

Unreal Smart Pointer Library

Custom implementation of shared pointers, including weak pointers and non-nullable shared references.




The **Unreal Smart Pointer Library** is a custom implementation of C++11 smart pointers designed to ease the burden of memory allocation and tracking. This implementation includes the industry standard **Shared Pointers**, **Weak Pointers**, and **Unique Pointers**. It also adds **Shared References** which act like non-nullable Shared Pointers. These classes cannot be used with the `UObject` system because Unreal Objects use a separate memory-tracking system that is better-tuned for game code.

Smart Pointer Types

Smart Pointers can affect the lifespan of the object they contain or reference. Different Smart Pointers have different limitations and effects on the object. The following table can be used to help determine when it is appropriate to use each type of Smart Pointer:

Smart Pointer Type	Use Case
Shared Pointers (<code>TSharedPtr</code>)	A Shared Pointer owns the object it references, indefinitely preventing deletion of that object, and ultimately handling its deletion when no Shared Pointer or Shared Reference (see below) references it. A Shared Pointer can be empty, meaning it doesn't reference any object. Any non-null Shared Pointer can produce a Shared Reference to the object it references.
Shared References (<code>TSharedRef</code>)	A Shared Reference acts like a Shared Pointer, in the sense that it owns the object it references. They differ with regard to null objects; Shared References must always reference a non-null object. Because Shared Pointers don't have that restriction, a Shared Reference can always be converted to a Shared Pointer, and that Shared Pointer is guaranteed to

Smart Pointer Type	Use Case
	reference a valid object. Use Shared References when you want a guarantee that the referenced object is non-null, or if you want to indicate shared object ownership.
Weak Pointers (<code>TWeakPtr</code>)	Weak Pointers are similar to Shared Pointers, but do not own the object they reference, and therefore do not affect its lifecycle. This property can be very useful, as it breaks reference cycles, but it also means that a Weak Pointer can become null at any time, without warning. For this reason, a Weak Pointer can produce a Shared Pointer to the object it references, ensuring programmers safe access to the object on a temporary basis.
Unique Pointers (<code>TUniquePtr</code>)	A Unique Pointer solely and explicitly owns the object it references. Since there can only be one Unique Pointer to a given resource, Unique Pointers can transfer ownership, but cannot share it. Any attempts to copy a Unique Pointer will result in a compile error. When a Unique Pointer goes out of scope, it will automatically delete the object it references.



Making a Shared Pointer or Shared Reference to an object that a Unique Pointer references is dangerous. This will not suspend the Unique Pointer's behavior of deleting the object upon its own destruction, even though other Smart Pointers still reference it. Similarly, you should not make a Unique Pointer to an object that is referenced by a Shared Pointer or Shared Reference.

|

Benefits of Smart Pointers

Benefit	Description
Prevents memory leaks	Smart Pointers (other than Weak Pointers) automatically delete objects when there are no more shared references.
Weak referencing	Weak Pointers break reference cycles and prevent dangling pointers.
Optional Thread safety	The Unreal Smart Pointer Library includes thread-safe code that manages reference counting across multiple threads. Thread safety can be traded out for better performance if it isn't needed.
Runtime safety	Shared References are never null and can always be dereferenced.
Confers intent	You can easily tell an object owner from an observer.

Benefit	Description
Memory	Smart Pointers are only twice the size of a C++ pointer in 64-bit (plus a shared 16-byte reference controller). The exception to this is Unique Pointers, which are the same size as C++ pointers.

Helper Classes and Functions

The Unreal Smart Pointer Library provides several helper classes and functions to make using Smart Pointers easier and more intuitive.

Helper	Description
Classes	
<code>TSharedFromThis</code>	Deriving your class from <code>TSharedFromThis</code> adds the <code>AsShared</code> or <code>SharedThis</code> functions. These functions enable you to acquire a <code>TSharedRef</code> to your object.
Functions	
<code>MakeShared</code> and <code>MakeShareable</code>	Creates a Shared Pointer from a regular C++ pointer. <code>MakeShared</code> allocates a new object instance and the reference controller in a single memory block, but requires the object to offer a public constructor. <code>MakeShareable</code> is less efficient, but works even if the object's constructor is private, enables you to take ownership of an object you didn't create, and supports customized behavior when deleting the object.
<code>StaticCastSharedRef</code> and <code>StaticCastSharedPtr</code>	Static cast utility function, typically used to downcast to a derived type.
<code>ConstCastSharedRef</code> and <code>ConstCastSharedPtr</code>	Converts a <code>const</code> Smart Reference or Smart Pointer to a <code>mutable</code> Smart Reference or Smart Pointer, respectively.

Smart Pointer Implementation Details

Smart Pointers in the Unreal Smart Pointer Library all share some general characteristics in terms of functionality and efficiency.

Speed

Always keep performance in mind when considering using Smart Pointers. Smart Pointers are well-suited for certain high-level systems, resource management, or tools programming. However, some Smart Pointer types are slower than raw C++ pointers, and this overhead makes them less useful in low-level engine code, such as rendering.

Some of the general performance benefits of Smart Pointers are:

- All operations are constant time.
- Dereferencing most Smart Pointers is just as fast as raw C++ pointers (in shipping builds).
- Copying Smart Pointers never allocates memory.
- Thread-safe Smart Pointers are lockless.

Performance drawbacks of Smart Pointers include:

- Creating and copying Smart Pointers involves more overhead than creating and copying raw C++ pointers.
- Maintaining reference counts adds cycles to basic operations.
- Some Smart Pointers use more memory than raw C++ pointers.
- There are two heap allocations for reference controllers. Using `MakeShared` instead of `MakeShareable` avoids the second allocation, and can improve performance.

Intrusive Accessors

Shared pointers are non-intrusive, which means the object does not know whether or not a Smart Pointer owns it. This is usually acceptable, but there may be cases in which you want to access the object as a Shared Reference or Shared Pointer. To do this, derive the object's class from `TSharedFromThis`, using the object's class as the template parameter.

`TSharedFromThis` provides two functions, `AsShared` and `SharedThis`, that can convert the object to a Shared Reference (and from there, to a Shared Pointer). This can be useful with class factories that always return Shared References, or when you need to pass your object to a system that expects a Shared Reference or Shared Pointer. `AsShared` will return your class as the type originally passed as the template argument to `TSharedFromThis`, which may be a parent type to the calling object, while `SharedThis` will derive the type directly from this and return a Smart Pointer referencing an object of that type. The following example code demonstrates both functions:

```
class FRegistryObject;
class FMyBaseClass: public TSharedFromThis<FMyBaseClass>
{
    virtual void RegisterAsBaseClass(FRegistryObject* RegistryObject)
    {
        // Access a shared reference to 'this'.
        // We are directly inherited from <TSharedFromThis> , so AsShared() and SharedThis() are available.
        TSharedRef<FMyBaseClass> ThisAsSharedRef = AsShared();
        // RegistryObject expects a TSharedRef<FMyBaseClass>, or a TSharedPtr<FMyBaseClass>.
        RegistryObject->Register(ThisAsSharedRef);
    }
};

class FMyDerivedClass: public FMyBaseClass
```

```

{
    virtual void Register(FRegistryObject* RegistryObject) override
    {
        // We are not directly inherited from TSharedFromThis<>, so AsShared() and Sl
        // AsShared() will return the type originally specified in TSharedFromThis<>
        // SharedThis(this) will return a TSharedRef with the type of 'this' - TShar
        // The SharedThis() function is only available in the same scope as the 'this
        TSharedRef<FMyDerivedClass> AsSharedRef = SharedThis(this);
        // RegistryObject will accept a TSharedRef<FMyDerivedClass> because FMyDeriv
        RegistryObject->Register(ThisAsSharedRef);
    }
};
class FRegistryObject
{
    // This function will accept a TSharedRef or TSharedPtr to FMyBaseClass or any
    void Register(TSharedRef<FMyBaseClass>);
};

```

 Copy full snippet

Do not call `AsShared` or `SharedThis` from constructors, since the shared reference is not initialized at that time and will cause a crash or assert.

Casting

You can cast Shared Pointers (and Shared References) through several support functions included in the Unreal Smart Pointer Library. Up-casting is implicit, as with C++ pointers. You can const cast with the `ConstCastSharedPtr` function, and static cast (often to downcast to derived class pointers) with `StaticCastSharedPtr`. Dynamic casting is not supported, as there is no run-time type information (RTTI); static casting should be used instead, as in the following code:

```

// This assumes we validated that the FDragDropOperation is actually an FAssetDr
TSharedPtr<FDragDropOperation> Operation = DragDropEvent.GetOperation();
// We can now cast with StaticCastSharedPtr.
TSharedPtr<FAssetDragDropOp> DragDropOp = StaticCastSharedPtr<FAssetDragDropOp>(

```

 Copy full snippet

Thread Safety

By default, Smart Pointers are only safe to access on a single thread. If you need multiple threads to have access, use the thread-safe versions of Smart Pointer classes:

- `TSharedPtr<T, ESPMode::ThreadSafe>`
- `TSharedRef<T, ESPMode::ThreadSafe>`
- `TWeakPtr<T, ESPMode::ThreadSafe>`

- `TSharedFromThis<T, ESPMode::ThreadSafe>`

These thread-safe versions are a bit slower than the defaults due to atomic reference counting, but their behavior is consistent with regular C++ pointers:

- Reads and copies are always thread-safe.
- Writes and resets must be synchronized to be safe.



If you know your pointer will never be accessed by more than one thread, you can get better performance by avoiding the thread-safe versions.

Tips and Limitations

- Avoid passing data to functions as `TSharedRef` or `TSharedPtr` parameters where possible, as these will incur overhead by dereferencing and reference-counting. Instead, pass the referenced object, preferably as a `const &`.
- You can forward-declare Shared Pointers to incomplete types.
- Shared Pointers are not compatible with Unreal objects (`UObject` and its derived classes). The Engine has a separate memory management system (see [Object Handling](#) documentation) for `UObject` management, and the two systems have no overlap with each other.