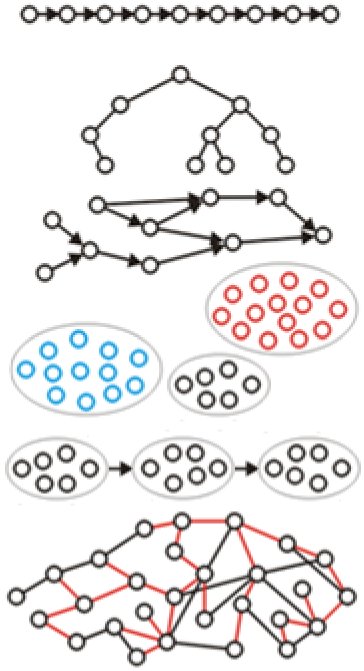
Sequential Containers



Carlos Moreno

cmoreno@uwaterloo.ca

EIT-4103

https://ece.uwaterloo.ca/~cmoreno/ece250

Sequential Containers

Standard reminder to set phones to silent/vibrate mode, please!

Sequential Containers

- Today's class:
  - We'll introduce sequential containers, implementation strategies, and related operations.
  - We'll investigate the two main strategies for these:
    - Contiguous (block) storage — today (typical example: arrays)
    - Node-based storage — next class (typical example: linked lists)
  - We'll also look into stacks and queues (useful for lab 1 next week!)

Sequential Containers

- Sequential containers are typically used for linearly ordered data — makes sense; they provide, precisely, a linear arrangement of the elements.
- They may also be useful for weakly ordered data, depending on what the constraints for equivalent elements may be (i.e., it could be a little more complicated than simply using a sequential container).

Sequential Containers

- Also, they may be useful (again, depending on the constraints in the particular situation) for data with no ordering/relations at all — we introduce the “artificial” linear ordering given by the order in which the elements appear in the sequential storage.

Sequential Containers

- Also, they may be useful (again, depending on the constraints in the particular situation) for data with no ordering/relations at all — we introduce the “artificial” linear ordering given by the order in which the elements appear in the sequential storage.
- Notice that this makes sense given that this is the simplest type of data structure, so it seems reasonable to use it with the data does not impose any requirements.

Sequential Containers

- Two main categories for the storage strategy:
  - Contiguous storage (block based)
  - Node (element) based
- Typical (simple) examples being arrays for contiguous storage, and linked lists for node based storage.
  - We will, of course, see many other examples for each of these categories, and several examples where either strategy may be used (and useful).

Sequential Containers

- For contiguous storage (at least in C and C++), we have two strategies:
  - The simple (boring? :-)) one — built-in arrays
  - Dynamic arrays (where the real fun is!)

Sequential Containers

- In this course, we won't care too much (or at all) about built-in arrays — they're just the simple (and limited) way to implement the simplest possible types of data.

Sequential Containers

- In this course, we won't care too much (or at all) about built-in arrays — they're just the simple (and limited) way to implement the simplest possible types of data.

(clipart courtesy of clipartof.com)

Sequential Containers

- For dynamic storage, we saw already a couple of fundamental concepts in C++ that play a role:
  - Pointers (and in particular, the close relationship between pointers and arrays — not the same thing, though!!)
  - Dynamic memory allocation

Sequential Containers

- There is a third notion that we haven't explicitly seen so far (but you did see it indirectly during Lab 0):
  - Handling dynamic memory allocations through classes.
  - The fancy name for this in the C++ community is: RAII (Resource Acquisition Is Initialization)
  - The idea being that initializing an object should handle the acquisition of resources, and destroying it should handle the release of such resources.

Sequential Containers

- RAII applies in general to all sorts of resources, but we'll focus on dynamically allocated memory as the only resource that we'll be dealing with.

Sequential Containers

- Bottom line (extremely simplified):
  - Constructor allocates memory (new or new [ ])
  - Destructor releases memory (delete or delete[])

Sequential Containers

- For a more complete picture, we need to take into account:
  - Copy constructor (creating an object as a copy of another — if we simply copy the values of the pointers, we're in trouble; two objects pointing to the same memory; one object will affect the other; there will be double delete — two destructors operating on the same memory)
  - Assignment operator — assigning one object from another (need to release and reallocate to copy the data from the other object)

Sequential Containers

- More on these subtleties in the Lab material...

Sequential Containers

- Using a block of dynamically allocated memory as an array (e.g., an array of ints) is straight-forward:

```
int * array = new int[size];
// Now use array[index]
// Example to fill the array with -1's:

for (int i = 0; i < size; i++)
{
    array[i] = -1;
}
```

Sequential Containers

- Most likely, we'll want to “templatize” that code; e.g., inside some template code parameterized by typename **Type**:

```
Type * array = new Type[size];
// An important difference is that in this case,
// all elements are default-initialized  (the
// default constructor is called for each of the
// size objects)

for (int i = 0; i < size; i++)
{
    array[i] = ?? ;  // What do we put here??
}
```

Sequential Containers

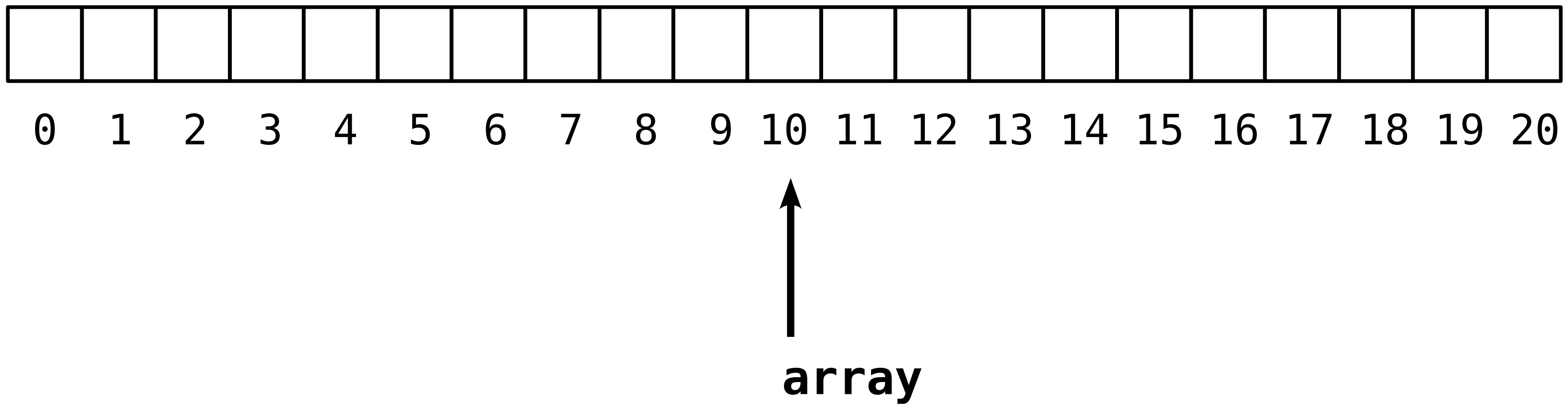
- Some interesting things that can be done with an array:
  - Signed subscripts (as in, both positive and negative subscripts)

Sequential Containers

- This is somewhat straightforward, given the close relationship between arrays and pointers:
  - Given a block of memory of, say, 21 elements, we set up a pointer pointing to half that space (in this example, pointing to the 11th element — element at index 10, when we count starting at index 0)

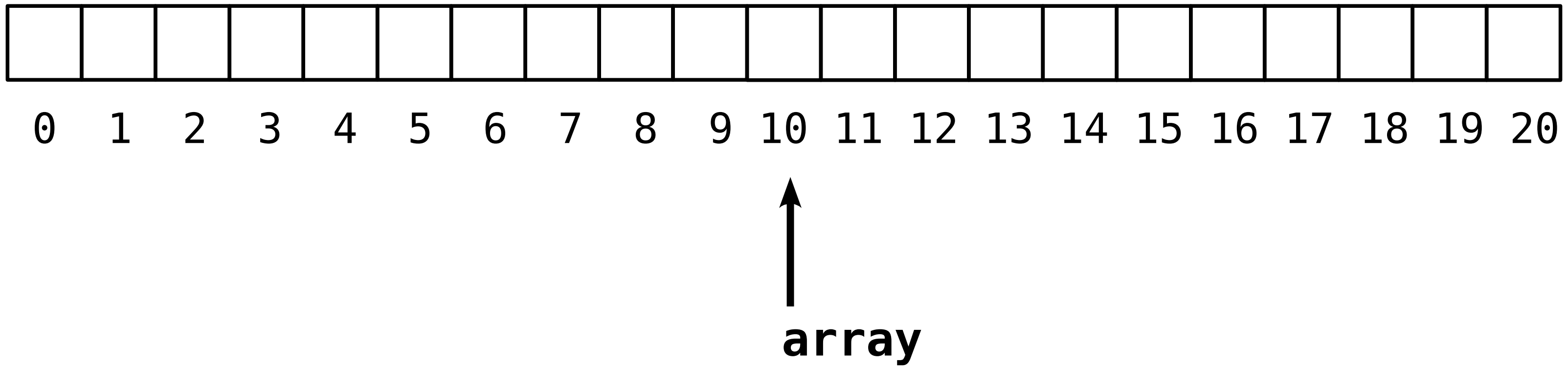
Sequential Containers

- This is somewhat straightforward, given the close relationship between arrays and pointers:
  - Given a block of memory of, say, 21 elements, we set up a pointer pointing to half that space (in this example, pointing to the 11th element — element at index 10, when we count starting at index 0)



Sequential Containers

- Given this configuration, what is array[0] ?? What is array[4] ?? What is array[−2] ??



- The trick is that array[index] is always done by going index positions after where array is pointing (that is, index is “added” to the pointer)

Sequential Containers

- Some interesting things that can be done with an array:
  - Circular array, or circular buffer.

Sequential Containers

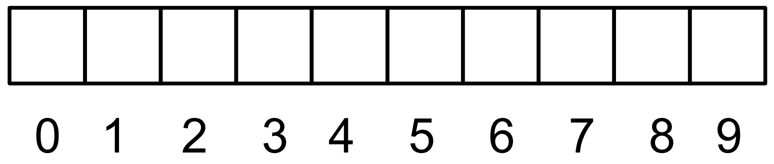
- Motivating example: Say that you want to keep track of the average temperature of the last week (as in, the most recent past 7 days — as opposed to Monday to Sunday last week or similar)
  - We can set up an array of 7 elements, but then, at each step (representing each day), we need to add one more temperature, and we run out of space.
  - Of course, now we don't need the temperature from what just became “8 days ago”, so we discard it

Sequential Containers

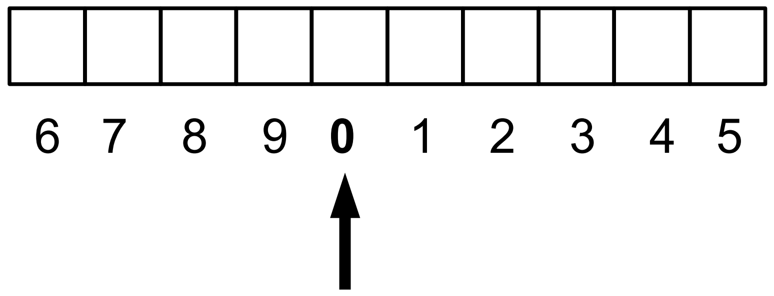
- Except that ... Oops, we need to then move 6 elements backwards (overwriting the first one), to then write the most recent value, and then compute average temperature.
- Instead, we could work with the array in a circular way, allowing the start to be at any position:
  - The positions of the other elements go sequentially from the starting position, cycling back to position 0 when reaching the highest index.

Sequential Containers

- Example for a circular buffer of size 10:
  - Normally, positions 0 to 9 will be like this:

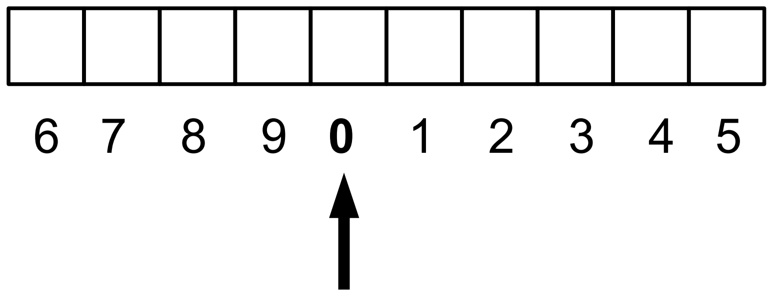


  - But might as well be something like this:



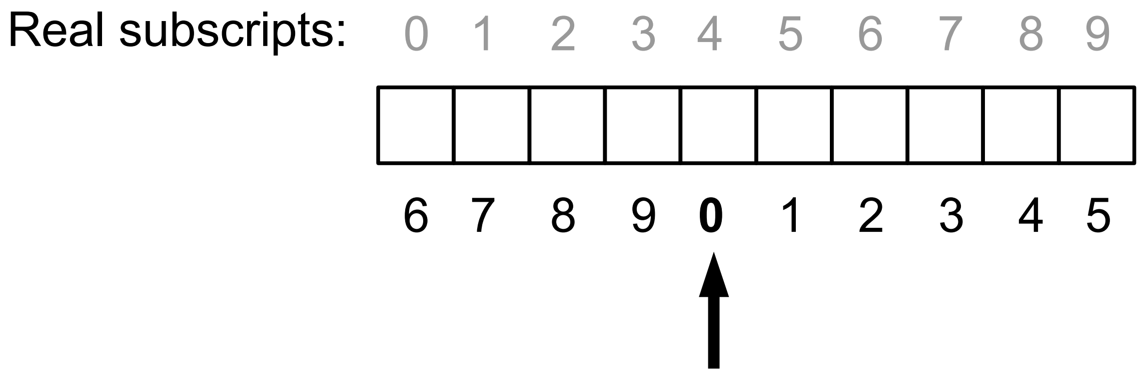
Sequential Containers

- The idea is similar to the way we handled arrays accepting negative subscripts — the difference being, the elements on the left are not really negative subscripts, but the remaining subscripts after the right-most subscript (5 in this example) is exceeded.



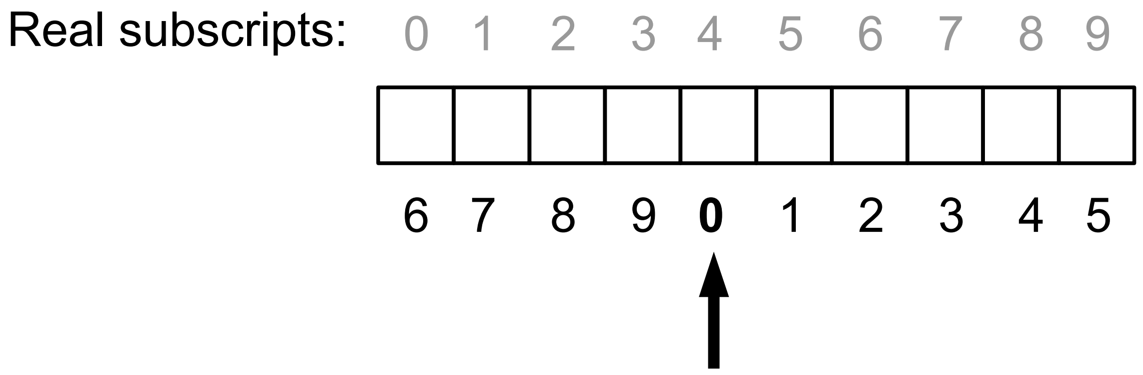
Sequential Containers

- Another difference is that here, we do not keep a pointer to the "start" of the array; instead, we keep the "real subscript" of the element that will be the "logical" 0 subscript.
- In the example below, start = 4.



Sequential Containers

- How do we obtain elements 3, or 4??
  (this one is easy — just look at the diagram below)
- How do we obtain, say, element 7, or 8??



Sequential Containers

- Dynamic (resizable) arrays:
  - We've seen examples of resizing arrays
    - With functions that resize an array given as parameter, there was the detail that the pointer needs to be passed by reference (since *the pointer* itself needs to be modified — and not simply the values pointed at)
    - If the array is encapsulated in a class, then the pointer would be a data member, and this detail would not apply: we simply have a resize() method that has direct access to the pointer, since it is a data member.

Sequential Containers

- We often want the array to automatically grow as needed.
- For example, a push_back() function (or method) would add one element at the end of the array, including the requirement that if we run out of space in the array, then we resize to increase the size and accommodate the added element.

Sequential Containers

- Question: what is the cost (in terms of the run time of the operation) of push_back() when a reallocation is needed?

Sequential Containers

- Question: what is the cost (in terms of the run time of the operation) of push_back() when a reallocation is needed?

- Follow-up question: what happens if we resize only to accommodate the added element (i.e., if we resize to size+1) ?
  - In particular, what is the cost of adding n elements?

Sequential Containers

- Hmm... Let's think about this:
  - Maybe we can do things such that, *on average*, the run time is low — all we need to do is distribute the cost of appending (which is necessarily $\Omega(n)$) among several appends.
  - That is, if we resize to more than size+1, then this append had cost Θ(size), but then the next several apends will have cost $\Theta(1)$ since they won't require resizing.

Sequential Containers

- How do we make this work? If we always resize to, say, the current size + 10, or + 20, then, as the size grows, then the advantage becomes negligible and on average each append still takes linear time.

Sequential Containers

- How do we make this work? If we always resize to, say, the current size + 10, or + 20, then, as the size grows, then the advantage becomes negligible and on average each append still takes linear time.

- Any ideas?

Sequential Containers

- How do we make this work? If we always resize to, say, the current size + 10, or + 20, then, as the size grows, then the advantage becomes negligible and on average each append still takes linear time.

- Any ideas?

  (we'll definitely discuss one in class!)

Sequential Containers

- Next, we'll look at stacks and queues.

Sequential Containers

- Next, we'll look at stacks and queues.
- These are still sequential containers, but with additional constraints in the temporal patterns of access.
- In particular, the order in which elements are inserted and removed.

Sequential Containers

- A queue represents the first-in-first-out pattern (useful for example to implement “first-come, first-served” policies).
- You could implement it as a regular sequential container (either an array or a linked list) where you insert elements on one end, and extract them from the other end (they come out in the same order as they went in)

Sequential Containers

- A queue represents the first-in-first-out pattern (useful for example to implement “first-come, first-served” policies).
- You could implement it as a regular sequential container (either an array or a linked list) where you insert elements on one end, and extract them from the other end (they come out in the same order as they went in)
  - With a linked list, this is trivial... But how would you accomplish this with an array? (would we want to?)

Sequential Containers

- Operations on a queue:
  - enqueue (insert an element to the queue)
  - dequeue (extract an element — the “next one in line”)

Sequential Containers

- A stack represents the last-in-first-out pattern.
- Literally, like a stack of books or any objects: you stack things up, and the first one that you stacked will be buried under the rest, so it would be the last one the come out.
- The operations on the stack are:
  - push (insert an element at the top of the stack)
  - pop (remove an element off the top of the stack)

Sequential Containers

- Implementation of a stack in terms of any sequential container (array or linked list) is also straightforward:
  - The difference is, you do both insertions and removals on the same end of the sequence.
  - For example, with a linked list, you either do push_front and pop_front, or push_back and pop_back — should be easy to visualize that the order of extractions will be the reverse of the order of insertions!