



# Internet of Things: Theory and Applications

## Module 1: C++ Programming – Part B

*Introduced by: Eng. Mohamed Hatem*

Teaching Assistant at the Faculty of Computers and Data Science in Alexandria University

Graduate of Nanotechnology and Nano-electronics Engineering Program at the Zewail City of Sciences, Technology, and Innovation

Graduate Student Member at IEEE

# Session Outline:

- Section 0: Arrays
- Section 1: Pointers
- Section 2: Strings
- Section 3: Object Oriented Programming using C++
- Section 4: Assignments



# Section 0:

## Arrays

**Internet of Things: Theory and Applications**

# What is an Array?

- An array is used to process a collection of data all of which is of the same type, such as a list of temperatures or a list of names.
- An **array** behaves like a list of variables with a uniform naming mechanism that can be declared in a single line of simple code.
- How to declare an array:

*Datatype ArrayName [ArraySize] ;*

Example: *int score[5];*

- Which declare the following variables:

*int score[0]; int score[1]; int score[2]; int score[3]; int score[4];*

- These variables are called indexed variables, subscripted variables, or elements. Then, the number in the square brackets is called index or subscript.
- The number of indexed variables in an array is called the **declared size/size** of the array, which is an always positive integer value.
- *int* is the base type of the array which is the data type of all indexed elements.

# More on Arrays

- you can declare arrays and regular variables together. For example, the following declares the two *int* variables next and max in addition to the array score:

```
int next, score[5], max;
```

- The index inside the square brackets need not be given as an integer constant.

```
int n = 2;  
score[n + 1] = 99;
```

- When an index expression evaluates to some value other than those allowed by the array declaration, the index is said to be out of range or simply **illegal**.
- Many compilers you cannot use a variable for the array size, such as the following:

```
cout << "Enter number of students:\n";  
cin >> number;  
int score[number]; //ILLEGAL ON MANY COMPILERS!
```

# More on Array initialization

- Arrays can be initialized during declaration
  - In this case, it is not necessary to specify the size of the array
    - Size determined by the number of initial values in the braces
- Example 1: `int Items[] = {12, 32, 16, 23, 45};`
- Example 2: `int items [10] = { 0 };`
- Example 3: `int items [10] = { 5, 7, 10 };`

# More on Arrays

- After declaring the array you can use the For .. Loop to initialize it with values submitted by the user.
- Using *for* loops to access array elements:

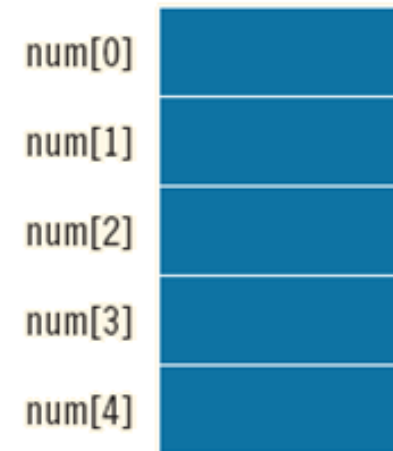
```
for (int i = 0; i < 10; i++)  
    //process list[i]
```

- Example:

```
for (int i = 0; i < 10; i++)  
    cin >> list[i];
```

# Tip: Using for Loops and Arrays

```
for (i = 0; i < 5; i++)  
cout << score[i] << " off by "  
    << (max - score[i]) << endl;
```





# Array in Memory

- A computer's memory consists of a list of numbered locations called bytes.
- The number of a byte is known as its address.
- A simple variable is implemented as a portion of memory consisting of some number of consecutive bytes.
- The number of bytes is determined by the type of the variable.
- When your program stores a value in the variable, what really happens is that the value (coded as 0s and 1s) is placed in those bytes of memory that are assigned to that variable.
- Remember: when a variable is given as a (call-by-reference) argument to a function, it is the address of the variable that is actually given to the calling function.
- The locations of the various array indexed variables are always placed next to one another in memory.

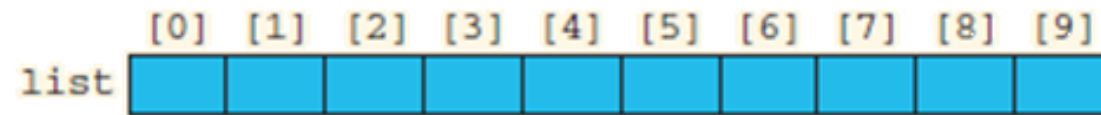
NB: a simple variable in memory is described by two pieces of information: an **address** in memory (giving the location of the first byte for that variable) and the type of the variable, which tells how many bytes of memory the variable requires.

# Array in Memory

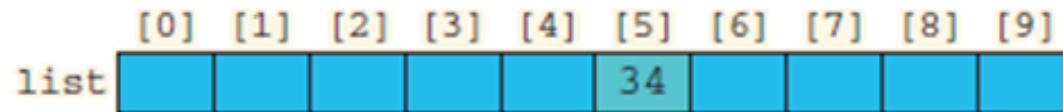
- *int* **a[6]**; When you declare this array, the computer reserves enough memory to hold six variables of type *int*.
- Computer then remembers the address of indexed variable **a[0]**, but it does not remember the address of any other indexed variable.
- Computer starts with the address of **a[0]** (which is a number). The computer then adds the number of bytes needed to hold three variables of type *int* to the number for the address of **a[0]**. The result is the address of **a[3]**.

# Array Initialization

Example: `int list[10];`



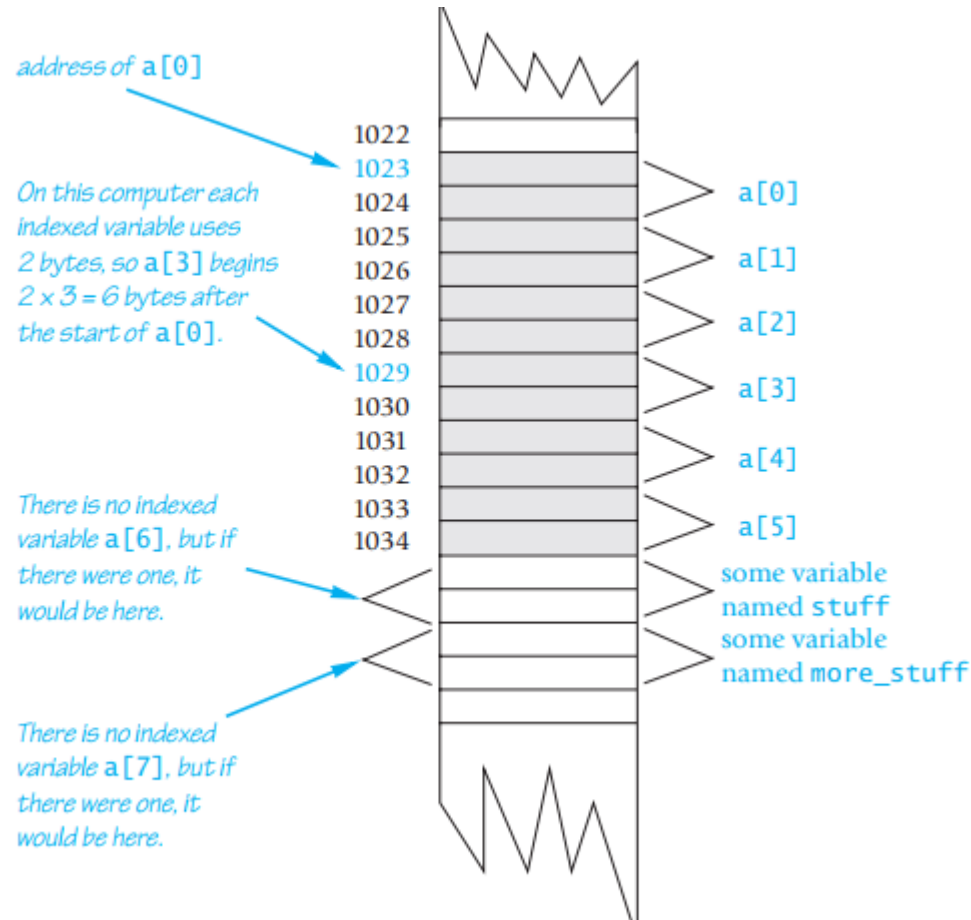
`list[5] = 34;`



`Cout << list [5];`

# Array in Memory

Viewed this way, an array has three parts: the address (location in memory) of the first indexed variable, the base type of the array (which determines how much memory each indexed variable uses), and the size of the array (that is, the number of indexed variables).



# Array Initialization

- An array can be initialized when it is declared. When initializing the array, the values for the various indexed variables are enclosed in braces and separated with commas.

Example: `int children[3] = {2, 12, 1};`

- This declaration is equivalent to the following code:

```
int children[3];  
children[0] = 2;  
children[1] = 12;  
children[2] = 1;
```

- NB: If you list fewer values than there are indexed variables, those values will be used to initialize the first few indexed variables, and the remaining indexed variables will be initialized to a 0 of the array base type.
- NB: Arrays with not initializer or other variable within any function won't be initialized.

# Array Initialization

- If you initialize an array when it is declared, you can omit the size of the array, and the array will automatically be declared to have the minimum size needed for the initialization values.

```
int b[ ] = {5, 12, 11};
```

# Arrays in Loops

- C++11 includes a new type of for loop, the range-based for loop, that simplifies iteration over every element in an array.
- Syntax:

```
    for (datatype varname : array)
    {
        // varname is successively set to each element in
        // the array
    }
```

# Example:

```
#include <iostream>

using namespace std;

int main()
{
    int arr[] = {2, 4, 6, 8};
    for (int x : arr)
        cout << x;
    cout << endl;

    return 0;
}
```

2468

Process returned 0 (0x0) execution time : 0.045 s  
Press any key to continue.



# Arrays in Loops

- The previous example is pass-by-value based loop.
- If we change x inside the loop it doesn't change the array.
- Define x as pass by reference using & and then changes to x will be made to the array.
- The following example below increments every element in the array and then outputs them.

# Example:

```
#include <iostream>

using namespace std;

int main()
{
    int arr[] = {2, 4, 6, 8};
    for (int& x : arr)
        x++;
    for (int x : arr)
        cout << x;
    cout << endl;
    return 0;
}
```

3579

Process returned 0 (0x0) execution time : 0.027 s  
Press any key to continue.

# Array in Functions

- you can use both array indexed variables and entire arrays as arguments to functions.
- An indexed variable can be an argument to a function in exactly the same way that any variable can be an argument. For example, suppose a program contains the following declarations:  

```
int i, n, a[10];
```
- The indexed expression shall be evaluated in order to determine exactly which indexed variable is given as the argument and can't be a variable.

# Example: Indexed Variable as a Function

- Given the allowed vacation days for a group of employees, the program aims to adjust the number of vacation days for each employee by adding 5 additional days. It then displays the revised number of vacation days for each employee.

- **Input:**

Program prompts the user to enter the allowed vacation days for employees 1 through 3.

- **Output:**

Program displays the revised number of vacation days for each employee after adjusting them by adding 5 days.

# Solution

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

const int NUMBER_OF_EMPLOYEES = 3;

int adjust_days(int old_days);

int main()
{
    int vacation[NUMBER_OF_EMPLOYEES], number;
    cout << "Enter allowed vacation days for employees 1 through " << NUMBER_OF_EMPLOYEES <<
    ":\n";
    for (number = 1; number <= NUMBER_OF_EMPLOYEES; number++)
        cin >> vacation[number - 1];
```

# Solution

```
for (number = 0; number < NUMBER_OF_EMPLOYEES; number++)
    vacation[number] = adjust_days(vacation[number]);
cout << "The revised number of vacation days are:\n";
for (number = 1; number <= NUMBER_OF_EMPLOYEES; number++)
    cout << "Employee number " << number << " vacation days = " <<
vacation[number - 1] << endl;
return 0;
}

int adjust_days(int old_days)
{
    return (old_days + 5);
}
```

# Output

```
Enter allowed vacation days for employees 1 through 3:  
4  
5  
6  
The revised number of vacation days are:  
Employee number 1 vacation days = 9  
Employee number 2 vacation days = 10  
Employee number 3 vacation days = 11  
  
Process returned 0 (0x0)   execution time : 2.498 s  
Press any key to continue.
```

# Arrays in Function: Entire Array as Function

- A function can have a formal parameter for an entire array so that when the function is called, the argument that is plugged in for this formal parameter is an entire array.
- However, a formal parameter for an entire array is neither a call-by-value parameter nor a call-by-reference parameter; it is a new kind of formal parameter referred to as an array parameter.
- An **array argument** is, of course, an array that is plugged in for an array parameter, such as `a[ ]`.
- The formal parameter `a` is merely a placeholder for the argument score.



# Arrays in Function: Entire Array as Function

- When an array is used as an argument in a function call, any action that is performed on the array parameter is performed on the array argument, so the values of the indexed variables of the array argument can be changed by the function.
- If the formal parameter in the function body is changed (for example, with a cin statement), then the array argument will be changed. 4
- That same function can be used to fill an array of any size, as long as the base type of the array is int.
- Example: `void sum_array(double sum, double a[ ], int size);`
- Array parameters are different from call-by-reference parameters, but why?

# Array Parameters & Call-by-reference Parameters

- When an array is used as an array argument to a function, only memory location and base type is given to the function.
- When an array argument is plugged in for its corresponding formal parameter, all that is plugged in is the address of the array's first indexed variable and the base type of the array argument must match the base type of the formal parameter, so the function also knows the base type of the array.
- The array argument does not tell the function the size of the array.
- When the code in the function body is executed, the computer knows where the array starts in memory and how much memory each indexed variable uses, but(unless you make special provisions) it does not know how many indexed variables the array has.
- That is why it is critical that you always have another *int* argument telling the function the size of the array. That is also why an array parameter is not the same as a call-by-reference parameter.

# Example: Function with Array Parameter

You are tasked with designing a program that works with arrays. The program should fulfill the following requirements:

- Declare an array of integers with a fixed size.
- Prompt the user to enter values for each element of the array.
- Display the array elements to the user.
- Implement a function that accepts the array as a parameter and prints its elements.
- Modify the array by doubling the value of each element.
- Display the modified array to the user.

Your task is to write the necessary code to meet these requirements. Ensure that the program adheres to best practices, such as proper variable naming and code readability.

Once you have completed the code, test it with different array sizes and values to ensure its correctness.

# Solution

```
#include <iostream>

using namespace std;

void print_array(int arr[], int size)
{
    for (int i = 0; i < size; i++)
    {
        cout << arr[i] << " ";
    }
    cout << endl;
}

int main()
{
    const int SIZE = 5;
    int numbers[SIZE] = {1, 2, 3, 4, 5};
```

# Solution

```
cout << "Array before modification: ";  
    print_array(numbers, SIZE);  
  
    // Modify the array  
    for (int i = 0; i < SIZE; i++)  
    {  
        numbers[i] *= 2;  
    }  
  
    cout << "Array after modification: ";  
    print_array(numbers, SIZE);  
  
    return 0;  
}
```

# Output

```
Array before modification: 1 2 3 4 5  
Array after modification: 2 4 6 8 10  
  
Process returned 0 (0x0)   execution time : 0.125 s  
Press any key to continue.
```

# The *const* Parameter Modifier

- To tell the compiler that an array argument should not be changed by your function, you insert the modifier *const* before the array parameter for that argument position.
- An array parameter that is modified with a *const* is called a **constant array parameter**.
- you normally have a function declaration in your program in addition to the function definition. When you use the *const* modifier in a function definition, you must also use it in the function declaration so that the function heading and the function declaration are consistent.
- Example:

```
void show_the_world(const int a[ ], int size_of_a)
```

# Multidimensional Arrays

- In C++, you can declare multidimensional arrays such as:

## **Syntax:**

*Type Array\_Name[Size\_Dim\_1][Size\_Dim\_2]...[Size\_Dim\_Last];*

An array declaration, of the form shown, defines one indexed variable for each combination of array indexes.



# Two Dimensional Arrays

- A two-dimensional array in C++ is a data structure that represents a collection of elements arranged in a two-dimensional grid or matrix. It can be visualized as a table or a grid with rows and columns.
- Syntax:

`datatype arrayName[rowSize][columnSize];`

- Example for declaring a two dimensional array:

`int element = matrix[1][2];`

	0	1	2	3
0				
1				
2				
3				
4			20	
5				

# Two Dimensional Arrays

- Two dimensional array initialization example:
- Example: `int matrix[3][4] = {{1, 2, 3, 4},  
{5, 6, 7, 8},  
{9, 10, 11, 12}};`
- After declaring the array you can use the For .. Loop to initialize it with values submitted by the user.
- Example: Using 2 nested *for* loops to access array elements:

```
for (int row = 0; row < 6; row++)  
    for (int col = 0; col < 4; col++)  
        cin >> marks[ row ][col];
```

# Re-call

- **Array** : is a collection of fixed number of elements, wherein all of elements have same data type.
- **Array Basics:**
  - Consecutive group of memory locations that all have the same type.
  - The collection of data is indexed, or numbered, and it starts at 0 and The highest element index is one less than the total number of elements in the array.
- **One-dimensional array:**
  - elements are arranged in list form.
- **Multi-dimensional array:**
  - elements are arranged in tabular form.

# Practice Problem:

- Write a program that prompts the user to enter five integers and stores them in an array. Then, create the following functions:
- 1) void displayArray(int arr[], int size): This function takes an array and its size as parameters and displays the elements of the array.
- 2) int sumArray(int arr[], int size): This function takes an array and its size as parameters and returns the sum of all the elements in the array.
- 3) int findMax(int arr[], int size): This function takes an array and its size as parameters and returns the maximum value in the array.
- 4) void reverseArray(int arr[], int size): This function takes an array and its size as parameters and reverses the elements in the array.

# Solution

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

void displayArray(int arr[], int size)
{
    cout << "Array elements: ";
    for (int i = 0; i < size; i++)
    {
        cout << arr[i] << " ";
    }
    cout << endl;
}
```

# Solution

```
int sumArray(int arr[], int size)
{
    int sum = 0;
    for (int i = 0; i < size; i++)
    {
        sum += arr[i];
    }
    return sum;
}
```

# Solution

```
int findMax(int arr[], int size)
{
    int max = arr[0];
    for (int i = 1; i < size; i++)
    {
        if (arr[i] > max)
        {
            max = arr[i];
        }
    }
    return max;
}
```

# Solution

```
void reverseArray(int arr[], int size)
{
    int start = 0;
    int end = size - 1;
```



# Solution

```
while (start < end)
{
    int temp = arr[start];
    arr[start] = arr[end];
    arr[end] = temp;
    start++;
    end--;
}
```

# Solution

```
int main()
{
    const int SIZE = 5;
    int numbers[SIZE];

    cout << "Enter five integers:\n";
    for (int i = 0; i < SIZE; i++)
    {
        cin >> numbers[i];
    }
}
```

# Solution

```
displayArray(numbers, SIZE);

int sum = sumArray(numbers, SIZE);
cout << "Sum of array elements: " << sum << endl;

int max = findMax(numbers, SIZE);
cout << "Maximum element in the array: " << max << endl;

reverseArray(numbers, SIZE);
displayArray(numbers, SIZE);

return 0;
}
```

# Output

```
Enter five integers:
1
2
3
4
5
Array elements: 1 2 3 4 5
Sum of array elements: 15
Maximum element in the array: 5
Array elements: 5 4 3 2 1

Process returned 0 (0x0)   execution time : 9.180 s
Press any key to continue.
```



# Section 1: Pointers

**Internet of Things: Theory and Applications**

# About Pointers

- A **pointer** is the memory address of a variable.
- Recall that the computer's memory is divided into numbered memory locations (called bytes) and that variables are implemented as a sequence of adjacent memory locations.
- An address that is used to name a variable in this way (by giving the address in memory where the variable starts) is called a *pointer* because the address can be thought of as “pointing” to the variable.
- For example, when the variable is used as a call-by reference argument, it is this address, not the identifier name of the variable, that is passed to the calling function.

# More on Pointers: Pointer Variables

- when a variable is a call-by-reference argument in a function call, the function is given this argument variable in the form of a pointer to the variable.
- A pointer can be stored in a variable.
- However, even though a pointer is a memory address and a memory address is a number, you cannot store a pointer in a variable of type *int* or *double* without type casting.
- A variable to hold a pointer must be declared to have a pointer type. For example, the following declares *p* to be a pointer variable that can hold one pointer that points to a variable of type *double*: *double \*p*
- *NB: The variable *p* can hold pointers to variables of type *double*, but it cannot normally contain a pointer to a variable of some other type, such as *int* or *char**

# More on Pointers: Pointer Variables

- You can use the operator & to determine the address of a variable, and you can then assign that address to a pointer variable.
- Syntax: `Type_Name *Variable_Name1, *Variable_Name2, . . . ;`
- Example: `double *pointer1, *pointer2;`
- When the asterisk is used in this way, it is often called the **dereferencing operator**, and the pointer variable is said to be **dereferenced**.



# Remember!

- A pointer is an address, and an address is an integer, but a pointer is not an integer. That is not crazy. That is abstraction!
- Why?

C++ insists that you use a pointer as an address and that you not use it as a number. A pointer is not a value of type *int* or of any other numeric type. You normally cannot store a pointer in a variable of type *int*. If you try, most C++ compilers will give you an error message or a warning message. Also, you cannot perform the normal arithmetic operations on pointers. (You can perform a kind of addition and a kind of subtraction on pointers, but they are not the usual integer addition and subtraction.)

You now have two ways to refer to `v1`: You can call it `v1` or you can call it “the variable pointed to by `p1`.”

# Example on Pointer Variables and Dereferencing:

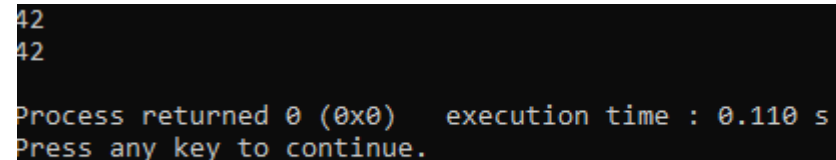
```
#include <iostream>
using namespace std;

int main()
{
    double v1 = 0; // Declare and initialize a double variable v1 with value 0
    double* p1 = &v1; // Declare a pointer to double and assign the address of v1 to it

    *p1 = 42; // Dereference the pointer p1 to access the value it points to, and assign it a value of 42

    cout << v1 << endl; // Print the value of v1, which is 42
    cout << *p1 << endl; // Print the value pointed to by p1, which is also 42

    return 0;
}
```



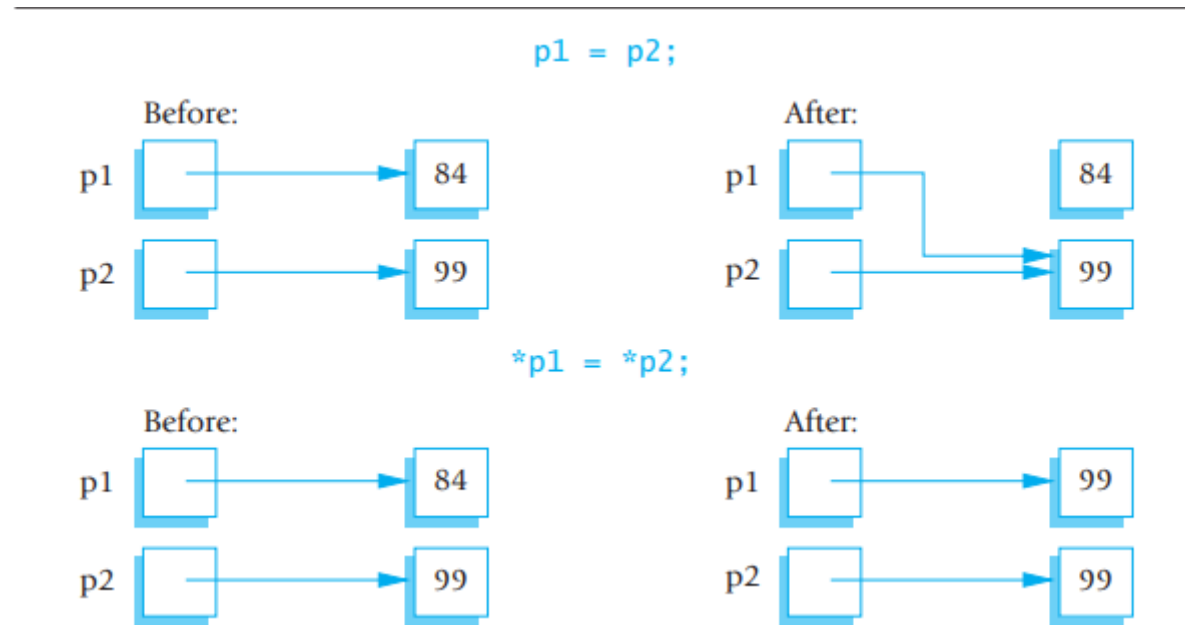
A screenshot of a terminal window with a black background and white text. The first two lines show the output '42' on separate lines. The third line shows 'Process returned 0 (0x0) execution time : 0.110 s'. The fourth line shows 'Press any key to continue.'.

```
42
42
Process returned 0 (0x0) execution time : 0.110 s
Press any key to continue.
```

# More Pointer Operations

- You can assign the value of one pointer variable to another pointer variable. Example: `p2 = p1;`
- This is not the same as `*p2 = *p1`
- When you add the asterisk, you are not dealing with the pointers `p1` and `p2`, but with the variables that the pointers are pointing to.
- The operator *new* can be used to create variables that have no identifiers to serve as their names. These nameless variables are referred to via pointers.
- For example, the following creates a new variable of type *int* and sets the pointer variable `p1` equal to the address of this new variable (that is, `p1` points to this new, nameless variable): Code: `p1 = new int;`
- This new, nameless variable can be referred to as `*p1` (that is, as the variable pointed to by `p1`).
- Variables that are created using the *new* operator are called **dynamic variables** because they are created and destroyed while the program is running.

# Example:



# Example on Dynamic Variables

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

int main()
{
    int *p1, *p2; // Declare two pointers to int

    p1 = new int; // Allocate memory for an int and assign the address to p1
    *p1 = 42; // Assign a value of 42 to the memory location pointed by p1

    p2 = p1; // Assign the value of p1 (the address of the dynamically allocated memory) to p2

    cout << "*p1 == " << *p1 << endl; // Print the value at the memory location pointed by p1 (42)
    cout << "*p2 == " << *p2 << endl; // Print the value at the memory location pointed by p2 (42)
```

# Example on Dynamic Variables

```
*p2 = 53; // Assign a value of 53 to the memory location pointed by p2

cout << "*p1 == " << *p1 << endl; // Print the updated value at the memory location pointed by p1 (53)
cout << "*p2 == " << *p2 << endl; // Print the updated value at the memory location pointed by p2 (53)

p1 = new int; // Allocate memory for another int and assign the address to p1
*p1 = 88; // Assign a value of 88 to the memory location pointed by p1

cout << "*p1 == " << *p1 << endl; // Print the value at the new memory location pointed by p1 (88)
cout << "*p2 == " << *p2 << endl; // Print the previous value at the memory location pointed by p2 (53)

cout << "Hope you got the point of this example!\n";

return 0;}
```

# Basic Memory Management

- A special area of memory, called the **freestore**, is reserved for dynamic variables. Any new dynamic variable created by a program consumes some of the memory in the freestore or heap.
- The *delete* operator eliminates a dynamic variable and returns the memory that the dynamic variable occupied to the freestore so that the memory can be reused.
- For example, the following eliminates the dynamic variable pointed to by the pointer variable p: *delete p*

# Note on Dangling Pointers

if some other pointer variable was pointing to the dynamic variable that was destroyed, then this other pointer variable is also undefined.

These undefined pointer variables are called **dangling pointers**. If `p` is a dangling pointer and your program applies the dereferencing operator `*` to `p` (to produce the expression `*p`), the result is unpredictable and usually disastrous.



# Automatic vs Static Variables

- The ordinary variables that we have been using (that is, the variables declared within main or within some other function definition) are called **automatic variables (ordinary variables)**.
- Automatic variables are automatically created when the function in which they are declared is called and automatically destroyed when the function call ends.
- *Remember: Global variables are variables that are declared outside of any function definition (including being outside of main).*

# Automatic vs Static Variables

- Static variable is a variable that is associated with a class, function, or block scope, but it retains its value even after the scope in which it is defined has ended. It has two main types:
  - 1) Class static variable.
  - 2) Function static variable.

# Pointer Types

- You can define a pointer type name so that pointer variables can be declared like other variables without the need to place an asterisk in front of each pointer variable.
- You can define the pointer type using the following:

```
typedef int* IntPtr;  
IntPtr p;
```

- An easier and less error-prone way to declare both p1 and p2 to be pointer variables is to use the defined type name IntPtr as follows:

```
IntPtr p1, p2;
```

# Pointer Types

- Advantage of using a defined pointer type, such as `IntPtr`, is seen when you define a function with a call-by-reference parameter for a pointer variable.
- Example: `void sample_function(IntPtr& pointer_variable);`

# Practice

Write a C++ program that defines a function called `swapValues` that takes two integer pointers as parameters and swaps the values stored at those memory locations. In the main function, prompt the user to enter two integers and then call the `swapValues` function to swap the values. Finally, display the swapped values.

# Solution

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

void swapValues(int* ptr1, int* ptr2)
{
    int temp = *ptr1;
    *ptr1 = *ptr2;
    *ptr2 = temp;
}

int main()
{
```

# Solution

```
int num1, num2;

cout << "Enter the first integer: ";
cin >> num1;

cout << "Enter the second integer: ";
cin >> num2;

int* ptr1 = &num1;
int* ptr2 = &num2;

swapValues(ptr1, ptr2);

cout << "Swapped values: " << *ptr1 << " " << *ptr2 << endl;

return 0;}
```

# Output

```
Enter the first integer: 3
Enter the second integer: 4
Swapped values: 4 3

Process returned 0 (0x0)   execution time : 2.136 s
Press any key to continue.
```





# Section 2:

# Strings

**Internet of Things: Theory and Applications**

# Representing Strings: Method 1: Using Arrays

- One way to represent a string is as an array with base type *char*.
- If the string is "Hello", it is handy to represent it as an array of characters with six indexed variables: five for the five letters in “Hello” plus one for the character '\0', which serves as an end marker.
- The character '\0' is called the **null character** and is used as an end marker because it is distinct from all the “real” characters.

# Representing Strings: Using Arrays

- A string stored in this way (as an array of characters terminated with '\0') is called a **C-string variable**.
- A **C-string variable** is just an array of characters.
- Thus, the following array declaration provides us with a C-string variable capable of storing a C-string value with nine or fewer characters: *char s[10];*
- C-string variable places the special symbol '\0' in the array immediately after the last character of the C-string. Thus, if s contains the string "Hi Mom"!, then the array elements are filled as shown here:

s[0]	s[1]	s[2]	s[3]	s[4]	s[5]	s[6]	s[7]	s[8]	s[9]
H	I		M	o	m	!	\0	?	?

# Representing Strings: Using Arrays

- A C-string variable is an array, so it has **indexed variables** that can be used just like those of any other array.
- **Syntax:** *char Array\_Name[Maximum\_C\_string\_Size + 1];*
- **Example:** *char my\_message[20] = "Hi there.";*
- **NB:** When you initialize a C-string variable, you can omit the array size. C++ will automatically make the size of the C-string variable 1 more than the length of the quoted string.
- **NB:** *char short\_string[] = {'a', 'b', 'c'};* and *char short\_string[] = "abc";* are not the same!

# C-strings in Loops

- A C-string inside a while loop could be a good way to define strings but the while loop shall change characters until it encounters a '\0' then stops.
- *NB: If the loop never encounters a '\0', then it could change a large chunk of memory to some unwanted values, which could make your program do strange things.*
- It's better to write as follows to avoid these unwanted values:
- Example:

```
int index = 0;
while ( (our_string[index] != '\0') && (index < SIZE) )
{
    our_string[index] = 'X';
    index++;
}
```

# Be Careful with the Null Character

- After this code is executed, the array `happy_string` (defined below) will still contain the six letters in the C-string "DoBeDo", but `happy_string` will no longer contain the null character `'\0'` to mark the end of the C string.
- Many string-manipulating functions depend critically on the presence of `'\0'` to mark the end of the C-string value so this may cause errors while using these functions.
- Example: `Char happy_string[7] = "DoBeDo";`  
`happy_string[6] = 'Z';`

# Assignments on C-string

- If you use **== to test C strings for equality, you will not get the result you expect.** The reason for these problems is that C strings and C-string variables are arrays!
- Example:

LEGAL	ILLEGAL
<pre>char happy_string[7] = "DoBeDo";</pre> <p>This is an initialization</p>	<pre>char a_string[10]; a_string = "Hello";</pre> <p>This is an assignment.</p>

# Important Functions on C-string

- There are a number of different ways to assign a value to a C-string variable. The easiest way is to use the predefined function strcpy as shown: `strcpy(a_string, "Hello");`
- To test whether two C strings are the same, you can use the predefined function strcmp.

For example:

```
if (strcmp(c_string1, c_string2))  
    cout << "The strings are NOT the same.";  
else  
    cout << "The strings are the same.";
```

**NB:** The function strcmp works differently than you might guess. The comparison is true if the strings do not match. The function strcmp compares the characters in the C-string arguments a character at a time. If at any point the numeric encoding of the character from c\_string1 is **less than** the numeric encoding of the corresponding character from c\_string2, the testing stops and a **negative number is returned**. If the character from c\_string1 is **greater than** the character from c\_string2, then a **positive number** is returned and if the C strings are the **same, a 0 is returned** !



# Functions on C-string

- Inside control flows (based on logical operations), the **nonzero value will be converted to true if the strings are different, and the zero will be converted to false.** Be sure that you remember this inverted logic in your testing for C-string equality!
- To concatenate two strings: `char string_var[20] = "The rain";  
strcat(string_var, "in Spain");`
- The functions **strcpy** and **strcmp** are in the library with the header file `<cstring>`, so to use them you would insert the following near the top of the file:

```
#include <cstring>
```

# Arguments & Parameters on C-String

- A C-string variable is an array, so a C-string parameter to a function is simply an array parameter.
- In the context of arrays, including C-strings, it is recommended to have an additional integer parameter in functions that modify the value of a C-string parameter. This extra parameter should indicate the size of the C-string variable being passed. This practice ensures safety and helps prevent buffer overflows or memory access issues.

# Arguments & Parameters on C-String

- In contrast, if a function only needs to access the value of a C-string argument without modifying it, there is **no requirement to include an additional parameter indicating the size of the C-string variable or the amount of the C-string array that is filled.**
- Instead, the null character '\0' can be used to determine the end of the C-string value stored in the C-string variable.
- By checking for the presence of the null character, the function can identify the end of the C-string without needing an explicit size parameter.

# Output using C-string

- C-strings can be output using the insertion operator (<<).
- Example:

```
cout << "Hello, " << name << "!"; (where name is a  
C-string variable)
```

```
cout << "The result is: " << result << endl; (where  
result is a C-string variable)
```

# Input using C-string

- C-strings can be filled using the input operator (>>).
- Example:

`cin >> input;` (where input is a C-string variable)

`cin >> username >> password;` (where username and password are C-string variables)

# Reading an Entire Line of Input in C-string

- The `getline` function can be used to read a full line of input and store it in a C-string variable.
- Example:

`getline(cin, sentence);` (where `sentence` is a C-string variable)

`getline(file, line);` (where `file` is a file input stream and `line` is a C-string variable)

# C-String-to-Number Conversions and Robust Input

- The C string "1234" and the number 1234 are not the same things. The first is a sequence of characters; the second is a number. In everyday life, we write them the same way and blur this distinction, but in a C++ program this distinction cannot be ignored.
- Your program then needs to convert this C string of digits to a number, which can easily be done with the predefined function `atoi`.
- The function `atoi` takes one argument that is a C string and returns the *int* value that corresponds to that C string. For example, `atoi("1234")` returns the integer 1234.
- The function `atoi` is in the library with header file `cstdlib`, so any program that uses it must contain the following directive:  
`#include <cstdlib>`
- NB: Function `atoi("#37")` returns 0, because the character '#' is not a digit.

# More Functions on C-String-to-Number Conversions

- The function `read_and_clean` shown will delete any nondigits from the string typed in, but it cannot check that the remaining digits will yield the number the user has in mind.
- Moreover, the `get_int` that reads in a number of type *double*, as opposed to a number of type *int*.
- Moreover, `atof ("9.99")` returns the value 9.99 of type double. If the argument does not correspond to a number of type double, then `atof` returns 0.0."



# Method 2: Standard String Class

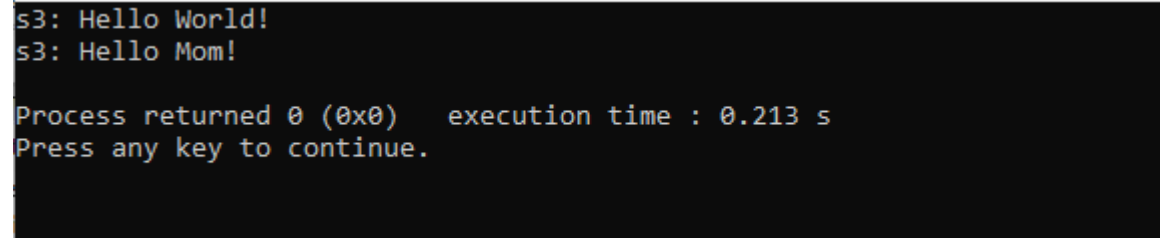
- The class string is defined in the library whose name is also <string>, and the definitions are placed in the std namespace. So, in order to use the class string, your code must contain the following (or something more or less equivalent):  
`#include <string>`  
`using namespace std;`
- You can use the = operator to assign a value to a string variable, and you can use the + sign to concatenate two strings.

# Standard String Class

- In C++, you can assign values to string variables using the = operator and concatenate strings using the + operator.
- When concatenating strings, if the resulting string exceeds the capacity of the destination string variable, more space is automatically allocated.
- Quoted strings, which are actually C strings, can be automatically typecast to values of type string.

# Example:

```
#include <iostream>
#include <string>
using namespace std;
int main() {
    string s1 = "Hello";
    string s2 = " World!";
    string s3;
    // Concatenating strings using the + operator
    s3 = s1 + s2;
    cout << "s3: " << s3 << endl; // Output: s3: Hello World!
    // Assigning a quoted string to a string variable
    s3 = "Hello Mom!";
    cout << "s3: " << s3 << endl; // Output: s3: Hello Mom!
    return 0;}
```



```
s3: Hello World!
s3: Hello Mom!

Process returned 0 (0x0)   execution time : 0.213 s
Press any key to continue.
```

# Standard String Class

The string class in C++ has a **default constructor that initializes a string object to an empty string.** It also has a constructor that takes a single argument, which can be a standard C string (quoted string).

This constructor initializes the string object with the same characters as the C string argument.

For example, `s3 = "Hello Mom!";` sets the value of the string variable `s3` to a string object with the same characters as in the C string "Hello Mom!".

# Standard String Class

Example:

```
String phrase;  
string noun("ants");
```

The first line declares the string variable `phrase` and initializes it to the empty string. The second line declares `noun` to be of type `string` and initializes it to a string value equivalent to the C string `"ants"`.

**NB: The quoted string "ants" is a C string, not a value of type string.**

# Standard String Class

- You can also use an alternate notation to declare a string variable and invoke the constructor. The following two lines are equivalent:

```
string noun("ants");  
string noun = "ants";
```

# Example:

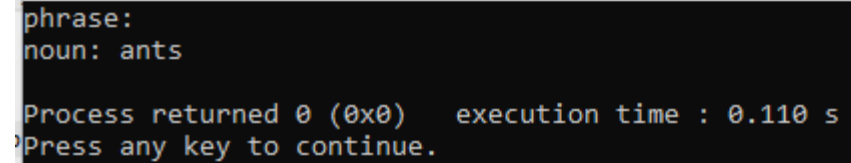
```
#include <iostream>
#include <string>

using namespace std;

int main() {
    string phrase;
    string noun("ants");

    cout << "phrase: " << phrase << endl; // Output: phrase:
    cout << "noun: " << noun << endl;      // Output: noun: ants

    return 0;
}
```

A screenshot of a terminal window with a black background and white text. It shows the output of the C++ program: 'phrase:' followed by a newline, and 'noun: ants' followed by a newline. Below this, it shows 'Process returned 0 (0x0) execution time : 0.110 s' and 'Press any key to continue.'

```
phrase:
noun: ants

Process returned 0 (0x0) execution time : 0.110 s
Press any key to continue.
```

# Input & Output using String Class

- cin work the same for string objects as for other data, but remember that the extraction operator **ignores initial whitespace and stops reading when it encounters more whitespace.**
- This is as true for strings as it is for other data. For example, consider the following code;  

```
string s1, s2;  
cin >> s1;  
cin >> s2;
```
- If the user types in:  

```
May the hair on your toes grow long and curly!
```
- Then s1 will receive the value "May" with any leading (or trailing) whitespace deleted.
- The variable s2 receives the string "the".
- Using the extraction operator >> and cin, you can only read in words; you cannot read in a line or other string that contains a blank. Sometimes this is exactly what you want, but sometimes it is not at all what you want.



# Input & Output using String Class

- If you want your program to read an entire line of input into a variable of type string, you can use the function `getline`.
- The syntax for using `getline` with string objects is a bit different from what we described for C strings.
- You do not use `cin.getline`; instead, you make `cin` the first argument to `getline`. (Thus, this version of `getline` is not a member function.)
- Example:

```
string line;  
cout << "Enter a line of input:\n";  
getline(cin, line);  
cout << line << "END OF OUTPUT\n";
```

When embedded in a complete program, this code produces a dialogue like the following:

```
Enter some input:  
Do bedo to you!  
Do bedo to you!END OF OUTPUT
```

# Input & Output using String Class

- When the `getline` sees the `'\n'` and stops reading, so `getline` reads an empty string.
- If you find your program appearing to mysteriously ignore input data, see if you have mixed these two kinds of input.
- To skip and discard the remaining characters in a line of input, including the newline character `'\n'`, you can use the `ignore` member function from the `iostream` library.
- By providing the arguments `cin.ignore(1000, '\n');`, the `ignore` function will read and discard the entire rest of the line up to and including the `'\n'` character. If the end of the line is not found after 1000 characters, it will continue discarding until the newline character is encountered.

# Note for Equality of Strings

- **= and == Are different for strings and C Strings:**
- The operators =, ==, !=, <, >, <=, >=, when used with the standard C++ type string, produce results that correspond to our intuitive notion of how strings compare. They do not misbehave as they do with the C strings.

# Functions in cstring

Function	Description	Cautions
<code>strcpy(Target_String_Var, Src_String)</code>	Copies the C-string value <i>Src_String</i> into the C-string variable <i>Target_String_Var</i> .	Does not check to make sure <i>Target_String_Var</i> is large enough to hold the value <i>Src_String</i> .
<code>strncpy(Target_String_Var, Src_String, Limit)</code>	The same as the two-argument <code>strcpy</code> except that at most <i>Limit</i> characters are copied.	If <i>Limit</i> is chosen carefully, this is safer than the two-argument version of <code>strcpy</code> . Not implemented in all versions of C++.
<code>strcat(Target_String_Var, Src_String)</code>	Concatenates the C-string value <i>Src_String</i> onto the end of the C string in the C-string variable <i>Target_String_Var</i> .	Does not check to see that <i>Target_String_Var</i> is large enough to hold the result of the concatenation.
<code>strncat(Target_String_Var, Src_String, Limit)</code>	The same as the two-argument <code>strcat</code> except that at most <i>Limit</i> characters are appended.	If <i>Limit</i> is chosen carefully, this is safer than the two-argument version of <code>strcat</code> . Not implemented in all versions of C++.
<code>strlen(Src_String)</code>	Returns an integer equal to the length of <i>Src_String</i> . (The null character, '\0', is not counted in the length.)	
<code>strcmp(String_1, String_2)</code>	Returns 0 if <i>String_1</i> and <i>String_2</i> are the same. Returns a value < 0 if <i>String_1</i> is less than <i>String_2</i> . Returns a value > 0 if <i>String_1</i> is greater than <i>String_2</i> (that is, returns a nonzero value if <i>String_1</i> and <i>String_2</i> are different). The order is lexicographic.	If <i>String_1</i> equals <i>String_2</i> , this function returns 0, which converts to <i>false</i> . Note that this is the reverse of what you might expect it to return when the strings are equal.
<code>strncmp(String_1, String_2, Limit)</code>	The same as the two-argument <code>strcmp</code> except that at most <i>Limit</i> characters are compared.	If <i>Limit</i> is chosen carefully, this is safer than the two-argument version of <code>strcmp</code> . Not implemented in all versions of C++.

# String Processing with the Class string

- String objects in C++ offer the same operations as C strings with additional advantages.
- Characters in a string can be accessed using array square brackets.
- String objects behave like partially filled arrays with a length member function.
- Example:

```
string str = "Hello";  
char ch = str[1]; // Accessing the second character 'e'  
int len = str.length(); // Getting the length of the  
string (5)
```

# String Processing with the Class string

- Array square brackets do not check for illegal indexes in string objects.
- The member function `at` checks for illegal indexes and provides better error handling.
- Changing a character in a string can be done using indexed variables or the `at` function.
- Example:

```
string str = "Hello";  
str[3] = 'p'; // Changing the fourth character to 'p'  
char ch = str.at(1); // Accessing the second character  
'e'
```

# String Processing with the Class string

- String objects have various member functions, such as length, for string manipulation.
- String objects exhibit intuitive behavior with comparison operators like `==`, `<`, `>`, etc.
- Comparison operators use lexicographic ordering based on the ASCII character set.
- Note: The summarized content may need further refinement to fit within the specific slide limits and design considerations.

## Example:

```
string str1 = "apple";  
string str2 = "banana";  
bool isEqual = (str1 == str2); // Comparing if the  
strings are equal (false)  
bool isLessThan = (str1 < str2); // Comparing if  
str1 is less than str2 (true)
```



# Converting Between string Objects and C Strings

- When working with C strings and string objects in C++, automatic type conversion is supported from C strings to string objects. For example, assigning a C string to a string variable works seamlessly:

```
char a_c_string[] = "This is my C string.";
string string_variable;
string_variable = a_c_string;
```

# Converting Between string Objects and C Strings

- However, the reverse assignment is not allowed:

```
a_c_string = string_variable; // Illegal
```

- To obtain the C string representation of a string object, an explicit conversion is required using the `c_str()` member function. For example:

```
strcpy(a_c_string, string_variable.c_str()); //  
Legal
```

# Converting Between string Objects and C Strings

- Note that the `strcpy` function is used for copying, and `c_str()` returns the C string corresponding to the string object. The assignment operator does not work between C strings and string objects.
- Example:

```
char a_c_string[20];  
string string_variable = "Hello";  
strcpy(a_c_string, string_variable.c_str()); //  
Copying the string to a C string
```

In this example, the contents of the string object `string_variable` are copied to the C string `a_c_string` using `strcpy()` and `c_str()`.

# Converting Between Strings and Numbers

- Prior to C++11 it was a bit complicated to convert between strings and numbers, but in C++11 it is simply a matter of calling a function. Use stof, stod, stoi, or stol to convert a string to a float, double, int, or long, respectively. Use to\_string to convert a numeric type to a string. These functions are illustrated in the following example:

```
int i;  
double d;  
string s;  
i = stoi("35"); // Converts the string "35" to an integer 35  
d = stod("2.5"); // Converts the string "2.5" to the double  
2.5  
s = to_string(d*2); // Converts the double 5.0 to a string  
"5.0000"  
cout << i << " " << d << " " << s << endl;  
The output is 35 2.5 5.0000
```

# Practice Problem

You are given a string that represents a sentence. Your task is to write a function that takes this sentence as input and returns the number of words in the sentence. A word is defined as a sequence of characters separated by spaces.

Write a program that uses the string class and functions to solve this problem.

# Solution

```
#include <iostream>
```

```
#include <string>
```

```
using namespace std;
```

```
int countWords(const string& sentence) {
```

```
    int wordCount = 0;
```

```
    bool isWord = false;
```

# Solution

```
for (char c : sentence) {  
    if (c == ' ') {  
        isWord = false;  
    } else {  
        if (!isWord) {  
            wordCount++;  
            isWord = true;  
        }  
    }  
}
```

# Solution

```
return wordCount;  
}
```

```
int main() {  
    string sentence;  
    cout << "Enter a sentence: ";  
    getline(cin, sentence);  
  
    int numWords = countWords(sentence);  
    cout << "Number of words: " << numWords << endl;  
  
    return 0;  
}
```



# Output

```
Enter a sentence: mohamed hatem mohamed  
Number of words: 3  
  
Process returned 0 (0x0)   execution time : 4.999 s  
Press any key to continue.
```



# Section 3:

## OOP using C++

**Internet of Things: Theory and Applications**

# Main Content Items

- 1) Structs
- 2) Classes & Objects
- 3) Inheritance

# About OOP

- Object-oriented programming (OOP) is a programming paradigm that organizes data and behavior into reusable structures called objects. C++ is a powerful programming language that supports OOP concepts, making it widely used for building complex software systems.

# Benefits of OOP

- **Modularity:** OOP promotes modular design, breaking down complex systems into manageable and reusable components (objects and classes). This enhances code organization, maintenance, and scalability.
- **Code Reusability:** Objects and classes can be reused in different parts of a program or in other projects, reducing redundant code and increasing development efficiency.
- **Encapsulation and Data Security:** Encapsulation protects data by restricting access to it and providing controlled interactions through methods. This enhances data security and reduces the risk of unintentional data modifications.
- **Flexibility and Extensibility:** OOP allows for easy modifications and extensions to existing code. New functionality can be added by creating derived classes or overriding existing methods, without impacting other parts of the program.
- **Abstraction and Simplicity:** Abstraction allows complex systems to be represented at a higher level of abstraction, focusing on essential features and hiding unnecessary details. This simplifies program understanding and maintenance.

# Main OOP Block: a Class & Struct?

- **Class:**
- A class is a reference type that can have fields, properties, methods, events, and more. It serves as a blueprint for creating objects.
- Objects of a class are created with their default constructor, which initializes their fields to their default values (e.g., null for reference types).
- Classes support single inheritance and can participate in class hierarchies using inheritance and polymorphism.
- **Struct:**
- A struct is a value type that can also have fields, properties, and methods. It represents a lightweight structure that can hold data.
- Structs can be created without explicitly calling a constructor. In this case, they are initialized with default values (e.g., 0 for numeric types).
- Structs do not support inheritance. They cannot inherit from other structs or classes and cannot be base classes.



# Section 3-A: Structs

**Internet of Things: Theory and Applications**

# More on Classes, Structs, & Objects

- an object is a variable that has member functions, and a class is a data type whose variables are objects.
- A structure (of the kind discussed here) can be thought of as an object without any member functions. After you learn about structures, it will be a natural extension to define classes.



# Struct Syntax

```
struct StructName {  
    // Member variables (fields)  
    DataType1 member1;  
    DataType2 member2;  
    // ...  
  
    // Member functions (methods)  
    ReturnType method1(ParameterType1 param1, ParameterType2 param2) {  
        // Method implementation  
    }  
    // ...  
};
```

Example:

```
struct CDAccount  
{  
double balance;  
double interest_rate;  
int term; //months until maturity  
};
```

# On Struct Syntax

- The keyword `struct` announces that this is a structure type definition.
- The identifier **CDAccount** is the name of the structure type.
- The name of a structure type is called the **structure tag**. The tag can be any legal identifier (but not a keyword).
- Although this convention is not required by the C++ language, structure **tags are usually spelled with a mix of uppercase and lowercase letters, beginning with an uppercase letter.**
- The identifiers declared inside the braces, `{}`, are called **member names**.
- As illustrated in next example, a structure type definition ends with both a brace, `}`, and a semicolon.

# On Struct Syntax

- A structure definition is usually placed outside of any function definition (in the same way that globally defined constant declarations are placed outside of all function definitions).
- The structure type is then available to all the code that follows the structure definition.

# On Struct Syntax

- Once a structure type definition has been given, the structure type can be used just like the predefined types `int`, `char`, and so forth.
- For example, the following will declare two variables, named `my_account` and `your_account`, both of type `CDAccount`:

```
CDAccount my_account, your_account;
```

# On Struct Syntax

- A **structure value** is a collection of smaller values called **member values**.
- There is one member value for each member name declared in the structure definition.
- For example, a value of the type CDAccount is a collection of three member values: two of type double and one of type int. The member values that together make up the structure value are stored in member variables

# On Struct Syntax

- Each of these member names can be used to pick out one smaller variable that is a part of the larger structure variable. These smaller variables are called **member variables**.
- Member variables are specified by giving the name of the structure variable followed by a dot (that is, followed by a period) and then the member name.
- For example, if account is a structure variable of type CDAccount, then the structure variable account has the following three member variables:  
`account.balance`  
`account.interest_rate`  
`account.term`

# On Struct Syntax

- The first two member variables are of type *double*, and the last is of type *int*.
- These member variables can be used just like any other variables of those types.
- For example, the member variables above can be given values with the following three assignment statements:

```
account.balance = 1000.00;  
account.interest_rate = 4.7;  
account.term = 11;
```



# On Struct Syntax

- Member variables can be used in all the ways that ordinary variables can be used.
- For example, the following line from the program will add the value contained in the member variable `account.balance` and the value contained in the ordinary variable `interest` and will then place the result in the member variable `account.balance`:  
`account.balance = account.balance + interest;`
- The only difference is that in the case of structures, the members are variables rather than functions.

# On Struct Syntax

- Although we will not use this feature, you can combine member names of the same type into a single list separated by commas. For example, the following is equivalent to the previous structure definition:

```
struct Automobile  
{  
    int year, doors;  
    double horse_power;  
    char model;  
};
```


- **Variables of a structure type** can be declared in the same way as variables of other types. For example:  
`Automobile my_car, your_car;`
- The member variables are specified using the **dot operator**. For example,  
`my_car.year, my_car.doors, my_car.horse_power, and my_car.model.`

# Note on Syntax

- When you add the final brace, }, to a structure definition, it feels like the structure definition is finished, but it is not. You must also place a semicolon after that final brace. There is a reason for this, even though the reason is a feature that we will have no occasion to use.
- **Dot operator:** the **dot operator** is used to specify a member variable of a structure variable.

## SYNTAX

*Structure\_Variable\_Name*.*Member\_Variable\_Name*



```
struct StudentRecord
{
    int student_number;
    char grade;
};

int main()
{
    StudentRecord your_record;
    your_record.student_number = 2001;
    your_record.grade = 'A';
}
```

# Reusing Members Names

In C++, multiple structure types can have the same member names without causing any problems.

For example, you can define two structures with member names like quantity, but they will be distinct within their respective structure types.

When using structure variables, you can access the members by specifying the structure variable name followed by the dot operator and the member name.

Example:

```
struct FertilizerStock {  
    double quantity;  
    double nitrogen_content;  
};
```

```
struct CropYield {  
    int quantity;  
    double size;  
};
```

# Structs & Functions

- Functions can have call-by-value or call-by-reference parameters of a structure type.
- A structure type can be the return type of a function.

# Example:

```
struct CDAccount {  
    double balance;  
    double interest_rate;  
    int term;};  
  
CDAccount shrink_wrap(double the_balance, double the_rate, int the_term) {  
    CDAccount temp;  
    temp.balance = the_balance;  
    temp.interest_rate = the_rate;  
    temp.term = the_term;  
    return temp;}  
  
int main() {  
    CDAccount new_account;  
    new_account = shrink_wrap(10000.00, 5.1, 11);}
```

# Using Hierarchical structures

- Sometimes it makes sense to have structures whose members are themselves smaller structures.
- For example, a structure type called `PersonInfo`, which can be used to store a person's height, weight, and birth date, can be defined as following example:



# Example:

```
struct Date
{
int month;
int day;
int year;
};
struct PersonInfo
{
double height; //in inches
int weight; //in pounds
Date birthday;
};
```

# Initializing Structures

- Structures can be initialized at the time of declaration.
- Initialization is done by providing a list of member values enclosed in braces.
- The values should be provided in the order that corresponds to the order of member variables in the structure type definition.
- If there are more initializers than struct members, it is an error.
- If there are fewer initializer values than struct members, the remaining members are initialized to zero values.

# Example:

```
struct Date {  
    int month;  
    int day;  
    int year;  
};
```

```
Date due_date = {12, 31, 2004};
```



# Section 3-B:

# Classes & Objects

## Internet of Things: Theory and Applications

# Classes

- Class is s a data type whose variables are objects.
- An **object** as a variable that has member functions as well as the ability to hold data values.
- Thus, within a C++ program, the definition of a class should be a data type definition that describes what kinds of values the variables can hold and also what the member functions are.

# Classes Syntax

The syntax for defining a class in C++ includes declaring the class using the `class` keyword, defining member variables and member functions within the class, and optionally specifying access specifiers for the members. Example:

```
class MyClass {  
    // Member variables  
    int myVar;  
  
public:  
    // Member functions  
    void myFunction();  
};
```

# Member Functions

- Member functions in C++ are functions that are defined within a class and operate on objects of that class.
- They provide the behavior or actions that objects of the class can perform.
- Member functions have direct access to the member variables of the class and are called using the dot (.) operator on an object of the class.

# Member Functions

- A member function is defined the same way as any other function except that the `Class_Name` and the scope resolution operator `::` are given in the function heading.
- Syntax

```
Returned_Type  
Class_Name::Function_Name(Parameter_List)  
{  
Function_Body_Statements}
```



# Example on Member Functions

```
class Circle {  
private:  
    double radius;  
public:  
    void setRadius(double r) {  
        radius = r;}  
    double calculateArea() {  
        return 3.14 * radius * radius;}  
};  
  
int main() {  
    Circle myCircle;  
    myCircle.setRadius(5.0);  
    double area = myCircle.calculateArea();  
    cout << "Area of the circle: " << area << endl;  
    return 0;}
```

# Example on Classes & Member Functions

- The code example demonstrates a simple class called DayOfYear which represents dates with month and day values.
- The class has two member variables: month and day, both of type int.
- The class also has a member function called output() which outputs the month and day values to the screen.
- Objects of the class DayOfYear are declared and used in the main() function.
- The user is prompted to enter today's date and their birthday, and the program compares the dates to determine if it's the user's birthday.
- The output() function is called on the today and birthday objects to display their respective dates.
- If the dates match, a "Happy Birthday!" message is displayed; otherwise, a "Happy Unbirthday!" message is displayed.

# Example

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

class DayOfYear {
public:
    void output();
    int month;
    int day;
};

int main() {
    DayOfYear today, birthday;
```

# Example

```
// Input today's date
cout << "Enter today's date:\n";
cout << "Enter month as a number: ";
cin >> today.month;
cout << "Enter the day of the month: ";
cin >> today.day;
// Input birthday date
cout << "Enter your birthday:\n";
cout << "Enter month as a number: ";
cin >> birthday.month;
cout << "Enter the day of the month: ";
cin >> birthday.day;
```

# Example

```
// Output dates and check for birthday
cout << "Today's date is ";
today.output();
cout << "Your birthday is ";
birthday.output();
if (today.month == birthday.month && today.day == birthday.day)
    cout << "Happy Birthday!\n";
else
    cout << "Happy Unbirthday!\n";
return 0;
}

void DayOfYear::output() {
    cout << "month = " << month << ", day = " << day << endl;
}
```

# Output

```
Enter today's date:  
Enter month as a number: 4  
Enter the day of the month: 23  
Enter your birthday:  
Enter month as a number: 4  
Enter the day of the month: 23  
Today's date is month = 4, day = 23  
Your birthday is month = 4, day = 23  
Happy Birthday!  
  
Process returned 0 (0x0)   execution time : 17.401 s  
Press any key to continue.
```

# Recall: Private and Public Members in Classes

- Classes allow you to define your own types that behave like the predefined types.
- You can build a library of class type definitions and use them in your programs.
- Class definitions should separate the rules for using the class from the implementation details.

# Example on Classes

```
class Rectangle {  
private:  
    double length;  
    double width;  
  
public:  
    void setLength(double len);  
    void setWidth(double wid);  
    double getArea();  
};
```



# Private and Public Members in Classes

- Private members in a class are not directly accessible outside the class.
- Private member variables and functions can only be accessed and manipulated through public member functions.
- Making member variables private helps enforce encapsulation and data integrity.

# Example

```
class BankAccount {  
private:  
    string accountNumber;  
    double balance;  
public:  
    void deposit(double amount);  
    void withdraw(double amount);  
    double getBalance();  
};  
void BankAccount::deposit(double amount) {  
    balance += amount;  
}
```

# Example

```
void BankAccount::withdraw(double amount) {  
    if (amount <= balance) {  
        balance -= amount;  
    } else {  
        cout << "Insufficient funds." << endl;  
    }  
}  
  
double BankAccount::getBalance() {  
    return balance;  
}
```

# More on Public and Private Members

- Public and private labels can be used in a class definition to specify the access level of the members.
- Public members can be accessed and used outside the class.
- Private members can only be accessed and used within the class.

# Example

```
class Student {  
public:  
    void setName(string name);  
    void setAge(int age);  
  
private:  
    string name;  
    int age;  
};
```

# More on Member Functions

- When defining a member function, the class name must be included to specify which class the function belongs to.
- The scope resolution operator `::` is used to separate the class name and the member function name.
- The class name preceding the scope resolution operator serves as a type qualifier, specializing the function to a particular class.

# Example

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

class Circle {
private:
    double radius;

public:
    void setRadius(double r);
    double calculateArea();
};
```

# Example

```
void Circle::setRadius(double r) {  
    radius = r;  
}  
  
double Circle::calculateArea() {  
    return 3.14 * radius * radius;  
}  
  
int main() {  
    Circle c;  
    c.setRadius(5.0);  
    double area = c.calculateArea();  
    cout << "Area of the circle: " << area << endl;  
    return 0;  
}
```



# More on Member Functions

- In the member function definition, you can directly use the names of the class members without using the dot operator.
- The member names in the function definition are specialized to the name of the calling object when the function is invoked.
- This allows accessing and manipulating the member variables of the object within the member function.

# Example

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

class Person {
private:
    string name;
    int age;

public:
    void setName(string n);
    void setAge(int a);
    void displayInfo();
};
```

# Example

```
void Person::setName(string n) {  
    name = n;}  
void Person::setAge(int a) {  
    age = a;}  
void Person::displayInfo() {  
    cout << "Name: " << name << endl;  
    cout << "Age: " << age << endl;}  
int main() {  
    Person p;  
    p.setName("John Doe");  
    p.setAge(30);  
    p.displayInfo();  
    return 0;}
```

# Note on Member Functions

- Member functions can be defined within the class declaration itself, known as inline member functions.
- Inline member functions are automatically inline expanded by the compiler, reducing the function call overhead.
- They are useful for small and frequently used member functions.

# Example

```
class Rectangle {  
private:  
    int length;  
    int width;  
  
public:  
    void setLength(int len) { length = len; }  
    void setWidth(int wid) { width = wid; }  
    int getArea() { return length * width; }  
};
```

# Classes & Objects Declaration

- A **class** is a type whose variables are **objects**. These objects can have both member variables and member functions.
- The syntax for a class definition is as follows:

- **Syntax:**

```
class Class_Name  
{  
  public:  
  Member_Specification_1  
  Member_Specification_2  
  ...  
  Member_Specification_n  
  private:  
  Member_Specification_n+1  
  Member_Specification_n+2  
  ...  
};
```

# Classes & Objects Declaration

- Each *Member\_Specification* *i* is either a member variable declaration or a member function declaration. (Additional *public* and *private* sections are permitted.)
- **Example:**

```
class Bicycle
{
public:
    char get_color();
    int number_of_speeds();
    void set(int the_speeds, char the_color);
private:
    int speeds;
    char color;
};
```
- Once a class is defined, an object (which is just a variable of the class type) can be declared in the same way as variables of any other type. For example, the following declares two objects of type Bicycle: *Bicycle my\_bike, your\_bike;*

# A Tip

- When defining a class, the normal practice is to make all member variables private. This means that the member variables can only be accessed or changed using the member functions.
- It is perfectly legal to use the assignment operator = with objects or with structures.

The following is then perfectly legal (provided the member variables of the object tomorrow have already been given values):

```
due_date = tomorrow;
```

The previous assignment is equivalent to the following:

```
due_date.month = tomorrow.month;  
due_date.day = tomorrow.day;
```



# Accessors and Mutators

- Accessors and mutators, also known as getter and setter methods, are member functions in a class that allow you to access and modify the values of private member variables, respectively.
- **Accessors (Getters):**
  - Accessor methods provide access to the private member variables.
  - They typically return the value of a member variable, allowing you to retrieve it.
  - Accessor methods are usually named with a "get" prefix.
- **Mutators (Setters):**
  - Mutator methods modify the values of private member variables.
  - They provide a way to change the values of the member variables.
  - Mutator methods are typically named with a "set" prefix.

# Example

```
class BankAccount {  
private:  
    double balance;  
  
public:  
    double getBalance() {  
        return balance;  
    }  
  
    void setBalance(double newBalance) {  
        balance = newBalance;  
    }  
};
```

# Example

```
int main() {  
    BankAccount account;  
    account.setBalance(1000.0);  
  
    double currentBalance = account.getBalance();  
    cout << "Current balance: $" << currentBalance << endl;  
  
    account.setBalance(2000.0);  
    currentBalance = account.getBalance();  
    cout << "Updated balance: $" << currentBalance << endl;  
  
    return 0;  
}
```

# Tip: Structs vs Classes

- Structures are normally used with all member variables being public and having no member functions. However, in C++ a structure can have private member variables and both public and private member functions. Aside from some notational differences, a C++ structure can do anything a class can do. Having said this and satisfied the “truth in advertising” requirement, we advocate that you forget this technical detail about structures. If you take this technical detail seriously and use structures in the same way that you use classes, then you have two names (with different syntax rules) for the same concept. On the other hand, if you use structures as we described them, then you will have a meaningful difference between structures (as you use them) and classes, and your usage will be the same as that of most other programmers.

# Recall: Properties of Classes

- Classes have member variables and member functions.
- Members can be either public or private.
- Member functions can have input/output stream arguments.
- Overloading is possible for member functions.
- Member variables are usually labeled as private.
- Private members can only be used within the class definition.
- Member functions can be overloaded with different parameters.
- Classes can use other classes as member variable types.
- Functions can have formal parameters of class types.
- Functions can return objects of a class type.

# Class & Constructors

- A **constructor** is a member function of a class that has the same name as the class. A constructor is called automatically when an object of the class is declared. Constructors are used to initialize objects. A constructor must have the same name as the class of which it is a member.
- Constructors are special member functions in a class that are automatically called when an object of that class is declared.
- A constructor must have the same name as the class and cannot have a return type specified.
- Constructors can be overloaded, allowing objects to be initialized in different ways.
- Constructors eliminate the need for separate setter functions in many cases.

# Class & Constructors

- Some or all of the member variables in a class can (optionally) be initialized in the constructor initialization section of a constructor definition.
- The constructor initialization section goes after the parentheses that end the parameter list and before the opening brace of the function body. The initialization section consists of a colon followed by a list of some or all the member variables separated by commas. Each member variable is followed by its initializing value in parentheses.

# Example:

```
BankAccount::BankAccount(int dollars, int cents,  
double rate)  
: balance(dollars + 0.01*cents), interest_rate(rate)  
{  
if ((dollars < 0) || (cents < 0) || (rate < 0))  
{  
cout <<  
"Illegal values for money or interest rate.\n";  
exit(1);  
}  
}
```

Notice that the initializing values can be given in terms of the constructor parameters.



# Calling Constructor

- A constructor is called automatically when an object is declared, but you must give the arguments for the constructor when you declare the object.
- A constructor can also be called explicitly in order to create a new object for a class variable.
- **Syntax (for an object declaration when you have constructors)**  
*Class\_Name Object\_Name(Arguments\_for\_Constructor);*
- **Example:**  
*BankAccount account1(100, 2.3);*

# Calling Constructor

- **Syntax (for an explicit constructor call):**

*Object = Constructor\_Name(Arguments\_For\_Constructor);*

- **Example**

*account1 = BankAccount(200, 3.5);*

- A constructor must have the same name as the class of which it is a member.
- Thus, in the syntax descriptions above, *Class\_Name* and *Constructor\_Name* are the same identifier.
- Initializers can also be specified if the object is created as a dynamic variable. *BankAccount \*myAcct; myAcct = new BankAccount(300, 4.2);*

# Calling Constructor

- A constructor is called automatically whenever you declare an object of the class type, but it can also be called again after the object has been declared.
- This allows you to conveniently set all the members of an object.

# Calling Constructor

- C++ does not always generate a default constructor for the classes you define.
- If you give no constructor, the compiler will generate a default constructor that does nothing. This constructor will be called if class objects are declared.
- On the other hand, if you give at least one constructor definition for a class, then the C++ compiler will generate no other constructors. every time you declare an object of that type, C++ will look for an appropriate constructor definition to use.

# Constructor with No Arguments

- When you declare an object and want the constructor with zero arguments to be called, you do not include any parentheses. For example, to declare an object and pass two arguments to the constructor, you might do the following:

```
BankAccount account1(100, 2.3);
```

- However, if you want the constructor with zero arguments to be used, declare the object as follows:

```
BankAccount account1;
```

- You do *not* declare the object as follows:

```
BankAccount account1(); //INCORRECT DECLARATION
```

*(The problem is that this syntax declares a function named account1 that returns a BankAccount object and has no parameters.)*

# Abstract Data Types

- An abstract data type (ADT) is a technique used to define a class so that it separates the specification of how the type is used from its implementation details.
- To create an ADT, follow these rules:
  - Make all member variables private members of the class.
  - Provide public member functions as the basic operations that the programmer needs, fully specifying how to use each function.
  - Use private member functions for helper functions that support the implementation.
- The interface of an ADT consists of the public member functions along with their usage documentation. It tells programmers how to interact with the ADT in their programs.

# Abstract Data Types

- The implementation of an ADT includes the private members, as well as the definitions of both public and private member functions.
- Programmers can use an ADT without knowing its implementation details. They only need to understand the interface and its proper usage.
- The interface and implementation of an ADT can be separated into different files, allowing programmers to use the ADT without seeing its implementation details.
- ADTs facilitate collaboration by enabling different programmers to work on different aspects: one designing and writing the ADT, and others using it.
- By using ADTs, even solo programmers can divide a larger task into smaller ones, making program design and debugging easier.

# Example

```
// BankAccount.h (interface)
class BankAccount {
public:
    BankAccount(int dollars, int cents, double rate);
    void set(int dollars, int cents, double rate);
    void update();
    double get_balance();
    void output(ostream& outs);

private:
    double balance;
    double interest_rate;
    double fraction(double percent);
};
```



# Example

```
// BankAccount.cpp (implementation)
BankAccount::BankAccount(int dollars, int cents, double rate) {
    // Constructor implementation
}

void BankAccount::set(int dollars, int cents, double rate) {
    // set() implementation
}

void BankAccount::update() {
    // update() implementation
}
```

# Example

```
double BankAccount::get_balance() {  
    // get_balance() implementation  
}
```

```
void BankAccount::output(ostream& outs) {  
    // output() implementation  
}
```

# Example

```
// main.cpp (usage)
int main() {
    BankAccount account(100, 50, 2.0);
    account.update();
    double balance = account.get_balance();
    account.output(cout);
    return 0;
}
```



# Section 3-C: Introduction to Inheritance

**Internet of Things: Theory and Applications**

# About Inheritance

- one of the most powerful features of C++ is the use of *derived classes*. The word *inheritance* is just another name for the topic of derived classes. When we say that one class was derived from another class, we mean that the derived class was obtained from the other class by adding features.

# About Inheritance

- Inheritance allows you to define a general class and then later define more specialized classes that add some new details to the existing general class.
- This saves work because the more specialized, or derived, class inherits all the properties of the general class and you, the programmer, need only program the new features.

# Derived Class

- In C++, some class **A** can be a derived class of some other class **B**, which in turn can be a derived class of some other class **C**, and so on.
- Derived classes are often discussed using the metaphor of inheritance and family relationships. If class B is a derived class of class A, then class B is called a **child** of class A and class A is called a **parent** of class B. The parent class is also referred to as the **base** class. The derived class is said to **inherit** the member functions of its parent class.

# Practice Problem

- Problem: Implement a Bank Account class that allows users to perform deposits and withdrawals. The class should handle overdrafts by imposing a penalty fee and display appropriate messages to the user.



# Solution

```
#include <iostream>
```

```
class BankAccount {
```

```
private:
```

```
    std::string accountNumber;
```

```
    std::string accountHolderName;
```

```
    double balance;
```

```
public:
```

```
    BankAccount(const std::string& number, const std::string& holderName, double initialBalance)
```

```
        : accountNumber(number), accountHolderName(holderName), balance(initialBalance) {}
```

```
    void deposit(double amount) {
```

```
        balance += amount;
```

```
        std::cout << "Deposit of $" << amount << " successful.\n";
```

```
    }
```

# Solution

```
void withdraw(double amount) {  
    double withdrawalAmount = amount;  
    if (amount > balance) {  
        std::cout << "Warning: Insufficient funds. Overdraft penalty applied.\n";  
        withdrawalAmount = balance;  
        balance = 0.0;  
    } else {  
        balance -= amount;  
    }  
    std::cout << "Withdrawal of $" << withdrawalAmount << " successful.\n";  
}
```

# Solution

```
void displayBalance() const {
    std::cout << "Account Holder: " << accountHolderName << std::endl;
    std::cout << "Account Number: " << accountNumber << std::endl;
    std::cout << "Current Balance: $" << balance << std::endl;
}

};

int main() {
    BankAccount account("1234567890", "John Doe", 1000.0);

    account.displayBalance();
    account.deposit(500.0);
    account.withdraw(200.0);
    account.withdraw(1500.0);
    account.displayBalance();

    return 0;
}
```



# Section 4: Assignment

**Internet of Things: Theory and Applications**

# Assignment 2 – Module 1 – Part B – P1

Write a C++ program that initializes an integer array numbers with the following values: 5, 10, 15, 20, 25. Then, using a loop, calculate and display the sum of all the numbers in the array.

# Assignment 2 – Module 1 – Part B – P2

- Write a C++ program that accepts an array of integers from the user and then calculates and displays the average of all the numbers using a function. The function should take the array and its size as parameters and return the average.

# Assignment 2 – Module 1 – Part B – P3

- You are given an array of integers, `arr`, of size `n`. Write a C++ program to find the maximum product of any three integers from the array. The program should output the three integers and their product.

# Assignment 2 – Module 1 – Part B – P4

Write a program that takes a string as input and counts the number of vowels (a, e, i, o, u) in the string. Use pointers to iterate through the characters of the string and a loop to iterate over all characters. Return the count of vowels.



# Assignment 2 – Module 1 – Part B – P5

Write a program that calculates the sum of elements in an integer array. Use a loop to iterate over the array and accumulate the sum.

# Assignment 2 – Module 1 – Part B – P6

Write a program that calculates the factorial of a given number using a loop. The factorial of a number  $n$  is the product of all positive integers from 1 to  $n$ .

# Assignment 2 – Module 1 – Part B – P7

- Create a Library class that represents a library. The library should have books with titles, authors, and unique IDs. Implement member functions to add books, remove books by ID, and display the details of all books in the library.

# Assignment 2 – Module 1 – Part B – P8

- Create a BankAccount class that represents a bank account. The account should have an account number, balance, and owner's name. Implement member functions to deposit money, withdraw money, and display the account details.

# Assignment 2 – Module 1 – Part B – P8

- Create a shape hierarchy with a base class called Shape. The Shape class should have a pure virtual function called area() that returns the area of the shape. Derive three classes from Shape: Circle, Rectangle, and Triangle. Implement the area() function for each derived class to calculate and return the area of the respective shape.

# Assignment 2 – Module 1 – Part B – Bonus

- You have been assigned a task to implement a complex sorting algorithm called "Merge Sort" using arrays, arrays as function parameters, arrays in loops, and pointers. Write a program that performs the following steps:
- 1) Prompt the user to enter the size of the array.
- 2) Dynamically allocate memory for the array based on the user's input.
- 3) Fill the array with random integer values between 1 and 100.
- 4) Display the original array.
- 5) Implement the Merge Sort algorithm to sort the array in ascending order.
- 6) Display the sorted array.
- 7) Find the median value of the sorted array and display it.
- 8) Deallocate the memory for the array.