

Making std::unique_ptr constexpr

Document Number: P2273 R3
Date: 2021-11-06
Project: ISO JTC1/SC22/WG21: Programming Language C++
Reply-to: Andreas Fertig ⟨isocpp@andreasfertig.info⟩
Audience: LWG

Contents

1 Introduction ................................................................. 1
2 Implementation ............................................................ 2
  2.1 What about relational operators? ................................ 2
  2.2 What about make_unique_for_overwrite? ..................... 2
3 What about other smart pointers ..................................... 3
  3.1 What about the missing atomics ................................. 6
  3.2 Further steps ..................................................... 6
4 Polls ............................................................................ 6
  4.1 LEWG 2021 May 26 (virtual) ..................................... 6
5 Proposed wording .......................................................... 7
6 Acknowledgements ....................................................... 10
7 Revision History ........................................................... 11
Bibliography ................................................................. 11

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:

```
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](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

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
1 template<typename T>
2 class unique_ptr {
3 public:
4    constexpr unique_ptr(T* ptr)
5        : _data{ptr} {} 
6    constexpr ~unique_ptr() { delete _data; }
7    constexpr T* get() { return _data; }
8 private:
9    T* _data;
10};
11
12 template<typename _Tp>
13 constexpr unique_ptr<_Tp> make_unique_for_overwrite()
14 {
15    return unique_ptr<_Tp>(new _Tp);
16 }
17 constexpr bool Fun() { auto x = make_unique_for_overwrite<int>();
18    *x.get() = 0; // without this init the next line causes an error
19    (*x.get())++;
20    return true;
21 }
22 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 [GHUPlmpl] also covers a partial shared_ptr and make_shared. The approach was to get the following code to compile and run:

```cpp
1 #include <memory>
2 #include <iostream>
3
4 constexpr auto fun() {
5    std::shared_ptr<int> p(new int{4});
6    return *p;
7 }
8 auto test() {
9```
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):

```cpp
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:

```cpp
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:

```cpp
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()
```
 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:

```
// 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 T1, class D1, class T2, class D2>
constexpr bool operator==(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);

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

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>
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);

requires three_way_comparable<typename unique_ptr<T, D>::pointer>

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* const);
    };
}  

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.dltr.dflt] 20.11.1.2:

// 20.11.1.3.2, constructors
constexpr unique_ptr() noexcept;
constexpr explicit unique_ptr(pointer p) noexcept;
constexpr unique_ptr(pointer p, sees below d1) noexcept;
constexpr unique_ptr(pointer p, sees 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();
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(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
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;

Change in [unique.ptr.single.ctor] Constructors:
constexpr unique_ptr() noexcept;
constexpr unique_ptr(nullptr_t) noexcept;

Constraints: is_pointer_v<deleter_type> is false and is_default_constructible_v<deleter_type> is true.

Expects: D meets the DefaultConstructible requirements (Table 27), and that construction does not throw an exception.

Effects: Constructs a unique_ptr object that owns nothing, value-initializing the stored pointer and the stored deleter.

Ensures: get() == nullptr. get_deleter() returns a reference to the stored deleter.

constexpr explicit unique_ptr(pointer p) noexcept;

Constraints: is_pointer_v<deleter_type> is false and is_default_constructible_v<deleter_type> is true.

Mandates: This constructor is not selected by class template argument deduction (over.match.class.deduct).

Expects: D meets the DefaultConstructible requirements (Table 27), and that construction does not throw an exception.

Effects: Constructs a unique_ptr which owns p, initializing the stored pointer with p and value-initializing the stored deleter.

Ensures: get() == p. get_deleter() returns a reference to the stored deleter.

Change all the following paragraphs accordingly.

Modify [version.syn]

#define __cpp_lib_constexpr_memory 201811L // 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 default_deleter, swap and comparisons against nullptr 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 make_unique_for_overwrite.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Rebased wording on [N4892].</td>
</tr>
<tr>
<td>3</td>
<td>2021-11-06</td>
<td>• Adjusted formatting for LWG review.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added operator==(up, up).</td>
</tr>
</tbody>
</table>

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