Shared Pointers

Smart pointers that support shared ownership, automatic invalidation, weak references, and more.

Shared Pointers are Smart Pointers that are both strong and nullable. Shared pointers inherently include all the benefits of basic Smart Pointers in that they prevent memory leaks, dangling pointers, and pointers to uninitialized memory, but they also provide additional features, such as:

- **Shared Ownership:** Reference counting enables multiple Shared Pointers to ensure that the data object they reference will never be deleted as long as any of them still points to it.
- **Automatic Invalidation:** Volatile objects can be safely referenced without worrying about dangling pointers.
- **Weak References:** Weak Pointers can break reference cycles.
- **Indication of Intent:** Distinguishes owners (see Shared References) from observers and provides non-nullable references.

Shared pointers have some basic characteristics worth noting, including:

- Very robust syntax
- Non-intrusive (but reflection is possible)
- Thread-safe (conditionally)
- Good performance, light on memory

> Shared Pointers are similar to Shared References, the main distinction being that Shared References are not nullable and therefore always reference valid objects. When choosing between Shared References and Shared Pointers, Shared References are the preferred option unless you need an empty or nullable object.

Declaration and Initialization

Because Shared Pointers are nullable, you can initialize them with or without a data object. Here are some examples of creating new shared pointers:

```
// Create an empty shared pointer
TSharedPtr<FMyObjectType> EmptyPointer;
// Create a shared pointer to a new object
TSharedPtr<FMyObjectType> NewPointer(new FMyObjectType());
// Create a Shared Pointer from a Shared Reference
TSharedRef<FMyObjectType> NewReference(new FMyObjectType());
TSharedPtr<FMyObjectType> PointerFromReference = NewReference;
// Create a Thread-safe Shared Pointer
TSharedPtr<FMyObjectType, ESPMode::ThreadSafe> NewThreadsafePointer =
MakeShared<FMyObjectType, ESPMode::ThreadSafe>(MyArgs);

```

Copy full snippet

In the second example, `NodePtr` effectively owns the new `FMyObjectType` object, since no other Shared Pointer references that object. If `NodePtr` goes out of scope without another Shared Pointer or Shared Reference pointing to the object, the object will be destroyed.

When you copy a Shared Pointer, the system adds one reference to the object that it references.

```
// Increase the reference count of whatever object ExistingSharedPointer
references.
TSharedPtr<FMyObjectType> AnotherPointer = ExistingSharedPointer;

```

Copy full snippet

The object will persist until no more Shared Pointers (or Shared References) reference it.

You can reset a Shared Pointer with the `Reset` function, or by assigning a null pointer to them, as follows:

```
PointerOne.Reset();
PointerTwo = nullptr;
// Both PointerOne and PointerTwo now reference nullptr.

```

Copy full snippet

You can transfer the contents of one Shared Pointer to another, leaving the original Shared Pointer empty, with the `MoveTemp` (or `MoveTempIfPossible`) function:

```
// Move the contents of PointerOne over to PointerTwo. PointerOne will
reference nullptr after this.
PointerTwo = MoveTemp(PointerOne);
```

```
// Move the contents of PointerTwo over to PointerOne. PointerTwo will reference nullptr after this.
PointerOne = MoveTempIfPossible(PointerTwo);

```

Copy full snippet

> `MoveTemp` and `MoveTempIfPossible` differ only in that `MoveTemp` includes static asserts to enforce that it can only execute on a non-const left-value (lvalue).

Converting Between Shared Pointers and Shared References

Converting between Shared Pointers and Shared References is a common practice. Shared References implicitly convert to Shared Pointers, and provide the additional guarantee that the new Shared Pointer will reference a valid object. Conversion is handled by the normal syntax:

```
TSharedPtr<FMyObjectType> MySharedPointer = MySharedReference;

```

Copy full snippet

You can create a Shared Reference from a Shared Pointer with the `Shared Pointer` function, `ToSharedRef`, as long as the Shared Pointer references a non-null object. Attempting to create a Shared Reference from a null Shared Pointer will cause the program to assert.

```
// Ensure your shared pointer is valid before dereferencing to avoid a potential assertion.
if (MySharedPointer.IsValid())
{
MySharedReference = MySharedPointer.ToSharedRef();
}

```

Copy full snippet

Comparison

You can test Shared Pointers against each other for equality. In this context, equality is defined as both Shared Pointers referencing the same object.

```
TSharedPtr<FTreeNode> NodeA, NodeB;
if (NodeA == NodeB)
{
```

```
// ...
}

```

Copy full snippet

The `IsValid` function and the `bool` operator will establish whether or not the Shared Pointer references a valid object. You can also call `Get` and see if it returns a valid (non-null) object pointer.

```
if (Node.IsValid())
{
// ...
}
if (Node)
{
// ...
}
if (Node.Get() != nullptr)
{
// ...
}

```

Copy full snippet

Dereferencing and Accessing

You can dereference, call methods, and access members in the same way you would with regular C++ pointers. You should also perform null-checking as you would for any C++ pointer, by calling the `IsValid` function or using the overloaded `bool` operator, before dereferencing it.

```
// Check that Node references a valid object before dereferencing.
if (Node)
{
// Any of the following three lines of code will dereference Node and call
ListChildren on its object:
Node->ListChildren();
Node.Get()->ListChildren();
(*Node).ListChildren();
}

```

Copy full snippet

Custom Deleters

Shared Pointers and Shared References support custom deleters for the objects they reference. To run custom deletion code, provide the lambda function you wish to run as a parameter when creating your Smart Pointer, like this:

```
void DestroyMyObjectType(FMyObjectType* ObjectAboutToBeDeleted)
{
// Custom deletion code goes here.
}
// These functions create Smart Pointers with custom deleters.
TSharedRef<FMyObjectType> NewReference(new FMyObjectType(), [](FMyObjectType*
Obj){ DestroyMyObjectType(Obj); });

TSharedPtr<FMyObjectType> NewPointer(new FMyObjectType(), [](FMyObjectType*
Obj){ DestroyMyObjectType(Obj); });
```