Dynamic memory and user-defined types

Michael R. Nowak
Texas A&M University

Dynamic arrays

- We will illustrate the association of dynamically allocated objects with user-defined types through the construction of a class, `DynamicIntArray`, that will encapsulate a dynamic array
- Our goal is to implement `DynamicIntArray` as a container capable of holding an arbitrary number of `int`eger values
  - As elements (i.e., `int`s) are added, we would like the array to "grow" as necessary to hold them
  - As elements (i.e., `int`s) are removed, we would like the array to "shrink" as necessary as not to waste space
- We would like to be able to:
  - declare a `DynamicIntArray dia` and
  - `push_back()` integer elements, storing each in the object's array data member (similar to what we've done with an `std::vector`

Dynamic arrays

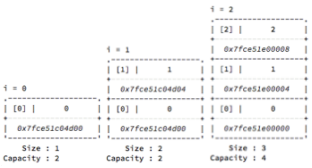
- At this point, you've hopefully recognized that we will need to maintain a pointer to a dynamically allocated array object as a private member of `DynamicIntArray`
  - As arrays do not know their own size, we will also need to maintain a private data member storing the capacity of the array
  - As well as an additional data member storing the number of elements stored in the array (we will call this the array's size)
- Moreover, we must provide some sort of interface to interact with these private data members, such that `int` objects can be added to our container
- And furthermore, private member functions that will help implement various aspects of the functionality promised by that interface

DynamicIntArray

- To illustrate what we'd like to happen, let's envision a `DynamicIntArray` that begins with the capacity of two;
- Once the number of elements (i.e., the size) reaches capacity, we'd like to resize the container's array by twice its current capacity
- For instance, we'd like

```
DynamicIntArray dia;
for (int i = 0; i < 3; ++i)
    dia.push_back(i);
```

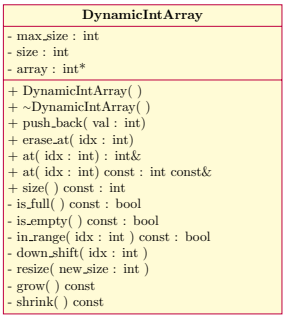
to produce:

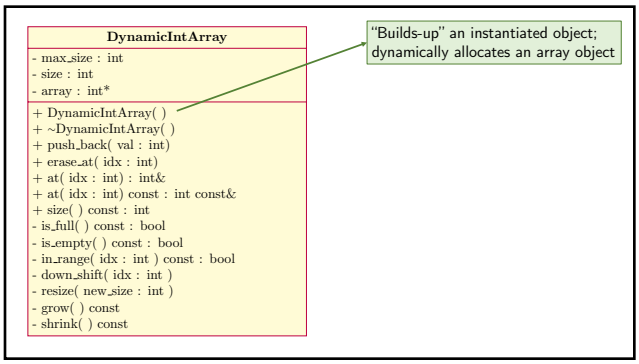


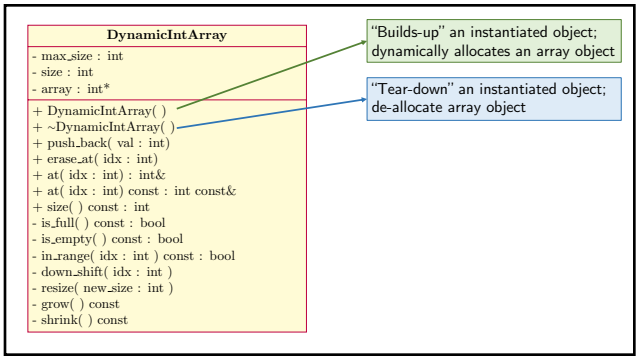
etc.

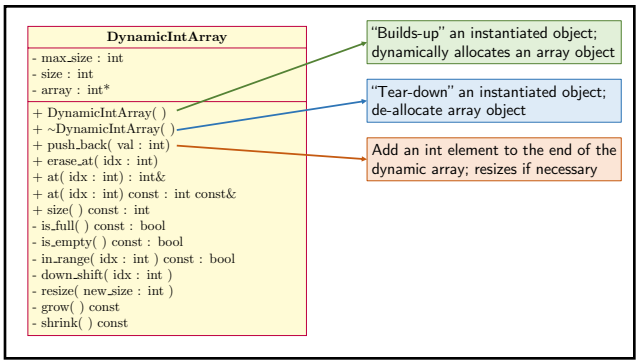
DynamicIntArray

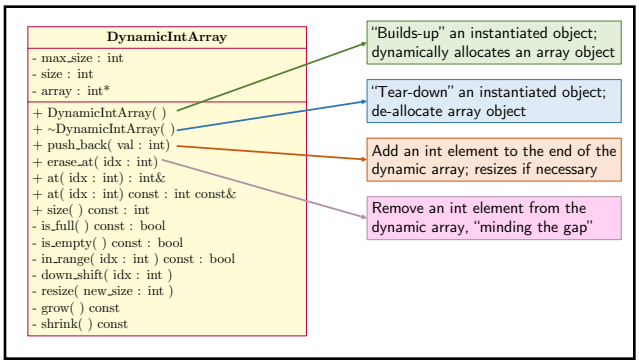
- Before we begin talking about the implementation details, we must discuss how our container should be designed
  - How should we construct our user-defined type?
    - What private data members are needed?
    - What functionality should comprise the public interface?
    - What functionality do we need, but want to keep private?
    - Do we need to provide any non-member helper functions?

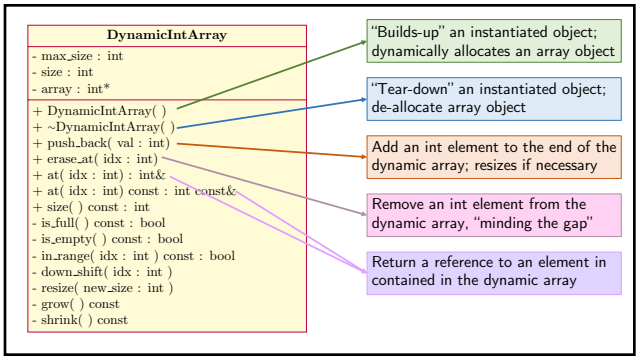


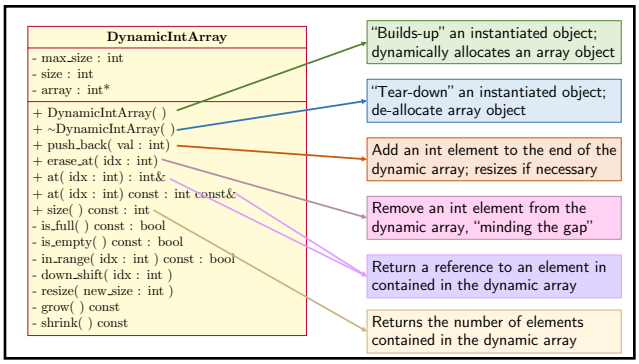

DynamicIntArray
- max_size : int
- size : int
- array : int*
+ DynamicIntArray( )
+ ~DynamicIntArray( )
+ push_back(val : int)
+ erase_at(idx : int)
+ at(idx : int) : int&
+ at(idx : int) const : int const&
+ size( ) const : int
- is_full( ) const : bool
- is_empty( ) const : bool
- in_range(idx : int) const : bool
- down_shift(idx : int)
- resize(new_size : int)
- grow( ) const
- shrink( ) const
"Builds-up" an instantiated object; dynamically allocates an array object

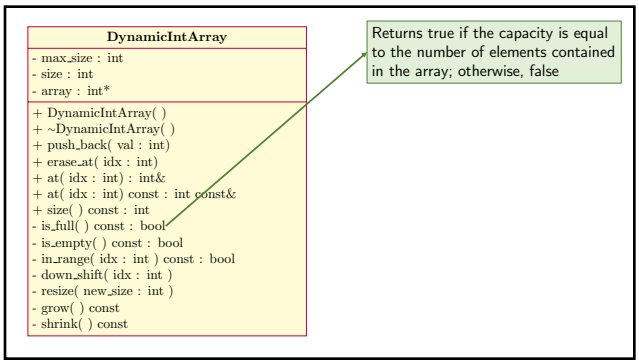

DynamicIntArray
- max_size : int
- size : int
- array : int*
+ DynamicIntArray( )
+ ~DynamicIntArray( )
+ push_back(val : int)
+ erase_at(idx : int)
+ at(idx : int) : int&
+ at(idx : int) const : int const&
+ size( ) const : int
- is_full( ) const : bool
- is_empty( ) const : bool
- in_range(idx : int) const : bool
- down_shift(idx : int)
- resize(new_size : int)
- grow( ) const
- shrink( ) const
"Builds-up" an instantiated object; dynamically allocates an array object
"Tear-down" an instantiated object; de-allocate array object

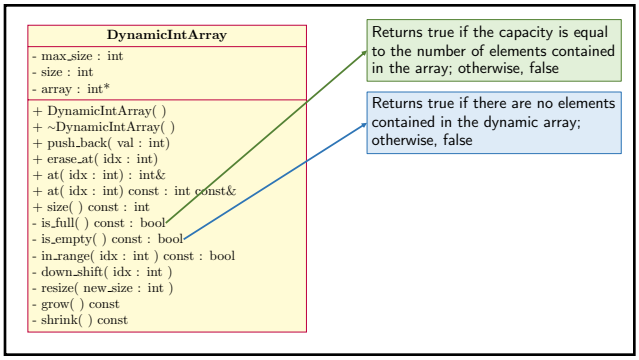

DynamicIntArray
- max_size : int
- size : int
- array : int*
+ DynamicIntArray( )
+ ~DynamicIntArray( )
+ push_back(val : int)
+ erase_at(idx : int)
+ at(idx : int) : int&
+ at(idx : int) const : int const&
+ size( ) const : int
- is_full( ) const : bool
- is_empty( ) const : bool
- in_range(idx : int) const : bool
- down_shift(idx : int)
- resize(new_size : int)
- grow( ) const
- shrink( ) const
"Builds-up" an instantiated object; dynamically allocates an array object
"Tear-down" an instantiated object; de-allocate array object
Add an int element to the end of the dynamic array; resizes if necessary

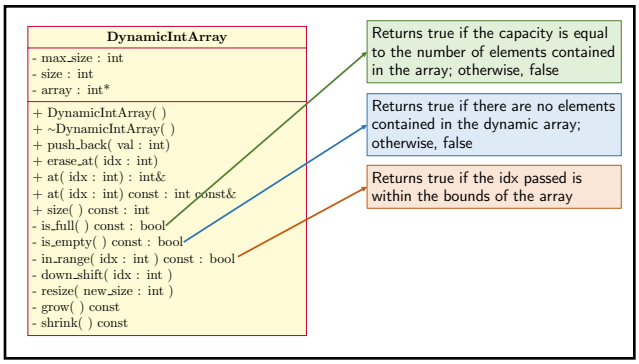

DynamicIntArray
- max_size : int
- size : int
- array : int*
+ DynamicIntArray( )
+ ~DynamicIntArray( )
+ push_back(val : int)
+ erase_at(idx : int)
+ at(idx : int) : int&
+ at(idx : int) const : int const&
+ size( ) const : int
- is_full( ) const : bool
- is_empty( ) const : bool
- in_range(idx : int) const : bool
- down_shift(idx : int)
- resize(new_size : int)
- grow( ) const
- shrink( ) const
"Builds-up" an instantiated object; dynamically allocates an array object
"Tear-down" an instantiated object; de-allocate array object
Add an int element to the end of the dynamic array; resizes if necessary
Remove an int element from the dynamic array, "minding the gap"

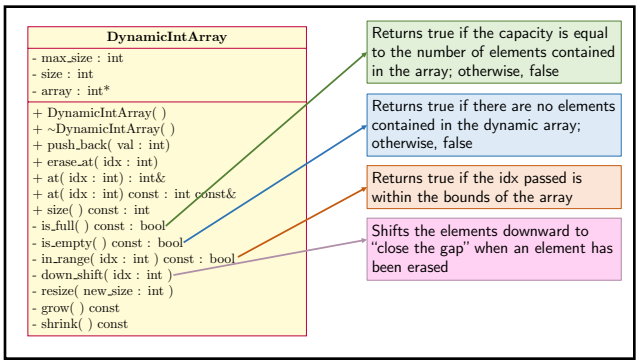

DynamicIntArray
- max_size : int
- size : int
- array : int*
+ DynamicIntArray( )
+ ~DynamicIntArray( )
+ push_back(val : int)
+ erase_at(idx : int)
+ at(idx : int) : int&
+ at(idx : int) const : int const&
+ size( ) const : int
- is_full( ) const : bool
- is_empty( ) const : bool
- in_range(idx : int) const : bool
- down_shift(idx : int)
- resize(new_size : int)
- grow( ) const
- shrink( ) const
"Builds-up" an instantiated object; dynamically allocates an array object
"Tear-down" an instantiated object; de-allocate array object
Add an int element to the end of the dynamic array; resizes if necessary
Remove an int element from the dynamic array, "minding the gap"
Return a reference to an element in contained in the dynamic array

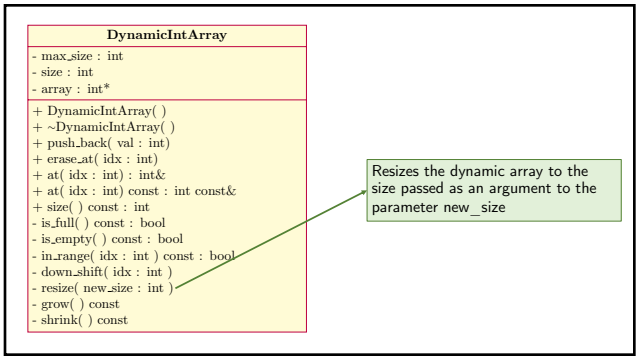

DynamicIntArray
- max_size : int
- size : int
- array : int*
+ DynamicIntArray( )
+ ~DynamicIntArray( )
+ push_back(val : int)
+ erase_at(idx : int)
+ at(idx : int) : int&
+ at(idx : int) const : int const&
+ size( ) const : int
- is_full( ) const : bool
- is_empty( ) const : bool
- in_range(idx : int) const : bool
- down_shift(idx : int)
- resize(new_size : int)
- grow( ) const
- shrink( ) const
"Builds-up" an instantiated object; dynamically allocates an array object
"Tear-down" an instantiated object; de-allocate array object
Add an int element to the end of the dynamic array; resizes if necessary
Remove an int element from the dynamic array, "minding the gap"
Return a reference to an element in contained in the dynamic array
Returns the number of elements contained in the dynamic array

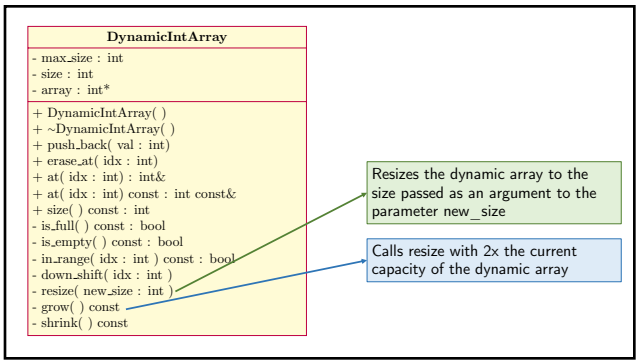

DynamicIntArray
- max_size : int
- size : int
- array : int*
+ DynamicIntArray( )
+ ~DynamicIntArray( )
+ push_back(val : int)
+ erase_at(idx : int)
+ at(idx : int) : int&
+ at(idx : int) const : int const&
+ size( ) const : int
- is_full( ) const : bool
- is_empty( ) const : bool
- in_range(idx : int) const : bool
- down_shift(idx : int)
- resize(new_size : int)
- grow( ) const
- shrink( ) const
Returns true if the capacity is equal to the number of elements contained in the array; otherwise, false

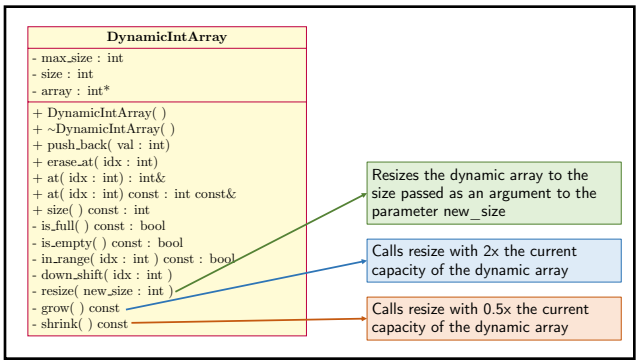

DynamicIntArray
- max_size : int
- size : int
- array : int*
+ DynamicIntArray( )
+ ~DynamicIntArray( )
+ push_back(val : int)
+ erase_at(idx : int)
+ at(idx : int) : int&
+ at(idx : int) const : int const&
+ size( ) const : int
- is_full( ) const : bool
- is_empty( ) const : bool
- in_range(idx : int) const : bool
- down_shift(idx : int)
- resize(new_size : int)
- grow( ) const
- shrink( ) const
Returns true if the capacity is equal to the number of elements contained in the array; otherwise, false
Returns true if there are no elements contained in the dynamic array; otherwise, false


DynamicIntArray
- max_size : int
- size : int
- array : int*
+ DynamicIntArray( )
+ ~DynamicIntArray( )
+ push_back(val : int)
+ erase_at(idx : int)
+ at(idx : int) : int&
+ at(idx : int) const : int const&
+ size( ) const : int
- is_full( ) const : bool
- is_empty( ) const : bool
- in_range(idx : int) const : bool
- down_shift(idx : int)
- resize(new_size : int)
- grow( ) const
- shrink( ) const
Returns true if the capacity is equal to the number of elements contained in the array; otherwise, false
Returns true if there are no elements contained in the dynamic array; otherwise, false
Returns true if the idx passed is within the bounds of the array


DynamicIntArray
- max_size : int
- size : int
- array : int*
+ DynamicIntArray( )
+ ~DynamicIntArray( )
+ push_back(val : int)
+ erase_at(idx : int)
+ at(idx : int) : int&
+ at(idx : int) const : int const&
+ size( ) const : int
- is_full( ) const : bool
- is_empty( ) const : bool
- in_range(idx : int) const : bool
- down_shift(idx : int)
- resize(new_size : int)
- grow( ) const
- shrink( ) const
Returns true if the capacity is equal to the number of elements contained in the array; otherwise, false
Returns true if there are no elements contained in the dynamic array; otherwise, false
Returns true if the idx passed is within the bounds of the array
Shifts the elements downward to "close the gap" when an element has been erased


DynamicIntArray
- max_size : int
- size : int
- array : int*
+ DynamicIntArray( )
+ ~DynamicIntArray( )
+ push_back(val : int)
+ erase_at(idx : int)
+ at(idx : int) : int&
+ at(idx : int) const : int const&
+ size( ) const : int
- is_full( ) const : bool
- is_empty( ) const : bool
- in_range(idx : int) const : bool
- down_shift(idx : int)
- resize(new_size : int)
- grow( ) const
- shrink( ) const
Resizes the dynamic array to the size passed as an argument to the parameter new_size


DynamicIntArray
- max_size : int
- size : int
- array : int*
+ DynamicIntArray( )
+ ~DynamicIntArray( )
+ push_back(val : int)
+ erase_at(idx : int)
+ at(idx : int) : int&
+ at(idx : int) const : int const&
+ size( ) const : int
- is_full( ) const : bool
- is_empty( ) const : bool
- in_range(idx : int) const : bool
- down_shift(idx : int)
- resize(new_size : int)
- grow( ) const
- shrink( ) const
Resizes the dynamic array to the size passed as an argument to the parameter new_size
Calls resize with 2x the current capacity of the dynamic array



DynamicIntArray::DynamicIntArray()

- When an object is instantiated from our `DynamicIntArray` class, the constructor is called implicitly and serves to initialize the object being constructed
  - The constructor is thus an excellent place to acquire a dynamically allocated array object from the free store
  - For the purposes of this example, we will initially acquire a dynamically allocated array of capacity two; I've declared a `static int constexpr` identified by `init_size` as a private member to store that value

```
DynamicIntArray::DynamicIntArray() :
     max_size(init_size), array(nullptr)
{
     array = new int[init_size];
}
```

DynamicIntArray::~DynamicIntArray()

- As we have seen, automatic variables deallocate their memory once as they leave the scope from which they were declared
- Furthermore, we have seen how dynamic memory for an object can be freed by by calling delete on a pointer to that object
- In both cases, a respective object's destructor is implicitly called
- The destructor is an appropriate place to deallocate the dynamic memory that we had acquired for the dynamic array

Aside: on destructors responsibility

- The *destructor* is ***responsible*** for *freeing* any *dynamic memory* that *belongs* to the *object*, ***before*** the *object's memory* is *freed*
- The destruction process proceeds by:
  1. Calling the object's destructor function
  2. Calling the destruction functions for each data member that is derived from a class
     - If a respective pointer refers to a dynamically allocated memory object, that object will remain on the free-store unless we have already deleted it by this time (e.g., in 2)
  3. Calling the destructor function of the object's base classes
     - Don't worry about this until we get to inheritance

DynamicIntArray::~DynamicIntArray()

```
DynamicIntArray::~DynamicIntArray()
{
    delete [] array;
}
```

Private member functions

- `void DynamicIntArray::resize(int new_size)`
- `void DynamicIntArray::grow()`
- `void DynamicIntArray::shrink()`
- `bool DynamicIntArray::is_full() const`
- `bool DynamicIntArray::is_empty() const`
- `bool DynamicIntArray::in_range(int idx) const`
- `void DynamicIntArray::down_shift(int idx)`

```
void DynamicIntArray::resize(int new_size)
{
    // 1. Create an entirely new array of the appropriate type and size
    int *cpy = new int[new_size];
    // 2. Copy the data from the old array into the new array
         (keeping them at the same position)
    int num_to_copy = (new_size > size) ? size : new_size;
    for (decltype(size()) i = 0 ; i < num_to_copy ; ++i)
         cpy[i] = array[i];
    // 3. Delete the old array as you don't need it anymore
    delete[] array;
    // 4a. Update the pointer to reflect the new address
    array = cpy;
    // 4b. Update the variable containing the array's max size
    max_size = new_size;
}
```

```
void DynamicIntArray::grow()
{
    int twice_max_size = max_size * 2;
    resize(twice_max_size);
}
```

```
void DynamicIntArray::shrink()
{
    int half_max_size = max_size / 2;
    resize(half_max_size);
}
```

```
bool DynamicIntArray::is_full()
const
{
    return (sz == max_size);
}
```

```
bool DynamicIntArray::is_empty()
const
{
    return (sz == 0);
}
```

```
bool DynamicIntArray::in_range(int
idx) const
{
    if (idx >= 0 && idx < max_size)
        return true;
    return false;
}
```

```
void DynamicIntArray::down_shift(int idx)
{
    // 1. Ensure that the idx passed is within the
       bounds of array
    if (!in_range(idx))
        throw std::out_of_range{""};
    // 2. Propogate the elements downward towards idx
       from the back of the array
     for (decltype(size()) i = idx; i < sz - 1; ++i)
        array[i] = array[i + 1];
}
```

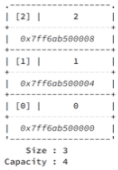
Public member functions (i.e., the interface)

- `void DynamicIntArray::push_back(int i)`
- `int& DynamicIntArray::at(int idx)`
- `int const& DynamicIntArray::at(int idx) const`
- `void DynamicIntArray::erase_at(int idx)`

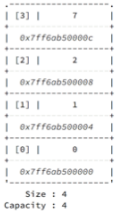
```
void DynamicIntArray::push_back(int i)
{
    // 1. Check whether the array is full; if so, grow the array
    if (is_full())
        grow();
    // 2. Assign the new element to the end of the array
    array[sz] = i;
    // 3. Increase the size by one to reflect this addition
    sz += 1;
}
```

void DynamicIntArray::push_back(int i)

Before call to .push_back(7)

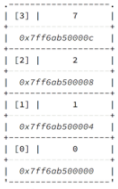


After call to .push_back(7)

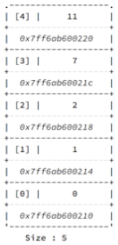


void DynamicIntArray::push_back(int i)

Before call to .push_back(11)

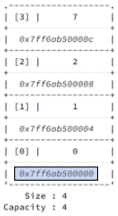


After call to .push_back(11)

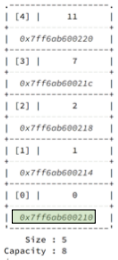


void DynamicIntArray::push_back(int i)

Before call to .push_back(11)



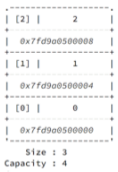
After call to .push_back(11)



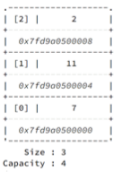
```
int& DynamicIntArray::at(int idx)
{
    // 1. Ensure that the idx passed is within the bounds of
       array
    if (!in_range(idx))
        throw std::out_of_range{""};
    // 2. Return reference to requested element
    return array[idx];
}
```

int& DynamicIntArray::at(int idx)

Before calls to: `dia.at(0) = 7;`
`dia.at(1) = 11;`



After calls to: `dia.at(0) = 7;`
`dia.at(1) = 11;`

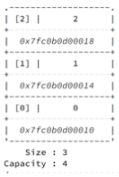



```
int const& DynamicIntArray::at(int idx) const
{
    // 1. Ensure that the idx passed is within the bounds of
       array
    if (!in_range(idx))
        throw std::out_of_range{""};
    // 2. Return reference to requested element
    return array[idx];
}
```

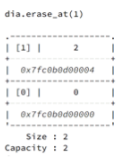
```
void DynamicIntArray::erase_at(int idx)
{
    // 1. Ensure that the idx passed is within the bounds of array
    if (!in_range(idx))
        throw std::out_of_range{""};
    // 2. Shift all of the elements down one position to "fill" the "gap"
       caused by the erase.
    down_shift(idx);
    // 3. Adjust the size to reflect the deleted element
    sz -= 1;
    // 4. If the size has decreased substantially (which would occur over
       multiple erases; not a single one), shrink the array to save space.
    if (sz == (max_size / 2))
        shrink();
}
```

void DynamicIntArray::erase_at(int idx)

Before call to .erase_at(1)



After call to .erase_at(1)



Dynamic Memory Allocation

Copy challenges

Dynamic Memory Allocation

Copy Challenge : Initialization

- In `main.cpp`, we have created an instance of `DynamicIntArray` with three integers using our push_back function:

```
DynamicIntArray dia;
for (int i = 0; i < 3; ++i) {
    dia.push_back(i);
}
```

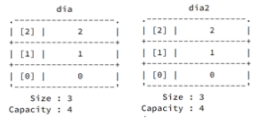
- We then decided to initialize a new instance of `DynamicIntArray` with `dia`

```
DynamicIntArray dia2 {dia};
```

Dynamic Memory Allocation

Copy Challenge : Initialization

- We then print the contents of each `DynamicIntArray`
- We observe that each object contains the same elements
- Interestingly, we note an interesting error message printed during the execution of dia2's destructor:

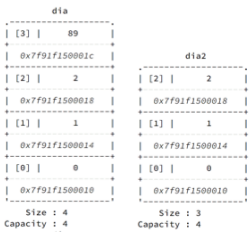



```
a.out(34987,0x7fff75a4d300) malloc: *** error for object
0x7f82a0500000: pointer being freed was not allocated
*** set a breakpoint in malloc_error_break to debug
[1]    34987 abort      ./a.out
```

Dynamic Memory Allocation

Copy Challenge : Initialization

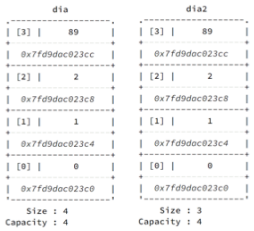
- Ignoring the error message (which is never a good idea)
- We decide to push_back the integer 89 to dia
- We print the contents of both `DynamicIntArray`s again
- Interestingly, we find that the elements in each object have the same addresses...
  - Why is this the case?



Dynamic Memory Allocation

Copy Challenge : Initialization

- Given that the first couple of elements have the same addresses, I decided to print the `DynamicIntArray`s based on their capacity, not size
- Hmm... what's going on?
  - Why are the sizes of `dia` and `dia2` different?
  - Why do the individual elements of `dia` and `dia2` have the same addresses?



Dynamic Memory Allocation

Copy Challenge : Shallow Copy

- The default copy mechanism for an object of a user-defined type class using an instance of the same type makes what is known as a ***shallow copy***
- A ***shallow copy*** means that the object is copied exactly as it is
  - Each data member is copied exactly to the corresponding member data location in the new object

Dynamic Memory Allocation

Copy Challenge : Shallow Copy

- When we wrote `DynamicIntArray dia2 {dia}` to initialize a new `DynamicIntArray` object, a shallow copy of `dia` was performed of the data members:

DynamicIntArray
- max_size : int - size : int - array : int*

- This means that the values of `dia` are `dia2` used to initialize the values of `dia2`
  - The value stored in `dia`'s max_size is used to initialize `dia2`'s
  - The value stored in `dia`'s size is used to initialize `dia2`'s
  - The value stored in `dia`'s array is used to initialize `dia2`'s

Dynamic Memory Allocation

Copy Challenge : Shallow Copy

- When we wrote `DynamicIntArray dia2 {dia}` to initialize a new `DynamicIntArray` object, a shallow copy of `dia` was performed of the data members:

DynamicIntArray
- max_size : int - size : int - array : int*

  - This means that the values of `dia` are `dia2` used to initialize the values of `dia2`
    - The value stored in `dia`'s max_size is used to initialize `dia2`'s
    - The value stored in `dia`'s size is used to initialize `dia2`'s
    - **The value stored in `dia`'s array is used to initialize `dia2`'s**

Dynamic Memory Allocation

Copy Challenge : Shallow Copy

- When there is a pointer inside an object that points to dynamic data, a shallow copy of that object is not sufficient because
  - Only the values from pointers are copied into the other object's corresponding pointers, but not the dynamic data that is pointed to
  - Consequentially, the copy will be pointing to the original dynamic data
  - This results in the dynamic data being shared between the objects

Dynamic Memory Allocation

Copy Challenge Resolution : Deep Copy

- In a deep copy, the objects pointed to in one object are first copied
- Then the other object's corresponding pointers are assigned the addresses of the copied objects
- After the deep copy has been completed, each identifier
  - Has the same values
  - Stored in different objects in memory

Dynamic Memory Allocation

Copy Challenge Resolution : Initialization

- In order have a deep copy performed during the initialization of an object of `DynamicIntArray` with an object of its own type, e.g.,

```
DynamicIntArray dia2 {dia};
DynamicIntArray dia2 = dia;
```

- We will need to:
  - Declare a copy constructor that takes an object of the same type as its formal argument
  - Define that constructor such that a deep copy of the passed object is used for initialization

Dynamic Memory Allocation

Copy Constructor : `DynamicIntArray`

- We would like to write a copy constructor for our `DynamicIntArray` class that performs a deep copy opposed to the shallow copy that is provided to user-defined classes by default
- The declaration of the copy constructor is written in the public interface of our `DynamicIntArray` as

```
DynamicIntArray(const DynamicIntArray&);
```

Dynamic Memory Allocation

Copy Constructor : `DynamicIntArray`

- The definition of the copy constructor is then defined as follows:

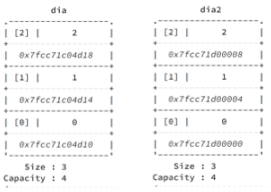
```
DynamicIntArray::DynamicIntArray(const DynamicIntArray& source) :
    max_size(source.capacity()),
    sz(source.size()),
    array(nullptr)
{
    // (1) allocate new memory: should always be done in function body
    array = new int[source.capacity()];

    // (2) copy data from source to new memory
    for (decltype(source.size()) i = 0; i < source.size(); ++i) {
        array[i] = source.at(i);
    }
}
```

Dynamic Memory Allocation

Copy Constructor : DynamicIntArray

- After defining the copy constructor for DynamicIntArray to perform a deep copy, when a new DynamicIntArray object is initialized with another DynamicIntArray object, each object will have its own copy of the data:



Dynamic Memory Allocation

Copy Challenge : Assignment

- In main.cpp, we have created an instance of DynamicIntArray with three integers using our push_back function:

```
DynamicIntArray dia;
for (int i = 0; i < 3; ++i) {
    dia.push_back(i);
}
```

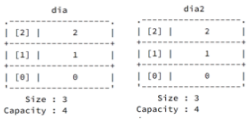
- We then decided to default initialize a new instance of DynamicIntArray and then assign it dia

```
DynamicIntArray dia2;
dia2 = dia;
```

Dynamic Memory Allocation

Copy Challenge : Assignment

- We then print the contents of each DynamicIntArray
- We observe that each object contains the same elements
- Interestingly, we note an interesting error message printed during the execution of dia2's destructor:

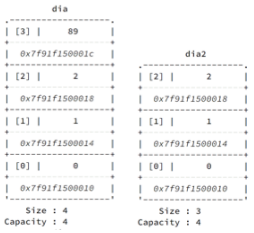



```
a.out(34987,0x7fff75a4d300) malloc: *** error for object
0x7f82a0500000: pointer being freed was not allocated
*** set a breakpoint in malloc_error_break to debug
[1]    34987 abort      ./a.out
```

Dynamic Memory Allocation

Copy Challenge : Assignment

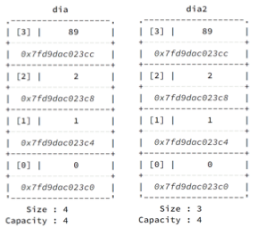
- Ignoring the error message (which is never a good idea)
- We decide to push_back the integer 89 to dia
- We print the contents of both `DynamicIntArray`s again
- Interestingly, we find that the first couple of elements have the same addresses...
  - Why is this the case?



Dynamic Memory Allocation

Copy Challenge : Assignment

- Given that the first couple of elements have the same addresses, I decided to print the `DynamicIntArray`s based on their capacity, not size
- Hmm... what's going on?
  - Why are the sizes of `dia` and `dia2` different?
  - Why do the individual elements of `dia` and `dia2` have the same addresses?



Dynamic Memory Allocation

Copy Challenge Resolution : Assignment

- In order have a deep copy performed during the initialization of an object of `DynamicIntArray` with an object of its own type, e.g.,

```
DynamicIntArray dia2;
dia2 = dia;
```

- We will need to:
  - Declare an overloaded assignment '=' operator that takes an object of the same type as its formal argument
  - Define that overloaded assignment '=' operator such that a deep copy of the passed object is used for assignment

Dynamic Memory Allocation

Copy assignment operator: DynamicIntArray

- We would like to write an overloaded assignment '=' operator for our DynamicIntArray class that performs a deep copy opposed to the shallow copy that is provided to user-defined classes by default
- The declaration of the overloaded assignment '=' operator is written in the public interface of our DynamicIntArray as

```
DynamicIntArray& operator=(const DynamicIntArray&);
```

Dynamic Memory Allocation

Copy assignment operator: DynamicIntArray

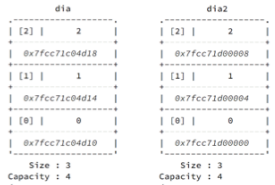
- The definition of the overloaded assignment '=' operator is then defined as follows:

```
DynamicIntArray& DynamicIntArray::operator=(const DynamicIntArray& source)
{
    // (1) Check for self-assignment
    if (this != &source) {
        // (2) Delete old data
        delete [] array;
        // (3) Allocate new memory
        array = new int[source.capacity()];
        // (4) Copy data
        sz = source.size();
        max_size = source.capacity();
        for (decltype(source.size()) i = 0; i < source.size(); ++i) {
            array[i] = source.at(i);
        }
    }
    return *this;
}
```

Dynamic Memory Allocation

Copy Constructor : DynamicIntArray

- After overloading the copy assignment operator for DynamicIntArray to perform a deep copy, when a new DynamicIntArray object assigned an DynamicIntArray object, each object will have its own copy of the data:



Copy Assignment Operator vs Copy Constructor

Copy Assignment Operator	**Copy Constructor**
• Return: reference to self	• Return: None (all constructors)
• Function name: **operator=**	• Function name: ***ClassName***
• Parameter: **reference to const source object** of same type	• Parameter: **reference to const source object** of same type
1. Check for self-assignment	
2. Delete old data	
3. Allocate new memory	1. Allocate new memory
4. Copy data from source	2. Copy data from source