TDDD38/726G82:
Adv. Programming in C++
Fundamentals III

Christoffer Holm

Department of Computer and information science

LINKÖPING UNIVERSITY

1 Lifetime & Special Member Functions
2 Value categories
3 Inheritance
4 Polymorphism

1 Lifetime & Special Member Functions
2 Value categories
3 Inheritance
4 Polymorphism

Lifetime & Special Member Functions

```
1  class Array
2  {
3  public:
4    explicit Array(std::size_t size)                // constructor
5      : size { size }, data { new int[size] }
6    { }
7
8    int& operator[](std::size_t index)              // non-const access
9    { return data[index]; }
10
11   int const& operator[](std::size_t index) const // const access
12   { return data[index]; }
13
14   std::size_t size() const                        // get the size
15   { return size; }
16
17 private:
18   std::size_t size;
19   int* data;
20 };
```

Lifetime & Special Member Functions

```
1  class Array
2  {
3  public:
4    explicit Array(std::size_t size)                // constructor
5      : size { size }, data { new int[size] }
6    { }
7
8    int& operator[](std::size_t index)              // non-const access
9    { return data[index]; }
10
11   int const& operator[](std::size_t index) const // const access
12   { return data[index]; }
13
14   std::size_t size() const                        // get the size
15   { return size; }
16
17 private:
18   std::size_t size;
19   int* data;
20 };
```

Do you see any problems with this class?

Lifetime & Special Member Functions

Problems

- The first problem is that we dynamically allocate memory in the constructor, but we *never* deallocate that memory.
- There is no (proper) way for the user of this class to actually `delete` the allocated data, since `data` is private.
- This means that it is *our* responsibility to actually handle the memory.

Lifetime & Special Member Functions

Problems

- Whenever we deal with a resource, such as: dynamic memory, files, sockets, threads etc. we have to *acquire* the resource at some point and then *release* it once we are done with it.
- But who is responsible for releasing the memory?
- This leads us to a very important concept: *ownership*
- The part of the code that is responsible for releasing the resource is said to *own* that resource.

Lifetime & Special Member Functions

Problems

- In the example class `Array` we acquire dynamic memory at construction.
- Since there is no other part of the code that has direct access to the resource, it is quite natural that `Array` has the *ownership*.
- But when should `data` be deallocated?
- We want said data to exist during the full *lifetime* of the array, meaning: as long as the array exists, the memory must exist (otherwise the class will not work as intended).
- Therefore it is quite natural to deallocate the memory once the `Array` object is destroyed.
- So we *acquire* the resource at construction and *release* the resource at destruction. This is commonly refered to as RAII (*Resource Acquisition Is Initialization*).

Lifetime & Special Member Functions

RAII

```
1  class Array
2  {
3  public:
4    explicit Array(std::size_t size)                // constructor
5      : size { size }, data { new int[size] }
6    { }
7
8    ~Array()
9    {
10     delete[] data;
11   }
12
13   // ...
14
15 private:
16   std::size_t size;
17   int* data;
18 };
```

Lifetime & Special Member Functions

RAII

```
1  class Array
2  {
3  public:
4    explicit Array(std::size_t size)                // constructor
5      : size { size }, data { new int[size] }
6    { }
7
8    ~Array()
9    {
10     delete[] data;
11   }
12
13   // ...
14
15 private:
16   std::size_t size;
17   int* data;
18 };
```

Destructor: is called during *destruction*

Lifetime & Special Member Functions

RAII

```
1  class Array
2  {
3  public:
4    explicit Array(std::size_t size)
5      : size { size }, data { new int[size] }
6    { }
7
8    ~Array()
9    {
10     delete[] data;
11   }
12
13   // ...
14
15 private:
16   std::size_t size;
17   int* data;
18 };
```

Acquire resource in constructor

Release resource in destructor

Lifetime & Special Member Functions

Problems

- We have now solved the first problem we identified.
- Are there more?
- Yes there are! It is related to how the compiler handle copies of `Array` objects.

Lifetime & Special Member Functions

Copies

```
1  Array create(int a, int b, int c)
2  {
3    Array result { 3 };
4    result[0] = a;
5    result[1] = b;
6    result[2] = c;
7    return result;
8  }
9
10 int main()
11 {
12   Array array { create(1, 2, 3) };
13   // ...
14 }
```

Lifetime & Special Member Functions

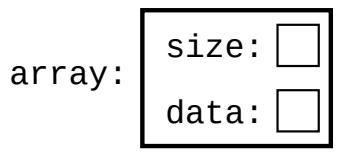
Copies

```
1  Array create(int a, int b, int c)
2  {
3    Array result { 3 };
4    result[0] = a;
5    result[1] = b;
6    result[2] = c;
7    return result;
8  }
9
10 int main()
11 {
12   Array array { create(1, 2, 3) };
13   // ...
14 }
```

Lifetime & Special Member Functions

Copies

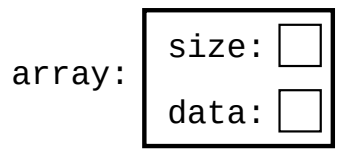
```
1  Array create(int a, int b, int c)
2  {
3    Array result { 3 };
4    result[0] = a;
5    result[1] = b;
6    result[2] = c;
7    return result;
8  }
9
10 int main()
11 {
12   Array array { create(1, 2, 3) };
13   // ...
14 }
```



Lifetime & Special Member Functions

Copies

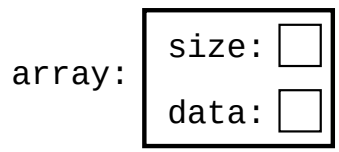
```
1  Array create(int a, int b, int c)
2  {
3    Array result { 3 };
4    result[0] = a;
5    result[1] = b;
6    result[2] = c;
7    return result;
8  }
9
10 int main()
11 {
12   Array array { create(1, 2, 3) };
13   // ...
14 }
```



Lifetime & Special Member Functions

Copies

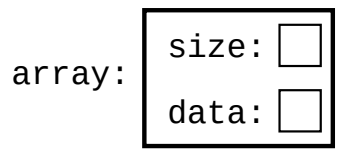
```
1  Array create(int a, int b, int c)
2  {
3    Array result { 3 };
4    result[0] = a;
5    result[1] = b;
6    result[2] = c;
7    return result;
8  }
9
10 int main()
11 {
12   Array array { create(1, 2, 3) };
13   // ...
14 }
```



Lifetime & Special Member Functions

Copies

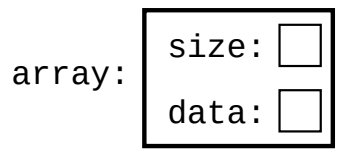
```
1  Array create(int a, int b, int c)
2  {
3    Array result { 3 };
4    result[0] = a;
5    result[1] = b;
6    result[2] = c;
7    return result;
8  }
9
10 int main()
11 {
12   Array array { create(1, 2, 3) };
13   // ...
14 }
```



Lifetime & Special Member Functions

Copies

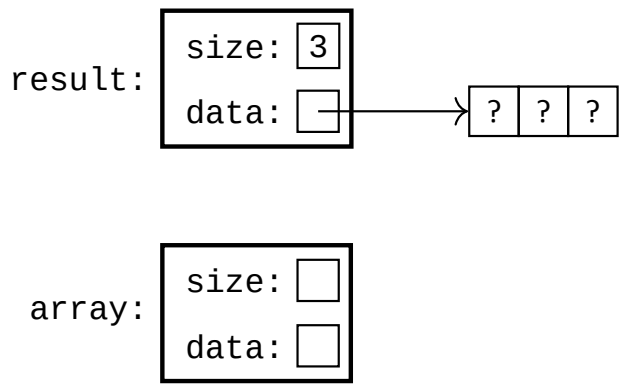
```
1  Array create(int a, int b, int c)
2  {
3    Array result { 3 };
4    result[0] = a;
5    result[1] = b;
6    result[2] = c;
7    return result;
8  }
9
10 int main()
11 {
12   Array array { create(1, 2, 3) };
13   // ...
14 }
```



Lifetime & Special Member Functions

Copies

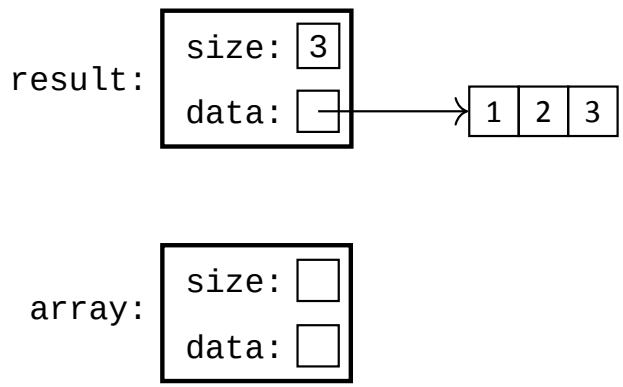
```
1  Array create(int a, int b, int c)
2  {
3    Array result { 3 };
4    result[0] = a;
5    result[1] = b;
6    result[2] = c;
7    return result;
8  }
9
10 int main()
11 {
12   Array array { create(1, 2, 3) };
13   // ...
14 }
```



Lifetime & Special Member Functions

Copies

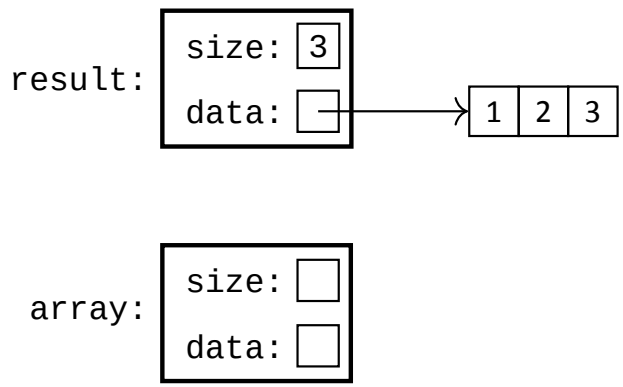
```
1  Array create(int a, int b, int c)
2  {
3    Array result { 3 };
4    result[0] = a;
5    result[1] = b;
6    result[2] = c;
7    return result;
8  }
9
10 int main()
11 {
12   Array array { create(1, 2, 3) };
13   // ...
14 }
```



Lifetime & Special Member Functions

Copies

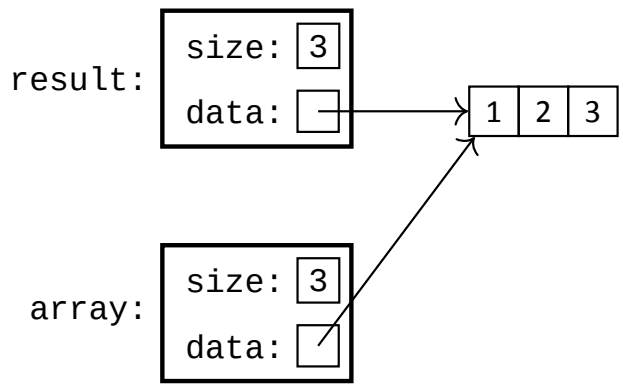
```
1  Array create(int a, int b, int c)
2  {
3    Array result { 3 };
4    result[0] = a;
5    result[1] = b;
6    result[2] = c;
7    return result;
8  }
9
10 int main()
11 {
12   Array array { create(1, 2, 3) };
13   // ...
14 }
```



Lifetime & Special Member Functions

Copies

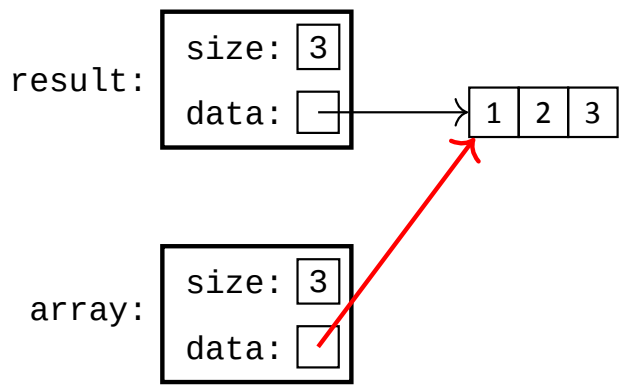
```
1  Array create(int a, int b, int c)
2  {
3    Array result { 3 };
4    result[0] = a;
5    result[1] = b;
6    result[2] = c;
7    return result;
8  }
9
10 int main()
11 {
12   Array array { create(1, 2, 3) };
13   // ...
14 }
```



Lifetime & Special Member Functions

Copies

```
1  Array create(int a, int b, int c)
2  {
3    Array result { 3 };
4    result[0] = a;
5    result[1] = b;
6    result[2] = c;
7    return result;
8  }
9
10 int main()
11 {
12   Array array { create(1, 2, 3) };
13   // ...
14 }
```



Lifetime & Special Member Functions

Copies

- When copying class types, the default behaviour is to copy the data members.
- This proves to be a problem when for example copying resources such as pointers, since we need to do something special to actually copy them.
- In this example we need to allocate a new array, but the compiler only makes a copy of the *pointer*, not the underlying data.
- This is called a *shallow* copy.

Lifetime & Special Member Functions

Copies

- In order to make a *deep* copy, i.e. a copy where we actually copy the resource (memory in this case) we need to tell the compiler how that is done.
- We do this by creating a special constructor called the *copy constructor*
- The copy constructor is the constructor with the following signature:

```
Array(Array const& other)
```

Lifetime & Special Member Functions

Copy constructor

```
1  class Array
2  {
3  public:
4    // ...
5
6    Array(Array const& other) // copy constructor
7      : size { other.size },
8        data { new int[size] }
9    {
10     // copy everything from other into our own allocation
11     for (std::size_t i { 0 }; i < size; ++i)
12       data[i] = other.data[i];
13   }
14
15   // ...
16 private:
17   std::size_t size;
18   int* data;
19 };
```

Lifetime & Special Member Functions

Copies

```
1  Array create(int a, int b, int c)
2  {
3    Array result { 3 };
4    result[0] = a;
5    result[1] = b;
6    result[2] = c;
7    return result;
8  }
9
10 int main()
11 {
12   Array array { create(1, 2, 3) };
13   // ...
14 }
```

Lifetime & Special Member Functions

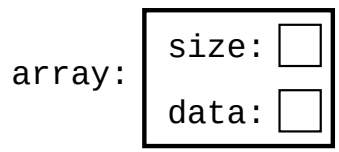
Copies

```
1  Array create(int a, int b, int c)
2  {
3    Array result { 3 };
4    result[0] = a;
5    result[1] = b;
6    result[2] = c;
7    return result;
8  }
9
10 int main()
11 {
12   Array array { create(1, 2, 3) };
13   // ...
14 }
```

Lifetime & Special Member Functions

Copies

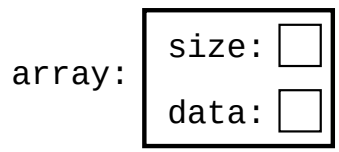
```
1  Array create(int a, int b, int c)
2  {
3    Array result { 3 };
4    result[0] = a;
5    result[1] = b;
6    result[2] = c;
7    return result;
8  }
9
10 int main()
11 {
12   Array array { create(1, 2, 3) };
13   // ...
14 }
```



Lifetime & Special Member Functions

Copies

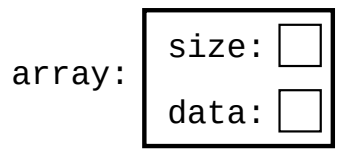
```
1  Array create(int a, int b, int c)
2  {
3    Array result { 3 };
4    result[0] = a;
5    result[1] = b;
6    result[2] = c;
7    return result;
8  }
9
10 int main()
11 {
12   Array array { create(1, 2, 3) };
13   // ...
14 }
```



Lifetime & Special Member Functions

Copies

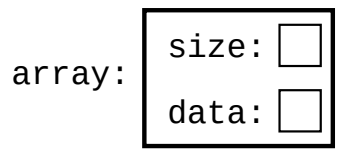
```
1  Array create(int a, int b, int c)
2  {
3    Array result { 3 };
4    result[0] = a;
5    result[1] = b;
6    result[2] = c;
7    return result;
8  }
9
10 int main()
11 {
12   Array array { create(1, 2, 3) };
13   // ...
14 }
```



Lifetime & Special Member Functions

Copies

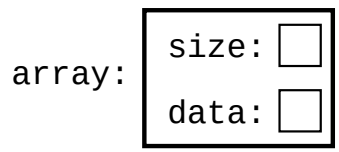
```
1  Array create(int a, int b, int c)
2  {
3    Array result { 3 };
4    result[0] = a;
5    result[1] = b;
6    result[2] = c;
7    return result;
8  }
9
10 int main()
11 {
12   Array array { create(1, 2, 3) };
13   // ...
14 }
```



Lifetime & Special Member Functions

Copies

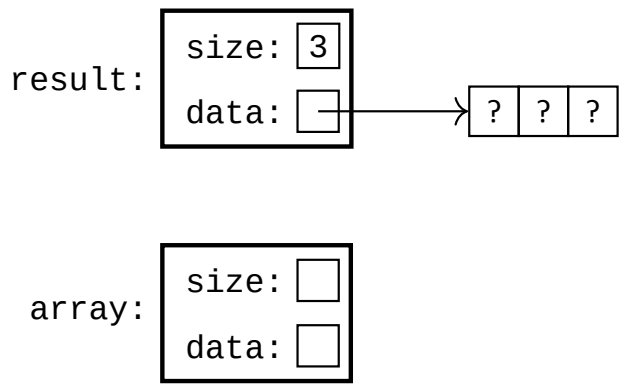
```
1  Array create(int a, int b, int c)
2  {
3    Array result { 3 };
4    result[0] = a;
5    result[1] = b;
6    result[2] = c;
7    return result;
8  }
9
10 int main()
11 {
12   Array array { create(1, 2, 3) };
13   // ...
14 }
```



Lifetime & Special Member Functions

Copies

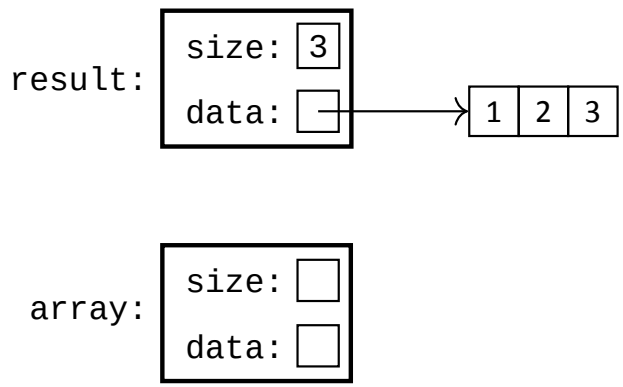
```
1  Array create(int a, int b, int c)
2  {
3    Array result { 3 };
4    result[0] = a;
5    result[1] = b;
6    result[2] = c;
7    return result;
8  }
9
10 int main()
11 {
12   Array array { create(1, 2, 3) };
13   // ...
14 }
```



Lifetime & Special Member Functions

Copies

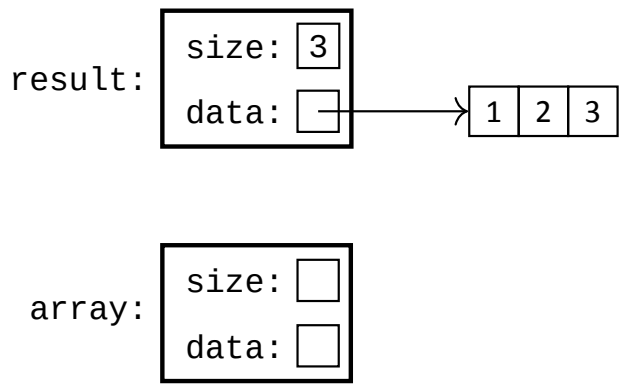
```
1  Array create(int a, int b, int c)
2  {
3    Array result { 3 };
4    result[0] = a;
5    result[1] = b;
6    result[2] = c;
7    return result;
8  }
9
10 int main()
11 {
12   Array array { create(1, 2, 3) };
13   // ...
14 }
```



Lifetime & Special Member Functions

Copies

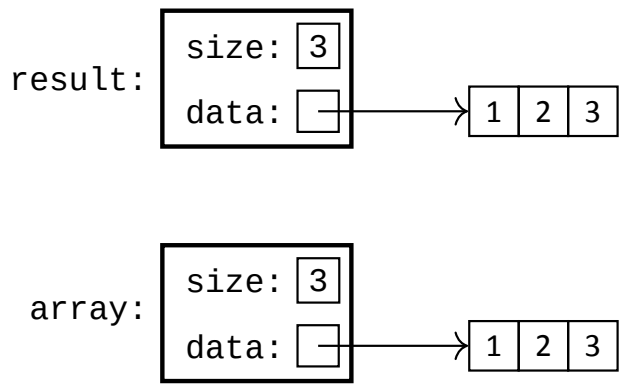
```
1  Array create(int a, int b, int c)
2  {
3    Array result { 3 };
4    result[0] = a;
5    result[1] = b;
6    result[2] = c;
7    return result;
8  }
9
10 int main()
11 {
12   Array array { create(1, 2, 3) };
13   // ...
14 }
```



Lifetime & Special Member Functions

Copies

```
1  Array create(int a, int b, int c)
2  {
3    Array result { 3 };
4    result[0] = a;
5    result[1] = b;
6    result[2] = c;
7    return result;
8  }
9
10 int main()
11 {
12   Array array { create(1, 2, 3) };
13   // ...
14 }
```



Lifetime & Special Member Functions

Copy assignment

- There is a second situation where copying can occur: namely during assignment
- Remember that the constructor only runs when the object is constructed
- So the copy constructor only works when we are trying to copy an object into something that is currently being constructed
- But with assignment we are dealing with already existing objects

Lifetime & Special Member Functions

Copy assignment

- This also means that the circumstances during copy assignment is slightly different compared to the copy constructor
- Specifically: When assigning to an object we are going to *overwrite* the previous resource, so we have to release that before assigning a new resource.
- There might be issues if we assign to ourselves (i.e. `x = x`).
- We have to keep this in mind when we implement our *copy assignment operator*.

Lifetime & Special Member Functions

Copy assignment

```
1 Array a { create(1, 2, 3) };
2 Array b { create(2, 3, 4) };
3
4 b = a; // copy a into b
```

Lifetime & Special Member Functions

Copy assignment operator

```
1  class Array
2  {
3  public:
4    // ...
5    Array& operator=(Array const& other)
6    {
7      if (this != &other)
8      {
9        delete[] data;
10       size = other.size;
11       data = new int[size];
12       for (std::size_t i { 0 }; i < size; ++i)
13         data[i] = other.data[i];
14     }
15     return *this;
16   }
17   // ...
18 private:
19   std::size_t size;
20   int* data;
21 };
```

Avoid code like this

Lifetime & Special Member Functions

Copy assignment operator

- Here we are utilizing operator overloading to define the *copy assignment operator*
- It makes sure that:
  1. self-assignment does nothing
  2. the previous memory is deleted
  3. `other` gets copied into `*this`
- However, there is *a lot* of code duplication here.
- Specifically: we are duplicating the work of both the copy constructor *and* the destructor.
- We can do better...

Lifetime & Special Member Functions

Copy-and-swap idiom

```
1  class Array
2  {
3  public:
4    // ...
5    Array& operator=(Array const& other)
6    {
7      // reuse copy constructor
8      Array copy { other };
9
10     // swap the members
11     std::swap(size, other.size);
12     std::swap(data, other.data);
13
14     // copy is destroyed which releases the previous data
15     return *this;
16   }
17   // ...
18 private:
19   std::size_t size;
20   int* data;
21 };
```

Prefer this!

Lifetime & Special Member Functions

Copy-and-swap idiom

- The copy-and-swap idiom states that instead of doing all the work manually (again) we re-use the copy constructor and destructor
- We utilize the fact that a local variable (`copy` in the example) is created and destroyed within the scope of the function
- Therefore we can copy `other` using the copy constructor and then *swap* all data members
- This means that `copy` now contains the *previous* state of `*this`
- Because of this the previous state will be destroyed once `copy` falls out of scope (i.e. when we return).

Lifetime & Special Member Functions

Opportunity for optimization

```
1  Array modify(Array array)
2  {
3    ++array[0];
4    return array;
5  }
6
7  int main()
8  {
9    Array a { create(1, 2, 3) };
10   Array b { modify(a) };
11 }
```

Lifetime & Special Member Functions

Opportunity for optimization

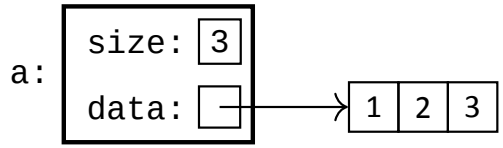
Construct a

```
1  Array modify(Array array)
2  {
3    ++array[0];
4    return array;
5  }
6
7  int main()
8  {
9    Array a { create(1, 2, 3) };
10   Array b { modify(a) };
11 }
```

Lifetime & Special Member Functions

Opportunity for optimization

Construct a

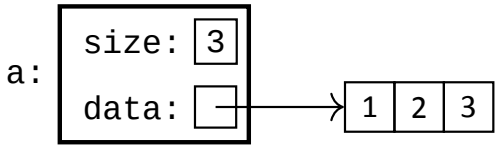



```
1  Array modify(Array array)
2  {
3    ++array[0];
4    return array;
5  }
6
7  int main()
8  {
9    Array a { create(1, 2, 3) };
10   Array b { modify(a) };
11 }
```

Lifetime & Special Member Functions

Opportunity for optimization

We begin construction of `b`. For now we can only allocate `b` on the stack, because we need to call `modify()` before we can actually initialize `b`.

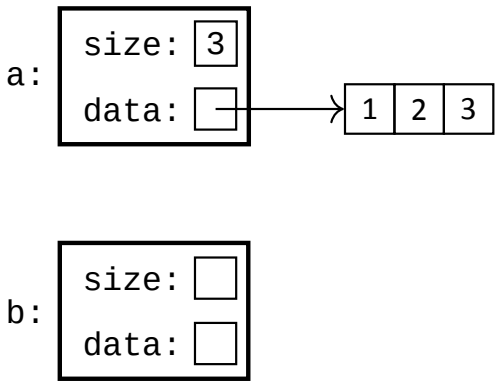



```
1  Array modify(Array array)
2  {
3    ++array[0];
4    return array;
5  }
6
7  int main()
8  {
9    Array a { create(1, 2, 3) };
10   Array b { modify(a) };
11 }
```

Lifetime & Special Member Functions

Opportunity for optimization

We begin construction of `b`. For now we can only allocate `b` on the stack, because we need to call `modify()` before we can actually initialize `b`.

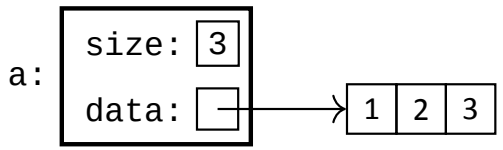


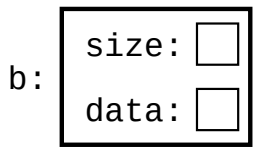

```
1  Array modify(Array array)
2  {
3    ++array[0];
4    return array;
5  }
6
7  int main()
8  {
9    Array a { create(1, 2, 3) };
10   Array b { modify(a) };
11 }
```

Lifetime & Special Member Functions

Opportunity for optimization

Call `modify()`



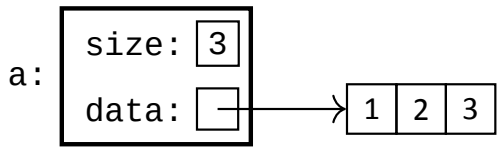


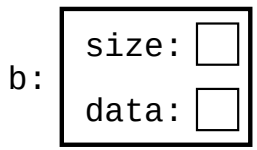

```
1  Array modify(Array array)
2  {
3    ++array[0];
4    return array;
5  }
6
7  int main()
8  {
9    Array a { create(1, 2, 3) };
10   Array b { modify(a) };
11 }
```

Lifetime & Special Member Functions

Opportunity for optimization

We have to copy a into `array`





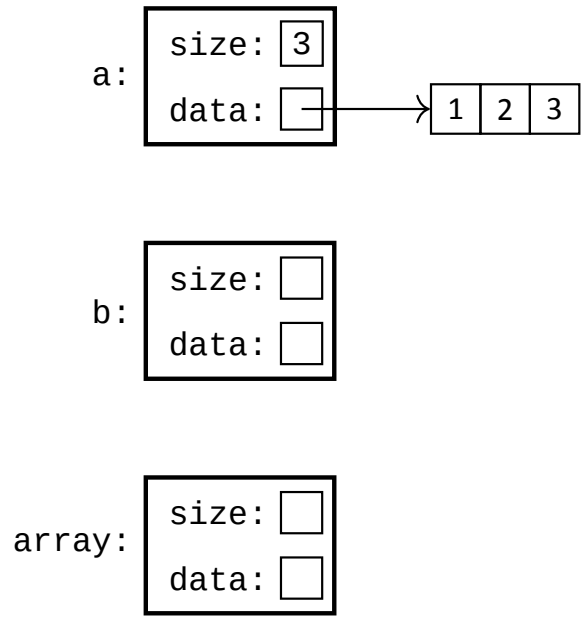

```
1  Array modify(Array array)
2  {
3    ++array[0];
4    return array;
5  }
6
7  int main()
8  {
9    Array a { create(1, 2, 3) };
10   Array b { modify(a) };
11 }
```

Lifetime & Special Member Functions

Opportunity for optimization

We have to copy a into `array`

```
1  Array modify(Array array)
2  {
3    ++array[0];
4    return array;
5  }
6
7  int main()
8  {
9    Array a { create(1, 2, 3) };
10   Array b { modify(a) };
11 }
```

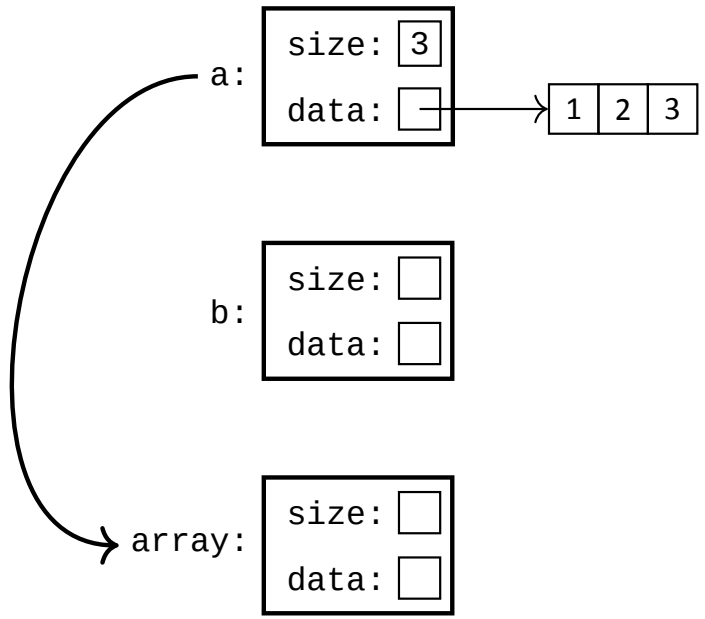


Lifetime & Special Member Functions

Opportunity for optimization

We have to copy a into `array`

```
1  Array modify(Array array)
2  {
3    ++array[0];
4    return array;
5  }
6
7  int main()
8  {
9    Array a { create(1, 2, 3) };
10   Array b { modify(a) };
11 }
```

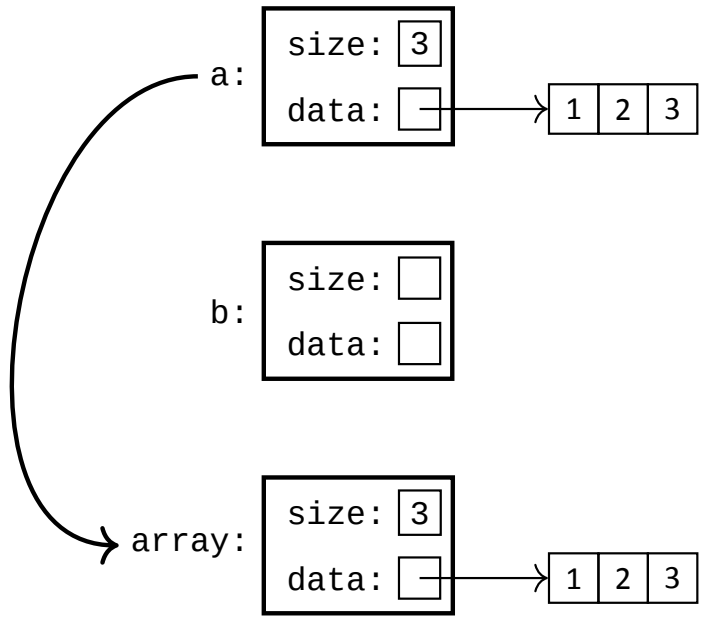


Lifetime & Special Member Functions

Opportunity for optimization

We have to copy a into `array`

```
1  Array modify(Array array)
2  {
3    ++array[0];
4    return array;
5  }
6
7  int main()
8  {
9    Array a { create(1, 2, 3) };
10   Array b { modify(a) };
11 }
```

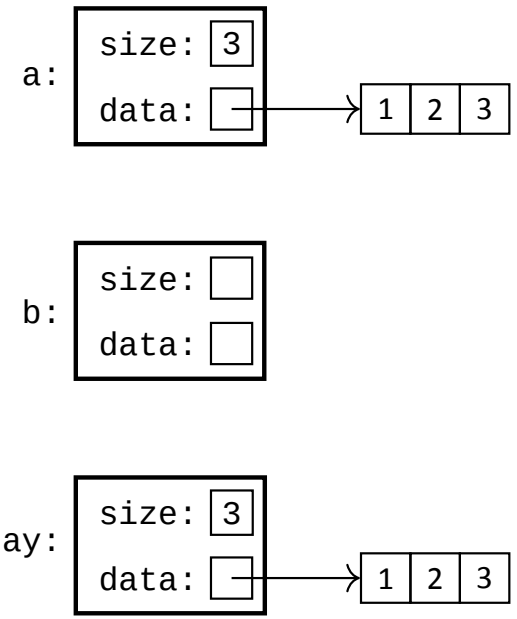


Lifetime & Special Member Functions

Opportunity for optimization

Modify the *local* array

```
1  Array modify(Array array)
2  {
3    ++array[0];
4    return array;
5  }
6
7  int main()
8  {
9    Array a { create(1, 2, 3) };
10   Array b { modify(a) };
11 }
```

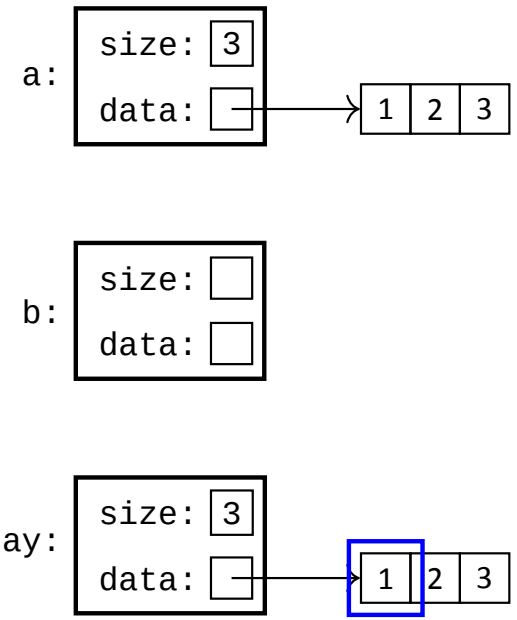


Lifetime & Special Member Functions

Opportunity for optimization

Modify the *local* array

```
1  Array modify(Array array)
2  {
3    ++array[0];
4    return array;
5  }
6
7  int main()
8  {
9    Array a { create(1, 2, 3) };
10   Array b { modify(a) };
11 }
```

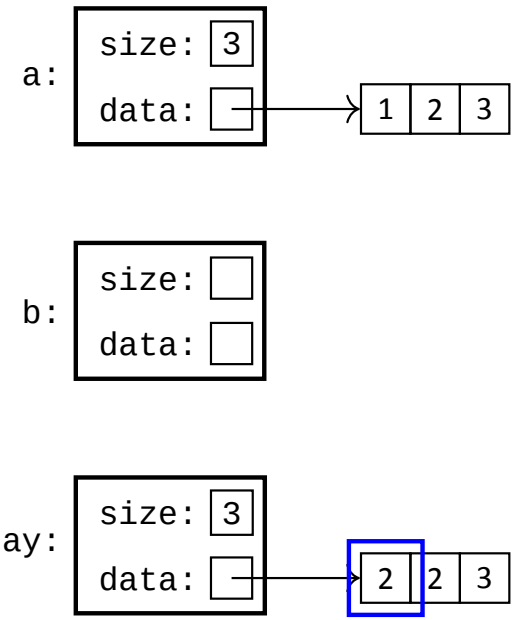


Lifetime & Special Member Functions

Opportunity for optimization

Modify the *local* array

```
1  Array modify(Array array)
2  {
3    ++array[0];
4    return array;
5  }
6
7  int main()
8  {
9    Array a { create(1, 2, 3) };
10   Array b { modify(a) };
11 }
```

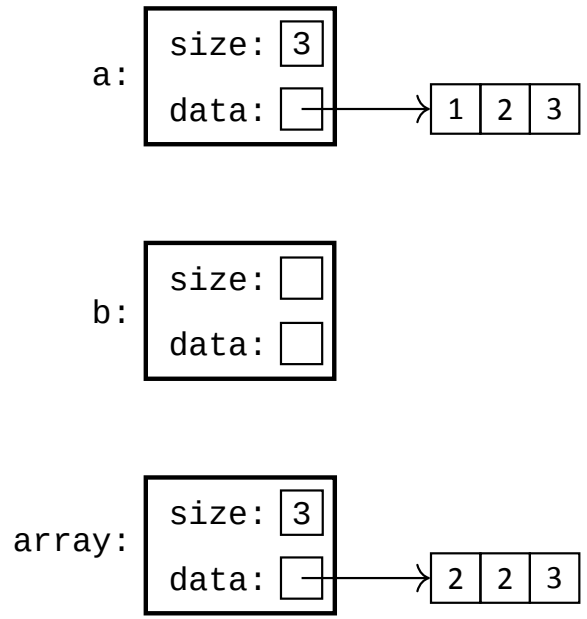


Lifetime & Special Member Functions

Opportunity for optimization

Return `array` to `main()`

```
1  Array modify(Array array)
2  {
3    ++array[0];
4    return array;
5  }
6
7  int main()
8  {
9    Array a { create(1, 2, 3) };
10   Array b { modify(a) };
11 }
```

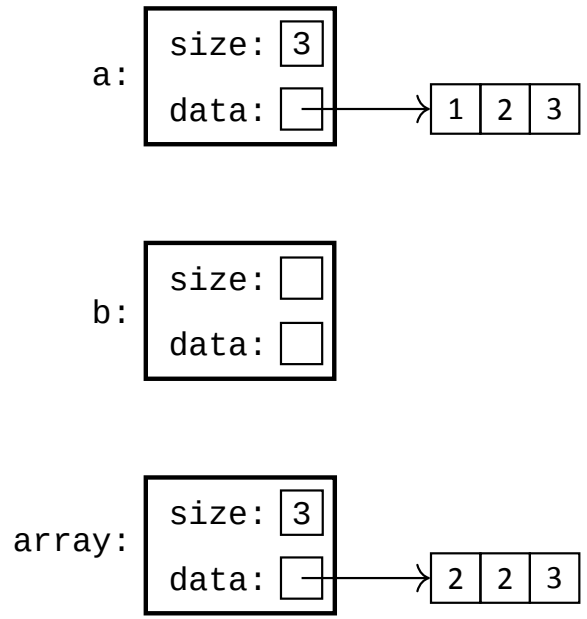


Lifetime & Special Member Functions

Opportunity for optimization

`array` will be destroyed once we've returned to `main()`. Because of this we have to copy `array` into a temporary object that is available in `main` *after* we've returned.

```
1  Array modify(Array array)
2  {
3    ++array[0];
4    return array;
5  }
6
7  int main()
8  {
9    Array a { create(1, 2, 3) };
10   Array b { modify(a) };
11 }
```

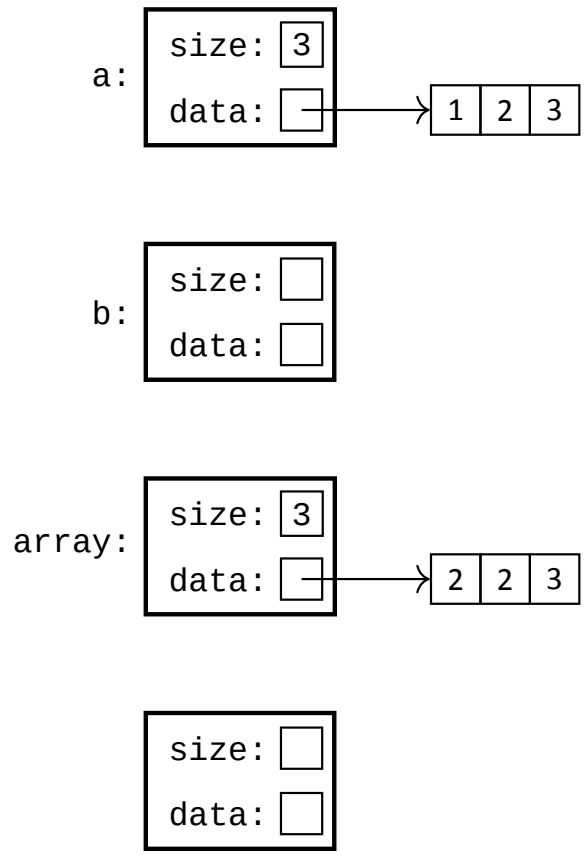


Lifetime & Special Member Functions

Opportunity for optimization

`array` will be destroyed once we've returned to `main()`. Because of this we have to copy `array` into a temporary object that is available in `main` *after* we've returned.

```
1  Array modify(Array array)
2  {
3    ++array[0];
4    return array;
5  }
6
7  int main()
8  {
9    Array a { create(1, 2, 3) };
10   Array b { modify(a) };
11 }
```

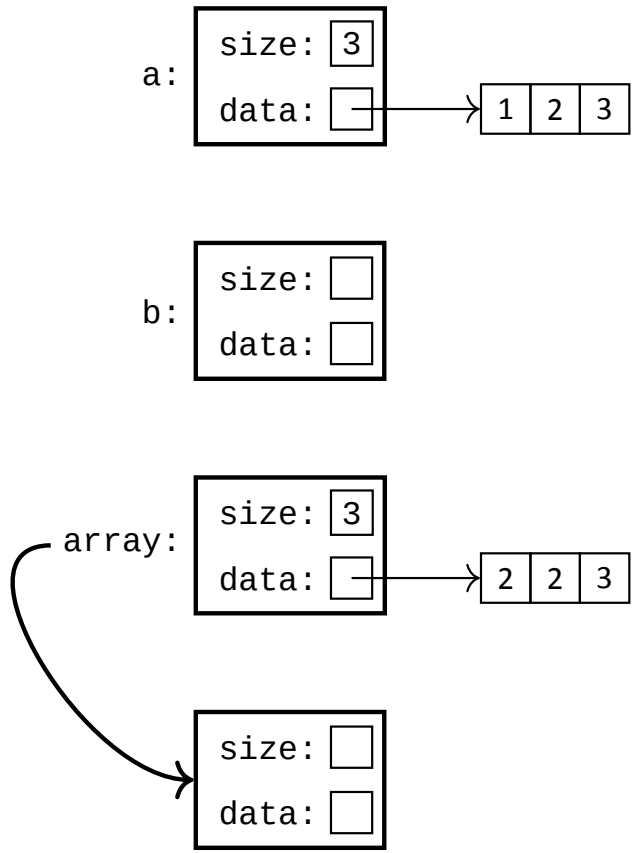


Lifetime & Special Member Functions

Opportunity for optimization

`array` will be destroyed once we've returned to `main()`. Because of this we have to copy `array` into a temporary object that is available in `main` *after* we've returned.

```
1  Array modify(Array array)
2  {
3    ++array[0];
4    return array;
5  }
6
7  int main()
8  {
9    Array a { create(1, 2, 3) };
10   Array b { modify(a) };
11 }
```

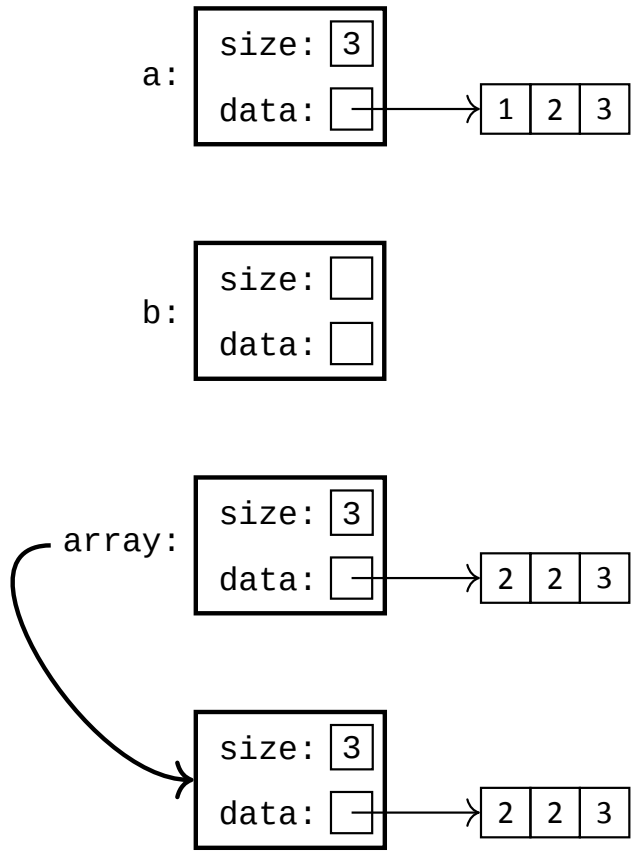


Lifetime & Special Member Functions

Opportunity for optimization

`array` will be destroyed once we've returned to `main()`. Because of this we have to copy `array` into a temporary object that is available in `main` *after* we've returned.

```
1  Array modify(Array array)
2  {
3    ++array[0];
4    return array;
5  }
6
7  int main()
8  {
9    Array a { create(1, 2, 3) };
10   Array b { modify(a) };
11 }
```

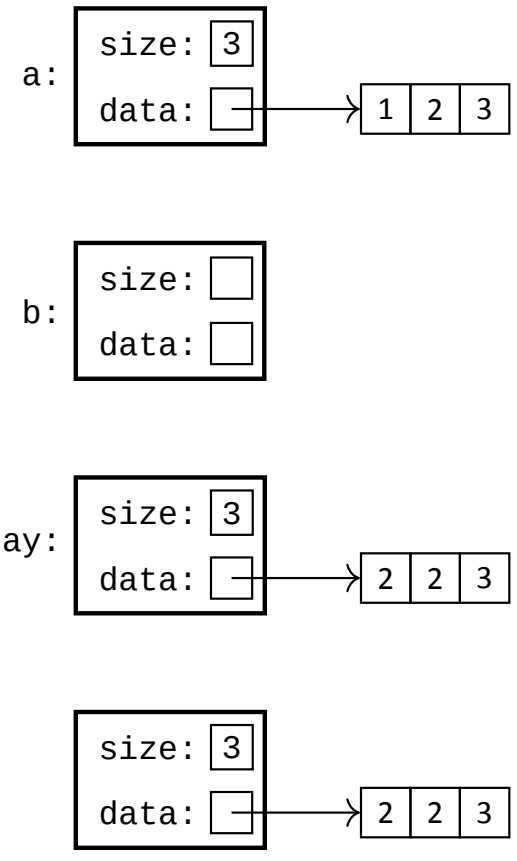


Lifetime & Special Member Functions

Opportunity for optimization

Destroy `array`

```
1  Array modify(Array array)
2  {
3    ++array[0];
4    return array;
5  }
6
7  int main()
8  {
9    Array a { create(1, 2, 3) };
10   Array b { modify(a) };
11 }
```

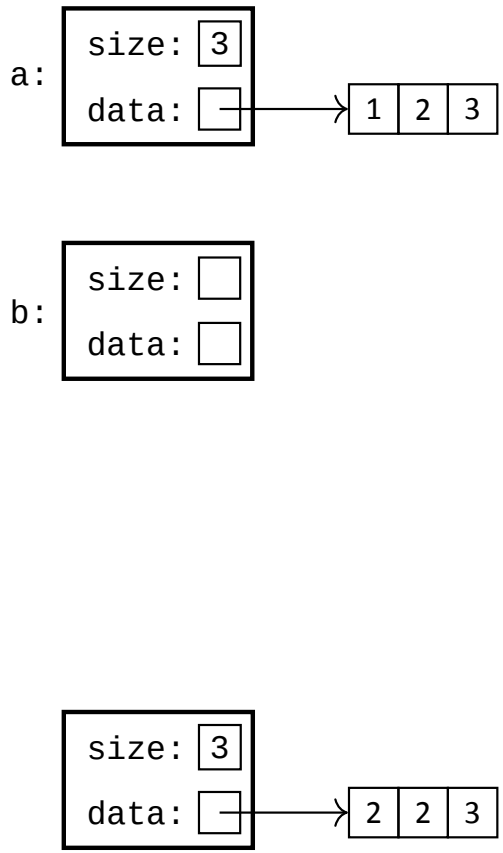


Lifetime & Special Member Functions

Opportunity for optimization

Destroy `array`

```
1  Array modify(Array array)
2  {
3    ++array[0];
4    return array;
5  }
6
7  int main()
8  {
9    Array a { create(1, 2, 3) };
10   Array b { modify(a) };
11 }
```

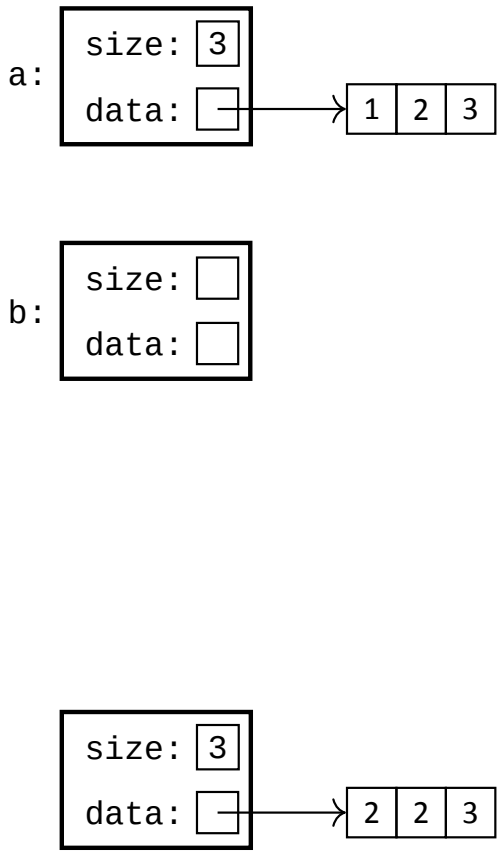


Lifetime & Special Member Functions

Opportunity for optimization

Finish construction of b by copying the temporary object

```
1  Array modify(Array array)
2  {
3    ++array[0];
4    return array;
5  }
6
7  int main()
8  {
9    Array a { create(1, 2, 3) };
10   Array b { modify(a) };
11 }
```

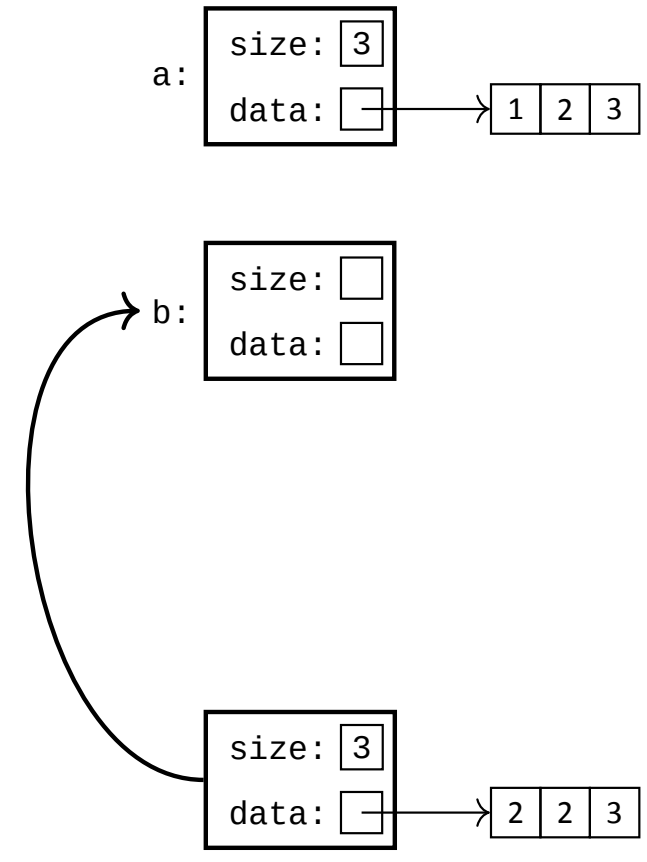


Lifetime & Special Member Functions

Opportunity for optimization

Finish construction of b by copying the temporary object

```
1  Array modify(Array array)
2  {
3    ++array[0];
4    return array;
5  }
6
7  int main()
8  {
9    Array a { create(1, 2, 3) };
10   Array b { modify(a) };
11 }
```

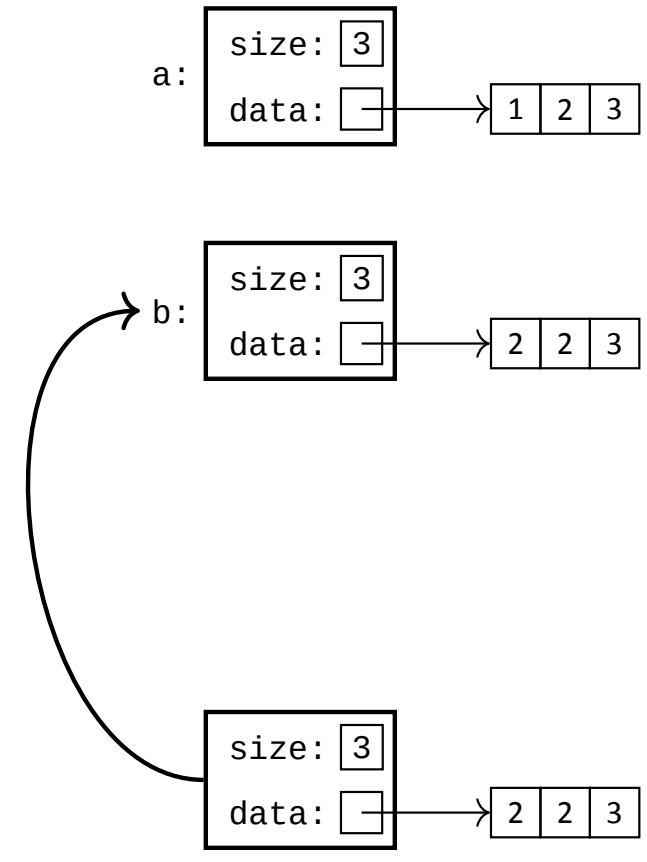


Lifetime & Special Member Functions

Opportunity for optimization

Finish construction of b by copying the temporary object

```
1  Array modify(Array array)
2  {
3    ++array[0];
4    return array;
5  }
6
7  int main()
8  {
9    Array a { create(1, 2, 3) };
10   Array b { modify(a) };
11 }
```

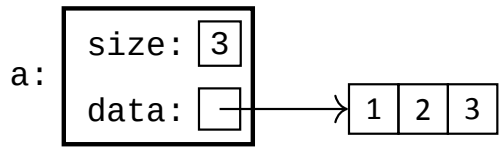


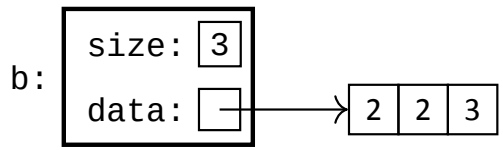
Lifetime & Special Member Functions

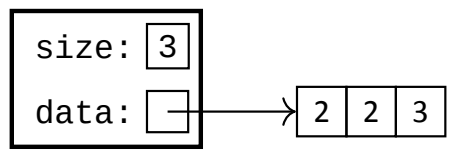
Opportunity for optimization

Destroy the temporary object

```
1  Array modify(Array array)
2  {
3    ++array[0];
4    return array;
5  }
6
7  int main()
8  {
9    Array a { create(1, 2, 3) };
10   Array b { modify(a) };
11 }
```





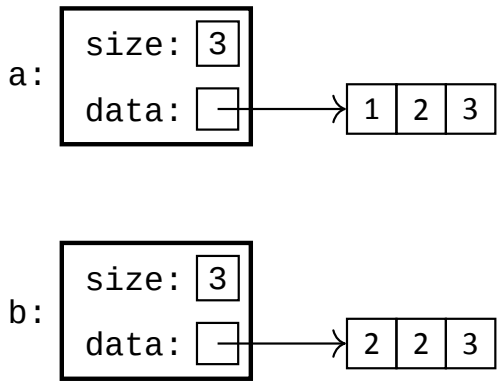


Lifetime & Special Member Functions

Opportunity for optimization

Destroy the temporary object

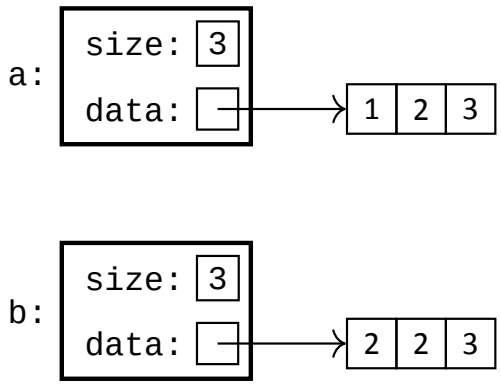
```
1  Array modify(Array array)
2  {
3    ++array[0];
4    return array;
5  }
6
7  int main()
8  {
9    Array a { create(1, 2, 3) };
10   Array b { modify(a) };
11 }
```



Lifetime & Special Member Functions

Opportunity for optimization

We are done!

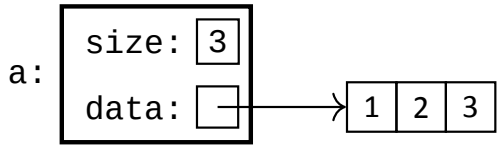


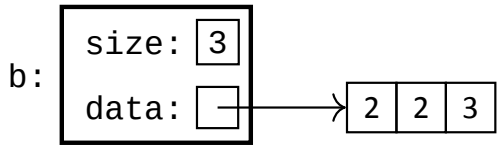

```
1  Array modify(Array array)
2  {
3    ++array[0];
4    return array;
5  }
6
7  int main()
8  {
9    Array a { create(1, 2, 3) };
10   Array b { modify(a) };
11 }
```

Lifetime & Special Member Functions

Opportunity for optimization

But didn't we make a lot of unneccesary dynamic allocations?





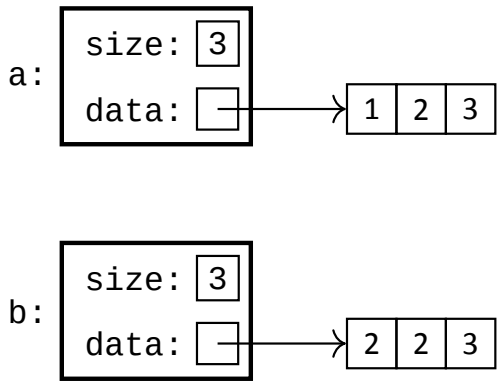

```
1  Array modify(Array array)
2  {
3    ++array[0];
4    return array;
5  }
6
7  int main()
8  {
9    Array a { create(1, 2, 3) };
10   Array b { modify(a) };
11 }
```

Lifetime & Special Member Functions

Opportunity for optimization

We made *four* dynamic allocations!

```
1  Array modify(Array array)
2  {
3    ++array[0];
4    return array;
5  }
6
7  int main()
8  {
9    Array a { create(1, 2, 3) };
10   Array b { modify(a) };
11 }
```



Lifetime & Special Member Functions

We can do better!

Lifetime & Special Member Functions

Opportunity for optimization

```
1  Array modify(Array array)
2  {
3    ++array[0];
4    return array;
5  }
6
7  int main()
8  {
9    Array a { create(1, 2, 3) };
10   Array b { modify(a) };
11 }
```

Lifetime & Special Member Functions

Opportunity for optimization

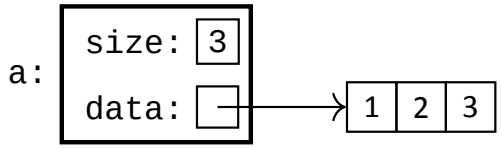
Construct a

```
1  Array modify(Array array)
2  {
3    ++array[0];
4    return array;
5  }
6
7  int main()
8  {
9    Array a { create(1, 2, 3) };
10   Array b { modify(a) };
11 }
```

Lifetime & Special Member Functions

Opportunity for optimization

Construct a

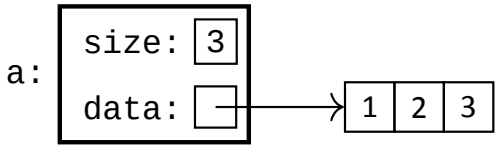



```
1  Array modify(Array array)
2  {
3    ++array[0];
4    return array;
5  }
6
7  int main()
8  {
9    Array a { create(1, 2, 3) };
10   Array b { modify(a) };
11 }
```

Lifetime & Special Member Functions

Opportunity for optimization

We begin construction of `b`. For now we can only allocate `b` on the stack, because we need to call `modify()` before we can actually initialize `b`.

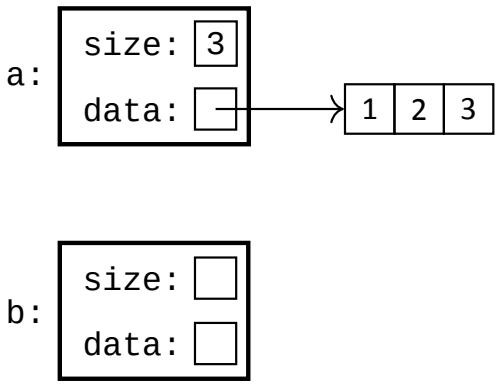



```
1  Array modify(Array array)
2  {
3    ++array[0];
4    return array;
5  }
6
7  int main()
8  {
9    Array a { create(1, 2, 3) };
10   Array b { modify(a) };
11 }
```

Lifetime & Special Member Functions

Opportunity for optimization

We begin construction of `b`. For now we can only allocate `b` on the stack, because we need to call `modify()` before we can actually initialize `b`.

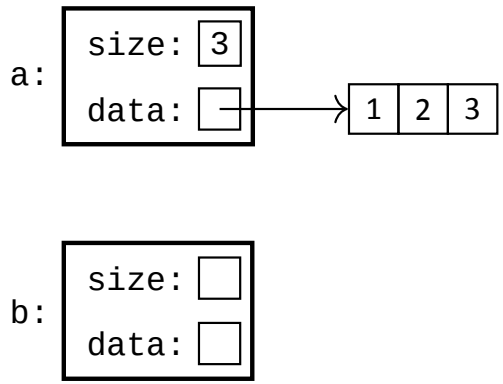



```
1  Array modify(Array array)
2  {
3    ++array[0];
4    return array;
5  }
6
7  int main()
8  {
9    Array a { create(1, 2, 3) };
10   Array b { modify(a) };
11 }
```

Lifetime & Special Member Functions

Opportunity for optimization

Call `modify()`



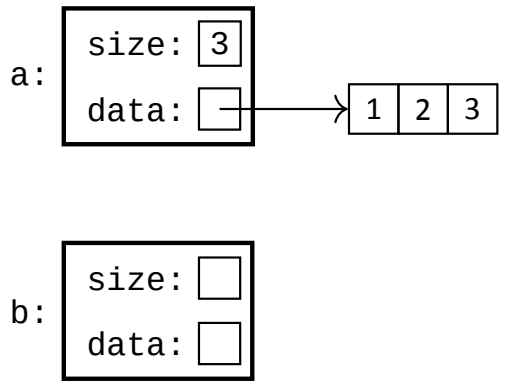

```
1  Array modify(Array array)
2  {
3    ++array[0];
4    return array;
5  }
6
7  int main()
8  {
9    Array a { create(1, 2, 3) };
10   Array b { modify(a) };
11 }
```

Lifetime & Special Member Functions

Opportunity for optimization

We **have** to copy a into `array`

```
1  Array modify(Array array)
2  {
3    ++array[0];
4    return array;
5  }
6
7  int main()
8  {
9    Array a { create(1, 2, 3) };
10   Array b { modify(a) };
11 }
```

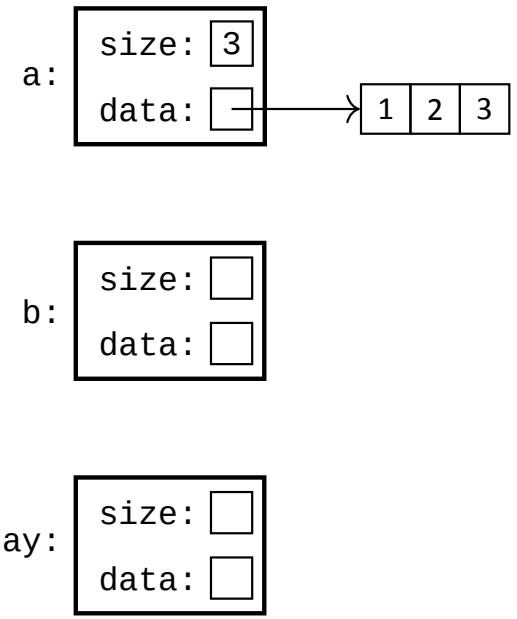


Lifetime & Special Member Functions

Opportunity for optimization

We **have** to copy a into `array`

```
1  Array modify(Array array)
2  {
3    ++array[0];
4    return array;
5  }
6
7  int main()
8  {
9    Array a { create(1, 2, 3) };
10   Array b { modify(a) };
11 }
```

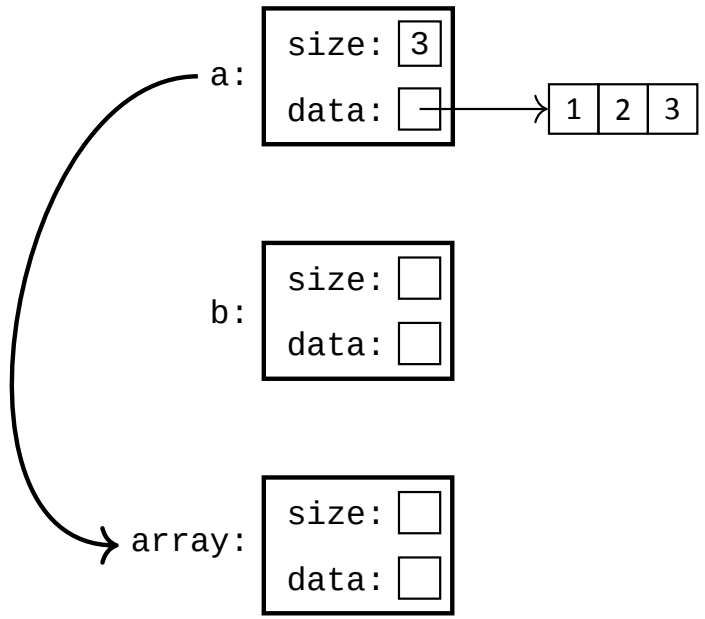


Lifetime & Special Member Functions

Opportunity for optimization

We **have** to copy a into `array`

```
1  Array modify(Array array)
2  {
3    ++array[0];
4    return array;
5  }
6
7  int main()
8  {
9    Array a { create(1, 2, 3) };
10   Array b { modify(a) };
11 }
```

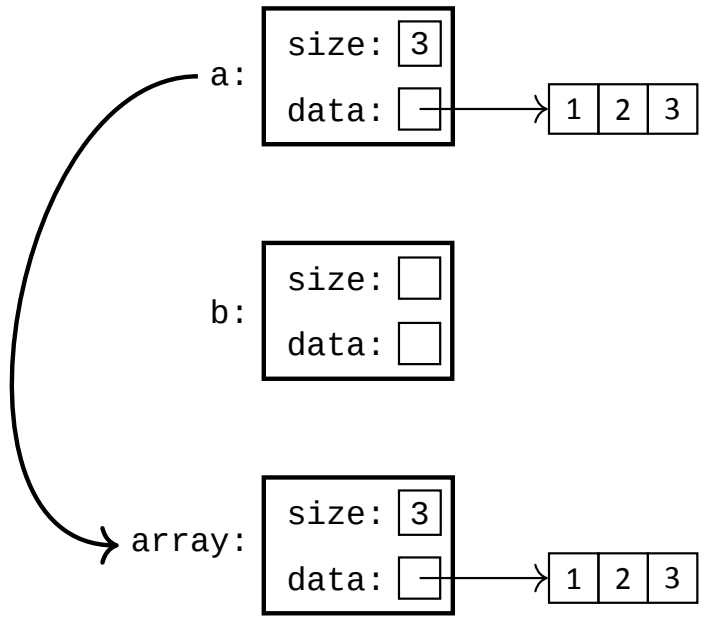


Lifetime & Special Member Functions

Opportunity for optimization

We **have** to copy a into `array`

```
1  Array modify(Array array)
2  {
3    ++array[0];
4    return array;
5  }
6
7  int main()
8  {
9    Array a { create(1, 2, 3) };
10   Array b { modify(a) };
11 }
```

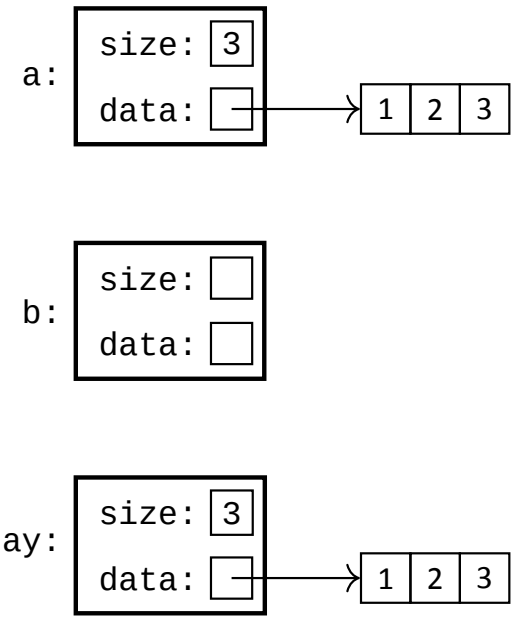


Lifetime & Special Member Functions

Opportunity for optimization

Modify the *local* array

```
1  Array modify(Array array)
2  {
3    ++array[0];
4    return array;
5  }
6
7  int main()
8  {
9    Array a { create(1, 2, 3) };
10   Array b { modify(a) };
11 }
```

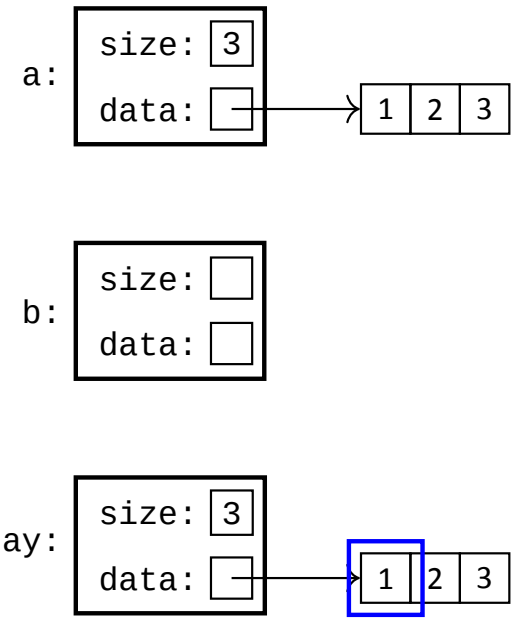


Lifetime & Special Member Functions

Opportunity for optimization

Modify the *local* array

```
1  Array modify(Array array)
2  {
3    ++array[0];
4    return array;
5  }
6
7  int main()
8  {
9    Array a { create(1, 2, 3) };
10   Array b { modify(a) };
11 }
```

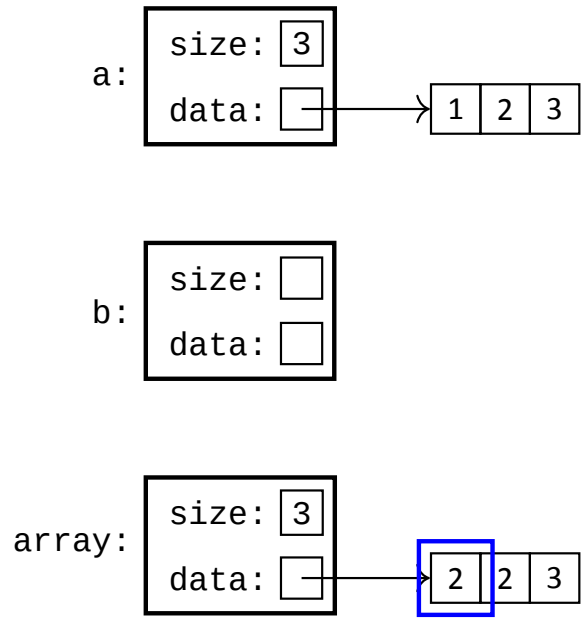


Lifetime & Special Member Functions

Opportunity for optimization

Modify the *local* array

```
1  Array modify(Array array)
2  {
3    ++array[0];
4    return array;
5  }
6
7  int main()
8  {
9    Array a { create(1, 2, 3) };
10   Array b { modify(a) };
11 }
```

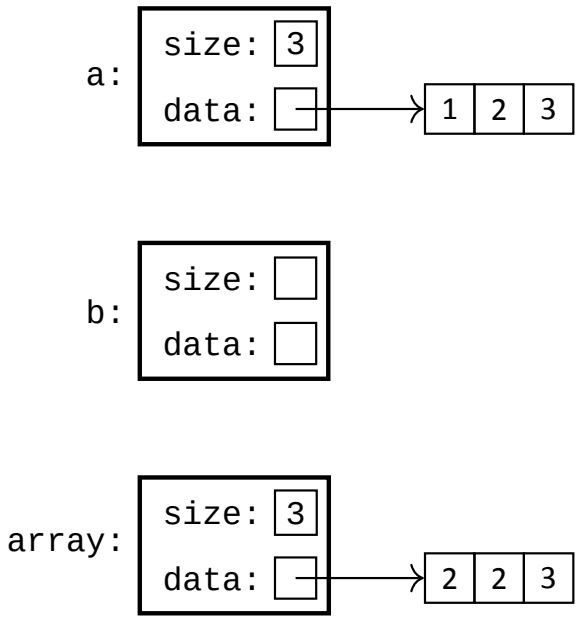


Lifetime & Special Member Functions

Opportunity for optimization

Return `array` to `main()`

```
1  Array modify(Array array)
2  {
3    ++array[0];
4    return array;
5  }
6
7  int main()
8  {
9    Array a { create(1, 2, 3) };
10   Array b { modify(a) };
11 }
```

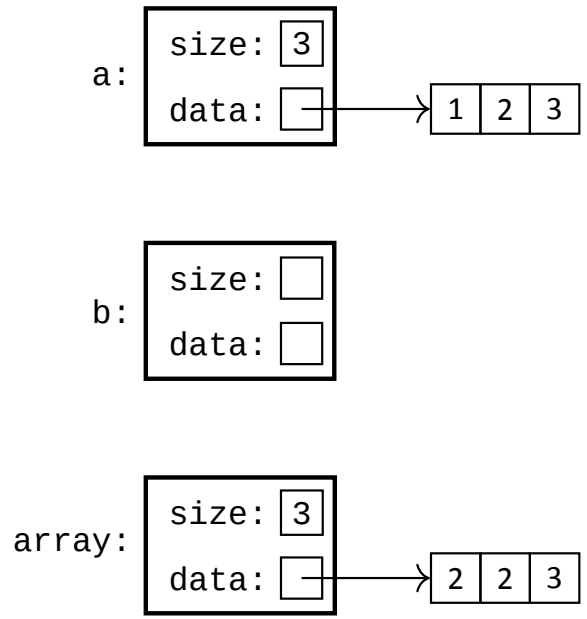


Lifetime & Special Member Functions

Opportunity for optimization

This is the first place where we can optimze: Instead of copying the array, we simply *move* it to the temporary.

```
1  Array modify(Array array)
2  {
3    ++array[0];
4    return array;
5  }
6
7  int main()
8  {
9    Array a { create(1, 2, 3) };
10   Array b { modify(a) };
11 }
```

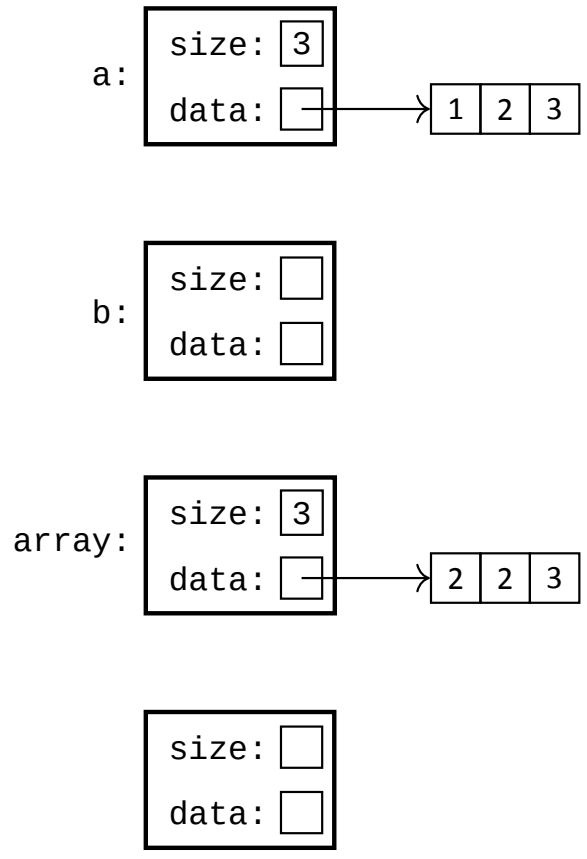


Lifetime & Special Member Functions

Opportunity for optimization

This is the first place where we can optimze: Instead of copying the array, we simply *move* it to the temporary.

```
1  Array modify(Array array)
2  {
3    ++array[0];
4    return array;
5  }
6
7  int main()
8  {
9    Array a { create(1, 2, 3) };
10   Array b { modify(a) };
11 }
```

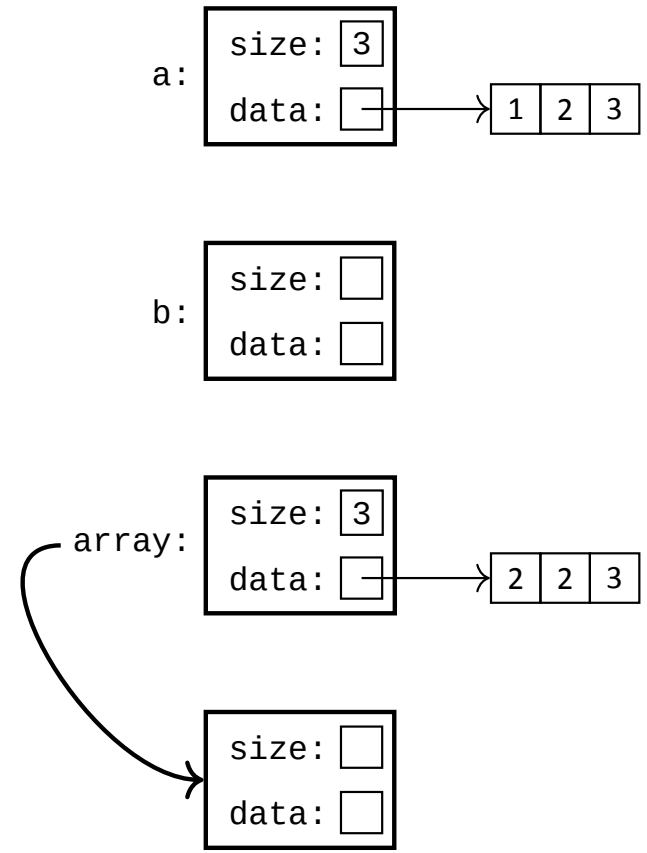


Lifetime & Special Member Functions

Opportunity for optimization

This is the first place where we can optimze: Instead of copying the array, we simply *move* it to the temporary.

```
1  Array modify(Array array)
2  {
3    ++array[0];
4    return array;
5  }
6
7  int main()
8  {
9    Array a { create(1, 2, 3) };
10   Array b { modify(a) };
11 }
```

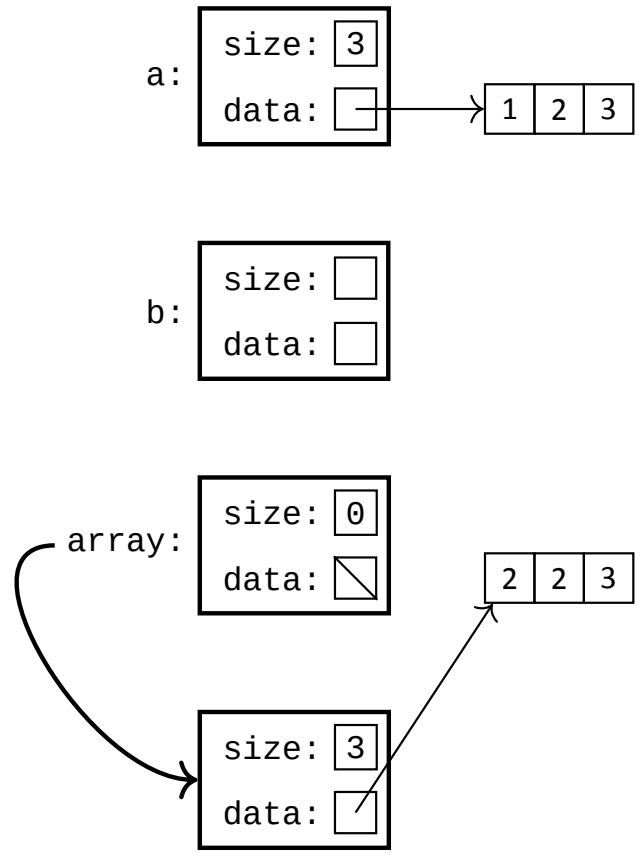


Lifetime & Special Member Functions

Opportunity for optimization

This is the first place where we can optimze: Instead of copying the array, we simply *move* it to the temporary.

```
1  Array modify(Array array)
2  {
3    ++array[0];
4    return array;
5  }
6
7  int main()
8  {
9    Array a { create(1, 2, 3) };
10   Array b { modify(a) };
11 }
```

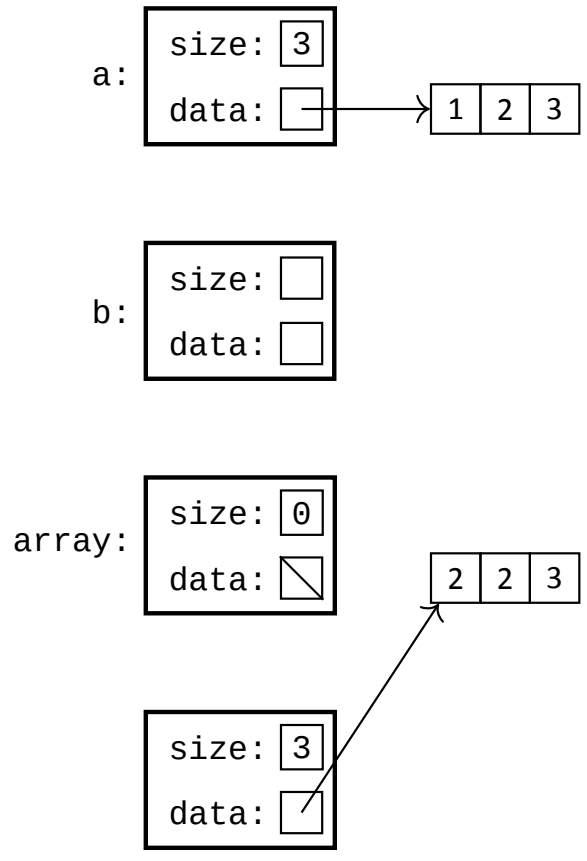


Lifetime & Special Member Functions

Opportunity for optimization

Destroy `array`

```
1  Array modify(Array array)
2  {
3    ++array[0];
4    return array;
5  }
6
7  int main()
8  {
9    Array a { create(1, 2, 3) };
10   Array b { modify(a) };
11 }
```

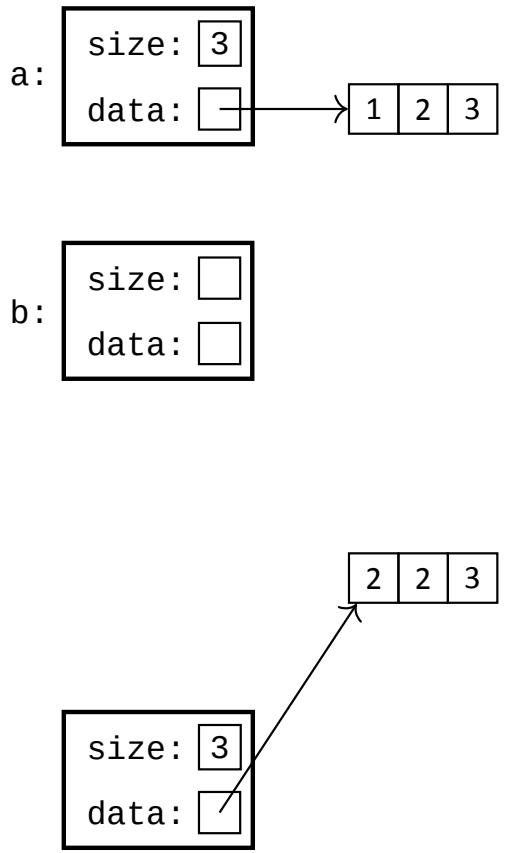


Lifetime & Special Member Functions

Opportunity for optimization

Destroy `array`

```
1  Array modify(Array array)
2  {
3    ++array[0];
4    return array;
5  }
6
7  int main()
8  {
9    Array a { create(1, 2, 3) };
10   Array b { modify(a) };
11 }
```

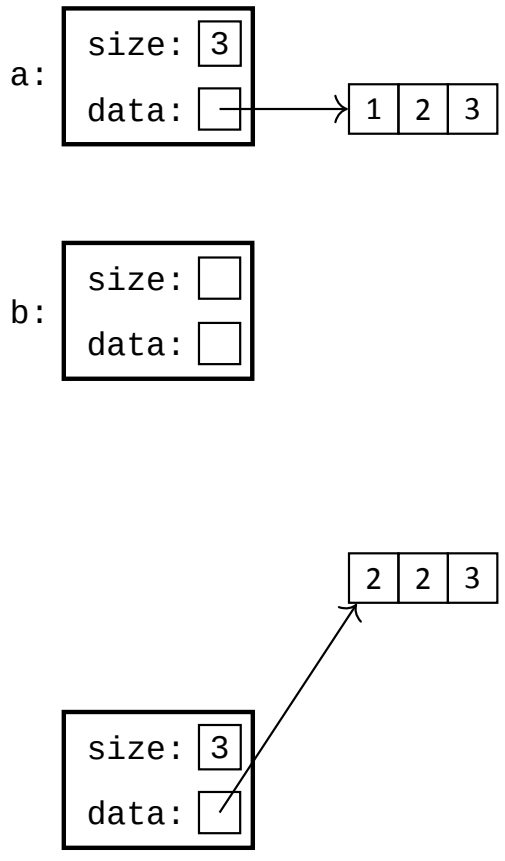


Lifetime & Special Member Functions

Opportunity for optimization

Finish construction of b by **moving** the temporary array

```
1  Array modify(Array array)
2  {
3    ++array[0];
4    return array;
5  }
6
7  int main()
8  {
9    Array a { create(1, 2, 3) };
10   Array b { modify(a) };
11 }
```

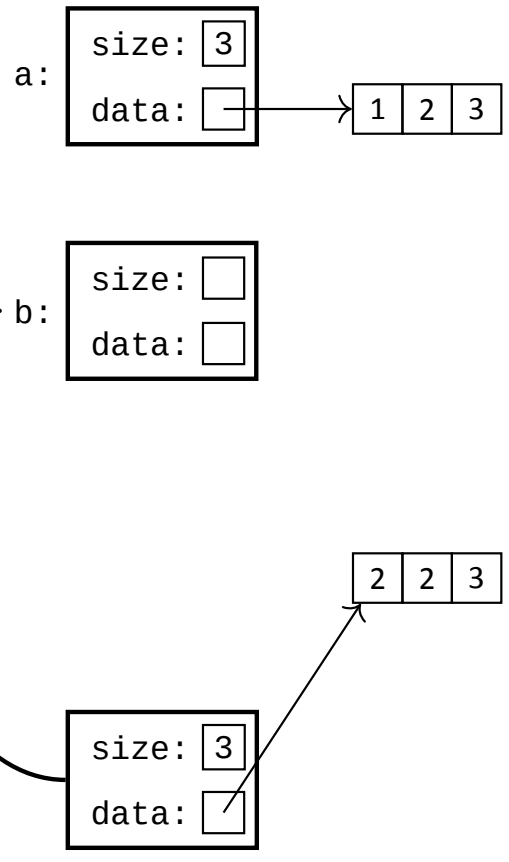


Lifetime & Special Member Functions

Opportunity for optimization

Finish construction of b by **moving** the temporary array

```
1  Array modify(Array array)
2  {
3    ++array[0];
4    return array;
5  }
6
7  int main()
8  {
9    Array a { create(1, 2, 3) };
10   Array b { modify(a) };
11 }
```

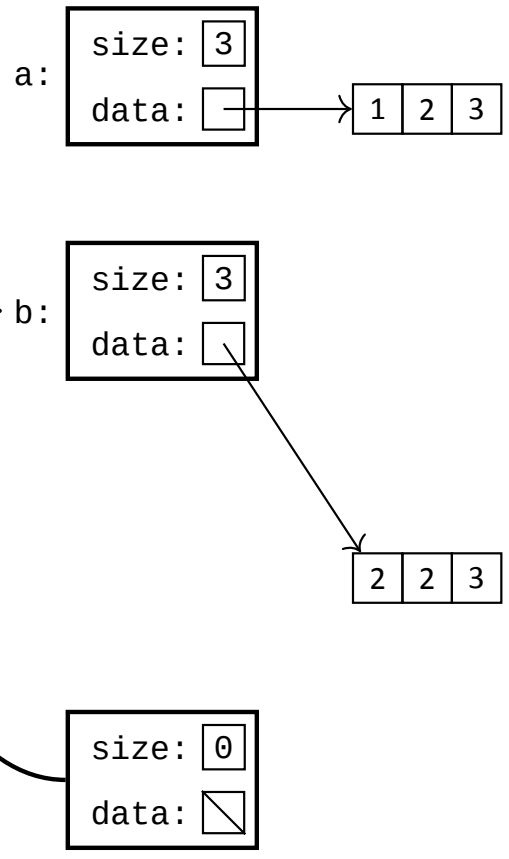


Lifetime & Special Member Functions

Opportunity for optimization

Finish construction of b by **moving** the temporary array

```
1  Array modify(Array array)
2  {
3    ++array[0];
4    return array;
5  }
6
7  int main()
8  {
9    Array a { create(1, 2, 3) };
10   Array b { modify(a) };
11 }
```

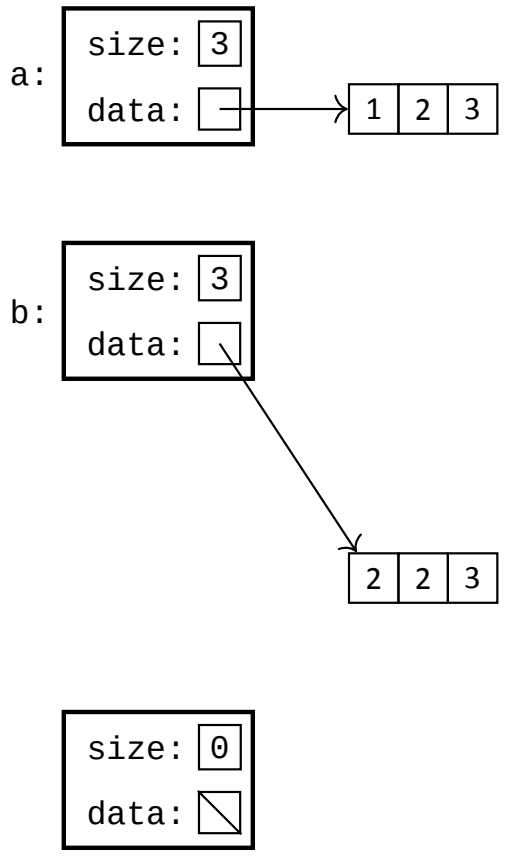


Lifetime & Special Member Functions

Opportunity for optimization

Destroy the temporary object

```
1  Array modify(Array array)
2  {
3    ++array[0];
4    return array;
5  }
6
7  int main()
8  {
9    Array a { create(1, 2, 3) };
10   Array b { modify(a) };
11 }
```

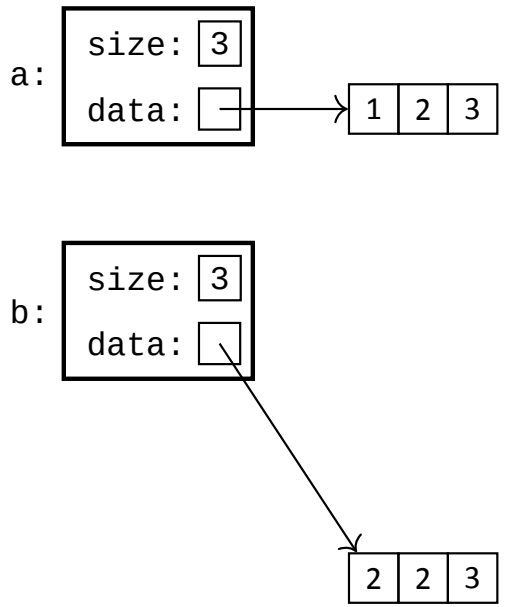


Lifetime & Special Member Functions

Opportunity for optimization

Destroy the temporary object

```
1  Array modify(Array array)
2  {
3    ++array[0];
4    return array;
5  }
6
7  int main()
8  {
9    Array a { create(1, 2, 3) };
10   Array b { modify(a) };
11 }
```

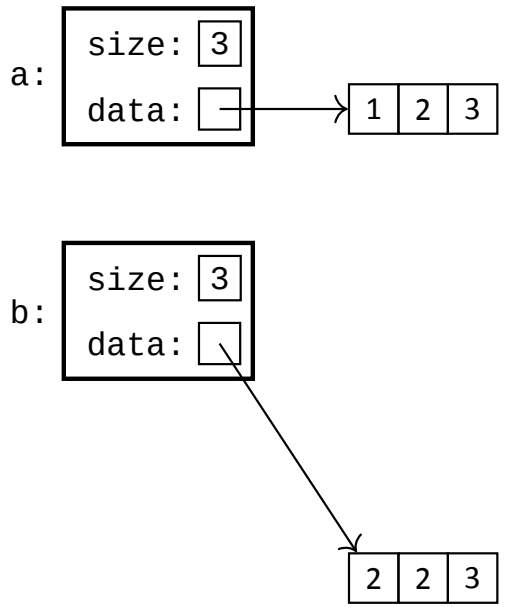


Lifetime & Special Member Functions

Opportunity for optimization

We are done!

```
1  Array modify(Array array)
2  {
3    ++array[0];
4    return array;
5  }
6
7  int main()
8  {
9    Array a { create(1, 2, 3) };
10   Array b { modify(a) };
11 }
```

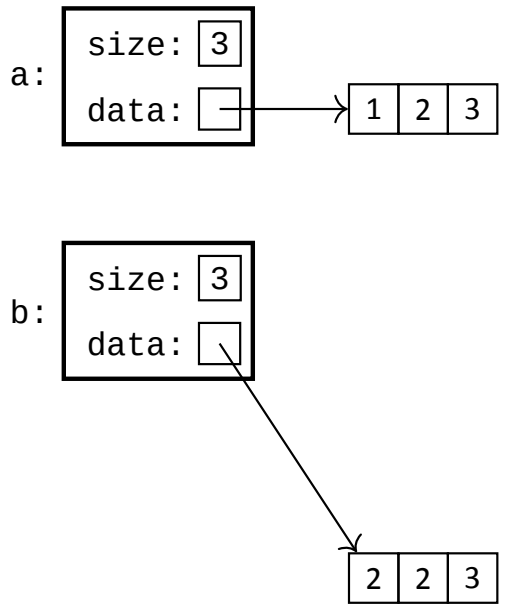


Lifetime & Special Member Functions

Opportunity for optimization

And we only made *two* dynamic allocations!

```
1  Array modify(Array array)
2  {
3    ++array[0];
4    return array;
5  }
6
7  int main()
8  {
9    Array a { create(1, 2, 3) };
10   Array b { modify(a) };
11 }
```



Lifetime & Special Member Functions

Move semantics

- This optimization is called *move semantics* and was introduced in C++11
- The beauty of it is that the *compiler* performs these optimizations *automatically*
- The compiler will opt to move whenever possible.
- Move will occur whenever we are trying to copy an *rvalue*
- There is also an opportunity for the compiler to move when *returning* a local variable (however, there are generally better optimizations the compiler can do in that case).
- Why didn't this work before C++11?
- Because C++11 introduced *rvalue references*

Lifetime & Special Member Functions

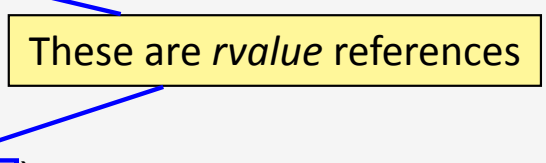
Let's implement *move* semantics

```
1  class Array
2  {
3  public:
4    // ...
5    Array(Array&& other)
6      : size { other.size }, data { other.data }
7    {
8      other.size = 0;
9      array.data = nullptr;
10   }
11
12   Array& operator=(Array&& other)
13   {
14     std::swap(size, other.size);
15     std::swap(data, other.data);
16     return *this;
17   }
18   // ...
19 };
```

Lifetime & Special Member Functions

Let's implement *move* semantics

```
1  class Array
2  {
3  public:
4    // ...
5    Array(Array&& other)
6      : size { other.size }, data { other.data }
7    {
8      other.size = 0;
9      array.data = nullptr;
10   }
11
12   Array& operator=(Array&& other)
13   {
14     std::swap(size, other.size);
15     std::swap(data, other.data);
16     return *this;
17   }
18   // ...
19 };
```



Lifetime & Special Member Functions

Move semantics

- To implement move semantics we just add a constructor and an assignment operator that takes *rvalues* (rather than copy constructors and assignment operators which takes constant *lvalues*).
- These will then be called whenever the value we are passing to the object is an *rvalue*.
- **Examples:**
  - `Array array { create(1, 2, 3) };`
  - `array = create(4, 5, 6);`
  - `array = std::move(other);`

Lifetime & Special Member Functions

Special member functions

```
1  class Array
2  {
3  public:
4    // ...
5    Array(Array const& other);            // copy constructor
6    Array(Array&& other);                 // move constructor
7    ~Array();                             // destructor
8    Array& operator=(Array const& other); // copy assignment operator
9    Array& operator=(Array&& other);      // move assignment operator
10   // ...
11 };
```

Lifetime & Special Member Functions

Rule of N

- rule of three
- rule of five
- rule of zero

Lifetime & Special Member Functions

Rule of N

- rule of three
  - Before C++11 (Note this concept is not valid in C++11 or later);
  - If a class require a destructor or copy operation;
  - it should (probably) implement the destructor, copy constructor and copy assignment.
- rule of five
- rule of zero

Lifetime & Special Member Functions

Rule of N

- rule of three
- rule of five
  - C++11 and onwards;
  - If a class requires a destructor, copy or move operations;
  - it should implement a destructor, copy operations and move operations.
- rule of zero

Lifetime & Special Member Functions

Rule of N

- rule of three
- rule of five
- rule of zero
  - If all resources used in the class take care of their own data;
  - the class should *not* have to implement any destructor, copy or move operations.

1 Lifetime & Special Member Functions
2 Value categories
3 Inheritance
4 Polymorphism

Value categories

rvalue references

```
1  int  x { 3 };
2  // lvalue references to x
3  int& lref { x };
4
5  // rvalue reference to x + 1
6  int&& rref { x + 1 };
7
8  ++x;
9  cout << lref << endl; // prints 4
10 cout << rref << endl; // also prints 4??
```

Value categories

rvalue references

- An lvalue reference (&) can **only** refer to lvalues
- An rvalue reference (&&) can **only** refer to rvalues
- However: rvalues are implicitly convertable to a constant lvalue
- This means that constant lvalue references (`const`&) can refer to both lvalues and rvalues.

Value categories

rvalue references

- If an rvalue reference is bound to a *temporary* object which has a memory location, then it behaves as expected (i.e. it refers to that specific object).
- This can for example occur if we do this:
  `Array&& aref { create(1, 2, 3) };`
- since `Array` dynamically allocates memory in its constructor it *must* have an address.
- These types of rvalues occurs everywhere in code. The easiest example to think of is when an array gets returned from a function
- In those cases we have no way to refer to the *actual* temporary object. We must either copy the object into a variable (i.e. `array = create(1, 2, 3)`) *or* bind it to an rvalue reference, in order to refer to it.

Value categories

rvalue references

- **But** we can also bind references to values that truly have no identity, for example: `int&& ref { 5 };`
- Here the value `5` doesn't have a memory location *at all*, so how can we make a reference to it?
- Well, the compiler will *create* a temporary variable with the value `5` that it then refers to.
- So `ref` does not have the same identity as the expression `5`.
- This is why `ref` prints `4` even though `x` changed: the compiler created a new variable which was initialized with the value of `x+1` (which was `4` when we created the reference). It does **not** refer to the actual expression `x+1`.

Value categories

rvalue references

- On their own rvalue references aren't particularly useful
- As previously mentioned: they were introduced in C++11
- In particular: rvalue references was introduced for the sole purpose of enabling *move semantics*
- The primary property of rvalue references that we care about is the fact that it can *only* bind to rvalues.
- This allows us to make different function overloads based on whether the passed in value is an lvalue or rvalue, which is the true power of rvalue references.

Value categories

rvalues and function overloads

```
1  class Array
2  {
3    // ...
4    void test(Array const&); // #1
5    void test(Array&&);      // #2
6    // ...
7  };
8
9  int main()
10 {
11   Array a1 { create(1, 2, 3) };
12   Array a2 { create(4, 5, 6) };
13
14   a1.test(a2);              // argument is an lvalue so call #1
15   a1.test(create(1, 2, 3)); // argument is an rvalue so call #2
16 }
```

Value categories

rvalues and function overloads

- lvalue references and rvalue references can be used in parameters to make distinct function overloads based on the value category of the argument.
- We have seen examples of this with the copy and move constructors of `Array`, as well as the copy and move assignment operators.
- But this can be done for *any* function.
- However, most of the time it is enough to just have an overload with `const&`
- We only use rvalue references when we can utilize move semantics to avoid copying resources.
- **NOTE:** moving from a value is destructive (i.e. we actually steal the content) so the compiler can only safely do move operations when it is dealing with rvalues, since lvalues can still be referenced afterwards.

Value categories

Consequences of rvalue references

```
1  void fun(Array&& array)
2  {
3    // can we move here?
4
5    Array other { array };
6
7    // ...
8  }
9
10 int main()
11 {
12   fun(create(1, 2, 3));
13 }
```

Value categories

Consequences of rvalue references

```
1  void fun(Array&& array)
2  {
3    // NO! We might reference array later
4    // which means we cannot steal its content
5    Array other { array };
6    // For example:
7    cout << array[0] << endl;
8  }
9
10 int main()
11 {
12   fun(create(1, 2, 3));
13 }
```

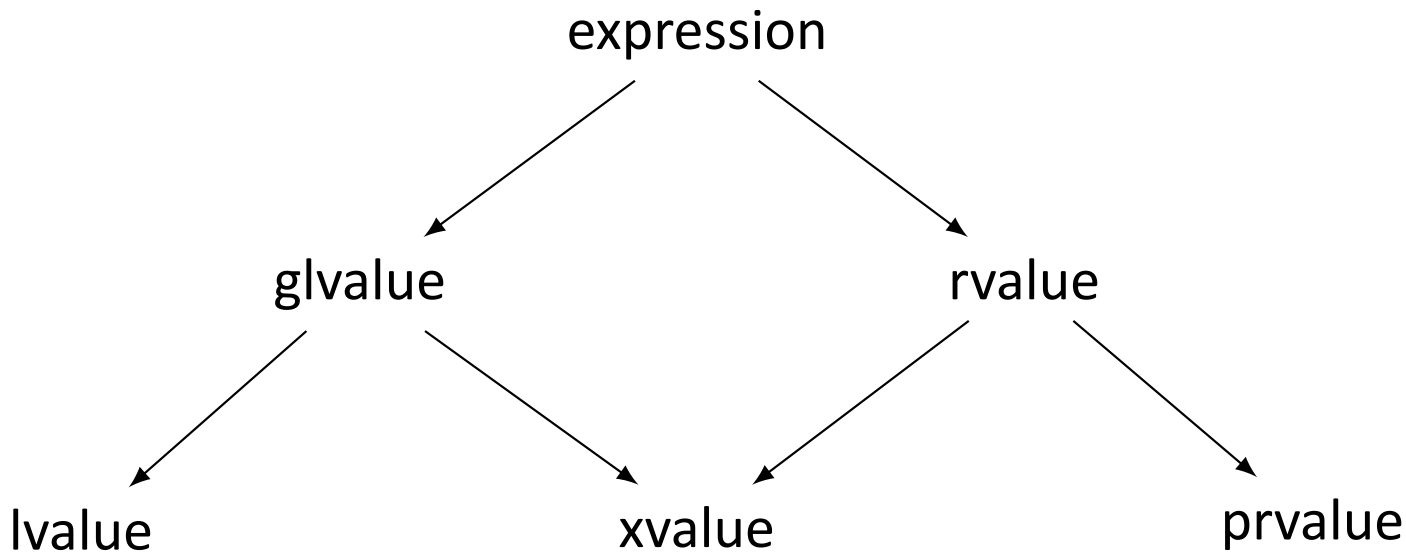
Value categories

Consequences of rvalue references

- So, `create(1, 2, 3)` is an rvalue, but as soon as we bind it to an rvalue reference it becomes an *lvalue*?!
- Yes, but also no...
- Since we give our rvalue identity by binding it to an rvalue reference, it fulfills our old definition of lvalues.
- **But** it is still *semantically* an rvalue.
- This means our current understanding isn't enough...

Value categories

Value categories: the complete picture



Value categories

Value categories: the complete picture

- We introduce *xvalues* (e**x**piring **values**) which are values that *temporarily* have identity (i.e. if they are bound to rvalue references)
- What we previously called *lvalues* are renamed to *glvalues* (**g**eneralized **lvalues**): expressions which refers to objects with identity
- What we previously called *rvalues* are renamed to *prvalues* (**p**ure **rvalues**): expressions *without* identity
- *lvalues* now refer to all glvalues that are **not** xvalues
- *rvalues* now refer to both xvalues and prvalues

1 Lifetime & Special Member Functions
2 Value categories
3 Inheritance
4 Polymorphism

Inheritance

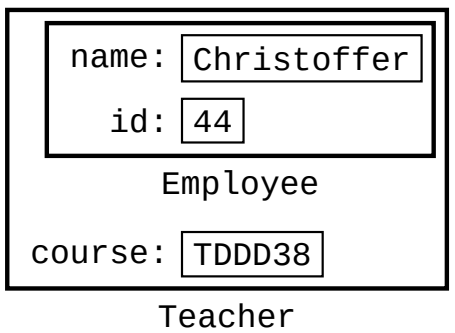
Mental Model

```
1  class Employee
2  {
3    string name{"Christoffer"};
4    int id{44};
5  };
6  class Teacher : public Employee
7  {
8    string course{"TDDD38"};
9  };
10 Teacher c{};
```

Inheritance

Mental Model

```
1  class Employee
2  {
3    string name{"Christoffer"};
4    int id{44};
5  };
6  class Teacher : public Employee
7  {
8    string course{"TDDD38"};
9  };
10 Teacher c{};
```



Inheritance

Protected members

```
1  class Base
2  {
3  public:
4    Base(int x)
5      : x{x} { }
6
7  private:
8    int x;
9  };
```

```
1  struct Derived : Base
2  {
3    Derived(int x)
4      : Base{x} { }
5    int get()
6    {
7      return x; // Error!
8    }
9  };
```

Inheritance

Protected members

```
1  class Base
2  {
3  public:
4    Base(int x)
5      : x{x} { }
6
7  protected:
8    int x;
9  };
```

```
1  struct Derived : Base
2  {
3    Derived(int x)
4      : Base{x} { }
5    int get()
6    {
7      return x; // OK!
8    }
9  };
```

Inheritance

Protected members

`protected` members are:

- inaccessible outside the class;
- accessible within derived classes;
- accessible by friends of the class.

Inheritance

Constructors

```
1  class Base
2  {
3  public:
4    Base(int x);
5  private:
6    int x;
7  };
8
9  Base::Base(int x)
10   : x{x}
11 {
12 }
```

```
1  class Derived : public Base
2  {
3  public:
4    Derived(int x, double y);
5  private:
6    double y;
7  };
8
9  Derived::Derived(int x, double y)
10   : Base{x}, y{y}
11 {
12 }
```

Inheritance

Initialization & Destruction

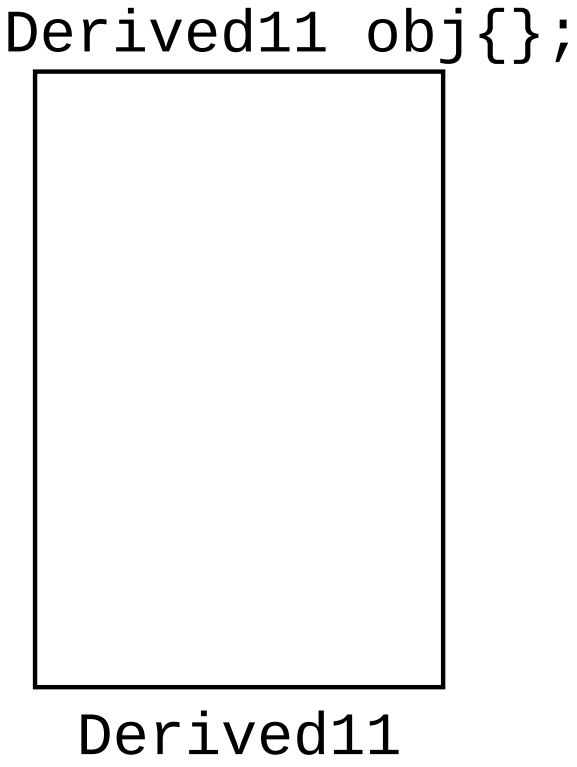
```
1  class Base
2  {
3    int x{1};
4  };
5  class Derived1 : public Base
6  {
7    double y{2.34};
8  };
9  class Derived11 final
10   : public Derived1
11 {
12   int z{56};
13 };
```

```
Derived11 obj{};
```

Inheritance

Initialization & Destruction

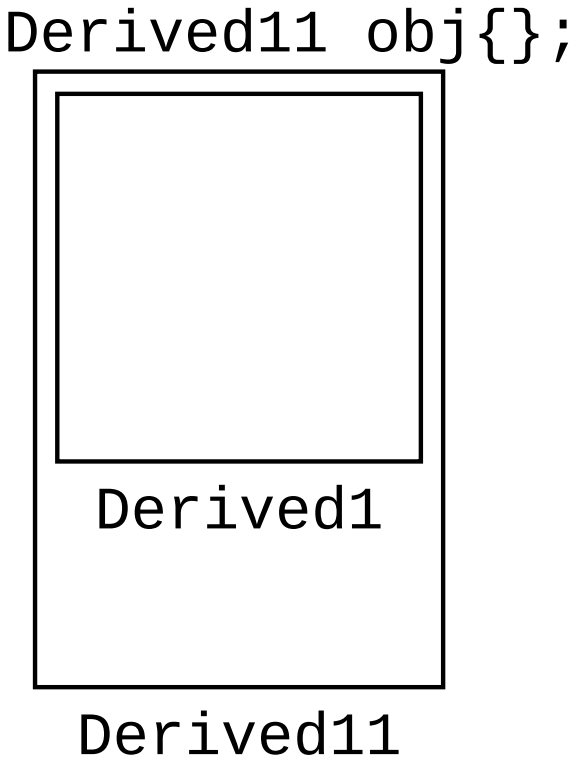
```
1  class Base
2  {
3    int x{1};
4  };
5  class Derived1 : public Base
6  {
7    double y{2.34};
8  };
9  class Derived11 final
10   : public Derived1
11 {
12   int z{56};
13 };
```



Inheritance

Initialization & Destruction

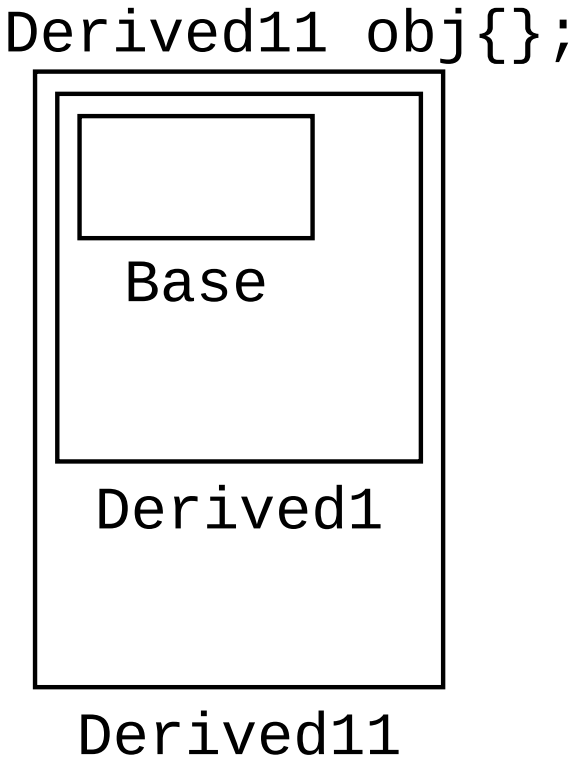
```
1  class Base
2  {
3    int x{1};
4  };
5  class Derived1 : public Base
6  {
7    double y{2.34};
8  };
9  class Derived11 final
10   : public Derived1
11 {
12   int z{56};
13 };
```



Inheritance

Initialization & Destruction

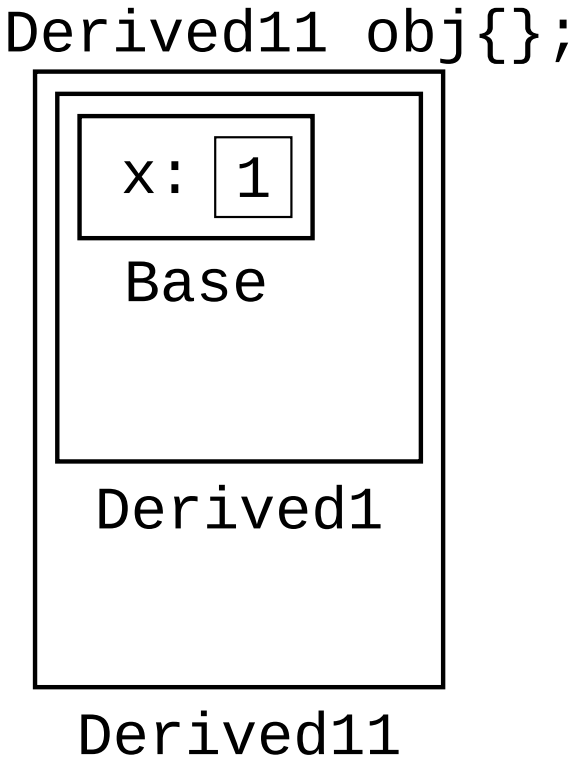
```
1  class Base
2  {
3    int x{1};
4  };
5  class Derived1 : public Base
6  {
7    double y{2.34};
8  };
9  class Derived11 final
10   : public Derived1
11 {
12   int z{56};
13 };
```



Inheritance

Initialization & Destruction

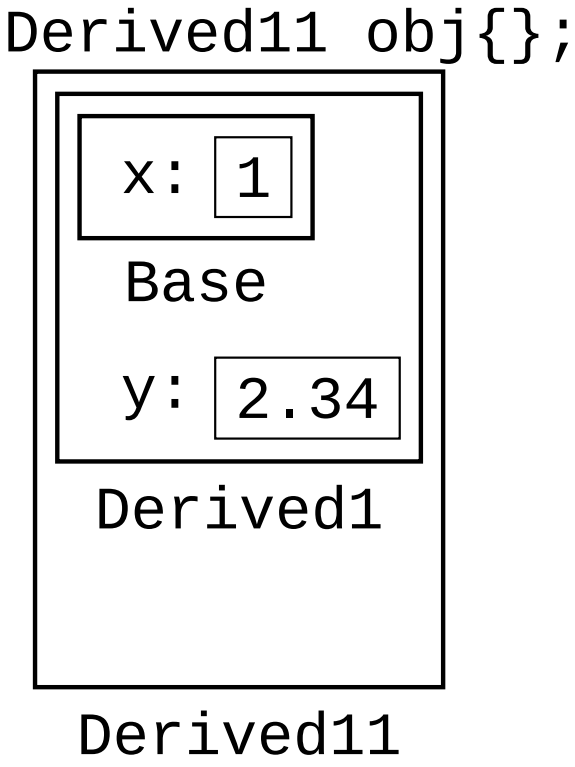
```
1  class Base
2  {
3    int x{1};
4  };
5  class Derived1 : public Base
6  {
7    double y{2.34};
8  };
9  class Derived11 final
10   : public Derived1
11 {
12   int z{56};
13 };
```



Inheritance

Initialization & Destruction

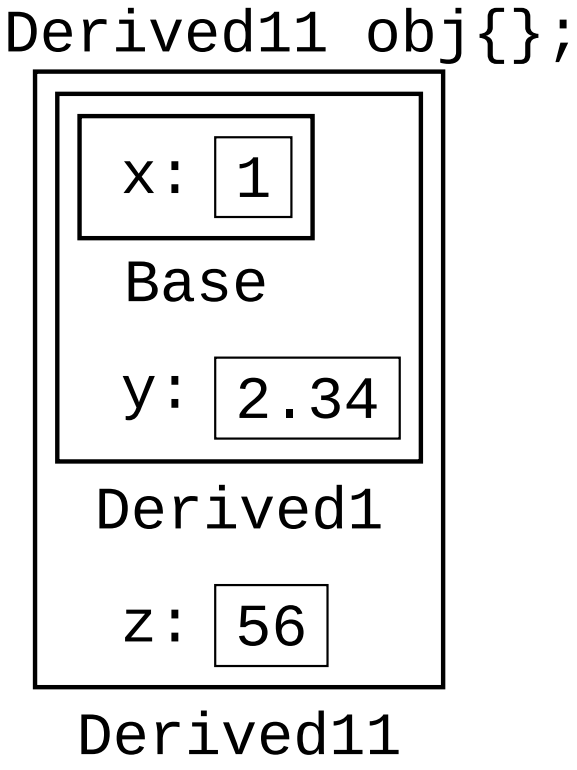
```
1  class Base
2  {
3    int x{1};
4  };
5  class Derived1 : public Base
6  {
7    double y{2.34};
8  };
9  class Derived11 final
10   : public Derived1
11 {
12   int z{56};
13 };
```



Inheritance

Initialization & Destruction

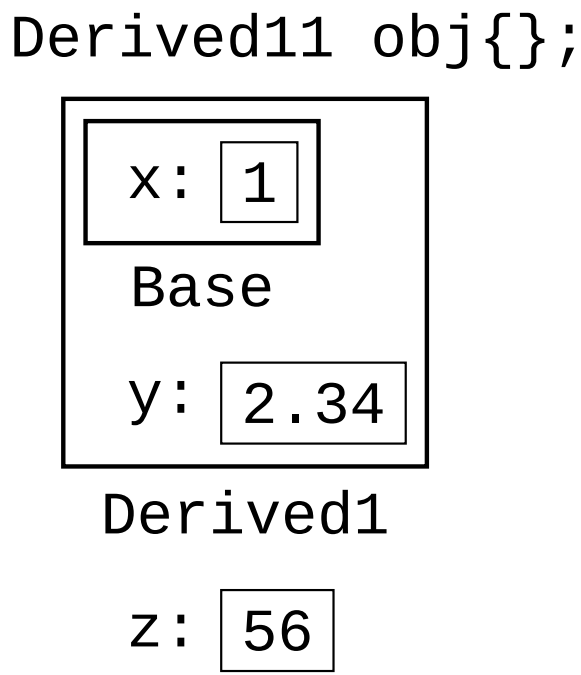
```
1  class Base
2  {
3    int x{1};
4  };
5  class Derived1 : public Base
6  {
7    double y{2.34};
8  };
9  class Derived11 final
10   : public Derived1
11 {
12   int z{56};
13 };
```



Inheritance

Initialization & Destruction

```
1  class Base
2  {
3    int x{1};
4  };
5  class Derived1 : public Base
6  {
7    double y{2.34};
8  };
9  class Derived11 final
10   : public Derived1
11 {
12   int z{56};
13 };
```

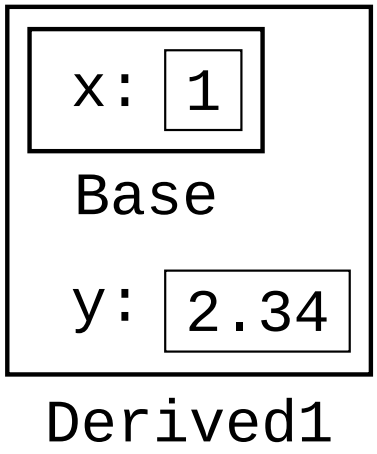


Inheritance

Initialization & Destruction

```
1  class Base
2  {
3    int x{1};
4  };
5  class Derived1 : public Base
6  {
7    double y{2.34};
8  };
9  class Derived11 final
10   : public Derived1
11 {
12   int z{56};
13 };
```

```
Derived11 obj{};
```

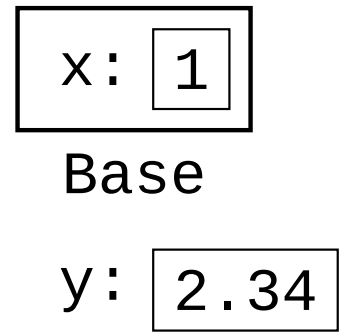


Inheritance

Initialization & Destruction

```
1  class Base
2  {
3    int x{1};
4  };
5  class Derived1 : public Base
6  {
7    double y{2.34};
8  };
9  class Derived11 final
10   : public Derived1
11 {
12   int z{56};
13 };
```

```
Derived11 obj{};
```

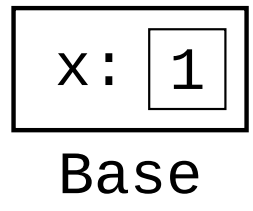


Inheritance

Initialization & Destruction

```
1  class Base
2  {
3    int x{1};
4  };
5  class Derived1 : public Base
6  {
7    double y{2.34};
8  };
9  class Derived11 final
10   : public Derived1
11 {
12   int z{56};
13 };
```

```
Derived11 obj{};
```



Inheritance

Initialization & Destruction

```
1  class Base
2  {
3    int x{1};
4  };
5  class Derived1 : public Base
6  {
7    double y{2.34};
8  };
9  class Derived11 final
10   : public Derived1
11 {
12   int z{56};
13 };
```

```
Derived11 obj{};
```

x: 1

Inheritance

Initialization & Destruction

```
1  class Base
2  {
3    int x{1};
4  };
5  class Derived1 : public Base
6  {
7    double y{2.34};
8  };
9  class Derived11 final
10   : public Derived1
11 {
12   int z{56};
13 };
```

```
Derived11 obj{};
```

Inheritance

Initialization & Destruction

An object is initialized in the following order:

1. initialize base classes (call constructors);
2. initialize all data members in declaration order.

An object is destroyed in the following order:

1. destroy all data members in reverse order;
2. destroy base classes in reverse order.

Inheritance

Types of Inheritance

- `public` inheritance
- `protected` inheritance
- `private` inheritance

Inheritance

Types of Inheritance

- `public` inheritance
  - `class Derived : public Base`
  - All public and protected members of `Base` are available as public and protected respectively in `Derived`.
- `protected` inheritance
- `private` inheritance

Inheritance

Types of Inheritance

- `public` inheritance
- `protected` inheritance
  - `class Derived : protected Base`
  - All public and protected members of `Base` are available as protected in `Derived`.
- `private` inheritance

Inheritance

Types of Inheritance

- `public` inheritance
- `protected` inheritance
- `private` inheritance
  - `class Derived : private Base`
  - All members of `Base` are inherited as private and therefore inaccessible from `Derived`.

Inheritance

Types of Inheritance

- `public` inheritance
- `protected` inheritance
- `private` inheritance

`struct` has public access and inheritance *by-default* while `class` has private by-default. This is their **only** difference.

Inheritance

What will happen? Why?

```
 1 struct Base
 2 {
 3   ~Base() { cout << "Base" << endl; }
 4 };
 5 struct Derived : public Base
 6 {
 7   ~Derived() { cout << "Derived" << endl; }
 8 };
 9 int main()
10 {
11   Derived d{};
12 }
```

1 Lifetime & Special Member Functions
2 Value categories
3 Inheritance
4 Polymorphism

Polymorphism

Dynamic dispatch

```
1  void print1()
2  {  cout << "1" << endl; }
3
4  struct Base
5  {
6    Base() = default;
7    void print()
8    {
9      foo();
10   }
11
12 protected:
13   using function_t = void (*)();
14
15   Base(function_t foo)
16     : foo{foo} { }
17
18 private:
19   function_t foo{print1};
20 };
```

```
1  void print2()
2  { cout << "2" << endl; }
3
4  struct Derived : public Base
5  {
6    // define default constructor
7    Derived()
8       // call the base constructor
9     : Base{print2} { }
10 };
11
12 int main()
13 {
14   Base* bp {new Base{}};
15   bp->print();
16   delete bp;
17
18   bp = new Derived{};
19   bp->print();
20 }
```

Polymorphism

Dynamic dispatch

- In the example above we emulate polymorphism using a function pointer. This is sometimes called *dynamic dispatch* since we are delegating the work of `print()` *during runtime* (since what `bp` is pointing to can only be evaluated once the program runs).
- This is not *exactly* what the compiler does, but it is pretty similar.
- Specifically note that `Base` has a private constructor that sets the function pointer based on a parameter.
- We use this constructor to properly initialize said pointer for the derived classes.

Polymorphism

Dynamic dispatch

- Since the constructor is protected, we can only call this constructor from the derived classes.
- The function pointer is *private* within `Base` which means *only* the base class has access to it.
- In the `Derived` constructor we then initialize the base object to point to `print2` rather than `print1` using the protected constructor.

Polymorphism

Dynamic dispatch

- Now the behaviour of `Base::print` will vary depending on what type it is *actually* pointing to.
- Note that since all classes that inherit from `Base` *contains* a `Base` object, we can always have a pointer to the base object.
- But the compiler then thinks it is actually pointing to a `Base` object rather than a `Derived1` (for example).
- So the function pointer is the only thing that controls the behaviour of `print()`.

Polymorphism

Easier dynamic dispatch

```
1  struct Base
2  {
3    virtual void print()
4    {
5      cout << "1" << endl;
6    }
7  };
8
9  struct Derived : public Base
10 {
11   void print() override
12   {
13     cout << "2" << endl;
14   }
15 };
```

```
1  int main()
2  {
3    Base* bp {new Base{}};
4    bp->print();
5    delete bp;
6
7    bp = new Derived{};
8    bp->print();
9  }
```

Polymorphism

Easier dynamic dispatch

- Luckily for us, we can instead define `virtual` functions that makes dynamic dispatching much easier.
- We just mark which functions in the base class should have varied behaviour based on the current *dynamic* type (i.e. what type `bp` *actually* points to).
- We then specify the behaviour of `print()` in `Derived` by *overriding* the function (this is equivalent to us setting `foo` in the previous example to `print2`).
- A class containing virtual functions is called a *polymorphic class*.

Polymorphism

What will happen? Why?

```
 1 struct Base
 2 {
 3   ~Base() { cout << "Base" << endl; }
 4 };
 5 struct Derived : public Base
 6 {
 7   ~Derived() { cout << "Derived" << endl; }
 8 };
 9 int main()
10 {
11   Base* bp{new Derived()};
12   delete bp;
13 }
```

Polymorphism

What will happen? Why?

```
1  struct Base
2  {
3    virtual ~Base() { cout << "Base" << endl; }
4  };
5  struct Derived : public Base
6  {
7    ~Derived() { cout << "Derived" << endl; }
8  };
9  int main()
10 {
11   Base* bp{new Derived()};
12   delete bp;
13 }
```

Polymorphism

Virtual destructor

- `bp` is of type `Base*` (the *static type* of `bp`);
- deleting `bp` will call the destructor of `Base` regardless of what the *dynamic type* of `bp` is;
- However, if the destructor of base is `virtual` the compiler will use dynamic dispatch to call the overriden destructor from `Derived`, which in turn will call the `Base` destructor.

Therefore we should always declare destructors as virtual for types which will be used through pointers.

Polymorphism

Virtual Table

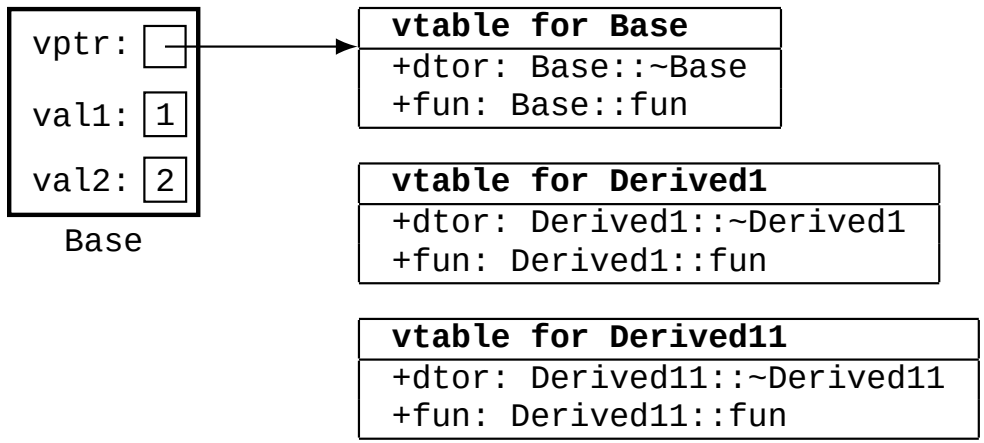
```
1  struct Base
2  {
3    virtual ~Base();
4    virtual void fun();
5    int val1{1};
6    int val2{2};
7  };
8  struct Derived1 : public Base
9  {
10   void fun() override;
11   double d{3.4};
12 };
13 struct Derived11 : public Derived1
14 {
15   void fun() final;
16 };
```

```
1  void Base::fun()
2  {
3    cout << val1 << ' ' << val2;
4  }
5
6  void Derived1::fun()
7  {
8    Base::fun();
9    cout << ' ' << d;
10 }
11
12 void Derived11::fun()
13 {
14   cout << "Derived11 ";
15   Derived1::fun();
16 }
```

Polymorphism

Virtual Table

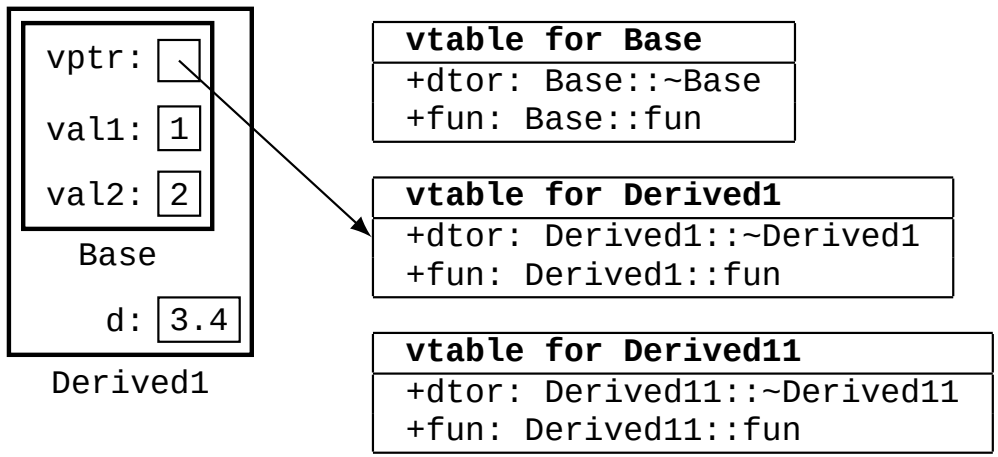
```
Base* bp{new Base{}};
```



Polymorphism

Virtual Table

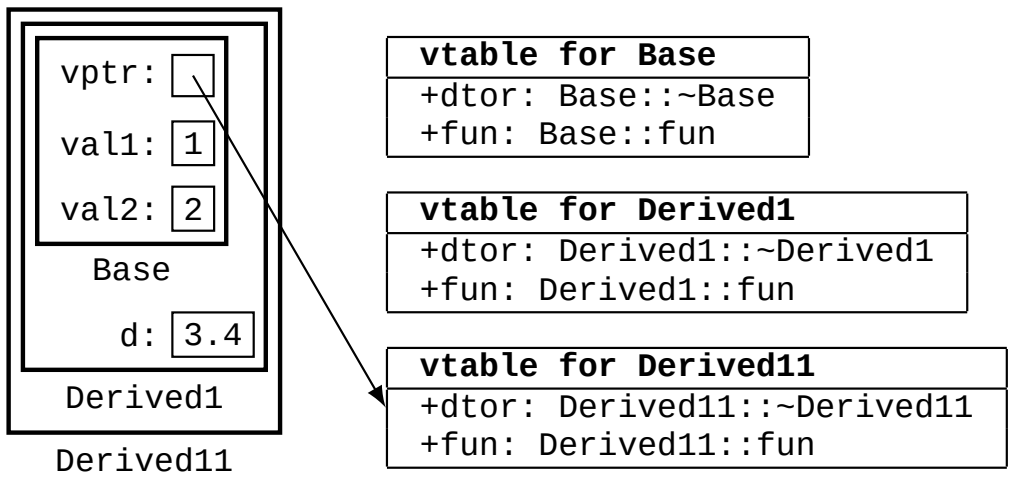
```
Base* bp{new Derived1{}};
```



Polymorphism

Virtual Table

```
Base* bp{new Derived11{}};
```



Polymorphism

Virtual Table

- As previously mentioned: virtual functions aren't exactly equivalent to having a function pointer stored *in the class*.
- Instead (at least for all compilers I know of) it utilizes something called a *virtual table*
- The virtual table is a table of function pointers which point to the *implementation* of each virtual function.
- Each polymorphic class has *one* extra data member called the *virtual pointer* (`vptr`) which is a pointer to the virtual table of the class it *currently* is (see diagram on previous slides).
- Whenever we call a virtual function the program will find corresponding function pointer in the virtual table and call that.
- This means that virtual functions incur additional costs during runtime in comparison with normal member functions...

Polymorphism

Run-time type information (RTTI)

- Each entry in the vtable contains information about the dynamic type;
- This data is accessible with `typeid`.

```
1  struct Base { virtual ~Base() = default; };
2  struct Derived1 : public Base { };
3  struct Derived11 : public Derived1 { };
4  int main()
5  {
6    Base b;
7    Derived1 d1, d2;
8    Derived11 d11;
9    cout << typeid(b).name() << endl;
10   cout << typeid(d1).hash_code() << endl;
11   cout << (typeid(d1) == typeid(b)) << endl;
12   cout << (typeid(d1) == typeid(d2)) << endl;
13   cout << (typeid(d1) == typeid(d11)) << endl;
14 }
```

Polymorphism

Run-time type information (RTTI)

- `typeid` is used to check the *exact* dynamic type;
- We can use `dynamic_cast` to cast pointers or references to objects into some pointer or reference which is compatible with the dynamic type of the object.

```
1  struct Base { virtual ~Base() = default; };
2  struct Derived1 : public Base { };
3  struct Derived11 : public Derived1 { };
4  int main()
5  {
6    Base* bp{new Derived1()};
7    cout << (dynamic_cast<Base*>(bp) == nullptr) << endl;
8    cout << (dynamic_cast<Derived11*>(bp) == nullptr) << endl;
9  }
```

Polymorphism

Run-time type information (RTTI)

```
1  struct Base
2  {
3    virtual ~Base() = default;
4  };
5  struct Derived1 : public Base
6  {
7    int foo() { return 1; }
8  };
9  struct Derived11 : public Derived1 { };
10 int main()
11 {
12   Base* bp{new Derived1()};
13   // won't work, since foo is a non-virtual function in Derived
14   cout << bp->foo() << endl;
15   // will work, since we converted bp to Derived* which has access to foo
16   cout << dynamic_cast<Derived1&>(*bp).foo() << endl;
17   // will throw an exception of type std::bad_cast
18   cout << dynamic_cast<Derived11&>(*bp).foo() << endl;
19 }
```

Polymorphism

What will happen? Why?

```
1  struct Base { virtual ~Base() = default; };
2  struct Derived1 : public Base { };
3  struct Derived11 : public Derived1 { };
4  struct Derived2 : public Base { };
5  int main()
6  {
7    Base* bp{new Derived1()};
8    if (dynamic_cast<Base*>(bp))
9      cout << "B ";
10   if (dynamic_cast<Derived1*>(bp))
11     cout << "D1 ";
12   if (dynamic_cast<Derived11*>(bp))
13     cout << "D11 ";
14   if (dynamic_cast<Derived2*>(bp))
15     cout << "D2 ";
16 }
```

Polymorphism

Slicing

```
1  struct Base
2  {
3    virtual void print() {cout << x;}
4    int x{1};
5  };
6  struct Derived : public Base
7  {
8    void print() override {cout << y;}
9    int y{2};
10 };
```

```
1  void print(Base b)
2  {
3    b.print();
4  }
5
6  int main()
7  {
8    Derived d{};
9    print(d);
10 }
```

- Copying `d` into `b` will cause *slicing*;
- Will only copy the `Base` part of `d` and thus lose all information about `d` being a `Derived`.
- Always use references or pointers!

www.liu.se

LINKÖPING UNIVERSITY