

C++ Strings

Original handout written by Neal Kanodia, with help from Steve Jacobson.

C++ Strings

One of the most useful data types supplied in the C++ libraries is the string. A string is a variable that stores a sequence of letters or other characters, such as "**Hello**" or "**May 10th is my birthday!**". Just like the other data types, to create a **string** we first declare it, then we can store a value in it.

```
string testString;  
testString = "This is a string.";
```

We can combine these two statements into one line:

```
string testString = "This is a string.";
```

Often, we use strings as output, and **cout** works exactly like one would expect:

```
cout << testString << endl;
```

will print the same result as

```
cout << "This is a string." << endl;
```

In order to use the string data type, the C++ string header `<string>` must be included at the top of the program. Also, you'll need to include `genlib.h` to make the short name **string** visible instead of requiring the cumbersome `std::string`. (As a side note, `std` is a C++ *namespace* for many pieces of functionality that are provided in standard C++ libraries. For the purposes of this class, you won't need to know about namespaces.) Thus, you would have the following `#include`'s in your program in order to use the **string** type.

```
#include <string>  
#include "genlib.h"
```

A string is basically a sequence of characters.

Basic Operations

Let's go into specifics about the string manipulations you'll be doing the most.

Counting the number of characters in a string. The **length** method returns the number of characters in a string, including spaces and punctuation. Like many of the string operations, **length** is a *member function*, and we invoke member functions using *dot notation*. The string that is the receiver is to the left of the dot, the member function we are

invoking is to the right, (e.g. `str.length()`). In such an expression, we are requesting the length from the variable `str`.

example program:

```
#include <string>
#include <iostream>
#include "genlib.h"

int main()
{
    string small, large;
    small = "I am short";
    large = "I, friend, am a long and elaborate string indeed";

    cout << "The short string is " << small.length()
          << " characters." << endl;
    cout << "The long string is " << large.length()
          << " characters." << endl;
    return 0;
}
```

output:

```
The short string is 10 characters.
The long string is 48 characters.
```

Accessing individual characters. Using square brackets, you can access individual characters within a string, similar to as if the string were an array. Just as with array access, positions are numbered starting from 0, and the last position is (length - 1). Using square brackets, you can both read the character at a position and assign to that position.

example program:

```
#include <string>
#include <iostream>
#include "genlib.h"

int main()
{
    string test;
    test = "I am Q the omnipot3nt";

    char ch = test[5];          // ch is 'Q'
    test[18] = 'e';           // we correct misspelling of omnipotent

    cout << test << endl;
    cout << "ch = " << ch << endl;
    return 0;
}
```

output:

```
I am Q the omnipotent
ch = Q
```

Be careful not to access positions outside the bounds of the string! The square bracket operator is not range-checked (for efficiency reasons) and thus reading from or writing to

an out-of-bounds index tends to produce difficult-to-track-down errors. There is an alternate member function **at(index)** that retrieves the character at a position with the benefit of built-in range-checking.

Passing, returning, assigning strings. C++ string variables are designed to behave like ordinary primitive types with regard to assignment. Assigning one string to another makes a new copy of the character sequence. For example:

```
string str1= "hello";  
string str2 = str1; // makes a new copy  
str1[0] = 'y';      // changes only str1, str2 is not changed
```

Similarly, passing and returning strings from functions copies the string contents. If you change a string parameter within a function, changes are not seen in the calling function unless you have specifically passed the string by reference.

Comparing two strings: You can compare two strings for equality using the ordinary **==** and **!=** operators (just like they were **ints**). Suppose you ask the user for his or her name. If the user is Julie, the program prints a warm welcome. If the user is not Neal, the program prints the normal message. Finally... if the user is Neal, it prints a less enthusiastic response.

example program:

```

#include <string>
#include <iostream>
#include "simpio.h"
#include "genlib.h"

int main() {
    string myName = "Neal";

    while (true) {
        cout << "Enter your name (or 'quit' to exit): ";
        string userName = GetLine();

        if (userName == "Julie") {
            cout << "Hi, Julie! Welcome back!" << endl << endl;
        } else if (userName == "quit") {
            // user is sick of entering names, so let's quit
            cout << endl;
            break;
        } else if (userName != myName) {
            // user did not enter quit, Julie, or Neal
            cout << "Hello, " << userName << endl << endl;
        } else {
            cout << "Oh, it's you, " << myName << endl << endl;
        }
    }
    return 0;
}

```

output:

```

Enter your name (or 'quit' to exit): Neal
Oh, it's you, Neal

Enter your name (or 'quit' to exit): Julie
Hi, Julie! Welcome back!

Enter your name (or 'quit' to exit): Leland
Hello, Leland

Enter your name (or 'quit' to exit): quit

Press any key to continue

```

You can use `<`, `<=`, `>`, and `>=` to order strings. These operators compare strings lexicographically, character by character and are case-sensitive. The following comparisons all evaluate to true: `"A" < "B"`, `"App" < "Apple"`, `"help" > "hello"`, `"Apple" < "apple"`. The last one might be a bit confusing, but the ASCII value for `'A'` is 65, and comes before `'a'`, whose ASCII value is 97. So `"Apple"` comes before `"apple"` (or, for that matter, any other word that starts with a lower-case letter).

Appending to a string: C++ strings are wondrous things. Suppose you have two strings, `s1` and `s2` and you want to create a new string of their concatenation. Conveniently, you can just write `s1 + s2`, and you'll get the result you'd expect. Similarly, if you want to append to the end of string, you can use the `+=` operator. You can append either another string or a single character to the end of a string.

example program:

```

#include <string>
#include <iostream>
#include "genlib.h"

int main()
{
    string firstname = "Leland";
    string lastname = " Stanford";

    string fullname = firstname + lastname;
    // concat the two strings

    fullname += ", Jr";      // append another string
    fullname += '.';        // append a single char

    cout << firstname << lastname << endl;
    cout << fullname << endl;

    return 0;
}

```

output:

```

Leland Stanford
Leland Stanford, Jr.

```

More (Less Used) Operations

The string class has many more operations; we'll show just a few of the more useful ones below.

Searching within a string. The string member function **find** is used to search within a string for a particular string or character. A sample usage such as **str.find(key)** searches the receiver string **str** for the **key**. The parameter **key** can either be a string or a character. (We say the **find** member function is *overloaded* to allow more than one usage). The return value is either the starting *position* where the key was found or the constant **string::npos** which indicates the key was not found.

Occasionally, you'll want to control what part of the string is searched, such as to find a second occurrence past the first. There is an optional second integer argument to **find** which allows you to specify the starting position; when this argument is not given, 0 is assumed. Thus, **str.find(key, n)** starts at position **n** within **str** and will attempt to find **key** from that point on. The following code should make this slightly clearer:

example program:

```

#include <string>
#include <iostream>
#include "genlib.h"

int main()
{
    string sentence =
        "Yes, we went to Gates after we left the dorm.";

    int firstWe = sentence.find("we");           // finds the first "we"
    int secondWe = sentence.find("we", firstWe+1); // finds "we" in "went"
    int thirdWe = sentence.find("we", secondWe+1); // finds the last "we"
    int gPos = sentence.find('G');
    int zPos = sentence.find('Z');              // returns string::npos

    cout << "First we: " << firstWe << endl;
    cout << "Second we: " << secondWe << endl;
    cout << "Third we: " << thirdWe << endl;

    cout << "Is G there? ";
    if (gPos == string::npos) {
        cout << "No!" << endl;
    } else {
        cout << "Yes!" << endl;
    }
    cout << "Is Z there? ";
    if (zPos == string::npos) {
        cout << "No!" << endl;
    } else {
        cout << "Yes!" << endl;
    }
    return 0;
}

```

output:

```

First we: 5
Second we: 8
Third we: 28
Is G there? Yes!
Is Z there? No!

```

Extracting substrings. Sometimes you would like to create new strings by excerpting portions of a string. The **substr** member function creates such substrings from pieces of the receiver string. You specify the starting position and the number of characters. For example, **str.substr(start, length)** returns a new string consisting of the characters from **str** starting at the position **start** and continuing for **length** characters. Invoking this member function does not change the receiver string, it makes a *new* string with a copy of the characters specified.

example program:

```

#include <string>
#include <iostream>
#include "genlib.h"

int main()
{
    string oldSentence;
    oldSentence = "The quick brown fox jumped WAY over the lazy dog";
    int len = oldSentence.length();

    cout << "Original sentence: " << oldSentence << endl;

    // Create a new sentence without "WAY ":

    // First, find the position where WAY occurs
    int i = oldSentence.find("WAY ");

    // Second, get the characters up to "WAY "
    string newSentence = oldSentence.substr(0, i);

    cout << "Modified sentence: " << newSentence << endl;

    // Finally, append the characters after "WAY "
    //
    // We use a special property of .substr here to make our life
    // easier.  If we don't specify the length, it just gets the
    // rest of the string.
    newSentence += oldSentence.substr(i + 4);

    cout << "Completed sentence: " << newSentence << endl;

    return 0;
}

```

output:

```

Original sentence: The quick brown fox jumped WAY over the lazy dog
Modified sentence: The quick brown fox jumped
Completed sentence: The quick brown fox jumped over the lazy dog

```

There are a couple of special cases for **substr(start, length)**. If **start** is negative, it will cause a run-time error. If **start** is past the end of the string, it will return an empty string (""). If **length** is longer than the number of characters from the start position to the end of the string, it truncates to the end of the string. If **length** is negative, then the behavior is undefined, so make sure that **length** is always non-negative. If you leave off the second argument, the number of characters from the starting position to the end of the receiver string is assumed.

Modifying a string by inserting and replacing. Finally, let's cover two other useful member functions that modify the receiver string. The first, **str1.insert(start, str2)**, inserts **str2** at position **start** within **str1**, shifting the remaining characters of **str1** over. The second, **str1.replace(start, length, str2)**, removes from **str1** a total of **length** characters starting at the position **start**, replacing them with a copy of **str2**. It is important to note that these member functions do modify the receiver string.

example program:

```

#include <string>
#include <iostream>
#include "genlib.h"

int main()
{
    int start_pos;
    string sentence = "CS 106X sucks.";
    cout << sentence << endl;

    // Insert "kind of" at position 8 in sentence
    sentence.insert(8, "kind of ");
    cout << sentence << endl;

    // Replace the 10 characters "kind of su"
    // with the string "ro" in sentence
    sentence.replace(8, 10, "ro");
    cout << sentence << endl;

    return 0;
}

```

output:

```

CS 106X sucks.
CS 106X kind of sucks.
CS 106X rocks.

```

Obtaining a C-style `char *` from a string

Remember, a C++ string is not the same thing as a C-style string (which is merely a **char*** pointer to a sequence of characters terminated by a null character `'\0'`). Although old-style C **char*** strings and C++ strings can co-exist in a program, almost all our use will be of C++ strings, since they have a much richer set of operations and are less error-prone to work with. I say "almost always" because in a few unavoidable situations, we are forced to use old-style C strings, most notably when working with file streams. We can convert a C++ string to the old-style representation using the `.c_str()` member function. One use we will see of this is to get a **char *** to pass to the `iostream::open` function.

example program:

```

#include <string>
#include <iostream>
#include <fstream>
#include "genlib.h"

int main()
{
    ifstream fs;
    string filename = "courseinfo.txt";
    string s;

    // open function requires a C-style string, must convert!
    fs.open(filename.c_str());

    if (fs.fail()) return -1; // could not open the file!

    ...

    fs.close();

    return 0;
}

```

The CS106 Library: `strutils.h`

In addition to the standard library support for strings, there are a few extensions that the CS106 libraries provide. To use these functions, the `strutils.h` library must be included:

```
#include <strutils.h>
```

IntegerToString, RealToString, StringToInteger, StringToReal: Often your programs will need to convert a string to a number or vice versa. These functions do just that, with the 'integer' functions operating on `int` and the 'real' functions on `double`.

example program:

```
#include <string>
#include <iostream>
#include "strutils.h"
#include "genlib.h"

int main()
{
    string str1 = "5.6";

    double num = StringToReal(str1);
    string str2 = IntegerToString(45);

    cout << "The original string is " << str1 << "." << endl;
    cout << "The number is " << num << "." << endl;
    cout << "The new string is " << str2 << "." << endl;

    return 0;
}
```

output:

```
The original string is 5.6.
The number is 5.6
The new string is 45.
```

Any integer or real can be safely converted to a string. However, when converting in the other direction, if you pass an improperly formatted string to convert to an integer or real, an error is raised by the conversion functions.

ConvertToLowerCase, ConvertToUpperCase: These functions take a string, and return a new string with all letters in lower or upper case respectively. These can be used to change two strings to a uniform case before comparing to allow a case-insensitive comparison.

example program:

```
#include <string>
#include <iostream>
#include "strutils.h"
#include "genlib.h"

int main()
{
    string appleFruit = "apples";
    string orangeFruit = "ORANGES";

    cout << "Do " << appleFruit << " come before "
         << orangeFruit << "? ";
    if (appleFruit < orangeFruit)
        cout << "Yes!" << endl;
    else
        cout << "Nope...." << endl;

    string lowerOrangeFruit = ConvertToLowerCase(orangeFruit);

    cout << "Do " << appleFruit << " come before "
         << lowerOrangeFruit << "? ";
    if (appleFruit < lowerOrangeFruit)
        cout << "Yes!" << endl;
    else
        cout << "Nope...." << endl;

    return 0;
}
```

output:

```
Do apples come before ORANGES?  Nope....
Do apples come before oranges?  Yes!
```