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

Optimizations & Restrictions

Christoffer Holm

Department of Computer and information science

LINKÖPING UNIVERSITY

1 As-if rule
2 Copy elison
3 Lifetime & Special Member Functions
4 Extended value categories

LINKÖPING UNIVERSITY

1 As-if rule
2 Copy elison
3 Lifetime & Special Member Functions
4 Extended value categories

LINKÖPING UNIVERSITY

As-if rule

Fundamental rule

- The *as-if* rule:

As-if rule

Fundamental rule

- The *as-if* rule:
- Any changes to code are OK...

As-if rule

Fundamental rule

- The *as-if* rule:
- Any changes to code are OK...
- ... If the *observable behaviour* stays the same!

As-if rule

Observable Behaviour

- External IO (terminal, files, network, graphics etc.)

As-if rule

Observable Behaviour

- External IO (terminal, files, network, graphics etc.)
- Most system calls

As-if rule

Observable Behaviour

- External IO (terminal, files, network, graphics etc.)
- Most system calls
- Read/write from/to `volatile` objects

As-if rule

Observable Behaviour

- External IO (terminal, files, network, graphics etc.)
- Most system calls
- Read/write from/to `volatile` objects
- Must all happen *as-if* the code was executed as written

As-if rule

Observable Behaviour

- External IO (terminal, files, network, graphics etc.)
- Most system calls
- Read/write from/to `volatile` objects
- Must all happen *as-if* the code was executed as written
- The compiler can therefore remove or change code if it can prove that it does not affect the sequence of observable behaviours

As-if rule

Example #1

Source code:

```
int main()
{
  int x { };
  std::cin >> x;

  int const y { 0 };
  int z { y * x + 3 };

  std::cout << z << std::endl;
}
```

As-if rule

Example #1

Source code:

```
int main()
{
  int x { };
  std::cin >> x;

  int const y { 0 };
  int z { y * x + 3 };

  std::cout << z << std::endl;
}
```

Transformation:

```
int main()
{

  std::cin.ignore();




  std::cout << "3\n";
}
```

As-if rule

Example #1

- The observable behaviour of the program is:
  1. Reading a value from the terminal
  2. Writting the value of `z` to the terminal
- The compiler can quite easily prove that `z` is *always* `3`, regardless of what value was written to `x`
- So all that needs to happen is that the user enters input (however it doesn't actually matter what the user entered, it does not affect any of the observable behaviour) and we then print `3` to the terminal

As-if rule

Example #2

Source code:

```
for (int i { 0 }; i < 3; ++i)
{
  std::cout << "a";
}
std::cout << std::endl;
```

As-if rule

Example #2

Source code:

```
for (int i { 0 }; i < 3; ++i)
{
  std::cout << "a";
}
std::cout << std::endl;
```

Transformation:

```



std::cout << "aaa\n";
```

As-if rule

Example #2

- The observable behaviour of the program is printing three `"a"`
- The compiler can choose to do so in a more efficient way, as in this case where it can simply remove the loop and merge all the strings into one big string.

As-if rule

Assumptions

- Undefined behaviour does not happen

As-if rule

Assumptions

- Undefined behaviour does not happen
  - The compiler can optimize code while assuming that no undefined behaviour happens
  - It is undefined what actually happens when a signed integer overflows so all code produced by the compiler is allowed to ignore whether or not overflow happened and just assume it cannot happen
  - Meanwhile, it is well-defined what happens when unsigned integers overflow, so the compiler must take that into consideration.

As-if rule

Assumptions

- Undefined behaviour does not happen
- Each object has its own identity

As-if rule

Assumptions

- Undefined behaviour does not happen
- Each object has its own identity
  - Two objects of different types cannot share the same identity
  - This means that trying to *reinterpret* objects as different types is generally undefined behaviour, and is therefore assumed to not happen by the compiler

As-if rule

Assumptions

- Undefined behaviour does not happen
- Each object has its own identity
- No external influences

As-if rule

Assumptions

- Undefined behaviour does not happen
- Each object has its own identity
- No external influences
  - Everything that affects the state of the program is well-known by the compiler
  - The compiler assumes that no outside influence can affect the code without the compiler knowing about it

As-if rule

Example #3: Undefined behaviour does not happen

Source code:

```
for (int i { 1 }; i > 0; ++i)
{
  std::cout << "Hello" << std::endl;
}
```

As-if rule

Example #3: Undefined behaviour does not happen

Source code:

```
for (int i { 1 }; i > 0; ++i)
{
  std::cout << "Hello" << std::endl;
}
```

Transformation:

```
while (true)
{
  std::cout << "Hello\n";
}
```

As-if rule

Example #3: Undefined behaviour does not happen

- As previously mentioned: undefined behaviour is assumed to not happen
- Recall: signed integer overflow is undefined behaviour
- $\Rightarrow$ Increasing (++) a positive integer is assumed to always produce a positive integer, even though the hardware clearly doesn't support it
- Therefore the `for`-loop is assumed to be an infinite loop
- So the compiler optimizes away `i` since it is not strictly needed.
- This behaviour is *surprising* but is a consequence of aggressive optimization (which is a *good* thing)
- This behaviour is what allows C and C++ to world-leading languages in terms of performance
- The cost is however that using the language can often be tricky and require deep understanding (as seen in this example)

As-if rule

Example #4: Now with `unsigned`

Source code:

```
for (unsigned i { 1 }; i > 0; ++i)
{
  std::cout << "Hello" << std::endl;
}
```

As-if rule

Example #4: Now with `unsigned`

Source code:

```
for (unsigned i { 1 }; i > 0; ++i)
{
  std::cout << "Hello" << std::endl;
}
```

Transformation:

```
unsigned const max { /* maximum */ };
for (unsigned i { 1 }; i <= max; ++i)
{
  std::cout << "Hello\n";
}
```

As-if rule

Example #4: Now with `unsigned`

- It **is** well-defined behaviour what happens when an `unsigned` overflows
- Therefore the compiler must take overflow into account
- It is *implementation-defined* behaviour **when** overflow occurs, so the program might differ between different platforms
- However, when compiling a program we target a specific platform so that is not an issue for the compiler.
- Since overflow always results in a wrap-around back to `0` the compiler knows the exact number of iterations, so it *could* produce a huge string and just print that, however the compiler probably realizes that this would take too much memory so instead it actually performs the loop.

As-if rule

Consequence

Rule-of-thumb: *always* use `unsigned` as loop-counters

As-if rule

Example #5: Each object has its own identity

Source code:

```
float x { 3.1415f };
int* ptr { reinterpret_cast<int*>(&x) };
std::cin >> x;
std::cout << *ptr << std::endl;
```

As-if rule

Example #5: Each object has its own identity

Source code:

```
float x { 3.1415f };
int* ptr { reinterpret_cast<int*>(&x) };
std::cin >> x;
std::cout << *ptr << std::endl;
```

Transformation:

```


std::cin.ignore();
std::cout << "1078530008\n";
```

As-if rule

Example #5: Each object has its own identity

- The compiler is allowed to assume that objects of different types refer to *different* identities
- Therefore `&x` and `ptr` must refer to different objects, since they refer to an `int` and a `float` respectively
- Therefore the following observations can be done:
    1. We write to `x`, but we never read it
    2. `ptr` refers to a value that is never modified, and only printed once
- So `x` can be removed, and `*ptr` can be replaced with its initial value (`1078530008` on my machine).
- This might also seem like a hostile rule, but this type of optimization allows the compiler to be more aggressive in its optimization (which we already established is a *good* thing)

1 As-if rule
2 Copy elison
3 Lifetime & Special Member Functions
4 Extended value categories

Copy elison

Special exception

- Special exception to the *as-if* rule:

Copy elison

Special exception

- Special exception to the *as-if* rule: *copy elison*

Copy elison

Special exception

- Special exception to the *as-if* rule: *copy elison*
- Creation of objects can be removed...

Copy elison

Special exception

- Special exception to the *as-if* rule: *copy elison*
- Creation of objects can be removed...
- ... even if there is observable behaviour...

Copy elison

Special exception

- Special exception to the *as-if* rule: *copy elison*
- Creation of objects can be removed...
- ... even if there is observable behaviour...
- ... but only during special circumstances!

Copy elison

Initial demonstration

Source code:

```
std::string str { std::string{"Hello"} };
```

Copy elison

Initial demonstration

Source code:

```
std::string str { std::string{"Hello"} };
```

str:

Copy elison

Initial demonstration

Source code:

```
std::string str { std::string{"Hello"} };
```

`str:`

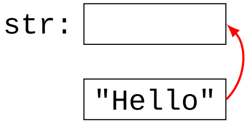
`"Hello"`

Copy elison

Initial demonstration

Source code:

```
std::string str { std::string{"Hello"} };
```

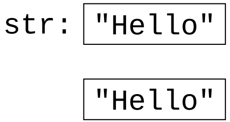


Copy elison

Initial demonstration

Source code:

```
std::string str { std::string{"Hello"} };
```



Copy elison

Initial demonstration

Source code:

```
std::string str { std::string{"Hello"} };
```

str: "Hello"

Copy elison

Initial demonstration

Transformation:

```
std::string str { "Hello" };
```

Copy elison

Initial demonstration

Transformation:

```
std::string str { "Hello" };
```

str: "Hello"

Copy elison

Initial demonstration

- We can initialize objects by creating a temporary object and then copying
- In the latest example, `str` is initialized by constructing a temporary string `std::string{"Hello"}` and then copying that string into `str` followed by destroying the temporary
- This leads to a lot of unnecessary work...
- So to mitigate this the compiler uses copy elison to remove the temporary object creation and just create `str` directly.
- This would occur, even if the constructors of `std::string` had observerable behaviours

Copy elison

Returning without optimization

```
std::string fun()
{
  return std::string{"my string"};
}

int main()
{
  std::string str { fun() };
}
```

Copy elison

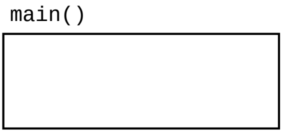
Returning without optimization

```
std::string fun()
{
  return std::string{"my string"};
}

int main()
{
  std::string str { fun() };
}
```

Copy elison

Returning without optimization

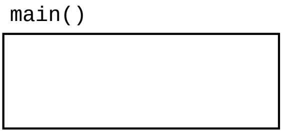
```
std::string fun()
{
  return std::string{"my string"};
}

int main()
{
  std::string str { fun() };
}
```



Copy elison

Returning without optimization

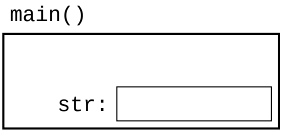
```
std::string fun()
{
  return std::string{"my string"};
}

int main()
{
  std::string str { fun() };
}
```



Copy elison

Returning without optimization

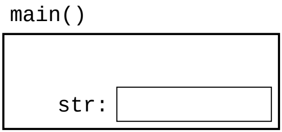
```
std::string fun()
{
  return std::string{"my string"};
}

int main()
{
  std::string str { fun() };
}
```



Copy elison

Returning without optimization

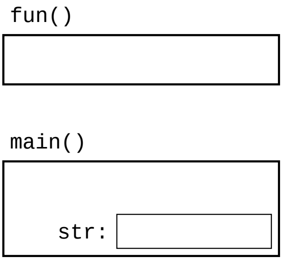
```cpp
std::string fun()
{
  return std::string{"my string"};
}

int main()
{
  std::string str { fun() };
}
```



Copy elison

Returning without optimization

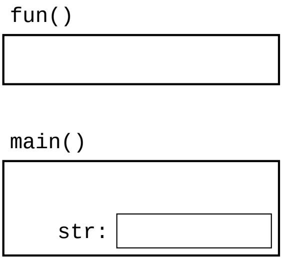
```
std::string fun()
{
  return std::string{"my string"};
}

int main()
{
  std::string str { fun() };
}
```



Copy elison

Returning without optimization

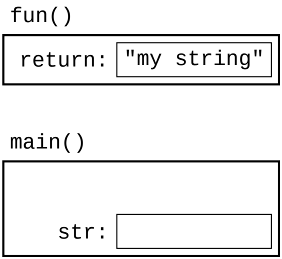
```
std::string fun()
{
  return std::string{"my string"};
}

int main()
{
  std::string str { fun() };
}
```



Copy elison

Returning without optimization

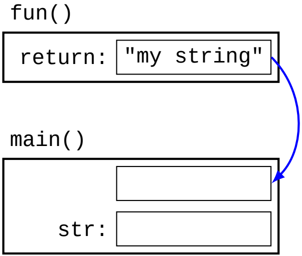
```cpp
std::string fun()
{
  return std::string{"my string"};
}

int main()
{
  std::string str { fun() };
}
```



Copy elison

Returning without optimization

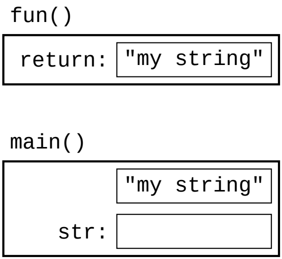
```cpp
std::string fun()
{
  return std::string{"my string"};
}

int main()
{
  std::string str { fun() };
}
```



Copy elison

Returning without optimization

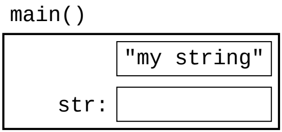
```cpp
std::string fun()
{
  return std::string{"my string"};
}

int main()
{
  std::string str { fun() };
}
```



Copy elison

Returning without optimization

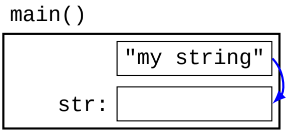
```cpp
std::string fun()
{
  return std::string{"my string"};
}

int main()
{
  std::string str { fun() };
}
```



Copy elison

Returning without optimization

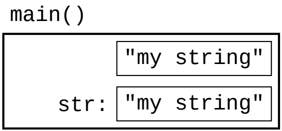
```
std::string fun()
{
  return std::string{"my string"};
}

int main()
{
  std::string str { fun() };
}
```



Copy elison

Returning without optimization

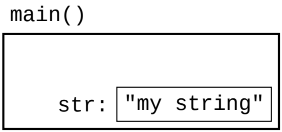
```
std::string fun()
{
  return std::string{"my string"};
}

int main()
{
  std::string str { fun() };
}
```



Copy elison

Returning without optimization

```
std::string fun()
{
  return std::string{"my string"};
}

int main()
{
  std::string str { fun() };
}
```



Copy elison

Returning without optimization

- When a objects gets constructed *during* the return-statement of a function, the object gets constructed within the scope of the function
- This object is then immediately *copied* to the scope of the caller before exiting the function scope (which results in the original object being destroyed)
- The program execution then returns back to the scope of the caller and (usually) copies the returned object into a variable within the scope before destroying the returned object.
- What this means is that code such as `std::string str { fun() };` might result in *three* different string objects being constructed, which is quite wasteful.
- Luckily, there is an optimization that is permitted thanks to copy elison...

Copy elison

Return **V**alue **O**ptimization (RVO)

```
std::string fun()
{
  return std::string{"my string"};
}

int main()
{
  std::string str { fun() };
}
```

Copy elison

Return **V**alue **O**ptimization (RVO)

```
std::string fun()
{
  return std::string{"my string"};
}

int main()
{
  std::string str { fun() };
}
```

Copy elison

Return **V**alue **O**ptimization (RVO)

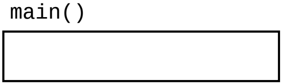
```
std::string fun()
{
  return std::string{"my string"};
}

int main()
{
  std::string str { fun() };
}
```

main()

Copy elison

Return **V**alue **O**ptimization (RVO)

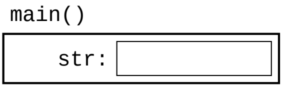
```cpp
std::string fun()
{
  return std::string{"my string"};
}

int main()
{
  std::string str { fun() };
}
```



Copy elison

Return **V**alue **O**ptimization (RVO)

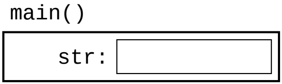
```cpp
std::string fun()
{
  return std::string{"my string"};
}

int main()
{
  std::string str { fun() };
}
```



Copy elison

Return **V**alue **O**ptimization (RVO)

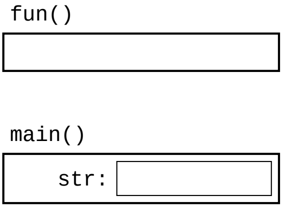
```cpp
std::string fun()
{
  return std::string{"my string"};
}

int main()
{
  std::string str { fun() };
}
```



Copy elison

Return **V**alue **O**ptimization (RVO)

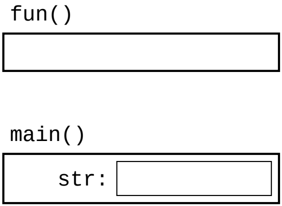
```
std::string fun()
{
  return std::string{"my string"};
}

int main()
{
  std::string str { fun() };
}
```



Copy elison

Return **V**alue **O**ptimization (RVO)

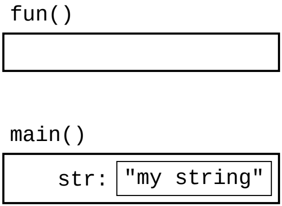
```cpp
std::string fun()
{
  return std::string{"my string"};
}

int main()
{
  std::string str { fun() };
}
```



Copy elison

Return **V**alue **O**ptimization (RVO)

```cpp
std::string fun()
{
  return std::string{"my string"};
}

int main()
{
  std::string str { fun() };
}
```



Copy elison

Return **V**alue **O**ptimization (RVO)

```
std::string fun()
{
  return std::string{"my string"};
}

int main()
{
  std::string str { fun() };
}
```

```
main()
    str: "my string"
```

Copy elison

Return **V**alue **O**ptimization (RVO)

- When *all* return-statements of a function constructs a new object then the return value is allowed to be constructed *directly* in the callers scope
- In particular, if the caller puts the return value into a local variable, then the compiler is allowed to treat that variable as the memory location of the return value.
- This technique is called return value optimization (RVO) and enables the cheapest possible function calls possible.
- There is an alternative variant of this...

Copy elison

Named **R**eturn **V**alue **O**ptimization (NRVO)

```cpp
std::string fun(int x)
{
  std::string value { "positive" };
  if (x >= 0)
    return value;
  value = "negative";
  return value;
}

int main()
{
  std::string str { fun(-2) };
}
```

Copy elison

Named **R**eturn **V**alue **O**ptimization (NRVO)

```
std::string fun(int x)
{
  std::string value { "positive" };
  if (x >= 0)
    return value;
  value = "negative";
  return value;
}

int main()
{
  std::string str { fun(-2) };
}
```

Declared first!

Copy elison

Named **R**eturn **V**alue **O**ptimization (NRVO)

```
std::string fun(int x)
{
  std::string value { "positive" };
  if (x >= 0)
    return value;
  value = "negative";
  return value;
}

int main()
{
  std::string str { fun(-2) };
}
```

Copy elison

Named **R**eturn **V**alue **O**ptimization (NRVO)

```
std::string fun(int x)
{
  std::string value { "positive" };
  if (x >= 0)
    return value;
  value = "negative";
  return value;
}

int main()
{
  std::string str { fun(-2) };
}
```

Same identity returned!

Copy elison

Named **R**eturn **V**alue **O**ptimization (NRVO)

```
std::string fun(int x)
{
  std::string value { "positive" };
  if (x >= 0)
    return value;
  value = "negative";
  return value;
}

int main()
{
  std::string str { fun(-2) };
}
```

Copy elison

Named **R**eturn **V**alue **O**ptimization (NRVO)

```cpp
std::string fun(int x)
{
  std::string value { "positive" };
  if (x >= 0)
    return value;
  value = "negative";
  return value;
}

int main()
{
  std::string str { fun(-2) };
}
```

main()

Copy elison

Named **R**eturn **V**alue **O**ptimization (NRVO)

```
std::string fun(int x)
{
  std::string value { "positive" };
  if (x >= 0)
    return value;
  value = "negative";
  return value;
}

int main()
{
  std::string str { fun(-2) };
}
```

main()

Copy elison

Named **R**eturn **V**alue **O**ptimization (NRVO)

```
std::string fun(int x)
{
  std::string value { "positive" };
  if (x >= 0)
    return value;
  value = "negative";
  return value;
}

int main()
{
  std::string str { fun(-2) };
}
```

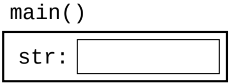
main()

str:

Copy elison

Named **R**eturn **V**alue **O**ptimization (NRVO)

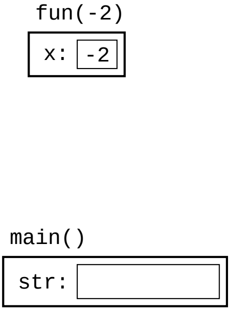
```cpp
std::string fun(int x)
{
  std::string value { "positive" };
  if (x >= 0)
    return value;
  value = "negative";
  return value;
}

int main()
{
  std::string str { fun(-2) };
}
```



Copy elison

Named **R**eturn **V**alue **O**ptimization (NRVO)

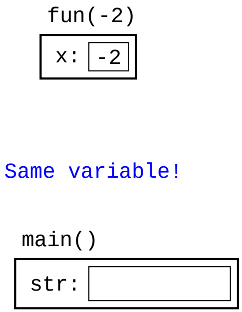
```
std::string fun(int x)
{
  std::string value { "positive" };
  if (x >= 0)
    return value;
  value = "negative";
  return value;
}

int main()
{
  std::string str { fun(-2) };
}
```



Copy elison

Named **R**eturn **V**alue **O**ptimization (NRVO)

```cpp
std::string fun(int x)
{
  std::string value { "positive" };
  if (x >= 0)
    return value;
  value = "negative";
  return value;
}

int main()
{
  std::string str { fun(-2) };
}
```



Copy elison

Named **R**eturn **V**alue **O**ptimization (NRVO)

```
std::string fun(int x)
{
  std::string value { "positive" };
  if (x >= 0)
    return value;
  value = "negative";
  return value;
}

int main()
{
  std::string str { fun(-2) };
}
```

fun(-2)

x: -2

Same variable!

main()

str: "positive"

Copy elison

Named **R**eturn **V**alue **O**ptimization (NRVO)

```cpp
std::string fun(int x)
{
  std::string value { "positive" };
  if (x >= 0)
    return value;
  value = "negative";
  return value;
}

int main()
{
  std::string str { fun(-2) };
}
```

fun(-2)

x: -2

main()

str: "positive"

Copy elison

Named **R**eturn **V**alue **O**ptimization (NRVO)

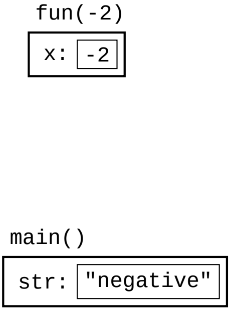
```
std::string fun(int x)
{
  std::string value { "positive" };
  if (x >= 0)
    return value;
  value = "negative";
  return value;
}

int main()
{
  std::string str { fun(-2) };
}
```

```
fun(-2)
x: -2

main()
str: "negative"
```

Copy elison

Named **R**eturn **V**alue **O**ptimization (NRVO)

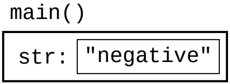
```
std::string fun(int x)
{
  std::string value { "positive" };
  if (x >= 0)
    return value;
  value = "negative";
  return value;
}

int main()
{
  std::string str { fun(-2) };
}
```



Copy elison

Named **R**eturn **V**alue **O**ptimization (NRVO)

```cpp
std::string fun(int x)
{
  std::string value { "positive" };
  if (x >= 0)
    return value;
  value = "negative";
  return value;
}

int main()
{
  std::string str { fun(-2) };
}
```



Copy elison

Named **R**eturn **V**alue **O**ptimization (NRVO)

- If a function returns a non-trivial type *by value* and each return statement in the function refers to the *exact* same (local) identity, then the compiler can perform *named return value optimization* (NRVO)
- I.e. if each return-statement in the function returns the exact same local variable, then the compiler is allowed to use the storage in the caller code as a substitute for that local variable.
- Meaning that any operation performed on that local variable actually directly modifies the target object in the calling scope, so the local object doesn't exists at all.

Copy elison

Observations

- Copy elision increases performance

Copy elison

Observations

- Copy elision increases performance
- RVO and NRVO are great

Copy elison

Observations

- Copy elision increases performance
- RVO and NRVO are great
- **BUT:** RVO and NRVO are very constrained

Copy elison

Observations

- Copy elision increases performance
- RVO and NRVO are great
- **BUT:** RVO and NRVO are very constrained
- Is there a compromise?

Copy elison

Observations

- Copy elision increases performance
- RVO and NRVO are great
- **BUT:** RVO and NRVO are very constrained
- Is there a compromise?
- To answer that, we must further explore object lifetime!

1 As-if rule
2 Copy elison
3 Lifetime & Special Member Functions
4 Extended value categories

LINKÖPING
UNIVERSITY

Lifetime & Special Member Functions

```cpp
template <typename T>
class Array
{
public:
  explicit Array(std::size_t size)              // constructor
    : size { size }, data { new T[size] }
  { }

  T& operator[](std::size_t index)              // non-const access
  { return data[index]; }

  T const& operator[](std::size_t index) const   // const access
  { return data[index]; }

  std::size_t size() const                       // get the size
  { return size; }

private:
  std::size_t size;
  T* data;
};
```

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

```
template <typename T>
class Array
{
public:
  explicit Array(std::size_t size)               // constructor
    : size { size }, data { new T[size] }
  { }

  ~Array()
  {
    delete[] data;
  }

  // ...

private:
  std::size_t size;
  T* data;
};
```

Lifetime & Special Member Functions

RAII

```
template <typename T>
class Array
{
public:
  explicit Array(std::size_t size)             // constructor
    : size { size }, data { new T[size] }
  { }

  ~Array()
  {
    delete[] data;
  }

  // ...

private:
  std::size_t size;
  T* data;
};
```

Destructor: is called during *destruction*

Lifetime & Special Member Functions

RAII

```cpp
template <typename T>
class Array
{
public:
  explicit Array(std::size_t size)             // constructor
    : size { size }, data { new T[size] }
  { }

  ~Array()
  {
    delete[] data;
  }

  // ...

private:
  std::size_t size;
  T* data;
};
```

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

```
Array<int> create(int a, int b, int c)
{
  Array<int> result { 3 };
  result[0] = a;
  result[1] = b;
  result[2] = c;
  return result;
}

int main()
{
  Array<int> array { create(1, 2, 3) };
  // ...
}
```

Lifetime & Special Member Functions

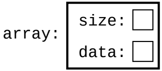
Copies

```
Array<int> create(int a, int b, int c)
{
  Array<int> result { 3 };
  result[0] = a;
  result[1] = b;
  result[2] = c;
  return result;
}

int main()
{
  Array<int> array { create(1, 2, 3) };
  // ...
}
```

Lifetime & Special Member Functions

Copies

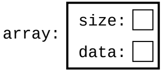
```cpp
Array<int> create(int a, int b, int c)
{
  Array<int> result { 3 };
  result[0] = a;
  result[1] = b;
  result[2] = c;
  return result;
}

int main()
{
  Array<int> array { create(1, 2, 3) };
  // ...
}
```



Lifetime & Special Member Functions

Copies

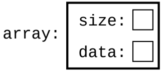
```
Array<int> create(int a, int b, int c)
{
  Array<int> result { 3 };
  result[0] = a;
  result[1] = b;
  result[2] = c;
  return result;
}

int main()
{
  Array<int> array { create(1, 2, 3) };
  // ...
}
```



Lifetime & Special Member Functions

Copies

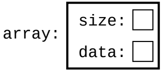
```
Array<int> create(int a, int b, int c)
{
  Array<int> result { 3 };
  result[0] = a;
  result[1] = b;
  result[2] = c;
  return result;
}

int main()
{
  Array<int> array { create(1, 2, 3) };
  // ...
}
```



Lifetime & Special Member Functions

Copies

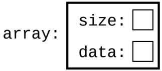
```
Array<int> create(int a, int b, int c)
{
  Array<int> result { 3 };
  result[0] = a;
  result[1] = b;
  result[2] = c;
  return result;
}

int main()
{
  Array<int> array { create(1, 2, 3) };
  // ...
}
```



Lifetime & Special Member Functions

Copies

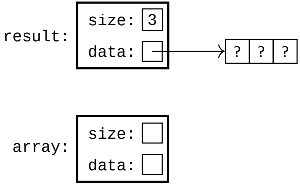
```
Array<int> create(int a, int b, int c)
{
  Array<int> result { 3 };
  result[0] = a;
  result[1] = b;
  result[2] = c;
  return result;
}

int main()
{
  Array<int> array { create(1, 2, 3) };
  // ...
}
```



Lifetime & Special Member Functions

Copies

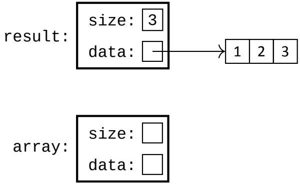
```
Array<int> create(int a, int b, int c)
{
  Array<int> result { 3 };
  result[0] = a;
  result[1] = b;
  result[2] = c;
  return result;
}

int main()
{
  Array<int> array { create(1, 2, 3) };
  // ...
}
```



Lifetime & Special Member Functions

Copies

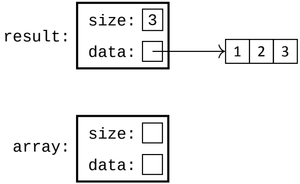
```cpp
Array<int> create(int a, int b, int c)
{
  Array<int> result { 3 };
  result[0] = a;
  result[1] = b;
  result[2] = c;
  return result;
}

int main()
{
  Array<int> array { create(1, 2, 3) };
  // ...
}
```



Lifetime & Special Member Functions

Copies

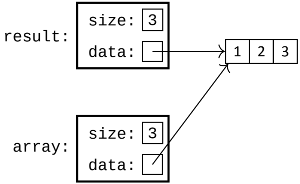
```
Array<int> create(int a, int b, int c)
{
  Array<int> result { 3 };
  result[0] = a;
  result[1] = b;
  result[2] = c;
  return result;
}

int main()
{
  Array<int> array { create(1, 2, 3) };
  // ...
}
```



Lifetime & Special Member Functions

Copies

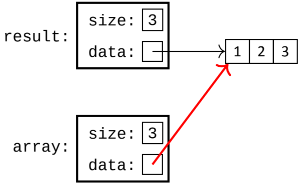
```
Array<int> create(int a, int b, int c)
{
  Array<int> result { 3 };
  result[0] = a;
  result[1] = b;
  result[2] = c;
  return result;
}

int main()
{
  Array<int> array { create(1, 2, 3) };
  // ...
}
```



Lifetime & Special Member Functions

Copies

```
Array<int> create(int a, int b, int c)
{
  Array<int> result { 3 };
  result[0] = a;
  result[1] = b;
  result[2] = c;
  return result;
}

int main()
{
  Array<int> array { create(1, 2, 3) };
  // ...
}
```



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<T> const& other)
```

Lifetime & Special Member Functions

Copy constructor

```
template <typename T>
class Array
{
public:
  // ...

  Array(Array const& other) // copy constructor
    : size { other.size },
      data { new T[size] }
  {
    // copy everything from other into our own allocation
    for (std::size_t i { 0 }; i < size; ++i)
      data[i] = other.data[i];
  }

  // ...
private:
  std::size_t size;
  T* data;
};
```

Lifetime & Special Member Functions

Copies

```
Array<int> create(int a, int b, int c)
{
  Array<int> result { 3 };
  result[0] = a;
  result[1] = b;
  result[2] = c;
  return result;
}

int main()
{
  Array<int> array { create(1, 2, 3) };
  // ...
}
```

Lifetime & Special Member Functions

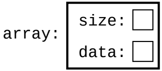
Copies

```
Array<int> create(int a, int b, int c)
{
  Array<int> result { 3 };
  result[0] = a;
  result[1] = b;
  result[2] = c;
  return result;
}

int main()
{
  Array<int> array { create(1, 2, 3) };
  // ...
}
```

Lifetime & Special Member Functions

Copies

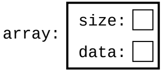
```
Array<int> create(int a, int b, int c)
{
  Array<int> result { 3 };
  result[0] = a;
  result[1] = b;
  result[2] = c;
  return result;
}

int main()
{
  Array<int> array { create(1, 2, 3) };
  // ...
}
```



Lifetime & Special Member Functions

Copies

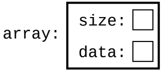
```
Array<int> create(int a, int b, int c)
{
  Array<int> result { 3 };
  result[0] = a;
  result[1] = b;
  result[2] = c;
  return result;
}

int main()
{
  Array<int> array { create(1, 2, 3) };
  // ...
}
```



Lifetime & Special Member Functions

Copies

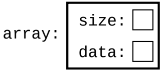
```cpp
Array<int> create(int a, int b, int c)
{
  Array<int> result { 3 };
  result[0] = a;
  result[1] = b;
  result[2] = c;
  return result;
}

int main()
{
  Array<int> array { create(1, 2, 3) };
  // ...
}
```



Lifetime & Special Member Functions

Copies

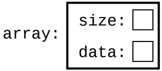
```cpp
Array<int> create(int a, int b, int c)
{
  Array<int> result { 3 };
  result[0] = a;
  result[1] = b;
  result[2] = c;
  return result;
}

int main()
{
  Array<int> array { create(1, 2, 3) };
  // ...
}
```



Lifetime & Special Member Functions

Copies

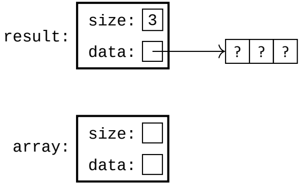
```
Array<int> create(int a, int b, int c)
{
  Array<int> result { 3 };
  result[0] = a;
  result[1] = b;
  result[2] = c;
  return result;
}

int main()
{
  Array<int> array { create(1, 2, 3) };
  // ...
}
```



Lifetime & Special Member Functions

Copies

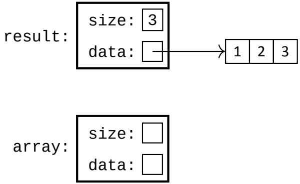
```
Array<int> create(int a, int b, int c)
{
  Array<int> result { 3 };
  result[0] = a;
  result[1] = b;
  result[2] = c;
  return result;
}

int main()
{
  Array<int> array { create(1, 2, 3) };
  // ...
}
```



Lifetime & Special Member Functions

Copies

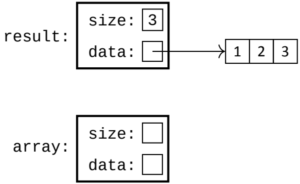
```
Array<int> create(int a, int b, int c)
{
  Array<int> result { 3 };
  result[0] = a;
  result[1] = b;
  result[2] = c;
  return result;
}

int main()
{
  Array<int> array { create(1, 2, 3) };
  // ...
}
```



Lifetime & Special Member Functions

Copies

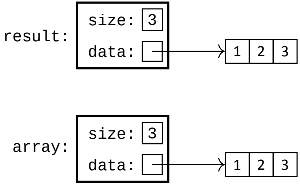
```cpp
Array<int> create(int a, int b, int c)
{
  Array<int> result { 3 };
  result[0] = a;
  result[1] = b;
  result[2] = c;
  return result;
}

int main()
{
  Array<int> array { create(1, 2, 3) };
  // ...
}
```



Lifetime & Special Member Functions

Copies

```
Array<int> create(int a, int b, int c)
{
  Array<int> result { 3 };
  result[0] = a;
  result[1] = b;
  result[2] = c;
  return result;
}

int main()
{
  Array<int> array { create(1, 2, 3) };
  // ...
}
```



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

```
Array a { create(1, 2, 3) };
Array b { create(2, 3, 4) };

b = a; // copy a into b
```

Lifetime & Special Member Functions

```
template <typename T>
class Array
{
public:
  // ...
  Array<T>& operator=(Array<T> const& other)
  {
    if (this != &other)
    {
      delete[] data;
      size = other.size;
      data = new T[size];
      for (std::size_t i { 0 }; i < size; ++i)
        data[i] = other.data[i];
    }
    return *this;
  }
  // ...
private:
  std::size_t size;
  T* data;
};
```

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

```cpp
template <typename T>
class Array
{
public:
  // ...
  Array<T>& operator=(Array<T> const& other)
  {
    // reuse copy constructor
    Array<T> copy { other };
    // swap the members
    std::swap(size, copy.size);
    std::swap(data, copy.data);
    // copy is destroyed which releases the previous data
    return *this;
  }
  // ...
private:
  std::size_t size;
  T* data;
};
```

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

```
Array<int> modify(Array<int> array)
{
  ++array[0];
  return array;
}

int main()
{
  Array<int> a { create(1, 2, 3) };
  Array<int> b { modify(a) };
}
```

Lifetime & Special Member Functions

Opportunity for optimization

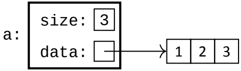
Construct a

```
Array<int> modify(Array<int> array)
{
  ++array[0];
  return array;
}

int main()
{
  Array<int> a { create(1, 2, 3) };
  Array<int> b { modify(a) };
}
```

Lifetime & Special Member Functions

Opportunity for optimization

Construct a

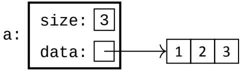



```
Array<int> modify(Array<int> array)
{
  ++array[0];
  return array;
}

int main()
{
  Array<int> a { create(1, 2, 3) };
  Array<int> b { modify(a) };
}
```

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.



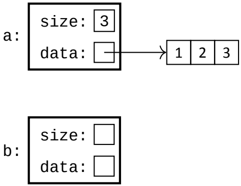

```
Array<int> modify(Array<int> array)
{
  ++array[0];
  return array;
}

int main()
{
  Array<int> a { create(1, 2, 3) };
  Array<int> b { modify(a) };
}
```

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.

```
Array<int> modify(Array<int> array)
{
  ++array[0];
  return array;
}

int main()
{
  Array<int> a { create(1, 2, 3) };
  Array<int> b { modify(a) };
}
```

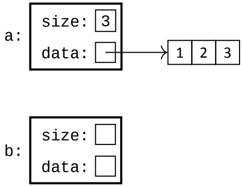


Lifetime & Special Member Functions

Opportunity for optimization

Call `modify()`

```
Array<int> modify(Array<int> array)
{
  ++array[0];
  return array;
}

int main()
{
  Array<int> a { create(1, 2, 3) };
  Array<int> b { modify(a) };
}
```

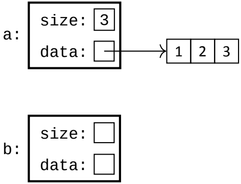


Lifetime & Special Member Functions

Opportunity for optimization

We have to copy a into `array`

```
Array<int> modify(Array<int> array)
{
  ++array[0];
  return array;
}

int main()
{
  Array<int> a { create(1, 2, 3) };
  Array<int> b { modify(a) };
}
```

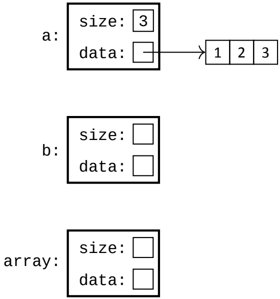


Lifetime & Special Member Functions

Opportunity for optimization

We have to copy a into `array`

```
Array<int> modify(Array<int> array)
{
  ++array[0];
  return array;
}

int main()
{
  Array<int> a { create(1, 2, 3) };
  Array<int> b { modify(a) };
}
```

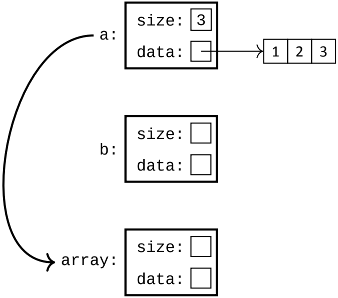


Lifetime & Special Member Functions

Opportunity for optimization

We have to copy a into `array`

```cpp
Array<int> modify(Array<int> array)
{
  ++array[0];
  return array;
}

int main()
{
  Array<int> a { create(1, 2, 3) };
  Array<int> b { modify(a) };
}
```

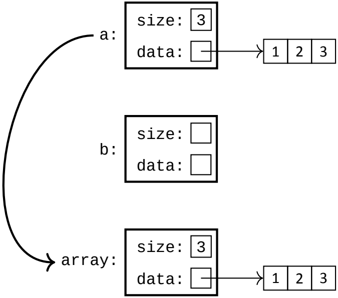


Lifetime & Special Member Functions

Opportunity for optimization

We have to copy a into `array`

```
Array<int> modify(Array<int> array)
{
  ++array[0];
  return array;
}

int main()
{
  Array<int> a { create(1, 2, 3) };
  Array<int> b { modify(a) };
}
```

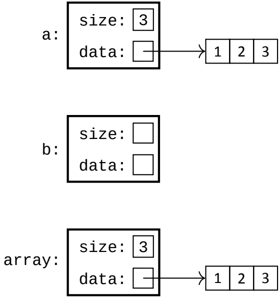


Lifetime & Special Member Functions

Opportunity for optimization

Modify the *local* array

```
Array<int> modify(Array<int> array)
{
  ++array[0];
  return array;
}

int main()
{
  Array<int> a { create(1, 2, 3) };
  Array<int> b { modify(a) };
}
```

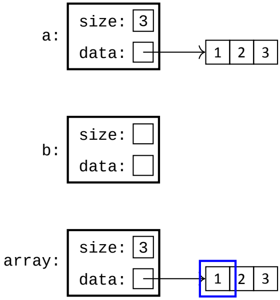


Lifetime & Special Member Functions

Opportunity for optimization

Modify the *local* array

```
Array<int> modify(Array<int> array)
{
  ++array[0];
  return array;
}

int main()
{
  Array<int> a { create(1, 2, 3) };
  Array<int> b { modify(a) };
}
```

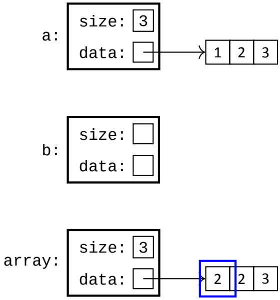


Lifetime & Special Member Functions

Opportunity for optimization

Modify the *local* array

```
Array<int> modify(Array<int> array)
{
  ++array[0];
  return array;
}

int main()
{
  Array<int> a { create(1, 2, 3) };
  Array<int> b { modify(a) };
}
```

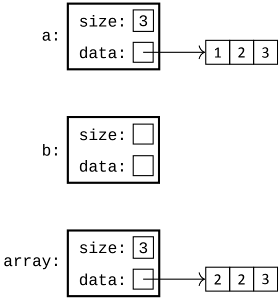


Lifetime & Special Member Functions

Opportunity for optimization

Return `array` to `main()`

```
Array<int> modify(Array<int> array)
{
  ++array[0];
  return array;
}

int main()
{
  Array<int> a { create(1, 2, 3) };
  Array<int> b { modify(a) };
}
```

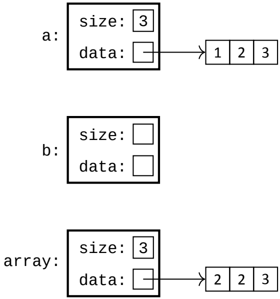


Lifetime & Special Member Functions

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.

```
Array<int> modify(Array<int> array)
{
  ++array[0];
  return array;
}

int main()
{
  Array<int> a { create(1, 2, 3) };
  Array<int> b { modify(a) };
}
```

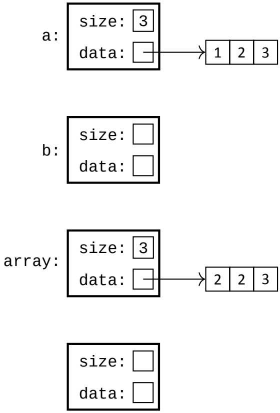


Lifetime & Special Member Functions

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.

```cpp
Array<int> modify(Array<int> array)
{
  ++array[0];
  return array;
}

int main()
{
  Array<int> a { create(1, 2, 3) };
  Array<int> b { modify(a) };
}
```

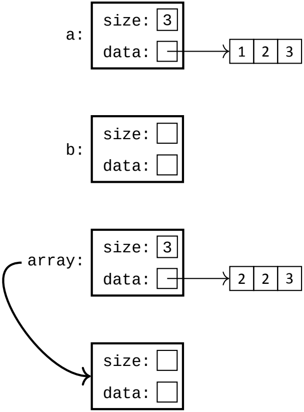


Lifetime & Special Member Functions

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.

```
Array<int> modify(Array<int> array)
{
  ++array[0];
  return array;
}

int main()
{
  Array<int> a { create(1, 2, 3) };
  Array<int> b { modify(a) };
}
```

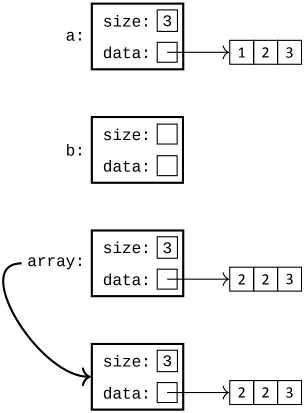


Lifetime & Special Member Functions

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.

```
Array<int> modify(Array<int> array)
{
  ++array[0];
  return array;
}

int main()
{
  Array<int> a { create(1, 2, 3) };
  Array<int> b { modify(a) };
}
```

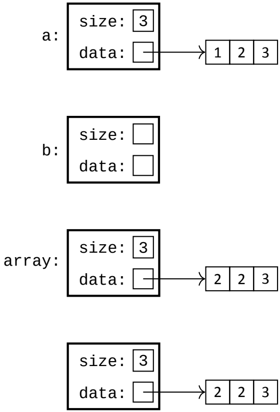


Lifetime & Special Member Functions

Opportunity for optimization

Destroy `array`

```
Array<int> modify(Array<int> array)
{
  ++array[0];
  return array;
}

int main()
{
  Array<int> a { create(1, 2, 3) };
  Array<int> b { modify(a) };
}
```

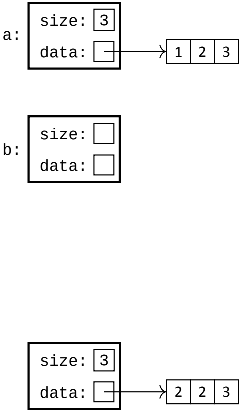


Lifetime & Special Member Functions

Opportunity for optimization

Destroy `array`

```
Array<int> modify(Array<int> array)
{
  ++array[0];
  return array;
}

int main()
{
  Array<int> a { create(1, 2, 3) };
  Array<int> b { modify(a) };
}
```

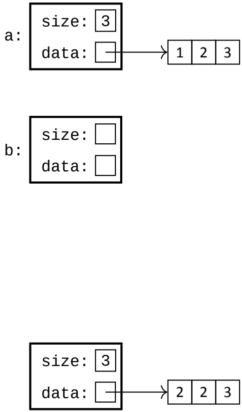


Lifetime & Special Member Functions

Opportunity for optimization

Finish construction of b by copying the temporary object

```
Array<int> modify(Array<int> array)
{
  ++array[0];
  return array;
}

int main()
{
  Array<int> a { create(1, 2, 3) };
  Array<int> b { modify(a) };
}
```

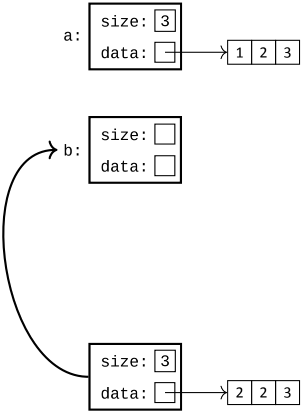


Lifetime & Special Member Functions

Opportunity for optimization

Finish construction of b by copying the temporary object

```
Array<int> modify(Array<int> array)
{
  ++array[0];
  return array;
}

int main()
{
  Array<int> a { create(1, 2, 3) };
  Array<int> b { modify(a) };
}
```

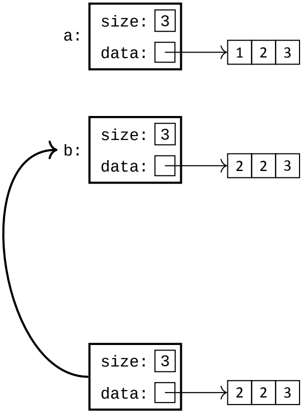


Lifetime & Special Member Functions

Opportunity for optimization

Finish construction of b by copying the temporary object

```
Array<int> modify(Array<int> array)
{
  ++array[0];
  return array;
}

int main()
{
  Array<int> a { create(1, 2, 3) };
  Array<int> b { modify(a) };
}
```

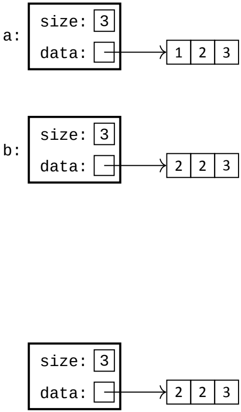


Lifetime & Special Member Functions

Opportunity for optimization

Destroy the temporary object

```
Array<int> modify(Array<int> array)
{
  ++array[0];
  return array;
}

int main()
{
  Array<int> a { create(1, 2, 3) };
  Array<int> b { modify(a) };
}
```

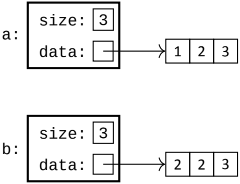


Lifetime & Special Member Functions

Opportunity for optimization

Destroy the temporary object

```
Array<int> modify(Array<int> array)
{
  ++array[0];
  return array;
}

int main()
{
  Array<int> a { create(1, 2, 3) };
  Array<int> b { modify(a) };
}
```

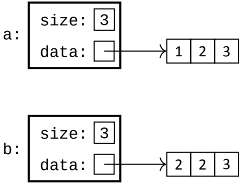


Lifetime & Special Member Functions

Opportunity for optimization

We are done!

```
Array<int> modify(Array<int> array)
{
  ++array[0];
  return array;
}

int main()
{
  Array<int> a { create(1, 2, 3) };
  Array<int> b { modify(a) };
}
```

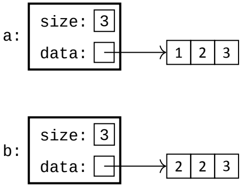


Lifetime & Special Member Functions

Opportunity for optimization

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

```cpp
Array<int> modify(Array<int> array)
{
  ++array[0];
  return array;
}

int main()
{
  Array<int> a { create(1, 2, 3) };
  Array<int> b { modify(a) };
}
```

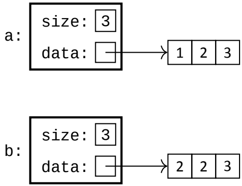


Lifetime & Special Member Functions

Opportunity for optimization

We made *four* dynamic allocations!

```
Array<int> modify(Array<int> array)
{
  ++array[0];
  return array;
}

int main()
{
  Array<int> a { create(1, 2, 3) };
  Array<int> b { modify(a) };
}
```



Lifetime & Special Member Functions

We can do better!

Lifetime & Special Member Functions

Opportunity for optimization

```
Array<int> modify(Array<int> array)
{
  ++array[0];
  return array;
}

int main()
{
  Array<int> a { create(1, 2, 3) };
  Array<int> b { modify(a) };
}
```

Lifetime & Special Member Functions

Opportunity for optimization

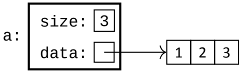
Construct a

```
Array<int> modify(Array<int> array)
{
  ++array[0];
  return array;
}

int main()
{
  Array<int> a { create(1, 2, 3) };
  Array<int> b { modify(a) };
}
```

Lifetime & Special Member Functions

Opportunity for optimization

Construct a

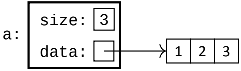



```cpp
Array<int> modify(Array<int> array)
{
  ++array[0];
  return array;
}

int main()
{
  Array<int> a { create(1, 2, 3) };
  Array<int> b { modify(a) };
}
```

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.



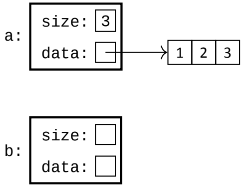

```
Array<int> modify(Array<int> array)
{
  ++array[0];
  return array;
}

int main()
{
  Array<int> a { create(1, 2, 3) };
  Array<int> b { modify(a) };
}
```

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.

```
Array<int> modify(Array<int> array)
{
  ++array[0];
  return array;
}

int main()
{
  Array<int> a { create(1, 2, 3) };
  Array<int> b { modify(a) };
}
```

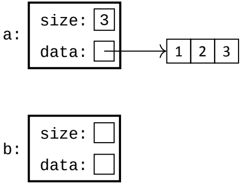


Lifetime & Special Member Functions

Opportunity for optimization

Call `modify()`

```
Array<int> modify(Array<int> array)
{
  ++array[0];
  return array;
}

int main()
{
  Array<int> a { create(1, 2, 3) };
  Array<int> b { modify(a) };
}
```

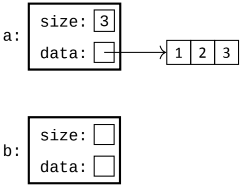


Lifetime & Special Member Functions

Opportunity for optimization

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

```
Array<int> modify(Array<int> array)
{
  ++array[0];
  return array;
}

int main()
{
  Array<int> a { create(1, 2, 3) };
  Array<int> b { modify(a) };
}
```

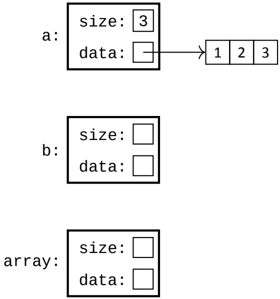


Lifetime & Special Member Functions

Opportunity for optimization

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

```
Array<int> modify(Array<int> array)
{
  ++array[0];
  return array;
}

int main()
{
  Array<int> a { create(1, 2, 3) };
  Array<int> b { modify(a) };
}
```

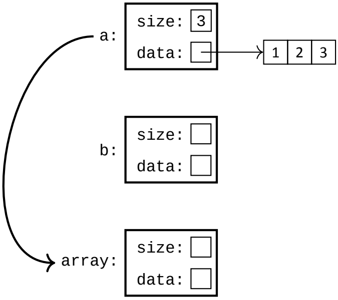


Lifetime & Special Member Functions

Opportunity for optimization

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

```
Array<int> modify(Array<int> array)
{
  ++array[0];
  return array;
}

int main()
{
  Array<int> a { create(1, 2, 3) };
  Array<int> b { modify(a) };
}
```

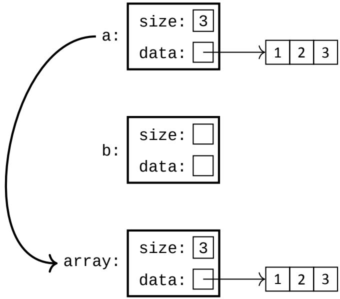


Lifetime & Special Member Functions

Opportunity for optimization

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

```
Array<int> modify(Array<int> array)
{
  ++array[0];
  return array;
}

int main()
{
  Array<int> a { create(1, 2, 3) };
  Array<int> b { modify(a) };
}
```

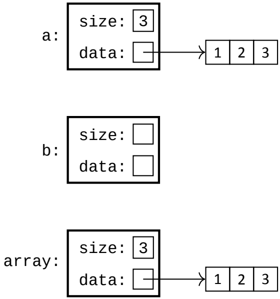


Lifetime & Special Member Functions

Opportunity for optimization

Modify the *local* array

```
Array<int> modify(Array<int> array)
{
  ++array[0];
  return array;
}

int main()
{
  Array<int> a { create(1, 2, 3) };
  Array<int> b { modify(a) };
}
```

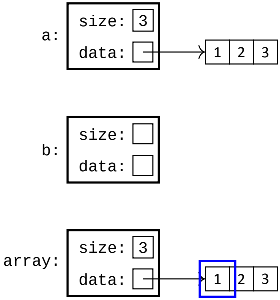


Lifetime & Special Member Functions

Opportunity for optimization

Modify the *local* array

```
Array<int> modify(Array<int> array)
{
  ++array[0];
  return array;
}

int main()
{
  Array<int> a { create(1, 2, 3) };
  Array<int> b { modify(a) };
}
```

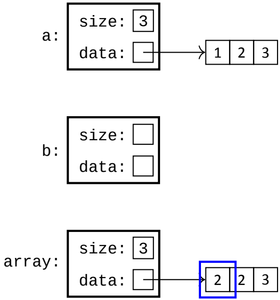


Lifetime & Special Member Functions

Opportunity for optimization

Modify the *local* array

```
Array<int> modify(Array<int> array)
{
  ++array[0];
  return array;
}

int main()
{
  Array<int> a { create(1, 2, 3) };
  Array<int> b { modify(a) };
}
```

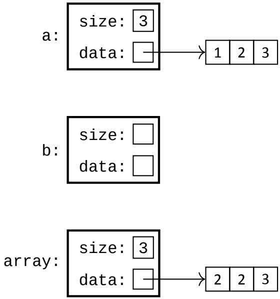


Lifetime & Special Member Functions

Opportunity for optimization

Return `array` to `main()`

```
Array<int> modify(Array<int> array)
{
  ++array[0];
  return array;
}

int main()
{
  Array<int> a { create(1, 2, 3) };
  Array<int> b { modify(a) };
}
```

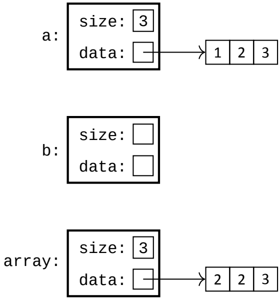


Lifetime & Special Member Functions

Opportunity for optimization

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

```
Array<int> modify(Array<int> array)
{
  ++array[0];
  return array;
}

int main()
{
  Array<int> a { create(1, 2, 3) };
  Array<int> b { modify(a) };
}
```

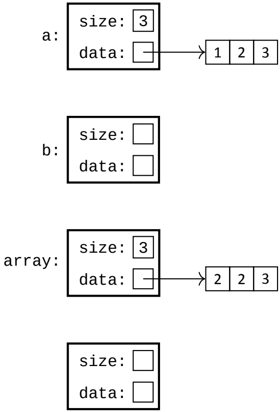


Lifetime & Special Member Functions

Opportunity for optimization

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

```
Array<int> modify(Array<int> array)
{
  ++array[0];
  return array;
}

int main()
{
  Array<int> a { create(1, 2, 3) };
  Array<int> b { modify(a) };
}
```

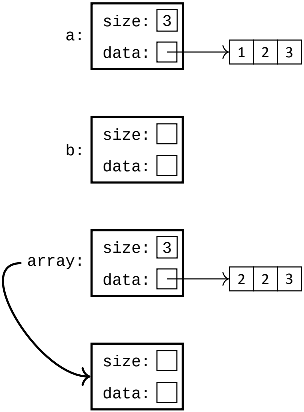


Lifetime & Special Member Functions

Opportunity for optimization

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

```
Array<int> modify(Array<int> array)
{
  ++array[0];
  return array;
}

int main()
{
  Array<int> a { create(1, 2, 3) };
  Array<int> b { modify(a) };
}
```

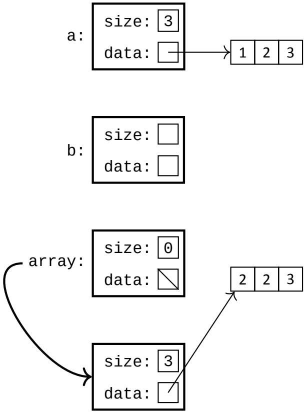


Lifetime & Special Member Functions

Opportunity for optimization

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

```
Array<int> modify(Array<int> array)
{
  ++array[0];
  return array;
}

int main()
{
  Array<int> a { create(1, 2, 3) };
  Array<int> b { modify(a) };
}
```

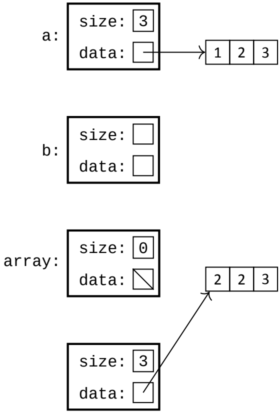


Lifetime & Special Member Functions

Opportunity for optimization

Destroy `array`

```
Array<int> modify(Array<int> array)
{
  ++array[0];
  return array;
}

int main()
{
  Array<int> a { create(1, 2, 3) };
  Array<int> b { modify(a) };
}
```

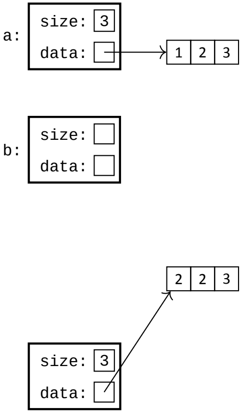


Lifetime & Special Member Functions

Opportunity for optimization

Destroy `array`

```
Array<int> modify(Array<int> array)
{
  ++array[0];
  return array;
}

int main()
{
  Array<int> a { create(1, 2, 3) };
  Array<int> b { modify(a) };
}
```

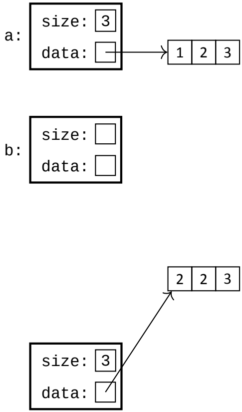


Lifetime & Special Member Functions

Opportunity for optimization

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

```
Array<int> modify(Array<int> array)
{
  ++array[0];
  return array;
}

int main()
{
  Array<int> a { create(1, 2, 3) };
  Array<int> b { modify(a) };
}
```

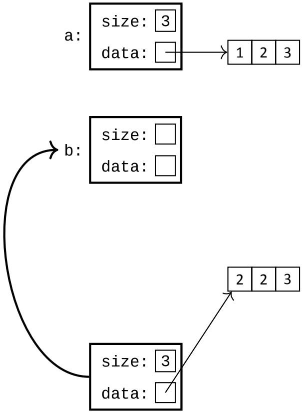


Lifetime & Special Member Functions

Opportunity for optimization

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

```
Array<int> modify(Array<int> array)
{
  ++array[0];
  return array;
}

int main()
{
  Array<int> a { create(1, 2, 3) };
  Array<int> b { modify(a) };
}
```

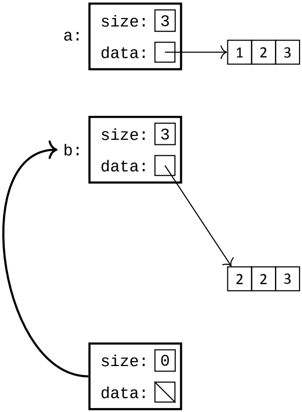


Lifetime & Special Member Functions

Opportunity for optimization

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

```
Array<int> modify(Array<int> array)
{
  ++array[0];
  return array;
}

int main()
{
  Array<int> a { create(1, 2, 3) };
  Array<int> b { modify(a) };
}
```

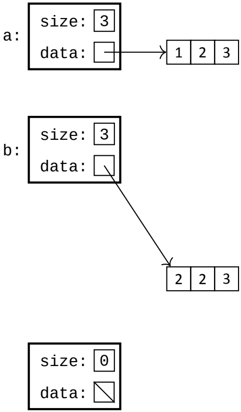


Lifetime & Special Member Functions

Opportunity for optimization

Destroy the temporary object

```cpp
Array<int> modify(Array<int> array)
{
  ++array[0];
  return array;
}

int main()
{
  Array<int> a { create(1, 2, 3) };
  Array<int> b { modify(a) };
}
```

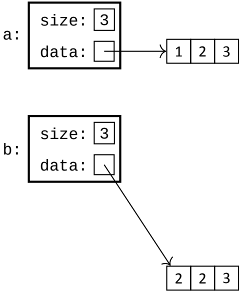


Lifetime & Special Member Functions

Opportunity for optimization

Destroy the temporary object

```
Array<int> modify(Array<int> array)
{
  ++array[0];
  return array;
}

int main()
{
  Array<int> a { create(1, 2, 3) };
  Array<int> b { modify(a) };
}
```



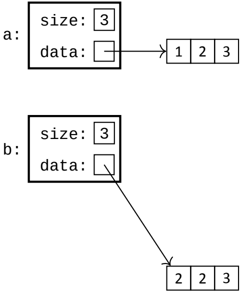
LINKÖPING UNIVERSITY

Lifetime & Special Member Functions

Opportunity for optimization

We are done!

```
Array<int> modify(Array<int> array)
{
  ++array[0];
  return array;
}

int main()
{
  Array<int> a { create(1, 2, 3) };
  Array<int> b { modify(a) };
}
```

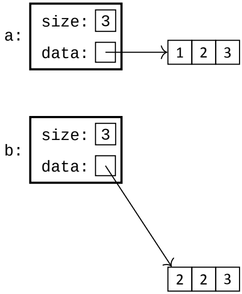


Lifetime & Special Member Functions

Opportunity for optimization

And we only made *two* dynamic allocations!

```
Array<int> modify(Array<int> array)
{
  ++array[0];
  return array;
}

int main()
{
  Array<int> a { create(1, 2, 3) };
  Array<int> b { modify(a) };
}
```



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

```
template <typename T>
class Array
{
public:
  // ...
  Array(Array<T>&& other)
    : size { other.size }, data { other.data }
  {
    other.size = 0;
    array.data = nullptr;
  }

  Array<T>& operator=(Array<T>&& other)
  {
    std::swap(size, other.size);
    std::swap(data, other.data);
    return *this;
  }
  // ...
};
```

Lifetime & Special Member Functions

Let’s implement *move* semantics

```
template <typename T>
class Array
{
public:
  // ...
  Array(Array<T>&& other)
    : size { other.size }, data { other.data }
  {
    other.size = 0;
    array.data = nullptr;
  }

  Array<T>& operator=(Array<T>&& other)
  {
    std::swap(size, other.size);
    std::swap(data, other.data);
    return *this;
  }
  // ...
};
```

These are *rvalue* references

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<int> array { create(1, 2, 3) };`
  - `array = create(4, 5, 6);`
  - `array = std::move(other);`

Lifetime & Special Member Functions

Special member functions

```
template <typename T>
class Array
{
public:
  // ...
  Array(Array<T> const& other);                 // copy constructor
  Array(Array<T>&& other);                      // move constructor
  ~Array();                                     // destructor
  Array<T>& operator=(Array<T> const& other); // copy assignment operator
  Array<T>& operator=(Array<T>&& other);        // move assignment operator
  // ...
};
```

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.

Lifetime & Special Member Functions

When does move happen?

```
Array<int> a { create(1, 2, 3) };
Array<int> b { a }; // move?
```

- Should a be *moved into* b?

Lifetime & Special Member Functions

When does move happen?

```
Array<int> a { create(1, 2, 3) };
Array<int> b { a }; // move?
```

- Should a be *moved into* b?
- No! a might still be used afterwards, so we cannot steal its data.

Lifetime & Special Member Functions

When does move happen?

```
Array<int> fun()
{
  Array<int> result { /* ... */ };
  // ...
  return result; // move?
}
```

- Should we move `result` into the callers scope?

Lifetime & Special Member Functions

When does move happen?

```
Array<int> fun()
{
  Array<int> result { /* ... */ };
  // ...
  return result; // move?
}
```

- Should we move `result` into the callers scope?
- Only if NRVO is impossible, which is rare

Lifetime & Special Member Functions

When does move happen?

```
Array<int> fun()
{
  Array<int> result { /* ... */ };
  // ...
  return result; // move?
}
```

- Should we move `result` into the callers scope?
- Only if NRVO is impossible, which is rare
- So as a general rule: no, move should be avoided (but is prefered to copying)

Lifetime & Special Member Functions

When does move happen?

```cpp
void fun(Array<int> arr);

int main()
{
  fun(Array<int>{ /* ... */ }); // move?
}
```

- Should we move the array from the caller to the function?

Lifetime & Special Member Functions

When does move happen?

```
void fun(Array<int> arr);

int main()
{
  fun(Array<int>{ /* ... */ }); // move?
}
```

- Should we move the array from the caller to the function?
- Yes!

Lifetime & Special Member Functions

When does move happen?

```
void fun(Array<int> arr);

int main()
{
  Array<int> array { /* ... */ };
  fun(array); // move?
}
```

- Should we move the array from the caller to the function?

Lifetime & Special Member Functions

When does move happen?

```cpp
void fun(Array<int> arr);

int main()
{
  Array<int> array { /* ... */ };
  fun(array); // move?
}
```

- Should we move the array from the caller to the function?
- No!

Lifetime & Special Member Functions

When does move happen?

Move happens when:

1. The object we are trying to “copy” is temporary (i.e. an rvalue)
2. The compiler can *prove* that the object is unused afterwards (and no better optimization can be used)
3. The compiler does not look *ahead* which limits how many situations the compiler can prove...

Lifetime & Special Member Functions

When does move happen?

Copying rvalue $\Rightarrow$ move!

Lifetime & Special Member Functions

When does move happen?

Copying rvalue $\Rightarrow$ move!

Returning by value $\Rightarrow$ (at worst) move!

Lifetime & Special Member Functions

When does move happen?

- The only situations where the compiler can definitively say that an object will not be used afterwards is exactly those situations
- I.e. when the object we are copying is temporary (an rvalue) or when returning by-value from functions
- Because of this, the easiest way to implement move seems to be to just make an overload of the copy operation, but filtered on *only* rvalues
- Enter rvalue references!

Lifetime & Special Member Functions

rvalue reference

```
template <typename T>
class Array
{
  // ...
  Array(Array const& other); // constant lvalues
  Array(Array&& other);      // rvalues
  // ...
};
```

Lifetime & Special Member Functions

rvalue reference

- Non-constant lvalues can be converted to constant lvalues
- However, rvalues can also be converted to constant lvalues
- So strictly speaking, copying is *enough* to make all situations *work*
- But to enable move semantics, we need to *detect* rvalues and pick corresponding overload
- In C++11 this was finally solved with the introduction of *rvalue references*

Lifetime & Special Member Functions

rvalue reference

Normal reference (lvalue references) can only bind to non-constant objects with identity (lvalues)

```
int x { 5 };
int const y { 3 };

int& lref1 { x };          // OK
int& lref2 { 5 };          // NOT OK
int& lref3 { y };          // NOT OK
```

Lifetime & Special Member Functions

rvalue reference

Constant lvalue references can bind to anything, since non-const lvalues and rvalues can both be implicitly converted to constant lvalues

```
int x { 5 };
int const y { 3 };

int const& clref1 { x }; // OK
int const& clref2 { 5 }; // OK
int const& clref3 { y }; // OK
```

Lifetime & Special Member Functions

rvalue reference

Rvalue references can *only* bind to rvalue references, i.e. things that does **NOT** have identity

```
int x { 5 };
int const y { 3 };

int&& rref1 { x };        // NOT OK
int&& rref2 { 5 };        // OK
int&& rref3 { y };        // NOT OK
```

Lifetime & Special Member Functions

rvalue reference

This is all summarized in this table

reference	lvalue	constant lvalue	rvalue
&	OK	NOT OK	NOT OK
`const`&	OK*	OK	OK*
&&	NOT OK	NOT OK	OK

* Requires type conversions

Lifetime & Special Member Functions

rvalue reference

- Because of these rules it is apparent that constant lvalue references are used for copy semantics since 1) it works for every value category and 2) it disallows us from accidentally modifying the object we are copying from.
- However, move requires us to actually modify the original object (since we want to steal its content), but this should only be done for rvalues (since those are the only value categories the compiler can prove are not used afterwards).
- Since rvalues can be converted to constant lvalues the copy overload is always *viable* but due to overload resolution the rvalue reference overload will always have higher priority.
- This shows that rvalue references is *how* move semantics is implemented.
- But the pressing question now is, what consequences does this have?

Lifetime & Special Member Functions

But can we trigger move manually?

```
#include <utility>

void fun(Array<int> arr);

int main()
{
  Array<int> array { /* ... */ };
  fun(std::move(array)); // <-- force a move
  // `array` is now empty
}
```

1 As-if rule
2 Copy elison
3 Lifetime & Special Member Functions
4 Extended value categories

LINKÖPING UNIVERSITY

Extended value categories

Is this an rvalue or an lvalue?

Rvalue references produces some strange results...

```
void fun(Array<int>&& other)
{
  // ...
}

int main()
{
  fun(Array<int>{ /* .. */ });
}
```

Extended value categories

Is this an rvalue or an lvalue?

What is the value category of the argument passed to `fun()`?

```
void fun(Array<int>&& other)
{
  // ...
}

int main()
{
  fun(Array<int>{ /* .. */ });
}
```

Extended value categories

Is this an rvalue or an lvalue?

It is a temporary, identity-less value, so it is an rvalue!

```
void fun(Array<int>&& other)
{
  // ...
}

int main()
{
  fun(Array<int>{ /* .. */ }); rvalue!
}
```

Extended value categories

Is this an rvalue or an lvalue?

But what about the `other` parameter in `fun()`?

```
void fun(Array<int>&& other)
{
  // ...
}

int main()
{
  fun(Array<int>{ /* .. */ }); rvalue!
}
```

Extended value categories

Is this an rvalue or an lvalue?

It is a reference that is bound to an rvalue, so it is clearly temporary...

```
void fun(Array<int>&& other) rvalue?
{
  // ...
}

int main()
{
  fun(Array<int>{ /* .. */ });rvalue!
}
```

Extended value categories

Is this an rvalue or an lvalue?

... but since we bound it to a reference it now clearly has identity...

```
void fun(Array<int>&& other)  lvalue?
{
  // ...
}

int main()
{
  fun(Array<int>{ /* .. */ });rvalue!
}
```

Extended value categories

Is this an rvalue or an lvalue?

> So strictly speaking it is an lvalue, but it conceptually should be thought of as an rvalue. We dub this an *xvalue*

```
void fun(Array<int>&& other)  xvalue!
{
  // ...
}

int main()
{
  fun(Array<int>{ /* .. */ });rvalue!
}
```

Extended value categories

Old terminology

- **lvalues**: Expressions which is an identity

Extended value categories

Old terminology

- **lvalues**: Expressions which is an identity
- **rvalues**: Expressions that produces temporary objects

Extended value categories

Altered terminology

- **xvalues**: Expressions that refers to an *expiring* identity

Extended value categories

Altered terminology

- **xvalues**: Expressions that refers to an *expiring* identity
- **lvalues**: Non-**xvalue** expression that is an identity

Extended value categories

Altered terminology

- **xvalues**: Expressions that refers to an *expiring* identity
- **lvalues**: Non-**xvalue** expression that is an identity
- **rvalues**: Non-**xvalue** expression producing a temporary

Extended value categories

Altered terminology

- **xvalues**: Expressions that refers to an *expiring* identity
- **lvalues**: Non-**xvalue** expression that is an identity
- **rvalues**: Non-**xvalue** expression producing a temporary
- This is confusing...

Extended value categories

Altered terminology

- An expiring identity is a temporary identity that is given to a temporary object (produced from an rvalue expression)
- The terminology is very specific
- To simplify the terminology and the definitions we restructure the value categories into five different categories
- This is mainly done to more easily refer to different categores
- We introduce *generalized lvalues* (**glvalues**) and *pure rvalues* (**prvalues**)

Extended value categories

New terminology

- **glvalues**: has identity (previous lvalues)

Extended value categories

New terminology

- **glvalues**: has identity (previous lvalues)
- **xvalues**: has an expiring identity

Extended value categories

New terminology

- **glvalues**: has identity (previous lvalues)
- **xvalues**: has an expiring identity
- **lvalues**: **glvalues** that are not **xvalues**

Extended value categories

New terminology

- **glvalues**: has identity (previous lvalues)
- **xvalues**: has an expiring identity
- **lvalues**: **glvalues** that are not **xvalues**
- **prvalue**: refers to temporary object

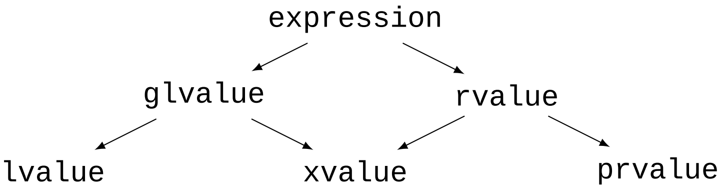
Extended value categories

New terminology

- **glvalues**: has identity (previous lvalues)
- **xvalues**: has an expiring identity
- **lvalues**: **glvalues** that are not **xvalues**
- **prvalue**: refers to temporary object
- **rvalue**: an **xvalue** or **prvalue** (previous rvalues)

Extended value categories

Full picture



www.liu.se

LINKÖPING UNIVERSITY