Programming Fundamentals

CC-111

Week-04-05

Dr. Muhammad Nadeem Majeed

Associate Professor
Department of Data Science
University of the Punjab, Lahore

Certifications:

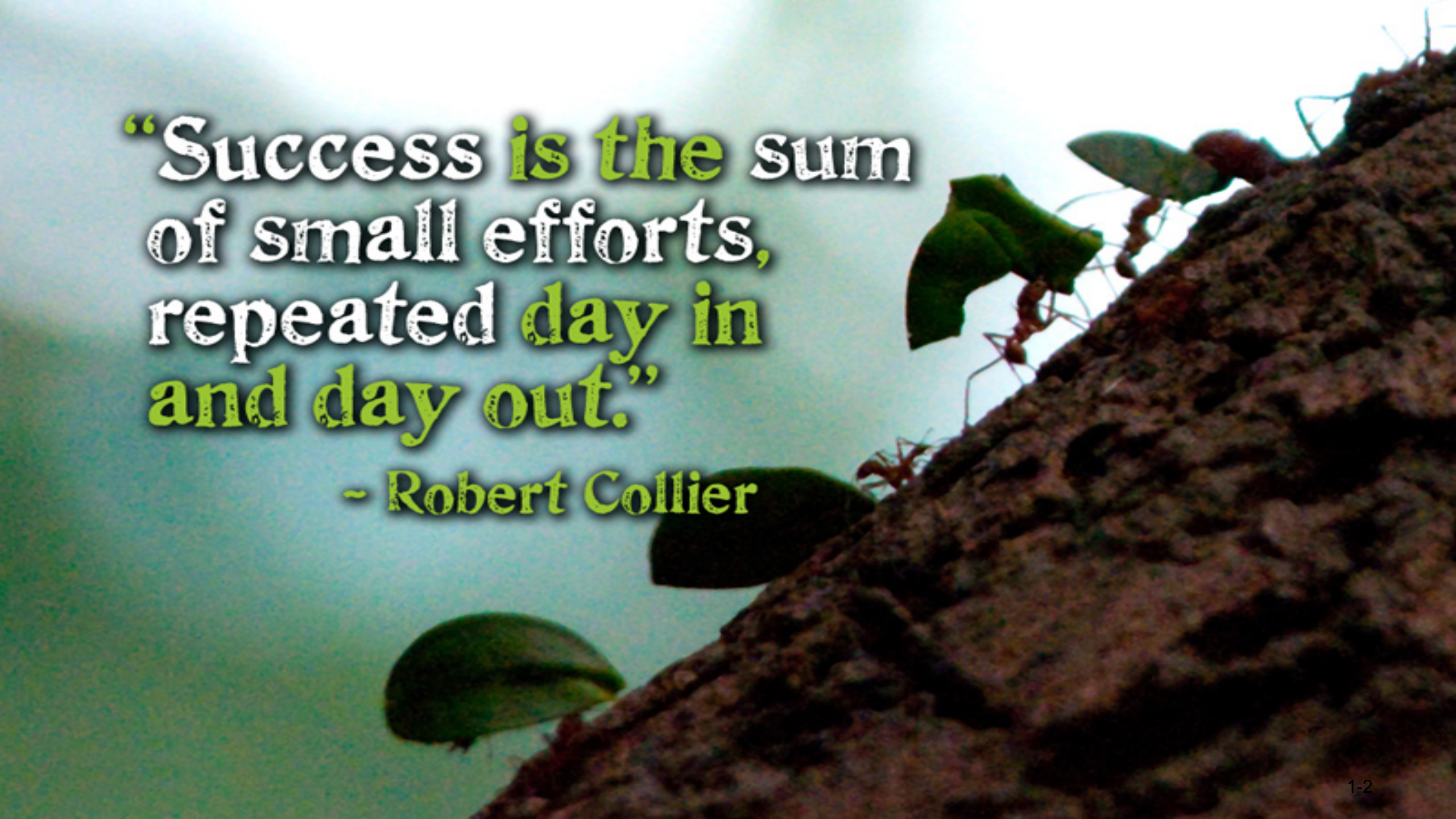
Project Management Professional (PMP)®

PRINCE2 Agile Practitioner

Professional Scrum Master (PSM)

Certified Lean Six Sigma Green Belt (CSSC)

ITIL Certified


"Success is the sum of small efforts, repeated day in and day out."
~ Robert Collier

Topics

3.1 The `cin` Object

3.2 Mathematical Expressions

3.3 Data Type Conversion and Type Casting

3.4 Overflow and Underflow

3.5 Named Constants

Topics (continued)

3.6 Multiple and Combined Assignment

3.7 Formatting Output

3.8 Working with Characters and Strings

3.9 More Mathematical Library Functions

3.10 Random Numbers

3.1 The `cin` Object

- **`cin`** is the standard input object
- Like **`cout`**, requires **`iostream`** file
- Used to read input from keyboard
- Often used with **`cout`** to display a user prompt first
- Data is retrieved from **`cin`** with **`>>`**, the stream extraction operator
- Input data is stored in one or more variables

The `cin` Object

- User input goes from keyboard to the input buffer, where it is stored as characters
- **`cin`** converts the data to the type that matches the variable

```
int height;
cout << "How tall is the room? ";
cin  >> height;
```

The `cin` Object

- Can be used to input multiple values

```
cin >> height >> width;
```

- Multiple values from keyboard must be separated by spaces or [Enter]
- Must press [Enter] after typing last value
- Multiple values need not all be of the same type
- Order is important; first value entered is stored in first variable, etc.

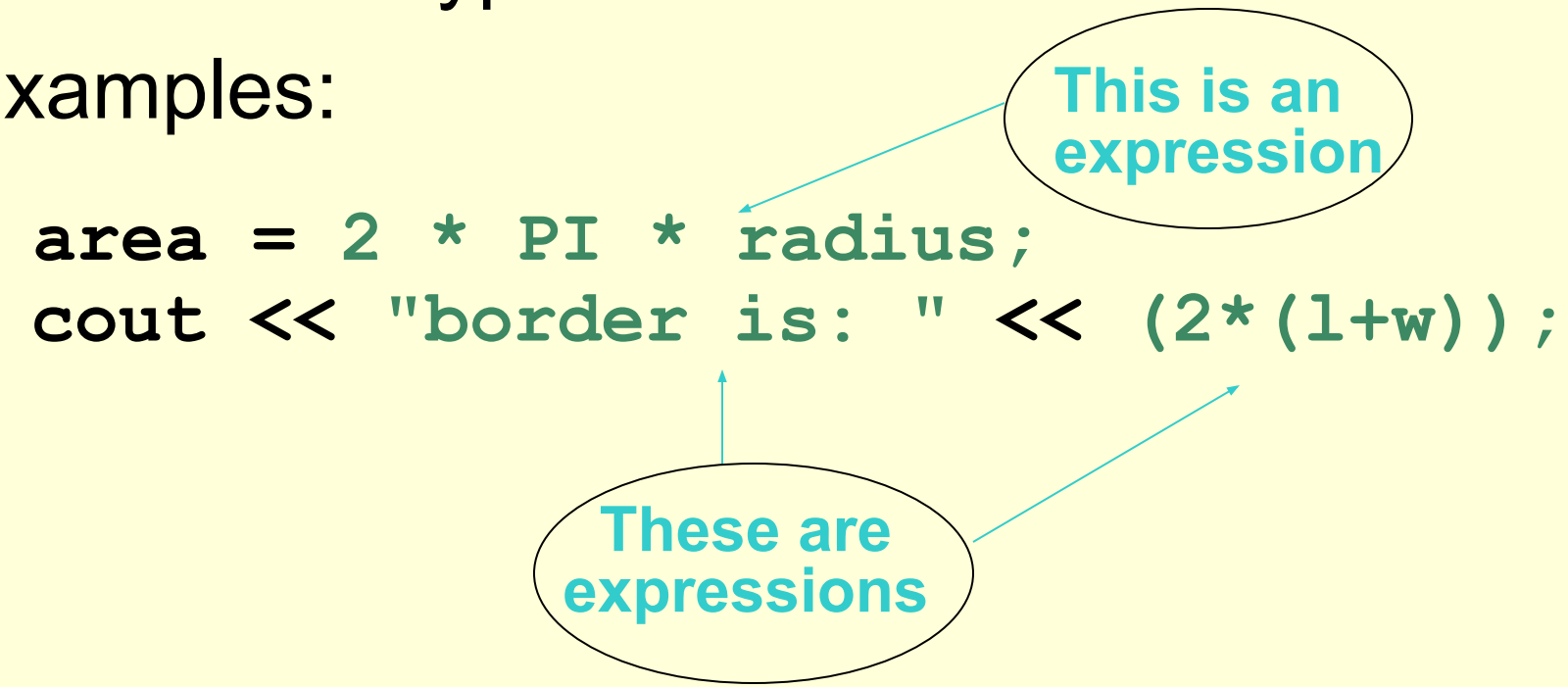
3.2 Mathematical Expressions

- An expression is something that can be evaluated to produce a value.
- It can be a constant, a variable, or a combination of constants and variables combined with operators and grouping symbols
- We can create complex expressions using multiple mathematical operators
- Examples of mathematical expressions:

```
2
height
a + b / c
```

Using Mathematical Expressions

- Can be used in assignment statements, with **`cout`**, and in other types of statements
- Examples:



Order of Operations

In an expression with > 1 operator, evaluate it in this order:

Do first: ( ) expressions in parentheses

Do next: `-` (unary negation) in order, left to right

Do next: `*` `/` `%` in order, left to right

Do last: `+` `-` in order, left to right

Ex: In the expression `2 + 2 * 2 - 2` ,

Associativity of Operators

- `-` (unary negation) associates right to left
- `*` `/` `%` `+` `-` all associate left to right
- parentheses `( )` can be used to override the order of operations

Expression	Value
`2 + 2 * 2 - 2`	4
`(2 + 2) * 2 - 2`	6
`2 + 2 * (2 - 2)`	2
`(2 + 2) * (2 - 2)`	0

Algebraic Expressions

- Multiplication requires an operator

  $Area = lw$ is written as `Area = l * w;`

- There is no exponentiation operator

  $Area = s^2$ is written as `Area = pow(s, 2);`

  (note: **`pow`** requires the **`cmath`** header file)

- Parentheses may be needed to maintain order of operations

  $m = \frac{y_2 - y_1}{x_2 - x_1}$ is written as `m = (y2-y1)/(x2-x1);`

3.3 Data Type Conversion and Type Casting

- Operations are performed between operands of the same type
- If operands do not have the same type, C++ will automatically convert one to be the type of the other
- This can impact the results of calculations

Hierarchy of Data Types

- Highest
- Lowest
- Ranked by largest number they can hold

```
long double
double
float
unsigned long long int
long long int
unsigned long int
long int
unsigned int
int
```

Type Coercion

- Coercion: automatic conversion of an operand to another data type
- Promotion: conversion to a higher type
- Demotion: conversion to a lower type

Coercion Rules

1) **`char`**, **`short`**, **`unsigned short`** are automatically promoted to **`int`**
2) When operating with values of different data types, the lower-ranked one is promoted to the type of the higher one.
3) When using the **=** operator, the type of expression on right will be converted to the type of variable on left

Coercion Rules – Important Notes

1) If demotion is required by the = operator,
   - the stored result may be incorrect if there is not enough space available in the receiving variable
   - floating-point values are truncated when assigned to integer variables

2) Coercion affects the value used in a calculation. It does not change the type associated with a variable.

Type Casting

- Is used for manual data type conversion
- Format

```
static_cast<Data Type>(Value)
```

- Example:

```
cout << static_cast<int>(4.2);
                        // Displays 4
```

More Type Casting Examples

```
char ch = 'C';
cout << ch << " is stored as "
     << static_cast<int>(ch);

gallons = static_cast<int>(area/500);

avg = static_cast<double>(sum)/count;
```

Older Type Cast Styles

```
double volume = 21.58;
int intVol1, intVol2;
intVol1 = (int) volume; // C-style
                        // type cast
intVol2 = int (volume); //Prestandard
                        // C++ style
                        // type cast
```

C-style cast uses prefix notation

Prestandard C++ cast uses functional notation

`static_cast` is the current standard

3.4 Overflow and Underflow

- Occurs when assigning a value that is too large (overflow) or too close to zero (underflow) to be held in a variable
- This occurs with both int and floating-point data types

Overflow Example

```
// Create a short int initialized to
// the largest value it can hold
short int num = 32767;

cout << num;      // Displays 32767
num = num + 1;
cout << num;      // Displays -32768
```

Handling Overflow and Underflow

Different systems handle the problem differently. They may

- display a warning / error message
- stop the program
- continue execution with the incorrect value

Using variables with appropriately-sized data types can minimize this problem

3.5 Named Constants

- Also called constant variables
- Variables whose content cannot be changed during program execution
- Used for representing constant values with descriptive names

```
const double TAX_RATE = 0.0775;
const int NUM_STATES = 50;
```

- Often named in uppercase letters

Defining and Initializing Named Constants

- The value of a named constant must be assigned when the variable is defined:

```
const int CLASS_SIZE = 24;
```

- An error occurs if you try to change the value stored in a named constant after it is defined:

```
// This won't work
CLASS_SIZE = CLASS_SIZE + 1;
```

Benefits of Named Constants

- They make program code more readable by documenting the purpose of the constant in the name:

```
const double TAX_RATE = 0.0775;

. . .

salesTax = purchasePrice * TAX_RATE;
```

- They improve accuracy and simplify program maintenance:

```
const double TAX_RATE = 0.0775;
```

3.6 Multiple and Combined Assignment

- The assignment operator (=) can be used multiple times in an expression

```
x = y = z = 5;
```

- Associates right to left

```
x = (y = (z = 5));
```

Done 3rd Done 2nd Done 1st

Combined Assignment

- Applies an arithmetic operation to a variable and assigns the result as the new value of that variable
- Operators: `+=` `-=` `*=` `/=` `%=`
- These are also called compound operators or arithmetic assignment operators
- Example:

`sum += amt;` is short for `sum = sum + amt;`

More Examples

```
x += 5;   means   x = x + 5;
x -= 5;   means   x = x - 5;
x *= 5;   means   x = x * 5;
x /= 5;   means   x = x / 5;
x %= 5;   means   x = x % 5;
```

The right hand side is evaluated before the combined assignment operation is done.

```
x *= a + b;  means  x = x * (a + b);
```

3.7 Formatting Output

- We can control how output displays for numeric and string data
  - size
  - position
  - number of digits
- This requires the **`iomanip`** header file

Stream Manipulators

- Are used to control features of an output field
- Some affect just the next value displayed

  `setw(x)`: Print a value in a field at least **`x`** spaces wide.

  –It will use more spaces if the specified field width is not big enough.

  –It right-justifies the value if it does not require **`x`** spaces.

  –Decimal points in floating-point values use a space.

  –All characters in strings, including space characters, use space

Stream Manipulators

- Some affect values until changed again
  - **`fixed`**: Use decimal notation (not E-notation) for floating-point values.
  - **`setprecision(x)`**:
    - When used with **`fixed`**, print floating-point value using **`x`** digits after the decimal.
    - Without **`fixed`**, print floating-point value using **`x`** significant digits.
    - Rounding is used if x is smaller than the number of significant digits

Stream Manipulators

- Some additional manipulators:
  - `showpoint`: Always print a decimal point for floating-point values. This is useful with **`fixed`** and **`setprecision`** when printing monetary values.
  - `left, right`: left- or right justification of a value in a field.

Manipulator Examples

```
const double e = 2.718;
double price = 18.0;
cout << setw(8) << e << endl;
cout << left << setw(8) << e
     << endl;
cout << setprecision(2);
cout << e << endl;
cout << fixed << e << endl;
cout << setw(6) << price;
```

Displays

```
^^^2.718

2.718^^^

2.7
2.72
18.00^
```

3.8 Working with Characters and Strings

- `char`: holds a single character
- `string`: holds a sequence of characters
- Both can be used in assignment statements
- Both can be displayed with **`cout`** and **`<<`**

String Input

Reading in a string object

```
string str;

cin >> str;                 // Reads in a string
                            // with no blanks

getline(cin, str);          // Reads in a string
                            // that may contain
                            // blanks
```

Character Input

Reading in a character:

```
char ch;

cin >> ch;     // Reads in any non-blank char

cin.get(ch); // Reads in any char

ch=cin.get();// Reads in any char

cin.ignore();// Skips over next char in
                    // the input buffer
```

`cin.ignore()`

General form: `cin.ignore(n,c);`

- **`n`** – number of characters to skip
- **`c`** – stop when character **`c`** is encountered

How it works:

- It stops if **`c`** is encountered before **`n`** characters have been skipped. Otherwise, **`n`** characters are skipped.
- Use **`cin.ignore();`** to skip a single character

string Member Functions

- **`length()`** – the number of characters in a string

```
string firstPrez="George Washington";
int size=firstPrez.length(); // size is 17
```

- **`length()`** includes blank characters

- **`length()`** does not include the `'\0'` null character that terminates the string

string Member Functions

- assign() – put repeated characters in a string.
- It can be used for formatting output.

```
    string equals;
    equals.assign(80,'=');
    . . .
    cout << equals << endl;
    cout << "Total: " << total << endl;
```

String Operators

= Assigns a value to a string

```
string words;
words = "Tasty ";
```

+ Joins two strings together

```
string s1 = "hot", s2 = "dog";
string food = s1 + s2; // food = "hotdog"
```

+= Concatenates a string onto the end of another one

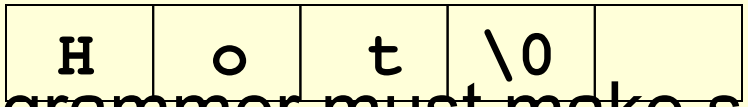
```
words += food; // words now = "Tasty hotdog"
```

Using C-Strings

- A C-string is stored as an array of characters
- The programmer must indicate the maximum number of characters at definition

```
const int SIZE = 5;
char temp[SIZE] = "Hot";
```

- NULL character (`\0`) is placed after final character to mark the end of the string



- The programmer must make sure that the array is big enough for desired use. `temp` can hold up to 4 characters plus the `\0`.

C-String and Keyboard Input

- Reading in a C-string

```
const int SIZE = 10;
char Cstr[SIZE];
cin >> Cstr;    // Reads in a C-string with no
                // blanks. It will write past the
                // end of the array if the input
                // string is too long.
cin.getline(Cstr, SIZE);
                // Reads in a C-string that may
                // contain blanks. Ensures that <= 9
          // chars are read in.
```

- You can also use **`setw()`** and **`width()`** to control input field widths

C-String and Input Field Width

- The **`setw()`** stream manipulator can be used with **`cin`** as well as with **`cout`**.
- When used with **`cin`** and a target C-string array, **`setw()`** limits the number of characters that are stored in the array

```
const int SIZE = 10;
char Cstr[SIZE];
cin >> setw(SIZE) >> Cstr;
```

- **`cin.width()`** can also provide this limit

```
cin.width(SIZE);
cin >> Cstr;
```

C-String Initialization vs. Assignment

- A C-string can be initialized at the time of its creation, just like a string object

```
const int SIZE = 10;

char month[SIZE] = "April";
```

- However, a C-string cannot later be assigned a value using the = operator; you must use the **`strcpy()`** function

```
char month[SIZE];
month = "August"              // wrong!
strcpy(month, "August"); //correct
```

More on C-Strings and Keyboard Input

- **`cin`** can be used to put a single word from the keyboard into a C-string
- The programmer must use **`cin.getline()`** to read an input string that contains spaces
- Note that **`cin.getline() ≠ getline()`**
- The programmer must indicate the target C-string and maximum number of characters to read:

```
const int SIZE = 25;
char name[SIZE];
cout << "What's your name? ";
cin.getline(name, SIZE);
```

3.9 More Mathematical Library Functions

- These require **`cmath`** header file
- They take **`double`** arguments and return a **`double`**
- Some commonly used functions

`abs`	Absolute value
`sin`	Sine
`cos`	Cosine
`tan`	Tangent
`sqrt`	Square root
`log`	Natural (e) log
`pow`	Raise to a power

3.10 Random Numbers

- Random number - a value that is chosen from a set of values. Each value in the set has an equal likelihood of being chosen.
- Random numbers are used in games and in simulations.
- You have to use the **`cstdlib`** header file

Getting Random Numbers

- `rand`
  - Returns a random number between `0` and the largest `int` the computer holds
  - Will yield the same sequence of numbers each time the program is run
- `srand(x)`
  - Initializes random number generator with `unsigned int x`. `x` is the “seed value”.
  - This should be called at most once in a program

More on Random Numbers

- Use **`time()`** to generate different seed values each time that a program runs:

```
#include <ctime> //needed for time()
…
unsigned seed = time(0);
srand(seed);
```

- Random numbers can be scaled to a range:

```
int max=6;
int num;
num = rand() % max + 1;
```

Thanks