# Weak Pointers

Smart pointers that store weak references and do not prevent their objects from being destroyed.

**Weak Pointers** store a weak reference to an object. Unlike **Shared Pointers** or **Shared References**, a Weak Pointer will not prevent destruction of the object it references.

Before accessing the object that a Weak Pointer references, you should use the `Pin` function to produce a Shared Pointer. This guarantees that the object will continue to exist while you are using it. If you only need to establish whether or not the Weak Pointer references an object, you can compare it to `nullptr` or call `IsValid` on it.

Using Weak Pointers can help confer intent — a Weak Pointer indicates an observation of the referenced object without ownership of it, and does not control its lifetime.

# Declaration, Initialization, and Assignment

You can create empty Weak Pointers, or build them from Shared References, Shared Pointers, or other Weak Pointers.

```
1  // Allocate a new data object and create a strong reference to it.
2  TSharedRef<FMyObjectType> ObjectOwner = MakeShared<FMyObjectType>();
3  // Create a weak pointer to the new data object.
4  TWeakPtr<FMyObjectType> ObjectObserver(ObjectOwner);
5
```

Copy full snippet

Weak Pointers do not prevent objects from being destroyed. In our example, resetting `ObjectOwner` will destroy the object, regardless of whether or not `ObjectObserver` is still in

scope:

```
// Assuming ObjectOwner was the only owner of its object, that object will be destroyed when ObjectOwner stops referencing it.
ObjectOwner.Reset();
// The Shared Pointer that Pin() generates will be null due to ObjectOwner referencing a null object. When treated as a bool, empty Shared Pointers evaluate to false.
if (ObjectObserver.Pin())
{
// This code will run only if ObjectOwner was not the sole owner of our object.
check(false);
}

```

Copy full snippet

Weak Pointers can be copied around safely, just like Shared Pointers, regardless of whether or not they reference a valid object:

```
TWeakPtr<FMyObjectType> AnotherObjectObserver = ObjectObserver;

```

Copy full snippet

You can reset a Weak Pointer when you are done with it:

```
// You can reset a Weak Pointer by setting it to nullptr.
ObjectObserver = nullptr;
// You can also use the Reset function.
AnotherObjectObserver.Reset();

```

Copy full snippet

# Converting To Shared Pointers

The `Pin` function creates a Shared Pointer to the Weak Pointer's object. As long as the Shared Pointer is in scope and references the object, the object will remain valid. In addition, Shared Pointers (including the one returned by the `Pin` function) can evaluate as type `bool` in conditionals, where `true` indicates a valid object. The following code pattern checks to see if the Weak Pointer references a valid object, and if so, guarantees that it will continue to be valid at least until the Shared Pointer (created by the `Pin` function) goes out of scope or is explicitly cleared.

```
// Acquire a Shared Pointer from the Weak Pointer and check that it references a valid object.
if (TSharedPtr<FMyObjectType> LockedObserver = ObjectObserver.Pin())
{
```

```
// The Shared Pointer is valid only within this scope.
// The object has been verified to exist, and the Shared Pointer prevents its deletion.
LockedObserver->SomeFunction();
}

```

Copy full snippet

# Dereferencing and Accessing

To access a Weak Pointer's object, first promote it to a Shared Pointer with the `Pin` function. You can then access it through the `Get` function on either the Shared Pointer or the Weak Pointer. This method ensures that the object remains valid while you are working with it.

# Breaking Reference Cycles

A reference cycle exists whenever two or more objects use Smart Pointers to keep strong references to each other. In these situations, the objects protect each other from deletion, as each one is always being referenced by one other object, and so neither can be deleted while the other still exists. If no outside object references either of the objects in the reference cycle, they will effectively leak. Weak Pointers can break these reference cycles, because Weak Pointers do not preserve the objects they reference. Use Weak Pointers when you want to reference objects without claiming ownership over them and potentially extending their lifespans.

# Usage Warnings

Weak Pointers are useful in cases where you don't want to guarantee that the data object will continue to exist, but this very property can be dangerous. The following situations require caution when using Weak Pointers:

- Usage as keys in Sets or Maps. Because Weak Pointers can become invalid at any time, and will not notify the container, Shared Pointers or Shared References are better-suited to act as keys. Weak Pointers are safe to use as values.
- Although Weak Pointers provide an `IsValid` function, checking `IsValid` does not guarantee that the object will remain valid for any length of time. This is especially true with thread-safe Shared Pointers, as they can become invalid at any time due to activity on another thread. The `Pin` function is the preferred method for any check that will lead to dereferencing or accessing the stored object, as the Shared Pointer that `Pin` returns will keep the object alive until your code clears it, or it goes out of scope.