C++ String Variables

The standard C++ library provides an object string type to complement the string literals used earlier:

```
string Name1,        // must #include <string>
       Name2;
```

A string variable may be assigned the value of a string literal or another string variable:

```
Name1 = “Bjarne Stroustrup”;
Name2 = Name1;

string myString;
myString = ‘N’;

const int asciiN = 78;
myString = asciiN;              // string equals “N”
```

Note: do not accidentally use `#include <string.h>` as this will include the C-style array of char string library. Standard C++ class libraries do not use the “**.h**” extension of the C libraries.

String Initialization

A string variable may be initialized at the point of declaration:

```
string Greet1 = “Hello”,
       Greet2 = Greet1;
```

It is not, however, legal to assign a char or int value to a string in the declaration:

```
string String1 = ‘N’,      // illegal
       String2 = 78;       // illegal
```

Since it would be legal to assign `‘N’` to `String1`, you might ask why it is illegal to initialize `String1` with `‘N’` in the declaration of `String1`. The full reason would take us into a discussion of the notion of a class in C++, a topic not covered in this course.

Suffice it to say at this point that the statements above are true.

String Initialization (continued)

It is not legal to have a line break within a string literal in C++:

```
string BadString = “It is as a tale told by an idiot, // not
                   full of sound and fury,           // legal
                   signifying nothing.”;
```

. . . however, somewhat perversely, this is OK:

```
string LongString = “It is as a tale told by an idiot, ”
                    “full of sound and fury, ”
                    “signifying nothing.”;
```

And, of course, it is legal for a string literal to contain a newline character:

```
string Prompt = “Type your user id below and press <Enter>:\n”;
```

String Output

A string variable may be printed by inserting it to an output stream, just as with any simple variable:

```
string Greetings = “Hello, world!”;
cout << Greetings << endl;
```

Just as with string literals, no whitespace padding is provided automatically, so:

```
cout << Greetings << “It’s a wonderful day!”;
```

would print:

```
Hello, world!It’s a wonderful day!
```

Of course, you can provide whitespace yourself:

```
cout << Greetings << “  ” << “It’s a wonderful day!”;
```

String Input: extraction

The stream extraction operator may be used to read characters into a string variable:

```
string Greetings;
cin >> Greetings;
```

The extraction statement reads a whitespace-terminated string into the target string (`Greetings` in this case), ignoring any leading whitespace and not including the terminating whitespace character in the target string.

The amount of storage allocated for the variable `Greetings` will be adjusted as necessary to hold the number of characters read. (There is a limit on the number of characters a string variable can hold, but that limit is so large it is of no concern.)

Of course, it is often desirable to have more control over where the extraction stops.

String Input: getline( )

The `getline( )` standard library function provides a simple way to read character input into a string variable, controlling the "stop" character.

Suppose we have the following input file:

```
Fred Flintstone        Laborer        13301
Barney Rubble          Laborer        43583
```

There is a single tab after the employee name, another single tab after the job title, and a newline after the ID number.

Assuming iFile is connected to the input file above, the statement

```
getline(iFile, String1);
```

would result in String1 having the value:

```
"Fred Flintstone  Laborer        13301"
```

String Input: getline( ) (continued)

As used on the previous slide, `getline( )` takes two parameters. The first specifies an input stream and the second a string variable.

Called in this manner, `getline( )` reads from the current position in the input stream until a newline character is found.

Leading whitespace is included in the target string.

The newline character is removed from the input stream, but not included in the target string.

It is also possible to call `getline( )` with three parameters. The first two are as described above. The third parameter specifies the "stop" character; i.e., the character at which `getline( )` will stop reading from the input stream.

By selecting an appropriate stop charcter, the `getline()`function can be used to read text that is formatted using known delimiters. The example program on the following slides illustrates how this can be done with the input file specified on the preceding slide.

String Input Example

```
#include <fstream>                          // file streams
#include <iostream>                         // standard streams
#include <string>                           // string variable support

using namespace std;                        // using standard library

void main() {

   string EmployeeName, JobTitle;           // strings for name and title
   int    EmployeeID;                       // int for id number

   ifstream iFile;                          // open input file
   iFile.open("String1.dat");
                                            // Priming read:
   getline(iFile, EmployeeName, '\t');      //    read to first tab
   getline(iFile, JobTitle, '\t');          //    read to next tab
   iFile >> EmployeeID;                     //    extract id number
   iFile.ignore(80, '\n');                  //    skip to start of next line
```

String Input Example (continued)

```
   while (iFile) {                                  // read to failure
      cout << "Next employee: " << endl;            // print record header
      cout << EmployeeName << endl                  // name on one line
           << JobTitle     << "     ”               // title and id number
           << EmployeeID   << endl << endl;         //    on another line

      getline(iFile, EmployeeName, '\t');           // repeat priming read
      getline(iFile, JobTitle, '\t');               //    logic
      iFile >> EmployeeID;
      iFile.ignore(80, '\n');
   }

   iFile.close();                                   // close input file
}
```

This program takes advantage of the formatting of the input file to treat each input line as a collection of logically distinct entities (a name, a job title, and an id number). That is almost certainly more useful than simply grabbing a whole line of input at once.

Strings are C++ Classes

Like the input and output streams, cin and cout and their file-oriented siblings, string variables (objects) in C++ are actually instances of the standard string class.

Being classes, every string object automatically has a number of associated functions (called member functions) that you can use to perform operations on that string or to obtain information about it. The following slides will present a few of the basic string member functions.

Recall the syntax for using a member function:

```
string myString = “Virginia Polytechnic Institute”;

int myLength = myString.length( );    // call the length() member
                                      //   function of the string
                                      //          object myString
```

To use a member function, you give the object name, followed by a period, followed by the function call (function name and parameter list, if any).

Length of a String

The length of a string is the number of characters it contains, including whitespace characters, if any. The length of the string currently stored in a string variable may be found by using the member function:

```
int length( );
```

```
string s1 = "Fred Flintstone";
int    sLength = s1.length( );        // sLength == 15
```

The length is probably most useful in formatting output. For example:

```
cout << s1;
cout << setw(20 - s1.Length( )) << Age;
```

will print the name in `s1`, followed by the value of `Age`, right-justified to column 20.

Testing if a String is Empty

The Boolean member function

```
bool empty( );
```

returns true if the string variable currently holds no characters and false otherwise. For example, one might use this function to determine whether a read attempt actually placed any characters into the target string:

```
string s1 = “”;
cin >> s1;
if (s1.empty( ))
   cout << “Read failed” << endl;
```

Of course, the test is only useful if you make certain that `s1` is empty before attempting to read something into it.

String Concatenation

Two strings may be concatenated; that is, one may be appended to another:

```
string Greet1    = “Hello”;
string Greet2    = “world”;
string Greetings = Greet1 + “, ” + Greet2 + “!”;
```

Here, the concatenation operator (+) is used to combine several strings, variable and literal and the result is assigned to `Greetings`. The effect of the statement above is the same as:

```
string Greetings = “Hello, world!”;
```

You may use the concatenation operator to combine string variables, string literals and characters:

```
Greetings = Greetings + ‘\n’;
```

Comparing Strings for Equality

Two strings may be compared for equality using the usual equals relational operator (==). So we may write the following:

```
string s1 = “yadda”;
string s2 = s1 + s1 + s1;
string s3 = “yadda yadda yadda”;
if (s2 == s3)
   cout << s2 << “ equals ” << s3 << endl;
else
   cout << s2 << “ doesn’t equal ” << s3 << endl;
```

We may also use the not-equals operator (!=) with string variables:

```
string s3 = “”;

while (s3 != s2)
   s3 = s3 + s1;
```

The other relational operators (<, <=, >, >=) may also be used with C++ string variables.

Lexicographic Comparison

Two strings may also be compared by using the member function

```
int compare( );
```

`s1.compare(s2)` returns:

- a negative value, if the first differing element in s1 compares less than the corresponding element in s2 (as determined by their ASCII codes), or if s1 is a prefix of s2, but s2 is longer;
- zero, if s1 == s2;
- a positive value, otherwise.

Lexicographic Comparison Example

Given the strings:

```
string Worker1 = “Fred Flintstone”;
string Worker2 = “Fred Munster”;
string Worker3 = “e e cummings”;
string Worker4 = “Fred Munst”;
```

the compare function would behave as follows:

```
int c1 = Worker1.compare(Worker2);    // c1 < 0

int c2 = Worker1.compare(Worker3);    // c2 < 0 (Why?)

int c3 = Worker2.compare(Worker4);    // c3 > 0

int c4 = Worker1.compare(Worker1);    // c4 == 0
```

Accessing String Elements

The character at a particular position in a string variable may be obtained by using the member function:

```
char at(int position);
// position:    position of desired element
```

For example:

```
string s1 = "mairsy doates and doesy doates";
char ch1 = s1.at(5);                    // ch1 == 'y'
```

Note that the positions in a string are numbered sequentially, starting at zero. So:

```
for (int i = 7; i <= 12; i++)
   cout << s1.at(i) << '\t';
```

would print: d o a t e s

Accessing String Elements (continued)

The character at a particular position in a string variable may also be referenced by indexing a string object.

For example:

```
string s1 = “mairsy doates and doesy doates”;
char ch1 = s1[5];              // ch1 == ‘y’
```

Note that the positions in a string are numbered sequentially, starting at zero. So:

```
for (int i = 7; i <= 12; i++)
   cout << s1[i] << ‘\t’;
```

would print: `d  o  a  t  e  s`

Inserting one String into Another

A string of characters may be inserted at a particular position in a string variable by using the member function:

```
string& insert(int startinsert, string s);

// startinsert:   position at which insert begins
// s:             string to be inserted
```

For example:

```
string Name = “Fred Flintstone”;
string MiddleInitial = “ G.”;
Name.insert(4, MiddleInitial);
cout << Name << endl;
```

prints: `Fred G. Flintstone`

The function returns (a reference to) the string `s1` which can be assigned to another string variable if desired; but the content of the original string is changed in any case.

Inserting a Part of one String into Another

Another version of the insert function takes four parameters:

```
string& insert(int startinsert, string s, int startcopy,
                                          int numtocopy);

// startinsert:   position at which insert begins
// s:             string to be inserted
// startcopy:     position (in s) of first element to be used
// numtocopy:     number of elements (of s) to be used
```

For example:

```
string s4 = “0123456789”;
string s5 = “abcdefghijklmnopqrstuvwxyz”;
s4.insert(3, s5, 7, 5);
cout << "s4: " << s4 << endl;
```

prints: `s4: 012hijkl3456789`

Note: a sequence of characters from a string is called a substring.

Extracting a Substring

A substring of a string may be extracted (copied) and assigned to another by using the member function:

```
string& substr(int startcopy, int numtocopy);

// startcopy:   position at which substring begins
// numtocopy:   length of substring
```

For example:

```
string s4 = “Fred Flintstone”;
string s5 = s4.substr(5, 10);
cout << s4 << endl << s5 << endl;
```

prints:
```
Fred Flintstone
Flintstone
```

Erasing a Substring

A substring may be deleted from a string by using the member function:

```
string& erase(int starterase, int numtoerase);

// starterase:   position of first element to be erased
// numtoerase:   number of elements to be erased
```

For example:

```
string s6 = “abcdefghijklmnopqrstuvwxyz”;
s6.erase(3, 5);
cout << "s6: " << s6 << endl;
```

would print: `s6: abcijklmnopqrstuvwxyz`

Replacing a Substring

A substring may be erased and replaced by another substring by using the member function:

```
string& replace(int startreplace, int numtoreplace,
                                  string s);

// startreplace: position of first element to be replaced
// numtoreplace: number of elements to be replaced
```

For example:

```
string s6 = “abcdefghijklmnopqrstuvwxyz”;
string s7 = “Fred Flintstone”;
s6.replace(3, 5, “01234”);
s7.replace(0, 4, “Bradley”);
cout << "s6: " << s6 << endl;
cout << "s7: " << s7 << endl;
```

would print:

```
s6: abc01234ijklmnopqrstuvwxyz
s7: Bradley Flintstone
```

Searching for a Substring

A string may be searched for an occurrence of a substring by using the member function:

```
int find(string s, int startsearch);

// s:           substring to be searched for
// startsearch: position at which to begin search
// returns      position at which matching substring
//              starts; -1 if no match is found
```

For example:

```
string s1 = “To be or not to be, that is the question.”;
int loc = s1.find(“be”, 0);
int newloc = s1.find(“be”, loc + 1);
cout << loc << ‘\t’ << newloc << endl;
```

prints: `3  16`

Note: using `loc` instead of `loc + 1` in the second call would result in finding the first occurrence again.

Just for Fun

Putting several of the member functions together:

```
string s1 = “But I have heard it works, even if you don’t believe in it.”;

s1.erase(0, 4);                          // erase initial “But ”

s1.replace(s1.find(“even”, 0), 4, “only”); // change “even” to “only”

s1.replace(s1.find(“don’t ”, 0), 5, “”);   // erase “don’t ” by replacing it
                                         //       with the empty string
cout << s1 << endl;
```

prints: `I have heard it works, only if you believe in it.`

And That's Just the Beginning

This chapter includes only a minimal introduction to the world of string objects in C++.

There are many additional member functions. For example, there are six different `compare` member functions and ten different `replace` member functions in the standard C++ string library.

The interested reader is referred to Bjarne Stroustrup's excellent *The C++ Programming Language, 3rd Ed.* for further details.