Programming in C/C++

Chapter 14 (Parts 1-2 only): C++ String Class and Character Manipulation

Kristina Shroyer

Objectives

- ***IMPORTANT:* Make sure and use techniques in the slides for homework (do *NOT* use things in the book I haven't taught)**
  - There are certain things I'm showing you to lead you into the next topic so I do NOT want you to use anything for HW #5 that is not in these slides (so don't use book functions I didn't specifically show you)

- The first few slides are a review of our "String Introduction" lecture
  - ***Remember anything in the "String Introduction" Lecture is fair game for the objective midterm #1 – BUT anything beyond that is NOT on the objective midterm #1 – I also have some review of data types material that is fair game for the objective midterm #1***
  - ***NOTHING from this lecture is on the Programming Midterm – it will instead be on Objective Midterm #2 and the Programming Final***

You should be able to describe:

- The `string` Class defined in the C++ library
  - Various Operations Provided by the class to create and initialize Strings
  - Input and Output of Strings
    - `cout, cin, getline`,
      - Phantom newline problem & `cin.ignore()`
  - `string` Processing Operations with `[]` and `length`
- Character Manipulation Methods

What is a String Constant/Literal?

- You have been using a lot of string constants/literals in your programs so far without a formal definition of what a string constant is
  - Character constants/literals only hold one character
    - Example: 'A'
  - To store a series of characters a string constant/literal needs to be used
    - Example: "Hello"
  - A string constant/literal is enclosed in double quotation marks and a character constant/literal is stored in single quotation marks

```
cout << 'H' << endl;
cout << "Hello" << endl;
```

What is a String Constant/Literal?

- Strings allow a series of characters to be stored in consecutive memory locations
  - When you use a string literal with a cout, it is utilized as a C-String, the sequence of characters are stored in adjacent memory locations with the null terminator character marking the end of the string

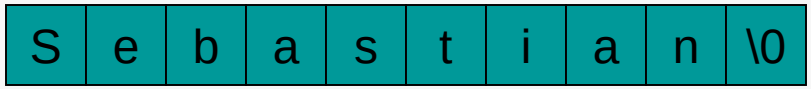
- String constants/literals can be virtually any length
  - This means that there must be some way for the program to know how long the string constant is
  - In C++ this is done by appending an extra byte to the end of string constants
    - In this last byte of the consecutive memory locations that make up the string, the null terminator also known as the null character is stored
      - The null terminator is represented by the '\0' character
      - The ASCII code for the null character is 0
      - Don't confuse this with the character '0' which is represented by the ASCII code 48

What is a String Constant?

- Example of how a string constant is stored in memory

  `"Sebastian"`

- First notice that the quotation marks are not stored with the string. They are just a way of marking the beginning and end of the string in your source code
- Notice the very last byte of the string contains the null character
- The addition of the null character to the last byte of the string means that even though the string is 9 characters long it occupies 10 bytes of memory
- C++ automatically places the null terminator at the end of a string constant



Storage of a String Constant vs. a Character Constant

- Suppose you have the constants ‘A’ and “A” in a program

‘A’ is stored as: | A | (A one byte element)

“A” is stored as: | A | \0 | (A two byte element)

- Since characters are really stored as ASCII codes this is what is actually being stored in memory

‘A’ is stored as: | 65 |

“A” is stored as: | 65 | 0 | Remember 0 is the ASCII for '\0'

- So even though some string constants look like character constants they aren’t
- There are also some characters that look like strings but aren’t
  - Example: ‘\n’ is a character constant represented by an ASCII code

Storage of String Constant vs. a Character Constant

- Important points regarding character constants and string constants
  - Characters normally occupy a single byte of memory (depends on whether ASCII or Unicode is being used)
  - Strings are consecutive sequences of characters that occupy consecutive bytes of memory
    - (always at least two bytes since the null terminator character is appended to the end of all string constants)
  - String constants have a null terminator at the end. This marks the end of a string.
  - Character constants are enclosed in single quotation marks
  - String constants are enclosed in double quotation marks
  - Escape sequences such as '\n' are stored as a single character

Working with `strings` in C++

- **There are three ways to work with strings of text in C++**

1. **C-Style strings** *(OLD SCHOOL BUT WE STILL NEED TO KNOW THE BASICS AT LEAST)*

   - C-Style strings (like the string literals we just discussed) are represented as arrays of characters terminated by a special character '\0' that represents the end of the string
     - We haven't discussed arrays yet:
       - Think of arrays as a set of contiguous memory locations where multiple data values of the same data type, like a set of character values can be stored together
   - The C Language provides a number of standard functions for working with C-Style strings, which are described in the **`<cstring>`** header file
   - In C prior to the introduction of the C++ **string** class, this is how all strings were created and manipulated (as character arrays)
   - C-Style strings are important to understand because they are still frequently used by C++ Programmers
     - Many don't like change so will always use the C-Style string over the more flexible C++ string type

Working with `strings` in C++

- **There are three ways to work with strings of text in C++**

1. **C-Style strings (continued)**
   - There are advantages and disadvantages of working with C-Strings vs. the C++ strings we're going to learn to work with today
   - Some Basic Advantages
     - Simple in that they make use of the character type and array structure
     - Lightweight and take up only the memory that they need is used properly
     - You can manipulate and copy them as raw memory
   - Some Basic Disadvantages
     - They are unforgiving and susceptible to memory bugs if not programmed properly
     - They don't use the object oriented nature of C++
     - The functions they come with aren't as easy to use as the C++ string class functions
     - Programmers must understand their underlying representation
   - We'll discuss these advantages and disadvantages in more detail when we learn C-Strings in a later class
   - We'll discuss C-Style strings in detail in a later lecture (time permitting)
     - Need to understand arrays and pointers first

Working with `strings` in C++

- **There are three ways to work with strings of text in C++**

2. **C++ strings – the C++ string class data type (this lecture)**

- Standard C++ provides a `string` class that allows a programmer to create a string type variable (an instance of a class data type)
  - This string class wraps the representation of a string into an easier to use string ***data type*** and attempts to standardize strings – the programmer uses a string by creating an instance and doesn't have to worry about the details on how the class works
  - The string class provides many services to the outside code using the class
    - Some of the services include:
      - functions for declaring, creating, manipulating, and initializing a string
    - Advantage: The string class takes care of any memory allocation you might need when manipulating the string (unlike C-Strings)
      - With C-Strings the programmer is responsible for memory allocation
  - The C++ string class does not **necessarily** represent a string by appending the '/0' character to the end of a string like the C-String method of representing a string does
    - Therefore although the C++ string class may represent strings internally by terminating them with a null character there is no guarantee that is the way they are represented
    - The C++ string class handles the processing of the string internally within the class, allowing the programmer to not need to worry about exactly how the string is represented or how the operations done on the string are performed

Working with `strings` in C++

- **There are three ways to work with strings of text in C++**
  3. **Nonstandard Strings**
     - Some programmers might create their own string data type in C++ and use that to represent strings

  - Reasons C++ Programmers might not use C++ strings
    - Some programmers learned C++ before the addition of this class and either aren't aware of it or don't want to change
    - Some programmers like to control exactly what memory their string consumes
    - Others want their strings to have different or more behaviors than the C++ string class has provided and have therefore defined their own string types
    - Most Common Reason: Some development frameworks and operating systems have their own way of representing strings

Basics of Using the C++ `string` Class

- The first step in using the string class is to **`#include`** the string header file in your program (as usual the include may differ depending on the compiler).
  - This is accomplished with the following preprocessor directive:
    - `#include <string>`
- The next step is to declare a string object and create a variable name to reference that string object in memory
    - `string movieTitle;`
- Now you can assign a string literal to the **`movieTitle`** with the assignment operator
    - `movieTitle = "The Matrix";`
- Now you can print the contents of movieTitle on the screen with the **`cout`** object.
    - `cout << "The movie title is " << movieTitle << endl;`

Basics of Using the C++ `string` Class

Example: (firstStringEx.cpp – Example 1)

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
   string movieTitle;
   movieTitle = "The Matrix";
   cout << "The movie title is " << movieTitle << "." << endl;
   system("PAUSE");
   return 0;
}
```

The C++ `string` Class - Basics

- With the introduction of the new ANSI/ISO C++ standard, a **class data type** named **`string`** was provided as part of the standard C++ library.

- A class is a definition for a **class (user defined) data type**.
  - Just like a primitive data type, a class data type is defined by a set of valid data values and the operations that can be performed on those data values
  - A class data type and a built in (primitive) data type are constructed differently
  - Remember our definition for a data type: a set of values AND the operations that can be performed on those values
  - A ***built in (primitive) data type***
    - is provided as an integral part of the compiler (programmer did not need to write extra code)
    - Provide standard operations like +,-,/,*…etc.
    - Storage areas are referred to as **variables**
      - Storage areas only need to be large enough to hold one value of the primitive type
  - A ***class data type***
    - is constructed by a programmer using C++ code (even if it's in the library it was programmer created)
    - The set of data can be more than one value
      - Address Book Entry
    - Most of the operations are programmer defined functions rather than built in operations
    - Storage areas are referred to as **objects (or instances)**
      - Storage areas for objects must be much larger, there has to be room for all of the data in the object…class data types can have more than one type of data, think of an address book data type

- Programmers can use class data types without understanding the details of their implementation (this is true for primitive types as well but the complexity of class data types really drives the point home)
  - This is Abstraction, to use the String class we only need to know what data it holds and what its functions do, we don't need to know the details of the class's implementation

The C++ `string` Class - Basics

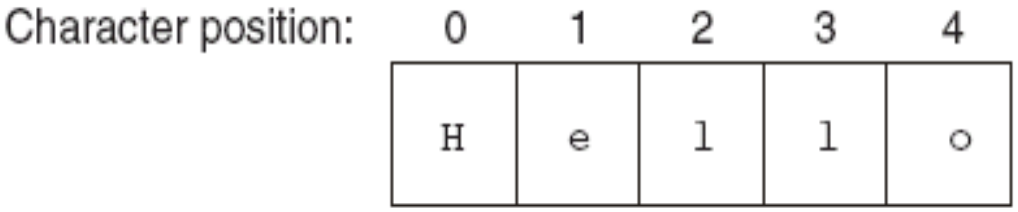
- The C++ **`string`** class provides a greatly expanded set of class functions/operations:
  - easy insertion and removal of characters from a string
  - automatic string expansion whenever a string's original capacity is exceeded
  - string contraction when characters are removed from the string
  - range checking to detect invalid character positions
  - a variety of other operations

- The values permitted by the string class (so the values that can be held in instances of string data types) are referred to as string literals
  - A **string literal**, as we have already seen, is any sequence of characters enclosed in ***double quotation marks***
  - As has also been noted, a string literal is also referred to as a string value, a string constant, and more conventionally, simply as a string
  - Examples of strings are **`"This is a string"`**, **`"Hello World!"`**, and **`"xyz 123 *!#@&"`** .
    - The double quotation marks are used to mark the beginning and ending points of the string and are never stored with the string.

The C++ `string` Class - Basics

- The Figure below shows the programming representation of the C++ string **`"Hello"`**
- Position numbers (index numbers in strings)
  - By convention, the first character in a string is always designated as position 0. This position value is also referred to as both the character's index value and its offset value.
- Note the string in the diagram is not terminated by the null character.
  - This doesn't necessarily mean it isn't represented that way
  - Remember the representation (implementation) of the C++ **`string`** in the C++ string class (different from a C-String) is hidden from the programmer

FIGURE 7.1 *The Storage of a* `string` *as a Sequence of Characters*



Creating C++ `string` Class Instances

- What makes class data types useful is:
  - We can use them without needed to know how they are implemented (don't need to worry about terminating null character)
  - There are several functions we can use to manipulate our string instances and we can also use these without needing to know the details of how they work
- Various functions/methods are provided by the **`string`** class for declaring, creating, and initializing a string.
  - These functions/methods are the operations that can be performed on our **`string`** data type
- When you create a new string a member function of the **`string`** class called a ***constructor*** is executed. The constructor creates and initializes your string type.
- We have already seen one way to create a string object:

```
string movieTitle = "The Matrix";
```

- A table (7.1 – in the old book) in your book lists the methods (Constructors) provided by the string class for creating and initializing a **`string`**.

`string` Class Member Functions

TABLE 7.1 `string` *Class Constructors (Required Header File Is* `string`*)*

Constructor	Description	Examples

`string objectName = value`	Creates and initializes a string object to `value`, which can be a string literal, previously declared string object, or an expression containing both string literals and string objects	`string str1 = "Good Morning";`
		`string str2 = str1;`
		`string str3 = str1 + str2;`

`string objectName(string-value)`	Produces the same initialization as above	`string str1("Hot");` `string str1(str1 + " Dog");`

The + is the concatenation operator which combines two strings into one

`string` Class Member Functions

Constructor	Description	Examples
`string objectName(str, n)`	Creates and initializes a string object with a substring of string object `str`, starting at index position `n` of `str`	`string str1(str2, 5)` If `str2` contains the string `Good Morning`, then `str1` becomes the string `Morning`
`string objectName(str, n, p)`	Creates and initializes a string object with a substring of string object `str`, starting at index position `n` of `str` and containing `p` characters.	`string str1(str2, 5,2)` If `str2` contains the string `Good Morning`, then `str1` becomes the string `Mo`

`string` Class Member Functions

Constructor	Description	Examples
`string objectName(n, char)`	Creates and initializes a string object and initializes it with `n` copies of `char`.	`string str1(5,'*')` `This makes str1 = "******"`
`string objectName`	Creates and initializes a string object to represent an empty character sequence. Same as `string objectName = "";` The length of the string is `0`.	`string message;`

`string` Class Member Functions

String creation: Example – (creatingStrings.cpp – Example 2) – Prog. 7.1

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
   string str1; // an empty string
   string str2("Good Morning");
   string str3 = "Hot Dog";
   string str4(str3);
   string str5(str4, 4);
   string str6 = "linear";
   string str7(str6, 3, 3);
   cout << "str1 is: " << str1 << endl;
   cout << "str2 is: " << str2 << endl;
   cout << "str3 is: " << str3 << endl;
   cout << "str4 is: " << str4 << endl;
   cout << "str5 is: " << str5 << endl;
   cout << "str6 is: " << str6 << endl;
   cout << "str7 is: " << str7 << endl;

   system("PAUSE");
   return 0;
}
```

`string` Input and Output

- In addition to functions listed in Table 7.1, strings can be:
  - Input from the keyboard
  - Displayed on the screen
  - Manipulated in other ways with other methods (operations of the string class)
- **`cout`**: General purpose screen output
- Using **`cout`** with a **`string`** (Example 3 – stringOutput.cpp)

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
   string name1 = "Mark Twain";
   string name2 = "Samuel Clemens";
   name2 = name1; //Now name2's value is Mark Twain
   cout << "name1 is " << name1 << "\nname2 is also " << name2 << endl;
   system("PAUSE");
   return 0;
}
```

`string` Input with `cin`

- Although it is possible to use **`cin`** with the >> operator to input strings, it can cause problems you need to be aware of.
  - When **`cin`** reads data it passes over and ignores any leading whitespace characters (spaces, tabs, line breaks).
  - Once **`cin`** comes to the first non-blank (non whitespace) character it starts reading data
  - **`cin`** continues to read data until it comes to the next whitespace character
    - Once it reaches the next whitespace character it STOPS reading data
    - This can be problematic when reading string data

`string` Input with `cin` (Example)

- **Example (stringWithcin.cpp – Example 4):**

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
   string name1;
   cout << "Enter a first and last name.\n";
   cin >> name1;  //enter a first and last name to see the problem
                  //with cin and strings
   cout << "name1 is only read up to the whitespace.  name1 is: " << name1 <<
    endl;

   system("PAUSE");
   return 0;
}
```

- **Program Output**
- Enter a first and last name.
- Mark Twain
- name1 is only read up to the whitespace. name1 is: Mark
- Press any key to continue .

`string` Input

- To solve this problem C++ provides a standard function (one in the standard library) called **`getline`**
  - **`getline`** will read in an **entire line** as input, including leading and embedded spaces, and store it in a string
    - Note that **`getline`** is not a member function of the **`string`** class but is designed to work with strings (part of standard C++ library)
  - **`getline(cin, str)`**:General purpose terminal input that inputs all characters entered into the string named **`str`** and **stops accepting characters when it receives a newline character** (**`\n`**)
  - **Example**: **`getline(cin, message)`**
    - Continuously accepts and stores characters entered at terminal until Enter key is pressed.
      - Pressing Enter key generates newline character, **`'\n'`**
      - All characters except newline are stored in string named **`message`**

string Input (getline Example)

- **Example (getline1Ex.cpp – Example 5)**

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
   string name1;
   cout << "Enter a first and last name.\n";
   getline(cin, name1);  //using getline to read in a string
   cout << "name1 contains the first and last name.  \nname1 is: " << name1 <<
    endl;

   system("PAUSE");
   return 0;
}
```

- Output:

```
Enter a first and last name.
Mark Twain
name1 contains the first and last name.
name1 is: Mark Twain
Press any key to continue . . .
```

`string` Input (another `cin` problem)

- Another good reason for using **`getline`** for reading in characters:
  - When reading in strings **`getline`** not only allows embedded blanks to be read but it also offers a protection **`cin`** does not
  - If **`cin`** is used to read in a string and the user embeds a blank character in the input any remaining characters will be left in the ***keyboard buffer***
    - Think of the keyboard buffer as a special place in memory where characters typed at the keyboard are stored before they are read into the program
    - The next read after **`cin`** does this will try to use those leftover characters
    - The next example illustrates this problem

`string` Input (another `cin` problem)

- **Example: (cinBufferEx.cpp – Example 6)**

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
   string name1;
   string city1;
   cout << "Enter a first and last name:\n";
   cin >> name1;  //enter a first and last name to see the problem
                  //with cin and strings
   cout << "Enter the city you live in:\n";
   cin >> city1;
   cout << "Hello. " << name1 << endl;
   cout << "You live in " << city1 << endl;

   system("PAUSE");
   return 0;
}
```

`string` Input (another `cin` problem)

- Output:

```
Enter a first and last name:
John Doe
Enter the city you live in:
Hello. John
You live in Doe
Press any key to continue . . .
```

- Notice that the user was never given the opportunity to enter a city
  - In the first input, when **`cin`** came to the space between John and Doe it stopped reading, storing just John as the value of **`name`**
  - In the second input statement, **`cin`** used the leftover characters it found in the keyboard buffer and stored Doe as the value of **`city`**

`string` Input

- **General form of `getline()` method:**

```
getline(cin, strObj, terminatingChar)
```

  - `strObj`: a string variable name
  - `terminatingChar`: an optional character constant or variable specifying the terminating character

  - If you don't enter a terminating character `'\n'` is used
  - Accepts all characters entered at the keyboard, including newline (if newline isn't the terminating character), until the terminating character is entered
  - Note that the terminating character will not be stored as part of the string
  - Once you enter the terminating character You must still hit "enter" to stop reading characters

- **Example:**
  - `getline(cin, message, '!')`

string Input

- **Example (getLineEx2.cpp – Example 7)**

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
   string message;
   cout << "Enter a message. Enter ! to signify the end of the message.\n";
   getline(cin, message, '!');  //using getline to read in a string and terminate
                                //with something other than newline
   cout << "The message is: \n" << message << endl;

   system("PAUSE");
   return 0;
}
```

- Output:

```
Enter a message. Enter ! to signify the end of the message.
hello!  //message entered
The message is:
hello  //message output
Press any key to continue . . .
```

Caution: The Phantom Newline Character

- **Unexpected results occur when:**
  - `cin` input stream and `getline()` method are used together to accept data
  - and when `cin` input stream is used to accept individual characters

- **Why do problems occur?**
  - When Enter is pressed to complete input using `cin`, `cin` accepts the input but leaves the `'\n'` character in the ***keyboard buffer***
  - If `getline()` is then called, `getline()` reads the `'\n'` character on the buffer and immediately terminates

- **Example: Program 8 on next slide (phantomNewline.cpp)**
  - When `value` is entered and Enter key is pressed, `cin` accepts `value` but leaves the `'\n'` in the buffer
  - `getline()` picks up the code for the Enter key as the next character and terminates further input

Caution: The Phantom Newline Character

- **Example (phantomNewline.cpp – Example 8)**

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
   int value;
   string message;
   cout << "Enter a number:\n";
   cin >> value;
   cout << "The number entered is " << value << endl;

   cout << "Enter a message:\n";
   getline(cin, message);
   cout << "The message is: \n" << message << endl;

   system("PAUSE");
   return 0;
}
```

- What happens if the `getline()` is replaced with `cin`?
  - Why is this not the solution to the problem?

Caution: The Phantom Newline Character

- Solutions to the “phantom” Enter key problem
  1. Do not mix **`cin`** with **`getline()`** inputs in the same program
  2. Follow the **`cin`** input with the call to **`cin.ignore()`**
  3. Accept the Enter key into a character variable and then ignore it

- Preferred solution is the first option
  - We'll use the first or second option in class (sometime we can't avoid mixing cin and getline in this class)
    - (the third option is more old school so let's stay away from that one)

cin.ignore()

- **cin.ignore(numChar,termination char)**
  - **cin.ignore()** takes two parameters
    - The first parameter is the number of characters to ignore
    - The second parameter is the termination character
    - So if you write:
      - **cin.ignore(80, '\n');**
        - up to 80 characters will be thrown away or characters will be thrown away until a newline character is found. The newline is then thrown away and the ignore() statement ends.

- **cin.ignore()** can be used with no arguments (***think of default parameters and overloading***), if you do this **cin** will only skip the very next character

The Phantom Newline Character – `cin.ignore()`

- **Example: (phantomNewline2.cpp – Example 9)**

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
   int value;
   string message;
   cout << "Enter a number:\n";
   cin >> value;
   cin.ignore();
   cout << "The number entered is " << value << endl;

   cout << "Enter a message:\n";
   getline(cin, message);
   cout << "The message is: \n" << message << endl;

   system("PAUSE");
   return 0;
}
```

Using `string` class Member Functions

- Functions for manipulating strings (Table 14.3 in Book):
  - So these are the operations (functions) that can be used on the set of values that define our class data type `string`
  - Most commonly used `string` class method is `length()` which returns the number of characters in the string
- The ***dot operator*** is used to access a class function (objectName (variable).functionName(); :
  - `stringvariable.length();`
    - **Basically this is saying execute the length() function on this string object (you can think of it as a variable if that's easier)**
- Assume the following string exists in a program:
  - `string town = "Charelston";`
- The following statement in the same program would assign the value 10 to the variable x
  - `int x;`
  - `x = town.length();`
- The `.length()` part of this line of code calls the string class member function length for the `string` object town that we created
- Note that spaces are counted as characters when determining the length of the string

String Processing – `length()` Example

- **Example (strLength.cpp – Example 10)**

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
   string town;

   cout << "Where do you live? ";
   getline(cin, town);
   cout << "Your town has " << town.length() << " characters." << endl;

   system("PAUSE");
   return 0;
}
```

- The string class also has `.size()` member function that does the same thing as the `.length()` member function

`string` Processing with Operators

- Many C++ operators (+, -, >, < etc) work with `string` objects
- This is different from Java…due to ***operator overloading***, you're going to see the == and some of the other relational operators that didn't work with Java strings will work with C++ strings
- **The assignment operator =** (we've already seen the first two uses)

1. Can be used to assign one string object's value to another string object
2. Can be used to assign a string literal to a string object
3. Can be used to assign a single char to a string object
   - **You can't assign a char literal to a string object, but you can assign a variable stored in a char to a string object**

Example (charEx.cpp – Example 11)

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
   //you can't do this....
   //string str = 'a';

   //but you can do this
   string str;
   char ch = 'a';
   str = ch;

   cout << "str contains " << str << endl;
   system("PAUSE");
   return 0;
}
```

When I assign HW #5, you'll want to come back to this slide and closely look at the next 10 or so slides

`string` Processing – Comparison Operators

- String expressions may be compared for equality using standard relational operators
  - This is because these operators are ***overloaded*** in the `string` class
  - (`<` , `>` , `!=` , `==` , `<=` , `<=`)

- String characters stored in binary using ASCII or Unicode code relate to each other as follows:
  - A blank precedes (is less than) all letters and numbers
  - Uppercase Letters are stored in order from A to Z
  - Lowercase Letters are stored in order from a to z
  - Digits stored in order from 0 to 9
  - Digits come before uppercase characters, which are followed by lowercase characters
  - **blank < digits < uppercase < lowercase**

`string` Processing – Comparison Operators

- **Procedure for comparing strings:**
  - Individual characters are compared a pair at a time
    - If no differences, the strings are equal
    - Otherwise, the string with the first **lower** character in ASCII is considered the smaller string

- **Examples:**
  - **`"Hello"` is greater than `"Good Bye"`** because the first `H` in `Hello` is greater than the first `G` in `Good Bye`
  - **`"Hello"` is less than `"hello"`** because the first `H` in `Hello` is less than the first `h` in `hello`

- Any of the two relational operators can be used to compare two strings
- Example:

```
string name1 = "Mary";
string name2 = "Mark";
```

  - **`name1 > name2` //true**
  - **`name1 <= name2` //false**
  - **`name1 != name2` //true**

- The value in the name Mary is greater than the value in the name Mark. This is because the first three characters in name1 have the same ASCII values as the first three characters in name2 but the 'y' in the fourth position of "Mary" has a greater ASCII value than the 'k' in the fourth position of "Mark"

`string` Processing – Comparison Operators

- **Example: (compareStr.cpp – Example 12)**

```
/*this is a program for comparing two strings*/
#include <iostream>
#include <string>
using namespace std;

int main()
{
   string string1;
   string string2;

   cout << "Enter the first string to compare: ";
   getline(cin, string1);
   cout << "Enter the second string to compare: ";
   getline(cin, string2);

   if(string1 > string2)
      cout << string1 << " is greater than " << string2 << endl;

   else if(string2 > string1)
      cout << string2 << " is greater than " << string1 << endl;

   else  //otherwise the strings are equal
      cout << string1 << " and " << string2 << " are equal." << endl;

   system("PAUSE");
   return 0;
}
```

`string` Processing – Concatenation (+) Operator

- **When the + operator is applied to two strings it concatenates them, or joins them together – *you can add a single char to a string as well with the + operator* (*HINT for HW #5*)**

- **Example**: (strConcat.cpp – Example 13)

```
/*string concatenation program*/
#include <iostream>
#include <string>

using namespace std;

int main()
{
   string greeting1 = "Hello ";
   string greeting2;
   string name1 = "World";
   string name2 = "People";

   //the following demonstrate how string concatenation works
   greeting2 = greeting1 + name1;  //now holds "Hello World"
   greeting1 += name2; //now holds "Hello People"

   cout << greeting2 << endl;
   cout << greeting1 << endl;

   system("PAUSE");
   return 0;
}
```

`string` Processing – Subscript Operator ([ ])

- **The subscript operator accesses one character in a string**
  - **Notice this operator does not check if a valid index was used**
  - **Also NOTE: instead of just saying `cout << str[i]` we could save `str[i]` in a char type variable**
  - **For HW #5, think of using the subscript operator with the `+` to add a char to a string (`str = str + str[i]`)**

- ***DO NOT COPY THIS FOR LOOP EXACTLY FOR HW #5, HW #5 does NOT ask you to just `cout` but asks you to actually CHANGE the string (so you need to change this a bit)***
- **Example**: (strSubscript.cpp – Example 14)

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
   string str = "Hello";

   //use subscript to change the first letter of the string to J
   str[0] = 'J';

   //use the subscript to print the new first letter of the string
   cout << "The first letter is now " << str[0] << endl;

   //use the subscript operator to print the string backwards
   cout << "The string reversed is ";
   for(int i = str.length()-1; i >=0; i--)
   {
      cout << str[i];
   }

   cout << endl;
   system("PAUSE");
   return 0;
}
```

This means strings ARE mutable (changeable) in C++. Remember strings were immutable (unchangeable) in Java.

Hint for HW #5 – do NOT copy exactly You need to think and modify it to work for HW #5

You need to combine this with the + and or += operator we just learned

You may NOT use library functions for HW #5 – we are going to use overloaded operators instead

`for` loop to manipulate strings – `[]` operator

- THIS IS REQUIRED TO BE USED IN HW #5 – DO NOT USE the at function, use only the subscript operator [] in HW #5

- This is a loop you can use to walk through each char in your string and visit and manipulate each char
- Assume for this example you have a variable string str; with a string in it

```
for (int counter = 0; counter < str.length(); counter++)
{
    //process your string accordingly here
    //- the loop is always the same (just change the name of your
    //string)
    //the code inside the loop changes
    //Example…if you wanted to change each char to a *
   //you would say str[counter] = '*';
}
```

Character Manipulation Methods

- So we can manipulate entire strings with string methods and we can use the subscript `[]` operator to extract a char from a string
  - Now we want to know some functions that we can use to manipulate characters
  - Important Point: string manipulation methods only manipulate strings and ***character manipulation methods only manipulate characters***
    - Don't try to manipulate strings with character methods and vice versa – don't forget data types
- C++ language provides a variety of useful ***character*** functions that operate on single characters (listed in Table 7.4)
- Function declarations (prototypes) for these functions are contained in header files **`string`** and **`cctype`**
  - So you will need this additional header to use the character function
  - **`#include <cctype>`**
- Headers file must be included in any program that uses these functions

Character Manipulation Functions-Test `char` Values

- The C++ library provides several functions that allow you to test the value of a ***character***
  - ***These work on CHARACTERS not on STRINGS***
  - These functions test a single **`int`** argument (a character is really numeric) and return **`true`** or **`false`**.
    - Remember these functions really return an **int** value. An non-zero return value indicates true and a zero return value indicates false.
    - The integer argument for these functions is the ASCII code of the character
  - One of the character testing functions is **`isupper`**
    - **`isupper`** returns true (any value but 0) if the character passed into it as an argument is an uppercase value and false (0) if it is not
  - Example: The following program demonstrates the **`isupper`** function
  - So how to you test the value of one character in a string?
    - You need to use the subscript operator to retrieve one character
    - Then test the character you retrieved
    - Keep in mind these functions work on characters NOT on strings

Character Manipulation Functions-Test `char` Values

- The following program uses the **`isupper`** function to determine if the character passed as an argument is an uppercase value
- What if I wanted to test if the first character in a string was uppercase? How would I do it?

- Example: (isUpper.cpp – Example 15)

```
#include <iostream>
#include <cctype>
using namespace std;

int main()
{
   char letter;
   cout << "Enter a letter: ";
   cin >> letter;

   if(isupper(letter))
      cout << "The letter " << letter << " is uppercase." << endl;

   else
      cout <<"The letter " << letter << " is NOT uppercase." << endl;

   system("PAUSE");
   return 0;
}
```

Character Manipulation Methods-Character Testing

- Other character testing methods (**remember to use the header `#include <cctype>`**)
- Each of these functions take an int as an argument and return true or false
  - **`isalpha`** – returns true if the argument is a letter of the alphabet
  - **`isalnum`** – returns true if the argument is a letter of the alphabet OR a digit
  - **`isdigit`** – returns true if the argument is a digit from 0 to 9
  - **`islower`** – returns true if the argument is a lowercase letter
  - **`isprint`** – returns true if the argument is a printable character (including a space)
  - **`ispunct`** – returns true if the argument is a printable character other than a digit, letter, or space
  - **`isupper`** – returns true if the argument is an uppercase letter
  - **`isspace`** – returns true if the argument is a whitespace character
    (whitespace characters: space ‘’ ,vertical tab ’\v’, newline ‘\n’, tab ‘\t’)

Character Manipulation Methods-Char Case Conversion

- The C++ library provides functions for **converting** a character to upper or lower case
  - `toupper` – returns the uppercase equivalent of its argument
  - `tolower` – returns the lowercase equivalent of its argument

- Both of these functions accept an integer representing the ASCII code of a character to be converted and return an integer representing the ASCII code of the uppercase or lowercase equivalent.
  - Note it says the functions accept an integer, but an char is an int so it really accepts a char as well, just remember it returns and int so you'll need to use a cast

- If the conversion cannot be made or is unnecessary, the function simply returns its argument
  - For example if the argument to `toupper` is not a lowercase letter, then `toupper` simply returns its argument unchanged

Character Manipulation Methods-Char Case Conversion

- These functions are in the header file cctype so you must have the preprocessor command: **`#include <cctype>`** in your program
- Prototypes for the **`toupper`** and **`tolower`** functions:
  - `int toupper(int ch);`
  - `int tolower(int ch);`
- The fact that these functions return an integer means that the following statement will print out the ASCII code of 'A' rather than printing out 'A' itself:
  - `cout << toupper('a');  //this will print 65`
- To get it to print the character instead of the ASCII code you can **cast** the return value to char
  - Below is a C-Style cast
  - `cout << (char)(toupper('a'));  //this will print 'A'`
  - Below is the equivalent C++ style compile time cast
  - `cout << char(toupper('a'));  //this will print 'A'`
  - Below is the equivalent C++ style run time cast
  - `cout << staticCast<char>(toupper('a'));//this will print 'A'`

Character Manipulation Methods-Char Case Conversion

- **Example: (toLower.cpp – Example 16)**

```
/*this program converts an uppercase letter to lowercase and uses three
 different casts to print the result to the screen three times*/
#include <iostream>
#include <cctype>
using namespace std;

int main()
{
   char letter;
   cout << "Enter an uppercase letter: ";
   cin >> letter;

   //c-style cast
   cout << "The lowercase equivalent using a c-style cast is "
        << (char)(tolower(letter)) << endl;

   //c++-style compile time cast
   cout << "The lowercase equivalent using a c++-style compile time cast is "
        << char(tolower(letter)) << endl;

   //c++-style run time cast
   cout << "The lowercase equivalent using a c++-style run time cast is "
        << static_cast<char>(tolower(letter)) << endl;

   system("PAUSE");
   return 0;
}
```

Character I/O

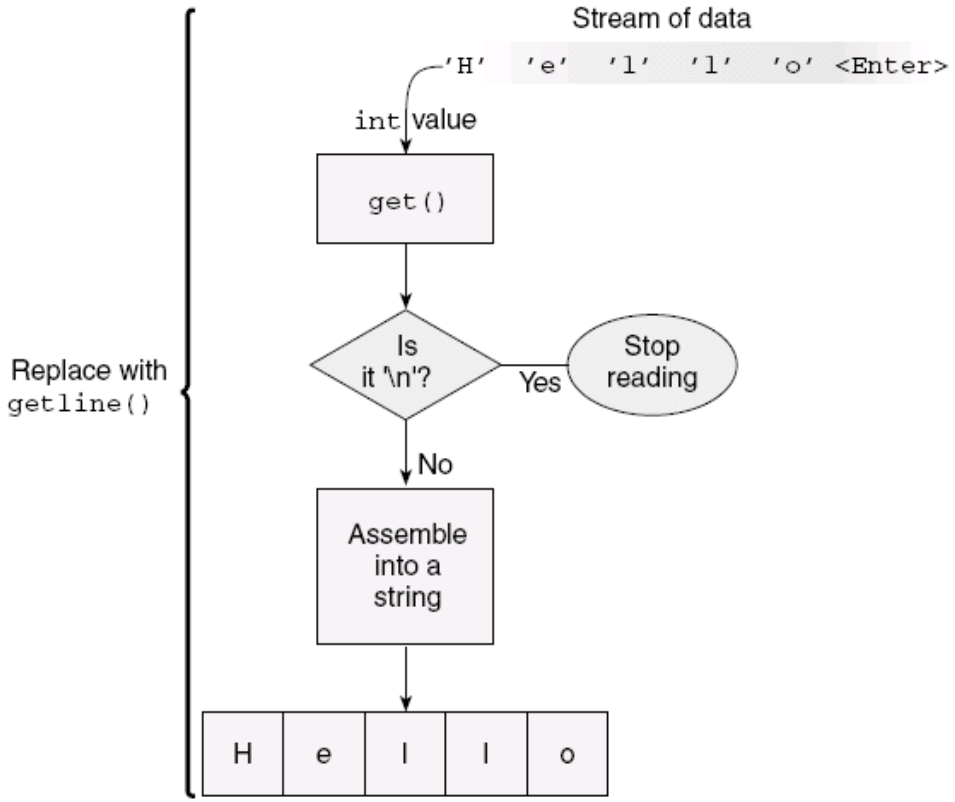
- It is important to understand how `cin` and `getline()` accept data entered from the keyboard, how they send the data to the program and how the program must react to process the data correctly.

- Entry of all data from keyboard, whether a string or a number, is done one character at a time
  - Entry of string `Hello` consists of pressing keys `H`, `e`, `l`, `l`, `o`, and the Enter Key (as in Figure 7.10)
  - At a very fundamental level, all input (as well as output), is done on a character-by-character basis.

- All of C++'s higher-level I/O methods and streams are based on lower-level character I/O

Character I/O

FIGURE 7.10 *Accepting Keyboard Entered Characters*



Character I/O

- The more elemental character methods, which can also be used directly by a programmer, are listed in Table 7.5 in the book

- The `cin.get(charVar)` function reads the next character in the input stream and assigns it to the function's character variable.
  - For example, a statement, such as:

    ```
    cin.get(nextChar);
    ```

    causes the next character entered at the keyboard to be stored in the character variable `nextChar`.

  - This function is useful for inputting and checking individual characters before they are assigned to a complete string or other C++ data type.

- `cout.put(charExp)` is the character output function corresponding to `get(charVar)`.
  - This function expects a single character argument and displays the character passed to it on the terminal.
  - For example, the statement `cout.put('A')` causes the letter A to be displayed on the screen.

- `cin.ignore()` function
  - We've already seen this function.
  - This function permits skipping over input until a designated character, such as `'\n'` is encountered. For example, the statement `cin.ignore(80, '\n')` will skip up to a maximum of the next 80 characters, or stop the skipping if the newline character is encountered

Character I/O

- Example (getPutEx.cpp – Example 17)

```
/*this program uses cin.get and cout.put*/
#include <iostream>
#include <cctype>
using namespace std;

int main()
{
   char ch1;
   char ch2 = 'B';
   char ch3 = '!';
   cout << "Enter a character: ";
   cin.get(ch1);

   cout.put(ch1);
   cout.put(ch2);
   cout.put(ch3);
   cout.put('\n');

   system("PAUSE");
   return 0;
}
```

Character I/O

- The **`cin.peek()`** function returns the next character on the stream, but does not remove it from the stream's buffer
  - For example, the expression **`cin.peek(nextChar)`** returns the next character input by the keyboard, but leaves it in the buffer.
    - This is sometimes useful for "peeking" ahead and seeing what the next character is, while leaving it in place for the next input.
- The **`putback()`** function places a character back on the stream so that it will be the next character read.
  - The argument passed to **`putback(charExp)`** can be any character expression that evaluates to a legitimate character value, and need not be the last input character.

The Phantom Newline Revisited

- Undesired results can occur when characters are input using the `get()` character method
  - Similar to the problem with the `getline()` method

- Two ways to avoid this:
  - Follow `cin.get()` input with the call `cin.ignore()`
  - Accept the Enter key into a character variable and then don't use it further

The Phantom Newline Revisited- Example

- Example: (phantomGet.cpp – Example 18)

```
/*example of phantom newline problem with cin.get()*/
#include <iostream>
#include <cctype>
using namespace std;

int main()
{
   char letter;
   char letter2;
   cout << "Enter a letter: ";
   cin.get(letter);

   cout << "The ASCII code for the letter is " << (int)letter << endl;

   cout << "Enter a second letter: ";
   cin.get(letter2);
   cout << "The ASCII code for the letter is " << (int)letter2 << endl;

   system("PAUSE");
   return 0;
}
```

The Phantom Newline Revisited- Example

Output:

```
Enter a letter: m
The ASCII code for the letter is 109
Enter a second letter: The ASCII code for the letter is 10
Press any key to continue . . .
```

- In typing m, two keys are usually pressed: the m key and the Enter key.
  - these two characters will be stored in a buffer immediately after they are pressed

- The first key pressed, **`m`** in this case, is taken from the buffer and stored in **`letter1`**.
  - This, however, still leaves the code for the Enter key in the buffer.
  - Thus, a subsequent call to **`get()`** for a character input will automatically pick up the code for the Enter key as the next character. (The newline character has an ASCII code of 10)

The Phantom Newline Revisited- Example

- In summary this is why the program didn't work properly
  - In entering m in response to the first prompt, the Enter key is also pressed. From a character stand point this represents the entry of two distinct characters.
  - The first character is m, which is coded and stored as the integer 109.
  - The second character also gets stored in the buffer with the numerical code for the Enter key.
  - The second call to `get()` picks up this code immediately, without waiting for any additional key to be pressed. The last `cout` stream displays the code for this key

The Phantom Newline Revisited- Example

- Every key has a numerical code, including the Enter, Spacebar, Escape, and Control keys.
  - These keys generally have no effect when entering numbers, because the input methods ignore them as leading or trailing input with numerical data.
  - Nor do these keys affect the entry of a single character requested as the first user data to be input
  - Only when a character is requested after the user has already input some other data does the usually invisible Enter key become noticeable.

The Phantom Newline Revisited- Example

- Use of `cin.ignore()` to fix the problem
- Example: (phantomGetFix.cpp – Example 19)

```
/*example of phantom newline problem with cin.get()*/
#include <iostream>
#include <cctype>
using namespace std;

int main()
{
   char letter;
   char letter2;
   cout << "Enter a letter: ";
   cin.get(letter);
   cin.ignore();

   cout << "The ASCII code for the letter is " << (int)letter << endl;

   cout << "Enter a second letter: ";
   cin.get(letter2);
   cout << "The ASCII code for the letter is " << (int)letter2 << endl;

   system("PAUSE");
   return 0;
}
```