Data Structures & Algo - Arrays

Table of Contents:

ARRAYS

Objectives:

1. Safe Array memory management
2. Element access in bounds
3. Coding techniques

Array: Collection of elements using contiguous memory

- Contiguous memory allocation is cache friendly for CPU's therefore fast
- All elements share same data type (homogenous)
- Allow easy access to array elements with index
- Quickly directly access an array element which is a bigO(1) or constant runtime
- The memory is allocated in one spot together

By default you should use provided data structures in c++ standard library

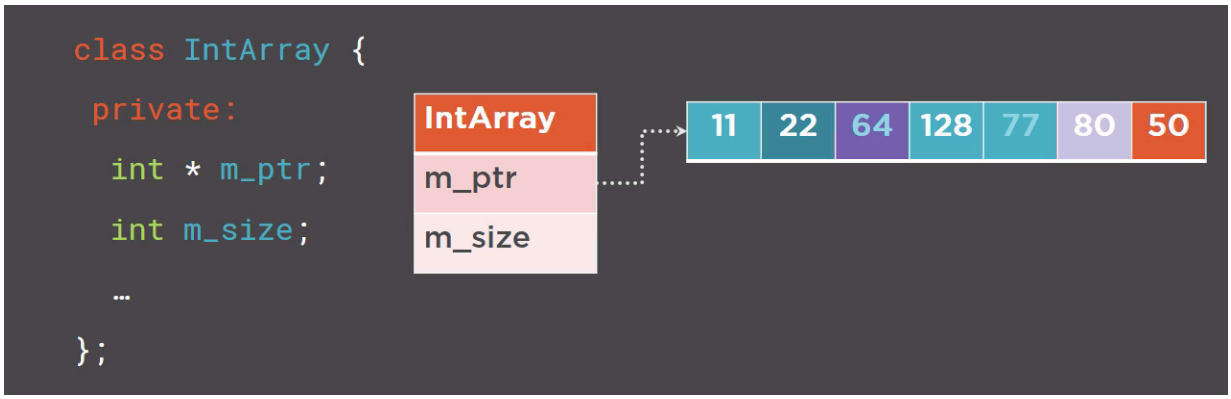
- The benefit of creating your own DS is to create an optimized but less versatile code.

Stack versus heap

- stack is faster but has limited space
- stack is ideal for smaller amounts of data
- whereas heap can hold large data such as pictures

When finished with array make sure to delete to prevent memory leak

Pictured is a bare-bone initialization of an array:



There are two fields that are needed, a pointer and a size.

- The pointer is allocated elsewhere, and chooses an element to describe

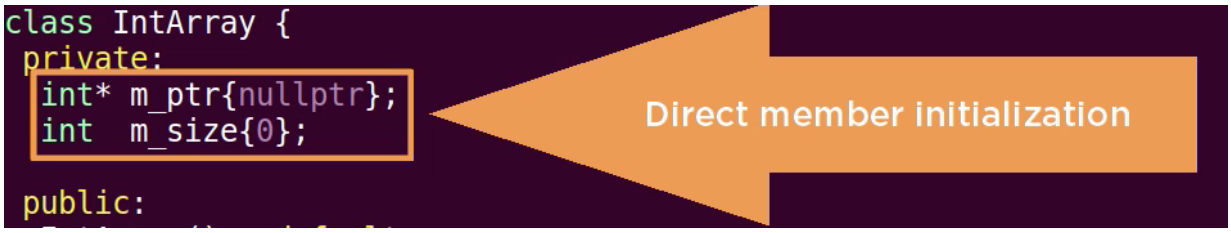
- The size determines the amount of elements

EXAMPLE CODE:

```
class IntArray {
 private:
  int* m_ptr{nullptr};
  int  m_size{0};

 public:
  IntArray() = default;

  explicit IntArray(int size) {
    if (size != 0) {
      m_ptr = new int[size]{};
      m_size = size;
    }
  }

  int Size() const {
    return m_size;
  }

  bool IsEmpty() const {
    return (m_size == 0);
  }
};
```

The pointer and size is directly initialized in the private section to keep them safe (Also known as the implementation details of the custom array class)

Ex.



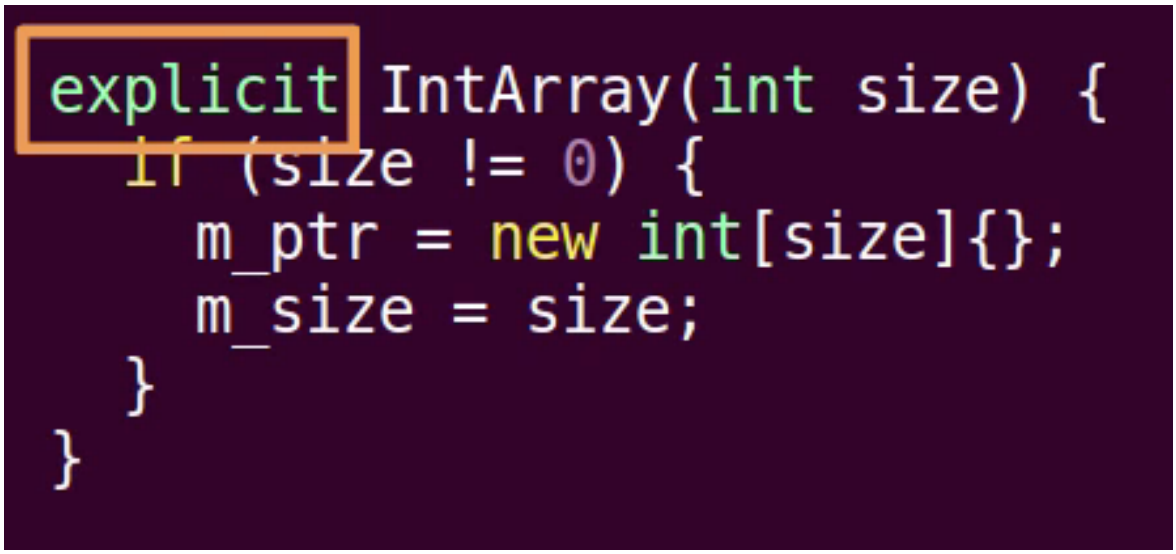
In the public section an empty array is created from the default constructor specified in the private section

Ex.

```
public:
 IntArray() = default;
```

The explicit section creates an array with the size of a given parameter.
The explicit keyword safeguards one-argument constructors from conversion. Otherwise the int parameter would be implicitly converted to an IntArray;

Ex.

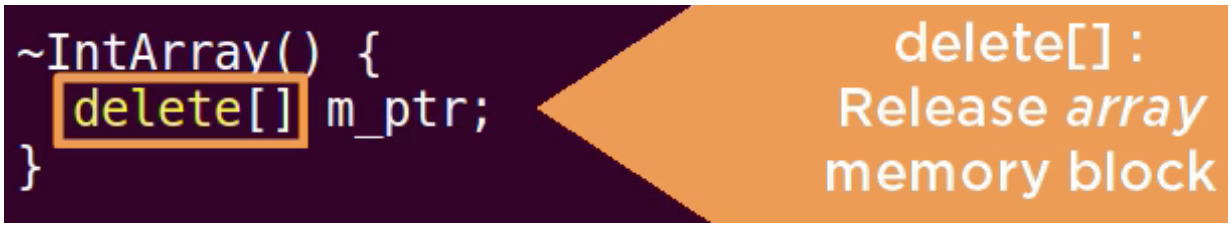


m_ptr = new int[size]{ }; - The open brackets properly initialize the array full of empty values rather than random

The other constructs have a function that return values of the array. They are marked const because they do not modify the array and are read-only.

Make sure to delete or use destructor to release memory
Memory allocated with new must be deleted. Using a destructor in a data structure will automatically remove when exiting scope

Ex.



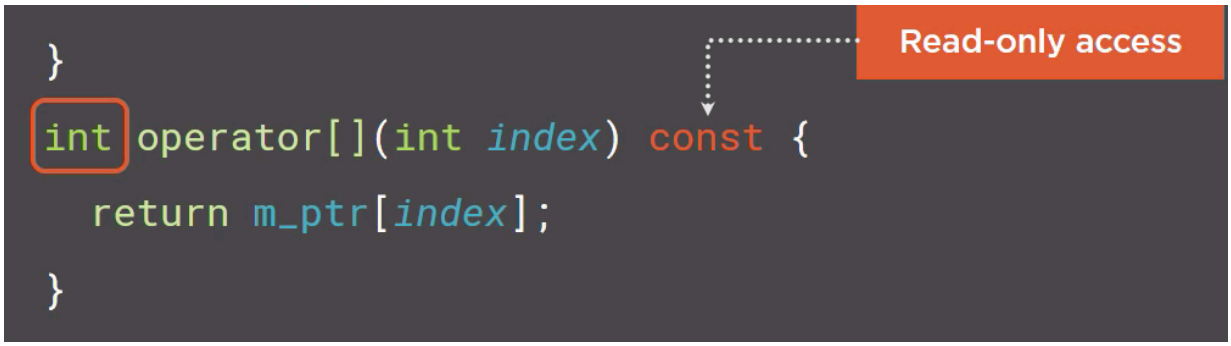
To allow a user to use the pointer from the private section we must create an overloaded constructor
In the example below, the overloaded operator[] returns the element that the pointer references

Ex.

```
int& operator[](int index) {
   return m_ptr[index];
}
```

You can also Implement it as a const so that it is read-only

Ex.



SAFELY ACCESSING ARRAY ELEMENTS:

- Bounds Checking

If the given element is out of bounds throw an exception

Ex.

```
int& operator[](int index) {
  if (! IsValidIndex(index)) {
    throw IndexOutOfBoundsException{};
  }
  return m_ptr[index];
}
```

you create a new exception class that provides information about the error and aborts the program

Using a try {

We attempt to access all the elements in the array
If there's an out of bounds we catch it and display a custom error message

Ex.

```
IntArray a{10};
for (int i = 0; i < a.Size(); i++) {
  a[i] = (i+1)*10;
}

cout << " Array elements: ";
for (int i = 0; i < a.Size(); i++) {
  cout << a[i] << ' ';
}
cout << '\n';

cout << " Array size is " << a.Size() << "\n";
cout << " Please enter an array index: ";
int index{};
cin >> index;

cout << " The element at index " << index << " is " << a[index] << '\n';
```

} catch {

```
catch (const IndexOutOfBoundsException& e) {
cout << "\n *** ERROR: Invalid array index!! \n";
```

** Notice how the class that's called references the original message that we set in IndexOutOfBounds
}

Block will try to access the elements but will catch the exception

USING CUSTOM ARRAY:

Tools:

- assert is a run-time check command
- Asan is address sanitizer which is a debugging tool
- This can be used to detect memory leaks
- In the code below, we create a new array but do not delete when done

```
cout << " Creating an empty array.\n";
IntArray a{};
cout << " a.Size() is " << a.Size() << '\n';
assert(a.IsEmpty());

cout << " ----------------------------------------\n";

cout << " Creating an array containing 10 elements.\n";
IntArray b{10};
cout << " b.Size() is " << b.Size() << '\n';
assert(! b.IsEmpty());
```

Output:

```
Creating an empty array.
a.Size() is 0
----------------------------------------
Creating an array containing 10 elements.
b.Size() is 10
```

IMPLEMENTATION

Objectives:

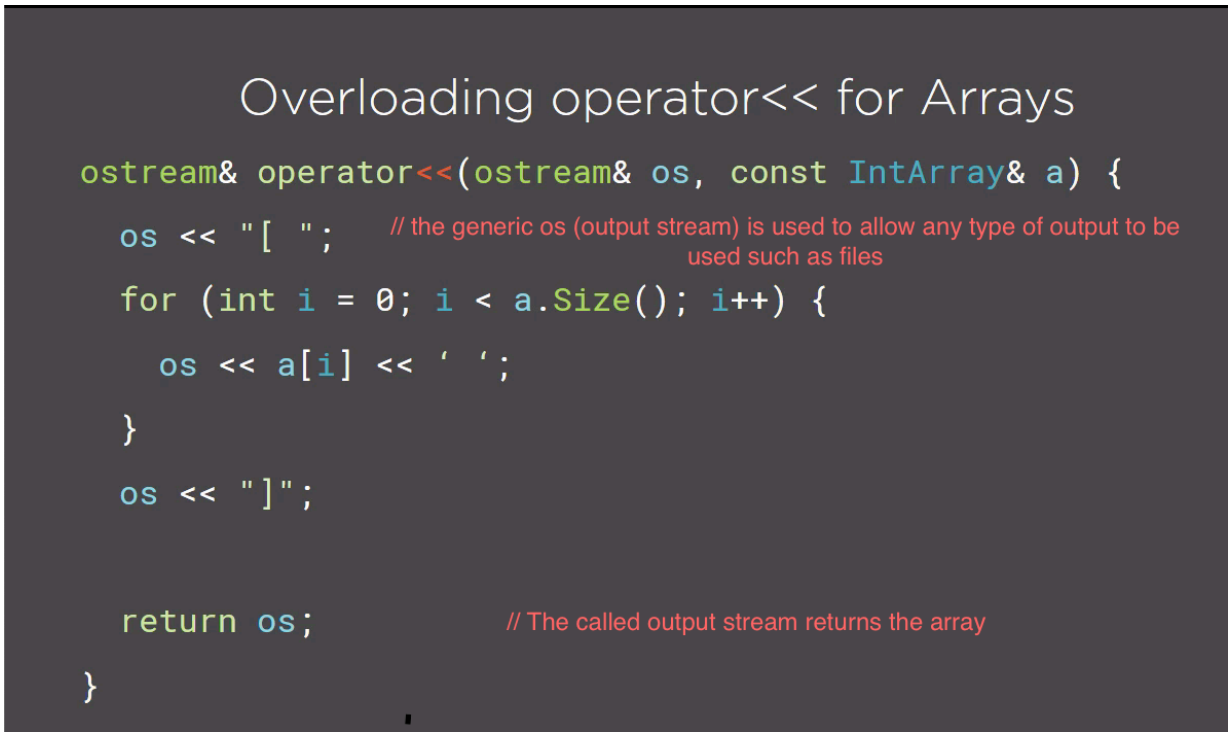
1. Using cout to print array
2. Copying arrays
    - Shallow vs deep copy
    - Copy and swap
3. Move semantics
4. Generic Array<T>

Using Cout:

- you can overload the insertion operator “<<“ by editing the stream (output stream)
- using a generic ostream(output stream) allows any form of output to use this overloaded function. This includes output files etc.

The constructor looks like:

Ex.



URL/example code:

https://github.com/AnthonyRonca/DataStructuresStudyGuide/blob/master/CustomArrayImplementation.cpp

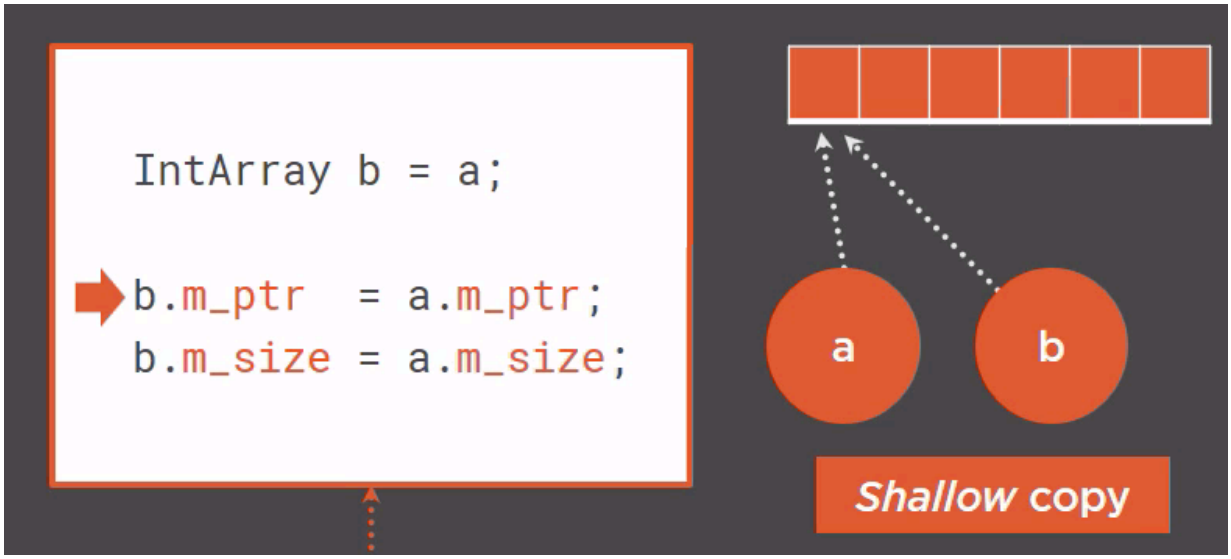
```cpp
#include <iostream>

using namespace std;

class IntArray { // Custom Array Class
    private:
        int* m_ptr{nullptr}; // Index pointer
        int m_size{0};       // Size of array

    public:
    IntArray() = default;    // Default constructor (no-arg)

    explicit IntArray(int size){        // Explicit keyword prevents conversion of parameter size to Int array
        if (size != 0) {
            m_ptr = new int[size]{};    // Creates Array of provided size
            m_size = size;
        }
    }

    // Overloading [] operator to access elements in array style
    int& operator[](int index) {
        return m_ptr[index];
    }

    int operator[](int index) const {
        return m_ptr[index];
    } // overloads operator to allow read-only access
```

```cpp
    int Size() const { // return size of array
        return m_size;
    }

    ~IntArray() { // deconstructor that allows memory to be deleted when done
        delete[] m_ptr;
    }

};

ostream& operator<<(ostream& os, const IntArray& a) { // Overloads << operator allowing arrays to be output with cout

    os << "[ ";
    for(int i = 0; i < a.Size(); i++) {
        os << a[i] << ' ';
    }
    os << ']';

    return os; // Since a generic ostream is used, any form of output including files can use this constructor
}

int main(int argc, const char * argv[]) {

    IntArray a{10}; // declare array of size 10
    for (int i = 0; i < a.Size(); i++) {
        a[i] = (i + 1) * 10; // populate array with integers
    }

    cout << " Array elements: " << a << "\n";
    // Terminal output:  "Array elements: [ 10 20 30 40 50 60 70 80 90 100 ]"
```

COPYING ARRAYS:

Default member-wise copy

- “Shallow copy” points to same memory block
- Example -> *“IntArray b = a;”*
- If one calls the delete function, it frees both



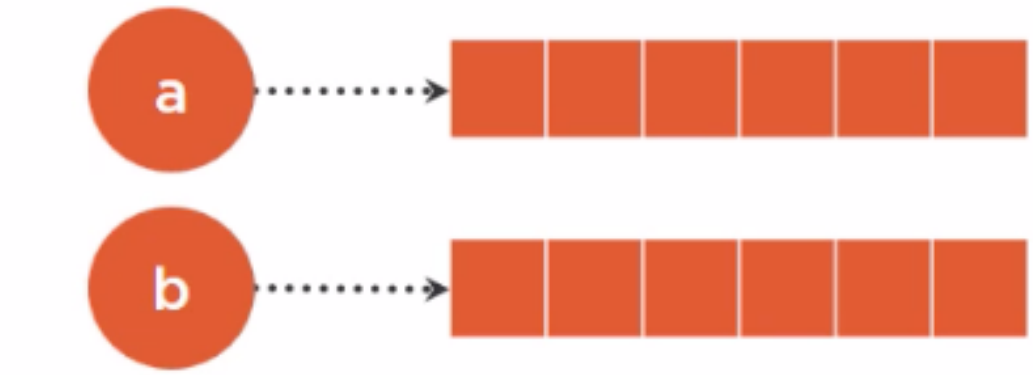
If you want to make a proper copy you have to implement a custom copy constructor AKA use the example below to create “deep” copies

Deep copy

- creates a separate array with it’s own memory block

- First allocate memory for the copy, then copy the original array into that clone
- Must write custom code to make the default

Ex.



This code overloads the copy constructor with a custom constructor that utilizes a for loop

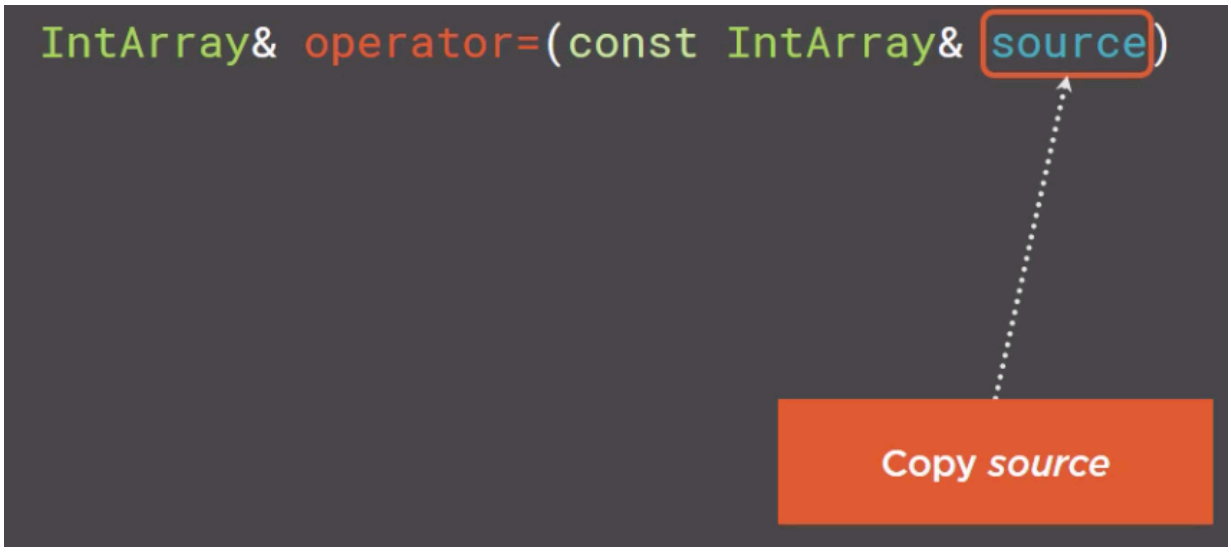
```
IntArray(const IntArray& source) {
  if (! source.IsEmpty()) {
    m_size = source.m_size;

    m_ptr = new int[m_size]{};

    for (int i = 0; i < m_size; i++) {
      m_ptr[i] = source.m_ptr[i];
    }
  }
}

~IntArray() {
  delete[] m_ptr;
}

int Size() const {
  return m_size;
}

bool IsEmpty() const {
  return (m_size == 0);
```

IMPLEMENTING COPY:

- The parameter takes the source and

returns it as reference for the new clone array

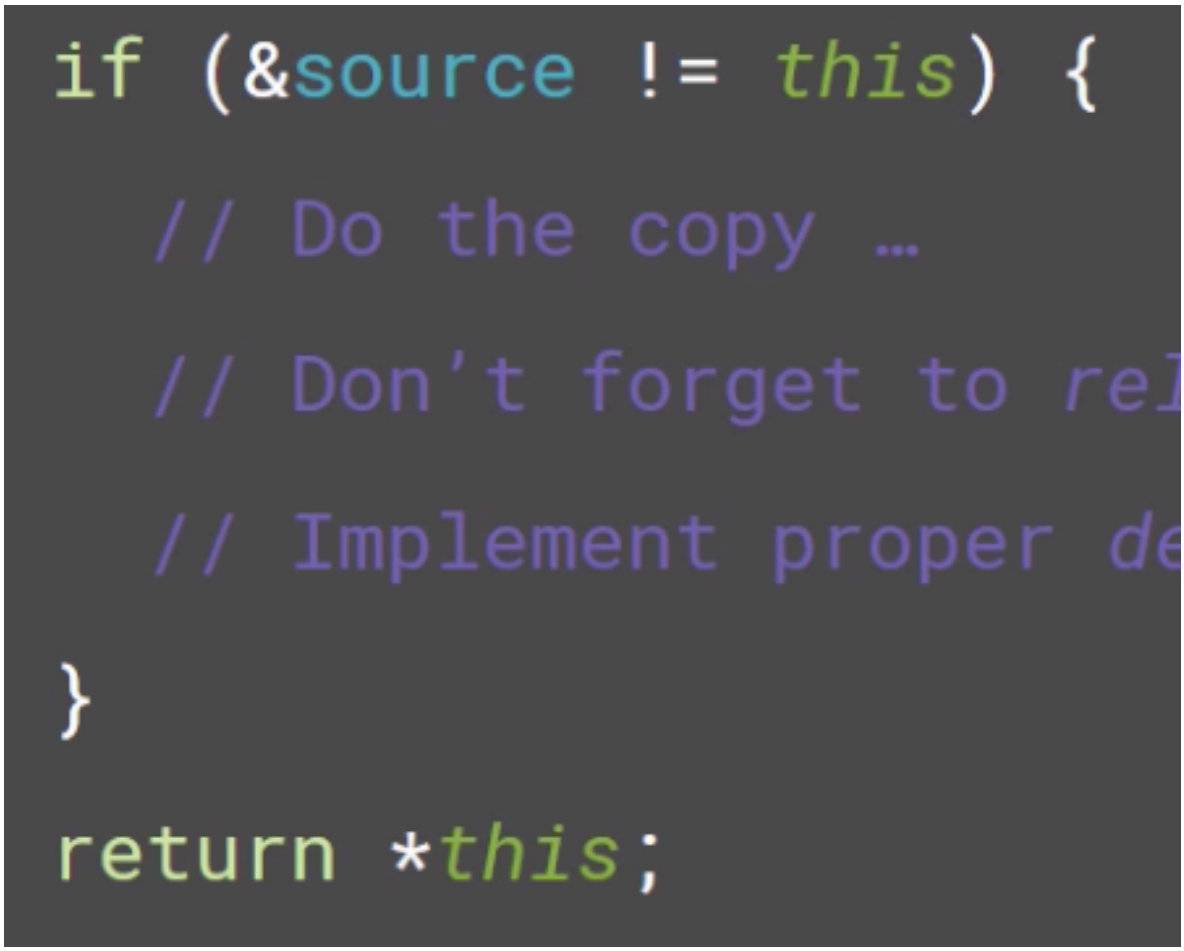
Ex.



IntArray& is an example of “this” and is referenced as such in the return statement

Also to prevent implicit conversion use an if statement to handle special cases

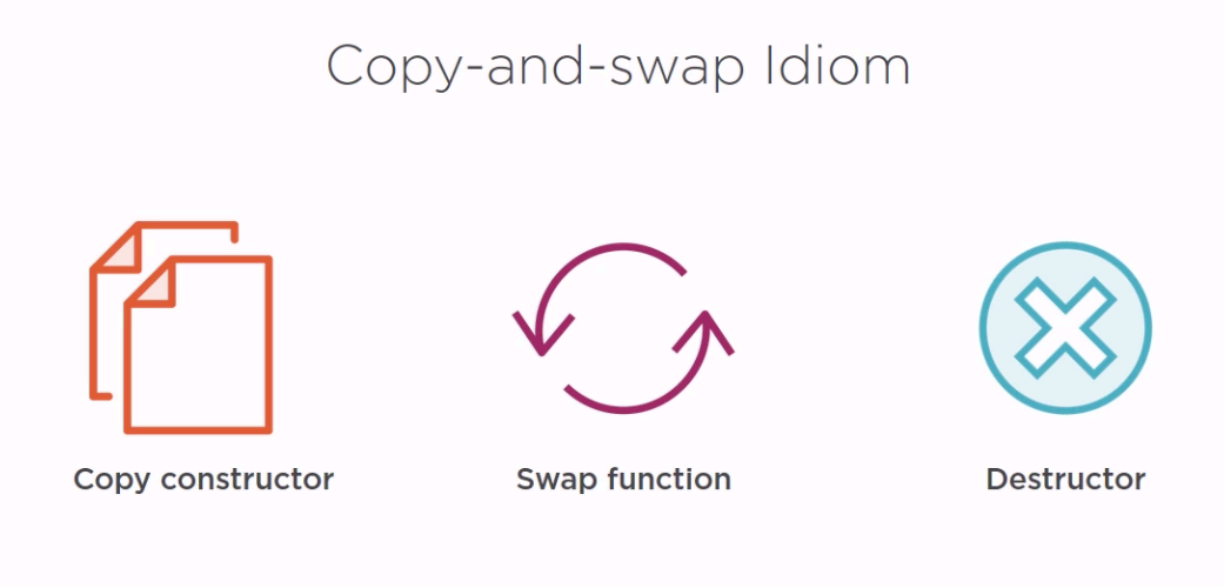
Must prevent memory leaks as well by handling the previous array block that was allocated



Copy-and-Swap Idiom

- Reuse copy constructor
- creates a copy of an array using the examples above
- Swap arrays
  - The swap algorithm is the theory of swapping two variables
  - It uses a temporary variable to store a

value while overwriting each other
    - Ex. C = a; a = b; b = c;
- Implementation includes three steps:
  1. Custom copy constructor
  2. Swap using a temporary
  3. Delete allocated data not in use after swap



Example code:

1.) copy

```
                                    COPY
                                    Pass by value
IntArray& operator=(IntArray source) {
  swap(*this, source);
  return *this;
}
```

2.) Swap

```
swap(*this, source);
```

Inside the swap function:

```
friend void swap(IntArray& a, IntArray& b) {
  using std::swap;
  // Memberwise swap
  swap(a.m_ptr,  b.m_ptr);
  swap(a.m_size, b.m_size);
}
```

ACCESS PRIVATE MEMBERS
IntArray's *m_ptr* and *m_size*

- Friend keyword means it's not a part of the class but has access to private members within the class
- std::swap; means that the standard library

swap was used

- The function swaps a& b ptr and size

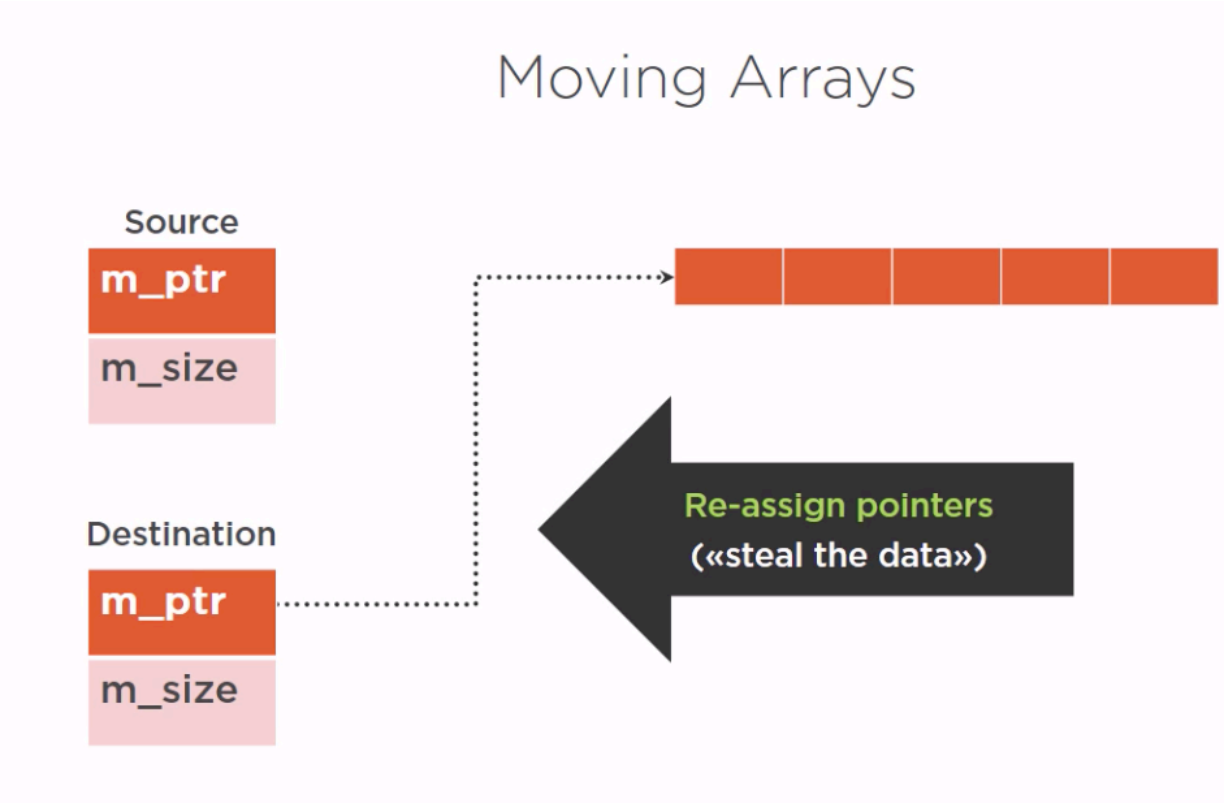
3.) Destroy

- When out of scope the delete constructor automatically runs

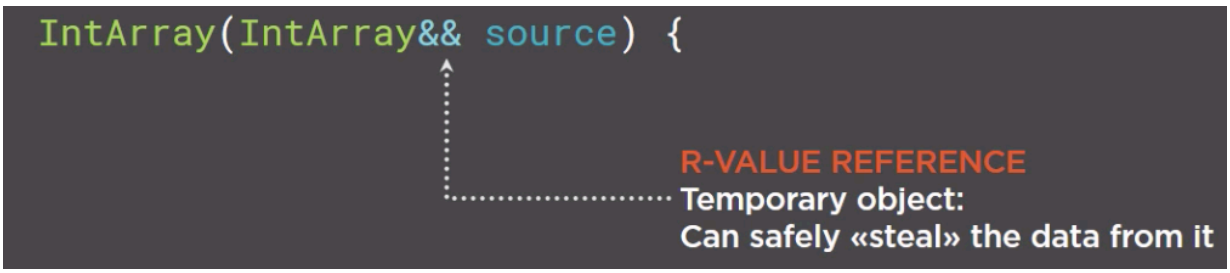
```
IntArray& operator=(IntArray source) {
  swap(*this, source);
  return *this;
} <..... DESTRUCTOR
```

Optimization

- Copying large objects is expensive and uses too much resources
    - Use move semantics which transfer without copying
    - Known as a move constructor
    - Moving an array means re-assigning the pointer



&& = R-Value reference which allows the reference to be moved



What happens in the code below:

- The new pointer and size for the destination are initialized with the source's pointer and size.
- Therefore the destination now points to the same memory.
- Finally the source is then deleted thus completing the move

Move Constructor for the Array Class

```
IntArray(IntArray&& source) {
  // Transfer ownership (steal data) from source
  m_ptr  = source.m_ptr;
  m_size = source.m_size;

  // Clear source
  source.m_ptr  = nullptr;
  source.m_size = 0;
}
```

Template for all data types:

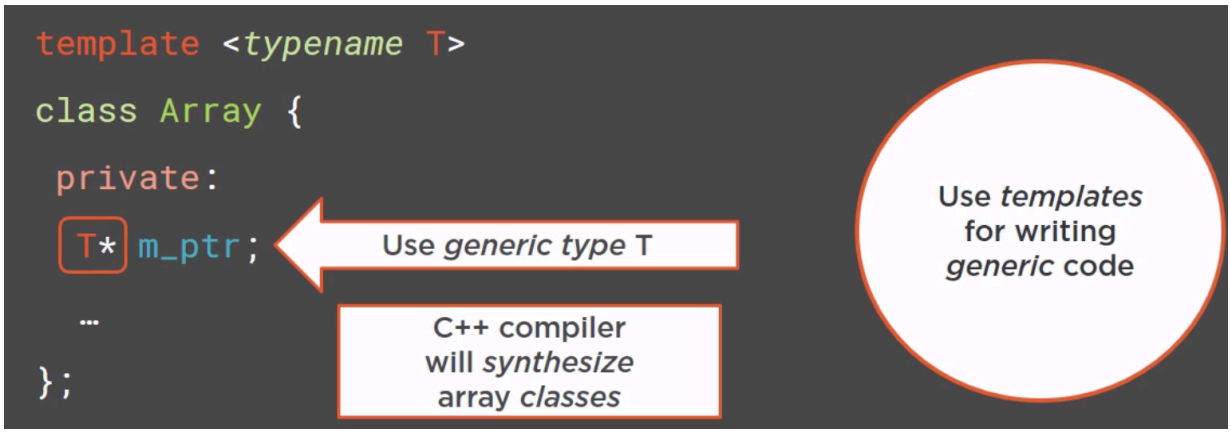
A generic array class must include the following:

1. Constructors/deconstructors
2. Bound checking
3. Deep copy capability
4. Move Semantics

The same logic of arrays from above can be used for different data types. The same exact code can be used if you create the proper template. Using a generic type allows the array to be used. For example:

T can handle any data type given to it.

Creating a template:



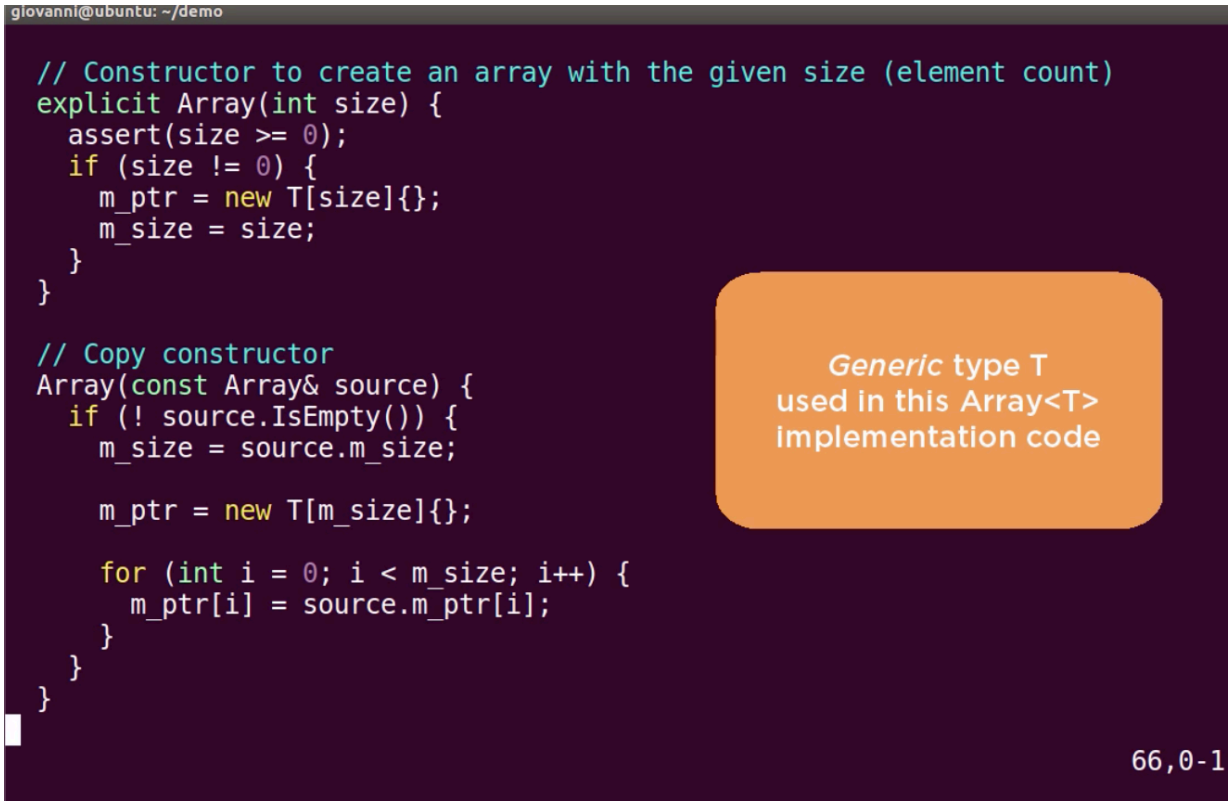
This class template is included in the header file.
This method doesn't affect performance and runs at compile time.
Header files end in .hpp and must be included in the header file to be used in the program

Ex.

```cpp
// Array.hpp -- A generic array class template
//
//              Implements several features from copy constructor,
//              to copy-and-swap idiom, move constructor,
//              printing via overloaded operator<<, etc.
//
// by Giovanni Dicanio

#ifndef ARRAY_HPP_INCLUDED
#define ARRAY_HPP_INCLUDED

#include <cassert>    // For assert
#include <ostream>    // For std::ostream
#include <utility>    // For std::swap


class IndexOutOfBoundsException{};

//------------------------------------------------------------------------
// A generic array of T
//------------------------------------------------------------------------
template <typename T>
class Array {
```

```cpp
//------------------------------------------------------------------------
// A generic array of T
//------------------------------------------------------------------------
template <typename T>
class Array {

  //
  // Implementation Details
  //
 private:
  T*  m_ptr{nullptr};
  int m_size{0};

  bool IsValidIndex(int index) const {
    return (index >= 0) && (index < m_size);
  }


  //
  // Public Interface
  //
public:
```

Generic type T used in this implementation code


giovanni@ubuntu: ~/demo
// Constructor to create an array with the given size (element count)
explicit Array(int size) {
assert(size >= 0);
if (size != 0) {
m_ptr = new T[size]{};
m_size = size;
}
}
// Copy constructor
Array(const Array& source) {
if (! source.IsEmpty()) {
m_size = source.m_size;
m_ptr = new T[m_size]{};
for (int i = 0; i < m_size; i++) {
m_ptr[i] = source.m_ptr[i];
}
}
}
Generic type T
used in this Array<T>
implementation code
66,0-1

#Arrays