

String Objects: The `string` class library

C-strings vs. string objects

- Recall that in C++ (and C), there is no built-in string type
 - Basic strings (C-strings) are implemented as arrays of type `char` that are terminated with the null character
 - string literals (i.e. strings in double-quotes) are automatically stored this way
- Advantages of C-strings:
 - Compile-time allocation and determination of size. This makes them more efficient, faster run-time when using them
 - Simplest possible storage, conserves space
- Disadvantages of C-strings:
 - Fixed size
 - Primitive C arrays do not track their own size, so programmer has to be careful about boundaries
 - While there are some library functions that work with C-strings, programmer still has to be careful to make correct calls. These library functions do not protect boundaries either!
 - Less intuitive notation for such usage (library features)

string objects

- Recall that C++ allows the creation of *objects*, specified in *class* libraries
- Along with this comes the ability to create new versions of familiar operators
- Coupled with the notion of *dynamic memory allocation* (not yet studied in this course), objects can store variable amounts of information inside
- Therefore, a `string` class could allow the creation of string objects so that:
 - the size of the stored string is variable and changeable
 - boundary issues are handled inside the class library
 - More intuitive notations can be created
- One of the standard libraries in C++ is just such a class

The `string` class library

- Recall that the `cstring` library consists of functions for working on C-strings. This is a C library
- The library called `string`, which is part of the "Standard Template Library" in C++, contains a class called `string`
- Strings are declared as regular variables (not as arrays), and they support:
 - the assignment operator `=`
 - comparison operators `==`, `!=`, etc
 - the `+` operator for concatenation
 - type conversions from c-strings to string objects
 - a variety of other member functions
- To use this library, make sure to `#include` it:

```
#include <string>
```

Declaring and initializing

- Declare like a regular variable, using the word `string` as the type:

```
string firstname;  
string lastname;  
string address;  
string s1;
```

- To initialize, you can use the assignment operator, and assign a string object to another, or a string literal to a string object:

```
firstname = "Joe";  
lastname = "Schmuckatelli";  
address = "123 Main St.";  
s1 = firstname; // s1 is now "Joe" also
```

- And you can initialize on the same line as the declaration:

```
string s2 = "How are you?";  
string s3 = "I am fine.";  
string s4 = "The quick brown duck jumped over the lazy octopus";
```

- You can also initialize on the declaration statement in this format:

```
string fname("Marvin");  
string lname("Dipwart");  
string s2(s1); // s2 is created as a copy of s1
```

Comparing string objects

- You can compare the contents of string objects with the standard comparison operators:

```
if (s1 == s2)  
    cout << "The strings are the same";  
  
if (s1 < s2)  
    cout << "s1 comes first lexicographically";
```

- You can also mix and match with C-strings, as long as one of the operands is a string object:

```
if (s1 == "Joe")  
    cout << "The first student is Joe";  
  
if (s2 > "apple")  
    cout << "s2 comes after apple in the dictionary";
```

- **Remember!** The ordering on strings is a lexicographical ordering, which goes by ASCII values of the characters. So it's not exactly the same as alphabetical
 - In the ASCII character set, upper case letters all come before the lower case letters
 - So...
 - "apple" < "apply"

- "apple" > "Apply"
- "apple" > "Zebra"

Concatenation with the + operator

- The + operator is overloaded in this library to perform string concatenation. It takes two strings, concatenates them, and returns the result
- Example:

```
string s1 = "Fish";
string s2 = "bait";
string s3;

s3 = s1 + s2;           // s3 is now "Fishbait"
```

- You can also concatenate a C-string onto a string object:

```
string s4 = s3 + " odor";    // s4 is now "Fishbait odor"
```

- The += operator is also supported, for concatenation ON to the string on the left side:

```
string t1 = "Bird";
string t2 = "Boogie";
t1 += "brain";             // t1 is now "Birdbrain"
t1 += " ";
t1 += t2;                  // t1 is now "Birdbrain Boogie"
```

Input and output

- The string library also includes versions of the insertion and extraction operators, for use with string objects
- The insertion operator is used as with any other variable, and it will print the contents of the string:

```
string s1 = "Hello, world";
string s2 = "Goodbye, cruel";

cout << s1;                // prints "Hello, world"
cout << s2 + " world";     // prints "Goodbye, cruel world"
```

- The extraction operator, like the one for c-strings, will skip leading white space, then read up to the next white space (i.e. one word only):

```
string s3;
cin >> s3;                // will read one word
```

- To read more than one word (i.e. through white space), you can use a stand-alone function called getline. This is similar to the getline function for c-strings, but the syntax is a little different.

```
char address[30];          // c-string
string addr;               // string object

cin.getline(address, 30);  // reads up to newline, for c-string
getline(cin, addr);        // reads up to newline, for string object
```

```
cin.getline(address, 30, ','); // reads up to comma, for c-string
getline(cin, addr, ',');      // reads up to comma, for string object
```

- For the string object version of `getline`, the first parameter is the input stream (like `cin`, or a file input object), the second parameter is the string object, and the third optional parameter is the delimiter. Notice that you are NOT required to pass in a size for string objects
- Recall this code example illustrating inputs on c-strings: [io1.cpp](#)
- Here's a similar example showing inputs on string objects: [io2.cpp](#)

The [] operator

- A string object stores the string *internally* as an array of characters, but it makes the notation easier, so that you don't have to declare it like an array
- However, sometimes we want to access individual letters or character positions in the string. The class supports the [] operator for this purpose
- Indexing starts at 0, just like with an array.
- Usage: `string_name[index]`. This call will return a reference to the character at position "index". This means it can be read and it can be changed through this operation
- Examples:

```
string s1 = "Apple pie and ice cream";
cout << s1[0];           // prints 'A'
cout << s1[4];           // prints 'e'

s1[4] = 'y';
s1[8] = 'g';
cout << s1;              // prints "Apply pig and ice cream"
```

Some useful member functions

- Recall that member functions are called through the dot-operator
- In the following descriptions, X and Y represent integers, and CH represents a character
- For a more complete listing of string class features, see your textbook (chapter 10) and/or [this description on cplusplus.com](#)

Member function descriptions

- `size()` -- returns the length of the string
- `length()` -- returns the length of the string (same as `size()`)
- `capacity()` -- returns the current allocated size of the string object (allocation might be larger than current usage, which is the length)
- `resize(X, CH)` -- changes the string's allocated size to X. If X is bigger than the currently stored string, the extra space at the end is filled in with the character CH
- `clear()` -- delete the contents of the string. Reset it to an empty string
- `empty()` -- return true if the string is currently empty, false otherwise
- `at(X)` -- return the character at position X in the string. Similar to using the [] operator

- **Substrings**

- `substr(X, Y)` -- returns a copy of the substring (i.e. portion of the original string) that starts at index X and is Y characters long
- `substr(X)` -- returns a copy of the substring, starting at index X of the original string and going to the end
- Examples:

```
string s1 = "Greetings, earthling";  
string s2 = s1.substr(11,5);           // s2 is now "earth"  
string s3 = s1.substr(4);             // s2 is now "tings, earthling"
```

- **Append** -- several versions. All of these append something onto the END of the original string (i.e. the calling object, before the dot-operator)
 - `append(str2)` -- appends str2 (a string or a c-string)
 - `append(str2, Y)` -- appends the first Y characters from str2 (a string or char array)
 - `append(str2, X, Y)` -- appends Y characters from str2, starting at index X
 - `append(X, CH)` -- appends X copies of the character CH
- **Assign** -- there are several functions called `assign` which are for assigning data to a string. There are versions with the same parameter lists as with the `append` functions, but these assign the string to the requested data, while `append` adds it to the end of an existing string
- **Compare** -- multiple versions
 - `str1.compare(str2)` -- performs a comparison, like the c-string function `strcmp`. A negative return means str1 comes first. Positive means str2 comes first. 0 means they are the same
 - `str1.compare(str2, X, Y)` -- compares the portions of the strings that begin at index X and have length Y. Same return value interpretation as above
- **Find** -- multiple versions
 - `str.find(str2, X)` -- returns the first position at or beyond position X where the string str2 is found inside of str
 - `str.find(CH, X)` -- returns the first position at or beyond position X where the character CH is found in str
- **Insert** -- multiple versions
 - `str.insert(X, Y, CH)` -- inserts the character CH into string str Y times, starting at position X
 - `str.insert(X, str2)` -- inserts str2 (string object or char array) into str at position X