# `std::array`: The Secure, Convenient Option for Fixed-Sized Sequences : Page 2

***Migrate your fixed-sized sequences to std::array, which offers a secure, efficient, and convenient alternative to built-in arrays—sans the overhead of vector.***

by Danny Kalev

Google + Jun 11, 2009

Page 2 of 3

## Instantiation and Usage

The `std::array` class template is defined in the standard header `<array>`, and it supports random access iterators. An instance of `array<T, N>` stores `N` elements of type `T`. The elements are stored contiguously on the stack (as opposed to `std::vector`, which stores elements on the free-store). Also, `std::array` has implicitly-declared special member functions (constructor, destructor, copy constructor, and assignment operator) with minimal runtime overhead.

You instantiate an array object like this:

```
#include <array>

std::array <int,5> a={1,2,3};
```

The first template argument specifies the type of the elements. The second argument specifies the array's size. Notice that the size is mandatory; you can't deduce it from an initializer-list:

```
std::array <int> a2={1,2,3}; //error, size missing
```

The initializer-list is a comma-separated list of up to `N` elements whose types are convertible to `T`. If the initializer list contains fewer initializers than `N`, the remaining elements are default-initialized. Thus, in the example above, `a[3]` and `a[4]` are initialized to zero.

## Security Enhancements

With respect to security, `std::array` doesn't implicitly convert to a pointer. This is to protect your code from inadvertent pointer-related bugs, which are so pervasive when using built-in arrays. If you want to convert `std::array` to a pointer, you must use the `data()` member function:

```
int* p = a; //error, no implicit conversion to int*
int* p2 = a.data(); //OK
const int* pc = a.data(); //OK
```

Similarly, `std::array` blocks implicit derived-to-base conversions, which might be quite dangerous:

```
struct Dog : Animal { /* ... */ };
struct Cat : Animal { /* ... */ };

void trap(array<Animal*,5>& animals)
{
 animals[3] = new Cat;
};

array<Dog*,5> dogs;
trap(dogs); /*error: can't convert array<Dog*,5> to
             array<Animal*,5>; */
```

If this implicit conversion were allowed, `dogs[3]` would now contain Cat.

Finally, unlike built-in arrays, which decay into pointers at the slightest pretext and thus lose track of their size, `std::array` always knows how many elements it contains:

```
int probe(const array<int,5>* parr)
{
 cout<< parr->size() <<endl; //output 5
 return parr->size();
}
```

Previous Page Next Page

« Previous Page 1 2 3 Next Page »

**0 Comments** (click to add your comment)

## Comment and Contribute

| | |
|---|---|
| | Your name/nickname |
| | Your email |
| | WebSite |
| | Subject |

(Maximum characters: 1200). You have 1200 characters left.

Submit Your Comment

Sitemap