### 1 Introduction

`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);
}
```
2 Implementation

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

```cpp
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):

```cpp
#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

### 2.1 What about relational operators?

The paper leaves the non-nullptr versions untouched. Except for the nullptr-versions the result for such an comparison is unspecified (see [expr.rel] p4.3). This seems like a general decision for LEWG how to treat such functions.

### 2.2 What about make_unique_for_overwrite?

During the presentation to LEWG in May 2021, the question was raised whether make_unique_for_overwrite is possible to implement, and those should stay in the paper. Since the adoption of [P1331R2] in
C++20 default initialization is allowed in `constexpr`-functions. Implementing `make_unique_for_overwrite` therefore is no issue.

The following code demonstrates this using Clang (https://godbolt.org/z/arebKdPvh):

```cpp
template<typename T>
class unique_ptr {
 public:
 constexpr unique_ptr(T* ptr) : _data{ptr} {}
 constexpr ~unique_ptr() { delete _data; }
 constexpr T* get() { return _data; }

 private:
 T* _data;
};

template<typename _Tp>
constexpr unique_ptr<_Tp> make_unique_for_overwrite() {
 return unique_ptr<_Tp>(new _Tp);
}

constexpr bool Fun() {
 auto x = make_unique_for_overwrite<int>();
 *=x.get() = 0; // without this init the next line causes an error
 (*x.get())++;
 return true;
}

int main() {
 constexpr auto v = Fun();
}
```

Listing 2.2: Minimal implementation of `make_unique_for_overwrite`

### 3 What about other smart pointers

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

```cpp
#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):

```
template<class _Tp>
template<class _Yp>
_LIBCPP_CONSTEXPR_AFTER_CXX20 shared_ptr<_Tp>::shared_ptr(_Yp* __p,
    typename enable_if<__compatible_with<_Yp, element_type>::value, __nat>::type)
    __ptr_(__p) {
    unique_ptr<_Yp> __hold(__p);
    typedef typename __shared_ptr_default_allocator<_Yp>::type _AllocT;
    typedef __shared_ptr_pointer<_Yp*, __shared_ptr_default_delete<_Tp, _Yp>, _AllocT> _CntrlBlk;
    __cntrl_ = new _CntrlBlk(__p, __shared_ptr_default_delete<_Tp, _Yp>(), _AllocT());
    __hold.release();
    __enable_weak_this(__p, __p);
}
```

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'
```
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;
}

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 3.3: 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/Jkxnm#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()
```
Listing 3.4: Reduced example of an allocation with `new` and deallocation with `std::allocator`.

```cpp
constexpr auto f = fun();
```

Interestingly GCC has no issue with that code, while Clang rejects it. The wording in [?] [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 Polls

#### 4.1 LEWG 2021 May 26 (virtual)

*Make unique_ptr constexpr as outlined in P2273 (not including comparisons for the purpose of this poll).*

<table>
<thead>
<tr>
<th>Strongly Favor</th>
<th>Weakly Favor</th>
<th>Neutral</th>
<th>Weakly Against</th>
<th>Strongly Against</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>10</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Attendance: 25
Outcome: Consensus in favor (no dissent)

*Make the ordered comparison operators taking two unique_ptrs constexpr.*

<table>
<thead>
<tr>
<th>Strongly Favor</th>
<th>Weakly Favor</th>
<th>Neutral</th>
<th>Weakly Against</th>
<th>Strongly Against</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>13</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Attendance: 25
Outcome: Strongly don’t care (no consensus either way)

---

Explore making `shared_ptr` and `make_shared constexpr`.

<table>
<thead>
<tr>
<th>Strongly Favor</th>
<th>Weakly Favor</th>
<th>Neutral</th>
<th>Weakly Against</th>
<th>Strongly Against</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7</td>
<td>4</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

Attendance: 24
Outcome: Weak consensus
WA: Not worth the time.

5 Proposed wording

This wording is base on the working draft [N4892].

Direction to the editor: please apply `constexpr` to the corresponding declarations in the detailed specification.

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;

template<class T, class D>
  __constexpr void swap(unique_ptr<T, D>& x, unique_ptr<T, D>& y) noexcept;

template<class T, class D>
  __constexpr bool operator==(const unique_ptr<T, D>& x, nullptr_t) noexcept;

template<class T, class D>
  __constexpr bool operator<(const unique_ptr<T, D>& x, nullptr_t);

template<class T, class D>
  __constexpr bool operator<(nullptr_t, const unique_ptr<T, D>& y);

template<class T, class D>
  __constexpr bool operator>(const unique_ptr<T, D>& x, nullptr_t);
```

```cpp
template<class T, class D>
  __constexpr bool operator>(nullptr_t, const unique_ptr<T, D>& y);
```
constexpr bool operator>(nullptr_t, const unique_ptr<T, D>& y);

template<class T, class D>
constexpr bool operator<=(const unique_ptr<T, D>& x, nullptr_t);

template<class T, class D>
constexpr bool operator<=(nullptr_t, const unique_ptr<T, D>& y);

template<class T, class D>
constexpr bool operator>=(const unique_ptr<T, D>& x, nullptr_t);

template<class T, class D>
constexpr bool operator>=(nullptr_t, const unique_ptr<T, D>& y);

template<class T, class D>
requires three_way_comparable<typename unique_ptr<T, D>::pointer>
compare_three_way_result_t<typename unique_ptr<T, D>::pointer>
constexpr operator<=>(const unique_ptr<T, D>& x, nullptr_t);

Change in [unique.ptr.dltr.dflt] 20.11.1.2:

namespace std {
  template<class T> struct default_delete {
    constexpr default_delete() noexcept = default;
    template<class U> constexpr default_delete(const default_delete<U>&) noexcept;
    constexpr void operator()(T* p) const;
  };
}

Change in [unique.ptr.dltr.dflti] 20.11.1.3:

namespace std {
  template<class T> struct default_delete<T[]> {
    constexpr default_delete() noexcept = default;
    template<class U> constexpr default_delete(const default_delete<U>&) noexcept;
    template<class U> constexpr void operator()(U* ptr) const;
  };
}

Change in [unique.ptr.single.general] 20.11.1.3.1:

// 20.11.1.3.2, constructors
constexpr unique_ptr() noexcept;
constexpr explicit unique_ptr(pointer p) noexcept;
constexpr unique_ptr(pointer p, see below d1) noexcept;
constexpr unique_ptr(pointer p, see below 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;
```cpp
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] 20.11.1.4.1:

// 20.11.1.4.2, 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(U p, unique_ptr&& u) noexcept;
template<class U, class E>
  constexpr unique_ptr(U p, unique_ptr&& 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;

// 20.11.1.4.4, observers
constexpr T& operator[](size_t i) const;
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;

// 20.11.1.4.5, modifiers
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;

Modify [version.syn]

#define __cpp_lib_constexpr_memory 201811L 202104 // also in <memory>
```
6 Acknowledgements

Thanks to Ville Voutilainen for encouraging me to also look into `shared_ptr` and reviewing a draft of this paper. Thanks to Barry Revzin and Tim Song for their feedback on Ro. Thanks to Tim Song for reviewing the wording.

7 Revision History

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2020-11-27</td>
<td>Initial draft</td>
</tr>
<tr>
<td>1</td>
<td>2021-04-06</td>
<td>- No new feature test macros, bump existing one.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Make <code>default_deleter</code>, swap and comparisons against <code>nullptr</code> constexpr as well.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Add instructor to editor regarding corresponding declarations.</td>
</tr>
<tr>
<td>2</td>
<td>2021-07-05</td>
<td>- Added poll results.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Clarified on <code>make_unique_for_overwrite</code>.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Rebased wording on [N4892].</td>
</tr>
</tbody>
</table>

Bibliography

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


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