Type-aware allocation and deallocation functions

Document #: D2719R0
Date: 2024-03-21
Project: Programming Language C++
Audience: Evolution
Reply-to: Louis Dionne
<ldionne@apple.com>
Oliver Hunt
<oliver@apple.com>

1 Introduction

C++ currently provides two ways of customizing the creation of 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 tweaking the search already performed by *new-expressions* and *delete-expressions* to also include a call to any free function `operator new` with an additional tag parameter of `std::type_identity<T>`. The following describes (roughly) the search process performed by the compiler before and after this paper:

Before

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

// compiler checks (in order):
T::operator new(sizeof(T), args...)
::operator new(sizeof(T), args...)
```

After

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

// compiler checks (in order):
T::operator new(sizeof(T), args...)
operator new(sizeof(T),type_identity< T>{}, args…)
::operator new(sizeof(T), args...)
```

Before

```
// user writes:
new (args...) T[n]

// compiler checks (in order):
T::operator new[](n*sizeof(T), args...)
::operator new[](n*sizeof(T), args...)
```

After

```
// user writes:
new (args...) T[n];

// compiler checks (in order):
T::operator new[](n*sizeof(T), args...)
operator new[](n*sizeof(T),type_identity< T>{}, args…)
::operator new[](n*sizeof(T), args...)
```

Before	After

`// user writes:`
`delete ptr`
 	
`// compiler checks (in order):`
`T::operator delete(void-ptr)`
`::operator delete(void-ptr)`
	`// user writes:`
	`delete ptr`
	
	`// compiler checks (in order):`
	`T::operator delete(void-ptr)`
	`operator delete(void-ptr,type_identity< T>{})`
	`::operator delete(void-ptr)`

Before	After

`// user writes:`
`delete[] ptr`
 	
`// compiler checks (in order):`
`T::operator delete[](void-ptr)`
`::operator delete[](void-ptr)`
	`// user writes:`
	`delete[] ptr`
	
	`// compiler checks (in order):`
	`T::operator delete[](void-ptr)`
	`operator delete[](void-ptr,type_identity< T>{})`
	`::operator delete[](void-ptr)`

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.

In the wild, we often see code bases overriding the global (and untyped) `operator new` via the usual link-time mechanism and running into issues because they really only intended for their custom `operator new` to be used within their own code, not by all the code in their process. We also run into issues where multiple libraries attempt to replace the global `operator new` and end up with a complex ODR violation bug. The simple change proposed by this paper provides a way for most code bases to achieve what they *actually* want, which is to override `operator new` for a family of types that they control without overriding it for the whole process. The overriding also happens at compile-time instead of link-time, which is both supported by all known implementations (unlike link-time) and better understood by users than e.g. weak definitions.

2.1 A concrete use case

A few years ago, Apple published a blog post explaining a technique used inside its kernel (XNU) to mitigate various exploits. At its core, the technique roughly 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 this one. 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. While not sufficient in itself to make C++ safer, the change proposed in this paper is a necessary building block for technology such as the above which can greatly improve the security of C++ applications.

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. Once the name lookup has been done and the compiler has decided whether it was looking for `T::operator new` or `::operator new`, name lookup will not be done again even if the steps that follow were to fail.

The compiler then performs overload resolution on the name it found in the previous step (let's call that name `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, with lookup being performed in the context of the dynamic type of the object, in case it differs from its static type. 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

Our proposal essentially adds a new step in the resolution above that also considers a free function `operator new` after it considers a member `T::operator new`, but before the global `::operator new`. That free function is called as

```
operator new(sizeof(T), std::type_identity<T>{}, placement-args...)
```

In particular, this means that it triggers ADL, which allows defining such an operator in e.g. a custom namespace. We mostly discuss `operator new` below, but `operator new[]`, `operator delete` and `operator delete[]` are all handled analogously.

4.1 Wording approach

This proposal is best explained as two distinct changes to the existing rules explained above. First, observe how we currently perform name lookup first to fix the name we're using (either `T::operator new` or `::operator new`), and then perform overload resolution to find the best candidate given that name. This needs to change because if we introduce a free function in the mix that is found by ADL, we could often end up finding that free function as the result of name lookup. However, the mere presence of such a free function doesn't guarantee at all that it is a viable candidate given the arguments we have (for example if the free function merely happened to be in one of the namespaces we looked in). As a result, we could often end up in a situtation where `new Foo(...)` fails during overload resolution simply because a non-viable `operator new` free function was located in one of `Foo`'s associated namespaces.

Consequently, the first change we need to make is to perform name lookup for `T::operator new`, and if found, then perform overload resolution on that name immediately. If overload resolution fails, then go on to the next candidate (which is `::operator new`) and perform overload resolution on that name. This change can be seen as a relaxation of the current rules and cannot affect any existing valid program. Indeed, code where `T::operator new` is currently selected but where overload resolution fails actually doesn't compile today, since the compiler falls flat after failing to perform overload resolution. After this change, such code would now fall back to trying overload resolution on `::operator new`, which might succeed.

The second change we need to make is the addition of a new step in the search, between `T::operator new` and `::operator new`. Assuming that the name lookup or the overload resolution for `T::operator new` fails, we would now perform an argument-dependent name lookup for a free function named `operator new` as-if we had the following expression:

```
operator new(sizeof(T), std::type_identity<T>{}, args...)
```

In other words, the set of associated namespaces would include the `std` namespace (via `std::type_identity`), the associated namespaces of `T` (by virtue of the template parameter of `std::type_identity`), and that of any other provided placement-arguments. In particular, note that this may or may not include the global namespace.

If this name lookup succeeds, we would perform overload resolution on this function in a way similar to what we currently do in expr.new#19, but tweaked to take into account the `std::type_identity` argument:

```
// first resolution attempt
operator new(sizeof(T), std::type_identity<T>{}, args...)

// second resolution attempt
operator new(sizeof(T), std::align_val_t(alignof(T)), std::type_identity<T>{}, args...)
```

For a type with new-extended alignment, we simply reverse the order of these overload resolution attempts. If this overload resolution fails, we would then move on to the next candidate found by name lookup, which is `::operator new`, and we would perform the usual overload resolution on it.

Delete expressions or array new/delete expressions work in a way entirely analoguous to what is described above for single-object new-expressions.

4.2 Example

```
namespace lib {
  struct Foo { };
  void* operator new(std::size_t, std::type_identity<Foo>); // (1)

  struct Foo2 { };
}

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

void* operator new(std::size_t); // (3)

void f() {
  new lib::Foo();  // calls (1)
  new Bar();       // calls (2)
  new lib::Foo2(); // (1) is seen but fails overload resolution, we end up calling (3)
  new int();       // calls (3)
}
```

5 Design choices and notes

5.1 Impact on the library

This proposal does not have any impact on the library, since this only tweaks the search process performed by the compiler when it evaluates a new-expression and a delete-expression. In particular, we do not propose adding new type-aware free function `operator new` variants in the standard library at this time, althought this could be investigated in the future.

5.2 Design choice: Order of arguments

When writing this paper, we went back and forth of the order of arguments. This paper proposes:

```
operator new(std::size_t, std::type_identity<T>, placement-args...)
operator new(std::size_t, std::align_val_t, std::type_identity<T>, placement-args...)

operator delete(void*, std::type_identity<T>)
operator delete(void*, std::size_t, std::type_identity<T>)
operator delete(void*, std::size_t, std::align_val_t, std::type_identity<T>)
```

Another approach would be:

```
operator new(std::type_identity<T>, std::size_t, placement-args...)
operator new(std::type_identity<T>, std::size_t, std::align_val_t, placement-args...)

operator delete(std::type_identity<T>, void*)
operator delete(std::type_identity<T>, void*, std::size_t)
operator delete(std::type_identity<T>, void*, std::size_t, std::align_val_t)
```

We would like input from EWG on this matter.

5.3 Design choice: Template argument vs `std::type_identity`

In an earlier draft, this paper was proposing the following (seemingly simpler) mechanism instead. Instead of reusing `std::type_identity`, the compiler would search as per the following expression:

```
operator new<T>(sizeof(T), args...)
```

The only difference here is that we're not passing `std::type_identity` as a first argument and we are passing it as a template argument instead. Unfortunately, this has two problems. First, this doesn't allow ADL to kick in, severely reducing the flexibility for defining the operator. The second problem is that existing code is allowed to have defined a global `operator new` like so:

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

We believe that this is not a far fetched possibility and that we may break some code if we went down that route. Even worse, an existing templated `operator new` could match even though it was never intended to be called. The result would be that the first template argument is explicitly provided by the compiler, which could result in a substitution failure (that is acceptable) or in a valid function call triggering an implicit conversion to the now-explicitly-provided first template argument, which would change the meaning of valid programs.

5.4 Design choice: No `std::type_identity` 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, and this also allows keeping the `operator delete` call consistent with the `operator new` call, which is a nice property.

5.5 Is the current name lookup for `T::operator new` done this way on purpose?

Currently, the search that happens for a new-expression is worded such that if a `T::operator new` is found and overload resolution fails, the program is ill-formed. As explained in this proposal, this is stricter than needed and we propose relaxing that.

However, one side effect of this strictness is that the compiler will error if a user defines some variants of `T::operator new` but forgets to define some other variants. For example:

```
struct arg { };

struct Foo {
  static void* operator new(std::size_t, arg);
};

int main() {
  new Foo();
}
```

Today, this is a compiler error because we find `Foo::operator new` and then fail to perform overload resolution, so we don't fall back to the global `::operator new`. I don't know whether this is by design or just an unintended consequence of the wording, however this seems a bit contrived.

In all cases, if we wanted this to remain ill-formed, we could either count on compiler diagnostics to warn in that case, or we could word the search process to say that if the overload resolution on `T::operator new` fails, the program is ill-formed and the search stops. This doesn't seem useful to me, but it's on the table.

5.6 Should a templated operator delete be allowed as a usual allocation function?

dynamic.deallocation#3 states that a template function is never considered as a usual allocation function. This prevents general allocators from specifying a general type-aware deallocation operator that can reflect the correct concrete type when invoking a deleting constructor. e.g.

```
class Root {
public:
  template <class T> static void *operator new(std::size_t, std::type_identity<T>);
  template <class T> static void operator delete(void *, std::type_identity<T>);
};

class A : public Root { };

class B : public Root {
  virtual ~B();
};
class C : public B { };

int foo(A* a, B* b, B* c /* actually points to a C object */) {
  delete a; // We want Root::operator delete<A>(...) to be called

  delete b; // We want Root::operator delete<B> to be called, and that happens via
            // the deleting virtual destructor.
            // TODO

  delete c; // The allocator would call the deleting virtual destructor, and expect this to invoke
            // Root::operator delete<C>
            // TODO
}
```

Allowing a template delete operator to be considered a usual allocation function could result in previously ignored delete declarations being invoked. We could mitigate this risk by constraining the selection of template `usual` allocations to solely allow the typed allocation signatures - the existing selection rules for usual allocation functions already have strict parameter type restrictions on what is required for an allocation function to be usual so this is not wholly without precedence.

6 Suggested polls

1. Do we want to solve the problem of providing a type aware `operator new` as a free function?
2. Do we prefer `std::type_identity` after the size and alignment (status quo), or first in the argument list?