std::shared_ptr

Defined in header `<memory>`

`template< class T > class shared_ptr;`	(since C++11)

`std::shared_ptr` is a smart pointer that retains shared ownership of an object through a pointer. Several `shared_ptr` objects may own the same object. The object is destroyed and its memory deallocated when either of the following happens:

- the last remaining `shared_ptr` owning the object is destroyed;
- the last remaining `shared_ptr` owning the object is assigned another pointer via `operator=` or `reset()`.

The object is destroyed using delete-expression or a custom deleter that is supplied to `shared_ptr` during construction.

A `shared_ptr` can share ownership of an object while storing a pointer to another object. This feature can be used to point to member objects while owning the object they belong to. The stored pointer is the one accessed by `get()`, the dereference and the comparison operators. The managed pointer is the one passed to the deleter when use count reaches zero.

A `shared_ptr` may also own no objects, in which case it is called *empty* (an empty `shared_ptr` may have a non-null stored pointer if the aliasing constructor was used to create it).

All specializations of `shared_ptr` meet the requirements of *CopyConstructible*, *CopyAssignable*, and *LessThanComparable* and are contextually convertible to `bool`.

All member functions (including copy constructor and copy assignment) can be called by multiple threads on different `shared_ptr` objects without additional synchronization even if these objects are copies and share ownership of the same object. If multiple threads of execution access the same `shared_ptr` object without synchronization and any of those accesses uses a non-const member function of `shared_ptr` then a data race will occur; the `std::atomic<shared_ptr>` can be used to prevent the data race.

Member types

Member type	Definition	
`element_type`	`T`	(until C++17)
	`std::remove_extent_t<T>`	(since C++17)
`weak_type` (since C++17)	`std::weak_ptr<T>`	

Member functions

(constructor)	constructs new `shared_ptr` (public member function)
(destructor)	destructs the owned object if no more `shared_ptr`s link to it (public member function)
`operator=`	assigns the `shared_ptr` (public member function)

Modifiers

`reset`	replaces the managed object (public member function)
`swap`	swaps the managed objects (public member function)

Observers

`get`	returns the stored pointer (public member function)
`operator*` **`operator->`**	dereferences the stored pointer (public member function)

operator[] (C++17)	provides indexed access to the stored array (public member function)
use_count	returns the number of `shared_ptr` objects referring to the same managed object (public member function)
unique (until C++20)	checks whether the managed object is managed only by the current `shared_ptr` object (public member function)
operator bool	checks if the stored pointer is not null (public member function)
owner_before	provides owner-based ordering of shared pointers (public member function)
owner_hash (C++26)	provides owner-based hashing of shared pointers (public member function)
owner_equal (C++26)	provides owner-based equal comparison of shared pointers (public member function)

Non-member functions

make_shared **make_shared_for_overwrite** (C++20)	creates a shared pointer that manages a new object (function template)
allocate_shared **allocate_shared_for_overwrite** (C++20)	creates a shared pointer that manages a new object allocated using an allocator (function template)
static_pointer_cast **dynamic_pointer_cast** **const_pointer_cast** **reinterpret_pointer_cast** (C++17)	applies `static_cast`, `dynamic_cast`, `const_cast`, or `reinterpret_cast` to the stored pointer (function template)
get_deleter	returns the deleter of specified type, if owned (function template)
operator== **operator!=** (removed in C++20) **operator<** (removed in C++20) **operator<=** (removed in C++20) **operator>** (removed in C++20) **operator>=** (removed in C++20) **operator<=>** (C++20)	compares with another `shared_ptr` or with `nullptr` (function template)
operator<<(std::shared_ptr)	outputs the value of the stored pointer to an output stream (function template)
std::swap(std::shared_ptr) (C++11)	specializes the `std::swap` algorithm (function template)
std::atomic_is_lock_free(std::shared_ptr) **std::atomic_load**(std::shared_ptr) **std::atomic_load_explicit**(std::shared_ptr) **std::atomic_store**(std::shared_ptr) **std::atomic_store_explicit**(std::shared_ptr) **std::atomic_exchange**(std::shared_ptr) **std::atomic_exchange_explicit**(std::shared_ptr) **std::atomic_compare_exchange_weak**(std::shared_ptr) **std::atomic_compare_exchange_strong**(std::shared_ptr) **std::atomic_compare_exchange_weak_explicit**(std::shared_ptr) **std::atomic_compare_exchange_strong_explicit**(std::shared_ptr) (deprecated in C++20) (removed in C++26)	specializes atomic operations for `std::shared_ptr` (function template)

Helper classes

std::atomic<std::shared_ptr> (C++20)	atomic shared pointer (class template specialization)
std::hash<std::shared_ptr> (C++11)	hash support for **`std::shared_ptr`** (class template specialization)

Deduction guides (since C++17)

Notes

The ownership of an object can only be shared with another `shared_ptr` by copy constructing or copy assigning its value to another `shared_ptr`. Constructing a new `shared_ptr` using the raw underlying pointer owned by another `shared_ptr` leads to undefined behavior.

`std::shared_ptr` may be used with an incomplete type T. However, the constructor from a raw pointer (`template<class Y> shared_ptr(Y*)`) and the `template<class Y> void reset(Y*)` member function may only be called with a pointer to a complete type (note that `std::unique_ptr` may be constructed from a raw pointer to an incomplete type).

The T in `std::shared_ptr<T>` may be a function type: in this case it manages a pointer to function, rather than an object pointer. This is sometimes used to keep a dynamic library or a plugin loaded as long as any of its functions are referenced:

```
void del(void(*)()) {}

void fun() {}

int main()
{
    std::shared_ptr<void()> ee(fun, del);
    (*ee)();
}
```

Implementation notes

In a typical implementation, `shared_ptr` holds only two pointers:

- the stored pointer (one returned by `get()`);
- a pointer to *control block*.

The control block is a dynamically-allocated object that holds:

- either a pointer to the managed object or the managed object itself;
- the deleter (type-erased);
- the allocator (type-erased);
- the number of `shared_ptr`s that own the managed object;
- the number of `weak_ptr`s that refer to the managed object.

When `shared_ptr` is created by calling `std::make_shared` or `std::allocate_shared`, the memory for both the control block and the managed object is created with a single allocation. The managed object is constructed in-place in a data member of the control block. When `shared_ptr` is created via one of the `shared_ptr` constructors, the managed object and the control block must be allocated separately. In this case, the control block stores a pointer to the managed object.

The pointer held by the `shared_ptr` directly is the one returned by `get()`, while the pointer/object held by the control block is the one that will be deleted when the number of shared owners reaches zero. These pointers are not necessarily equal.

The destructor of `shared_ptr` decrements the number of shared owners of the control block. If that counter reaches zero, the control block calls the destructor of the managed object. The control block does not deallocate itself until the `std::weak_ptr` counter reaches zero as well.

In existing implementations, the number of weak pointers is incremented ([1] (https://stackoverflow.com/questions/43297517/stdshared-ptr-internals-weak-count-more-than-expected) , [2] (https://www.reddit.com/r/cpp/comments/3eia29/stdshared_ptrs_secret_constructor/ctfeh1p)) if there is a shared pointer to the same control block.

To satisfy thread safety requirements, the reference counters are typically incremented using an equivalent of `std::atomic::fetch_add` with `std::memory_order_relaxed` (decrementing requires stronger ordering to safely destroy the control block).

Example

Run this code

```
#include <chrono>
#include <iostream>
#include <memory>
#include <mutex>
#include <thread>

using namespace std::chrono_literals;

struct Base
{
    Base() { std::cout << "Base::Base()\n"; }

    // Note: non-virtual destructor is OK here
    ~Base() { std::cout << "Base::~Base()\n"; }
};

struct Derived : public Base
{
    Derived() { std::cout << "Derived::Derived()\n"; }

    ~Derived() { std::cout << "Derived::~Derived()\n"; }
};

void print(auto rem, std::shared_ptr<Base> const& sp)
{
    std::cout << rem << "\n\tget() = " << sp.get()
              << ", use_count() = " << sp.use_count() << '\n';
}

void thr(std::shared_ptr<Base> p)
{
    std::this_thread::sleep_for(987ms);
    std::shared_ptr<Base> lp = p; // thread-safe, even though the
                                  // shared use_count is incremented
    {
        static std::mutex io_mutex;
        std::lock_guard<std::mutex> lk(io_mutex);
        print("Local pointer in a thread:", lp);
    }
}

int main()
{
    std::shared_ptr<Base> p = std::make_shared<Derived>();

    print("Created a shared Derived (as a pointer to Base)", p);

    std::thread t1{thr, p}, t2{thr, p}, t3{thr, p};
    p.reset(); // release ownership from main

    print("Shared ownership between 3 threads and released ownership from main:", p);

    t1.join();
    t2.join();
    t3.join();

    std::cout << "All threads completed, the last one deleted Derived.\n";
}
```

Possible output:

```
Base::Base()
Derived::Derived()
Created a shared Derived (as a pointer to Base)
        get() = 0x118ac30, use_count() = 1
```

```
Shared ownership between 3 threads and released ownership from main:
        get() = 0, use_count() = 0
Local pointer in a thread:
        get() = 0x118ac30, use_count() = 5
Local pointer in a thread:
        get() = 0x118ac30, use_count() = 4
Local pointer in a thread:
        get() = 0x118ac30, use_count() = 2
Derived::~Derived()
Base::~Base()
All threads completed, the last one deleted Derived.
```

Example

Run this code

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

struct MyObj
{
    MyObj() { std::cout << "MyObj constructed\n"; }

    ~MyObj() { std::cout << "MyObj destructed\n"; }
};

struct Container : std::enable_shared_from_this<Container> // note: public inheritance
{
    std::shared_ptr<MyObj> memberObj;

    void CreateMember() { memberObj = std::make_shared<MyObj>(); }

    std::shared_ptr<MyObj> GetAsMyObj()
    {
        // Use an alias shared ptr for member
        return std::shared_ptr<MyObj>(shared_from_this(), memberObj.get());
    }
};

#define COUT(str) std::cout << '\n' << str << '\n'

#define DEMO(...) std::cout << #__VA_ARGS__ << " = " << __VA_ARGS__ << '\n'

int main()
{
    COUT("Creating shared container");
    std::shared_ptr<Container> cont = std::make_shared<Container>();
    DEMO(cont.use_count());
    DEMO(cont->memberObj.use_count());

    COUT("Creating member");
    cont->CreateMember();
    DEMO(cont.use_count());
    DEMO(cont->memberObj.use_count());

    COUT("Creating another shared container");
    std::shared_ptr<Container> cont2 = cont;
    DEMO(cont.use_count());
    DEMO(cont->memberObj.use_count());
    DEMO(cont2.use_count());
    DEMO(cont2->memberObj.use_count());

    COUT("GetAsMyObj");
    std::shared_ptr<MyObj> myobj1 = cont->GetAsMyObj();
    DEMO(myobj1.use_count());
    DEMO(cont.use_count());
    DEMO(cont->memberObj.use_count());
    DEMO(cont2.use_count());
    DEMO(cont2->memberObj.use_count());
    COUT("Copying alias obj");
    std::shared_ptr<MyObj> myobj2 = myobj1;
    DEMO(myobj1.use_count());
    DEMO(myobj2.use_count());
    DEMO(cont.use_count());
    DEMO(cont->memberObj.use_count());
    DEMO(cont2.use_count());
    DEMO(cont2->memberObj.use_count());

    COUT("Resetting cont2");
    cont2.reset();
    DEMO(myobj1.use_count());
    DEMO(myobj2.use_count());
    DEMO(cont.use_count());
    DEMO(cont->memberObj.use_count());

    COUT("Resetting myobj2");
    myobj2.reset();
    DEMO(myobj1.use_count());
    DEMO(cont.use_count());
    DEMO(cont->memberObj.use_count());

    COUT("Resetting cont");
    cont.reset();
    DEMO(myobj1.use_count());
    DEMO(cont.use_count());
}
```

Output:

```
Creating shared container
cont.use_count() = 1
cont->memberObj.use_count() = 0

Creating member
MyObj constructed
cont.use_count() = 1
cont->memberObj.use_count() = 1

Creating another shared container
cont.use_count() = 2
cont->memberObj.use_count() = 1
cont2.use_count() = 2
cont2->memberObj.use_count() = 1

GetAsMyObj
myobj1.use_count() = 3
cont.use_count() = 3
cont->memberObj.use_count() = 1
cont2.use_count() = 3
cont2->memberObj.use_count() = 1

Copying alias obj
myobj1.use_count() = 4
myobj2.use_count() = 4
cont.use_count() = 4
cont->memberObj.use_count() = 1
cont2.use_count() = 4
cont2->memberObj.use_count() = 1

Resetting cont2
myobj1.use_count() = 3
myobj2.use_count() = 3
cont.use_count() = 3
cont->memberObj.use_count() = 1

Resetting myobj2
myobj1.use_count() = 2
cont.use_count() = 2
cont->memberObj.use_count() = 1
```

```
Resetting cont
myobj1.use_count() = 1
cont.use_count() = 0
MyObj destructed
```

See also

`unique_ptr` (C++11)	smart pointer with unique object ownership semantics (class template)
`weak_ptr` (C++11)	weak reference to an object managed by **`std::shared_ptr`** (class template)