Unity Collections package

This package provides unmanaged data structures that can be used in jobs and Burst-compiled code.

The collections provided by this package fall into three categories:

- The collection types in `Unity.Collections` whose names start with `Native-` have safety checks for ensuring that they're properly disposed and are used in a thread-safe manner.
- The collection types in `Unity.Collections.LowLevel.Unsafe` whose names start with `Unsafe-` do not have these safety checks.
- The remaining collection types are not allocated and contain no pointers, so effectively their disposal and thread safety are never a concern. These types hold only small amounts of data.

The `Native-` types perform safety checks to ensure that indexes passed to their methods are in bounds, but the other types in most cases do not.

Several `Native-` types have `Unsafe-` equivalents, for example, `NativeList` has `UnsafeList`, and `NativeHashMap` has `UnsafeHashMap`.

While you should generally prefer using the `Native-` collections over their `Unsafe-` equivalents, `Native-` collections cannot contain other `Native-` collections (owing to the implementation of their safety checks). So if, say, you want a list of lists, you can have a `NativeList<UnsafeList<T>>` or an `UnsafeList<UnsafeList<T>>`, but you cannot have a `NativeList<NativeList<T>>`.

When safety checks are disabled, there is generally no significant performance difference between a `Native-` type and its `Unsafe-` equivalent. In fact, most `Native-` collections are implemented simply as wrappers of their `Unsafe-` counterparts. For example, `NativeList` is comprised of an `UnsafeList` plus a few handles used by the safety checks.

For more information on the specific collection types, see the documentation on Collection types

Collection types

Array-like types

A few key array-like types are provided by the core module, including `Unity.Collections.NativeArray<T>` and `Unity.Collections.NativeSlice<T>`. This package itself provides:

Data structure	**Description**
@Unity.Collections.NativeList`1	A resizable list. Has thread- and disposal-safety checks.
@Unity.Collections.LowLevel.Unsafe.UnsafeList`1	A resizable list.
@Unity.Collections.LowLevel.Unsafe.UnsafePtrList`1	A resizable list of pointers.
@Unity.Collections.NativeStream	A set of append-only, untyped buffers. Has thread- and disposal-safety checks.
@Unity.Collections.LowLevel.Unsafe.UnsafeStream	A set of append-only, untyped buffers.
@Unity.Collections.LowLevel.Unsafe.UnsafeAppendBuffer	An append-only untyped buffer.
@Unity.Collections.NativeQueue`1	A resizable queue. Has thread- and disposal-safety checks.
@Unity.Collections.LowLevel.Unsafe.UnsafeRingQueue`1	A fixed-size circular buffer.
@Unity.Collections.FixedList32Bytes`1	A 32-byte list, including 2 bytes of overhead, so 30 bytes are available for storage. Max capacity depends upon T.

`FixedList32Bytes<T>` has variants of larger sizes: `FixedList64Bytes<T>`, `FixedList128Bytes<T>`, `FixedList512Bytes<T>`, `FixedList4096Bytes<T>`.

There are no multi-dimensional array types, but you can simply pack all the data into a single-dimension. For example, for an `int[4][5]` array, use an `int[20]` array instead (because `4 * 5` is `20`).

When using the Entities package, a DynamicBuffer component is often the best choice for an array- or list-like collection.

See also @Unity.Collections.NativeArrayExtensions, @Unity.Collections.ListExtensions, @Unity.Collections.NativeSortExtension.

Map and set types

Data structure	Description
@Unity.Collections.NativeHashMap`2	An unordered associative array of key-value pairs. Has thread- and disposal-safety checks.
@Unity.Collections.LowLevel.Unsafe.UnsafeHashMap`2	An unordered associative array of key-value pairs.
@Unity.Collections.NativeHashSet`1	A set of unique values. Has thread- and disposal-safety checks.
@Unity.Collections.LowLevel.Unsafe.UnsafeHashSet`1	A set of unique values.
@Unity.Collections.NativeMultiHashMap`2	An unordered associative array of key-value pairs. The keys do not have to be unique, *i.e.* two pairs can have equal keys. Has thread- and disposal-safety checks.
@Unity.Collections.LowLevel.Unsafe.UnsafeMultiHashMap`2	An unordered associative array of key-value pairs. The keys do not have to be unique, *i.e.* two pairs can have equal keys.

See also @Unity.Collections.HashSetExtensions, @Unity.Collections.NotBurstCompatible.Extensions, and @Unity.Collections.LowLevel.Unsafe.NotBurstCompatible.Extensions

Bit arrays and bit fields

Data structure	Description
@Unity.Collections.BitField32	A fixed-size array of 32 bits.
@Unity.Collections.BitField64	A fixed-size array of 64 bits.
@Unity.Collections.NativeBitArray	An arbitrary-sized array of bits. Has thread- and disposal-safety checks.
@Unity.Collections.LowLevel.Unsafe.UnsafeBitArray	An arbitrary-sized array of bits.

String types

Data structure	Description
@Unity.Collections.NativeText	A UTF-8 encoded string. Mutable and resizable. Has thread- and disposal-safety checks.
@Unity.Collections.FixedString32Bytes	A 32-byte UTF-8 encoded string, including 3 bytes of overhead, so 29 bytes available for storage.

`FixedString32Bytes` has variants of larger sizes: `FixedString64Bytes`, `FixedString128Bytes`, `FixedString512Bytes`, `FixedString4096Bytes`.

See also @Unity.Collections.FixedStringMethods

Other types

Data structure	Description
@Unity.Collections.NativeReference`1	A reference to a single value. Functionally equivalent to an array of length 1. Has thread- and disposal-safety checks.
@Unity.Collections.LowLevel.Unsafe.UnsafeAtomicCounter32	A 32-bit atomic counter.
@Unity.Collections.LowLevel.Unsafe.UnsafeAtomicCounter64	A 64-bit atomic counter.

Job safety checks

The purpose of the job safety checks is to detect job conflicts. Two jobs conflict if:

1. Both jobs access the same data.
2. One job or both jobs have write access to the data.

In other words, there's no conflict if both jobs just have read only access to the data.

For example, you generally wouldn't want one job to read an array while meanwhile another job is writing the same array, so the safety checks consider that possibility to be a conflict. To resolve such conflicts, you must make one job a dependency of the other to ensure their execution does not overlap. Whichever of the two jobs you want to run first should be the dependency of the other.

When the safety checks are enabled, each `Native-` collection has an `AtomicSafetyHandle` for performing thread-safety checks. Scheduling a job locks the `AtomicSafetyHandle`'s of all `Native-` collections in the job. Completing a job releases the `AtomicSafetyHandle`'s of all `Native-` collections in the job.

While a `Native-` collection's `AtomicSafetyHandle` is locked:

1. Jobs which use the collection can only be scheduled if they depend upon all the already scheduled job(s) which also use it.
2. Accessing the collection from the main thread will throw an exception.

Read only access in jobs

As a special case, there's no conflict between two jobs if they both strictly just read the same data, *.e.g.* there's no conflict if one job reads from an array while meanwhile another also job reads from the same array.

The @Unity.Collections.ReadOnlyAttribute marks a `Native-` collection in a job struct as being read only:

```
public struct MyJob : IJob
{
    // This array can only be read in the job.
    [ReadOnly] public NativeArray<int> nums;

    public void Execute()
    {
        // If safety checks are enabled, an exception is thrown here
        // because the array is read only.
        nums[0] = 100;
    }
}
```

Marking collections as read only has two benefits:

1. The main thread can still read a collection if all scheduled jobs that use the collection have just read only access.
2. The safety checks will not object if you schedule multiple jobs with read only access to the same collection, even without any dependencies between them. Therefore these jobs can run concurrently with each other.

Enumerators

Most of the collections have a `GetEnumerator` method, which returns an implementation of `IEnumerator<T>`. The enumerator's `MoveNext` method advances its `Current` property to the next element.

```
NativeList<int> nums = new NativeList<int>(10, Allocator.Temp);
```

```
// Calculate the sum of all elements in the list.
int sum = 0;
NativeArray<int>.Enumerator enumerator = nums.GetEnumerator();

// The first MoveNext call advances the enumerator to the first element.
// MoveNext returns false when the enumerator has advanced past the last element.
while (enumerator.MoveNext())
{
    sum += enumerator.Current;
}

// The enumerator is no longer valid to use after the array is disposed.
nums.Dispose();
```

Parallel readers and writers

Several of the collection types have nested types for reading and writing from parallel jobs. For example, to write safely to a `NativeList<T>` from a parallel job, you need a `NativeList<T>.ParallelWriter`:

```
NativeList<int> nums = new NativeList<int>(1000, Allocator.TempJob);

// The parallel writer shares the original list's AtomicSafetyHandle.
var job = new MyParallelJob {NumsWriter = nums.AsParallelWriter()};
```

```
public struct MyParallelJob : IJobParallelFor
{
    public NativeList<int>.ParallelWriter NumsWriter;

    public void Execute(int i)
    {
        // A NativeList<T>.ParallelWriter can append values
        // but not grow the capacity of the list.
        NumsWriter.AddNoResize(i);
    }
}
```

Note that these parallel readers and writers do not usually support the full functionality of the collection. For example, a `NativeList` cannot grow its capacity in a parallel job (because there is no way to safely allow this without incurring significantly more synchronization overhead).

Deterministic reading and writing

Although a `ParallelWriter` ensures the safety of concurrent writes, the *order* of the concurrent writes is inherently indeterminstic because it depends upon the happenstance of thread scheduling (which is controlled by the operating system and other factors outside of your program's control).

Likewise, although a `ParallelReader` ensures the safety of concurrent reads, the *order* of the concurrent reads is inherently indeterminstic, so it can't be known which threads will read which values.

One solution is to use either @Unity.Collections.NativeStream or @Unity.Collections.LowLevel.Unsafe.UnsafeStream, which splits reads and writes into a separate buffer for each thread and thereby avoids indeterminism.

Alternatively, you can effectively get a deterministic order of parallel reads if you deterministically divide the reads into separate ranges and process each range in its own thread.

You can also get a deterministic order if you deterministically sort the data after it has been written to the list.

Using unmanaged memory

The `Native-` and `Unsafe-` collections in this package are allocated from unmanaged memory, meaning their existence is unknown to the garbage collector. You are responsible for deallocating any unmanaged memory that you no longer need. Failing to deallocate large or numerous allocations can lead to wasting more and more memory, which may eventually slow down or even crash your program.

Allocators

An *allocator* governs some unmanaged memory from which you can make allocations. Different allocators organize and track their memory in different ways. The three standard provided allocators are:

`Allocator.Temp`

The fastest allocator. For very short-lived allocations. Temp allocations *cannot* be passed into jobs.

Each frame, the main thread creates a Temp allocator which is deallocated in its entirety at the end of the frame. Each job also creates one Temp allocator per thread, and these are deallocated in their entireties at the end of the job. Because a Temp allocator gets discarded as a whole, you actually don't need to manually deallocate your Temp allocations (in fact, doing so is a no-op).

Temp allocations are only safe to use within the thread where they were allocated. So while Temp allocations can be made *within* a job, **main thread Temp allocations cannot be passed into a job**. For example, a NativeArray that's Temp allocated in the main thread cannot be passed into a job.

`Allocator.TempJob`

The next fastest allocator. For short-lived allocations. TempJob allocations can be passed into jobs.

You are expected to deallocate your TempJob allocations within 4 frames of their creation. The number 4 was chosen because it's common to want allocations that last a couple frames: the limit of 4 accommodates this need with a comfortable extra margin.

For the `Native-` collection types, the disposal safety checks will throw an exception if a TempJob allocation lives longer than 4 frames. For the `Unsafe-` collection types, you are still expected to deallocate them within 4 frames, but no safety checks are performed to ensure you do so.

`Allocator.Persistent`

The slowest allocator. For indefinite lifetime allocations. Persistent allocations can be passed into jobs.

Because Persistent allocations are allowed to live indefinitely, no safety check can detect if a Persistent allocation has outlived its intended lifetime. Consequently, you should be extra careful to deallocate a Persistent allocation when you no longer need it.

Disposal (deallocation)

Each collection retains a reference to the allocator from which its memory was allocated because deallocation requires specifying the allocator.

- An `Unsafe-` collection's `Dispose` method deallocates its memory.
- A `Native-` collection's `Dispose` method deallocates its memory and frees the handles needed for safety checks.
- An enumerator's `Dispose` method is a no-op. The method is included only to fulfill the `IEnumerator<T>` interface.

We often want to dispose a collection after the jobs which need it have run. The `Dispose(JobHandle)` method creates and schedules a job which will dispose the collection, and this new job takes the input handle as its dependency. Effectively, the method differs disposal until after the dependency runs:

```
NativeArray<int> nums = new NativeArray<int>(10, Allocator.TempJob);

// Create and schedule a job that uses the array.
ExampleJob job = new ExampleJob { Nums = nums };
JobHandle handle = job.Schedule();

// Create and schedule a job that will dispose the array after the ExampleJob has run.
// Returns the handle of the new job.
handle = nums.Dispose(handle);
```

The `IsCreated` property

The `IsCreated` property of a collection is false only in two cases:

1. Immediately after creating a collection with its default constructor.
2. After `Dispose` has been called on the collection.

Understand, however, that you're not intended to use a collections's default constructor. It's only made available because C# requires all structs to have a public default constructor.

Also note that calling `Dispose` on a collection sets `IsCreated` to false *only in that struct*, not in any copies of the struct. Consequently, `IsCreated` may still be true even after the collection's underlying memory was deallocated if...

- `Dispose` was called on a different copy of the struct.
- Or the underlying memory was deallocated *via* an alias.

Aliasing

An *alias* is a collection which does not have its own allocation but instead shares the allocation of another collection, in whole or in part. For example, an UnsafeList can be created that doesn't allocate its own memory but instead uses a NativeList's allocation. Writing to this shared memory *via* the UnsafeList affects the content of the NativeList, and *vice versa*.

You do not need to dispose aliases, and in fact, calling `Dispose` on an alias does nothing. Once an original is disposed, the aliases of that original can no longer be used:

```
NativeList<int> nums = new NativeList<int>(10, Allocator.TempJob);
nums.Length = 5;

// Create an array of 5 ints that aliases the content of the list.
NativeArray<int> aliasedNums = nums.AsArray();

// Modify the first element of both the array and the list.
aliasedNums[0] = 99;

// Only the original need be disposed.
nums.Dispose();

// Throws an ObjectDisposedException because disposing
// the original deallocates the aliased memory.
aliasedNums[0] = 99;
```

Aliasing can be useful in a few scenarios:

- Getting a collection's data in the form of another collection type without copying the data. For example, you can create an UnsafeList that aliases a NativeArray.
- Getting a subrange of a collection's data without copying the data. For example, you can create an UnsafeList that aliases a subrange of another list or array.
- Array reinterpretation.

Perhaps surprisingly, it's allowed for an `Unsafe-` collection to alias a `Native-` collection even though such cases undermine the safety checks. For example, if an UnsafeList aliases a NativeList, it's not safe to schedule a job that accesses one while also another job is scheduled that accesses the other, but the safety checks do not catch these cases. It is your responsibility to avoid such mistakes.

Array reinterpretation

A *reinterpretation* of an array is an alias of the array that reads and writes the content as a different element type. For example, a `NativeArray<int>` which reinterprets a `NativeArray<ushort>` shares the same bytes, but it reads and writes the bytes as ints instead of ushorts; because each int is 4 bytes while each ushort is 2 bytes, each int corresponds to two ushorts, and the reinterpretation has half the length of the original.

```
NativeArray<int> ints = new NativeArray<int>(10, Allocator.Temp);

// Length of the reinterpreted array is 20
// (because it has two shorts per one int of the original).
NativeArray<short> shorts = ints.Reinterpret<int, short>();

// Modifies the first 4 bytes of the array.
shorts[0] = 1;
shorts[1] = 1;

int val = ints[0];   // val is 65537 (2^16 + 2^0)

// Like with other aliased collections, only the original
// needs to be disposed.
ints.Dispose();

// Throws an ObjectDisposedException because disposing
// the original deallocates the aliased memory.
shorts[0] = 1;
```

Known issues

All containers allocated with `Allocator.Temp` on the same thread use a shared `AtomicSafetyHandle` instance rather than each having their own. On the one hand, this is fine because Temp allocated collections cannot be passed into jobs. On the other hand, this is problematic when using `NativeHashMap`, `NativeMultiHashMap`, `NativeHashSet`, and `NativeList` together in situations where their secondary safety handle is used. (A secondary safety handle ensures that a NativeArray which aliases a NativeList gets invalidated when the NativeList is reallocated due to resizing.)

Operations that invalidate an enumerator for these collection types (or invalidate the `NativeArray` returned by `NativeList.AsArray`) will also invalidate all other previously acquired enumerators. For example, this will throw when safety checks are enabled:

```
var list = new NativeList<int>(Allocator.Temp);
list.Add(1);

// This array uses the secondary safety handle of the list, which is
// shared between all Allocator.Temp allocations.
var array = list.AsArray();

var list2 = new NativeHashSet<int>(Allocator.Temp);

// This invalidates the secondary safety handle, which is also used
// by the list above.
list2.TryAdd(1);

// This throws an InvalidOperationException because the shared safety
// handle was invalidated.
var x = array[0];
```

This defect will be addressed in a future release.