

# Array versus container classes

Richel Bilderbeek 

# Goal

- Why should one prefer a container class over a plain array?
  - Why arrays?
  - Why use container classes?
  - What are the problems with arrays?
  - Benchmark of arrays versus `std::vector` [1]

[1] In C++11, one would use `std::array`

# The history of arrays

# Why use container classes?

- ...

# Because the literature says so ...

- Bjarne Stroustrup. A tour of C++. 2014. ISBN: 978-0-321-958310. Chapter 11.7.11: 'Prefer array over built-in arrays'
- Bjarne Stroustrup. The C++ Programming Language (3rd edition). ISBN: 0-201-88954-4 Chapter 5.8.4 'Use vector and valarray rather than built-in (C-style) arrays'
- Bjarne Stroustrup. The C++ Programming Language (4th edition). 2013. ISBN: 978-0-321-56384-2. Chapter 7.8. Advice. page 199: '[6] Use containers (e.g., vector, array, and valarray) rather than built-in (C-style) arrays'
- Herb Sutter and Andrei Alexandrescu. C++ coding standards: 101 rules, guidelines, and best practices. ISBN: 0-32-111358-6. Chapter 77: 'Use vector and string instead of arrays'



Bjarne Stroustrup



Herb Sutter



Andrei Alexandrescu

# Because the literature says so ...

- Marshall Cline, Greg Lomow and Mike Girou. C++ FAQs. ISBN: 0-201-3098301, FAQ 28.02: 'Are arrays good or evil?' (Answer: 'Arrays are evil')
- Joint Strike Fighter Air Vehicle C++ Coding Standards for the System Development and Demonstration Program. Document Number 2RDU00001 Rev C. December 2005. AV Rule 97: 'Arrays shall not be used in interfaces. Instead, the Array class should be used.'
- Scott Meyers. C++ And Beyond 2012 session: 'Initial thoughts on Effective C++11'. 2012. 'Prefer `std::array` to built-in Arrays'
- Scott Meyers. Effective Modern C++ (1st Edition). 2014. ISBN: 978-1-491-90399-5. Item 1, page 17: 'Of course, as a modern C++ developer, you'd naturally prefer a `std::array` to a built-in array'



Scott Meyers



F-35 Lightning II

# Problems according to the literature

- An array does not know its size
- Cannot assign plain arrays
- Implicit conversion to pointer

# Example, adapted from [1]

```
void iota(int a[], const int sz) {  
    for (int i=0; i!=sz; ++i) {  
        a[i] = i;  
    }  
}
```

```
int main() {  
    int v[20];  
    int w[10];  
    iota(v,20);  
    iota(w,20); //Oops  
}
```



# Example, adapted from [1]

```
int main()
{
    int v[] = {1,2};
    int w[] = {3,4};
    v = w; //Invalid array assignment


    //Cannot be fixed by std::memcpy
    //for non-PODs
}
```

# Example, adapted from [1]

```
void f()  
{  
    Derived v[3];  
    Base * const w = v; //Disaster waiting to happen  
    std::cout << w[1];  
}
```

# Example, adapted from [1]

```
struct Base { int m_base; };  
struct Derived : public Base { int m_derived; };  
void f() {  
    const int sz = 3;  
    Derived v[sz];  
    //Initialize  
    for (int i=0; i!=sz; ++i) { v[i].m_base = i+1; v[i].m_derived = i+4; }  
    //Display  
    for (int i=0; i!=sz; ++i) {  
        std::cout << v[i].m_base << " " << v[i].m_derived << '\n'; }  
    Base * const w = v; //Disaster waiting to happen  
    //Display  
    for (int i=0; i!=sz; ++i) { std::cout << w[i].m_base << '\n'; }  
}
```



1	4
2	5
3	6



1
4
2

Base has a different offset than Derived

# My definition of the problem

- You abandon compile-time and run-time checking
- I do not like long nights of debugging

# My example

```
void f(const int * const p, const int sz)
{
    //Cannot check if p indeed has size sz
}
```

```
void g()
{
    /* ... */
    //Cannot check if i is a valid index from v only
    v[i] = 42;
}
```

# Alternative

```
void f(const std::vector<int>& v)
{
    //Can get the true size of v
}
```

```
void g()
{
    /* ... */
    //Checks if i is a valid index
    v.at(i) = 42;
}
```

# Speed

- Common misconception: arrays are faster
- Many benchmarks compare array and C++ containers incorrectly using those containers

# Experiment

- Google for 'std::vector slower than array'
- Compile the first benchmark
- Measure
- Improve the benchmark
- Measure again



# Benchmark

- Found post on [Stack Overflow](#)
- Benchmark on Travis CI with [this GitHub](#)
- Measured these results:


```
UseArray completed in 6.191 seconds  
UseVector completed in 44.706 seconds
```

# Original code


```
std::vector<Pixel> pixels;  
pixels.resize(dimension * dimension);  
for(int i = 0; < dimension * dimension; ++i)  
{  
    pixels[i].r = 255;  
    pixels[i].g = 0;  
    pixels[i].b = 0;  
}
```

Writes Pixels twice!

# Original code



```
std::vector<Pixel> pixels;  
pixels.resize(dimension * dimension);  
for(int i = 0; < dimension * dimension; ++i)  
{  
    pixels[i].r = 255;  
    pixels[i].g = 0;  
    pixels[i].b = 0;  
}
```



Indexed access

# Equivalent code

```
std::vector<Pixel> pixels(  
    dimension * dimension,  
    Pixel(255, 0, 0)  
);
```

# Second benchmark

- In debug mode:

```
UseArray completed in 6.965 seconds  
UseVector completed in 19.794 seconds
```

# Third benchmark

- In release mode:

```
UseArray completed in 2.438 seconds  
UseVector completed in 2.437 seconds
```

# Conclusion

- The literature advises to use container classes
- Use of arrays can result in errors that cannot be discovered at compile-time nor run-time
- Speed benchmarks incorrectly suggest array is faster, due to naive author
  - Do not forget to benchmark in release mode