**NativeContainer**

The drawback to the [safety system’s](https://docs.unity3d.com/Manual/JobSystemSafetySystem.html) process of copying data is that it also isolates the results of a job within each copy. To overcome this limitation you need to store the results in a type of shared memory called [NativeContainer](https://docs.unity3d.com/ScriptReference/Unity.Collections.LowLevel.Unsafe.NativeContainerAttribute.html).

**What is a NativeContainer?**

A NativeContainer is a managed value type that provides a relatively safe C# wrapper for native memory. It contains a pointer to an unmanaged allocation. When used with the Unity C# Job System, a NativeContainer allows a job to access data shared with the main thread rather than working with a copy.

**What types of NativeContainer are available?**

Unity ships with a NativeContainer called [NativeArray](https://docs.unity3d.com/ScriptReference/Unity.Collections.NativeArray_1.html). You can also manipulate a NativeArray with [NativeSlice](https://docs.unity3d.com/ScriptReference/Unity.Collections.NativeSlice_1.html) to get a subset of the NativeArray from a particular position to a certain length.

**Note**: The [Entity Component System](https://github.com/Unity-Technologies/EntityComponentSystemSamples) (ECS) package extends the Unity.Collections namespace to include other types of NativeContainer:

* NativeList - a resizable NativeArray.
* NativeHashMap - key and value pairs.
* NativeMultiHashMap - multiple values per key.
* NativeQueue - a first in, first out ([FIFO](https://en.wikipedia.org/wiki/FIFO_(computing_and_electronics))) queue.

**NativeContainer and the safety system**

The safety system is built into all NativeContainer types. It tracks what is reading and writing to any NativeContainer.

**Note**: All safety checks on NativeContainer types (such as out of bounds checks, deallocation checks, and race condition checks) are only available in the Unity **Editor** and **Play Mode**.

Part of this safety system is the [DisposeSentinel](https://docs.unity3d.com/ScriptReference/Unity.Collections.LowLevel.Unsafe.DisposeSentinel.html) and [AtomicSafetyHandle](https://docs.unity3d.com/ScriptReference/Unity.Collections.LowLevel.Unsafe.AtomicSafetyHandle.html). The DisposeSentinel detects memory leaks and gives you an error if you have not correctly freed your memory. Triggering the memory leak error happens long after the leak occurred.

Use the AtomicSafetyHandle to transfer ownership of a NativeContainer in code. For example, if two scheduled jobs are writing to the same NativeArray, the safety system throws an exception with a clear error message that explains why and how to solve the problem. The safety system throws this exception when you schedule the offending job.

In this case, you can schedule a job with a dependency. The first job can write to the NativeContainer, and once it has finished executing, the next job can then safely read and write to that same NativeContainer. The read and write restrictions also apply when accessing data from the main thread. The safety system does allow multiple jobs to read from the same data in parallel.

By default, when a job has access to a NativeContainer, it has both read and write access. This configuration can slow performance. The C# Job System does not allow you to schedule a job that has write access to a NativeContainer at the same time as another job that is writing to it.

If a job does not need to write to a NativeContainer, mark the NativeContainer with the [ReadOnly] attribute, like so:

[ReadOnly]

public NativeArray<int> input;

In the above example, you can execute the job at the same time as other jobs that also have read-only access to the first NativeArray.

**Note**: There is no protection against accessing static data from within a job. Accessing static data circumvents all safety systems and can crash Unity. For more information, see [C# Job System tips and troubleshooting](https://docs.unity3d.com/Manual/JobSystemTroubleshooting.html).

**NativeContainer Allocator**

When creating a NativeContainer, you must specify the type of memory allocation you need. The allocation type depends on the length of time the job runs. This way you can tailor the allocation to get the best performance possible in each situation.

There are three [Allocator](https://docs.unity3d.com/ScriptReference/Unity.Collections.Allocator.html) types for NativeContainer memory allocation and release. You need to specify the appropriate one when instantiating your NativeContainer.

* **Allocator.Temp** has the fastest allocation. It is for allocations with a lifespan of one frame or fewer. You should not pass NativeContainer allocations using Temp to jobs. You also need to call the Dispose method before you return from the method call (such as [MonoBehaviour.Update](https://docs.unity3d.com/ScriptReference/MonoBehaviour.Update.html), or any other callback from native to managed code).
* **Allocator.TempJob** is a slower allocation than Temp but is faster than Persistent. It is for allocations within a lifespan of four frames and is thread-safe. If you don’t Dispose of it within four frames, the console prints a warning, generated from the native code. Most small jobs use this NativeContainer allocation type.
* **Allocator.Persistent** is the slowest allocation but can last as long as you need it to, and if necessary, throughout the application’s lifetime. It is a wrapper for a direct call to [malloc](http://www.cplusplus.com/reference/cstdlib/malloc/" \t "_blank). Longer jobs can use this NativeContainer allocation type. You should not use Persistent where performance is essential.

For example:

NativeArray<float> result = new NativeArray<float>(1, Allocator.TempJob);

**Note**: The number 1 in the example above indicates the size of the NativeArray. In this case, it has only one array element (as it only stores one piece of data in result).