TSet

TSets are a fast container class to store (usually) unique elements in a context where the order is irrelevant.

`TSet` is similar to `TMap` and `TMultiMap`, but with an important difference: Rather than associating data values with independent keys, a `TSet` uses the data value itself as the key, through an overridable function that evaluates the element. `TSet` is very fast (constant time) for adding, finding, and removing elements. By default, `TSet` does not support duplicate keys, but this behavior can be activated with a template parameter.

TSet

`TSet` is a fast container class to store unique elements in a context where order is irrelevant. For most use cases, just one parameter — the element type — is needed. However, `TSet` can be set up with different template parameters to change its behavior and make it more versatile. You can specify a derived struct based on `DefaultKeyFuncs` to provide hashing functionality, as well as permit multiple keys with the same value to exist in the set. Finally, like the other container classes, you can provide a custom memory allocator for its data storage.

Like `TArray`, `TSet` is a homogeneous container, meaning that all of its elements are strictly the same type. `TSet` is also a value type, and supports the usual copy, assignment, and destructor operations, as well as strong ownership of its elements, which are destroyed when the `TSet` is. The key type must also be a value type.

`TSet` uses hashes, which means that the `KeyFuncs` template parameter, if provided, tells the set how to determine the key from an element, how to compare two keys for equality, how to hash the key, and whether or not to permit duplicate keys. These have defaults that will return a reference to the key, use `operator==` for equality, and the non-member `GetTypeHash` function for hashing. By default, the set will not permit duplicate keys. If your key type supports these functions, it is usable as a set key without the need to provide a custom `KeyFuncs`. To write a custom `KeyFuncs`, extend the `DefaultKeyFuncs` struct.

Finally, `TSet` can take an optional allocator to control memory allocation behavior. Standard Unreal Engine 4 (UE4) allocators (such as `FHeapAllocator` and `TInlineAllocator`) cannot be used as allocators for `TSet`. Instead, `TSet` uses set allocators, which define how many hash buckets the set should use, and which standard UE4 allocators to use for element storage. See `TSetAllocator` for more information.

Unlike `TArray`, the relative order of `TSet` elements in memory is not reliable or stable, and iterating over the elements is likely to return them in a different order from the order in which they were added. Elements are also unlikely to be laid out contiguously in memory. The backing data structure of a set is a sparse array, which is an array that efficiently supports gaps between its elements. As elements are removed from the set, gaps in the sparse array will appear. Adding new elements into the array can then fill these gaps. However, even though `TSet` doesn't shuffle elements to fill gaps, pointers to set elements may still be invalidated, as the entire storage can be reallocated when it is full and new elements are added.

Creating and Filling a Set

You can create a `TSet` like this:

```
TSet<FString> FruitSet;

```

Copy full snippet

This creates an empty `TSet` that will hold `FString` data. The `TSet` will compare elements directly with `operator==`, hash them using `GetTypeHash`, and use the standard heap allocator. No memory has been allocated at this point.

The standard way to populate a set is to use the `Add` function and provide a key (element):

```
FruitSet.Add(TEXT("Banana"));
FruitSet.Add(TEXT("Grapefruit"));
FruitSet.Add(TEXT("Pineapple"));
// FruitSet == [ "Banana", "Grapefruit", "Pineapple" ]

```

Copy full snippet

While the elements are listed here in the order of insertion, there is no guarantee as to their actual order in memory. For a new set, they are likely to be in order of insertion, but as more insertions and removals happen, it becomes increasingly unlikely that new elements will appear at the end.

Since this set uses the default allocator, keys are guaranteed to be unique. The following is the result of attempting to add a duplicate key:

```
FruitSet.Add(TEXT("Pear"));
FruitSet.Add(TEXT("Banana"));
// FruitSet == [ "Banana", "Grapefruit", "Pineapple", "Pear" ]
// Note: Only one banana entry.

```

Copy full snippet

The set now contains four elements. "Pear" brought the count up to four from three, but the new "Banana" didn't change the number of elements in the set because it replaced the old "Banana" entry.

Like `TArray`, we can also use `Emplace` instead of `Add` to avoid the creation of temporaries when inserting into the set:

```
FruitSet.Emplace(TEXT("Orange"));
// FruitSet == [ "Banana", "Grapefruit", "Pineapple", "Pear", "Orange" ]

```

Copy full snippet

Here, the argument is passed directly to the constructor of the key type. This avoids the creation of a temporary `FString` for the value. Unlike `TArray`, it's only possible to emplace elements into a set with single-argument constructors.

It's also possible to insert all the elements from another set by using the `Append` function to merge them:

```
TSet<FString> FruitSet2;
FruitSet2.Emplace(TEXT("Kiwi"));
FruitSet2.Emplace(TEXT("Melon"));
FruitSet2.Emplace(TEXT("Mango"));
FruitSet2.Emplace(TEXT("Orange"));
FruitSet.Append(FruitSet2);
// FruitSet == [ "Banana", "Grapefruit", "Pineapple", "Pear", "Orange",
"Kiwi", "Melon", "Mango" ]

```

Copy full snippet

In the example above, the resulting set is equivalent to using `Add` or `Emplace` to add the elements individually. Duplicate keys from the source set will replace their counterparts in the target.

Editing UPROPERTY TSets

If you mark the `TSet` with the `UPROPERTY` macro and one of the "editable" keywords (`EditAnywhere`, `EditDefaultsOnly`, or `EditInstanceOnly`), you can add and edit elements in Unreal Editor.

```
UPROPERTY(Category = SetExample, EditAnywhere)
TSet<FString> FruitSet;
```

```

```

Copy full snippet

Iteration

Iteration over `TSets` is similar to `TArrays`. You can use the C++ ranged-for feature:

```cpp
for (auto& Elem : FruitSet)
{
FPlatformMisc::LocalPrint(
*FString::Printf(
TEXT(" \"%s\"\n"),
*Elem
)
);
}
// Output:
// "Banana"
// "Grapefruit"
// "Pineapple"
// "Pear"
// "Orange"
// "Kiwi"
// "Melon"
// "Mango"

```

Copy full snippet

You can also create iterators with the `CreateIterator` and `CreateConstIterators` functions. `CreateIterator` will return an iterator with read-write access, while `CreateConstIterator` returns a read-only iterator. In either case, you can use the `Key` and `Value` functions of these iterators to examine the elements. Printing the contents of our example "fruit" set using iterators would look like this:

```cpp
for (auto It = FruitSet.CreateConstIterator(); It; ++It)
{
FPlatformMisc::LocalPrint(
*FString::Printf(
TEXT("(%s)\n"),
*It
)
);
}

```

Copy full snippet

Queries

To find out how many elements are currently in the set, call the `Num` function.

```
int32 Count = FruitSet.Num();
// Count == 8

```

Copy full snippet

In order to determine whether or not a set contains a specific element, call the `Contains` function, as follows:

```
bool bHasBanana = FruitSet.Contains(TEXT("Banana"));
bool bHasLemon = FruitSet.Contains(TEXT("Lemon"));
// bHasBanana == true
// bHasLemon == false

```

Copy full snippet

You can use the `FSetElementId` struct to find the index of a key within the set. You can then use that index with `operator[]` to retrieve the element. Calling `operator[]` on a non-const set will return a non-const reference, and calling it on a const set will return a const reference.

```
FSetElementId BananaIndex = FruitSet.Index(TEXT("Banana"));
// BananaIndex is a value between 0 and (FruitSet.Num() - 1)
FPlatformMisc::LocalPrint(
*FString::Printf(
TEXT(" \"%s\"\n"),
*FruitSet[BananaIndex]
)
);
// Prints "Banana"

FSetElementId LemonIndex = FruitSet.Index(TEXT("Lemon"));
// LemonIndex is INDEX_NONE (-1)
FPlatformMisc::LocalPrint(
*FString::Printf(
TEXT(" \"%s\"\n"),
*FruitSet[LemonIndex]
)
); // Assert!

```

Copy full snippet

If you are unsure whether or not your set contains a key, you could check with the `Contains` function, and then use `operator[]`. However, this is suboptimal, since a successful retrieval involves two lookups on the same key. The `Find` function combines these behaviors with a single lookup. `Find` returns a pointer to the value of the element if the set contains the key, or a null pointer if it does not. Calling `Find` on a const set will cause the pointer it returns to be const as well.

```
FString* PtrBanana = FruitSet.Find(TEXT("Banana"));
FString* PtrLemon = FruitSet.Find(TEXT("Lemon"));
// *PtrBanana == "Banana"
// PtrLemon == nullptr

```

Copy full snippet

The `Array` function returns a `TArray` populated with a copy of all the elements in the `TSet`. The array that you pass in will be emptied at the beginning of the operation, so the resulting number of elements will always equal the number of elements in the set:

```
TArray<FString> FruitArray = FruitSet.Array();
// FruitArray == [
"Banana","Grapefruit","Pineapple","Pear","Orange","Kiwi","Melon","Mango" ]
(order may vary)

```

Copy full snippet

Removal

Elements can be removed by index with the `Remove` function, though this is recommended only for use while iterating through the elements. The Remove function returns the number of elements removed, and will be 0 if the key provided was not contained in the set. If a `TSet` supports duplicate keys, `Remove` will remove all matching elements.

```
FruitSet.Remove(0);
// FruitSet == [
"Grapefruit","Pineapple","Pear","Orange","Kiwi","Melon","Mango" ]

```

Copy full snippet

Removing elements can leave holes in the data structure, which you can see when visualizing the set in Visual Studio's watch window, but they have been omitted here for clarity.

```
int32 RemovedAmountPineapple = FruitSet.Remove(TEXT("Pineapple"));
// RemovedAmountPineapple == 1
// FruitSet == [ "Grapefruit","Pear","Orange","Kiwi","Melon","Mango" ]
FString RemovedAmountLemon = FruitSet.Remove(TEXT("Lemon"));
// RemovedAmountLemon == 0

```

Copy full snippet

Finally, you can remove all elements from the set with the `Empty` or `Reset` functions.

```
TSet<FString> FruitSetCopy = FruitSet;
// FruitSetCopy == [ "Grapefruit","Pear","Orange","Kiwi","Melon","Mango" ]

FruitSetCopy.Empty();
// FruitSetCopy == []

```

Copy full snippet

`Empty` and `Reset` are similar, but `Empty` can take a parameter to indicate how much slack to leave in the set, while `Reset` always leaves as much slack as possible.

Sorting

A `TSet` can be sorted. After sorting, iteration over the set will present the elements in sorted order, but this behavior is only guaranteed until the next time you modify the set. Sorting is unstable, so equivalent elements in a set that supports duplicate keys may appear in any order.

The `Sort` function takes a binary predicate which specifies the sort order, as follows:

```
FruitSet.Sort([](const FString& A, const FString& B) {
return A > B; // sort by reverse-alphabetical order
});
// FruitSet == [ "Pear", "Orange", "Melon", "Mango", "Kiwi", "Grapefruit" ] (order is temporarily guaranteed)

FruitSet.Sort([](const FString& A, const FString& B) {
return A.Len() < B.Len(); // sort strings by length, shortest to longest
});
// FruitSet == [ "Pear", "Kiwi", "Melon", "Mango", "Orange", "Grapefruit" ] (order is temporarily guaranteed)

```

Copy full snippet

Operators

Like `TArray`, `TSet` is a regular value type and as such can be copied with the standard copy constructor or assignment operator. Sets strictly own their elements, so copying a set is deep; the new set will have its own copy of the elements.

```
TSet<int32, FString> NewSet = FruitSet;
NewSet.Add(TEXT("Apple"));
NewSet.Remove(TEXT("Pear"));
// FruitSet == [ "Pear", "Kiwi", "Melon", "Mango", "Orange", "Grapefruit" ]
// NewSet == [ "Kiwi", "Melon", "Mango", "Orange", "Grapefruit", "Apple" ]
```

Copy full snippet

Slack

Slack is allocated memory that doesn't contain an element. You can allocate memory without adding elements by calling `Reserve`, and you can remove elements without deallocating the memory they were using by calling `Reset` or by calling `Empty` with a non-zero slack parameter. Slack optimizes the process of adding new elements to the set by using pre-allocated memory instead of having to allocate new memory. It can also help with element removal, since the system does not need to deallocate memory. This is especially efficient when you are emptying a set that you expect to repopulate immediately with the same number of elements or fewer.

`TSet` does not provide a way of checking how many elements are preallocated, like the `Max` function in `TArray` does.

The following code removes all elements from the set without deallocating any memory, resulting in the creation of slack:

```
FruitSet.Reset();
// FruitSet == [ <invalid>, <invalid>, <invalid>, <invalid>, <invalid>,
<invalid> ]

```

Copy full snippet

To create slack directly, such as to preallocate memory before adding elements, use the `Reserve` function.

```
FruitSet.Reserve(10);
for (int32 i = 0; i < 10; ++i)
{
FruitSet.Add(FString::Printf(TEXT("Fruit%d"), i));
}
// FruitSet == [ "Fruit9", "Fruit8", "Fruit7" ... "Fruit2", "Fruit1",
"Fruit0" ]

```

Copy full snippet

Preallocating the slack has caused the new elements to be added in reverse order. Unlike arrays, sets do not attempt to maintain element order, and code dealing with sets should not expect element order to be stable or predictable.

To remove all slack from a `TSet`, use the `Collapse` and `Shrink` functions. `Shrink` removes all slack from the end of the container, but this will leave any empty elements in the middle or at the start.

```
// Remove every other element from the set.
for (int32 i = 0; i < 10; i += 2)
{
FruitSet.Remove(FSetElementId::FromInteger(i));
}
// FruitSet == ["Fruit8", <invalid>, "Fruit6", <invalid>, "Fruit4",
<invalid>, "Fruit2", <invalid>, "Fruit0", <invalid> ]

FruitSet.Shrink();
// FruitSet == ["Fruit8", <invalid>, "Fruit6", <invalid>, "Fruit4",
<invalid>, "Fruit2", <invalid>, "Fruit0" ]

```

Copy full snippet

`Shrink` only removed one invalid element in the code above because there was only one empty element at the end. To remove all slack, the `Compact` or `CompactStable` function should be called first, so that the empty spaces will be grouped together in preparation for `Shrink`. As its name implies, `CompactStable` maintains element order while consolidating empty elements.

```
FruitSet.CompactStable();
// FruitSet == ["Fruit8", "Fruit6", "Fruit4", "Fruit2", "Fruit0", <invalid>,
<invalid>, <invalid>, <invalid> ]
FruitSet.Shrink();
// FruitSet == ["Fruit8", "Fruit6", "Fruit4", "Fruit2", "Fruit0" ]

```

Copy full snippet

DefaultKeyFuncs

As long as a type has an `operator==` and a non-member `GetTypeHash` overload, TSet can use it, since the type is both the element and the key. However, it may be useful to use types as keys where it is undesirable to overload those functions. In these cases, you can provide your own custom `DefaultKeyFuncs`. To create `KeyFuncs` for your key type, you must define two typedefs and three static functions, as follows:

- `KeyInitType` — Type used to pass keys around. Usually drawn from the ElementType template parameter.

- `ElementInitType` — Type used to pass elements around. Also usually drawn from the ElementType template parameter, and therefore identical to KeyInitType.
- `KeyInitType GetSetKey(ElementInitType Element)` — Returns the key of an element. For sets, this is usually just the element itself.
- `bool Matches(KeyInitType A, KeyInitType B)` — Returns `true` if `A` and `B` are equivalent, `false` otherwise.
- `uint32 GetKeyHash(KeyInitType Key)` — Returns the hash value of `Key`.

`KeyInitType` and `ElementInitType` are typedefs to the normal passing convention of the key/element type. Usually, these will be a value for trivial types and a const reference for non-trivial types. Remember that the element type of a set is also the key type, which is why `DefaultKeyFuncs` uses only one template parameter, `ElementType`, to define both.

`TSet` assumes that two items that compare equal using `Matches` (in `DefaultKeyFuncs`) will also return the same value from `GetKeyHash` (in `KeyFuncs`).

Do not modify the key of an existing element in a way that will change the results from either of these functions, as this will invalidate set's internal hash. These rules also apply to the overloads of `operator==` and `GetKeyHash` when using the default implementation of `DefaultKeyFuncs`.

Miscellaneous

The `CountBytes` and `GetAllocatedSize` functions estimate how much memory the internal array is currently utilizing. `CountBytes` takes an `FArchive` parameter, while `GetAllocatedSize` does not. These functions are typically used for stats reporting.

The `Dump` function takes an `FOutputDevice` and writes out some implementation information about the contents of the set. There is also a `DumpHashElements` function that lists all elements from all hash entries. These functions are usually used for debugging.