Containers

# Introduction

Containers are used to store a number of elements. While the C++ standard contains several containers like std::vector and std::set, the storage of these containers is allocated on the heap. In embedded applications, more control over the storage is needed. For this reason, the infra package contains two types of containers: the Bounded containers and the Intrusive containers. While their storage is not on the heap, the interfaces of these containers closely mimic the interfaces of the standard containers.

# The Bounded Containers

The first strategy to deal with storage in containers is by allocating memory inside the containers themselves. The size of a container therefore includes the storage of its elements. The maximum number of elements to be stored is passed as a template parameter to the instantiation of the container, e.g. infra::BoundedVector<int>::WithMaxSize<5> myVector declares a container myVector which holds up to 5 integers. Since the size of the storage is known up front, the maximum number of elements that can be placed in the container is bounded, hence their name.

The Bounded counterparts of the containers of the standard library are the following:

|  |  |
| --- | --- |
| infra::BoundedVector<T>::WithMaxSize<N> | std::vector<T> |
| infra::BoundedDeque<T>::WithMaxSize<N> | std::deque<T> |
| infra::BoundedString<T>::WithStorage<N> | std::string<T> |
| infra::BoundedPriorityQueue<T, N> | std::priority\_queue<T> |
| infra::BoundedList<T>::WithMaxSize<N> | std::list<T> |
| infra::BoundedForwardList<T>::WithMaxSize<N> | Std::forward\_list<T> |

These bounded containers conform to the same specifications as placed by the C++ standard on their counterparts where possible, so they generally have the same constructors, accessors, types, and complexity requirements. One notable omission is the allocator passed to standard containers, since bounded containers allocate their elements inside their own space. These containers have two additional accessors: std::size\_t max\_size() and bool full().

While the intention of this document is not to repeat the documentation available on standard containers, for quick reference a summary of available functions is given here:

|  |  |
| --- | --- |
| bool empty() | Returns true if and only if the container is empty |
| std::size\_t size() | Returns the number of currently held elements |
| iterator begin() | Returns an iterator to the first element |
| iterator end() | Returns an iterator which points to one past the last element |
| void clear() | Removes all elements |
| void push\_back(element) | Inserts one element to the back |
| void push\_front(element) | Inserts one element to the front |
| void emplace\_back(...) | Constructs one element at the back, parameters are forwarded to the constructor of the element |
| void emplace\_front(...) | Constructs one element at the front, parameters are forwarded to the constructor of the element |
| void pop\_back() | Removes one element from the back |
| void pop\_front() | Removes one element from the front |
| iterator insert(position, element) | Inserts an element at the given position |
| Iterator erase(position) | Removes an element at the given position |

## References to Bounded Containers

Often, the class that holds a container of elements is not the class that makes the decision how big that container should be. For example, the DiCommRouter holds a BoundedVector for storing the DiCommServers to which it refers. It doesn’t know, however, whether it should be able to hold 2, 10, or 500 DiCommServers. Allocating too much space would be wasteful; but at the place in the code where the DiCommRouter is instantiated this limit is known. In order to specify the maximum storage for a container at a later moment, BoundedVector<T> (and similarly other containers) may be created as a reference to a BoundedVector<T>::WithStorage<N>. By constructing the actual BoundedVector at the point of instantiation of DiCommRouter, the choice for the maximum number of elements is delayed. For example:

class DiCommRouter

: public DiCommServer

{

public:

...

explicit DiCommRouter(infra::BoundedVector<SuccessorNode>& storage);

protected:

infra::BoundedVector<SuccessorNode>& successors;

};

The DiCommRouter can now be instantiated as follows:

infra::BoundedVector<SuccessorNode>::WithMaxSize<10> storage;

DiCommRouter router(storage);

The DiCommRouter now uses a BoundedVector with a maximum size of 10 elements.

In order to simplify instantiation of the DiCommRouter, it uses the infra::WithStorage helper class to feed a constructed BoundedVector to the DiCommRouter. Using this WithMaxSize parameter, instantiation is simplified to:

class DiCommRouter

: public DiCommServer

{

public:

...

template<std::size\_t Max>

using WithMaxSize = infra::WithStorage<DiCommRouter, infra::BoundedVector<SuccessorNode>::WithMaxSize<Max>>;

explicit DiCommRouter(infra::BoundedVector<SuccessorNode>& storage);

protected:

infra::BoundedVector<SuccessorNode>& successors;

};

DiCommRouter::WithMaxSize<10> router(storage);

# The Intrusive Containers

Another approach for storing elements in a container is to not allocate additional storage for an object, but to make the user of a container responsible for storage allocation. By adding some additional administration to an object, a container can keep track of its objects by jumping from object to object via that additional administration. For instance, objects used in an infra::IntrusiveForwardList each have a next pointer, so that when the container keeps track of its first element it can enumerate all its elements by following the next pointer.

In order to inject the administration into elements, the elements derive from container::NodeType. The container’s administration therefore intrudes into the element, hence their name Intrusive containers.

A consequence of this is that an element can only be assigned to one container at a time, but unlike the Bounded containers, Intrusive containers cannot become full.

Like the Bounded containers, the infra package contains counterparts of containers found in the standard library that closely mimic their behaviour. The provided Intrusive containers are:

|  |  |
| --- | --- |
| infra::IntrusiveForwardList<T> | std::forward\_list<T> |
| infra::IntrusiveList<T> | std::list<T> |
| infra::IntrusiveSet<T, Compare> | std::set<T, Compare> |
| infra::IntrusivePriorityQueue<T, Compare> | std::priority\_queue<T, Compare> |

# std::array

One standard container of note is std::array<T, Size>. This container does not use the heap and has a fixed size. It is a drop-in replacement of the C array, with a C++ interface. When a container of a fixed-size is needed, std::array should be used.