Constexpr allocation support

New operations required:

- For `allocate`: allocate an object of type `T[N]`, start the lifetime of the array object, do not start the lifetime of any of the contained `T` objects
- For `deallocate`: deallocate an object allocated by `allocate`
- For `construct` / `construct_at`: given a pointer to a `T` allocated by `allocate`, construct an object of type `T` in that storage

Survey of existing library implementations

libc++

`std::allocator<T>` calls `__builtin_operator_new` or `__builtin_operator_delete`. These Clang builtins are identical to calling the corresponding `::operator new` or `::operator delete` functions, except that they permit allocation optimization.

libstdc++

`std::allocator<T>`'s actual implementation is provided by `__allocator_base`, which is determined at configuration time and picks one of various allocator implementations shipped with libstdc++. Commonly-used is `new_allocator`, which implements `allocate` and `deallocate` in terms of `::operator new` and `::operator delete`; other options include `malloc_allocator` (calling `malloc` and `free`), `mt_allocator` (which is a thread-caching pool allocator), and so on.

MSVC STL

`std::allocator<T>` is implemented in terms of `::operator new` and `::operator delete`. One wrinkle: large allocations with low alignment requirements are manually realigned by overallocating and inserting a cookie.

Exposing functionality to the library -- alternatives

Option 1: add a suite of builtins to be used by `std::allocator<T>`

Idea: add a set of builtins that perform the three new operations, such as:

`T *__builtin_allocate(typename T, size_t N)`

Allocate an array `T[N]` and start its lifetime, do not start the lifetime of any array elements.

`void __builtin_deallocate(T *p)`

Deallocate an array allocated by `__builtin_allocate`. Do not run any destructors.

`void __builtin_construct(T *p, ...)`

Equivalent to `new(p) T(args)`.

Pro: conceptually simple interface, one-to-one mapping to necessary operations
Con: substantial frontend complexity, especially for `__builtin_construct`, but also for `__builtin_allocate` (builtins taking a type are complex, at least for Clang).
Con: need to use `is_constant_evaluated` in `allocator`:

```
constexpr T *allocator<T>::allocate(size_t n) {
  if (std::is_constant_evaluated()) return __builtin_allocate(T, n);
  // whatever you would have done normally
}
```

Note that libstdc++ will likely want to do this regardless, and then dispatch to `__allocator_base` to perform the runtime allocation.

Option 2: use *new-expression*s and *delete-expression*s with some kind of modifier

Idea: add new language syntax to represent the allocation and deletion of an array separately from its array elements, such as:

`new T[N] {__uninit_array__}`

Allocate an array `T[N]` and start its lifetime, do not start the lifetime of any array elements. (`__uninit_array__` would be a new keyword.)

`delete[] __uninit_array__ p`

Deallocate an array without running destructors for array elements.

`new (p, __some_magic_tag__) T(...)`

Construct a `T` object in-place. (`__some_magic_tag__` would be defined by the library and recognized by the compiler.)

Pro: general functionality that can be used unconditionally to implement `std::allocator`
Pro: provides an optimizable new/delete pair without the need for a compiler builtin such as `__builtin_operator_new`
Con: adds non-localized complexity (new keywords) and magic tag that we would likely expect to become redundant eventually (once we permit placement new in constant expressions in general)

Option 3: allow additional expression forms to be evaluated inside `std::allocator<T>` and `std::allocator_traits<T>` members

Idea: no changes to standard library implementation. Instead, when the evaluator enters a member of `std::allocator<T>` or `std::allocator_traits<T>` (or `std::construct_at` or `std::ranges::construct_at`) permit constant evaluation of some additional expression forms.

For libc++, we require the following:

- A call to `__builtin_operator_new(size, …)` can be evaluated.
- A call to `__builtin_operator_delete(p, …)` can be evaluated.
- A pointer returned by `__builtin_operator_new` can be cast from `void*` to `T*`.
- `new (p) T(...)` can be evaluated, where `p` points to an element of an array created by `__builtin_operator_new`.

For libstdc++, we require the following:

- For `new_allocator`: as for libc++, but with `::operator new` and `::operator delete` in place of the builtins.
- Will need to deal with other `__allocator_base` options somehow, perhaps by always using `new_allocator` when `is_constant_evaluated()`.

For MSVC STL:

- As for libc++, but with `::operator new` and `::operator delete` in place of the builtins.
- Will need to disable "alignment boosting" code path for large allocations (eg, using `is_constant_evaluated`).

Pro: directly implements the standard's rule:

> For the purposes of determining whether an expression is a core constant expression, the evaluation of a call to a member function of `std::allocator<T>` as defined in allocator.members, where `T` is a literal type, does not disqualify the expression from being a core constant expression, even if the actual evaluation of such a call would otherwise fail the requirements for a core constant expression.

Pro: simple in frontend and in library implementation, at least for Clang + libc++, only requires adding `constexpr` to the standard library implementation
Con: quite "magical" compared to explicit opt-in syntax, only works within the specially-identified member functions, may need additional magic to prevent this from applying within user-defined constructors invoked by `allocator_traits<T>::construct` and user specializations of `std::allocator_traits` (and `std::allocator`?).

Proposal: option 3 (implemented in Clang)

Transitively within the evaluation of a call to a specified member function of `std::allocator<T>`, the following are permitted in constant expressions:

`__builtin_operator_new(size, …)`
`::operator new(size, …)`

> Allocate an array `T[size / sizeof(T)]` and start its lifetime, do not start the lifetime of any array elements. Arguments after `size` are ignored. Callable transitively within `allocate`.

`__builtin_operator_delete(p, …)`
`::operator delete(p, …)`

> Deallocate an object allocated by one of the above allocation functions. Callable transitively within `deallocate`.

`static_cast<cv T*>(p)`

> Cast a pointer returned by `__builtin_operator_new` to its appropriate type. Usable transitively within `allocate`.

Additionally, directly within any function defined in namespace `std`[1], the following is permitted in constant expressions:

`new (p) T(...)`

> *Requires:* `p` is the result of casting a pointer to an object of type `T` to `void*`.

[1] … and we should probably just permit this in constant expressions in general.