), args...)
```

If that fails, it would finally proceed to check for the presence of an untyped allocation function that takes `align_val_t`, as usual:

```
NEW(sizeof(T), std::align_val_t(alignof(T)), args...)
```

For a type `T` with new-extended alignment, the two `align_val_t` overload resolution steps would happen before the two non-`align_val_t` steps, analogously to what happens today. In summary, overload resolution would now look like:

```
NEW< T>(std::typed_alloc,sizeof(T), args…)                                  // (1)
NEW(sizeof(T), args...)                                                     // (2)
NEW< T>(std::typed_alloc,sizeof(T), std::align_val_t(alignof(T)), args…)    // (3)
NEW(sizeof(T), std::align_val_t(alignof(T)), args...)                       // (4)
```

As a result, users can define the following `operator new`s if they want to override allocation behavior while retaining type information:

```
template <class T>
void* operator new(std::typed_alloc_t, std::size_t size, args...);

template <class T>
void* operator new(std::typed_alloc_t, std::size_t size, std::align_val_t align, args...);
```

> Note that these operators are free to return `void*` or `T*`. In the end, the new-expression is what's calling those operators, and the new-expression needs to be able to convert the result to a `T*` like it normally does for an untyped allocation function.

For example:

```
template <class T>
  requires (!std::integral<T>)
void* operator new(std::typed_alloc_t, std::size_t size); // (1)

struct Foo { };

struct Bar {
  void* operator new(std::size_t size); // (2)
};

void f() {
  Foo* foo = new Foo(); // calls (1)
  Bar* bar = new Bar(); // calls (2)
  int* baz = new int(); // calls standard library operator new
}
```

For delete-expressions, the compiler would start by doing name lookup like it normally does. Then, it would perform overload resolution like so, using a new tag type `std::typed_dealloc_t` introduced in the library. Here, `ptr` is a pointer to an object of dynamic type `T`:

```
DELETE(ptr, std::destroying_delete)                                         // (1)
DELETE< T>(std::typed_dealloc,ptr[, size])                                  // (2)
DELETE(static_cast<void*>(ptr)[, size])                                     // (3)
```

```
DELETE< T>(std::typed_dealloc,ptr[, size], std::align_val_t(alignof(T))) // (4)
DELETE(static_cast<void*>(ptr)[, size], std::align_val_t(alignof(T)))    // (5)
```

For example:

```
template <class T>
  requires (!std::integral<T>)
void operator delete(std::typed_dealloc_t, void* ptr, std::size_t size); // (2)

struct Foo { };

struct Bar {
  void operator delete(void* ptr); // (3)
};

void f() {
  Foo* foo = ...;
  Bar* bar = ...;
  int* baz = ...;

  delete foo; // calls (2)
  delete bar; // calls (3)
  delete baz; // calls standard library operator delete
}
```

5 Design choices, notes and possible extensions

5.1 Impact on the library

This proposal does not have a significant impact on the library, which only gets new `std::typed_alloc_t` and `std::typed_dealloc_t` tag types and `std::typed_alloc` and `std::typed_dealloc` global objects of those types. Adding new global allocation/deallocation functions is not necessary, since the compiler will find the existing ones by default, and those are sufficient for a program to function.

5.2 Design choice: Order of arguments

Should `std::typed_alloc_t` come before or after the size? We started with the size first and changed it because we felt it made it clearer that typed allocation was a "different kind of allocation function". A similar comment can be made for `std::typed_dealloc_t`, which could go after the `ptr` argument of `operator delete`.

At the end of the day, we don't care strongly but we think typed allocation and deallocation should be consistent with each other.

However, we do feel like `std::typed_alloc_t` should come before `std::align_val_t` since `std::typed_alloc_t` is checked with higher precedence than `std::align_val_t` in the overload resolution steps.

5.3 Design choice: No `std::typed_alloc_t` for `operator new`

In an earlier draft, this paper was proposing the following (seemingly simpler) mechanism instead. Instead of introducing `std::typed_alloc_t`, the compiler would perform the following overload resolution steps (picking the first one that succeeds):

```
NEW< T>(sizeof(T),args…)                                  // (1)
NEW(sizeof(T), args...)                                   // (2)
NEW< T>(sizeof(T),std::align_val_t(alignof(T)), args…)    // (3)
NEW(sizeof(T), std::align_val_t(alignof(T)), args...)     // (4)
```

The only difference here is that we're not passing `std::typed_alloc_t` as a first argument. The problem with this approach is that existing code is free to have defined a global `operator new` like so:

```
template <class ...Args>
void* operator new(std::size_t size, Args ...args);
```

In steps (1) and (3), an existing templated `operator new` could match even though it was never intended to be called in this case. The result would be that the first template argument is explicitly provided by the compiler. This could result in a substitution failure (which would be fine), but could also result in a valid function call triggering an implicit conversion to the now-explicitly-provided first template argument, which would change the meaning of existing valid programs.

Overrides of placement-new are not extremely frequent and cases where this would cause a behavior change are even less frequent. If the Committee is comfortable with this potential change in semantics, this design could also be used instead.

5.4 Design choice: No `std::typed_dealloc_t` for `operator delete`

We could perhaps simplify the design for delete-expressions by passing the type of the object directly:

```
DELETE(static_cast<T*>(ptr), std::destroying_delete)
DELETE(static_cast<T*>(ptr), std::destroying_delete, std::align_val_t(alignof(T)))
DELETE(static_cast< T*>(ptr))
DELETE(static_cast<void*>(ptr))
DELETE(static_cast< T*>(ptr),std::align_val_t(alignof(T)))
DELETE(static_cast<void*>(ptr), std::align_val_t(alignof(T)))
```

However, we believe that having using a tag type makes it clearer that typed deallocation functions do not destroy the object.

5.5 Design choice: Using `std::identity` instead of a tag type

We could perform overload resolution on the following expressions instead:

```
NEW(std::type_identity, sizeof(T),args…)
NEW(sizeof(T), args...)
NEW(std::type_identity, sizeof(T),std::align_val_t(alignof(T)), args…)
NEW(sizeof(T), std::align_val_t(alignof(T)), args...)
```

Allocation functions would then be defined like:

```
template <class T>
void* operator new(std::type_identity<T>, std::size_t size, args...);
```

We don't have a strong preference, but we feel that it's a bit of a gimmick.

5.6 Design choice: Should in-class typed allocation functions be supported?

A consequence of the proposal as currently worded is that overload resolution on `NEW<T>(std::typed_alloc, size, args...)` is attempted even in the case where the `operator new` name was found in `T`'s class scope instead of in the global scope. This means that an in-class typed allocation function would be valid:

```
struct Foo {
  template <class T>
  void* operator new(std::size_t size, std::typed_alloc_t, args...);
};
```

This is a natural consequence of the current wording, and we believe it is a useful feature. For example:

```
struct Bar : Foo { };
new Bar(); // calls Foo::operator new<Bar>(...)
```

This makes it easy to override the allocation behavior in a whole class hierarchy without having to redefine `operator new` in each derived class, which is a pain point in some of the code bases we've surveyed.

All of this also applies to typed `operator delete` defined inside a class.

5.7 Extension: Name lookup in a class' enclosing namespace

A possible extension of this proposal would be to also look up `operator new` and `operator delete` in the class' enclosing namespace. We do not propose that at this time to avoid adding complexity to the name lookup performed by new-expressions and delete-expressions, but this could probably be incorporated to this proposal if the Committee deems it useful.

6 Proposed wording

This is probably incomplete, but we still include it to show that the proposal is essentially a minor extension to the existing overload resolution rules in new-expressions.

TODO

7 Implementation experience

We made a partial implementation in Clang and did not run into major roadblocks.