Making std::unique_ptr constexpr

std::unique_ptr is currently not constexpr friendly. With the loosening of requirements on constexpr in [P0784R10] and the ability to use new and delete in a constexpr-context, we should also provide a constexpr std::unique_ptr. Without it, users have to fall back to the pre-C++11 area and manually manage the memory. A non-constexpr unique_ptr also reduces the use-cases where users can benefit from the dual nature of constexpr, having the same code that runs at compile- and run-time.

There is no reason that the code below does not compile and users are forced into manually managing the memory.

```cpp
constexpr auto fun() {
    auto p = std::make_unique<int>(4);
    return *p;
}

int main () {
    constexpr auto i = fun();
    static_assert(4 == i);
}
Listing 1.1: unique_ptr test case 1: make_unique

2 Implementation

This proposal was implemented in a fork of libc++ from the author [GHUPImpl]. The only issue that was encountered was that the comparisons $<$, $<=$, $>$, $>$= lead to an error with Clang:

```
note: comparison has unspecified value
```

which makes the code not a constant expression. Below is a simplified version of the code triggering the error (online: https://godbolt.org/z/cqadjr):

```
#include <functional>
constexpr bool f() {
  int* a = new int{4};
  int* b = new int{5};
  return std::less<int*>(a, b);
}

int main() {
  constexpr bool b = f();
  return b == true;
}
```

Listing 2.1: Simplified issue in unique_ptr when using comparisons

3 What about other smart pointers

The implementation in [GHUPImpl] also covers a partial shared_ptr and make_shared. The approach was to get the following code to compile and run:

```
#include <memory>
#include <iostream>
constexpr auto fun() {
  std::shared_ptr<int> p{new int{4}};
  return *p;
}
auto test() {
  std::shared_ptr<int> p{new int{4}};
  return p;
}
```
int main() {
    constexpr auto i = fun();
    static_assert(i == 4);
    auto s = test();
    std::cout << *s << '\n';
}

Listing 3.1: shared_ptr test case 2: using make_shared.

The attempt was brute-force, compile it and add constexpr to all the methods Clang complained about not being usable in a constant expression. For unique_ptr that approach worked well. For shared_ptr it stopped working when the following allocation happened (https://git.io/Jkxnm#L3702):

Listing 3.2: Object _CntrlBlk cannot be used in a constant expression

The error was:

note: non-literal type '_CntrlBlk' (aka '_shared_ptr_pointer<int *, / _shared_ptr_default_delete<int, int>, allocator<int>>') cannot be used in a / constant expression

The cause of the error was from the atomics a shared_ptr needs internally in the control block. The approach was to wrap all uses of atomics with std::is_constant_evaluated (see https://git.io/Jkxnm#L3136 for an example). In one case, a wrapper was needed (see https://git.io/JkAFz#L3257). __release_weak has the implementation in memory.cpp presumably to hide some atomic includes. The newly introduced wrapper uses std::is_constant_evaluated to switch between constexpr and run-time.

The next issue was the following:

memory:1581:13: note: 'std::allocator<...>::deallocate' used to delete pointer to / object allocated with 'new'
    : operator delete(__p);
We are looking at a variation of the first test-case 3.1, this time the `shared_ptr` is created and a pre-allocated object is passed to the constructor:

```cpp
#include <memory>
#include <iostream>

constexpr auto fun() {
    std::shared_ptr<int> p{new int{4}};
    return *p;
}

test() {
    std::shared_ptr<int> p{new int{4}};
    return p;
}

int main() {
    constexpr auto i = fun();
    static_assert(i == 4);
    auto s = test();
    std::cout << *s << '\n';
}
```

Listing 3.3: shared_ptr test case 2.

The implementation of libc++ uses `allocator::deallocate` to free the memory in `__on_zero_shared_weak` (see https://git.io/lknmm#L3330), which is a specialization for the case when a `shared_ptr` can have a custom deleter, like when it is created directly by its constructor with pre-allocated memory. However, in that case, the memory was previously allocated with `new` by a user. A simplified example of the situation is the following (https://godbolt.org/z/oPG8Ea):

```cpp
#include <memory>

constexpr auto fun() {
    int* i = new int{4};
    std::allocator<int> a{};
    a.deallocate(i, 1);
    return 0;
}

int main() {
    constexpr auto f = fun();
```
Listing 3.4: Reduced example of a allocation with new and deallocation with std::allocator

Interestingly GCC has no issue with that code, while Clang rejects it. The wording in [N4868] [allocator.members] p6 says that deallocate must be called with memory previously allocated with allocate. The implementation of Clang seems to be the correct one. It further seems that the constant evaluation path did reveal UB in libc++. Coming back to making shared_ptr constexpr. The change in libc++ was using delete in the case described instead of referring to allocator.

After sprinkling some more constexpr in the minimal examples 3.1 and 3.3 did successfully compile and run.

3.1 What about the missing atomics

With the implementation as provided, a constexpr shared_ptr does not use atomics to maintain the internal count. Is this an issue? The author thinks no. Currently, there is no support for concurrency in a constant expression. Thus the absence of atomics is not observable to users. Should the language allow concurrency at some point at compile-time, the now missing atomic support will likely be available, and we can build a constexpr shared_ptr with atomics.

3.2 Further steps

A dedicated paper is planned to propose a constexpr shared_ptr.

4 Proposed wording

This wording is base on the working draft [N4868].

Change in [memory.syn] 20.11.1:

```cpp
// 20.11.1, class template unique_ptr
template<class T, class... Args>
constexpr unique_ptr<T> make_unique(Args&&... args);

template<class T>
constexpr unique_ptr<T> make_unique(size_t n);

template<class T, class... Args>
unspecified make_unique(Args&&...) = delete;

template<class T>
constexpr unique_ptr<T> make_unique_for_overwrite();

template<class T>
constexpr unique_ptr<T> make_unique_for_overwrite(size_t n);

template<class T, class... Args>
unspecified make_unique_for_overwrite(Args&&...) = delete;
```

Change in [unique.ptr.single.general] 20.11.3.1:
// 20.11.1.3.2, constructors
constexpr unique_ptr() noexcept;
constexpr explicit unique_ptr(pointer p) noexcept;
constexpr unique_ptr(pointer p, @seebelow@ d1) noexcept;
constexpr unique_ptr(pointer p, @seebelow@ d2) noexcept;
constexpr unique_ptr(unique_ptr&& u) noexcept;
constexpr unique_ptr(nullptr_t) noexcept;
template<class U, class E>
  __constexpr unique_ptr(unique_ptr<U, E> && u) noexcept;

// 20.11.1.3.3, destructor
constexpr ~unique_ptr();

// 20.11.1.3.4, assignment
constexpr unique_ptr& operator=(unique_ptr&& u) noexcept;
template<class U, class E>
  __constexpr unique_ptr& operator=(unique_ptr<U, E>&& u) noexcept;
constexpr unique_ptr& operator=(nullptr_t) noexcept;

// 20.11.1.3.5, observers
constexpr add_lvalue_reference_t<T> operator*() const;
constexpr pointer operator->() const noexcept;
constexpr pointer get() const noexcept;
constexpr deleter_type& get_deleter() noexcept;
constexpr const deleter_type& get_deleter() const noexcept;
constexpr explicit operator bool() const noexcept;

// 20.11.1.3.6, modifiers
constexpr pointer release() noexcept;
constexpr void reset(pointer p = pointer()) noexcept;
constexpr void swap(unique_ptr& u) noexcept;

Change in [unique.ptr.runtime.general] 20.11.4.1:

// 20.11.4.2, constructors
constexpr unique_ptr() noexcept;
template<class U> constexpr explicit unique_ptr(U p) noexcept;
template<class U> constexpr unique_ptr(U p, see below d) noexcept;
template<class U> constexpr unique_ptr(U p, see below d) noexcept;
constexpr unique_ptr(unique_ptr&& u) noexcept;
template<class U, class E>
  __constexpr unique_ptr(unique_ptr<U, E>&& u) noexcept;
constexpr unique_ptr(nullptr_t) noexcept;

// destructor
constexpr ~unique_ptr();

// assignment
constexpr unique_ptr& operator=(unique_ptr&& u) noexcept;
template<class U, class E>
  __constexpr unique_ptr& operator=(unique_ptr<U, E>&& u) noexcept;
constexpr unique_ptr& operator=(nullptr_t) noexcept;

// 20.11.4.4, observers
```cpp
constexpr T& operator[](size_t i) const;
constexpr pointer get() const noexcept;
constexpr deleter_type& get_deleter() noexcept;
constexpr deleter_type& get_deleter() const noexcept;
constexpr explicit operator bool() const noexcept;
```

// 20.11.1.4.5, modifiers
```cpp
constexpr pointer release() noexcept;
template<class U> constexpr void reset(U p) noexcept;
constexpr void reset(nullptr_t = nullptr) noexcept;
constexpr void swap(unique_ptr& u) noexcept;
```

In [version.syn], add the new feature test macro `__cpp_lib_constexpr_unique_ptr` with the corresponding value for header `<memory>`.

In [version.syn], add the new feature test macro `__cpp_lib_constexpr_make_unique` with the corresponding value for header `<memory>`.

5 Acknowledgements

Thanks to Ville Voutilainen for encouraging me to also look into `shared_ptr` and reviewing a draft of this paper.

Bibliography

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p0784r7.html


[GHUPImpl] Andreas Fertig: “libc++ constexpr unique_ptr implementation on GitHub”.  
https://github.com/andreasfertig/llvm-project/tree/af-constexprUniquePtr