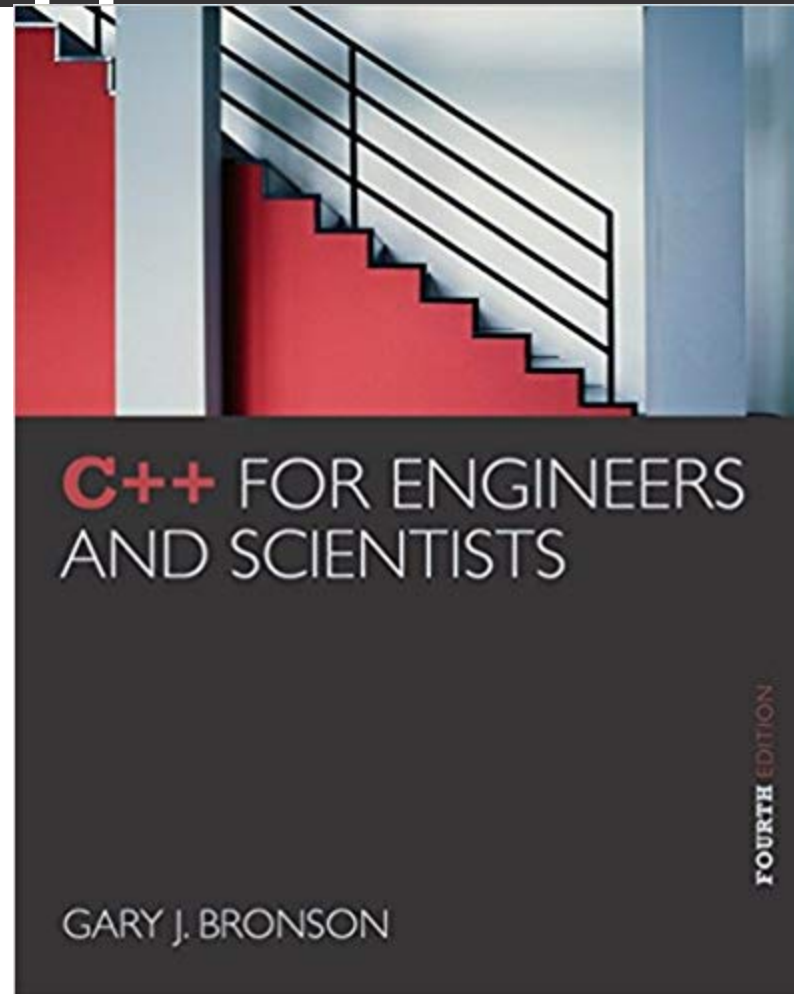


# ELEG 1043

## Computer Applications in Engineering





# Chapter 1: Preliminaries

**C++** FOR ENGINEERS  
AND SCIENTISTS

# Acknowledgement

- Some of the slides or images are from various sources. The copyright of those materials belongs to their original owners.

# Objectives

In this chapter, you will learn about:

- Unit analysis
- Exponential and scientific notations
- Software development
- Algorithms
- Software, hardware, and computer storage
- Common programming errors

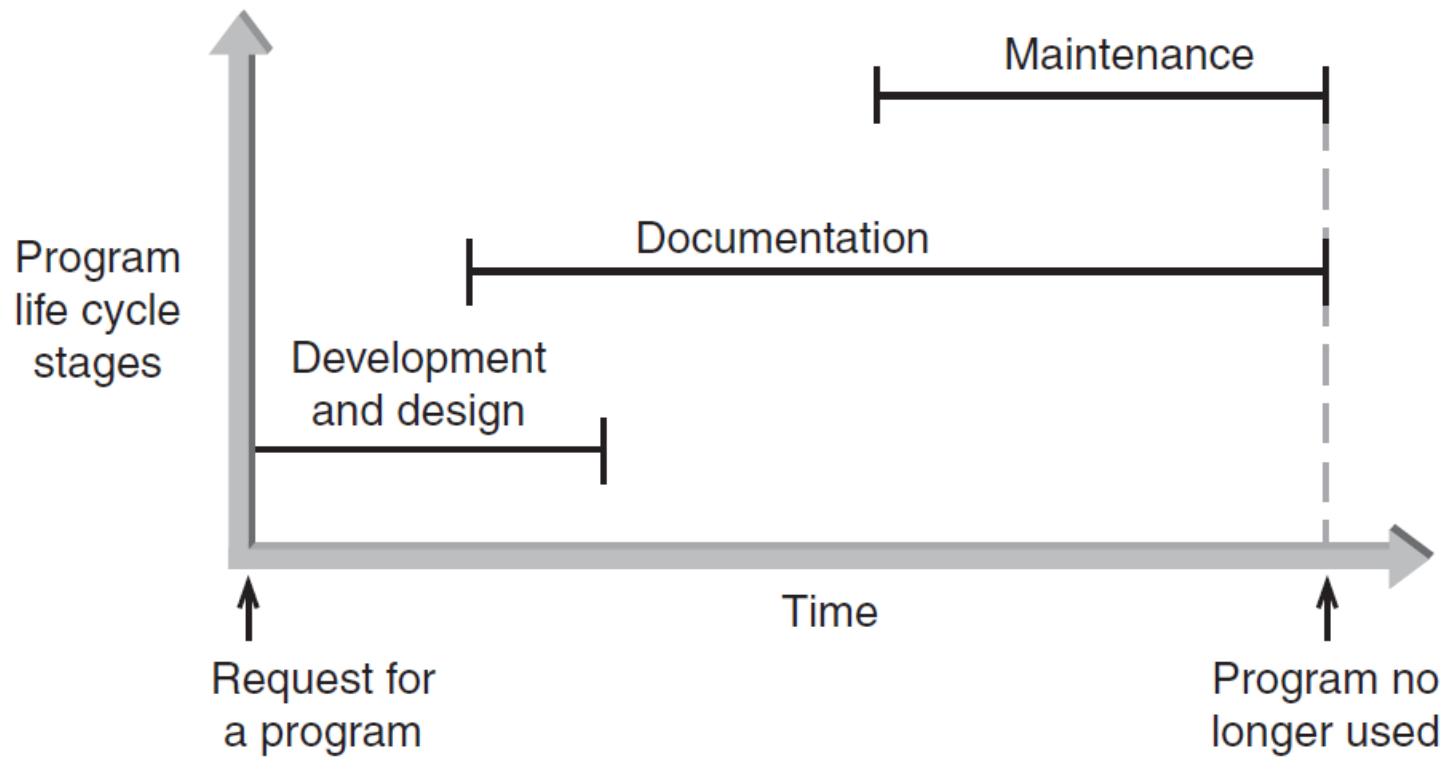
# Preliminary Three: Software Development

- **Computer program:** Self-contained set of instructions used to operate a computer to produce a specific result
  - Also called **software**
  - Solution developed to solve a particular problem, written in a form that can be executed on a computer

# Preliminary Three: Software Development (continued)

- **Software development procedure:** Helps developers understand the problem to be solved and create an effective, appropriate software solution
- **Software engineering:**
  - Concerned with creating readable, efficient, reliable, and maintainable programs and systems
  - Uses software development procedure to achieve this goal

# Preliminary Three: Software Development (continued)



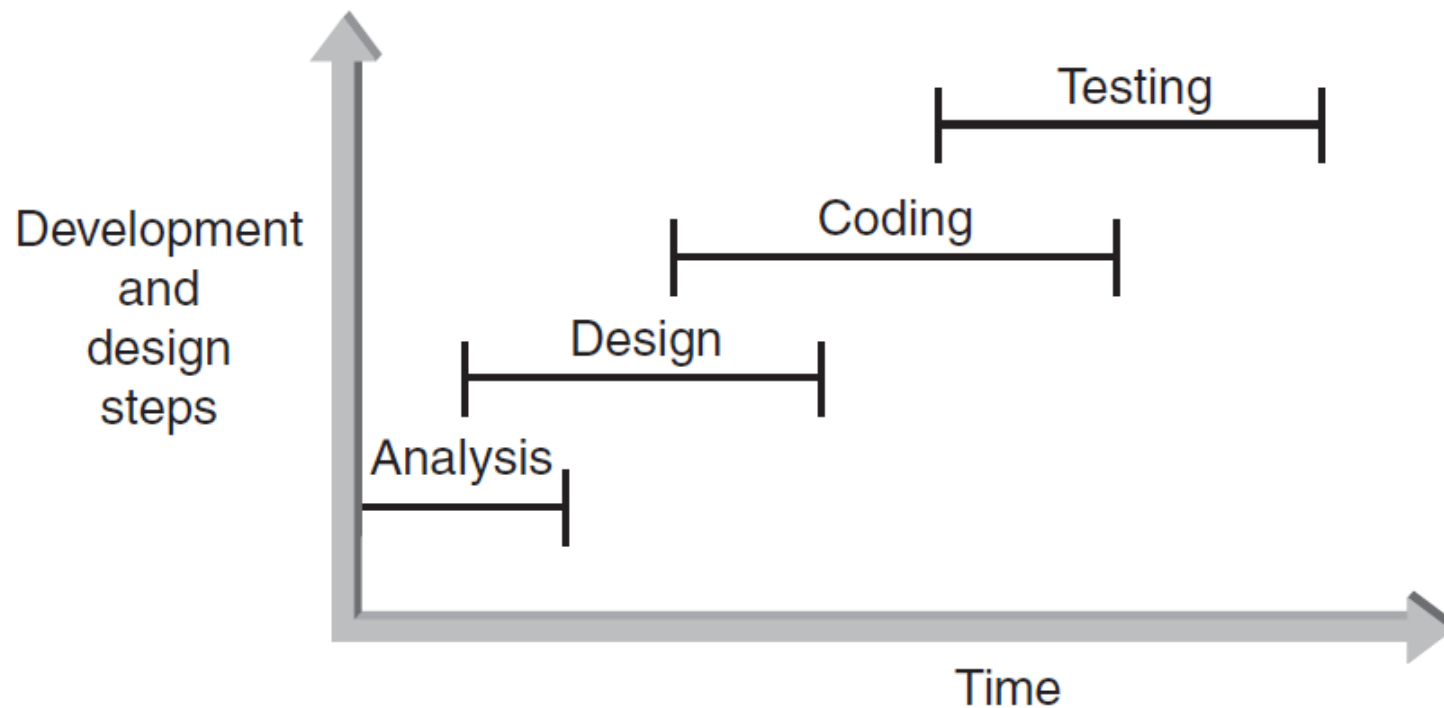
**Figure 1.2** The three phases of program development

# Phase I: Development and Design

- **Program requirement:** Request for a program or a statement of a problem
- After a program requirement is received, Phase I begins:
- Phase I consists of **four steps**:
  - Analysis
  - Design
  - Coding
  - Testing



# Phase I: Development and Design (continued)



**Figure 1.3** The development and design steps

# Phase II: Documentation

- Five main documents for every problem solution:
  - Program description
  - Algorithm development and changes
  - Well-commented program listing
  - Sample test runs
  - Users' manual

# Phase III: Maintenance

- **Maintenance** includes:
  - Ongoing correction of newly discovered bugs
  - Revisions to meet changing user needs
  - Addition of new features
- Usually the longest phase
- May be the primary source of revenue
- Good documentation vital for effective maintenance



# Chapter 2: Problem Solving Using C++

# Acknowledgement

- Some of the slides or images are from various sources. The copyright of those materials belongs to their original owners.

# Objectives

In this chapter, you will learn about:

- Modular programs
- Programming style
- Data types
- Arithmetic operations
- Variables and declaration statements
- Common programming errors

# Introduction to C++

- **Modular program:** A program consisting of interrelated segments (or **modules**) arranged in **a logical and understandable form**
  - Easy to develop, correct, and modify
- Modules in C++ can be **classes or functions**

# Introduction to C++ (continued)

- **Function:** Accepts an input, processes the input, and produces an output
  - A function's processing is encapsulated and hidden within the function

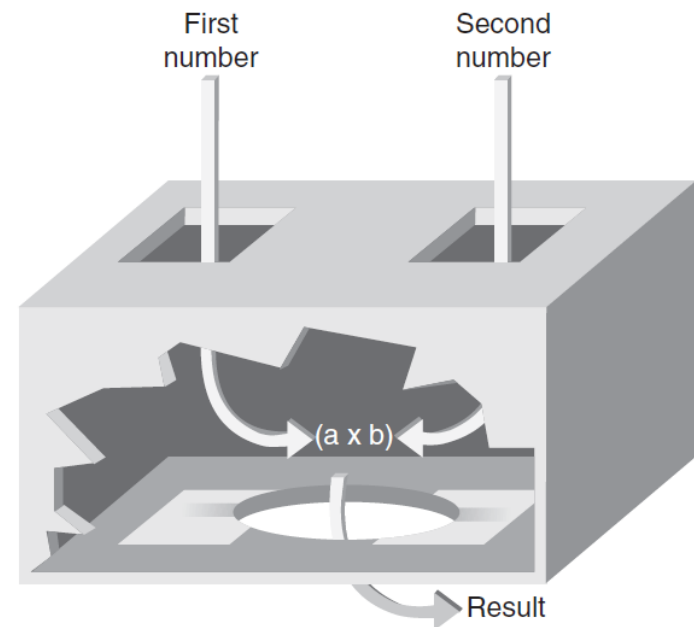


Figure 2.2 A multiplying function



# Introduction to C++ (continued)

- **Class:** Contains both **data and functions** used to manipulate the data
- **Identifier:** A name given to an element of the language, such as a class or function
  - Rules for forming identifier names:
    - **First character** must be a letter or underscore
    - Only letters, digits, or underscores may follow the initial letter (**no blanks allowed**)
    - **Keywords** cannot be used as identifiers
    - **Maximum length** of an identifier = **1024 characters**

# Introduction to C++ (continued)

- **Keyword:** A **reserved name** that represents a **built-in object** or function of the language

auto	delete	goto	public	this
break	do	if	register	template
case	double	inline	return	typedef
catch	else	int	short	union
char	enum	long	signed	unsigned
class	extern	new	sizeof	virtual
const	float	overload	static	void
continue	for	private	struct	volatile
default	friend	protected	switch	while

**Table 2.1:** Keywords in C++

# Introduction to C++ (continued)

- Examples of valid C++ identifiers:

`degToRad`   `intersect`   `addNums`  
`slope`   `bessell`   `multTwo`  
`findMax`   `density`

- Examples of invalid C++ identifiers:

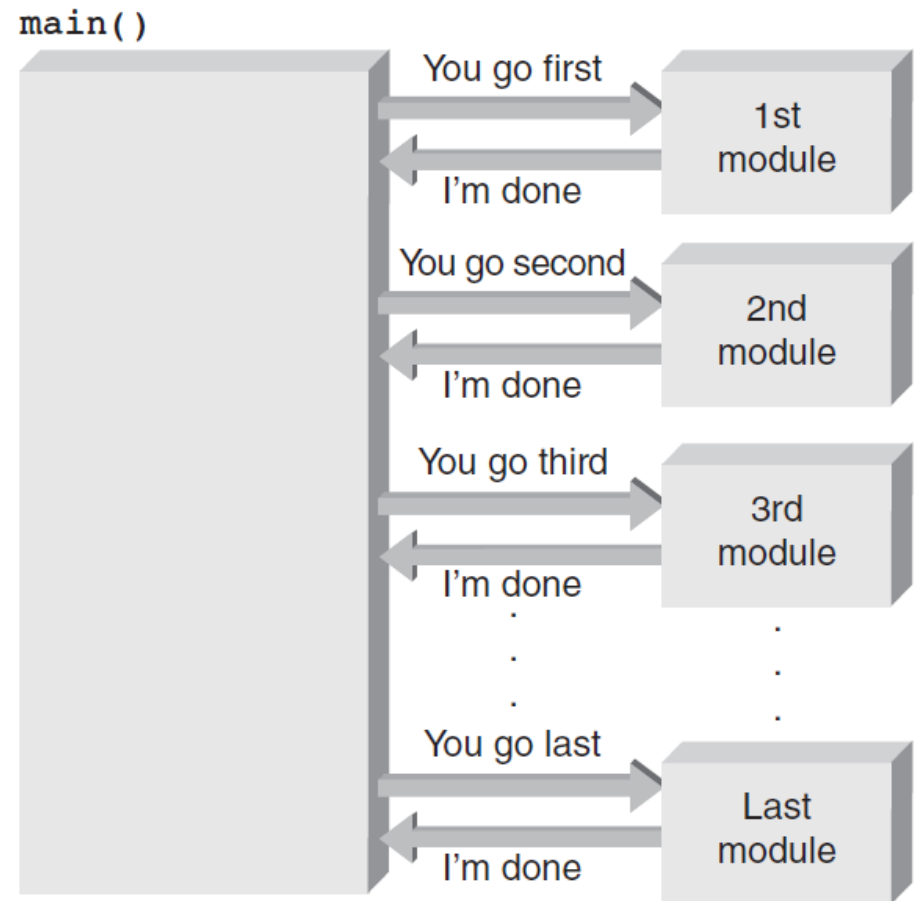
`1AB3` (begins with a number)  
`E*6` (contains a special character)  
`while` (this is a keyword)

# Introduction to C++ (continued)

- Function names
  - Require a set of parentheses at the end
  - Can use mixed upper and lower case
  - Should be meaningful, or be a **mnemonic**
- Examples of function names:  
`addNums ()`    `multTwoNums ()`
- Note that C++ is a **case-sensitive** language!
  - **addNums()** is different from **AddNums()**

# The `main()` Function

- Overall structure of a C++ program contains one function named `main()`, called the **driver function**
- All other functions are invoked from `main()`



**Figure 2.3** The `main()` function directs all other functions.

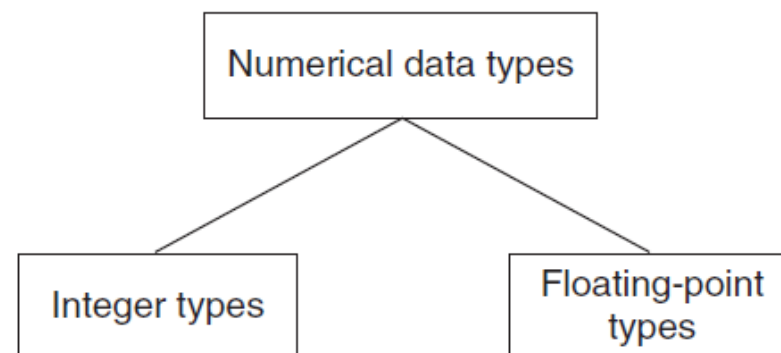
# The `main()` Function

- Write a program ....

```
#include <iostream>
using namespace std;
int main()
{
    return 0;
}
```

# Data Types

- **Data type:** A set of values and the operations that can be applied to these values
- Two fundamental C++ data groupings:
  - **Class data type** (a class): Created by the programmer
  - **Built-in data type** (primitive type): Part of the C++ compiler



**Figure 2.5** Built-in data types

# Data Types (continued)

Built-in Data Type	Operations
Integer	<code>+</code> , <code>-</code> , <code>*</code> , <code>/</code> , <code>%</code> , <code>=</code> , <code>==</code> , <code>!=</code> , <code>&lt;=</code> , <code>&gt;=</code> , <code>sizeof()</code> , and bit operations (see Chapter 15, available online)
Floating point	<code>+</code> , <code>-</code> , <code>*</code> , <code>/</code> , <code>=</code> , <code>==</code> , <code>!=</code> , <code>&lt;=</code> , <code>&gt;=</code> , <code>sizeof()</code>

**Table 2.2** Built-In Data Type Operations



# Data Types (continued)

- **Literal (constant):** An actual value

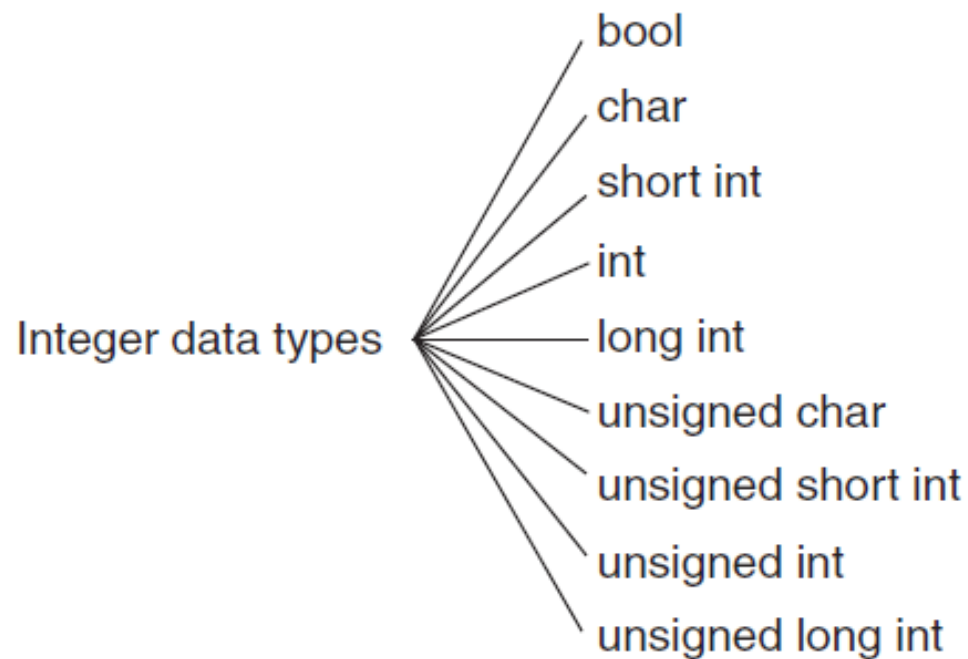
- Examples:

- `3.6        //numeric literal`

- `"Hello"   //string literal`

- **Integer:** A whole number
- C++ has nine built-in integer data types

# Integer Data Types



**Figure 2.6** C++ integer data types

# Integer Data Types (continued)

- **int** data type: Whole numbers (integers), optionally with plus (+) or minus (−) sign
  - Example: 2, −5
- **char** data type: Individual character; any letter, digit, or special character enclosed in **single quotes**
  - Example: 'A'

# Floating-Point Types

- **Floating-point number** (real number): Zero or any positive or negative number containing a decimal point
  - Examples: `+10.625`, `5.`, `-6.2`
  - Three floating-point data types in C++:
    - `float` (single precision)
    - `double` (double precision)
    - `long double`

# Floating-Point Types (continued)

- `float literal`: Append an `f` or `F` to the number
- `long double literal`: Append an `l` or `L` to the number

– Examples:

`9.234`     `// a double literal`

`9.234F`    `// a float literal`

`9.234L`    `// a long double literal`

# Arithmetic Operations

- C++ supports **addition**, **subtraction**, multiplication, **division**, and **modulus division**
- **Different** data types can be used in the **same** arithmetic expression
- Arithmetic operators are binary operators
  - **Binary operators:** Require two operands
  - **Unary operator:** Requires **only one** operand
  - **Negation operator (-):** **Reverses the sign** of the number

# Arithmetic Operations (continued)

Operation	Operator
Addition	+
Subtraction	-
Multiplication	*
Division	/
Modulus division	%

# Integer Division

- Integer division: Yields an integer result
  - Any **fractional remainders are dropped** (truncated)
  - Example:  $15/2$  yields 7
- Modulus (remainder) operator: Returns **only the remainder**
  - Example:  $9 \% 4$  yields 1



# Summary

- A C++ program consists of one or more **modules**, called **functions**, one of which **must be called `main()`**
- All C++ statements must be **terminated by a semicolon “;”**
- Data types include **`int`, `float`, `bool`**

# Summary (continued)

- Variables
  - Variables must be **declared with their data type**
  - A variable can be used **only after it has been declared**
  - Variables may be initialized when declared



# Chapter 3: Assignment, Formatting, and Interactive Input

**C++** FOR ENGINEERS  
AND SCIENTISTS

# Acknowledgement

- Some of the slides or images are from various sources. The copyright of those materials belongs to their original owners.

# Objectives

In this chapter, you will learn about:

- Assignment operations
- Formatting numbers for program output
- Using mathematical library functions
- Program input using the **cin** object
- Symbolic constants
- A case study involving acid rain
- Common programming errors

# Assignment Operations

- **Assignment Statement:** Assigns the value of the expression on the right side of the = to the variable on the left side of the =
  - `int a = 2;`  
    `left`      `right`
- Another assignment statement using the same variable will overwrite the previous value with the new value

Examples:

```
slope = 3.7;
```

```
slope = 6.28; (Overwrite)
```

# Assignment Operations (continued)

- Additional assignment operators provide short cuts: **+=**, **-=**, **\*=**, **/=**, **%=**

Example:

```
sum = sum + 10;
```

is equivalent to: **sum += 10;**

```
price *= rate + 1;
```

is equivalent to:

```
price = price * (rate + 1);
```

# Assignment Operations (continued)

- **Increment operator `++`**: Unary operator for the special case when a variable is increased by 1
- **Prefix increment operator** appears **before** the variable
  - Example: `++i`
- **Postfix increment operator** appears **after** the variable
  - Example: `i++`



# Assignment Operations (continued)

- Example: `k = ++n; //prefix increment`  
is equivalent to:

```
n = n + 1; //increment n first
```

```
k = n; //assign n's value to k
```

- Example: `k = n++; //postfix increment`  
is equivalent to

```
k = n; //assign n's value to k
```

```
n = n + 1; //and then increment n
```

# Assignment Operations (continued)

- **Decrement operator --**: Unary operator for the special case when a variable is decreased by 1
- **Prefix decrement operator** appears before the variable
  - Example: `--i;`
- **Postfix decrement operator** appears after the variable
  - Example: `i--;`

# Program Input Using **cin**

- **cin Object:** Allows **data entry** to a running program
- Use of the **cin** object causes the program to **wait for input from the keyboard**
- When keyboard entry is complete, the program **resumes** execution, **using the entered data**
- An output statement preceding the **cin** object statement provides a **prompt** to the user

# Program Input Using **cin**

- Number received from keyboard

```
int num;  
cin>>num;
```

# Program Input Using **cin** (continued)



## Program 3.13

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

int main()
{
    int num1, num2, num3;
    double average;

    cout << "Enter three integer numbers: ";
    cin >> num1 >> num2 >> num3;
    average = (num1 + num2 + num3) / 3.0;
    cout << "The average of the numbers is " << average << endl;

    return 0;
}
```

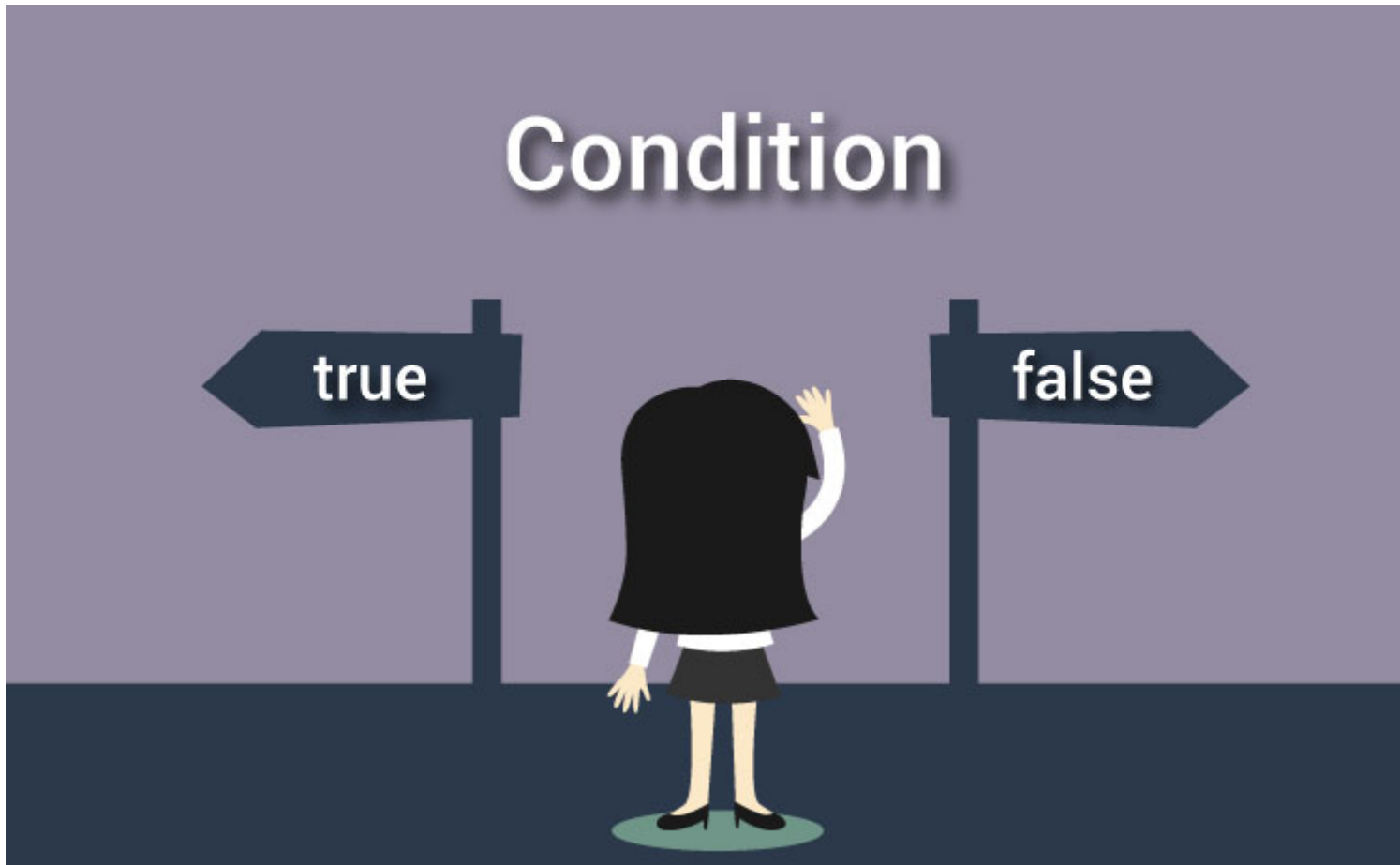


# Chapter 4: Selection Structures

# Objectives

- In this chapter, you will learn about:
  - Selection criteria
  - The **if-else** statement
  - Nested **if** statements
  - The **switch** statement
  - Program testing
  - Common programming errors

# Selection Criteria



<https://www.programiz.com/c-programming/c-if-else-statement>



# Selection Criteria

- **if-else** statement: Implements a decision structure for two alternatives

Syntax:

*if (condition)*

*statement executed if condition is **true**;*

*else*

*statement executed if condition is **false**;*

# Selection Criteria (continued)

- The condition is evaluated to its **numerical value**:
  - A **non-zero value** is considered to be **true**
  - A **zero** value is considered to be **false**
- The **else** portion is optional
  - **Executed only if the condition is false**

# Logical Operators

- AND (&&): Condition is true only if **both expressions are true**
- OR (||): Condition is true if **either one or both of the expressions is true**
- NOT (!): Changes an expression to its opposite state; **true becomes false, false becomes true**

# Logical Operators

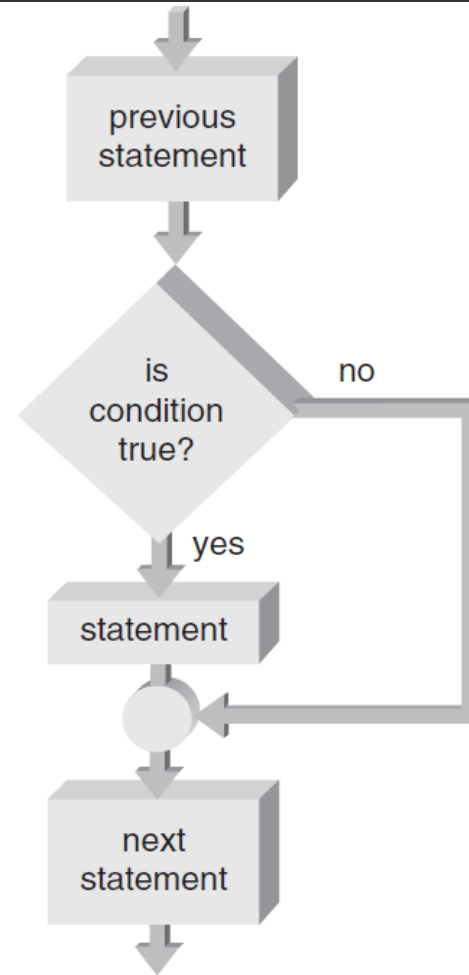
- $A \ \&\& \ B \rightarrow$  True when both A and B are True, Otherwise False
- $A \ || \ B \rightarrow$  False when both A and B are False, Otherwise True

# One-Way Selection

- **One-way selection:** An **if** statement without the optional **else** portion

```
int a = 1;  
if(a > 0)  
{  
    cout<<a;  
}
```

**Figure 4.3** A one-way selection **if** statement

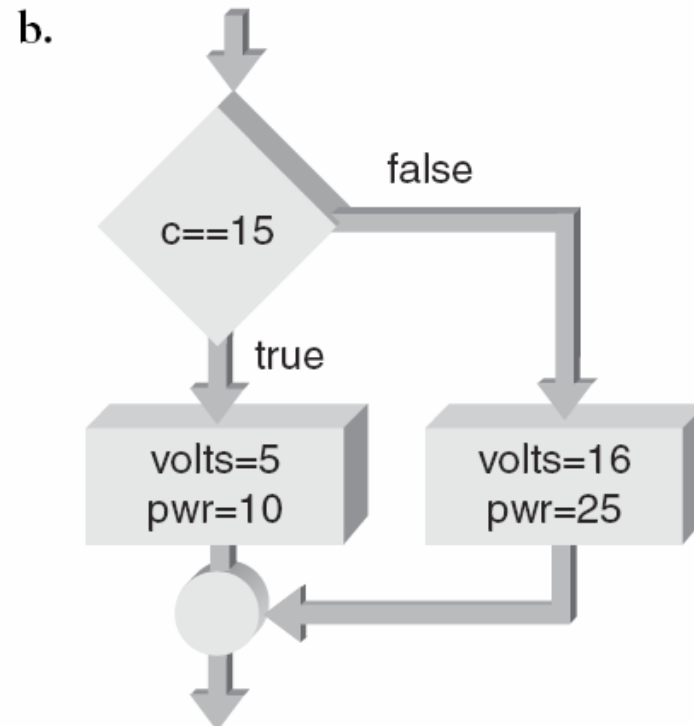
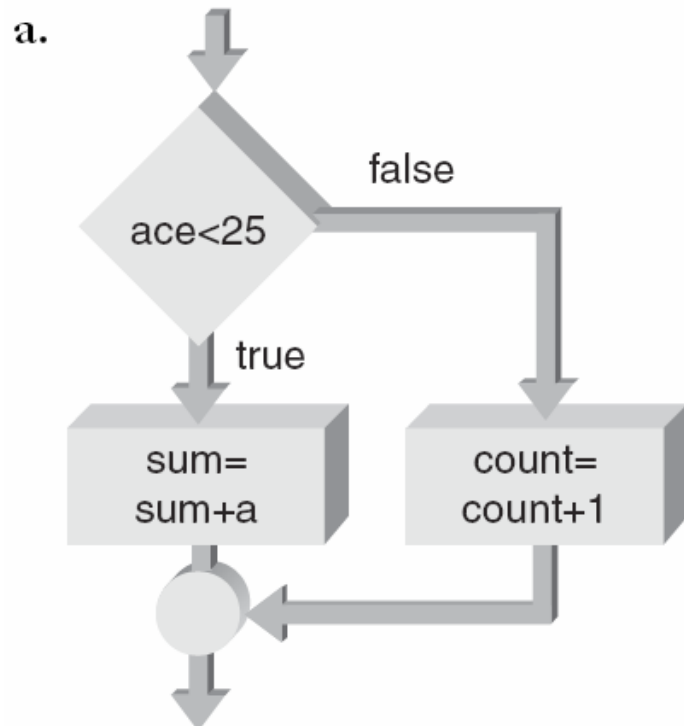


# Nested `if` Statements

- `if-else` statement can contain any valid C++ statement, including another `if-else`
- Nested `if` statement: an `if-else` statement completely contained within another `if-else`
- Use braces to block code, especially when inner `if` statement does not have its own `else`

# Exercise

(Practice) Write `if` statements corresponding to the conditions illustrated in the following flowcharts:



# Answer

```
a.  
if(a < 25)  
{  
    sum = sum + a;  
}  
else  
{  
    count = count + 1;  
}  
b.  
if(c == 15)  
{  
    volt = 5; power = 10;  
}  
else  
{  
    volt = 16; power = 25;  
}
```





# Chapter 5: Repetition Statements

**C++** FOR ENGINEERS  
AND SCIENTISTS

# Objectives

In this chapter, you will learn about:

- Basic loop structures
- **while** loops
- Interactive **while** loops
- **for** loops
- Loop programming techniques

# Objectives (continued)

- Nested loops
- **do while** loops
- Common programming errors

# Basic Loop Structures

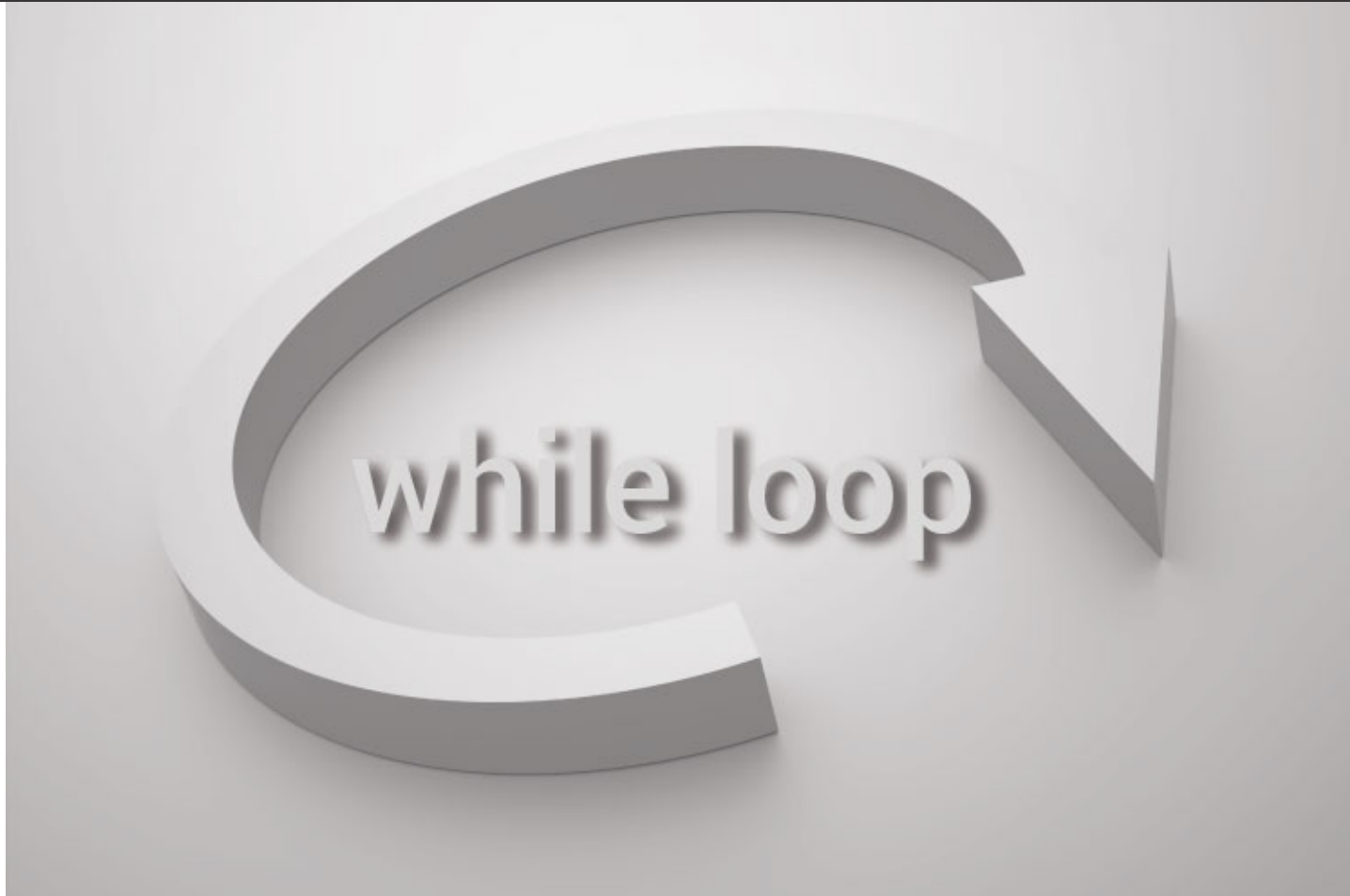
- Repetition structure has **four** required elements:
  - **Repetition statement**
  - Condition to **be changed**
  - **Initial** value for the condition
  - Loop **termination**
- Repetition statements include:
  - **while**
  - **for**
  - **do while**

# Example

- (a) Repetition statement
- (b) Condition to be changed
- (c) Initial value for the condition
- (d) Loop termination

```
        (c)      (d)      (b)
for(int i = 0; i < 10; i++)
{ cout<<i<<endl; (a)
}
```

# while Loops



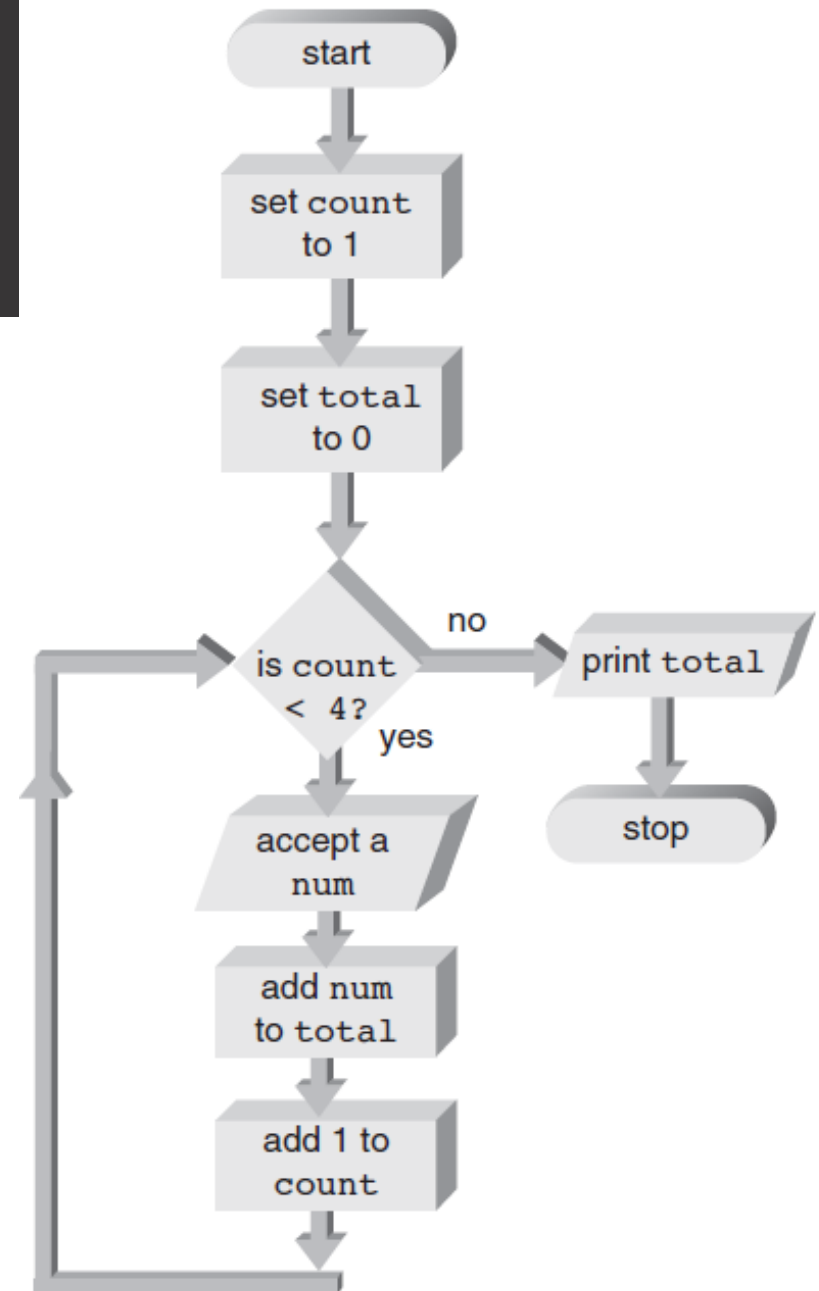
# while Loops

- **while statement** is used to create a `while` loop
  - Syntax:

***while (expression)***  
***statement;***

- Statements following the expressions are executed as long as the expression condition remains true (evaluates to a non-zero value)

# Interactive while Loops (cont'd)





# Interactive while Loops (cont'd)

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

int main()
{
    int count = 0, total = 0;

    while(count < 4)
    {
        int num;
        cout<<"Enter a num \n";
        cin>>num;
        total = total + num;
        count = count + 1;
    }

    cout<<"The total is "<<total;
    return 0;
}
```

# Exercise

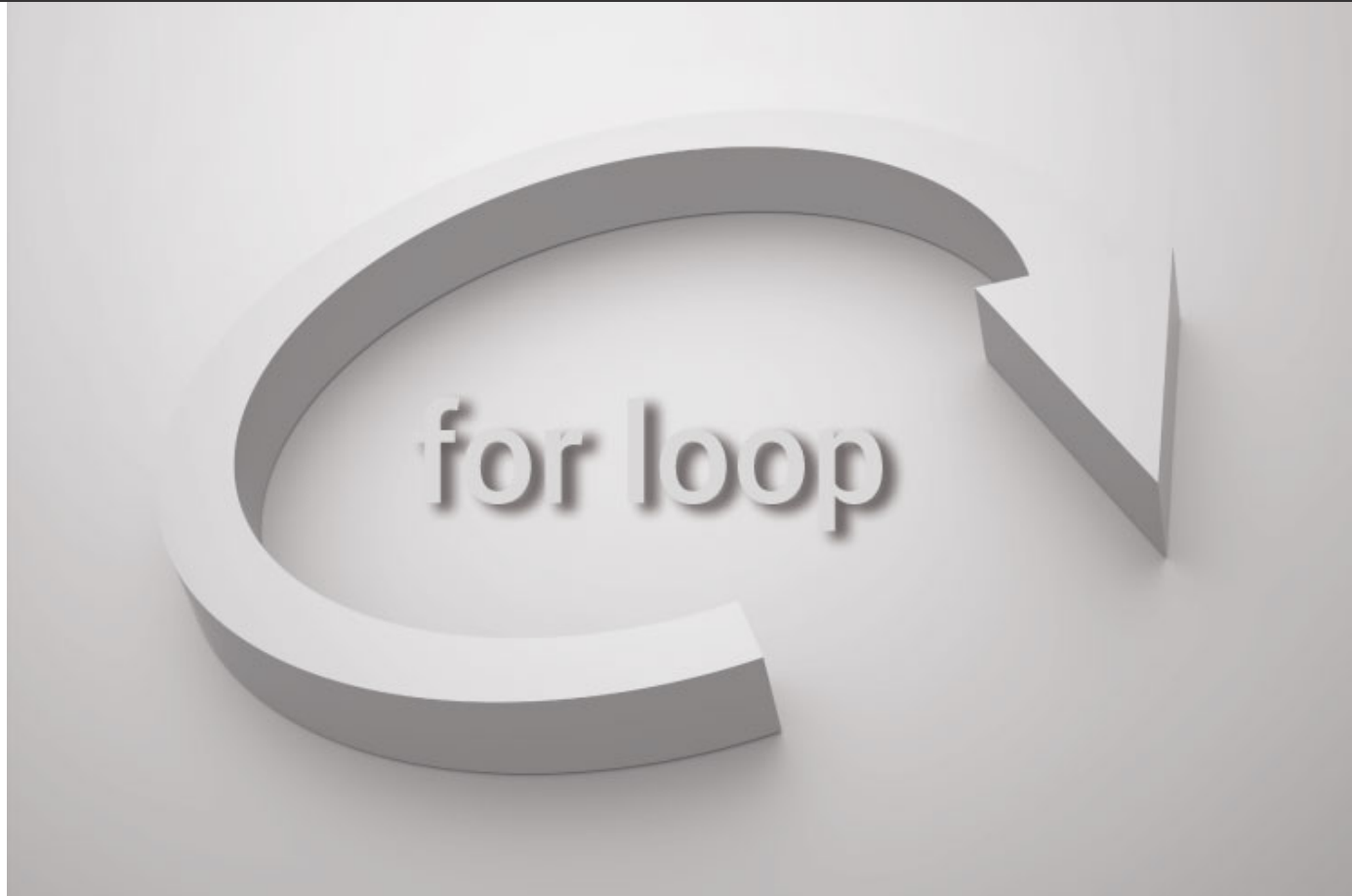
- Write a Program to print the numbers from 1 to 10 in increments of 3 with loop statment. The output of your program should be the following:

1 4 7 10

# Answer

```
#include <stream>
using namespace std;
int main(){
int num = 1;
while(num < 11){
    cout<<num<<" ";
    num = num + 3;}
return 0;
}
```

# for Loops



# for Loops

- **for** statement: A loop with a fixed count condition that handles alteration of the condition
  - Syntax:  
***for (initializing list; expression; altering list)***  
***statement;***
- **Initializing list:** Sets the starting value of a counter
- **Expression:** Contains the maximum or minimum value the counter can have; determines when the loop is finished

# Exercise

**(Desk check)** Determine the value in `total` after each of the following loops is executed:

a. `total = 0;`

```
    for (i = 1; i <= 10; i = i + 1)
        total = total + 1;
```

b. `total = 1;`

```
    for (count = 1; count <= 10; count = count + 1)
        total = total * 2;
```

c. `total = 0;`

```
    for (i = 10; i <= 15; i = i + 1)
        total = total + i;
```

d. `total = 50;`

```
    for (i = 1; i <= 10; i = i + 1)
        total = total - i;
```

# Answer

a.  $\text{total} = 0 + 1 * 10 = 10$

b.  $\text{total} = 1 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 = 1,024$

c.  $\text{total} = 0 + 10 + 11 + 12 + 13 + 14 + 15 = 75$

d.  $\text{total} = 50 - 1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9 - 10 = -5$

# Exercise

- Write a Program to display the numbers from 1 to 10 in increments of 2 with the loop statement. The output of your program should be the following:

1 3 5 7 9



# Answer

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

int main()
{
    for(int i = 1; i <10; i = i + 2)
    {
        cout<<i<<" ";
    }
    return 0;
}
```

# Exercise

- What will be displayed after each of the following loops is executed? (10 points) :

```
total = 0;
for (i = 3; i <= 10; i = i + 1)
{
    total = total + 3;
    cout<<total<<"  ";}
```



# Chapter 6: Modularity Using Functions

**C++** FOR ENGINEERS  
AND SCIENTISTS

# Function and Parameter Declarations

- Interaction with a function includes:
  - **Passing data (Arguments)** to a function correctly when its called
  - **Returning values** from a function when it ceases operation
- A function is called by (1) **giving the function's name** and (2) **passing arguments in the parentheses** following the **function name**

*function-name* (*data passed to function*);

This identifies the called function      This passes data to the function

**Figure 6.1** Calling and passing data to a function

# Function and Parameter Declarations (continued)

- **Before a function is called**, it must be **declared** to function that will do calling
- **Declaration statement for a function** is referred to as function prototype
- **Function prototype** tells calling function:
  - **Type of value** that will be formally **returned**, if any
  - **Data type and order of the values (arguments)** the calling function should transmit to the called function
- Function prototypes can be placed with the variable declaration statements **above** the calling function name or in a separate header file

# Defining a Function

- Every C++ function consists of two parts:
  - **Function header**
  - **Function body**
- Function header's purpose:
  - **Identify data type** of value function returns, **provide** function with **name**, and **specify** number, order, and type of **arguments** function expects
- Function body's purpose:
  - To operate on passed data and return, **at most**, one value directly back to the calling function

# Calling a Function

- Requirements when calling a function include:
  - Using the **name** of the function
  - Enclosing **any data passed (Arguments)** to the function in the parentheses following the function name, using the **same order and type** declared in the function prototype

# Exercise

- Write a program that is to build a function to receive 5 numbers from the keyboard and display "You lose!" if the number is less than 0, and call this function in main function.



# Answer

```
#include <iostream>
using namespace std;
void game() {
    for(int i = 0; i < 5; i++){
        int num;
        cin>>num;
        if(num < 0){
            cout<<"You lose!"<<endl;
        }
    }
}
int main() {
    game();
    return 0;
}
```



# Chapter 7: Arrays

**C++** FOR ENGINEERS  
AND SCIENTISTS

# Acknowledgement

- Some of the slides or images are from various sources. The copyright of those materials belongs to their original owners.

# Objectives

In this chapter, you will learn about:

- One-dimensional arrays
- Array initialization
- Declaring and processing two-dimensional arrays
- Arrays as arguments
- Statistical analysis

# Objectives (continued)

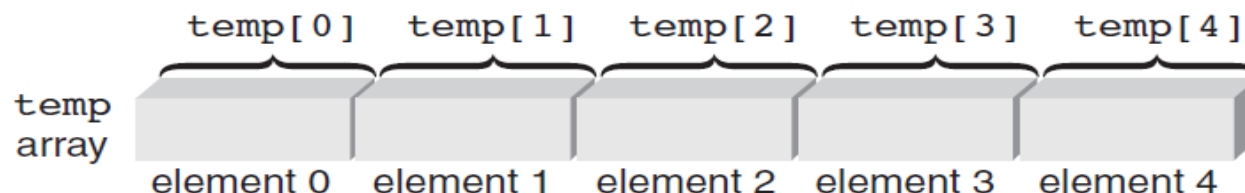
- The Standard Template Library (STL)
- Searching and sorting
- Common programming errors

# One-Dimensional Arrays

- **One-dimensional array:** A **list** of related values with the **same data type**, stored using a **single group name** (called the **array name**)
  - Syntax:  
`dataType arrayName[number-of-items]`
- By convention, **the number of items is first declared as a constant**, and the constant is used in the array declaration

# One-Dimensional Arrays (continued)

- **Element:** An item in the array
  - Array storage of elements is **contiguous**
- **Index (or subscript)** of an element: The **position** of the element within the array
  - Indexes are **zero**-relative
- To **reference an element**, use the **array name** and the **index** of the element



**Figure 7.2** Identifying array elements

# One-Dimensional Arrays (continued)

- All of the elements of an array can be processed by using **a loop**
- The loop counter is used as the **array index** to specify the element
- Example:

```
int sum = 0;  
    int temp[5] = {1,2,3,4,5};  
for (int i=0; i<5; i++)  
    sum = sum + temp[i];
```



# Homework Assignment 4

2. (Desk check) Determine the output produced by the following program:

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

int main()
{
    int i, j, val[3][4] = {8,16,9,52,3,15,27,6,14,25,2,10};

    for (i = 0; i < 3; ++i)
        for (j = 0; j < 4; ++j)
            cout << "  " << val[i][j];

    return 0;
}
```

# Homework Assignment 4

```
int a[20] =  
    {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20};  
for (int k = 1; k < 5; k = k + 3)  
    cout << a[k] << " ";
```

# Input and Output of Array Values

- Array elements can be **assigned values interactively** using a **cin** stream object
- Out of range array indexes are **not checked** at compile-time
  - May produce **run-time errors**
  - May overwrite a value in the referenced memory location and cause other errors
- Array elements can be displayed using the **cout** stream object

# Declaring and Processing Two-Dimensional Arrays

- **Two-dimensional array:** Has both rows and columns
  - Also called a **table**
- **Both dimensions** must be **specified** in the array declaration
  - **Row is specified first**, then column
- **Both dimensions must be specified** when referencing an array element

# Declaring and Processing Two-Dimensional Arrays (cont'd)

- Two-dimensional arrays can be initialized in the declaration by listing values within **braces, separated by commas**
- Braces can be used to distinguish rows, but are not required
- **Nested for** loops are used to process two-dimensional arrays
  - **Outer loop** controls the **rows**
  - **Inner loop** controls the **columns**

# Arrays as Arguments

- An individual array element can be passed as an argument just like any individual variable
- The called function receives a copy of the array element's value
- Passing an entire array to a function causes the function to receive a reference to the array, not a copy of its element values
- The function must be declared with an array as the argument
- Single element of array is obtained by adding an offset to the array's starting location

# Exercise 1

- Write a program to input 10 positive integer numbers in an array named **Mini** and determine and display the **minNum** value entered, where the numbers are received from keyboard

# Answer

```
#include <iostream>
using namespace std;
int main(){
    int Mini [10] = {0};
    int num = -1;
    cin>>num;
    Mini [0] = num;
    int minNum = num;
    for(int i = 1; i < 10; i++){
        cin>>num;
        Mini [i] = num;
        if(minNum > num)
            {minNum = num;} }
    cout<<minNum<<endl;
    return 0;}
```



## Exercise 2

- Write a program to build a function named **multiply** to input the following integer numbers in an array named grades: 12.3, 16.4, and 30.6. As each number is input, multiply the numbers to a variable **mul** and return the **mul** value.

# Answer

```
#include <iostream>
using namespace std;
double multiply(double arr[], int length);
int main(){
    double arr[3] = {12.3, 16.4, 30.6};
    cout<< multiply(arr, 3);
    return 0;
}

double multiply(double arr[], int length){
    double mul = 1.0;
    for(int i = 0; i < length; i++){
        mul = mul*arr[i];
    }
    return mul;
}
```

# Summary

- An array is a data structure that stores a list of values having the same data type
  - Array elements: stored in **contiguous memory** locations; referenced by **array name/index position**
  - Two-dimensional arrays have **rows and columns**
  - Arrays may be initialized when they are declared
  - Arrays may be passed to a function by **passing the name of the array as the argument**
    - **Arrays passed as arguments are passed by reference**
    - Individual array elements as arguments are passed **by value (copy)**



# Chapter 10: Pointers

**C++** FOR ENGINEERS  
AND SCIENTISTS

# Objectives

- In this chapter you will learn about:
  - Addresses and pointers
  - Array names as pointers
  - Pointer arithmetic
  - Passing addresses
  - Common programming errors

# Addresses and Pointers

- The address operator, `&`, accesses a variable's address in memory
- The address operator placed in front of a variable's name refers to the address of the variable

`&num` means the address of `num`

# Storing Addresses (continued)

- Example statements store **addresses of the variable** `m`, `list`, and `ch` in the variables `d`, `tabPoint`, and `chrPoint`

```
d = &m;
```

```
tabPoint = &list;
```

```
chrPoint = &ch;
```

- `d`, `tabPoint`, and `chrPoint` are called **pointer variables or pointers**

# Storing Addresses (continued)

Variable:	Contents:
d	Address of m
tabPoint	Address of list
chrPoint	Address of ch

**Figure 10.3** Storing more addresses

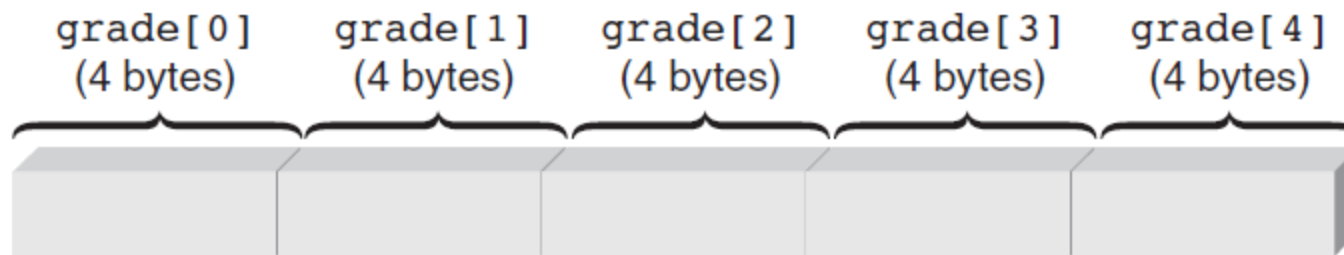


# Declaring Pointers

- Like all variables, pointers must be **declared before** they can be used to store an address
- When declaring a pointer variable, C++ requires **specifying the type of the variable** that is pointed to
  - Example: `int *numAddr;`

# Array Names as Pointers

- There is a direct and simple relationship between array names and pointers



**Figure 10.9** The `grade` array in storage

- Using subscripts, the fourth element in `grade` is referred to as `grade[3]`, address calculated as:

```
&grade[3] = &grade[0] + (3 *  
sizeof(int))
```

# Array Names as Pointers (continued)

Array Element	Subscript Notation	Pointer Notation
Element 0	<code>grade[0]</code>	<code>*gPtr</code> or <code>(gPtr + 0)</code>
Element 1	<code>grade[1]</code>	<code>*(gPtr + 1)</code>
Element 2	<code>grade[2]</code>	<code>*(gPtr + 2)</code>
Element 3	<code>grade[3]</code>	<code>*(gPtr + 3)</code>
Element 4	<code>grade[4]</code>	<code>*(gPtr + 4)</code>

**Table 10.1** Array Elements Can Be Referenced in Two Ways

# Exercise 1

- Replace each of the following references to a subscripted variable with a **pointer** reference
  - `year[10]`
  - `seconds[30]`
  - `students[0]`

## Exercise 2

- Replace each of the following pointer references with a subscript (index) reference
  - `* (year + 2)`
  - `* (seconds + 20)`
  - `* (students)`

# Homework Assignment 5

1. (Practice) Replace each of the following references to a subscripted variable with a pointer reference:

a. `prices[5]`

d. `dist[9]`

g. `celsius[16]`

b. `grades[2]`

e. `mile[0]`

h. `num[50]`

c. `yield[10]`

f. `temp[20]`

i. `time[12]`

2. (Practice) Replace each of the following pointer references with a subscript reference:

a. `*(message + 6)`

c. `*(yrs + 10)`

e. `*(rates + 15)`

b. `*amount`

d. `*(stocks + 2)`

f. `*(codes + 19)`

# Passing Arrays

- When an array is passed to a function, its address is the only item actually passed
  - “Address” means **the address of the first location** used to store the array
  - **First location is always element zero of the array**

# Common Programming Errors

- Attempting to store address in a variable **not declared as pointer**
- Using pointer to access **nonexistent array elements**
- Initialized pointer variables incorrectly



# Summary

- Every variable has a data type, **an address**, and a value
- In C++, obtain **the address of variable by using the address operator, &**
- A pointer is a variable used to store the address of another variable
  - Must be declared
  - Use indirection operator, **\***, to declare the pointer variable and access the variable whose address is stored in pointer

# Summary (continued)

- **Array name** is a pointer constant
- Arrays are passed to functions as addresses
- When a one-dimensional array is passed to a function, the function's parameter declaration can be an array declaration or a pointer declaration
- Pointers can be incremented, decremented, compared, and assigned



# Chapter 11: Introduction to Matlab

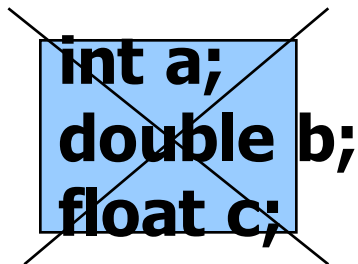
**C++** FOR ENGINEERS  
AND SCIENTISTS

# Objectives

- In this chapter you will learn about:
  - What is Matlab?
  - Matlab Screen
  - Variables, array, matrix, indexing
  - Operators (Arithmetic, relational, logical )
  - Display Facilities
  - Flow Control
  - Using of M-File
  - Debugging

# Variables

- No need for types. i.e.,



```
int a;  
double b;  
float c;
```

- All variables are created with **double precision** unless specified and they are matrices.

**Example:**

```
>>x=5;  
>>x1=2;
```

- After these statements, the variables are **1x1 matrices** with double precision

# Workspace

- The workspace is Matlab's memory
- Can **manipulate variables** stored in the workspace

```
>> a=12;
```

```
>> b=10;
```

```
>> c=a+b
```

```
c =
```

```
22
```

# Variables

- **View variable contents by simply typing the variable name at the command prompt**

```
>> a
```

```
a =
```

```
12
```

```
>>
```

```
>> a*2
```

```
a =
```

```
24
```

```
>>
```

# Array, Matrix

- A vector  $\mathbf{x} = [1 \ 2 \ 5 \ 1]$

$\mathbf{x} =$   
1    2    5    1

- A matrix  $\mathbf{t} = [1 \ 2 \ 3; \ 5 \ 1 \ 4; \ 3 \ 2 \ -1]$

$\mathbf{t} =$   
1        2        3  
      5        1        4  
      3        2       -1

- Transpose  $\mathbf{y} = \mathbf{x}'$      $\mathbf{y} =$   
      1  
      2  
      5  
      1



# The : operator

- VERY important operator in Matlab
- Means 'to'

```
>> 1:10
```

```
ans =
```

```
1 2 3 4 5 6 7 8 9 10
```

```
>> 1:2:10
```

```
ans =
```

```
1 3 5 7 9
```



Try the following

```
>> x=0:pi/12:2*pi;
```

```
>> y=sin(x)
```

# The **:** operator

```
>>A(3,2:3)
```

```
ans =
```

```
1    7
```

```
>>A(:,2)
```

```
ans =
```

```
2
```

```
1
```

```
1
```

```
A =
```

```
3    2    1  
5    1    0  
2    1    7
```



What'll happen if you type  
**A(:,,:) ?**

# Long Array, Matrix

- **`t = 1:10`**

```
t =  
 1   2   3   4   5   6   7   8   9  10
```

- **`k = 2:-0.5:-1`**

```
k =  
 2  1.5  1  0.5  0 -0.5 -1
```

- **`X = [1:4; 5:8]`**

```
x =  
 1   2   3   4  
 5   6   7   8
```

# Generating Vectors from functions

- **zeros(M,N)** MxN matrix of zeros

```
x = zeros (1,3)
```

```
x =
```

```
0      0      0
```

---

- **ones(M,N)** MxN matrix of ones

```
x = ones (1,3)
```

```
x =
```

```
1      1      1
```

---

- **rand(M,N)** MxN matrix of  
uniformly distributed  
random

```
x = rand (1,3)
```

```
x =
```

```
0.9501  0.2311  0.6068
```

# Matrix Index

- The matrix indices begin from **1** (not 0 (as in C))
  - The matrix indices must be **positive integer**
- Given:

```
A =  
  
    3    5    3  
    6    8    2  
    2    7    3
```

```
>> A(6)  
  
ans =  
  
    7
```

```
>> A(3,2)  
  
ans =  
  
    7
```

```
>> A(2,:)   
  
ans =  
  
    6    8    2
```

```
>> A(1:2,2)  
  
ans =  
  
    5  
    8
```

# Operators (arithmetic)

- + addition
- - subtraction
- \* multiplication
- / division
- ^ power
- ' matrix transpose

# Matrices Operations

**Given A and B:**

```
>> A = [1 2 3;4 5 6;7 8 9]
```

A =

1	2	3
4	5	6
7	8	9

```
>> B = [3 5 2; 5 2 8; 3 6 9]
```

B =

3	5	2
5	2	8
3	6	9

**Addition**

```
>> X = A + B
```

X =

4	7	5
9	7	14
10	14	18

**Subtraction**

```
>> Y = A - B
```

Y =

-2	-3	1
-1	3	-2
4	2	0

**Product**

```
>> Z = A * B
```

Z =

22	27	45
55	66	102
88	105	159

**Transpose**

```
>> T = A'
```

T =

1	4	7
2	5	8
3	6	9

# Operators (Element by Element)

- `.*` element-by-element multiplication
- `./` element-by-element division
- `.^` element-by-element power

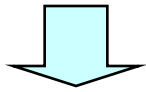


# The use of “.” – “Element” Operation

```
A = [1 2 3; 5 1 4; 3 2 1]
```

A =

1	2	3
5	1	4
3	2	-1



```
x = A(1,:)
```

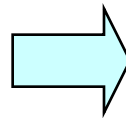
x=

1	2	3
---	---	---

```
y = A(3 ,:)
```

y=

3	4	-1
---	---	----



```
b = x .* y
```

b=

3	8	-3
---	---	----

```
d = x .^2
```

d=

1	4	9
---	---	---