Allowing allocation in static initialization

Document #:	P2639R0
Date:	2022-09-12
Project:	Programming Language C++
Audience:	EWG, LEWG

Reply-to:	Torben Thaysen <thaysentorben@gmail.com>

Contents

1 Introduction

Probably the most severe limitation to compile time computation in C++ is that the result of the computation can not contain any dynamic allocations. The obvious solution would be to allow non-transient `constexpr` allocations. However these have a serious problem with ensuring consistency in the destructor of the allocated objects as they might have been modified at runtime. For this reason the feature was removed from its initial proposal [P0784R7]. [P1974R0] proposed a solution using `propconst` but at the cost of a significant extension of the type system. At the same time not all use cases would be served with these proposals, in particular the ability to modify these allocations at runtime is very limited. For example if we had a `constexpr std::vector` and wanted to modify or insert even a single element, we would have to make a dynamically allocated copy of it. This is very undesirable particularly for embedded applications.

Thus it is proposed to allow non-transient allocations during constant initialization of non-constexpr variables of static storage duration. Since the distinguishing feature of such allocations is that they are created during static initialization they will be referred to as static allocations in this proposal. The proposed model for static allocations is such that they are indistinguishable from ordinary allocations to all runtime code. This inparticular requires that deleting them at runtime must be well defined.

Currently constant initialization is required to be a constant expression except that it may invoke constructors of non-literal types. This proposal adds to that by allowing creation of non-transient allocations aswell. The rules for constant expressions however remain unaltered.

For an example suppose we have a `constexpr` function `compute_my_vector()` that can be evaluated at compile time:

```
std::vector<int> my_vector = compute_my_vector();
```

With this proposal `my_vector` could be statically initialized where currently it could not. At the same time the implementation is not required to perform constant initialization is this case. To force that behaviour `constinit` can be added to the definition.

2 Allowing Deletions

Several models should be considered for the deletion of these allocations. One approach would be to not allow deletion at all. This would be the simplest method but would encourage the (harmful) practice of creating types that allocate memory in the constructor that is never freed. That could be made more acceptable by introducing an allocation function specifically for static allocations. But that has the problem that the existing containers can not be used and completely new types have to be created.

A different approach could be to allow deletion only in the destructor. The idea would be that because the variable has static storage duration the destructor is only called when the program terminates and thus we could avoid calling it as long as it does not have sideeffects. But there would be no easy safeguard against accidentally invoking a method that ends up deleting the allocation at runtime and using standard containers in this manner would therefore be quite brittle.

However when it is specified that static allocations can be deleted this feature becomes very robust: To all runtime code a static allocation would be indistinguishable from an allocation created during dynamic initialization, which is exactly the property that static initialization should have.

3 User Replacements

3.1 Minimal solution

The main difficulty with allowing deletions is that user replacements of `delete` need to be able to handle such calls (see 3.3). A minimal solution to this would be to introduce a library function that checks whether an address represents a static allocation:

```
namespace std {
  bool is_static_allocation(void* address);
}
```

This could then be used by user replacements to identify and ignore such `delete` calls. Practically `is_static_allocation` could be efficiently implemented by comparing `address` against the bounds of the data segment.

3.2 Transparent solution

However it might be preferable to allow the existing logic of user replacements to handle static allocations gracefully, which could even enable reallocating the space later. For this a more customisable and transparent approach is suggested that consists of two tools: The first allows the user to insert padding before and after the allocation that can be used to store metadata created at runtime or compile time. While the second tool grants the program a complete view of all static allocations at runtime.

3.2.1 Allocation Wrapper

To store metadata with a static allocation the user may declare a template of the form:

```
namespace std {
  template<std::size_t size, std::size_t alignment, bool array>
  struct static_allocation_wrapper;
}
```

The `size` and `alignment` parameter provide the size and alignment requirements of the allocation, while the `array` parameter indicates whether an array allocation function was used. A typical implementation of the template would have some members that contain metadata about the allocation as well as an appropriately sized buffer to hold the actual allocation.

It is unspecified when this template is instantiated, in particular for allocations that are not referenced at runtime it might not be instantiated at all. Every instantiation of this template must be default constructable and the result of default construction should be a constant expression.

Additionally the template must define a `const` member function named `construct_at` with no parameters that returns a `const void*`. This function indicates to the compiler where in the wrapper to place the allocation and should also result in a constant expression. For this the previously default constructed instance of the template is considered to have static storage duration and to be usable in a constant expression, but not to have begun its lifetime in the same constant expression. This ensures that the instance can be read from but not written to. If the address returned by `construct_at` is not inside the wrapper, not properly aligned or executing an appropriate placement new at that address would not be allowed, the program is ill-formed.

3.2.2 Runtime Information

The following type is proposed to provide runtime information about static allocations:

```
namespace std {
  class static_allocation_info {
  public:
    void* allocation_begin() const;
    void* allcoation_end() const;
    size_t allocation_alignment() const;
    size_t allocation_size() const;
    bool user_wrapped() const;
    void* wrapper_begin() const;
    void* wrapper_end() const;
    size_t wrapper_alignment() const;
    size_t wrapper_size() const;
  };
}
```

Where the `user_wrapped` function indicates whether the allocation was wrapped with a user supplied `std::static_allocation_wrapper`, if it returns `false` the values returned by the `wrapper_-` functions are implementation defined. If the allocation is user wrapped the memory accessible through all pointers is specified to be writable.

Note that unlike user replaced allocation functions the `std::static_allocation_wrapper` template must be defined in every translation unit that creates static allocations. For a program that relies on all allocations being wrapped with the template it defines, the `user_wrapped` function provides a simple way to validate this at program startup.

To iterate the allocations the following is proposed:

```
namespace std {
  class static_allocation_range {
  public:
    using iterator = implementation-defined;
    iterator begin() const;
    iterator end() const;
  };
  extern const static_allocation_range static_allocations;
}
```

A likely approach to be adopted by compilers to facilitate this iteration, is to emit the required information into a separate section that is then iterated at runtime. Because the linker would concatenate these sections and the data sections of the allocations in the same order the iteration would be naturally ordered by the address of the allocation. And since this behaviour is likely to be very useful to some programs, this proposal recommends specifying the order of iteration.

3.3 Example

I will illustrate the contents of this section by considering a (very) contrived example. Suppose we have a user replacement for the allocation functions that stores an abstract `AllocationManager` right before the allocation. The replacement for `operator delete` could then look like:

```
void operator delete(void* ptr) throw() {
  AllocationManager* manager = static_cast<AllocationManager**>(ptr)[-1];
  manager->deallocate(ptr);
}
```

Now when a static allocation is deleted and a pointer to it is passed to this function the behaviour would be undefined and the program would likely crash in practice.

Using `std::is_static_allocation` from 3.1 this could be avoided:

```
void operator delete(void* ptr) throw() {
  if (std::is_static_allocation(ptr))
    return;
  AllocationManager* manager = static_cast<AllocationManager**>(ptr)[-1];
  manager->deallocate(ptr);
}
```

And using the the allocation wrappers from 3.2 the static allocations could be made to work with existing logic of the user replacement:

```
template<std::size_t size, std::size_t alignment, bool array>
struct std::static_allocation_wrapper {
  // there needs to be padding here for alignment > sizeof(void*)
  static_assert(alignment <= sizeof(void*));
  AllocationManager* manager = nullptr;

  // this needs to be initialized, otherwise default construction
  // will not result in a constant expression
  std::byte storage alignas(alignment) [size] = {};

  const void* construct_at() const { return storage; }
};

// called somewhere at startup
void setup_static_allocation_manager(AllocationManager* manager) {
  for (const auto& alloc_info : std::static_allocations) {
    if (!alloc_info.user_wrapped())
      // generate error here
    static_cast<AllocationManager**>(alloc_info.wrapper_begin())[0] = manager;
  }
}

// no modification needed here
void operator delete(void* ptr) throw() {
  AllocationManager* manager = static_cast<AllocationManager**>(ptr)[-1];
  manager->deallocate(ptr);
}
```

References

[P0784R7] Peter Dimov, Louis Dionne, Nina Ranns, Richard Smith, and Daveed Vandevoorde. More constexpr containers. http://wg21.link/p0784r7, 2019.

[P1974R0] Jeff Snyder, Louis Dionne, and Daveed Vandevoorde. Non-transient constexpr allocation using propconst. http://wg21.link/p1974r0, 2020.