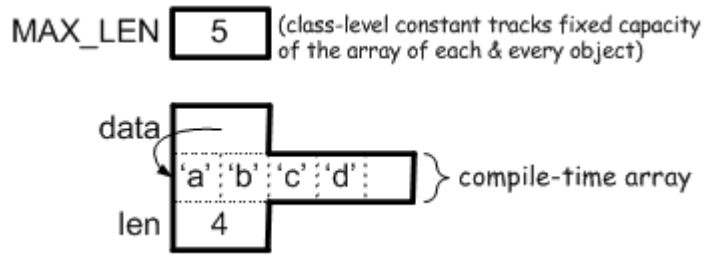
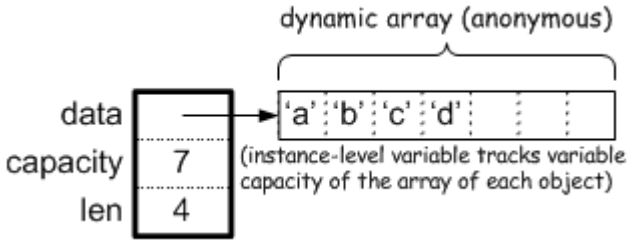
- Other related items:
  - More Involved `class` Supplement 00
  - More Involved `class` Supplement 01
  - Logical Memory Pictures - Deep Copying and Resizing
  - Object-based Data Type Development Using C++: Some Logical Associations
  - Abstraction vis-à-vis Problem Solving
  - CS3358 View of C++ Support for Problem Solving
- Ending portion of Review/Augment by Example 1 repeated:
  - To next get into `ourStr` version 2 -> use *runtime/resizable* (instead of *compile-time/fixed-sized*) array to make data type more flexible (string length not limited to certain pre-determined fixed size).

  Conceptual ideas (behind "how-to-make-more-flexible"):
  - Comparing *logical memory picture*s of an `ourStr`-*version-1* object and an `ourStr`-*version-2* object:



Bulk of storage:
-> "internal" to object (*indigenous*)
-> fixed-sized
-> *same constant capacity* for *all objects* at *all times*
(`data` is a pointer *constant*)
-> allocation/deallocation done by system
-> analogy: TxState warehouse built *on*-campus



Bulk of storage:
-> "external" to object (*exogenous*)
-> resizable
-> capacities *vary* with *objects* and *time*
(`data` is a pointer *variable*)
-> allocation/deallocation done by programmer
-> analogy: TxState warehouse built *off*-campus

- Changes made in representation (in `ourStr.h`):
  - Comparing *instance-level* variables:

```
char data[MAX_LEN];
int len;
```

```
char* data;
int capacity;
int len;
```

  - Comparing *class-level* constant:

```
static const int MAX_LEN = 5;
```

Represents fixed capacity of compile-time array.

```
static const int DEF_CAP = 1;
```

Represents *default initial* size of dynamic array.

- Change 1 made in operation – `void resize(int newSize)` member function (in `ourStr.h` and `ourStr.cpp`):
  - For use by developer (when implementing other member functions), not for use by end-user.
    - A *utility* (or *helper* or *facilitator*) function for the `class`.
    - Included as *private* (not *public*) member function (in `ourStr.h`).
    - Documentation included in implementation file (`ourStr.cpp`), not in header file (`ourStr.h`).
  - For changing the size (capacity) of invoking object's dynamic array.
    - Invoking object should still represent the *exact same data* after resizing.

- Resizing algorithm:
  - Conceptual formulation using associated logical memory picture (picture at the bottom of the third handout above).
  - Corresponding code (in `ourStr.cpp`) is nothing more than an expression of the conceptual formulation in C++.
- *Caller*'s responsibility to provide the desired *new capacity* when calling.
  - When new capacity must be higher, typically want to choose a new capacity that is *some percentage higher* and *at least 1 higher* than the old capacity.
  - As exemplified by the following statement in the implementation of `setChar`:

    ```
    resize( int(1.5*capacity) + 1 );
    ```

    "`+ 1`" is a simple way to ensure that "new" capacity specified is *at least 1 higher than* the "old" capacity.
  - Some past students always used a "new" capacity that is 1 greater than the "old" capacity, *i.e.*, by calling the function as follows:

    ```
    resize(capacity + 1);
    ```

    This can cause the program to take a *very long time* to run to completion when resizing (which is *expensive*) has to be done *many many times* on a *large* array.
- Implementation should check that caller-provided *new capacity* is valid and make appropriate adjustments where necessary.
  - New capacity must enable the invoking object to still represent the *exact same data*.
  - New capacity must be 1 or greater.

  DON'T want to use `assert` here, but make appropriate adjustments where necessary.

- Change 2 made in operation – `ourStr(int init_cap = DEF_CAP)` member function (in `ourStr.h` and `ourStr.cpp`):
  - Can take on 2 roles -> collapsing 2 constructors into one by taking advantage of C++'s *function with default arguments* feature:
    - Default constructor.
      - Client does not supply a value for `init_cap` (e.g.: `ourStr s1;`).
      - `init_cap` will have the value of `DEF_CAP`.
    - One-argument constructor.
      - Client does supply a value for `init_cap` (e.g.: `ourStr s1(25);`).
      - `init_cap` will have the client-supplied value.
  - NOTES:
    - C++ does not allow a default value (`DEF_CAP` for `init_cap` in our case) to be specified in both the *function prototype* (in the header file) and the *function definition* (in the implementation file).
      - We typically want to specify default values in the interface (header file) to make them visible to the client.
    - In the implementation, the client-supplied value (that `init_cap` gets) should be checked for validity (*i.e.*, it must be >= 1) because we don't want to dynamically allocate an array of invalid size (0 or negative).
    - The client-supplied value should be *adjusted where appropriate* (*i.e.*, if it's invalid), by setting it to `DEF_CAP` or 1, for instance.
      - As in `resize`, DON'T want to use `assert` in this case (more generally, in constructors and also in destructor).
    - In general, a constructor with *1 or more arguments* but have a *default value specified for each and every argument* will cover the default constructor.

- In view of this and if our class has such a constructor, we DON'T want to separately write a default constructor (because that will create an ambiguous situation for the compiler and cause an error during compilation).

- Change 3 made in operation – `~ourStr()` member function (in `ourStr.h` and `ourStr.cpp`):
  - Destructor:
    - Automatically called, no return type (not even `void`), same name as the class preceded with ~, parameterless, cannot be overloaded.
    - Does all necessary "cleaning-up" chores for an object of the class before the object goes out of scope (*i.e.*, before the object's lifetime ends).
    - In general, the "cleaning-up" chores are related to freeing up *non-automatically managed* resources (*i.e.*, those not managed by the system) associated with the object.
      - In this course, the *non-automatically managed* resources are almost invariably *dynamic memory* associated with the object.

- Change 4 made in operation – `ourStr(const ourStr& src)` member function (in `ourStr.h` and `ourStr.cpp`):
  - Copy constructor:
    - Automatically called, no return type (not even `void`), same name as the class, 1 single parameter *of the type the class implements*.
    - Situation 1 for its call:
      - Creating a new object and initializing it to an existing object in a *declaration* statement.
      - Two syntactic alternatives (but the *same situation*) through examples (assuming `s1` is an existing `ourStr` object at the point of the declaration statement):

        ```
        » ourStr s2 = s1;
        » ourStr s2(s1);
        ```

        NOTE:

        If we instead write the following two statements (*i.e.*, a *declaration* statement followed by an *executable* statement):

        ```
        ourStr s2;
        s2 = s1;
        ```

        the *default constructor* will be called in the first statement and the *assignment operator* (`operator=`) will be called in the second statement. The copy constructor will *not* be involved. It is obviously more expensive to do so (calling the default constructor followed by the assignment operator instead of calling only the copy constructor).
    - Situation 2 for its call:
      - Passing an object of the class *by value* to a function.

        NOTE:

        Because of this, the parameter in the copy constructor itself *cannot be passed by value*; doing so will cause an error because it will lead to an endless series of calls to the copy constructor itself.

        The parameter should be *passed by* `const` *reference* instead; although passing the parameter *by reference* won't cause a similar problem (as passing the parameter by value), doing so violates the *principle of least privilege*.
    - Situation 3 for its call:
      - Returning an object of the class *by value* from a function.
    - What we want to do (*deep* copying) vs what automatic (compiler-supplied) version will inadequately do (*shallow* copying):
      - Associated logical memory picture (picture at the top of the third handout above).

- Change 5 made in operation – `ourStr& operator=(const ourStr& rhs)` member function (in `ourStr.h` and `ourStr.cpp`):
  - Overloaded (copy) assignment operator:
    - In comparison to copy constructor (similarities and differences):
      - Both perform copying (cloning) but one or the other is called depending on which one of two copying situations applies:
        - » Copy constructor to construct a *new* object as a clone of an *existing* object.
        - » Assignment operator to change an *existing* object so that it becomes a clone of another *existing* object.
      - *Self-assignment* ("aliasing") is possible in copy assignment (but not in copy construction) and good to always trap it:
        ```
        if (this != &rhs) // if not self-assignment
        {
           ... // code to do deep-copying
        }
        ```
      - There's dynamic memory associated with the cloning object in copy assignment but not in copy construction:
        - » Not freeing up dynamic memory associated with the cloning object (in *copy assignment* and when such memory must be replaced) causes *memory leak*.
        - » Attempting to free up dynamic memory not associated with the cloning object (in *copy construction*) can lead to *fatal error at runtime*.
    - In comparison to overloading the == operator (discussed earlier when introducing operator overloading):
      - Function's name is `operator=` (as opposed to and not to be confused with `operator==` seen earlier)
      - C++ allows the assignment operator to be overloaded only with a *member function* (not a non-member function).
    - Typically want to have the function return a *reference to an object of the class*:
      - To enable the operator to be used in *chained fashion* (*e.g.*: `s3 = s2 = s1;`).
      - In the implementation (in `ourStr.cpp`), notice that the reference returned (`return *this;`) is actually a *reference to the invoking object*.
        - » (this is so because the associativity for the assignment operator is *right-to-left*)
- Other changes made in operation – all pertinent member functions (in `ourStr.h` and `ourStr.cpp`):
  - A tell tale to help in locating some (if not all) of these:
    - Those that have class-level constant (that is meant for tracking the capacity of compile-time fixed-sized array, which is no longer relevant) appearing in them.
    - Such appearances can occur in the interface and/or the implementation, including documentation (such as preconditions and postconditions).
  - Documentation changes – for pertinent functions where appropriate (especially in preconditions and/or postconditions)
  - Implementation changes – for pertinent functions where appropriate (especially *mutators* that must be modified to incorporate *resizability* in storage capacity).
- ***Assignment 2*** tip – putting the concepts/techniques into action:
  - Implementations for `resize`, "default-cum-one-argument-" constructor, destructor, copy constructor and assignment operator in `ourStr.cpp` should be helpful.
  - Simply make all necessary adaptations.
