- `ourStr` Version 1 Build A:
  - Interface – `ourStr.h`:
    - Typical entry (see **304s02SepCompFileOrgInCpp**) → documentation (for *client*-should-also-know items only).
      - Predominantly → *preconditions* and *postconditions* for each meant-for-use-by-*client* function.
        - In regard to *contract-oriented design* → user-developer (*client*-supplier) contract.
          - (recall) 3 key ingredients: *preconditions*, *postconditions*, class invariant.
          - (recall) 3 key action verbs: *expect*, *guarantee*, maintain.
            - Which to expect and which to guarantee depends on which standpoint (user or developer).
        - Preconditions *may or may not* be applicable; postconditions *always* applicable.
        - In reference to ***Assignment 1***:
          - Want to carefully read/understand the preconditions (if any) and postconditions given for each function → they tell what the function is meant to do.
    - Typical entry (see **304s02SepCompFileOrgInCpp**) → *macro guard* or *inclusion guard*.
      - ```
        #ifndef <named_constant>
        #define <named_constant>
        ... // header file proper
        #endif
        ```
        - `<named_constant>` → conventionally based on associated header file name.
          - Raising case, and inserting or replacing with underscore (`_`); in our *e.g.*: `ourStr.h` → `OUR_STR_H`.
      - Guards against *multiple inclusions* of (the contents of this header) file in another (header or source) file.
        - Multiple inclusions → typically lead to (re-definition) compilation error.
    - Typical entry (see **304s02SepCompFileOrgInCpp**) → (module-specific) *component layouts*.
      - In our *e.g.* at this point (Build A) → data type (`ourStr`) declaration using C++ class.
        - `{` and `}` (part of class syntax) → delimits *class scope*.
          - Note that the terminating `;` is required (part of syntax) for C++. (*Aside*: not so for Java).
        - Class members → names introduced in *class scope* (within `{` and `}`).
          - A class member may be *data member* (*member variable* or *field*) or *member function* (*method*).
          - Where the *inheritance* feature is not used, the name of a class member can only be used as follows:
            - Within the class scope (anywhere within `{` and `}`, including *before* the name's introduction).
            - In a member function of the class. (There're restrictions if function is `static` → won't bother us.)
            - After the `.` (dot) operator applied to an object of the class.
            - After the `->` (arrow) operator applied to a pointer to an object of the class.
            - After the `::` (scope resolution) operator applied to the *name of the class*.

              (**NOTE**: Any source files to which layout of class is made visible can use the name of the class.)
          - (Above member-name usage is further regulated by *member access specifiers* discussed next.)
        - `public` and `private` → member access specifiers.
          - Members in `public` section → accessible *by all* functions to which layout of class is made visible.
          - Members in `private` section → accessible (directly) *only by* member functions (methods) and `friend` functions of the class.
          - (We will have little/no use for `protected` → relevant only when *inheritance* is involved).
        ```

- » More about data member (member variable):
  - ➝ May be *class-level* (decorated with `static`) or *instance-level* (not decorated with `static`).
    - ✦ In our *e.g.*: `MAX_LEN` is a class-level data member; `data` and `len` are instance-level data members.
  - ➝ A class with a *class-level* data member will have *only one copy* of that data member.
    - ✦ Memory for a *class-level* data member is allocated right away → thus OK to initialize here.
  - ➝ Each instance (object) of a class will have *its own copy* of each of the *instance-level* data members.
    - ✦ Memory for an *instance-level* data member is allocated only when the associated object is being created (constructed/instantiated) → thus *not* OK to initialize here.
- » More about member function (method):
  - ➝ Important to differentiate between *member* and *non-member* (*ordinary*, *free*, *regular*, …) functions.
    - ✦ *Member* function → declared/defined within class scope (within `{` and `}` mentioned above).
      - − Exception: a `friend` function (declared/defined within class scope) is a *non-member* function.
    - ✦ (A member function can also be `static` or non-`static` → we'll have little/no use for the former).
      - − (`static` when used w/ member function → *not* same meaning as when used w/ data member).
    - ✦ A (non-`static`) member function must be called with an *invoking object*.
      - − Other terms you may see used: *activating object*, *calling object*, *current object*.
      - − *E.g.*: (to process `ourStr` object `s1` via member function named `proc`, say)

        `s1.proc(...)` → `s1` is the invoking object

        (not standalone like `proc(...)`, the way it'd be if `proc` were *not* a member function)

- ▸ Notable *non*-entry (see **304s02SepCompFileOrgInCpp**) → `using namespace std;` directive.
  - ○ Principle of *don't make soup too salty*.

- • Implementation – `ourStr.cpp`:
  - ▸ Typical entry (see **304s02SepCompFileOrgInCpp**) → documentation (for *developer*-only-should-know items).
  - ○ Prominently → class invariant (*state-rules* to *maintain* so objects can be properly understood/handled)
    - » In regard to *contract-oriented design* → developer-developer (supplier-supplier) contract.
      - ➝ (recall) 3 key ingredients: preconditions, postconditions, *class invariant*.
      - ➝ (recall) 3 key action verbs: expect, guarantee, *maintain*.
    - » In our *e.g.*, 2 state-rules to maintain (and bring to bear) by fellow developers of `ourStr` data type:
      - ➝ Rule 1:
        - ✦ A function having access to an `ourStr` object `s` (in proper state) can inspect the object's `len` member (`s.len`) to get the # of characters there are in the string currently represented by `s`.
        - ✦ Should a function having access to an `ourStr` object `s` cause a change in the # of characters there are in the string represented by `s`, the function must *maintain* `s.len` (*i.e.*, update it accordingly and at the appropriate time) to reflect the change.
      - ➝ Rule 2:
        - ✦ Upon completion of any manipulations (by a function having access to an `ourStr` object `s`), all relevant characters of the string now represented by `s` are in `s.data[i]` where `i = 0, 1, ..., len - 1`, and `s.data[0]` has the 1st character, `s.data[1]` has the 2nd character, …, and `s.data[len - 1]` has the `len`th character.
          - − Any remaining portion of the object's `data` member (`s.data[i]` where `i >= len`) does NOT contain anything of interest.

            NOTE: *No unneeded work* should be wasted to fill such remaining portion with *anything at all* → any such wasted work not only trivializes the invariant but also invites confusion.

- » In reference to ***Assignment 1***:
  - ➝ Want to carefully read/understand the class invariant → it tells about the state-defining "*vital signs*" (<u>what</u> they are and <u>how</u> to interpret them) that apply to objects of the class.
    - + Treat it as "rules of communication" regarding object state that developers must use and uphold.
    - + When implementing each function, imagine yourself as a "different and isolated developer" that has no knowledge whatsoever about other developers (that implement other functions).
      - For an <u>accessor</u>, you rely on class invariant to get *as-is* information on object(s) involved.
      - For a <u>mutator</u>, you (1) rely on class invariant to get *as-is* information on object(s) involved, and (2) do all needed maintenance to ensure *updated* information on object(s) involved will be correctly communicated (to other developers) through the class invariant.
- ▸ Typical entry (see **304s02SepCompFileOrgInCpp**) → interface (header) file.

  QUOTE:

  Any file having ties to a component must `#include` that component's interface file.

  The implementation file implementing the component must do so.

  END_QUOTE:

- Application – `ourStrApp.cpp`:
  - ▸ Typical entry (see **304s02SepCompFileOrgInCpp**) → interface (header) file.

    QUOTE:

    Any file having ties to a component must `#include` that component's interface file.

    An application file making use of the component must do so.

    END_QUOTE:

  - ▸ Typical entry (see **304s02SepCompFileOrgInCpp**) → putting component to use.
    - (In our *e.g.*, we test `ourStr`'s various capabilities as a way to put the component to use.)
      - » (This is typical how things are for our purpose → programmer plays dual role of developer and client.)
    - `cout << "ourStr::MAX_LEN = " << ourStr::MAX_LEN << endl;`
      - » Being `static` (class level), `MAX_LEN` can be referenced even *before* there's any `ourStr` object.
      - » Being `static` (class level), `MAX_LEN` is referenced via the <class name>`::` syntax (`ourStr::MAX_LEN`).
        - ➝ Once an `ourStr` object (`s`, say) exists, we can also reference `MAX_LEN` via the <object name>`.` syntax (`s.MAX_LEN`).
        - ➝ Using the <class name>`::` syntax is better since it corresponds with *class*-level.
        - ➝ Avoid using the <object name>. syntax as it corresponds with *instance*-level and can be confusing.
    - `ourStr s1;`
      - » An `ourStr` object is instantiated/constructed and labeled/named/tagged `s1`.
        - ➝ `object` → *instance* of a class.
        - ➝ instantiation → process of creating/constructing an *instance* (*i.e.*, object) of a class.
      - » The `ourStr` class definition (as it appears in `ourStr.h`) by itself does NOT cause any `ourStr` instances (objects) to be created/constructed.
        - ➝ (In memory terms, no memory for any `ourStr` objects has been allocated.)
          - + (It does cause memory for `MAX_LEN` to be allocated, however.)
            - (In fact, that's part of what *class*-level and `static` mean.)
            - (That's also why it's OK to initialize `MAX_LEN` but not `len` and `data`.)

          previously mentioned
        - ➝ It acts like a *blueprint* showing how each `ourStr` object (once created/constructed) is structured.

- (There's not much more that we can do to put `ourStr` to use at this point.)
  - (Since we have not really done anything yet to render the component useful.)
  - (But that's for what's to come, by design.)
- (Compiling/running the program.)
  - (Separately *compile* each source file, *link* the object files, and *run* the executable file.)
  - (*make* utility → can help reduce chore of having to re-do the same thing over and over.)
    - (For ***Assignment 1*** and future assignments, be sure to not overlook any supplied *Makefile*'s.)

- `ourStr` Version 1 Build B:
  - To add *display-string* capability.
    - Handy capability for showing, whenever desired, the string represented by an `ourStr` object.
  - In reference to Build B's `ourStr.h`:
    - Documentation added:
      - Interface (function prototype) → `void showStr(std::ostream& out) const`.
        - By design, function receives an `ostream` object → for flexibility.
          - Able to display not just to `cout` but also to other `ostream`-compatible (such as `fstream`) objects.
        - With avoidance of `using namespace std;` directive → need to prefix `ostream` with `std::`.
          - Small price to pay for not violating *principle of don't make soup too salty*.
        - `const` appearing after closing parenthesis: ← `void showStr(std::ostream& out) const`
          - `showStr` is an *accessor* (not a *mutator*).
            - `showStr` shall have no business causing any *side effect* to its *invoking object*.

        NOTE: If there's no `const`, `showStr` will be a *mutator* (by default) capable of causing *side effect* to its *invoking object* → violates *principle of least privilege* (an aspect of *defensive programming*).

        NOTE: *Principle of least privilege* → reveal/enable/… only what's necessary, nothing more.

        NOTE: Consistent use of `const` to indicate that a function is meant to be an accessor → in line with good style of *explicitly indicating intent* and can also help *avoid unexpected errors*.

        NOTE: The `const` keyword is used in C++ in quite a few different contexts with different meanings, three of which (in passing and for now) are:
        (1) Declaring constants → *e.g.*: `const double PI = 3.14;` (also seen used with `MAX_LEN`).
        (2) Specifying that a member function is *accessor* → as seen used with `showStr` above.
        (*<u>Caveat</u>:* This use of `const` is valid for non-`static` member functions only.)
        (3) Passing by `const` reference → will see this use often in time.
      - No newline is inserted at the end → *principle of don't make soup too salty*.
    - Relevant header file added:
      - `#include <iostream>` → without it, use of `ostream` (in `showStr`) will cause compiler to flag an error.
    - Accessibility assigned:
      - Prototype of `showStr` placed under `public` → making `showStr` publicly accessible.
  - In reference to Build B's `ourStr.cpp`:
    - `using namespace std;` directive used:
      - Such use is really not necessary and there are downsides to doing so.
        - Done so in our *e.g.* for convenience and to avoid cluttering → striking a compromise.
    - Function name (`showStr`) prefixed with `ourStr::` (in function header):
      - (recall) 1 of the ways to use a class member listed earlier → especially needed when *outside class scope*:

QUOTE:

After the `::` (scope resolution) operator applied to the *name of the* *class*.

END_QUOTE:

NOTE: Outside class scope → outside the `{` and `}` delimiting the class' definition. Here, it's even in a file (`ourStr.cpp`) separate from the one (`ourStr.h`) containing the class' definition.

» Another way to see why the `ourStr::` prefix is necessary:

+ Without it, the compiler won't know that `showStr` is a member function of `ourStr` class → will thus treat it as a regular/ordinary/free function (and flag an error when coming upon the `const`).

- Work of `showStr` (more generally, any non-`static` member function) centered on its *invoking object*:

NOTE: (*invoking object* from earlier discussion)

QUOTE:

A (non-`static`) member function must be called with an *invoking object*.

Other terms you may see used: *activating object*, *calling object*, *current object*.

E.g.: (to process `ourStr` object `s1` via member function named `proc`, say)

`s1.proc(...)` → `s1` is the invoking object

(not standalone like `proc(...)`, the way it'd be if `proc` were *not* a member function)

END_QUOTE:

- ***Q***: In the `for` loop implementing `showStr`, which `len` and `data` are actually getting involved?
  A: The invoking object's `len` and the invoking object's `data`.

- More about invoking object:

» Invoking object's *address* is implicitly (transparently) passed to each (non-`static`) member function.

→ The address can be explicitly referenced via `this` (a keyword that's really a constant pointer).

→ Thus, `len` in the `for` loop implementing `showStr` is really short for `this->len`.
(Similarly, `data` in the same `for` loop is really short for `this->data`.)

NOTE: Had separate *local variables* `len` and `data` been declared within `showStr`, the use of `this->` would not be optional to reference the invoking object's *member variables* `len` and `data` (*i.e.*, `len` and `data` would reference the *local variables* and `this->len` and `this->data` would reference the invoking object's *member variables* `len` and `data`).

NOTE: (looking ahead → another use of `this` we will see)

Returning (a reference to) the invoking object → `return *this;`

» There's no invoking object associated with an ordinary/regular function or a `static` member function.

» `const` appearing after a member function's closing parenthesis (specifying that the member function is an *accessor* and not a *mutator*) → protects ONLY the invoking object (NOT ANY OTHER objects).

- In reference to Build B's `ourStrApp.cpp`:

  - `s1.showStr(cout);` added:

    - `s1` is the invoking object (or `showStr` is invoked on the object `s1`).
    - `cout` is the stream object application wants `showStr` to display the contents of `s1` to.

  - Application adds a newline (`cout << endl;`) as desired:

    - Bad to have `endl` effected by `showStr` (as the application may not want it) → "too salty" so to speak.

  - (Re-compiling/running the program.)

    - (Plausible explanation for the program's behavior?)

» (Re-compiling/running with `long i = -1;` added before 1st `cout` statement → program crashed.)

- `ourStr` Version 1 Build C:
  - To add (automatic) *default-initialization* capability.
    - Capability that automatically kicks in each time a named `ourStr` object is created (in default fashion) and initializes it to some consistent state.
      - In default fashion:
        - » In a way not indicating any particular desired initial state.
        - » *E.g.*: `ourStr s1;`
          - **NOTE**: Don't confuse this with `ourStr s1();` (which to the compiler means the *prototype* of a "function named `s1` that has no parameters and returns an `ourStr` object by value").
      - Automatically:
        - » Conceptually, a default-initialization function (`def_init`, say) could be written for the purpose, provided that `def_init` gets called every time a new `ourStr` object is created, for *e.g.*:

```
ourStr s1;      // creating ourStr object s1 without initialization
s1.def_init();  // do default initialization on s1
```

        - » Doing it this way suffers from the unnecessary risk of not having the desired initialization done (when `def_init` fails to be called), in which case the `ourStr` object created would be in an *inconsistent state* (that can lead to unexpected/undesirable consequences).
        - » To avoid the unnecessary risk, C++ uses *special initialization functions* (called *constructors*) that are called automatically to do one of various desired initializations every time a new object is created.
        - » For default initialization, the *special initialization function* is fittingly called *default constructor*.
  - (looking ahead → more on C++ `class`-related special functions, which we'll refer to as **Big-4**)
    - *Special initialization functions* (*constructors*):
      - All constructors have these special properties:
        (1) Same name as the class
        (2) No return type (not even `void`)
        (3) Automatically called (for usual cases where *named* objects are created)
      - For default initialization, to repeat, the *special initialization function* is called *default constructor*, and it has the additional property of having *NO parameter at all* (parameterless).
      - The *default constructor* is important enough that C++ requires that every C++ `class` MUST have it, thus making it one of the **Big-4**.
      - There's another *constructor* also important enough that C++ requires that every C++ `class` MUST have it (thus making it yet another one of the **Big-4**), called *copy constructor*, and it has the additional property (besides the 3 listed above) of having *only ONE parameter* that MUST be an object of the SAME `class`.

        (More about copy constructor to come → for now, just the above.)
    - *Special cleaning-up function* (*destructor*):

      (More about destructor to come → for now, just note that it is yet another one of the **Big-4**.)
    - *Special assignment function* (*assignment operator*):

      (More about assignment operator to come → for now, just note that it is yet another one of the **Big-4**.)
  - In reference to Build C's `ourStr.h`:
    - Documentation added:
      - Interface (function prototype) → `ourStr();`
        - » No return type (not even `void`) ⊕ same name as the class ⊕ parameterless.
        - » No `const` appearing after closing parenthesis → mutator (must be able to cause side effect to invoking object, which is the object being constructed/initialized).

- Postcondition → 1-character string containing the caret character (`'^'`):
  - Chosen so `showStr` will display something non-whitespace → null string more appropriate if for real.
- Accessibility assigned:
  - Prototype of `ourStr` placed under `public` → making `ourStr` publicly accessible.
    - `ourStr` is typically called automatically (the same goes with any other constructors), it must still be made `public` for the automatic call to be possible.

- In reference to Build C's `ourStr.cpp`:
  - Rather unusual look for default constructor's implementation:

```
ourStr::ourStr() : len(1)
{
   data[0] = '^';
}
```

    - Function header → `ourStr::ourStr()`:
      - Has no return type (not even `void`) as in the prototype..
      - Need for `ourStr::` (as "member function of `ourStr` class" tag) is just like for `showStr` (see Build B).
      - `ourStr()` is function's name → *same name as the class* to indicate that the function is a constructor.
      - **NOTE**: Don't confuse 1st `ourStr` (*class* name) with the 2nd `ourStr` (*function* name) → the look-alike is due to the "constructor of a class must have the same name as the class" requirement of C++.
    - Initializer list (or initialization list) → `: len(1)`:
      - Most expect the constructor's implementation to look as follows:

```
ourStr::ourStr()
{
   data[0] = '^';
   len = 1;
}
```

        Doing so will have the same effect (in this case) but (except in rare situations) it is better to use the *initializer list syntax* whenever possible when implementing constructors.
        - **NOTE**: `data[0] = '^';` isn't included in the initializer list because the compiler used will flag an error with the inclusion → notice the "whenever possible" qualification above.
        - **NOTE**: The initializer list syntax can only be used for the implementation of constructors.
        - **NOTE**: For more on initializer list, see **304s01mInitializerListInCpp**.

- In reference to Build C's `ourStrApp.cpp`:
  - (Unchanged from Build B.)
  - (Re-compiling/running the program.)
    - (`^` output as expected.)
      - (Re-compiling/running with `long i = -1;` added before 1st `cout` statement → program not crashing.)

■ `ourStr` Version 1 Build D:

- To add 2 accessors (`getLen` and `charAt`) and 2 mutators (`setStr` and `setChar`).
  - Some like to refer to accessors as "getters" and mutators as "setters", which is fine; however, be aware that the names of accessors/mutators don't have to have "get"/"set" in them.
  - As far as the compiler is concerned, whether a member function is an accessor or a mutator depends solely on whether or not (respectively) the function has a `const` appearing after the closing parenthesis.
- In reference to Build D's `ourStr.h`:
  - Documentation added:
    - Interface for 1st accessor → `int getLen() const`

- Interface for 2nd accessor → `char charAt(int pos) const`
  - » **NOTE**: Client is to indicate the desired character by specifying the position (begins with 1), not the array index (begins with 0 for C++); this is done so that the client is not unnecessarily exposed to (and perhaps baffled by) implementation details → *principle of client-oriented design*.
  - » **QUIZ**: Why is the 2nd precondition given as `pos <= getLen()` and not `pos <= len`?
- Interface for 1st mutator → `void setStr(const char cStr[])`
  - » **NOTE**: Per the precondition, client is to supply a null-terminated C-string.
  - » **QUIZ**: Should this precondition be checked (in the function's implementation)?
- Interface for 2nd mutator → `void setChar(int pos, char newChar)`
  - » **QUIZ**: `pos <= getLen() + 1` and `pos <= MAX_LEN` are 2 of the preconditions, why?

▸ Accessibility assigned:

- Prototypes of `setStr` and `setChar` placed under `public` → they are meant to be publicly accessible.

• In reference to Build D's `ourStr.cpp`:

▸ Trapping and handling of precondition violations:

- Although the responsibility of ensuring that preconditions are met rests on the caller (client), good developers follow the good practice of trapping and handling violations where such violations can be expediently detected.
  - » Doing so would be an important part of "idiot proofing" in real world software development.
  - » Arguably, the best C++ tool for the task is the language's *exception handling* feature.
  - » Using the exception handling feature, however, does incur some extra costs (in time and distractions) that if avoided can make the learning environment more conducive to a better focus on data structures and algorithms objectives.
  - » With the above in mind, it is preferable for our purpose to use the "less elaborate/graceful but simpler" method of simply inserting assertion statements (using the `assert` function from the `cassert` library) at where trapping and handling of precondition violations should take place.
  - » Using `charAt` as an example, the precondition of the function

```
//     pre:  pos >= 1 && pos <= getLen()
```

are attended to with these assertion statements

```
assert(pos >= 1);
assert( pos <= getLen() );
```

When compiled using GCC, the program would terminate with the following assertion failure message when the `pos >= 1` assertion fails (`charAt` invoked with `0` as the value for `pos`)

```
d: ourStr.cpp:30: char ourStr::charAt(int) const: Assertion `pos >= 1' failed.
```

QUIZ: Why is it better to <u>not</u> "lump the 2 conditions `pos >= 1` and `pos <= getLen()`and include the compound condition in one single assertion statement"?

QUIZ: Why is it better to <u>not</u> "write the 2nd condition as `pos <= len` instead of `pos <= getLen()`" although the former would seem more efficient (since it doesn't involve a function call)?

- Note that the precondition for `setStr` cannot be expediently detected (without risking getting out of array bound).

• In reference to Build D's `ourStrApp.cpp`:

▸ `ShowStrSpaced` added to show how client can develop custom functions for processing `ourStr` objects:

- Note that `ShowStrSpaced` is not a member function.
  - » It has to rely on available public member functions (`getLen` and `charAt`) to get to object's private data.

▸ Note how each of the newly added functions (including `ShowStrSpaced`) are being put to use.

- The statement `s1.setChar(6, 'm');` is included to show the effect of assertion failure.

- `ourStr` Version 1 Build E:
  - To add a *compare-for-equality* operation.
    - Must first define what equality means.
      - 2 `ourStr` objects are equal if they contain the same number of characters and each of the pairs of corresponding characters in them are equal, case-sensitively.
    - Must then decide on the C++ mechanism to use.
      - C++ gives us more than 1 way to do it.
      - We will first look at 3 different ways to do it:
        - » Using a *member function* → will name it `equal_m`
        - » Using a *non-member (ordinary, free) function that is not a friend* → will name it `equal_nmnf`
        - » Using a *non-member (ordinary, free) function that is a friend* → will name it `equal_nmf`
      - Only need to pick 1 way to do it if it is for real:
        - » Some organizations have guidelines on which way to pick (based on the kind of operation involved).
        - » We will do it multiple ways here to learn the mechanics of each mechanism.
      - For each mechanism, we must know the *how-to*'s in 3 places:
        - » In the *interface* (*header*) *file* → *how* and *where* to incorporate (write the *function prototype*).
        - » In the *implementation file* → *how* to implement (write the *function definition*).
          - Unless the function is relatively simple, this is usually the most difficult since it requires algorithm design (thus problem-solving ability).
          - In ***Assignment 1***, this part is left mostly to you (since the *header* and *driver* files have been written for you and you are not to change them).
        - » In the *client/application file* → *how* to use (make *calls to the function*).

      <u>CAUTION</u>: Even though the header and driver files have been written for you (and you are not to change them) in ***Assignment 1***, you still have to *study and understand* them. In exams, you will typically be required to know how to write the code in all the 3 places (involving one or more of the mechanisms studied).

  - In reference to Build E's `ourStr.h`:
    - Remarks regarding documentation:
      - Extra note to inform user what equality means.
    - Remarks regarding `equal_m`:
      - Included *<u>inside</u> the scope of the class* (within the pair of curly braces).
      - Specified as an *accessor* → presence of `const` after closing parenthesis.
      - Has one parameter that is passed by `const` reference.
        - » But comparing for equality is a binary operation → function will compare the *object passed* with the *invoking object*.
    - Remarks regarding `equal_nmnf`:
      - Included *<u>outside</u> the scope of the class* (outside the pair of curly braces).
      - Has two parameters that are passed by `const` reference.
        - » A non-member function cannot have an invoking object and comparing for equality is a binary operation, so the two objects to be compared must both be *passed* as parameters.
      - Cannot be specified as an accessor (no `const` after closing parenthesis).

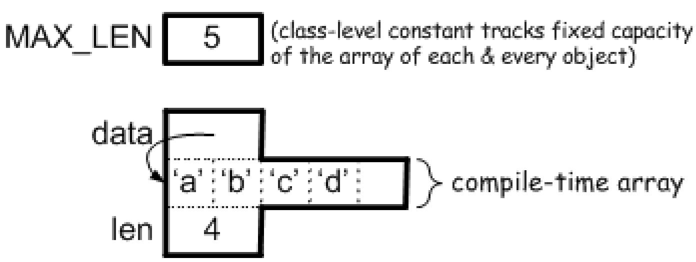
      - That use of `const` is to protect the invoking object, but there's no invoking object associated with a non-member function → attempting to do so will lead to compilation error.
  - Remarks regarding `equal_nmf`:
    - Included <u>*inside*</u> *the scope of the class* (within the pair of curly braces).
      - Preceded by keyword `friend` to indicate that it's a *friend* function.
    - About *friend* functions (and the concept of *friendship*):
      - A `friend` function is a *non-member* function that's granted the special privilege to *directly access* the non-public members (data and functions) of a class.
        - Friendship is irrelevant to a member function (since it already has that privilege).
      - Much like a non-club member given special VIP status that enables him/her to enjoy club privileges.
      - Use of `friend` function is considered by many as a bad idea → breaks encapsulation.
        - Should only be used sparingly and where warranted.
        - Typically as a compromise for performance reasons (to avoid requiring a heavily-used function having to keep calling public functions to access non-public members of a class).
        - A counter-argument (perhaps saving grace) offered by some: owner of the class is the one who decides on (thus has control over) whether to grant friendship.
        - <u>CAUTION</u>: For assignments, there will be severe penalty if, to avoid compilation errors due to illegal access of non-public class members, a non-friend function is made into a friend function.
    - Interface is essentially identical to that of `equal_nmnf` since it is still a non-member function.
  - Remarks regarding the 3 mechanisms (member, non-member non-friend, non-member friend):
    - <u>Common pattern</u>: implementing a given operation via a member function (compared to a non-member function) usually results in the function receiving *1 parameter less*.
      - The invoking object takes the place of one of the parameters a non-member function would receive.
    - When implementing a *binary operation* using a *member function*, the *invoking object* is implicitly the LHS (Left Hand Side) object.
    - When implementing a *binary operation* using a *non-member function*, the *object received as first parameter* is implicitly the LHS (Left Hand Side) object.
    - When calling a function implementing a *binary operation*, which object should be the LHS and which the RHS (Right Hand Side) is important if the operation is *non-commutative* (doesn't commute).
      - Example commutative operations involving numbers: addition and multiplication.
      - Example non-commutative operations involving numbers: subtraction and division.
- In reference to Build E's `ourStr.cpp`:
  - Note that there's no `ourStr::` appearing in the function header of `equal_nmnf` and `equal_nmf`.
    - Since they are not members of the `ourStr` class.
  - Note that keyword `friend` cannot be included in the function header of `equal_nmnf`.
  - Note that within `equal_m`, private data of the invoking object and the passed object is accessed directly.
    - Misconception: a member function can directly access the private members of ***only*** *the invoking object*.
    - Truth: a member function can directly access the private members of ***all*** *objects of the class*.
  - Incidentally and confusingly to many:
    - Misconception: an accessor cannot cause any side effect on ***all*** *objects of the class*.
    - Truth: an accessor cannot cause any side effect on ***only*** *the invoking object*.
  - Comparing the code for `equal_nmnf` and `equal_nmf`:
    - Accessing non-public data via public member (by `equal_nmnf`) functions versus directly (by

      `equal_nmf`).
      - Of course, `equal_nmf` can still do like `equal_nmnf` does but that will constitute not taking advantage of the friendship.
    - Note the "principle of don't do the same thing more than once if avoidable" is being observed in `equal_nmnf` (and to some extent in `equal_nmf` also).
  - In reference to Build E's `ourStrApp.cpp`:
    - Note how each (of `equal_m`, `equal_nmnf` and `equal_nmf`) is called.
      - Note that it doesn't matter (in this case) which of the 2 objects (to be compared) is made the LHS (invoking object in `equal_m` and first argument in `equal_nmnf` and `equal_nmf`) because comparing for equality is commutative.
    - Note that `equal_nmnf` and `equal_nmf` are called in the same fashion.
      - The interface is the same, only the implementation is different.
  - Prelude to Build F:
    - What would be a nicer and more intuitive way for a user to compare 2 `ourStr` objects for equality?
      - `if (s1 == s2) ...`
    - What would happen if an attempt at that is made?
      - Compilation error.
    - Because for each class, C++ provides only 4 operators for free (won't cause compilation error):
      - The *size-of* operator (`sizeof`).
      - The *address-of* operator (`&`).
      - The *member-of* (or dot) operator (`.`).
      - The *copy assignment* operator (`=`).

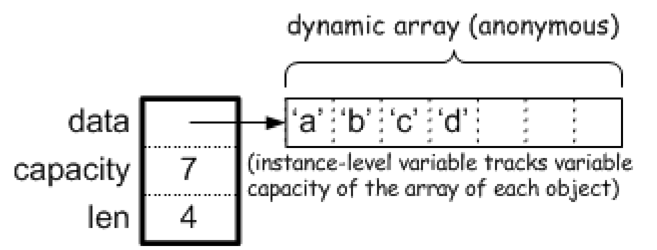
      We will see later that the free (compiler-supplied) copy assignment operator *may not be adequate*.
    - But C++ provides a mechanism called *operator overloading* for achieving that.
      - The main goal for studying operator overloading here is actually to be able to write a custom version of the copy assignment operator when the free (compiler-supplied) version is not adequate.
      - Once learned, however, there's no reason to not use it for other operators.
- `ourStr` Version 1 Build F:
  - To enable the use of == as compare-for-equality operator.
    - To give a simple/basic introduction to the *operator overloading* feature of C++.
    - Overloading the == operator turns out to be among the simplest.
      - Should not think that overloading other operators is just as simple.
  - Key idea: write a function named `operator<sym>` where `<sym>` is the symbol used for the operator.
    - Thus, to enable the use of ==, a function named `operator==` must be written.
  - In reference to Build F's `ourStr.h` and `ourStr.cpp`:
    - The code for one of the 3 compare-for-equality functions (`equal_m`, `equal_nmnf` and `equal_nmf`) is simply adopted for the purpose.
    - Everything remains unchanged except that the function name has been changed to `operator==`.
    - Note that only one of the 3 possible versions of `operator==` is made active (the other 2 versions are commented out, in both files).
  - In reference to Build F's `ourStrApp.cpp`:
    - Note the use of `if (s1 == s2) ...`.

- It is still `if (s1 == s2) ...` even if the code adopted is changed to 1 of the other 2 versions → this is so because, through operator overloading, the C++ compiler is essentially providing an additional service as follows:
  - Replace `s1 == s2` with `s1.operator==(s2)` if overloading is done via a member function.
  - Replace `s1 == s2` with `operator==(s1, s2)` if overloading is done via a non-member function.
  - In fact, the user can also call the `operator==` function directly (although doing so would be bypassing the operator overloading benefit).

- This marks the end of `ourStr` Version 1 discussion.
  - 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 pictures* 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