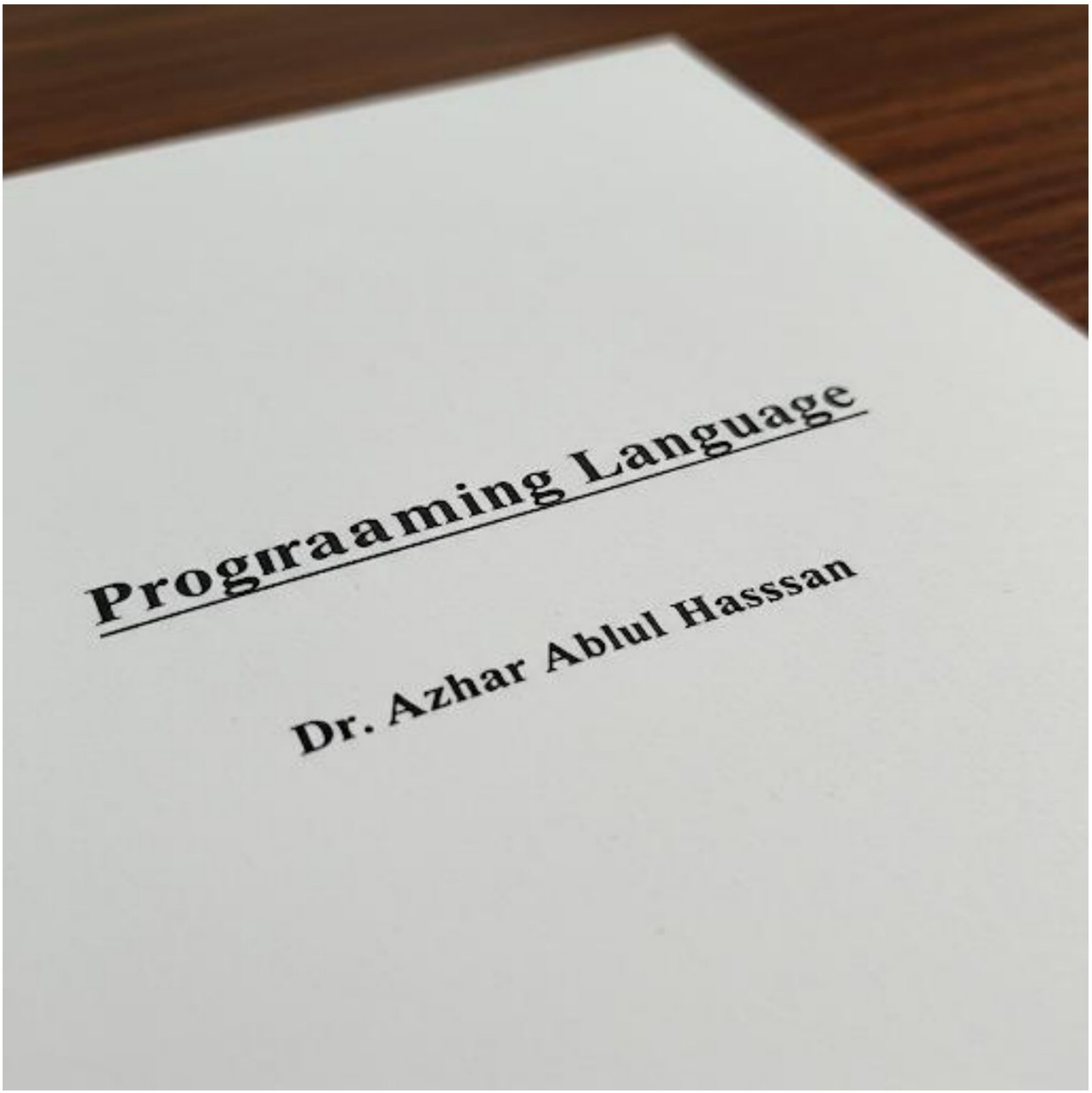

Progiraaming Language
Dr. Azhar Ablul Hasssan

Programming is the process of writing instructions for a computer in a certain order to solve a problem.

Special Characters: In C++ , all characters other than listed treated as special characters for example:

+	-	*	/	^
(	[	{	}	]
)	<	=	>	, (Comma)
" (Double Conations)	. (Dot)	: (Colon)	; (Semicolon)	␣ (Blank Space)

Reserved words cannot be used as variable names or constant. The following words are reserved for use as keywords:

Some of C++ Language Reserved Words:				
break	case	char	cin	cout
delete	double	else	enum	false
float	for	goto	if	int
long	main	private	public	short
sizeof	switch	true	union	void

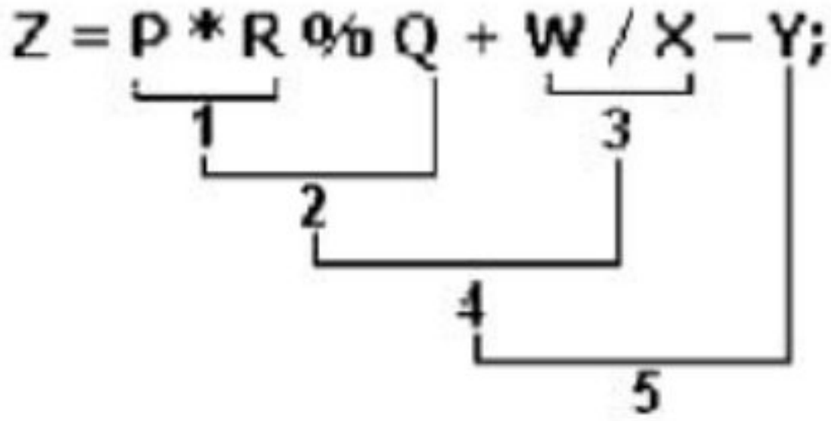
Example 2:

State the order of evaluation for the following expression:

$Z = P * R \% Q + W / X - Y;$

Solution:

1. *
2. %
3. /
4. +
5. -



The "math.h" library contains the common mathematical function used in the scientific equations.

Common function from math.h library:	
Mathematical Expression	**C++ Expression**
e^n	Exp(x)
Log(x)	Log10(x)
Ln(x)	Log(x)
Sin(x)	Sin(x)
x^n	Pow(x,n)
$\sqrt{x}$	Sqrt(x)

Example:

Write the following equation as a C++ expression and state the order of evaluation of the binary operators:

$$f = \sqrt{\frac{\sin(x) - x^5}{\ln(x) + \frac{x}{4}}}$$

Solution:

f = sqrt ((sin(x) – pow(x,5)) / (log(x) + x/4))

Using For Statement	Using While Statement	Using Do/While Statement

Q1: Find the summation of the numbers between 1 and 100.

```
for( i=1 ; i<=100 ; i++ )
   s = s + i;
```

```
i = 1;
while ( i <= 100)
{
   s = s + i;
   i++;
}
```

```
i = 1;
do
{
   s = s + i;
   i++;
}
while ( i <= 100 );
```

Q2: Find the factorial of n.

```
cin >> n;
for( i=2 ; i<=n ; i++ )
   f = f * i;
```

```
cin >> n;
i = 2;
while ( i <= n)
{
   f = f * i;
   i++;
}
```

```
cin >> n;
i = 2;
do
{
   f = f * i;
   i++;
}
while ( i <= n);
```

Q3: To find the result of the following: $\sum_{i=1}^{20} a_i^2$.

```
for( i=1 ; i<=20 ; i++ )
   s = s + (i *i);
```

```
i = 1;
while ( i <= 20)
{
   s = s + (i *i);
   i++;
}
```

```
i = 1;
do
{
   s = s + (i *i);
   i++;
}
while ( i <= 20);
```

Q4: Read 10 numbers, and find the sum of the positive numbers only.

```
for( i=1 ; i<=10 ; i++ )
{
   cin >> x;
   if ( x>0 ) s = s + x;
}
```

```
i = 1;
while ( i <= 10)
{
   cin >> x;
   if ( x>0 ) s = s + x;
   i++;
}
```

```
i = 1;
do
{
   cin >> x;
   if ( x>0 ) s = s + x;
   i++;
}
while ( i <= 10);
```

Q5: Represent the following series: 1, 2, 4, 8, 16, 32, 64.

```
for( i=1 ; i<65 ; i*=2 )
   cout << i;
```

```
i = 1;
while ( i<65)
{
   cout << i;
   i*=2;
}
```

```
i = 1;
do
{
   cout << i;
   i*=2;
}
while ( i<65);
```

Q6: Find the sum of the following s = 1 + 3 + 5 + 7 + ... + 99.

```
for( i=1 ; i<=99 ; i+=2 )
   s = s + i;
```

```
i = 1;
while ( i<=99)
{
   s = s + i;
   i+=2;
}
```

```
i = 1;
do
{
   s = s + i;
   i+=2;
}
while ( i<=99);
```

Q7: Find the sum and average of the 8 degrees of the student.

```
for( i=1 ; i<=8 ; i++ )
{
   cin >> d;
   s = s + d;
}
av = s / 8;
```

```
i = 1;
while ( i<=8)
{
   cin >> d;
   s = s + d;
   i++;
}
av = s / 8;
```

```
i = 1;
do
{
   cin >> d;
   s = s + d;
   i++;
}
while ( i<=8);
av = s / 8;
```

Structures are typically used to group several data items together to form a single entity. It is a collection of variables used to group variables into a single record. Thus a structure (the keyword struct is used in C++) is used. Keyword struct is a data-type, like the following C++ data-types (int, float, char, etc ...). This is unlike the array, which all the variables must be the same type. The data items in a structure are called the members of the structure.

```
#include <iostream.h>

struct  data
{
    char *name;
    int age;
};

void  main()
{
   struct data student;
   student . name="ahmed";
   student . age=20;
      :
}
```

Structured/Procedural Programming	Object Oriented Programming
Code is divided into modules or functions	Code is made up of classes and objects
Town-down approach	Bottom-up approach
Difficult to modify / manage	Easy to modify / manage
Main function calls other functions	Objects communicate by passing messages
Data is not secured	Data is secure
Less reusability of code	More reusability of code
Less flexibility and abstraction	More flexibility and abstraction

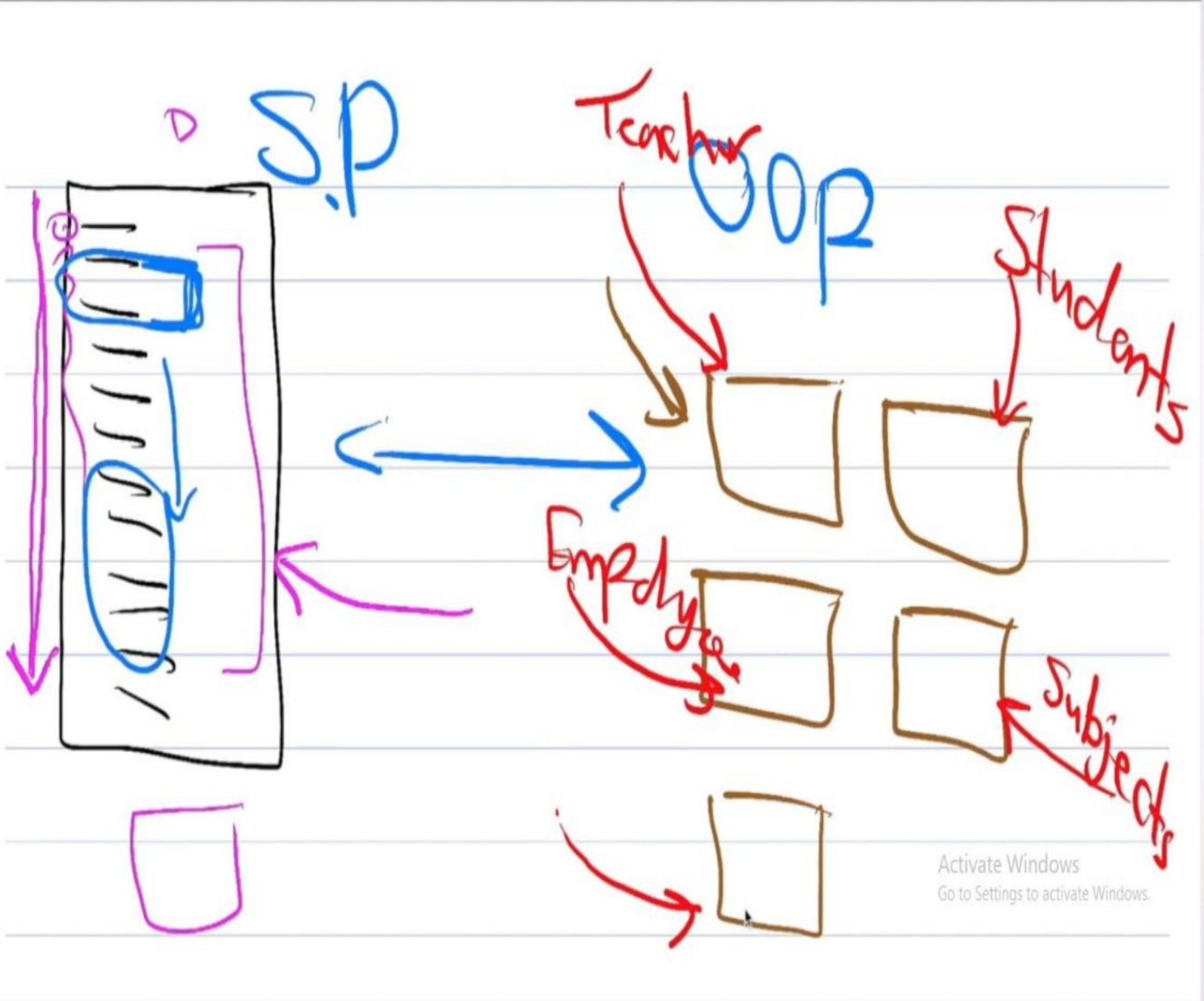
OOP

Object oriented programing

Object Oriented Programming (OOP)

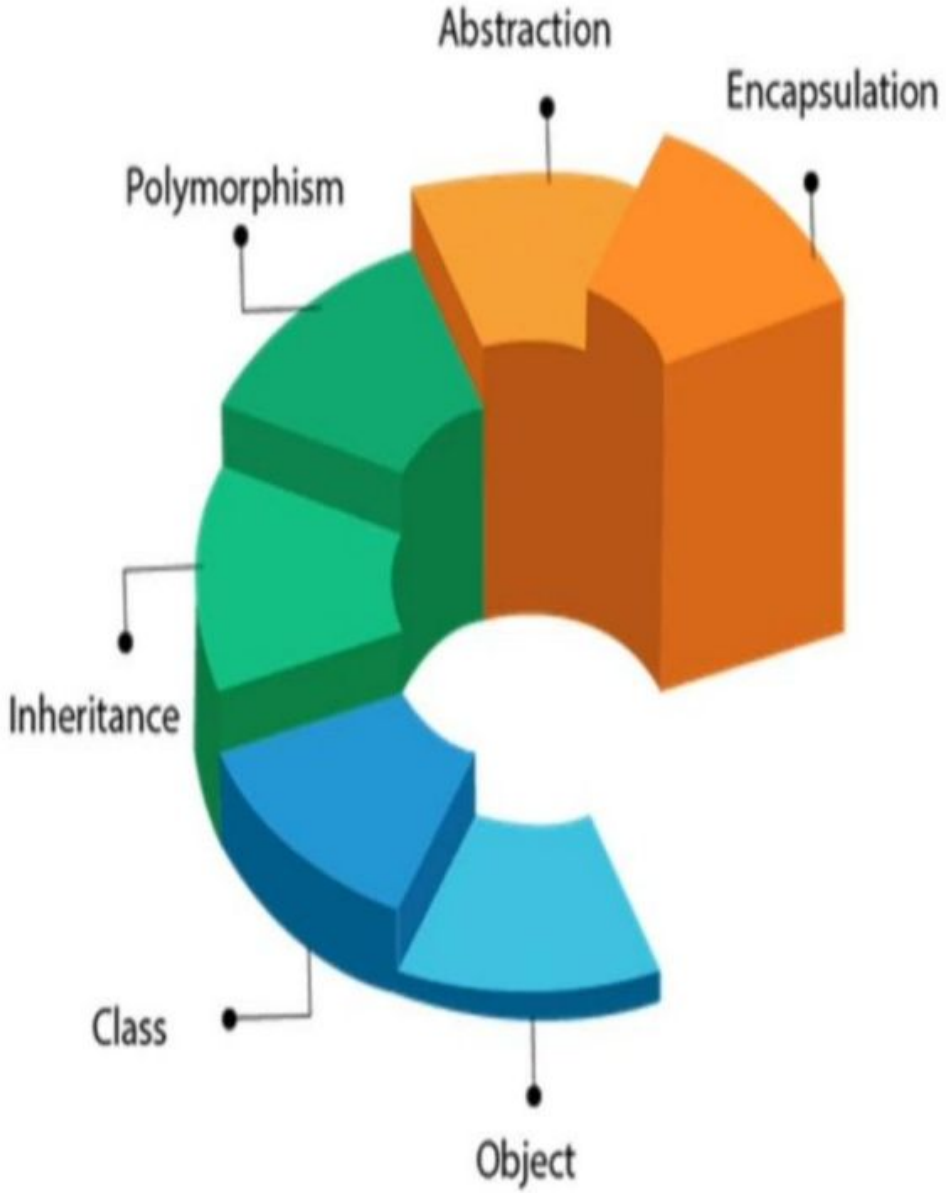
Object-oriented programming – As the name suggests uses objects in programming. Object-oriented programming aims to implement real-world entities like inheritance, hiding, polymorphism, etc in programming.

hiding

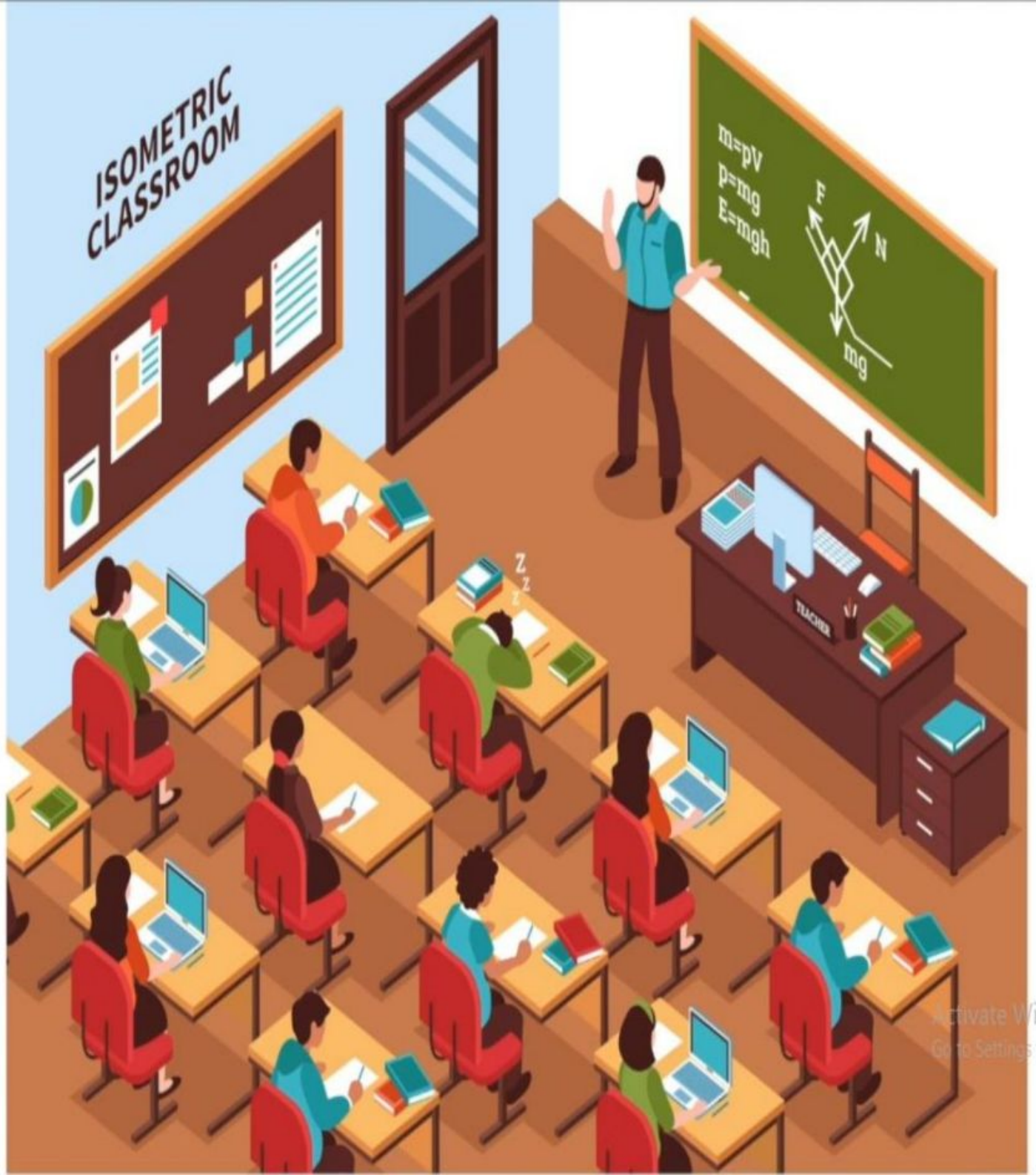


Structured Programming	Object Oriented Programming
1. The program is represented as a logical structure.	The program is written as a collection of objects which communicate with each other.
2. The flow of execution of the programming is dependent on the structure of the program.	The basic entity is object. Each computation is performed using objects only.
3. Code is given more importance.	Data is given more importance.
4. Can handle up to moderately complex programs.	Can handle very complex programs.
5. Less code reusability.	More code reusability.
6. Less data security.	More data security.
7. Abstraction is less.	Abstraction is more.
8. Flexibility is less.	Flexibility is more.

OOPs (Object-Oriented Programming System)



Anything
In
The
World
Is
An
OBJECT.



Class VS Object

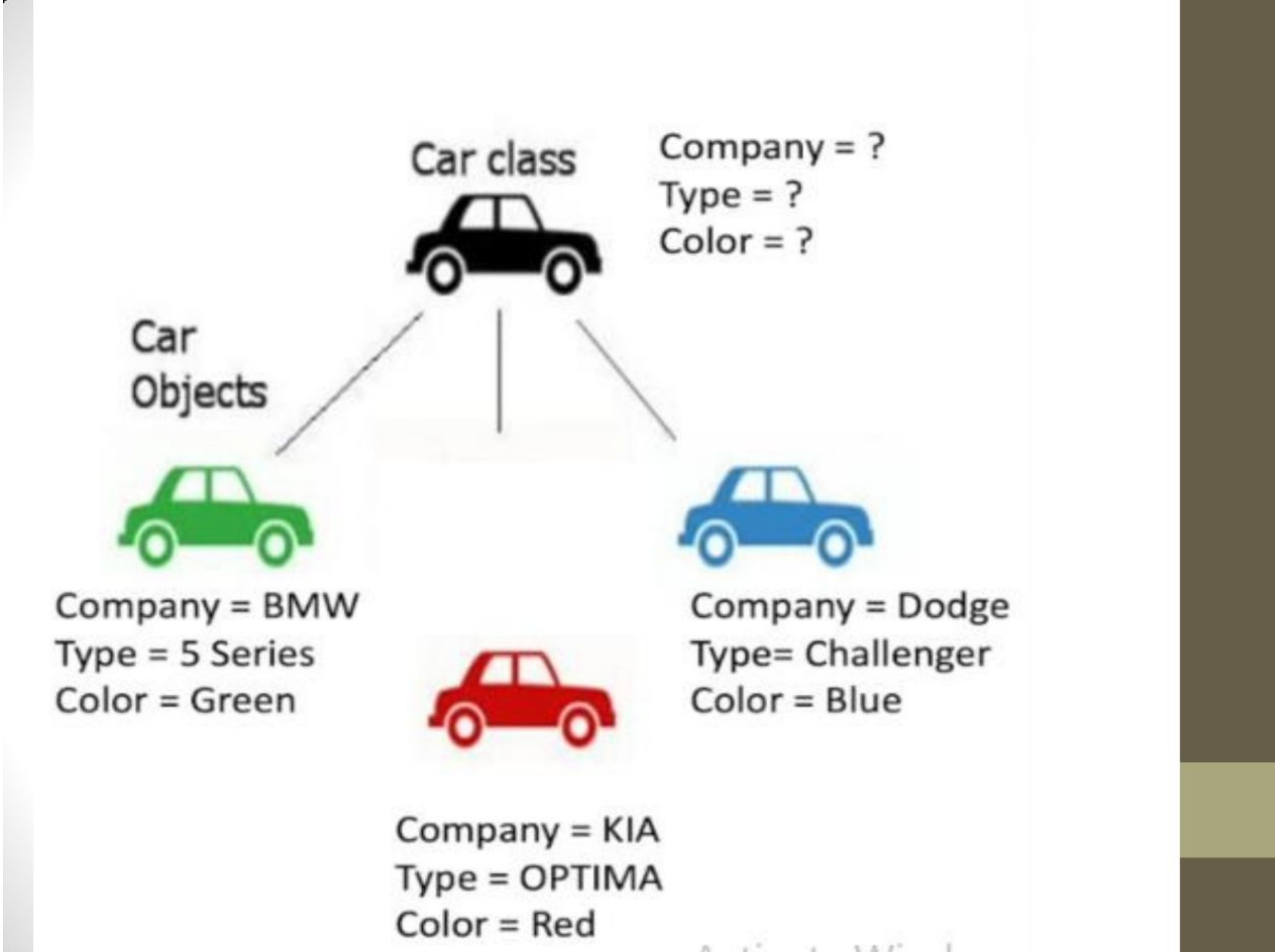
- **Object:** It is a basic unit of Object-Oriented Programming and represents the real-life entities. An object is a thing.

- **Class:** A category of *objects*. The class defines all the common properties of the different objects that belong to it.

- An **Object** is an instance of a Class.

Student (Class)

Class	Object
1) Class is a collection of similar objects	1) Object is an instance of a class
2) Class is conceptual (is a template)	2) Object is real
3) No memory is allocated for a class.	3) Each object has its own memory
4) Class can exist without any objects	4) Objects can't exist without a class
5) Class does not have any values associated with the fields	5) Every object has its own values associated with the fields


Car class
Company = ?
Type = ?
Color = ?
Car
Objects
Company = BMW
Type = 5 Series
Color = Green
Company = Dodge
Type= Challenger
Color = Blue
Company = KIA
Type = OPTIMA
Color = Red

What is a Template?

Cake Pan (Class)

Oop

Lecture 2

Car (Class)

Company = ?
Type = ?
Model = ?
Color = ?
...

obj1

Company = Mercedes-Benz
Type = CLA-Class
Model = 2018
Color = White

obj2

Company = Land Rover
Type = Range Rover
Model = 2017
Color = Red

obj3

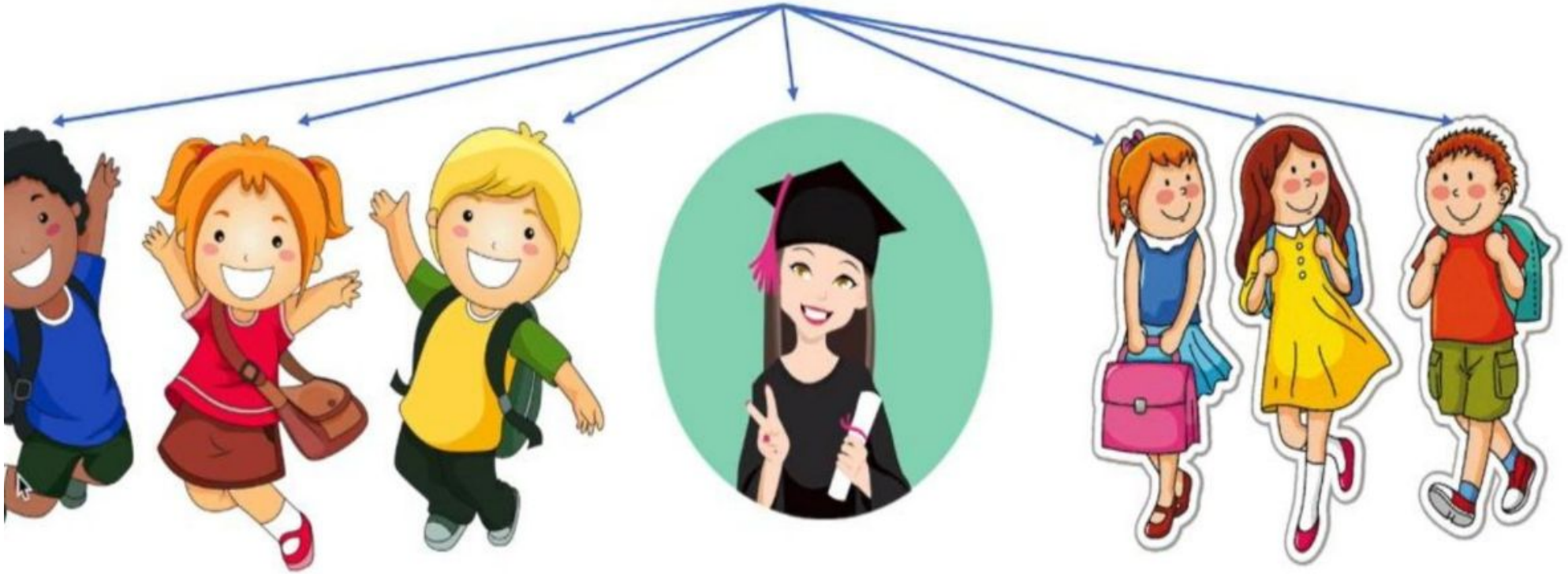
Company = BMW
Type = X5
Model = 2017
Color = White

Student (Class)

ID = ?

Name = ?

...



100 Ali

101 Sara

102 Ahmed

400 Ola

200 Noor

201 Ban

202 Samir

Book (Class)

ID = ?

Title = ?

Author = ?

...

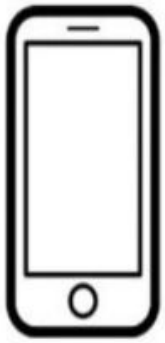
ID = XXX

Title = XXX

Author = XXX

...

Phone (Class)



Company = ?

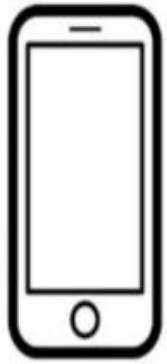
Name = ?

Screen = ?

Price = ?

...

Phone (Class)



Company = ?

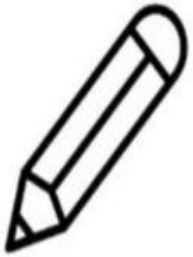
Name = ?

Screen = ?

Price = ?

...

Pen (Class)



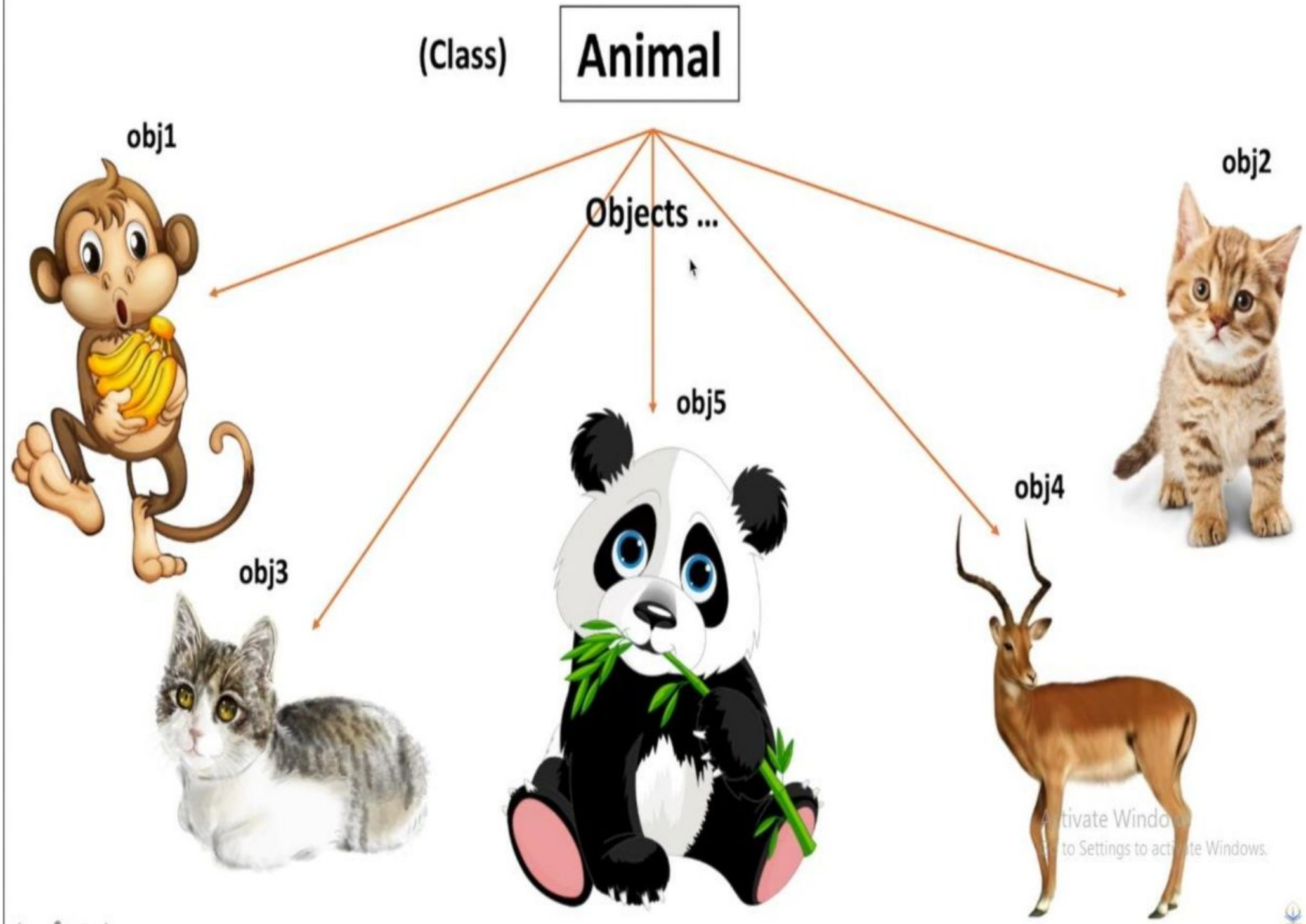
Type = ?

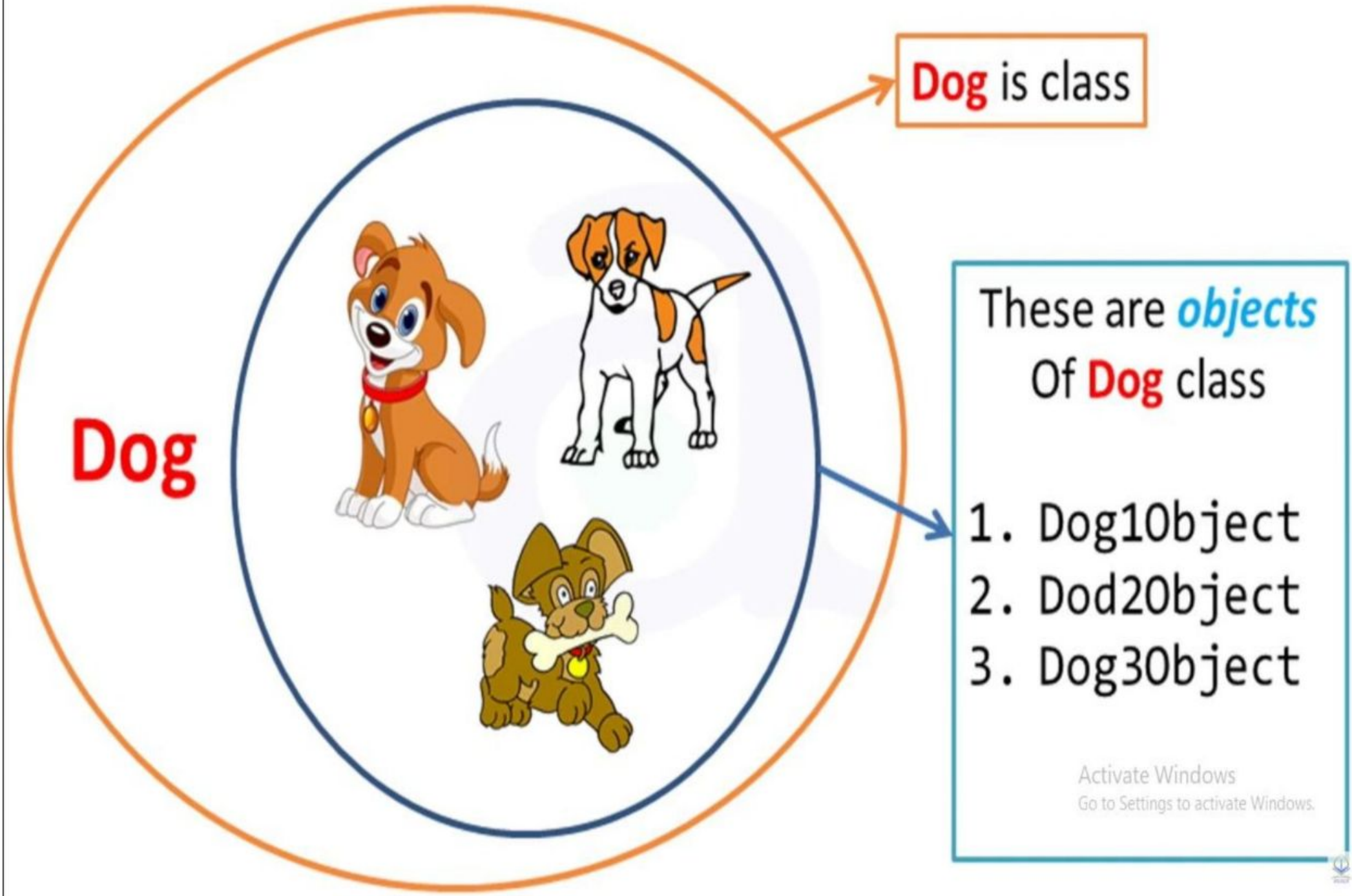
Color = ?

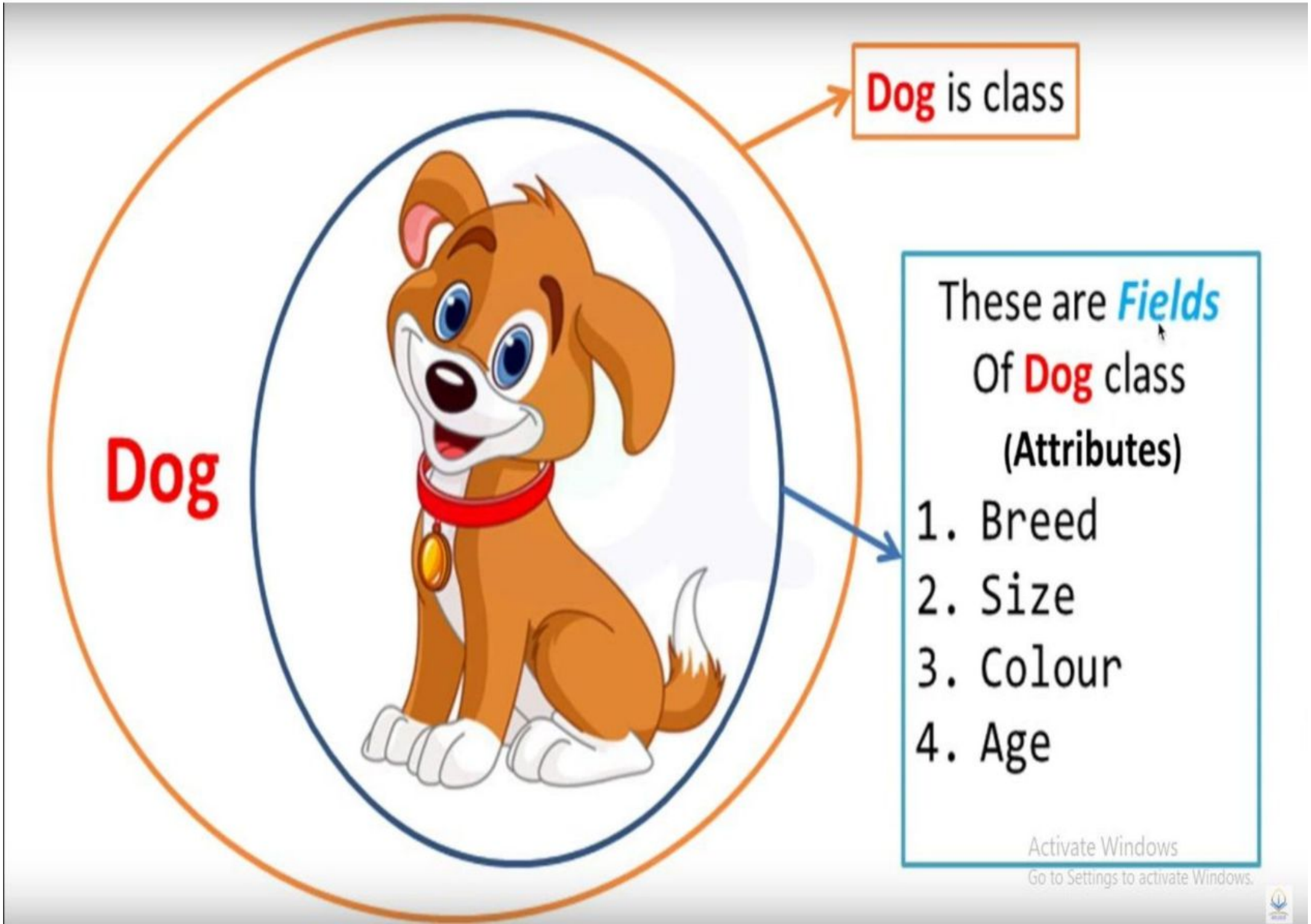
Price = ?

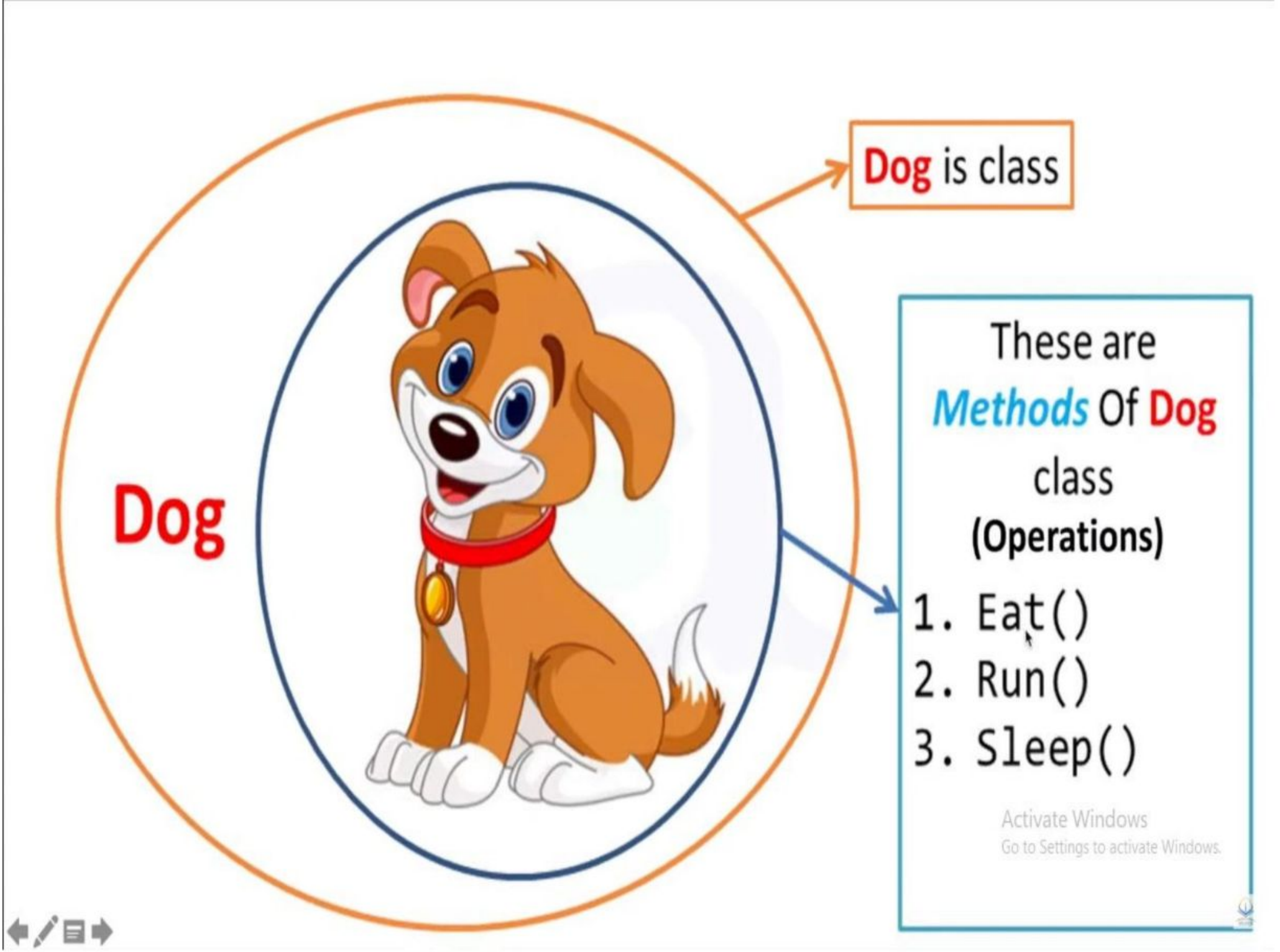
...

(Class) **Animal**


(Class)
Animal
Objects ...
obj1
obj2
obj3
obj4
obj5


Dog
Dog is class
These are objects
Of Dog class
1. Dog1Object
2. Dod2Object
3. Dog3Object
Activate Windows
Go to Settings to activate Windows.


Dog is class
Dog
These are Fields
Of Dog class
(Attributes)
1. Breed
2. Size
3. Colour
4. Age
Activate Windows
Go to Settings to activate Windows.


Dog
Dog is class
These are
Methods Of Dog
class
(Operations)
1. Eat()
2. Run()
3. Sleep()
Activate Windows
Go to Settings to activate Windows.

Dog is class

These are ***Methods*** Of **Dog** class **(Operations)**

1. Eat()
2. Run()
3. Sleep()

Breed: Bulldog
Size: large
Colour: light gray
Age: 5 years

Dog1Object

Breed: Beagle
Size: large
Colour: orange
Age: 6 years

Dog2Object

Breed: German Shepherd
Size: large
Colour: white & orange
Age: 6 years

Dog3Object

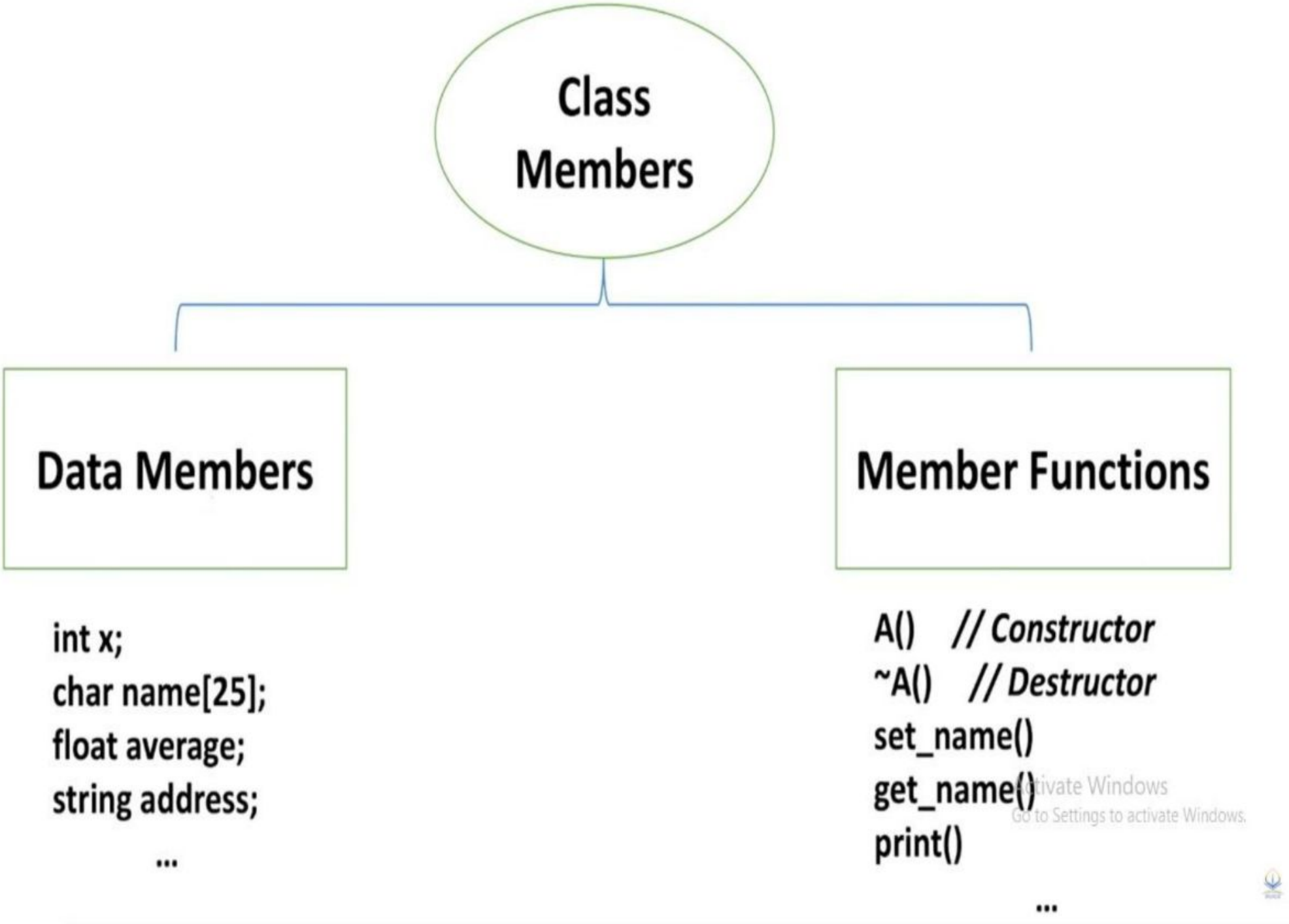
Dog

Fields

Breed
Size
Colour
Age

Methods

Eat()
Run()
Sleep()
Name()



Class Diagram

- In software engineering, a class diagram in the Unified Modeling Language (UML): is a type of static structure diagrams that describes the structure of a system by showing the system's **classes**, their **attributes**, **operations** (or methods), and the relationships among objects.

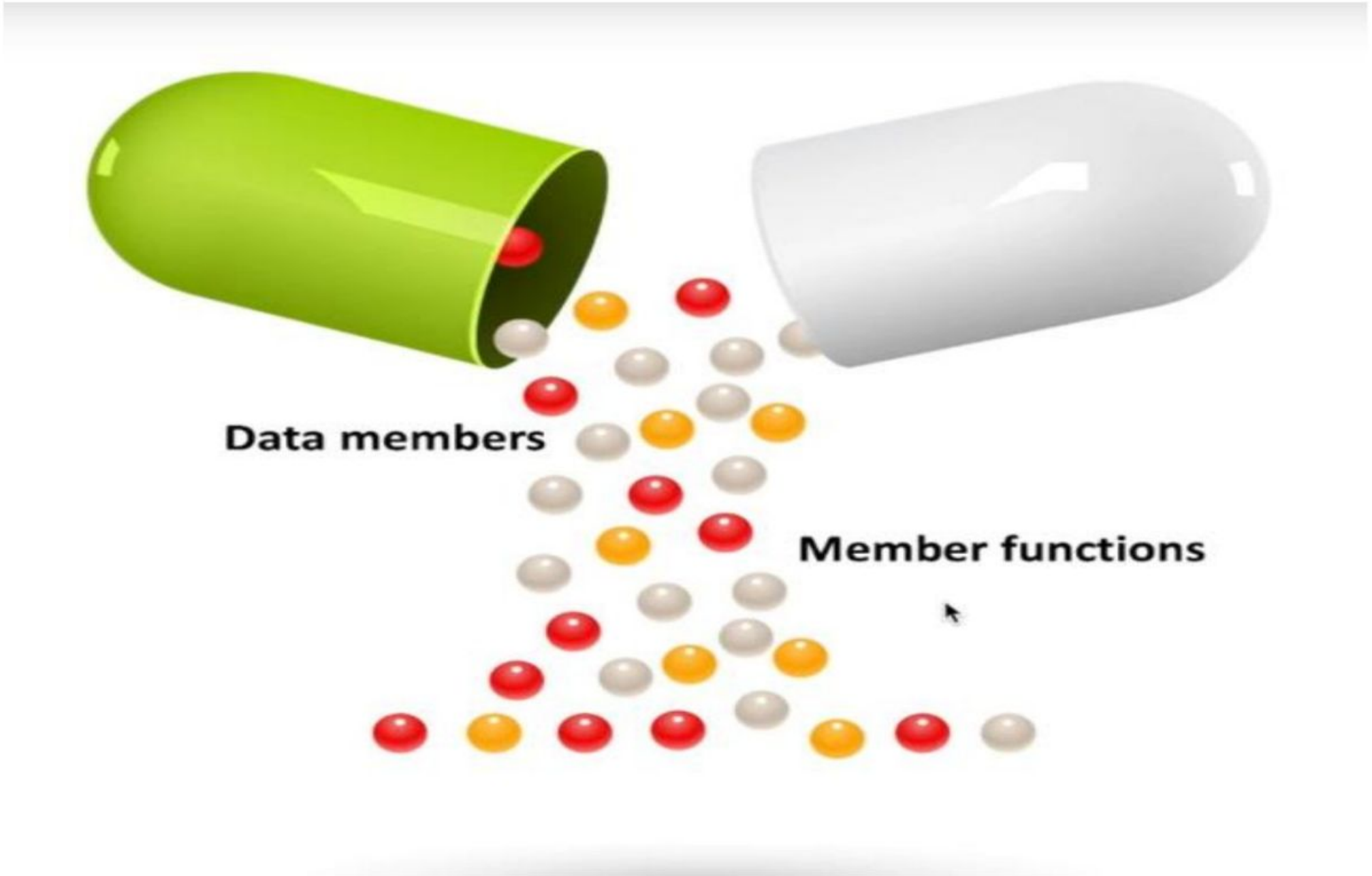
Class Notation

- **Class Name**
  - The name of the class appears in the first partition.
- **Class Attributes**
  - Attributes are shown in the second partition.
  - The attribute type is shown after the colon (:).
  - Attributes map onto member variables (data members) in code.
- **Class Operations** (Functions)
  - Operations are shown in the third partition. They are services the class provides.
  - The return type of a function is shown after the colon (:) at the end of the function signature (header).
  - The types of function's parameters are shown inside ().
  - Operations map onto class functions (member functions) in code.

The Structure of Class in Class Diagram

Encapsulation

Data encapsulation refers to the process of *binding* together data and functions or methods operating on this data into a single unit so that it is protected from outside interference and misuse.


Data members
Member functions

To Achieve Encapsulation:

Private = inside the class only.

Public = inside and outside the class

Member functions

To Achieve Encapsulation:

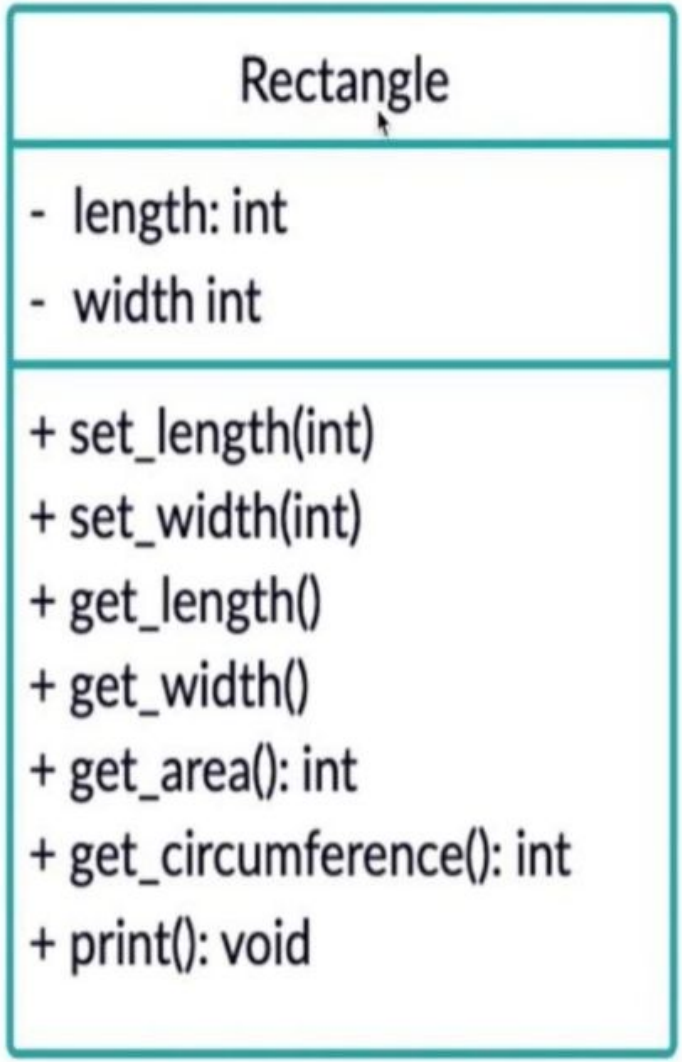
← private

← public

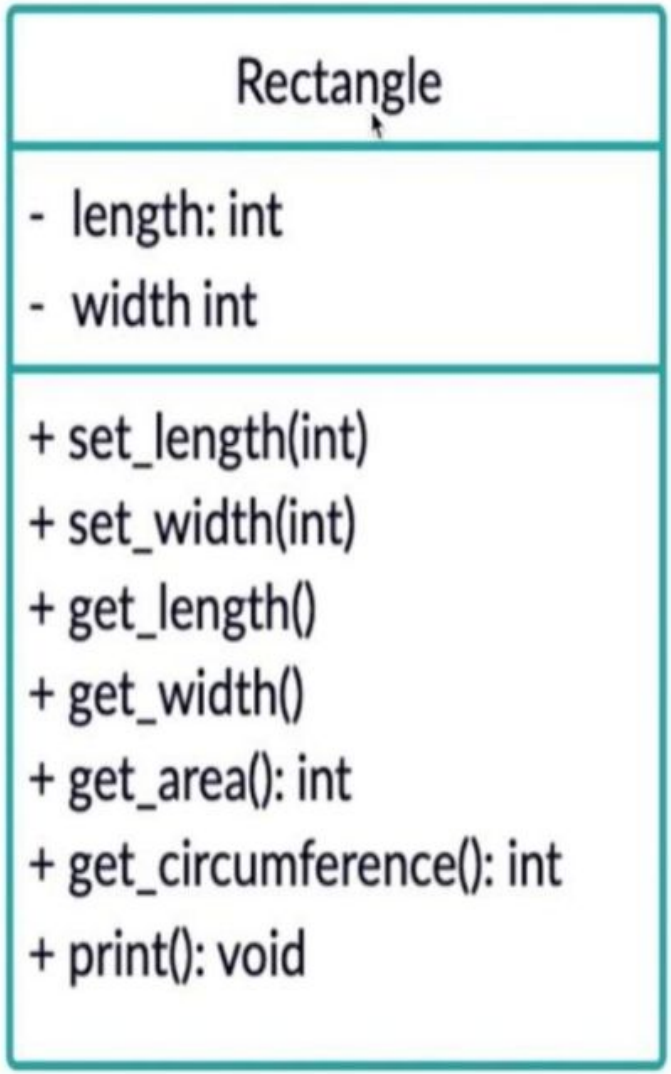
Example bank

- Mony
- Set=input
- Get =out
- -= private
- +=public

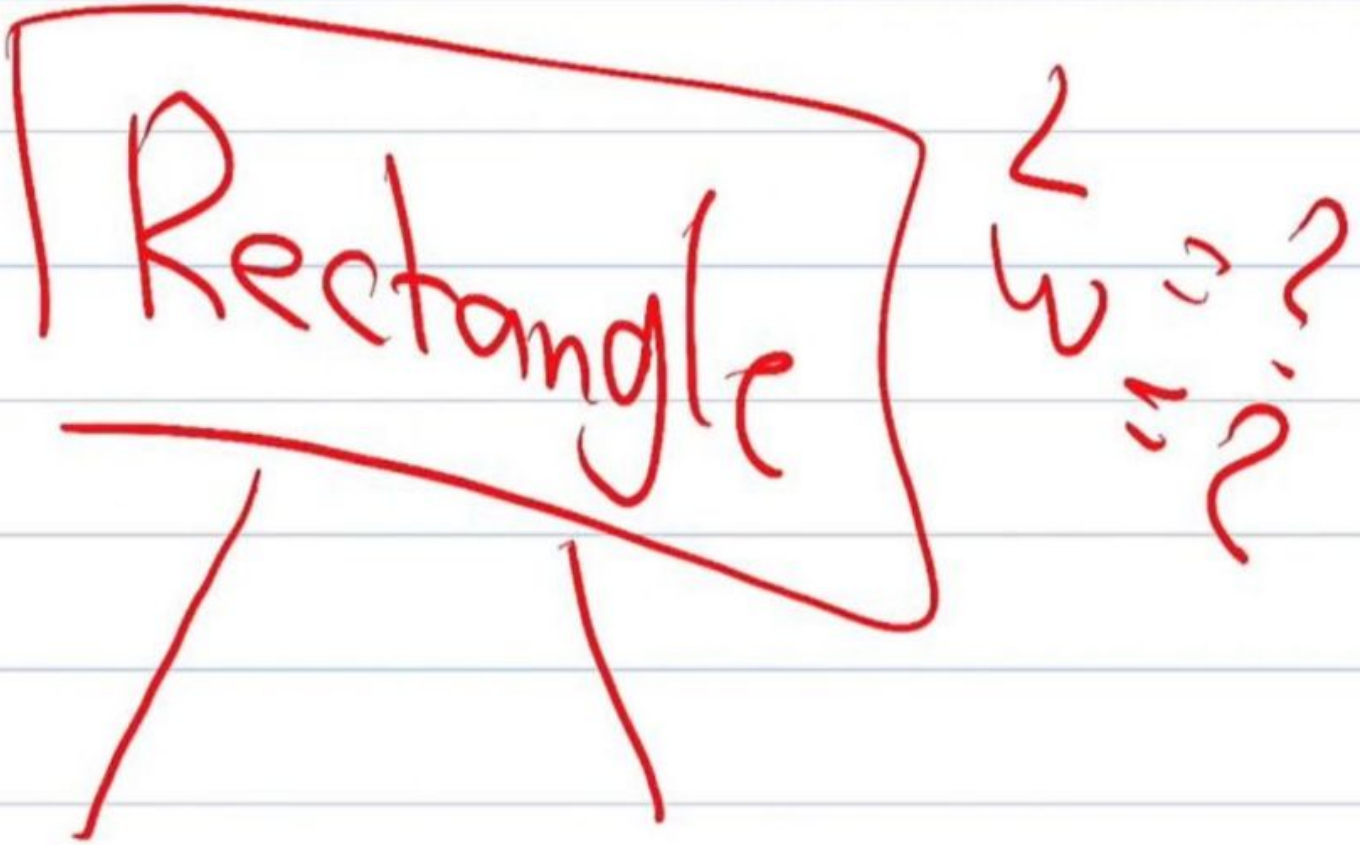
Example 1:



Example 1:



```cpp
using namespace std;

class Rectangle
{
private:
     int length;
     int width;

public:
     void set_length(int l)
     {
          length = l;
     }
     void set_width(int w)
     {
          width = w;
     }
     int get_length()
     {
          return length;
     }
     int get_width()
     {
          return width;
     }
     int get_area()
     {
          return length * width;
     }

     }
     int get_cir()
     {
          return 2 * (length + width);
     }
     void print()
     {
          cout<<"Length = "<<length<<endl;
          cout<<"Width = "<<width<<endl;
     }
};

int main()
{
     Rectangle r1;
     r1.set_width(20);
     r1.set_width(10);

     Rectangle r2;
     r2.set_length(100);
     r2.set_width(130);
}
```

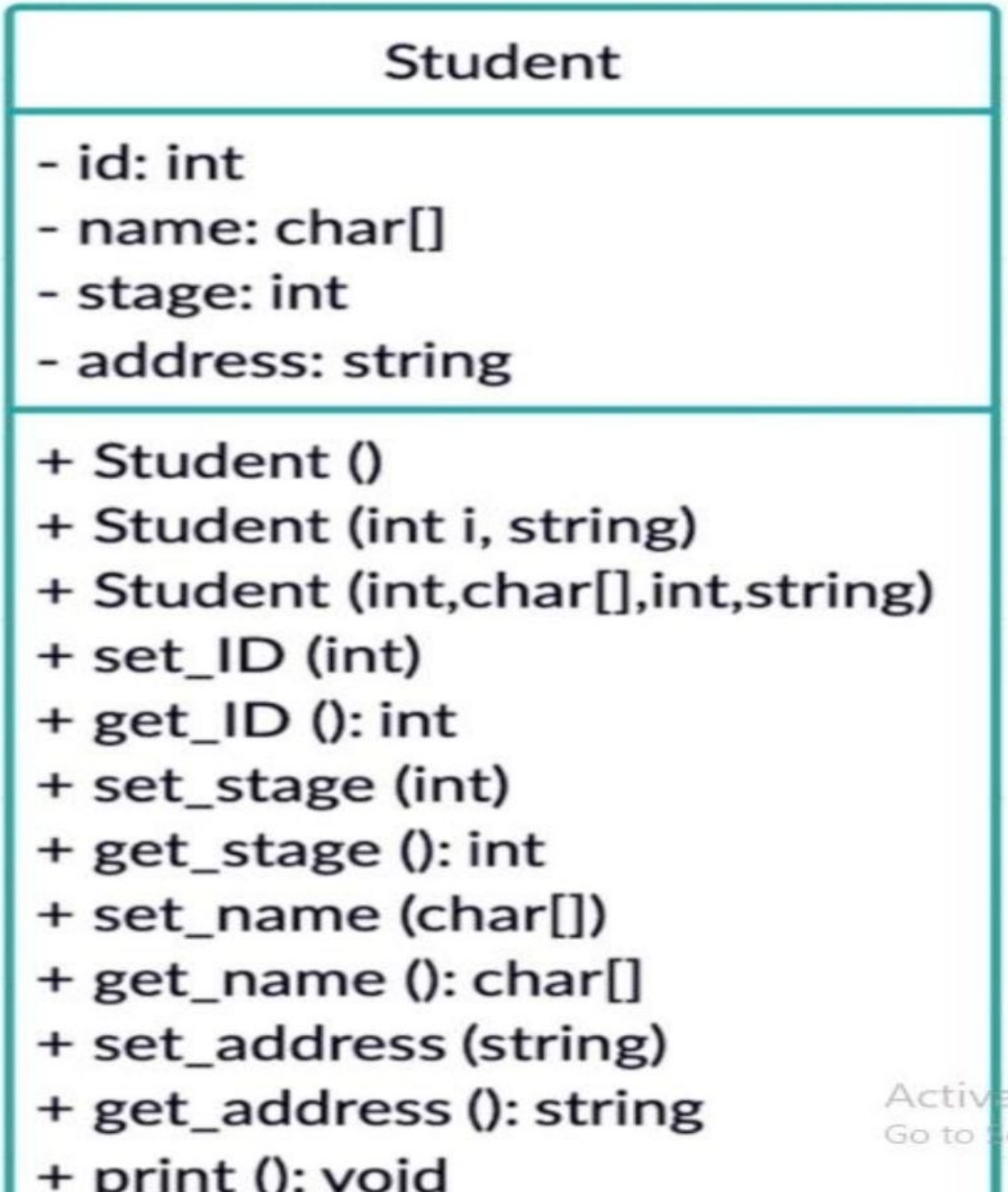


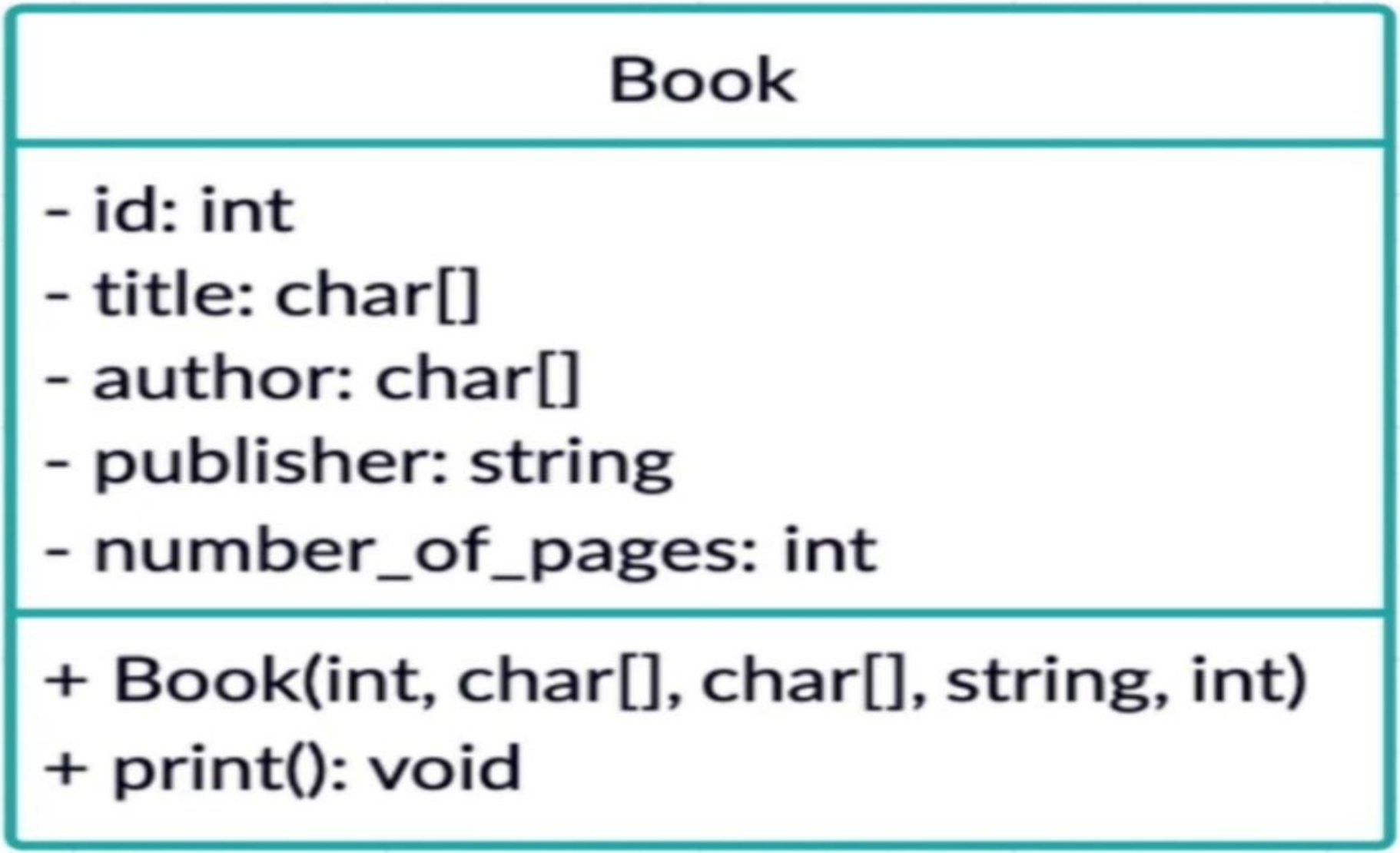
L = 20

W = 10

L = 100

W = 130




Book
- id: int
- title: char[]
- author: char[]
- publisher: string
- number_of_pages: int
+ Book(int, char[], char[], string, int)
+ print(): void

To create object:

- Class_name obj_name;
- Class _name obj_name();

Al-shatrah University
College of engnaring

كليـــةالهندسة

هندسة الحاسوب

Subject: Object Oriented Programming (OOP)

First Stage

Lecturer: Dr. Azhaar abd alhussan

Lecture (3)

Built-in Functions in C++

Built-in functions in C++ are predefined functions provided by the compiler, and they are part of standard libraries. These functions help developers perform various operations like working with numbers,

strings, and memory management. The most common categories of built-in functions are:

1. Mathematical Functions
2. String Handling Functions
3. Input/Output Functions
4. Utility Functions

We'll discuss some of these functions and demonstrate their usage in code.

1. Mathematical Functions in `<cmath>` Library

1.1 `ceil(x)` and `floor(x)`

- `ceil(x)`: Returns the smallest integer greater than or equal to `x`.

- `floor(x)`: Returns the largest integer less than or equal to `x`.

Example:

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

int main() {
   double num1 = 4.3;
   double num2 = 4.8;

   cout << "Ceil of " << num1 << " is: " << ceil(num1) << endl;   // Output: 5
   cout << "Floor of " << num2 << " is: " << floor(num2) << endl; // Output: 4
   return 0;
}
```

1.2 `round(x)`

- Description: Returns the value of `x` rounded to the nearest integer.

Example:

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

int main() {
```

```
    double num = 5.7;
    cout << "Round of " << num << " is: " << round(num) << endl;  // Output: 6
    return 0;
}
```

1.3 `sqrt(x)`

- Description: Calculates the square root of `x`.

Example:

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

int main() {
    double num = 16;
    cout << "Square root of " << num << " is: " << sqrt(num) << endl;  // Output: 4
    return 0;
}
```

1.4 `pow(x, y)`

- Description: Returns `x` raised to the power of `y` (i.e., \(x^y \)).

Example:

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

int main() {
    double base = 3;
    double exponent = 4;
    cout << base << " raised to the power of " << exponent << " is: " << pow(base, exponent) << endl;  // Output: 81
    return 0;
}
```

1.5 `abs(x)`

- Description: Returns the absolute value of `x`.

Example:

```
#include <iostream>
```

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

int main() {
   int num = -42;
   cout << "Absolute value of " << num << " is: " << abs(num) << endl;  // Output: 42
   return 0;
}
```

1.6 `sin(x)`, `cos(x)`, `tan(x)`

- Description: Compute the sine, cosine, and tangent of an angle `x` in radians.

Example:

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

int main() {
   double angle = 45.0;
   // Convert angle to radians
   double radian = angle * M_PI / 180.0;

   cout << "Sin(" << angle << ") = " << sin(radian) << endl;
   cout << "Cos(" << angle << ") = " << cos(radian) << endl;
   cout << "Tan(" << angle << ") = " << tan(radian) << endl;

   return 0;
}
```

1.7 `log(x)` and `log10(x)`

- `log(x)`: Returns the natural logarithm (base `e`) of `x`.

- `log10(x)`: Returns the base-10 logarithm of `x`.

Example:

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

int main() {
   double num = 100.0;
   cout << "Natural logarithm of " << num << " is: " << log(num) << endl;
```

```
    cout << "Base-10 logarithm of " << num << " is: " << log10(num) << endl;

    return 0;
}
```

Math Functions in C Standard Library

Function Name	Math Name	Value	Example		
`abs(x)`	absolute value	$\|x\|$	`abs(-1)`	returns	1
`fabs(x)`	absolute value	$\|x\|$	`fabs(-3.2)`	returns	3.2
`pow(x,y)`	raise to the power	x^y	`pow(2.0, 3.0)`	returns	8.0
`sqrt(x)`	square root	$x^{0.5}$	`sqrt(2.0)`	returns	1.414...
`exp(x)`	exponential	e^x	`exp(1.0)`	returns	2.718...
`log(x)`	natural logarithm	$\ln x$	`log(2.718...)`	returns	1.0
`log10(x)`	common logarithm	$\log x$	`log10(100.0)`	returns	2.0
`sin(x)`	sine	$\sin x$	`sin(3.14...)`	returns	0.0
`cos(x)`	cosine	$\cos x$	`cos(3.14...)`	returns	-1.0
`tan(x)`	tangent	$\tan x$	`tan(3.14...)`	returns	0.0
`ceil(x)`	ceiling	$\lceil x \rceil$	`ceil(2.5)`	returns	3.0
`floor(x)`	floor	$\lfloor x \rfloor$	`floor(2.5)`	returns	2.0

2. Random Number Functions in `<cstdlib>` Library

2.1 `rand()`: Generates a pseudo-random number.

Example:

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

int main() {
    cout << "Random number: " << rand() << endl;  // Output: A random number
    return 0;
}
```

2.2 `srand(seed)` : Seeds the random number generator with a value `seed`. Using different seeds ensures different sequences of random numbers.

Example:

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

int main() {
   srand(time(0)); // Seed with current time
   cout << "Random number: " << rand() << endl;  // Output: A random number
   return 0;
}
```

3. Rounding Functions in `<cmath>` Library

3.1 `trunc(x)`: Returns the integer part of `x`, removing the fractional part.

Example:

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

int main() {
   double num = 5.78;
   cout << "Truncated value of " << num << " is: " << trunc(num) << endl;  // Output: 5
   return 0;
}
```

3.2 `fmod(x, y)`: Returns the remainder of `x` divided by `y`.

Example:

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

int main() {
   double num1 = 7.0;
   double num2 = 3.0;

   cout << "Remainder of " << num1 << " / " << num2 << " is: " << fmod(num1, num2) <<
endl;  // Output: 1.0
   return 0;
}
```

4. Utility Functions:

4.1 `exit(status)`: Terminates the program immediately. The `status` indicates whether the program terminated normally (`0`) or with an error (any non-zero value).

Example:

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

int main() {
   cout << "Program will terminate now." << endl;
   exit(0);  // Terminate with a normal status
   return 0; // This line will not be executed
}
```

4.2 **`system(command)`:**Executes a system command.

Example:

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

int main() {
          cout<<"hello world1"<<endl;  // will be deleted
          cout<<"hello world2"<<endl;  // also deleted
          cout<<"hello world3"<<endl;  //deleted from secreen
   system("CLS");  // Clears the console (on Windows)
   cout << "Screen cleared." << endl;
   return 0;
}
```

Other using parameter

system("color b") ➔ change the color.

system("date")➔ deal with date.

4.3 swap(a, b): Swaps the values of a and b, in Library: <algorithm>

Example:

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

int main() {
   int x = 10, y = 20;
   cout << "Before swap: x = " << x << ", y = " << y << endl;
   swap(x, y);
   cout << "After swap: x = " << x << ", y = " << y << endl;  // Output: x = 20, y = 10
   return 0;
}
```

4.4 max(a, b) and min(a, b): Returns the maximum or minimum of (a and b). Library: <algorithm>.

Example:

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

int main() {
   int a = 15, b = 30;
   cout << "Maximum: " << max(a, b) << endl;  // Output: 30
   cout << "Minimum: " << min(a, b) << endl;  // Output: 15
   return 0;
}
```

Summary:

By using built-in functions, students can simplify their code and perform complex operations with minimal effort. The `<cmath>` library provides powerful mathematical functions like `sqrt()`, `pow()`, `ceil()`, and `floor()`, while `<cstdlib>` offers utilities for generating random numbers and interacting with the system. Understanding and utilizing these functions are crucial for solving mathematical problems efficiently in C++.

Quiz: Built-in Functions in C++

1. Which of the following headers is required to use the *sqrt* function?

a) <iostream>
b) <cmath>
c) <algorithm>
d) <stdlib.h>

2. What does the *ceil* function do?

a) Rounds a number down to the nearest integer
b) Rounds a number up to the nearest integer
c) Returns the absolute value of a number
d) Computes the square root of a number

3. What will *pow(2, 3)* return?

a) 2
b) 6
c) 8
d) 9

4. To round a number down to the nearest integer, you can use the _____ function.

5. The function _____ returns the absolute value of a number.

الفصل السادس (Section 6)

1.6- الدوال (Functions):-

ورثت اللغة C++ من اللغة C مكتبة ضخمة وغنية بدوال تقوم بتنفيذ العمليات الرياضية، التعامل مع السلاسل والأحرف، الإدخال والإخراج، اكتشاف الأخطاء والعديد من العمليات الأخرى المفيدة مما يسهل مهمة المبرمج الذي يجد في هذه الدوال معيناً كبيراً له في عملية البرمجة.يمكن للمبرمج كتابة دوال تقوم بأداء عمليات يحتاج لها في برامجه وتسمى مثل هذه الدوال انها دوال معرفة من قبل المبرمج(Programmer- defined functions).
تعرف الدالة على أنها جملة أو مجموعة جمل أو تعليمات ، ذات كيان خاص، تقوم بعملية أو مجموعة عمليات ، سواء عمليات إدخال أو إخراج أو عمليات حسابية أو منطقية ، وتحتل الدالة موقعا من البرنامج ، آي أنها جزء منه ، أو يمكن القول أن لغة C++ تتكون من مجموعة من الدوال.
كما تعرف الدالة عبارة عن مجموعة من العبارات المصممة لإنجاز مهمة . لقد أظهرت التجربة أن أفضل طريقة لتطوير وصيانة برنامج كبير هي بنائه من قطع أصغر (وحدات نمطية). تسمى الوحدات النمطية في C++ بالدوال.
الدوال مفيدة جدا لقراءة وكتابة وتصحيح وتعديل البرامج المعقدة، يمكن أيضًا دمجها بسهولة في البرنامج الرئيسي. في C ++، main ( ) نفسها هي دالة تعني أن الوظيفة الرئيسية هي استدعاء الوظائف الأخرى لأداء مهام مختلفة.
الدوال تمكن المبرمج من تقسيم البرنامج إلى وحدات modules، كل دالة في البرنامج تمثل وحدة قائمة بذاتها، ولذا نجد أن المتغيرات المعرفة في الدالة تكون متغيرات محلية (Local) ونعنى بذلك أن المتغيرات تكون معروفة فقط داخل الدالة. أغلب الدوال تمتلك لائحة من الوسائط (Parameters) والتي هي أيضاً متغيرات محلية.

المزايا الرئيسية لاستخدام الدوال هي:-

1 -تساعد الدوال المخزنة في ذاكرة الحاسب على اختصار البرنامج إذ يكتفى باستعادتها باسمها فقط لتقوم بالعمل المطلوب.
2 -تساعد البرامج المخزنة في ذاكرة الحاسب ، أو التي يكتبها المبرمج على تلافي عمليات التكرار في خطوات البرنامج التي تتطلب عملا طويلا وشاقا.
3 -تساعد الدوال الجاهزة على تسهيل عملية البرمجة نفسها.
4 -توفر مساحة من الذاكرة المطلوبة.
5 -اختصار عمليات زمن البرمجة وتنفيذ البرنامج بأسرع وقت ممكن.

وللتدليل على أهمية الدوال في برمجة C++
فمثلا لو أردنا كتابة خوارزمية لخطوات صنع كأس من الشاي فأننا نكتب ما يأتي:

١ -ضع الماء في غلاية الشاي.
٢ -سخن الماء حتى يغلى.
٣ -أضف شايا إلى الماء.
٤.- أطفئ النار.
٥.- أصب شايا في كأس.
٦.- أضف سكرا إليه.

افرض الآن أننا نود طلب كأس من الشاي من مقهى مجاور : أن خطوات الخوارزمية التي نحتاجها الآن هي خطوه واحده فقط وهي:

١ -استدع كأس من الشاي.

تخيل الآن كم وفرنا من الخطوات لو استعملنا الدوال الجاهزة (أو التي يجهزها المبرمج من قبل) بدلا من خطواتها التفصيلية وبخاصة في برنامج يتطلب حسابات وعمليات كثيرة وكم يكون البرنامج سهلا وواضحا وقت ذاك.

2.6- تحديد أو تعريف الدالة (Defining a Function):-

أن الدالة قد تعتمد على متغير أو اكثر ، وقد لا تعتمد على آي متغير ، وفي كلا الحالتين ، يستعمل بعد اسم الدالة قوسين ( ) سواء كان بينهما متغيرات أم لا .

يأخذ تعريف الدوال في C++ الشكل العام التالي:

return-value-type function-name (parameter list)

```
{
        declarations and statements
}
```

حيث:

return-value-type: نوع القيمة المعادة بواسطة الدالة والذي يمكن أن يكون أي نوع من أنواع بيانات C++. وإذا كانت الدالة لا ترجع أي قيمة يكون نوع إعادتها void.

function-name: اسم الدالة والذي يتبع في تسميته قواعد تسمية المعرفات (identefiers).

parameter list: هي لائحة الوسيطات الممررة إلى الدالة وهي يمكن أن تكون خالية (void) أو تحتوى على وسيطة واحدة أو عدة وسائط تفصل بينها فاصلة ويجب ذكر كل وسيطة على حدة.

void square(int a, int b) → a,b are the formal arguments.

float display(void) → function without formal arguments

declarations and statements: تمثل جسم الدالة والذي يطلق عليه في بعض الأحيان block . يمكن أن يحتوى الـ block على إعلانات المتغيرات ولكن تحت أي ظرف لا يمكن أن يتم تعريف دالة داخل جسم دالة أخرى. السطر الأول في تعريف الدالة يدعى المصرح declarator والذي يحدد اسم الدالة ونوع البيانات التي تعيدها الدالة وأسماء وأنواع وسيطاتها.

3.6- استدعاء الدالة (Function Call):-

يتم استدعاء الدالة (التسبب بتنفيذها من جزء آخر من البرنامج، العبارة التي تفعل ذلك هي استدعاء الدالة) يؤدى استدعاء الدالة إلي انتقال التنفيذ إلي بداية الدالة.

يمكن تمرير بعض الوسيطات إلى الدالة عند استدعائها وبعد تنفيذ الدالة يعود التنفيذ للعبارة التي تلي استدعاء الدالة.

بإمكان الدالة أن تعيد قيم إلي العبارة التي استدعتها. ويجب أن يسبق اسم الدالة في معرفها وإذا كانت الدالة لا تعيد شيئاً يجب استعمال الكلمة الأساسية void كنوع إعادة لها للإشارة إلي ذلك .

هنالك ثلاث طرق يمكن بها إرجاع التحكم إلى النقطة التي تم فيها استدعاء الدالة:

- إذا كانت الدالة لا ترجع قيمة يرجع التحكم تلقائياً عند الوصول إلى نهاية الدالة.
- باستخدام العبارة **return;**
- إذا كانت الدالة ترجع قيمة فالعبارة return expression; تقوم بإرجاع قيمة التعبير expression إلى النقطة التي استدعتها .

Example 1:- Write C++ program, to uses a function called square to calculate the number squares from 1 to 10?

```
#include<iostream.h>
int square(int y)
{
        return y*y;
}
main()
{
for(int x=1;x<=10;x++)
  cout<<square(x)<<" ";
  cout<<endl;
}
```

يتم استدعاء الدالة square داخل الدالة main وذلك بكتابة square(x). تقوم الدالة square بنسخ قيمة x في الوسيط y ثم تقوم بحساب y*y ويتم إرجاع النتيجة إلى الدالة main مكان استدعاء الدالةsquare ، حيث يتم عرض النتيجة وتتكرر هذه العملية عشر مرات باستخدام حلقة التكرار.for
تعريف الدالة square( ) يدل على أنها تتوقع وسيطة من النوع int . و int التي تسبق اسم الدالة تدل على أن القيمة المعادة من الدالة square هي من النوع int أيضاً . العبارة return تقوم بإرجاع ناتج الدالة إلى الدالة main.

Example 2:- Write C++ program, to print the statement "Mustansiriyah of University" by using function ?

```
#include<iostream.h>
void printmessage ( )
{
cout << "University of Technology";
}
void main ( )
{
printmessage( );
}
```

Example 3:- Write C++ program, to prints the largest number between two numbers entered by the user using function?

```
 #include < iostream.h>
int x,y;
max( )
{
if (x>y)
cout<<x;
else
cout<<y;
}
main()
{
cin>>x>>y;
max( );
return0;
}
```

Example 4:-Write C++ program using function to calculate the average of two numbers entered by the user in the main program?

```
#include<iostream.h>
float aver (int x1, int x2)
{
float z;
z = ( x1 + x2) / 2.0;
return ( z);
}
void main( )
```

```
{
float x;
int num1,num2;
cout << "Enter 2 positive number  \n";
cin >> num1  >> num2;
x = aver (num1,  num2);
cout << x;
}
```

Example 5:-Write C++ program, using function, to find the summation of the following Series?

$$\sum_{i=1}^{n} i^2 = 1^2 + 2^2 + 3^2 + \cdots + n^2$$

```
#include<iostream.h>
int summation ( int x)
{
int i = 1, sum = 0;
while ( i <= x )
{
sum += i * i ;
i++;
}
return (sum);
}
void main ( )
{
int n ,s;
cout << "enter positive  number";
cin >> n;
s = summation ( n );
cout << "sum is: " << s << endl;
}
```

Example 6:-Write a function to find the largest integer among three integers entered by the user
in the main function?

```
#include <iostream.h>
int max(int  y1, int y2, int y3)
{
int big;
big=y1;
if (y2>big) big=y2;
if (y3>big) big=y3;
return (big);
}
void main( )
{
int largest,x1,x2,x3;
```

```
cout<<"Enter 3 integer numbers:";
cin>>x1>>x2>>x3;
largest=max(x1,x2,x3);
cout<<largest;
}
```

Example 7:-Write program in C++, using function, presentation for logic gates (AND, OR ,NAND, X-OR,NOT) by in A,B enter from user?

```
#include<iostream.h>
void ANDF(int a,int b)
{
cout<<(a&&b);
}
void ORF(int a,int b)
{
cout<<(a||b);
}
void XORF(int a,int b)
{
cout<<(a^b);
}
void NOTF(int a)
{
cout<<(!a);
}
void main()
{ char s;int a,b;
cout<<"Enter the value A,B :";
cin>>a>>b;
cout<<"Enter the select value \n";
cout<<"\ta--(AND gate)\n\to--(OR gate)\n\tx--(X-OR)\n\tn--(NOT
gate)\n\te--(<<EXIT>> :";
cin>>s;
switch(s)
{
case 'a':ANDF(a,b);break;
case 'o':ORF(a,b);break;
case 'x':XORF(a,b);break;
case 'n':NOTF(a);cout<<"  ";NOTF(b);break;
case 'e':break;
deafult:cout<<":bad choose";
}
}
```

Example 8:-write C++ program, using function, to find (search) X value in array, and return the index of it's location?

```
#include<iostream.h>
int search( int a[ ], int y)
```

```
{
int i= 0;
while ( a [ i ] != y )
i++;
return ( i );
}
void main ( )
{
int X, f;
int a [ 10 ] = { 18, 25, 36, 44, 12, 60, 75, 89, 10, 50 };
cout << "enter value to find it: ";
cin >> X;
f= search (a, X);
cout << "the value " << X << " is found in location "<< f;
}
```

4.6- نموذج الدالة (Function Prototype):-

عندما نحتاج لاستدعاء دالة ما فإنه يحتاج إلى معرفة اسم الدالة وعدد وسيطاتها وأنواعها ونوع قيمة الإعادة، لذا علينا كتابة نموذج أو (تصريح) للدالة قبل إجراء أي استدعاء لها وتصريح الدالة هو سطر واحد عبارة عن اسم الدالة وعدد وسيطاتها وأنواعها ونوع القيمة المعادة بواسطة الدالة. يشبه تصريح الدالة، السطر الأول في تعريف الدالة، لكن تليه فاصلة منقوطة. فمثلا في تصريح الدالة التالي:-

int anyfunc(int);

النوع int بين القوسين يخبر بأن الوسيط الذي سيتم تمريره إلى الدالة سيكون من النوع int و int التي تسبق اسم الدالة تشير إلى نوع القيمة المعادة بواسطة الدالة.

عند تعريف الدالة هناك طريقتين **الاولى** تدعى definition وهي انه الدالة تكون مكتوبة قبل الاستدعاء اي يكون الاستدعاء من الاسفل الى الاعلى كما هو مكتوب في كل الامثلة السابقة في الدوال.

اما الطريقة الثانية تدعى declaration كما تدعى احيانا header وهي انه الدالة تكون مكتوبة بعد الاستدعاء اي يكون الاستدعاء من الاعلى الى الاسفل كما في المثال التالي.

Example:- Write C++ program, using function, to print the cub of values?

```
#include<iostream.h>
int cube(int);
void main()
{
cube(2);
}
int cube(int x)
{
return x*x*x;
}
```

كما ان استدعاء الدالة ليس شرطا ان يكون من دالة main فقط ،اذ يمكن ان تستدعى من دالة أخرى وتلك الدالة تستدعى من main كما هو موضح في المثال التالي :-

```
#include<iostream.h>
Int cube(int);
void call()
{
cout<<cube(2);
```

```
}
void main()
{
call();
}
int cube(int x)
{
return x*x*x;
  }
```

5.6- تمرير الوسيطات (Passing Parameters):-

الوسيطات او المعلمات بالأساس يمكن تصنيفها الى مجموعتين (الفعلية والرسمية).

الوسيطة الفعلية(actual arguments):هي متغير أو تعبير مضمن في استدعاء دالة الذي يحل محل المعلمة الرسمية التي هو جزء من إعلان الدالة. في بعض الأحيان ، قد تكون دالة استدعت جزء من البرنامج مع بعض المعلمات وهذه تُعرف بالمعلمة الفعلية.

الوسائط الرسمية(formal arguments): هي المعلمات الموجودة في تعريف الدالة والتي يمكن أيضا أن تسمى بالمعلمة الوهمية أو محدودة المتغيرات. عند استدعاء الدالة ، تكون المعلمات الرسمية استبدلت من قبل المعلمات الفعلية. قد يتم الإعلان عن الوسائط الرسمية بنفس الاسم أو بأسماء مختلفة عند استدعاء جزء من البرنامج أو في دالة الاستدعاء ولكن يجب أن تكون أنواع البيانات هي نفسها في كلا المجموعتين.

example:

```
# include <iostream.h>
Void main()
{
Int x,y;
Void output (int x, int y); // function declaration
—
—
Output  (x,y); // x and y are actual arguments
}
Void output (int a, int b) // formal arguments
{
// body of function
}
```

هناك طريقتان رئيسيتان لتمرير المعلمات أو الوسيطات إلى البرنامج :-

1- التمرير بالقيمة (passing by value)

2- التمرير بالمرجع (passing by reference)

التمرير بالقيمة (passing by value):-

عندما يتم تمرير المعلمات حسب القيمة ، نسخة من المعلمات يتم أخذ القيمة من دالة الاستدعاء وتمريرها إلى الدالة المطلوبة. لن تتغير المتغيرات الأصلية داخل دالة الاستدعاء ، بغض النظر عن التغييرات التي تجريها الدالة عليها.

لنفرض أننا لدينا متغيرين صحيحين في برنامج ونريد استدعاء دالة تقوم بتبديل قيمتي الرقمين ،لنفرض أننا عرفنا الرقمين كالآتي:

```
int x=1;
int y=2;
```

ترى هل تقوم الدالة التالية بتبديل القيمتين:

```
#include <iostream.h>
void swap (int a, int b)
{
int temp =a;
a=b.
b=temp;
}
main ( )
{
int x= 1;
int y= 2;
 swap (x, y);
}
```

تقوم هذه الدالة بتبديل قيمتي a و b ، لكن إذا استدعينا هذه الدالة كالآتي:

```
swap( x,y);
```

سنجد أن قيمتي x و y لم تتغير وذلك لأن الوسيطات الاعتيادية للدالة يتم تمريرها بالقيمة وتنشئ الدالة متغيرات جديدة كلياً هي a و b في هذا المثال لتخزين القيم الممررة إليها وهي (1,2) ثم تعمل على تلك المتغيرات الجديدة وعليه عندما تنتهي الدالة ورغم أنها قامت بتغيير a إلى 2 و b إلى 1 لكن المتغيرات x و y في استدعاء الدالة لم تتغير.

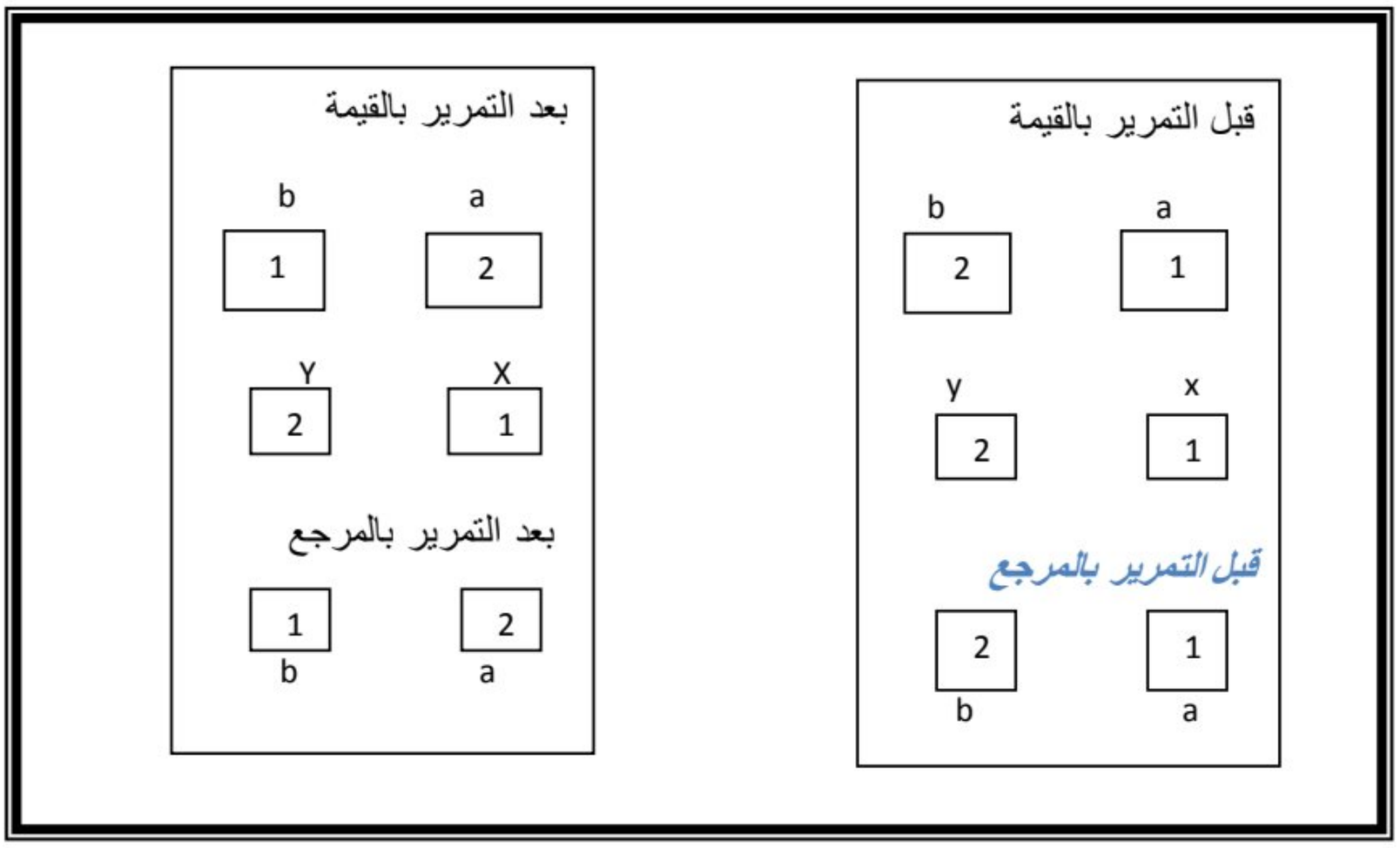
التمرير بالمرجع (passing by reference):-

عندما يتم تمرير المعلمات بالرجوع يتم نسخ عناوينهم إلى الوسيطات المقابلة في الدالة المدعومة، بدلاً من نسخ قيمها. وبالتالي، عادةً ما تستخدم المؤشرات في قائمة وسيطات الدوال لتلقي المراجع التي تم تمريرها.

التمرير بالمرجع هو طريقة تمكن الدالة swap( ) من الوصول إلى المتغيرات الأصلية x و y والتعامل معها بدلاً من إنشاء متغيرات جديدة . ولإجبار تمرير الوسيطة بالمرجع نضيف الحرف & إلى نوع بيانات الوسيطةفي تعريف الدالة وتصريح الدالة .

```
#include <iostream.h>
void swap (int& a, int & b)
{
cout <<"Original  value of a is " << a<<endl;
int temp =a;
a=b;
b=temp;
cout <<"swapped value of a is " << a<<endl;
}
main ( )
{
int x= 1;
int y= 2;
 swap (x, y);
return 0;
}
```

الحرف & يلي int في التصريح والتعريف وهو يبلغ المصرف أن يمرر هذه الوسيطات بالمرجع، أي أن الوسيطة a هي مرجع إلى x وb هي مرجع إلى y ولا يستعمل & في استدعاء الدالة.



شكل يوضح طريقتي التمرير بالمرجع والتمرير

طريقة التمرير بالمرجع أكثر فاعلية وتوفر سرعة تنفيذ أعلى من طريقة التمرير بالقيمة ، ولكن التمرير بالقيمة أكثر مباشرة وسهلة الاستخدام.

5.6- أنواع الدوال (Types of Functions):-

يمكن تصنيف الدوال المعرفة من قبل المستخدم بالطرق الثلاث التالية بناءً على الوسائط الرسمية التي تم تمريرها واستخدام بيان الإرجاع ، وبناءً على ذلك ، هناك ثلاث دوال محددة من قبل المستخدم:

1. يتم استدعاء الدالة دون تمرير أي وسيطة رسمية من الجزء المتصل بالبرنامج وأيضًا لا تقوم الدالة بإرجاع أي قيمة إلى الدالة التي تم استدعاؤها.
2. يتم استدعاء الدالة باستخدام الوسائط الرسمية من جزء الاستدعاء في البرنامج ، لكن الدالة لا تُرجع أي قيمة إلى جزء الاستدعاء.
3. يتم استدعاء الدالة باستخدام الوسائط الرسمية من جزء الاستدعاء في البرنامج الذي يعيد قيمة إلى بيئة الاستدعاء.

فيما يلي برنامجان يقومان بطباعة مربع العدد باستخدام عبارة الإرجاع مرة وتارة عدم استخدامها.

# include <iostream.h> 	
void square(int n)
{
float value;
value=n*n;
cout<<"i="<<n<<"square="<<value<<endl;
}
void main()
{
int max;
cout<<"Enter a value for n ?\n";
cin>>max;
for (int i=0;i<=max-1;++i)
square (i)
}
	# include <iostream.h>
	float square(float n)
	{
	float value;
	value=n*n;
	return (value);
	}
	void main()
	{
	float I,max,value;
	max=1.5;
	i=-1.5;
	while (i<=max) {
	value=square(i);
	cout<<"i="<<i<<"square="<<value<<endl;
	i=i+0.5;
	}
	}

Example:- Write program in C++, using function print sum for two integer values?

```
#include<iostream.h>
void sum()
{
int x = 5 , y = 4;
int z = x + y;
cout<<z;
}
void main()
{
sum();
}
```

لاحظ هنا الدالة من نوع void اي لا ترجع شيء و بنفس الوقت لا تمرر وسيطات او معلمات اي الاقواس فارغة بالتالي لا تاخذ شيء.

6.6- دالة اعادة الاستدعاء(Recursive Functions):-

دالة التي تدعو نفسها مباشرة أو غير مباشرة مرارا وتكرارا هي المعروف بأسم دالة اعادة الاستدعاء. و هذه الدوال مفيدة جدًا أثناء إنشاء هياكل البيانات مثل القوائم المرتبطة(linked lists) والقوائم المرتبطة المزدوجة(double linked lists) والأشجار(trees). هناك اختلاف واضح بينها و بين الدوال العادية . سيتم استدعاء الدالة العادية بواسطة الدالة الرئيسية(main) كلما تم استخدام اسم الدالة ، في حين سيتم استدعاء دالة اعادة الاستدعاء من تلقاء نفسها بشكل مباشر أو غير مباشر طالما تم استيفاء الشرط المحدد،فهي تستخدم عوضا عن ايعازات التكرار(loop statements) و التي تتطلب نهاية لحلقة التكرار(end point) لذا نستخدم مع هذا النوع من الدوال (Recursive Functions)عبارات التحكم (control statement)،لكن البرنامج المستخدم فيه دالة اعادة

الاستدعاء تتطلب ذاكرة اكبر من البرنامج المستخدم فيه ايعازات التكرار و بالتالي عملية تنفيذه تكون ابطأ ورغم ذلك فهو افضل في البرامج المعقدة ،و دالة اعادة الاستدعاء تعمل وفق مبدا المكدس(stack).
المثال ادناه يوضح عمل دالة اعادة الاستدعاء:-

```
# include <iostrem.h>
void main(void)
{
void func1(); //function declaration
_____
_____
func1(); //function calling
}
void func1() //function definition
{
____
____
func1(); //function calls recursively
}
```

Example 1:-write program to find the sum of the given non negative integer numbers using a recursive function sum=1+2+3+4+…+n

```
#include <iostream.h>
void main(void)
{
int sum(int);
int n,temp;
cout<<"Enter any integer number"<<endl;
cin>>n;
temp=sum(n);
cout<<"value="<<n<<"and its sum="<<temp;
}
int sum(int n) //recursive function
{
int sum(int); //local function declaration
int value=0;
if (n==0)
return (value);
else
value=n+sum(n-1);
return (value);
}
```

Example 2:- write program to find the factorial (n!) of the given number using the recursive function.Its the product of all integers from 1 to n (n is non negative)
(so n!=1 if n=0 and n!=n(n-1) if n>0)

```
#include <iostream.h>
void main(void)
{
long int fact (long int);
int x,n;
cout<<"Enter any integer number"<<endl;
cin>>n;
x=fact(n);
cout<<"value="<<n<<"and its factorial=";
cout<<x<<endl;
}
long int fact (long int n) //recursive function
{
long int fact(long int); //local function declaration
int value =1;
if (n==1)
return(value)
else
{
value=n*fact(n-1);
return(value);
}
}
```

Home Work (6)

Q1: Write a C++ program, using function, to counts uppercase letter in a 20 letters entered by the user in the main program?

Q2: Write a C++ program, using function, that reads two integers (feet and inches) representing distance, then converts this distance to meter?

Note: 1 foot = 12 inch

1 inch = 2.54 Cm

Q3: Write a C++ program, using function, which reads an integer value (T) representing time in seconds, and converts it to equivalent hours (hr), minutes (mn), and seconds (sec), in the following form:- **hr : mn : sec**

Q4: Write a C++ program, using function, to see if a number is an integer (odd or even) or not an integer?

Q5: Write a C++ program, using function, to represent the permutation of n?

Q6: Write a C++ program, using function, to inputs a student's average and returns 4 if student's average is 90-100, 3 if the average is 80-89, 2 if the average is 70-79, 1 if the average is 60-69, and 0 if the average is lower than 60?

Q7: The Fibonacci Series is: 0, 1, 1, 2, 3, 5, 8, 13, 21, … It begins with the terms 0 and 1and has the property that each succeeding term is the sum of the two preceding terms, Write a C++ program, using function, to calculate the nth Fibonacci number?

Q8: Write a C++ program, using function, to calculate the factorial of an integer entered by the user at the main program?

Q9: Write a C++ program, using function, to evaluate the following equation:

$$z = \frac{x! - y!}{(x - y)!}$$

Q10: Write a C++ program, using function, to test the year if it's a leap or not.

Note: use y % 4 == 0 && y % 100 != 0 :: y % 400 ==0

Q11: Write a C++ program, using function, to find x^y.

Note: use pow instruction with math.h library.

Q12:Write C++ program, using function, to inverse an integer number: For example:

765432 ⟶ 234567

Q13: Write C++ program, using function, to find the summation of student's marks, and it's average, assume the student have 8 marks?

Q14: Write C++ program, using function, to convert any char. From capital to small or from small to capital?

Q15: Write C++ program using recursive function to find the power of numbers?

Q16: Write a C++ program, using function, to find if the array's elements are in order or Not?

Q17: Write a C++ program, using function, to compute the number of zeros in the array?

Q18: Write a C++ program, using function, to find the value of array C from add array A and array B. C[i] = A [i] + B [i];

Q19: Write a C++ program, using function, to multiply the array elements by 2?

A[i] = A [i] * 2;

Q20: Write a C++ program, using function, to reads temperatures over the 30 days and calculate the average of them?

Q21: Write a C++ program, using function, to merge two arrays in one array?

الفصل الخامس (Section 5)
المصفوفات (Arrays)

1.5- مقدمة :-

أن طرق التعامل مع أسماء المتغيرات والثوابت العددية والرمزية ، التي وردت في الفصول السابقة ، تعد صالحة للتعامل مع عدد محدود من هذه الثوابت والمتغيرات ،سواء في عمليات الإدخال والإخراج أو في العمليات الحسابية والمنطقية ، وعندما يصبح عدد المتغيرات كبيرا جدا ، تصبح تلك الطرق غير عملية ، فمثلا لو أردنا إدخال مائة قيمة للمتغيرات x1, x2,....,x100، فكم الحيز المطلوب من البرنامج لعمليات الإدخال والإخراج والعمليات الحسابية والمنطقية لهذه المتغيرات ؟

هذا من جهة ، ومن جهة أخرى : فأننا نوفر مخزنا خاصا لك لمتغير نتعامل معه ،أثناء تنفيذ البرنامج ، ولذلك لحفظ قيمته في مخزن ، ومن ثم لاستعمال قيمته في عمليات أخرى تالية ، ومن ناحية ثالثة ، فان من الصعوبة بمكان ، بل من المستحيل استعمال اسم المتغير العددي أو الرمزي كمصفوفة ذات بعدين ، وثلاثة أبعاد... الخ.

للأسباب الواردة أعلاه ، جاءت فكرة استعمال متغير جماعي يضم تحت اسمه عددا من العناصر يسمى المصفوفة (Array)، المصفوفة هي نوع من أنواع بنية البيانات وهي مجموعة من المواقع المتجاورة في الذاكرة، لها عدد محدود ومرتب من العناصر التي تكون جميعها من نفس النوع type، فمثلاً يمكن أن تكون جميعها صحيحة int أو غير صحيحة float ولكن لا يمكن الجمع بين نوعين مختلفين في نفس المصفوفة .

الشكل التالي يبين مصفوفة C تحتوى على 13 عنصر من النوع int، ويمكن الوصول إلي أي من هذه العناصر بذكر اسم المصفوفة متبوعاً برقم موقع العنصر في المصفوفة محاطاً بالأقواس [ ] .

يرمز لرقم العنصر في المصفوفة بفهرس العنصر (index). فهرس العنصر الأول في المصفوفة هو 0 ولهذا يشار إلى العنصر الأول في المصفوفة C بـ C[0] والثاني C[1] والسابع C[6] وعموماً يحمل العنصر i في المصفوفة C الفهرس C[i-1] .تتبع تسمية المصفوفات نفس قواعد تسمية المتغيرات.

C[0]	-45
C[1]	6
C[2]	0
C[3]	72
C[4]	1543
C[5]	-89
C[6]	0
C[7]	62
C[8]	-3
C[9]	1
C[10]	6453
C[11]	78
C[12]	15

أحياناً يسمى فهرس العنصر برمز منخفض subcript ويجب أن يكون الفهرس integer أو تعبير جبري تكون نتيجته integer . فمثلاً إذا كانت a=5 وb=6 فالعبارة: **C[a+b]+=2**
معناها نقوم بإضافة 2 إلي العنصر الثاني عشر C[11] في المصفوفة C.
يحمل العنصر 0 في المصفوفة C القيمة 45- والعنصر 1 القيمة 6. لطباعة مجموع الثلاثة عناصر الأولى في المصفوفة C يمكن كتابة:

```
cout<<C[0]+C[1]+C[2]<<endl;
```

من الفوائد المهمة للمصفوفات : هو استعمالها في الترتيب التصاعدي والتنازلي للعناصر والقيم المختلفة ، وعمليات ترتيب الأسماء الأبجدية او النصوص الرمزية ، وفي عمليات ضرب المصفوفات ، وإيجاد معكوس المصفوفة وعملياتها الأخرى ، وفي التحليل العددي ... الخ.

2.5- كيفية الإعلان عن المصفوفة(How to declare the array):-

تحتل المصفوفات حيزاً في الذاكرة لذا يجب على المبرمج تحديد نوع عناصر المصفوفة وعددها حتى يتسنى للمعرف تخصيص الحيز اللازم من الذاكرة لحفظ المصفوفة و الصيغة العامة لها هي

data-type array name of array [*size*];

نوع بيانات المصفوفة — اسم المصفوفة — حجم المصفوفة

3.5- تهيئة عناصر المصفوفة (Initializing Array Elements):-

عند اعلان مصفوفة فأنه ممكن أن نذكر نوع المصفوفة و اسمها و ابعاد المصفوفة دون تعيين قيم او محتويات للمصفوفة و في مثل هذه الحالة تكون قيمتها 0 كما في المثال الاتي:- Int

```
a[20];
```

كذلك ممكن ان نعلن عن محتويات المصفوفة عند تعريفها و المثال التالي يوضح ذلك

```
int a [10] = { 8, 10, 13, 15, 0, 1, 17, 22};

int x [ ] = { 12, 3, 5, 0, 11, 7, 30, 100, 22 };

age [9] = { 18, 17, 18 ,18 ,19, 20 ,17, 18 ,19 };
```

4.5- كيفية الوصول إلى عناصر المصفوفة(How to Accessing Array Elements):-

نحن نصل إلى كل عنصر من عناصر المصفوفة بالاسم المكتوب للمصفوفة ، يليهاالأقواس التي تحدد المتغير (أو الثابت) في الأقواس التي تسمى فهرس المصفوفة وللوصول الى عناصر المصفوفة نقوم بتحديد موقع العنصر داخل المصفوفة والمطلوب الوصول إليه كما في الامثلة التالية التابع للمثال اعلاه:-

- The first element of array age:

```
age [0] = 18;
```

- The last element of array age:

```
age [9] = 19;
```

مثال اخر للمصفوفة اسمها num[10]

- Accessing the first element of array num to variable x:

x = num [0]; تعيين العنصر الاول من المصفوفة للمتغير x

- Accessing the last element of array num to variable y:

y = num [9]; تعيين العنصر الاول من المصفوفة للمتغير y

- cout << num [0] + num [9];

- num [0] = num [1] + num[2];

-num [7] = num [7] + 3; ⟷ num [7] += 3;

5.5 - انواع المصفوفات (types of array):-

و تقسم المصفوفات الى عدة انواع بالاعتماد على ابعاد المصفوفة و هي :-

1. **المصفوفة ذو البعد الواحد (One-dimensional Array):-**

هو مصفوفة ذات بعد واحد أو متجه (vector) ويمثل في الجبر على النحو الأفقي [a_1 a_2 ...a_n] أو العمودي .

الصيغة العامة لمصفوفة البعد الواحد هي :-

data-type Array-name of array [*size*];

[حجم المصفوفة][اسم المصفوفة] [نوع بيانات المصفوفة]

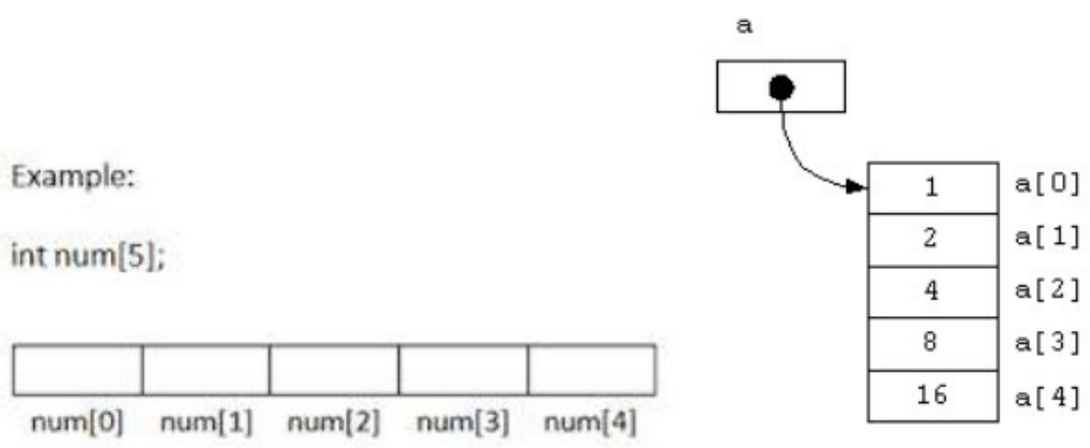
Examples:

int age [10];

int num [30];

float degree[5];

char a [15];



وحتى تخبر الذاكرة بأن يخصص حيزاً لـ 12 عنصر من النوع int في مصفوفة C ، استخدم الإعلان:

int C[12];

يمكن تخصيص الذاكرة لعدة مصفوفات باستخدام نفس الإعلان وذلك كالأتي:

int b[100], x[20];

أيضاً يمكن الإعلان عن مصفوفات من أي نوع بيانات آخر ، فمثلاً للإعلان عن مصفوفة عناصرها من النوع char نكتب:

char ch[20];

إعطاء قيمة أولية للمصفوفة ذات البعد الواحد(Array Initialization):-

مثال على إدخال عدة عناصر من مصفوفة الدرجات grade[ ]

int grade[5]={80,90,54,50,95};

ومثال أخر على إدخال قيم عناصر المصفوفة الرمزية name[ ]

Char name[4]="nor";

لاحظ أن المتغير المرقم name[ ] مكون من أربعة عناصر بينما تم إعطاؤه ثلاثة عناصر فقط والسبب أن العنصر الرابع بالنسبة إلى المعطيات الرمزية يكون خاليا.

```
#include < iostream.h>
main ()
{
int a[6]={40,60,50,70,80,90}
int I;
for(I=0;I<6;I++)
{
cout<<a[I]<<endl;
}
}
```

Example 1:- Write a program that finds a total, average student score in 5 subjects, and theseThe grades are as follows: 87,67,81,90,55?

```
#include < iostream.h>
main ()
{
int a[5]={87,67,81,90,55}
int s=0;
for(int i=0;i<5;i++)
s=s+a[i];
float avg=s/5;
cout<<avg<<endl;<<s<<endl;
}
```

Example 2-: Write C++ program to display 2nd and 5th elements of array distance?

```
#include<iostream.h>
void main( )
{
double distance[ ] = { 23.14, 70.52, 104.08, 468.78, 6.28};
cout << "2nd element is: " << distance[1] << endl;
cout << "5th element is: " << distance[4];
}
```

Example 3:-Write C++ program to read 5 numbers and print it in reverse order?

```
#include<iostream.h>
void main( )
{
int a [5];
cout << "Enter 5 numbers \n";
for ( int i =0; i <5; i++ )
{
cout << i << “: “;
cin >> a [ i ];
cout << “\n”;
}
cout << “The reverse order is: \n”;
for ( i =4; i >=0; i-- )
cout << i << “: “ << a [ i ] << endl;
}
```

Example 4:-Write C++ program, to find the summation of array elements?

```
#include<iostream.h>
void main ( )
{
int const L = 10;
int a [L];
int sum = 0;
cout << "enter 10 numbers \n";
for ( int i =0; i <L; i++ )
{
cout << “enter value “ << i << “: “;
cin >> a [ i ];
sum += a [ i ];
}
cout << "sum is: " << sum << endl;
}
```

Example 5:-Write C++ program, to find the minimum value in array of 8 numbers?

```
#include<iostream.h>
void main ( )
{
 int n = 8; int a [ ] = { 18, 25, 36, 44, 12, 60, 75, 89 };
int min = a [ 0 ];
for ( int i = 0; i < n; i++ )
if ( a [ i ] < min )
min = a [ i ];
cout << "The minimum number in array is: " << min;
 }
```

Example 6:- Write C++ program, to split the odd numbers and even numbers of one array into two arrays:

a = [1, 2, 3, 4, 5, 6, 7, 8, … , 20]
aodd = [1, 3, 5, 7, … , 19]
aeven = [2, 4, 6, 8, … , 20]

```
#include<iostream.h>
void main ( )
{
int a [ 20 ]= { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20 };
int aodd[20], aeven [20];
int i ,o=0, e=0;
for ( i=0 ; i<20; i++ )
if (a[i] % 2 !=0)
{
aodd[o]=a[i];
o=o+1;
}
else
{
aeven[e]=a[i];
e=e+1;
}
for ( i=0 ; i<o; i++ )
cout<<aodd[i]<<" ";
cout<<endl;
for ( i=0 ; i<e; i++ )
cout<<aeven[i]<<" ";
}
```

Example 7:- write C++ program to enter seven prime values for the array and finds the number of values that are greater than 50 and less than 50?

```
#include < iostream.h>
main ()
{
int i,S,L;
int a[5]={87,67,41,90,55,53,38}
for(i =0; i<=6;i++)
  {
    if(a[i]>=50)
     {
          L=L+1;
```

```
    }
   else
    {
       S=S+1;
    }
 }
  cout<<"50 اكبر من "<<L<<Endl;
  cout<<"50 اصغر من "<<S<<Endl;
}
```

Example 8:- write C++ program to enter ten initial values for the array and finds the number of negative values and the number of positive values?

```
#include < iostream.h>
main ()
{
int i,N,P;
int a[5]={87,-167,141,90,55,53,38,-4,-2, 1}
for(i =0; i<=9;i++)
 {
   if(a[i]>=0)
    {
        P=P+1;
    }
   else
    {
       N=N+1;
    }
 }
  cout<<"الاعداد الموجبة  "<<P<<Endl;
  cout<<"الاعداد السالبة "<<N<<Endl;
}
```

2. المصفوفة ذو البعد ين (Two-dimensional Array):-

يمكن للمصفوفات في C+ + أن تكون متعددة الأبعاد ويمكن كذلك أن يكون كل بعد بحجم مختلف ، الاستعمال الشائع للمصفوفات متعددة الأبعاد هو تمثيل الجداول Tables التالي تحتوي على بيانات مرتبة في صورة صفوف وأعمدة ولتمثيل الجدول نحتاج لبعدين الأول يمثل الصفوف والثاني يمثل الأعمدة.

الشكل التالي يبين مصفوفة A تحتوى على ثلاثة صفوف وأربع أعمدة.

	Column 0	Column1	Column2	Column 3
Row 0	A[0][0]	A[0][1]	A[0][2]	A[0][3]
Row 1	A[1][0]	A[1][1]	A[1][2]	A[1][3]
Row 2	A[2][0]	A[2][1]	A[2][2]	A[2][3]

تشبه المصفوفة ذات البعدين في طريقة تعاملها ، المصفوفة ذات البعد الواحد إلا أن لها عدادين(2 index) إحداهما عداد للصفوف ، والأخر عداد للأعمدة ويأخذ الإعلان عن المصفوفة الشكل العام التالي:

Type-specifier arraey_name [index 1][index 2];

يتم تمثيل أي عنصر في المصفوفة A على الصورة A[i][j] حيث:-
A : اسم المصفوفة.
i : رقم الصف الذي ينتمي إليه العنصر.
j : رقم العمود الذي ينتمي إليه العنصر.
لاحظ أن كل العناصر الموجودة في الصف الأول مثلاً يكون الفهرس الأول لها هو 0 وكل العناصر الموجودة في العمود الرابع يكون الفهرس الثاني لها هو 3.
يتم الإعلان عن مصفوفة a تحتوى على x صف و y عمود هكذا:

int a[x][y];

2. تهيئة عناصر مصفوفة ثنائية الأبعاد:
يمكن تمهيد قيمة المصفوفة المتعددة الأبعاد عند الإعلان عنها وذلك كالآتي:

int b[2][2]={{1,2},{3,4}};

حيث:

b[1][1]=4, b[1][0]=3, b[0][1]=2, b[0][0]=1

أيضاً هنا في المصفوفة متعددة الأبعاد إذا تم تمهيدها عند قيم لا يتوافق عددها مع حجم المصفوفة فإن المصرف سيملأ بقية العناصر أصفار.

Example 1:-write C++ program to enter the grades of 5 students into 3 subjects?

```
#include <iostream.h>
main ()
{
int m[5][3];
int I,j;
for(I=0;I<5;I++)
 {
for(j=0;j<3;j++)
  {
    cin>>m[I][j];
  }
 }
}
```

Example 2:-Write C++ program, to read 15 numbers, 5 numbers per row, the print them?

```
#include<iostream.h>
void main ( )
{
int a [ 3 ] [ 5 ];
int i , j;
```

```
for ( i = 0 ; i < 3; i++ )
for ( j = 0 ; j < 5; j++ )
cin >> a [ i ] [ j ];
for ( i = 0 ; i < 3; i++ )
{
for ( j = 0 ; j < 5; j++ )
cout << a [ i ] [ j ];
cout << endl;
}
}
```

Example 3:- Write C++ program, to read 4*4 2D-array, then find the summation of the array elements, finally print these elements?

```
#include<iostream.h>
void main ( )
{
int a [ 4 ] [ 4 ];
int i , j, sum = 0;
for ( i = 0 ; i < 4; i++ )
for ( j = 0 ; j < 4; j++ )
cin >> a [ i ] [ j ];
for ( i = 0 ; i < 4; i++ )
for ( j = 0 ; j < 4; j++ )
sum += a [ i ] [ j ];
cout << "summation is: " << sum << endl;
for ( i = 0 ; i < 4; i++ )
{
for ( j = 0 ; j < 4; j++ )
cout << a [ i ] [ j ];
cout << endl;
}
}
```

Example 4:- Write C++ program, to read 3*4 2D-array, then find the summation of each row?

```
#include<iostream.h>
void main ( )
{
int a [ 3 ] [ 4 ];
int i , j, sum = 0;
for ( i = 0 ; i < 3; i++ )
for ( j = 0 ; j < 4; j++ )
cin >> a [ i ] [ j ];
for ( i = 0 ; i < 3; i++ )
{
sum = 0;
for ( j = 0 ; j < 4; j++ )
sum += a [ i ] [ j ];
```

```
cout << “summation  of row “ << i << “ is: “ << sum << endl;
}
}
```

Example 5:-Write C++ program, to read 3*4 2D-array, then replace each value equal 5 with 0?

```
#include<iostream.h>
void main ( )
{
int a [ 3 ] [ 4 ];
int i , j;
for ( i = 0 ; i < 3; i++ )
for ( j = 0 ; j < 4; j++ )
cin >> a [ i ] [ j ];
for ( i = 0 ; i < 3; i++ )
for ( j = 0 ; j < 4; j++ )
if ( a [ i ] [ j ] == 5 ) a [ i ] [ j ] = 0;
for ( i = 0 ; i < 3; i++ )
{
for ( j = 0 ; j < 4; j++ )
cout << a [ i ] [ j ];
cout << endl;
}
}
```

Example 6:-Write C++ program, to addition two 3*4 arrays?

```
#include<iostream.h>
void main ( )
{
int a [ 3 ] [ 4 ], b [ 3 ] [ 4 ], c [ 3 ] [ 4 ];
int i , j;
cout << "enter element of array A: \n";
for ( i = 0 ; i < 3; i++ )
for ( j = 0 ; j < 4; j++ )
cin >> a [ i ] [ j ];
cout << "enter element of array B: \n";
for ( i = 0 ; i < 3; i++ )
for ( j = 0 ; j < 4; j++ )
cin >> b [ i ] [ j ];
for ( i = 0 ; i < 3; i++ )
for ( j = 0 ; j < 4; j++ )
c [ i ] [ j ] = a [ i ] [ j ] + b [ i ] [ j ];
for ( i = 0 ; i < 3; i++ )
{
for ( j = 0 ; j < 4; j++ )
cout << c [ i ] [ j ];
cout << endl;
}
}
```

Example 7:-Write C++ program, to convert 2D-array into 1D-array?

```
#include<iostream.h>
void main ( )
{
int a [ 3 ] [ 4 ];
int b [ 12 ];
int i , j, k = 0;
for ( i = 0 ; i < 3; i++ )
for ( j = 0 ; j < 4; j++ )
cin >> a [ i ] [ j ];
for ( i = 0 ; i < 3; i++ )
for ( j = 0 ; j < 4; j++ )
{
b [ k ] = a [ i ] [ j ];
k++;
}
for ( i = 0 ; i < k; i++ )
cout << b [ i ];
}
```

Example 8:-Write C++ program, print the square root of an array?

```
#include<iostream.h>
void main ( )
{
int a [ 3 ] [ 3 ] , b [ 3 ] [ 3 ];
int i , j;
for ( i = 0 ; i < 3; i++ ) {
for ( j = 0 ; j < 3; j++ ) {
b[ i ][ j ]= sqrt(a[ i ][ j ]);
cout << b [ i ] [ j ];
}
 }
}
```

0,0	0,1	0,2
1,0	1,1	1,2
2,0	2,1	2,2

i = j
Main diameter (diagonal)

0,0	0,1	0,2
1,0	1,1	1,2
2,0	2,1	2,2

i + j = n-1
secondary diagonal

0,0	0,1	0,2
1,0	1,1	1,2
2,0	2,1	2,2

i > j
under main diagonal

0,0	0,1	0,2
1,0	1,1	1,2
2,0	2,1	2,2

i < j
above main diagonal

Example 9:-Write C++ program, to replace each element in the main diameter (diagonal) with zero?

```
#include<iostream.h>
void main ( )
{
int a [ 3 ] [ 3 ];
int i , j;
for ( i = 0 ; i < 3; i++ )
for ( j = 0 ; j < 3; j++ )
cin >> a [ i ] [ j ];
for ( i = 0 ; i < 3; i++ )
for ( j = 0 ; j < 3; j++ )
if ( i == j ) a [ i ] [ j ] = 0;
for ( i = 0 ; i < 3; i++ )
{
for ( j = 0 ; j < 3; j++ )
cout << a [ i ] [ j ];
cout << endl;
}
}
```

Example 10:-Write C++ program, to read 3*3 2D-array, then find the summation of the main diagonal and its secondary diagonal of the array elements, finally print these elements?

```
#include<iostream.h>
void main ( )
{
int a [ 3 ] [ 3 ];
int i , j, x , y;
for ( i = 0 ; i < 3; i++ ) {
for ( j = 0 ; j < 3; j++ ) {
cin >> a [ i ] [ j ];
if ( i == j )
x=x+a[ i ][ j ];
if ( i + j =4)
y=y+a[ i ][ j ];
}
 }
cout << "summation  of diagonal is: " << x << endl;
cout << "summation  of inverse diagonal is: " << y << endl;
}
```

Home Work (5)

Q1: Write a C++ program, to enter 10 initial values for the array and print the largest value between them?

Q2: Write a C++ program, to enter 10 initial values for the array, and the value of the first element A [0] changes with the last element A [9] then print the array after change?

Q3: Write C++ program, to read 3*4 2D-array, then find the summation of each col?

Q4: Write C++ program, to replace each element in the second diameter (diagonal) with zero?

Q5: Write C++ program, to replace the elements of the main diameter with the elements of the second diameter?

Q6: Write C++ program, to find the summation of odd numbers in 2D-array?

Q7:Write C++ program, to find (search) X value in 2D-array, and return the index of it's location?

Q8: Write C++ program, to convert 1D-array that size [16] to 2D-array that size of [4] [4]?

Q9: Write C++ program, to read A[n, n] of character, then find array B and array C, such that B contain only capital letters and C contain only small letters?

Q10: Write C++ program, to read A[n, n] of numbers, then put 10 instead each even positive number?

Q11: Write C++ program, to read A[n, n] of numbers, then put 10 instead each even positive number in the first diagonal?

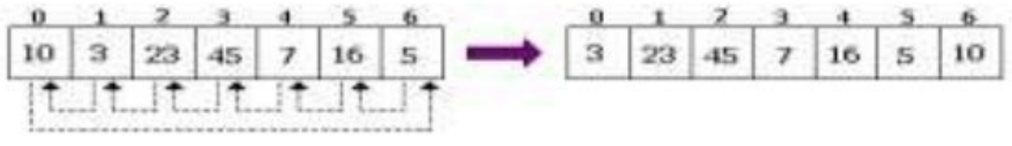
Q12: Write C++ program, to read A[n, n] of numbers, then find the minimum number in array?

Q13: Write C++ program, to exchange row1 and row3 in 4*3 array?

Q14: Write C++ program, to exchange row0 with col3 in 4*4 array?

Q15: Write C++ program, to find the greatest number in the second diagonal, in 3*3 array?

Q16: Write C++ program, to read X[n], and rotate the elements to the left by one position?



Q17: Write C++ program, to read A[n] and a location Z then delete the number at location Z from the array, and print the new array after deletion?

Q18: Write C++ program to order the array in ascending and descending order?

Q19: Write C++ program to read (n) no.s and find the average of the even no. on it?

Q20: Write C++ program to Create the array (b) from (a)?

a	b
1 2 3	6
4 5 6	10
7 8 9	10

Q21: Write C++ program to Create the arrays bellow?

1 1 1 1	2 1 1 1
2 2 2 2	1 2 1 1
3 3 3 3	1 1 2 1
4 4 4 4	1 1 1 2

كليـــة الهندسة

قسم هندسة الحاسوب

Subject: Object Oriented Programming (OOP)

first Stage

Lecturer: Dr. Azhaar A. hussan

Lecture (4)

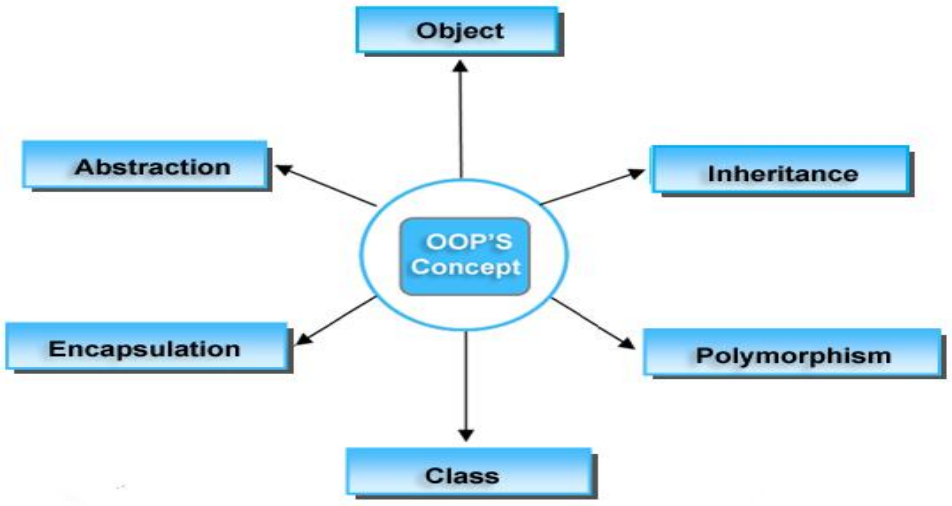
Introduction to Object-Oriented Programming (OOP)

1. Introduction to Object-Oriented Programming:

Object-Oriented Programming (OOP) is a programming paradigm that uses 'objects' to represent data and methods that operate on that data. This approach allows programmers to model real-world entities, making code more modular, reusable, and easier to maintain. Core Concepts of OOP:

- **Object**: An object is an instance of a class that contains attributes (data) and methods (functions) that operate on the data. Think of it as a real-world entity, like a car or a person.
- **Class**: A class is a blueprint for creating objects. It defines the attributes and methods that its objects will have.

2. Key Concepts of OOP:



2.1. Abstraction:

Abstraction is the concept of hiding complex implementation details and exposing only the necessary parts. This allows users to interact with objects without worrying about their internal workings.

Example: In a car, you interact with the steering wheel without needing to know how the engine works.

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

class Car {
public:
   void startEngine() {
      cout << "Engine started!" << endl;
   }
   void drive() {
      cout << "Car is driving!" << endl;
   }
};

int main() {
   Car myCar;
   myCar.startEngine();
   myCar.drive();
   return 0;
}
```

Explanation:

- The Car class abstracts the complexity of starting and driving a car. The user doesn't need to know how the engine starts internally, just that they need to call the startEngine() and drive() methods.
- When we create an object myCar, we can use these functions to simulate starting and driving the car, without needing to understand the inner workings of the engine.

2.2. Encapsulation:

Encapsulation is the bundling (تجميع) of data (attributes) and methods (functions) that operate on the data into a single unit or class. It also restricts direct access to some components, which is essential for protecting data integrity.

```
#include <iostream>
```

```
using namespace std;

class BankAccount {
private:
   int balance;

public:
   BankAccount(int initialBalance) {
      balance = initialBalance;
   }

   void deposit(int amount) {
      balance += amount;
   }

   void withdraw(int amount) {
      if (amount <= balance) {
         balance -= amount;
      } else {
         cout << "Insufficient funds!" << endl;
      }
   }

   int getBalance() {
      return balance;
   }
};

int main() {
   BankAccount account(500);
   account.deposit(200);
   account.withdraw(100);
   cout << "Current balance: " << account.getBalance() << endl;
   return 0;
}
```

Explanation:

- The BankAccount class encapsulates the balance attribute as a private variable. Users cannot modify it directly but can interact with it via public methods like deposit(), withdraw(), and getBalance().
- This protects the balance from being accidentally modified, ensuring that all changes occur through the proper methods.

2.3. Inheritance:

Inheritance allows one class to inherit (يرث) the attributes and methods of another class, promoting (يتميز) code reuse. The new class, known as the 'derived' or 'child' class, can also have its own additional attributes and methods.

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

class Animal {
public:
   void eat() {
      cout << "This animal is eating!" << endl;
   }
};

class Dog : public Animal {
public:
   void bark() {
      cout << "The dog is barking!" << endl;
   }
};

int main() {
   Dog myDog;
   myDog.eat();  // Inherited method
   myDog.bark(); // Dog-specific method
   return 0;
}
```

Explanation:

- The Dog class inherits from the Animal class. This means that all Dog objects can use the eat() method defined in Animal.
- Inheritance allows us to reuse the functionality of the Animal class while adding more specific methods, like bark(), to the Dog class.

3. Conclusion:

Object-Oriented Programming makes software design more organized and intuitive by breaking down a problem into objects. Through abstraction, encapsulation, inheritance, and polymorphism, OOP encourages code reuse, modularity, and ease of maintenance. Key Takeaways:

- **Abstraction** simplifies complex systems.
- **Encapsulation** protects data and ensures controlled access.
- **Inheritance** promotes code reuse.

كليـــة الهندسة

قسم هندسة الحاسوب

Subject: Object Oriented Programming (OOP)

First Stage

Lecturer: Dr. Azhaar a.hussan

Lecture (5)

Introduction to Object-Oriented Programming (OOP) -Continue

2.4. Polymorphism:

Polymorphism allows objects of different types to be treated as objects of a common superclass. It enables a single function or method to behave differently based on the object that calls it.

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

class Animal {
public:
   virtual void makeSound() {
      cout << "Some generic animal sound" << endl;
   }
};

class Dog : public Animal {
public:
   void makeSound() override {
      cout << "Woof!" << endl;
   }
};

class Cat : public Animal {
public:
   void makeSound() override {
      cout << "Meow!" << endl;
   }
};

int main() {
   Animal* myAnimal;
   Dog myDog;
   Cat myCat;

   myAnimal = &myDog;
   myAnimal->makeSound();  // Outputs: Woof!

   myAnimal = &myCat;
   myAnimal->makeSound();  // Outputs: Meow!

   return 0;
}
```

Explanation:

- Here, we use polymorphism to treat `Dog` and `Cat` objects as `Animal` objects. The method `makeSound()` behaves differently depending on whether it's called by a `Dog` or a `Cat`.
- Polymorphism enables flexibility and the ability to handle multiple object types through a common interface (in this case, `Animal`).

Example: Let's consider a simple example where we have different geometric shapes like circles, rectangles, and triangles. We want to write a method that can draw any shape, but the way we draw each shape is different.

To achieve this, we can create a base class Shape that has a virtual method draw(). Each derived class will override the draw() method to provide its specific implementation.

```
#include <iostream>

using namespace std;

class Shape {                    // Base class (superclass)

public:

    virtual void draw() {            // Virtual function to allow overriding in derived classes

      cout << "Drawing a generic shape" << endl;

   }

};

class Circle : public Shape {       // Derived class representing a circle

public:

      void draw() override {        // Overriding the draw() function for Circle

      cout << "Drawing a circle" << endl;

   }

};

class Rectangle : public Shape {      // Derived class representing a rectangle

public:

   void draw() override {          // Overriding the draw() function for Rectangle

      cout << "Drawing a rectangle" << endl;

   }

};

class Triangle : public Shape {     // Derived class representing a triangle

public:

   void draw() override {               // Overriding the draw() function for Triangle

      cout << "Drawing a triangle" << endl;
```

```
    }
};
int main() {
    Shape* shape;  // Pointer to base class (Shape)
    Circle circle;
    Rectangle rectangle;
    Triangle triangle;
    shape = &circle;        // Assign pointer to Circle object and call draw()
    shape->draw();       // Output: Drawing a circle
    shape = &rectangle;     // Assign pointer to Rectangle object and call draw()
    shape->draw();  // Output: Drawing a rectangle
    shape = &triangle;      // Assign pointer to Triangle object and call draw()
    shape->draw();  // Output: Drawing a triangle
    return 0;
}
```

3. Conclusion:

Object-Oriented Programming makes software design more organized and intuitive by breaking down a problem into objects. Through abstraction, encapsulation, inheritance, and polymorphism, OOP encourages code reuse, modularity, and ease of maintenance. Key Takeaways:

- **Abstraction** simplifies complex systems.
- **Encapsulation** protects data and ensures controlled access.

- **Inheritance** promotes code reuse.
- **Polymorphism** allows for flexibility in method behavior.

Al-shatrah University

كليـــة الهندسة

قسم هندسة الحاسوب

Subject: Object Oriented Programming (OOP)

First Stage

Lecturer: Dr. Azhaar a.hussan

Lecture (4)

Objects and Member Functions in Object-Oriented Programming

2024-2025

Objects and Member Access:

Objects are instances of a class, and through them, we can access the class's members (both attributes and functions). The two key concepts related to member access are:

1. **Public Members:** Accessible from outside the class.
2. **Private Members:** Accessible only from within the class.

When using objects, we access class members using the dot operator (.) for direct member access, or through member functions which are responsible for interacting with private data.

Example:

```
#include <iostream>
using namespace std;
class Student {
public:
    string name;
    int age;
    void displayInfo() {
        cout << "Name: " << name << ", Age: " << age << endl;
    }
};

int main() {
```

```
    Student student1;
    student1.name = "Alice";
    student1.age = 20;
    student1.displayInfo(); // Accessing member function
    return 0;
}
```

Explanation:

- We define a Student class with two public members (name and age).
- The displayInfo() function is a member function that prints the values of the name and age attributes.
- In main(), we create a Student object (student1) and access its members using the dot operator. We call displayInfo() to display the student's information.

---

2. Defining Member Functions:

There are two ways to define member functions in C++:

1. **Inside the class (Inline functions):** Member functions defined directly inside the class body.
2. **Outside the class (Using scope resolution):** Member functions defined outside the class using the scope resolution operator ::.

Example (Member Function Defined Inside Class):

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

```
class Rectangle {
public:
    int length, width;
    void setDimensions(int l, int w) {
        length = l;
        width = w;
    }
    int calculateArea() {
        return length * width;
    }
};
int main() {
    Rectangle rect;
    rect.setDimensions(5, 10);
    cout << "Area: " << rect.calculateArea() << endl;
    return 0;
}
```

Explanation:

- The setDimensions() function sets the values for length and width attributes, while calculateArea() computes the area.
- Both member functions are defined inside the class itself, making them **inline functions**.

Example (Member Function Defined Outside Class):

```
#include <iostream>
using namespace std;
class Circle {
private:
    double radius;
public:
    void setRadius(double r);
    double calculateArea();
};
void Circle::setRadius(double r) {
    radius = r;
}
double Circle::calculateArea() {
    return 3.14 * radius * radius;
}
int main() {
    Circle circle;
    circle.setRadius(7);
    cout << "Area of the circle: " << circle.calculateArea() << endl;
    return 0;
}
```

Explanation:

- The setRadius() and calculateArea() functions are defined **outside** the class using the scope resolution operator ::.
- This is useful when you want to keep the class definition clean, especially for large functions.

---

3. Object as Function Arguments:

In C++, you can pass objects as arguments to functions in two ways:

1. **By Value:** The object is copied, and any changes made inside the function do not affect the original object.
2. **By Reference:** The original object is passed, allowing the function to modify its contents.

Example of Passing Objects by Value:

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

class Book {
public:
    string title;
    int pages;
    void setDetails(string t, int p) {
        title = t;
        pages = p;
    }
    void display() {
        cout << "Title: " << title << ", Pages: " << pages << endl;
    }
};
void printBook(Book b) {  // Object passed by value
    b.title = "New Title";
    b.display();          // Changes do not affect the original object
}
int main() {
    Book book1;
```

```
    book1.setDetails("C++ Programming", 350);
    printBook(book1);      // Passing by value
    book1.display();       // Original object remains unchanged
    return 0;
}
```

Explanation:

- In printBook(), the Book object is passed **by value**. The function receives a copy of the object, so changes made inside printBook() do not affect the original book1 object.

Example of Passing Objects by Reference:

```
#include <iostream>

using namespace std;


class Book {

public:

    string title;

    int pages;

    void setDetails(string t, int p) {

        title = t;

        pages = p;

    }

    void display() {

        cout << "Title: " << title << ", Pages: " << pages << endl;

    }

};
```

```
void modifyBook(Book &b) {  // Object passed by reference
    b.title = "Advanced C++";
}
int main() {
    Book book1;
    book1.setDetails("C++ Programming", 350);
    modifyBook(book1);      // Passing by reference
    book1.display();        // The original object is modified
    return 0;
}
```

Explanation:

- In modifyBook(), the Book object is passed **by reference**. Any changes made inside the function affect the original book1 object, as demonstrated by the change in the title.

---

4. Object as Return Type:

A function can also return an object. This is useful when we want to return a modified object or create new objects within a function.

Example of Object as Return Type:

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

```
class Complex {
private:
    int real, imag;

public:
    void setValues(int r, int i) {
        real = r;
        imag = i;
    }

    void display() {
        cout << "Complex number: " << real << " + " << imag << "i" << endl;
    }

    Complex add(Complex c) {
        Complex result;
        result.real = real + c.real;
        result.imag = imag + c.imag;
        return result;
    }
};
```

```
int main() {
    Complex c1, c2, sum;
    c1.setValues(3, 4);
    c2.setValues(5, 6);

    sum = c1.add(c2);  // Returning an object from the function
    sum.display();     // Displaying the result
    return 0;
}
```

Explanation:

- The add() function returns a new Complex object that represents the sum of two complex numbers.
- This demonstrates how functions can create and return new objects.

---

5. Conclusion:

- **Member Access:** Objects access class members via the dot operator or member functions.
- **Defining Member Functions:** Functions can be defined either inside or outside the class.
- **Objects as Function Arguments:** Objects can be passed by value (copies the object) or by reference (modifies the original object).
- **Objects as Return Type:** Functions can return objects, enabling more flexible manipulation of class instances.

Al-shatrah University

كليـــة الهندسة

قسم هندسة الحاسوب

Subject: Object Oriented Programming (OOP)

First Stage

Lecturer: Dr. Azhaar a.hussan

Lecture (7)

Objects and Member Functions in Object-Oriented Programming

Objects and Member Access:

Objects are instances of a class, and through them, we can access the class's members (both attributes and functions). The two key concepts related to member access are:

1. **Public Members:** Accessible from outside the class.
2. **Private Members:** Accessible only from within the class.

When using objects, we access class members using the dot operator (.) for direct member access, or through member functions which are responsible for interacting with private data.

Example:

```
#include <iostream>
using namespace std;
class Student {
public:
    string name;
    int age;
    void displayInfo() {
        cout << "Name: " << name << ", Age: " << age << endl;
    }
};

int main() {
```

```
    Student student1;
    student1.name = "Alice";
    student1.age = 20;
    student1.displayInfo(); // Accessing member function
    return 0;
}
```

Explanation:

- We define a Student class with two public members (name and age).
- The displayInfo() function is a member function that prints the values of the name and age attributes.
- In main(), we create a Student object (student1) and access its members using the dot operator. We call displayInfo() to display the student's information.

---

2. Defining Member Functions:

There are two ways to define member functions in C++:

1. **Inside the class (Inline functions):** Member functions defined directly inside the class body.
2. **Outside the class (Using scope resolution):** Member functions defined outside the class using the scope resolution operator ::.

Example (Member Function Defined Inside Class):

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

```
class Rectangle {
public:
    int length, width;
    void setDimensions(int l, int w) {
        length = l;
        width = w;
    }
    int calculateArea() {
        return length * width;
    }
};
int main() {
    Rectangle rect;
    rect.setDimensions(5, 10);
    cout << "Area: " << rect.calculateArea() << endl;
    return 0;
}
```

Explanation:

- The setDimensions() function sets the values for length and width attributes, while calculateArea() computes the area.
- Both member functions are defined inside the class itself, making them **inline functions**.

Example (Member Function Defined Outside Class):

```
#include <iostream>
using namespace std;
class Circle {
private:
    double radius;
public:
    void setRadius(double r);
    double calculateArea();
};
void Circle::setRadius(double r) {
    radius = r;
}
double Circle::calculateArea() {
    return 3.14 * radius * radius;
}
int main() {
    Circle circle;
    circle.setRadius(7);
    cout << "Area of the circle: " << circle.calculateArea() << endl;
    return 0;
}
```

Explanation:

- The setRadius() and calculateArea() functions are defined **outside** the class using the scope resolution operator ::.
- This is useful when you want to keep the class definition clean, especially for large functions.

---

3. Object as Function Arguments:

In C++, you can pass objects as arguments to functions in two ways:

1. **By Value:** The object is copied, and any changes made inside the function do not affect the original object.
2. **By Reference:** The original object is passed, allowing the function to modify its contents.

Example of Passing Objects by Value:

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

class Book {
public:
    string title;
    int pages;
    void setDetails(string t, int p) {
        title = t;
        pages = p;
    }
    void display() {
        cout << "Title: " << title << ", Pages: " << pages << endl;
    }
};
void printBook(Book b) {  // Object passed by value
    b.title = "New Title";
    b.display();          // Changes do not affect the original object
}
int main() {
    Book book1;
```

```
    book1.setDetails("C++ Programming", 350);
    printBook(book1);      // Passing by value
    book1.display();       // Original object remains unchanged
    return 0;
}
```

Explanation:

- In printBook(), the Book object is passed **by value**. The function receives a copy of the object, so changes made inside printBook() do not affect the original book1 object.

Example of Passing Objects by Reference:

```
#include <iostream>

using namespace std;


class Book {

public:

    string title;

    int pages;

    void setDetails(string t, int p) {

        title = t;

        pages = p;

    }

    void display() {

        cout << "Title: " << title << ", Pages: " << pages << endl;

    }

};
```

```
void modifyBook(Book &b) {  // Object passed by reference
    b.title = "Advanced C++";
}
int main() {
    Book book1;
    book1.setDetails("C++ Programming", 350);
    modifyBook(book1);      // Passing by reference
    book1.display();        // The original object is modified
    return 0;
}
```

Explanation:

- In modifyBook(), the Book object is passed **by reference**. Any changes made inside the function affect the original book1 object, as demonstrated by the change in the title.

---

4. Object as Return Type:

A function can also return an object. This is useful when we want to return a modified object or create new objects within a function.

Example of Object as Return Type:

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

```
class Complex {
private:
    int real, imag;

public:
    void setValues(int r, int i) {
        real = r;
        imag = i;
    }

    void display() {
        cout << "Complex number: " << real << " + " << imag << "i" << endl;
    }

    Complex add(Complex c) {
        Complex result;
        result.real = real + c.real;
        result.imag = imag + c.imag;
        return result;
    }
};
```

```
int main() {
    Complex c1, c2, sum;
    c1.setValues(3, 4);
    c2.setValues(5, 6);

    sum = c1.add(c2);  // Returning an object from the function
    sum.display();     // Displaying the result
    return 0;
}
```

Explanation:

- The add() function returns a new Complex object that represents the sum of two complex numbers.
- This demonstrates how functions can create and return new objects.

---

5. Conclusion:

- **Member Access:** Objects access class members via the dot operator or member functions.
- **Defining Member Functions:** Functions can be defined either inside or outside the class.
- **Objects as Function Arguments:** Objects can be passed by value (copies the object) or by reference (modifies the original object).
- **Objects as Return Type:** Functions can return objects, enabling more flexible manipulation of class instances.

كليـــة الهندسة
قسم هندسة الحاسوب

Subject: Object Oriented Programming (OOP)

Second Stage

Lecturer: Dr. Azhaar A. hussan

Lecture (8)

Scope Operator Resolution, Member Initialization List, Constant Members and Static Members

1. Scope Operator Resolution (::)

The scope resolution operator :: in C++ is used to define or access a global variable when there is a local variable with the same name or to access members of a namespace or class.

Example:

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

int var = 100;          // متغير عام عالمي

class MyClass {
public:
    int var;             // متغير خاص بالكائن

    MyClass(int value) {
        var = value;          // تهيئة المتغير الخاص بالكائن
    }

    void display() {
      int var = 10;                    // متغير محلي داخل الدالة
      cout << "Local var: " << var << endl;     // يطبع المتغير المحلي
      cout << "Object's var: " << MyClass::var << endl; // يطبع متغير الكائن الحالي
      cout << "Global var: " << ::var << endl;        // يطبع المتغير العام
    }
};

int main() {
    MyClass obj(20);   // إنشاء كائن وتهيئة المتغير الخاص به إلى ٢٠
    obj.display();

    return 0;
}
```

2. Member Initialization List

In C++, a member initialization list allows initializing class members directly before the constructor's body. This is particularly useful for `const` members and references.

Example:

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

class Student {
private:
    const int id;   // const member
    string name;
public:
    // Member Initialization List
    Student(int id, string name) : id(id), name(name) {
        // Constructor body can be empty for initialization
    }
```

```
    void display() {
        cout << "ID: " << id << ", Name: " << name << endl;
    }
};

int main() {
    Student student1(101, "Alice");
    student1.display();
    return 0;
}
```

Explanation:

- The `id` member is `const` and can only be initialized in a member initialization list. Without this syntax, initializing `const` or reference members would result in a compilation error.

3. Constant Members

Constant members in C++ are members that cannot be modified once they are initialized. This concept extends to function arguments and member functions.

- **Constant Function Argument**: Declares an argument as `const` to prevent it from being modified inside the function.
- **Constant Member Function**: Declares a member function as `const`, ensuring it does not modify any class members.

Example:

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

class Rectangle {
private:
    int width, height;
public:
    Rectangle(int w, int h) : width(w), height(h) {}

    // Constant member function
    int area() const {
        // width++; // This line would cause a compilation error
        return width * height;
    }

    // Function with const argument
    void display(const string &prefix) const {
        cout << prefix << " Area: " << area() << endl;
    }
};

int main() {
    Rectangle rect(5, 3);
    rect.display("Rectangle");
```

```
    return 0;
}
```

Explanation:

- `area()` is a constant member function, meaning it cannot modify the class members `width` and `height`.
- `display()` takes a constant string reference as an argument, which prevents modification of the input string.

4. Static Members

Static members belong to the class rather than any particular object. They retain their values across all instances of the class, and there is only one copy of each static member.

Example:

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

class Counter {
public:
    static int count=0; // Static member variable

    Counter() {
        count++;
    }

    static void showCount() {
        cout << "Count: " << count << endl;
    }
};

// Initialize static member
int Counter::count = 0;

int main() {
    Counter c1, c2, c3;
    Counter::showCount(); // Accessing static member function
    return 0;
}
```

Explanation:

- `count` is a static member variable shared among all objects of `Counter`.
- Each time a `Counter` object is created, `count` is incremented.
- The `showCount()` function is a static member function that can access only static members and can be called without an object instance.

Summary

In this lecture, we covered advanced C++ concepts that are critical for efficient and organized code development:

1. **Scope Operator Resolution**: Used to access specific variables within scopes.
2. **Member Initialization List**: Allows efficient and necessary initialization of class members, particularly `const` and references.
3. **Constant Members**: Ensures data immutability within certain contexts, including const arguments and const functions.
4. **Static Members**: Allows shared variables and functions that are independent of individual objects.

These concepts are fundamental for writing clean, maintainable C++ code in both simple and complex applications.

كليـــة الهندسة
قسم هندسة الحاسوب

Subject: Object Oriented Programming (OOP)

First Stage

Lecturer: Dr. Azhaar A. hussan

Lecture (9)

Object Pointers and this Pointer in C++

Topic Overview

1. **Pointers to Objects**: How pointers can point to instances of classes and access member functions.

2. **`this` Pointer**: How to use the `this` pointer to refer to the current instance of a class.
3. **Reference Members**: Explanation of how to use references within classes.

---

1. Pointers to Objects

In C++, a pointer is a variable that holds the memory address of another variable, including objects. You can declare a pointer to an object, allowing you to dynamically allocate, access, and manage the object's members through the pointer.

Syntax:

```
ClassName *pointerName = &objectName;
```

Example 1: Writing an OOP program to read and display student information using an object pointer.

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

class Student {
private:
    int stageNumber;
    int age;
    char gender;
    float height;
    float weight;
public:
    void getInfo() {
        cout << "Enter stage number: "; cin >> stageNumber;
        cout << "Enter age: "; cin >> age;
        cout << "Enter gender (M/F): "; cin >> gender;
        cout << "Enter height (in cm): "; cin >> height;
        cout << "Enter weight (in kg): "; cin >> weight;
    }

    void displayInfo() const {
        cout << "Stage number: " << stageNumber << endl;
        cout << "Age: " << age << endl;
        cout << "Gender: " << gender << endl;
        cout << "Height: " << height << endl;
        cout << "Weight: " << weight << endl;
    }
};

int main() {
    Student *ptr = new Student;     // الطريقة الاولى لتعريف البوينتر للاوبجكت
    Student std;    // الطريقة الثانية لتعريف البوينتر للاوبجكت
    Student *ptr =std;
    cout << "Enter the following information:\n";
```

```
    ptr->getInfo();                 // Using pointer to access getInfo()
    cout << "\nStudent Information:\n";
    (*ptr).displayInfo();       // Using pointer to access displayInfo()
    delete ptr;                 // Free allocated memory
    return 0;
}
```

2. Using the `this` Pointer

The `this` pointer in C++ is an implicit pointer that points to the current object of a class. It is especially useful for differentiating between class members and function parameters that have the same name.

Example 1: Displaying the **memory address** of an object using `this`.

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

class Sample {
public:
    void showAddress() const {
        cout << "Object address: " << this << endl;
    }
};

int main() {
    Sample obj1, obj2, obj3;
    obj1.showAddress();
    obj2.showAddress();
    obj3.showAddress();
    return 0;
}
```

Explanation:

- Each object (`obj1`, `obj2`, `obj3`) will output a unique memory address when `showAddress` is called.
- The `this` pointer is implicitly (ضمنياً) used to refer to the object that called the function.

Example 2: Using `this` to access members.

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

class Sample {
private:
    int value;

public:
    Sample(int value) {
        this->value = value; // Assigning parameter 'value' to member 'value'
    }
```

```
    void display() const {
        cout << "Value: " << this->value << endl;
    }
};

int main() {
    Sample obj(42);
    obj.display();                // Displays: Value: 42
    return 0;
}
```

Explanation:

- The `this` pointer is used to resolve ambiguity between the member variable `value` and the constructor parameter `value`.

3. Reference Members

In C++, reference members allow you to initialize class members that are references to other variables or objects. However, references must be initialized when declared and cannot be changed to refer to different objects.

Example: Using a reference member in a class.

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

class Wrapper {
private:
    int &ref;                        // Reference member

public:
    Wrapper(int &val) : ref(val) {} // Initialize reference member in constructor

    void showValue() const {
        cout << "Referenced value: " << ref << endl;
    }

    void updateValue(int newVal) {
        ref = newVal; // Update the original variable referenced by 'ref'
    }
};

int main() {
    int num = 10;
    Wrapper wrapper(num);

    wrapper.showValue();      // Displays: Referenced value: 10

    wrapper.updateValue(20);
    cout << "Updated original value: " << num << endl; // Displays: Updated original value: 20
```

```
    return 0;
}
```

Explanation:

- The reference `ref` in the `Wrapper` class directly refers to `num` in `main`. Changes to `ref` will directly impact `num`, demonstrating how reference members work in classes.

4. Class Object Members

In C++, a data member of a class can be an object of another class. This is known as **class composition**. It allows a class to contain objects of other classes as members, facilitating a "has-a" relationship between classes.

Example 2: Summing the Coordinates of Points Using a Class Object Member

Here, we have two classes: `Point` and `PointSummation`. The `PointSummation` class contains two `Point` objects as members and calculates the sum of their coordinates.

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

// Class representing a Point with x and y coordinates
class Point {
private:
    int x, y;

public:
    Point(int xCoord, int yCoord) : x(xCoord), y(yCoord) {}

    int getX() const { return x; }
    int getY() const { return y; }
};

// Class that uses Point objects to sum their coordinates
class PointSummation {
private:
    Point point1, point2; // Two Point objects as members

public:
    PointSummation(int x1, int y1, int x2, int y2)
        : point1(x1, y1), point2(x2, y2) {}
// Initializing Point objects in constructor

    void displaySummation() const {
        int sumX = point1.getX() + point2.getX();
        int sumY = point1.getY() + point2.getY();
        cout << "Sum of X coordinates: " << sumX << endl;
        cout << "Sum of Y coordinates: " << sumY << endl;
    }
};
```

```
int main() {
    int x1, y1, x2, y2;
    cout << "Enter coordinates for point 1 (x y): ";
    cin >> x1 >> y1;
    cout << "Enter coordinates for point 2 (x y): ";
    cin >> x2 >> y2;

    PointSummation summation(x1, y1, x2, y2);
    summation.displaySummation();

    return 0;
}
```

Explanation:

- The `Point` class holds x and y coordinates.
- The `PointSummation` class contains two `Point` objects as members.
- The `displaySummation` function in `PointSummation` calculates and displays the sum of the x and y coordinates of the two `Point` objects.

كليـــة الهندسة
قسم هندسة الحاسوب

Subject: Object Oriented Programming (OOP)

First Stage

Lecturer: Dr. Aazhaar A. hussan

Lecture (10)

Arrays as Class Data Members

In C++, arrays can be used as data members in a class. They allow you to store multiple values or objects of the same type within a single class. This is particularly useful when

handling multiple elements that belong to the same category or when performing operations on a group of objects.

This lecture covers:

1. **Arrays as Class Data Members**
2. **Object Arrays**
3. **An Array of Pointers to Objects**

---

1. Arrays as Class Data Members

When an array is used as a data member in a class, it can store multiple values related to that class. Arrays can be of basic data types (like `int` or `float`) or user-defined types (like objects of a class). This approach allows encapsulation of multiple values or objects within a single class instance.

Example: Storing Marks of Multiple Subjects

In the following example, a class `Student` has an array `marks` as a data member to store the scores of multiple subjects for a single student.

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

class Student {
private:
    int marks[5];        // Array to store marks of 5 subjects

public:
    void setMarks(int m[]) {
        for (int i = 0; i < 5; ++i) {
            marks[i] = m[i];
        }
    }

    void displayMarks() const {
        cout << "Marks: ";
        for (int i = 0; i < 5; ++i) {
            cout << marks[i] << " ";
        }
        cout << endl;
    }
};

int main() {
    Student stu1;
    int subjectMarks[5] = {90, 85, 76, 88, 92};
```

```
    stu1.setMarks(subjectMarks);
    stu1.displayMarks();

    return 0;
}
```

Explanation:

- The `Student` class has an array `marks` to store scores of 5 subjects.
- The `setMarks` function accepts an array as input and assigns it to the `marks` array.
- The `displayMarks` function outputs the marks stored in the array.

2. Object Arrays

An **Object Array** is an array where each element is an object of a particular class. This is useful when dealing with multiple instances of a class.

Example: Managing Multiple Employees Using an Object Array

In this example, the `Company` class has an array of `Employee` objects as a data member. Each `Employee` object contains information about individual employees.

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

class Employee {
private:
    string name;
    int id;
public:
    void setData(string n, int i) {
        name = n;
        id = i;
    }
    void display() const {
        cout << "Employee ID: " << id << ", Name: " << name << endl;
    }
};

class Company {
private:
    Employee employees[3]; // Array of Employee objects
public:
    void setEmployeeData() {
        string name;
        int id;
        for (int i = 0; i < 3; ++i) {
          cout << "Enter ID and name for employee " << (i + 1) << ": ";
          cin >> id >> name;
          employees[i].setData(name, id);
        }
```

```
    }

    void displayEmployees() const {
        cout << "Company Employees:" << endl;
        for (int i = 0; i < 3; ++i) {
            employees[i].display();
        }
    }
};

int main() {
    Company company;

    company.setEmployeeData();
    company.displayEmployees();

    return 0;
}
```

Explanation:

- `Company` class contains an array of `Employee` objects.
- The `setEmployeeData` method allows input of employee details for each object in the array.
- The `displayEmployees` method outputs the details of each employee in the array.

3. An Array of Pointers to Objects

An **Array of Pointers to Objects** is an array where each element is a pointer to an object. This allows dynamic allocation and more flexibility, as you can decide when to create or delete objects.

Example: Managing Library Books Using an Array of Pointers to Objects

In this example, the `Library` class has an array of pointers to `Book` objects. This allows creating books only as needed.

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

class Book {
private:
    string title;
    string author;

public:
    Book(string t, string a) : title(t), author(a) {}

    void display() const {
        cout << "Title: " << title << ", Author: " << author << endl;
    }
};
```

```
class Library {
private:
    Book* books[5]; // Array of pointers to Book objects
    int count;

public:
    Library() : count(0) {}

    void addBook(string title, string author) {
        if (count < 5) {
            books[count] = new Book(title, author); // Dynamically
allocate a new Book
            ++count;
        } else {
            cout << "Library is full." << endl;
        }
    }

    void displayBooks() const {
        cout << "Library Books:" << endl;
        for (int i = 0; i < count; ++i) {
            books[i]->display();
        }
    }

    ~Library() {
        for (int i = 0; i < count; ++i) {
            delete books[i]; // Free allocated memory
        }
    }
};

int main() {
    Library library;

    library.addBook("1984", "George Orwell");
    library.addBook("To Kill a Mockingbird", "Harper Lee");

    library.displayBooks();

    return 0;
}
```

Explanation:

- `Library` has an array of pointers to `Book` objects.
- The `addBook` method dynamically allocates a new `Book` object and stores its pointer in the array.
- The `displayBooks` method displays details of each book.
- In the destructor of `Library`, we use `delete` to free the allocated memory, preventing memory leaks.

Summary

- **Arrays as Class Data Members**: Arrays can store multiple values of basic or user-defined types in a single class.
- **Object Arrays**: An array of objects within a class allows handling multiple instances of that class type.
- **Array of Pointers to Objects**: This approach allows dynamic memory allocation, giving flexibility in creating and deleting objects as needed.

كليـــة الهندسة
قسم هندسة الحاسوب

Subject: Object Oriented Programming (OOP)

First Stage

Lecturer: Dr. Azhaar A. hussan

Lecture (11)

Operator Overloading

This lecture will focus on the concept of operator overloading in C++. By the end of this lecture, you will understand how to redefine standard operators such as **+, -, and ==** to work with user-defined types like classes and objects.

1. Introduction to Operator Overloading

Operator overloading allows you to redefine the way operators work for user-defined types. For instance, you can define how the **+ operator** adds two objects of a class or how the == operator compares them.

- **Why is it useful?**
  - To enhance readability and expressiveness in code.
  - To enable object-oriented behavior for standard operators.

---

2. Overloading Unary Operators

Unary operators are those that operate on a single operand, such as **++, --, or -.**

Steps to Overload Unary Operators

1. Define the operator as a member function or a friend function.
2. Ensure the operator takes no arguments.
3. Use the `operator` keyword followed by the operator symbol.

Example: Overloading the - Operator

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

class Number {
    int value;

public:
    Number(int v) : value(v) {}
    // Overloading the unary minus (-) operator
    Number operator-() {
        return Number(-value);
    }
    void display() {
        cout << "Value: " << value << endl;
    }
};
int main() {
    Number n(5);
    Number neg = -n; // Calls overloaded operator-
    neg.display();   // Output: Value: -5
    return 0;
}
```

3. Operator Arguments, Return Values, and Postfix Notation

Operator Arguments

Some operators, like binary operators, require arguments to function. For example, the + operator will need two operands.

Return Values

An overloaded operator can return:

- A modified object.
- A value of a primitive type (e.g., `int`, `bool`).

Prefix vs. Postfix Notation

For postfix operators like `x++`, the compiler uses an additional `int` parameter to distinguish it from prefix `++x`. The *int* inside the parentheses does not represent an integer value being passed to the function. Instead, it is a **dummy argument** used to distinguish the postfix increment operator from the prefix increment operator which takes no arguments. The presence of this *int* tells the compiler that this is the postfix version of the ++ operator.

Example: Overloading the Prefix vs. Postfix

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

class Counter {
private:
    int count; // Counter value
public:
    // Constructor to initialize counter
    Counter(int c) : count(c) {}
    // Prefix increment (++obj)
    Counter operator++() {
        return Counter(++count);     // Increment first, then return
    }
    // Postfix increment (obj++)
    Counter operator++(int) {
        return Counter(count++);     // Return current value, then increment
    }
    // Function to print the counter value
    void print() const {
        cout << count << endl;
    }
};
int main() {
    Counter c1(1), c2(1); // Initialize two Counter objects with value 1
                          // Demonstrate prefix increment
    c2 = ++c1;   // c1=2, c2=2 (increment c1 first, then assign to c2)
    cout << "After prefix increment:" << endl;
    cout << "c1: ";
    c1.print();         // c1 = 2
    cout << "c2: ";
    c2.print();         // c2 = 2

    // Demonstrate postfix increment
    c2 = c1++; // c1=3, c2=2 (assign c1 to c2, then increment c1)
    cout << "After postfix increment:" << endl;
    cout << "c1: ";
    c1.print();         // c1 = 3
    cout << "c2: ";
    c2.print();         // c2 = 2
    return 0;
}
```

The output:

After prefix increment:
c1: 2
c2: 2
After postfix increment:

c1: 3
c2: 2

4. Overloading Binary Operators

Binary operators operate on two operands. Examples include **+, -, *, /, and comparison operators like == or <.**

Steps to Overload Binary Operators

1. Define the operator as a member function or a friend function.
2. Pass the second operand as an argument.
3. Return the result of the operation.

Example: Overloading the + Operator

```
#include <iostream>
using namespace std;
class Complex {
    float real, imag;
public:
    Complex(float r, float i) : real(r), imag(i) {}

    // Overloading the + operator
    Complex operator+(const Complex& other) {
        return Complex(real + other.real, imag + other.imag);
    }
    void display() {
        cout << real << " + " << imag << "i" << endl;
    }
};
int main() {
    Complex c1(2.5, 3.5), c2(1.5, 4.5);
    Complex sum = c1 + c2;              // Calls overloaded operator+
    sum.display();                      // Output: 4.0 + 8.0i
    return 0;
}
```

5. Overloading Arithmetic Operators

Arithmetic operators include +, -, *, /, etc. These can be overloaded to perform operations on objects of a class.

Example: Overloading the * Operator for a Matrix Class

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

class Matrix {
    int value;
public:
    Matrix(int v) : value(v) {}

    Matrix operator*(const Matrix& other) {
        return Matrix(value * other.value);
    }
    void display() {
        cout << "Result: " << value << endl;
    }
};
```

```
int main() {
    Matrix m1(3), m2(4);
    Matrix result = m1 * m2; // Calls overloaded operator*
    result.display();        // Output: Result: 12
    return 0;
}
```

6. Overloading Comparison Operators

Comparison operators like ==, <, and > can also be overloaded to compare objects based on their data members.

Example: Overloading the == Operator

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

class Point {
    int x, y;
public:
    Point(int x, int y) : x(x), y(y) {}

    bool operator==(const Point& other) {
        return (x == other.x) && (y == other.y);
    }
};
int main() {
    Point p1(2, 3), p2(2, 3);
    if (p1 == p2) { // Calls overloaded operator==
        cout << "Points are equal" << endl;
    } else {
        cout << "Points are not equal" << endl;
    }
    return 0;
}
```

Unary Operators: Unary operators operate on a single operand.

Operator	Purpose	Example
+	Unary plus (typically redundant, but can be used for marking positive values explicitly).	+a
-	Unary minus (used to negate a value).	-a
*	Dereference operator (used to access the value at a memory address).	*ptr
!	Logical NOT (inverts a boolean value).	!a
~	Bitwise NOT (inverts all bits of a number).	~a
&	Address-of operator (returns the memory address of a variable).	&a
++	Increment operator (adds 1 to a variable).	++a (prefix) or a++ (postfix)
--	Decrement operator (subtracts 1 from a variable).	--a (prefix) or a-- (postfix)
()	Function call operator (can be overloaded to make an object callable like a function).	object()
->	Member access operator (used to access class members via a pointer).	ptr->member
->*	Pointer-to-member operator (used to access members of a class via a pointer to member).	ptr->*memberPointer
new	Memory allocation operator (allocates memory dynamically).	int *p = new int;
delete	Memory deallocation operator (frees dynamically allocated memory).	delete p;

Binary Operators: Binary operators operate on two operands.

Operator	Purpose	Example
+	Addition operator.	a + b

Operator	Purpose	Example
-	Subtraction operator.	`a - b`
*	Multiplication operator.	`a * b`
/	Division operator.	`a / b`
%	Modulus operator (remainder of division).	`a % b`
&	Bitwise AND operator.	`a & b`
`	`	Bitwise OR operator.
^	Bitwise XOR operator.	`a ^ b`
<<	Left shift operator (shifts bits to the left).	`a << b`
>>	Right shift operator (shifts bits to the right).	`a >> b`
+=	Add and assign operator.	`a += b` (equivalent to `a = a + b`)
-=	Subtract and assign operator.	`a -= b` (equivalent to `a = a - b`)
/=	Divide and assign operator.	`a /= b` (equivalent to `a = a / b`)
%=	Modulus and assign operator.	`a %= b` (equivalent to `a = a % b`)
&=	Bitwise AND and assign operator.	`a &= b` (equivalent to `a = a & b`)
^=	Bitwise XOR and assign operator.	`a ^= b` (equivalent to `a = a ^ b`)
<<=	Left shift and assign operator.	`a <<= b` (equivalent to `a = a << b`)
>>=	Right shift and assign operator.	`a >>= b` (equivalent to `a = a >> b`)
==	Equality comparison operator.	`a == b` (true if `a` equals `b`)
!=	Inequality comparison operator.	`a != b` (true if `a` is not equal to `b`)
<	Less-than operator.	`a < b`
>	Greater-than operator.	`a > b`
<=	Less-than-or-equal-to operator.	`a <= b`
>=	Greater-than-or-equal-to operator.	`a >= b`
&&	Logical AND operator.	`a && b` (true if both `a` and `b` are true)
[]	Subscript operator (used for accessing array elements or containers).	`array[index]`
()	Function call operator (allows an object to be treated as a function).	`object(params)`

Operators That Cannot Be Overloaded

Operator	Description
.	Dot operator
.* (or ->)	Access member operator
::	Scope resolution operator
?:	Conditional operator (ternary)
sizeof	Size of file or type operator

7. Summary of Key Points

- Operator overloading enhances the usability of custom classes.
- Unary operators operate on a single object; binary operators require two operands.
- Always maintain the operator's expected behavior to avoid confusion.
- Operator overloading is commonly used for mathematical classes, string manipulation, and comparisons.

Assignment for Students

Q: Overload the == operator for a `Student` class to compare two students based on their IDs.

Hint:

- **Define the Student Class:**

Create a class named `Student` with the following attributes:

  - `id`: A unique identifier for the student.
  - Any additional attributes like `name`, `age`, etc. (optional, for context).

- **Overload the == Operator:**

Implement the `operator==` function for the `Student` class. The purpose of this operator is to compare the `id` attribute of two `Student` objects and return `true` if their IDs are equal and `false` otherwise.

- **Write a Main Function:**
  - Create multiple objects of the `Student` class.
  - Use the `==` operator to compare these objects.
  - Print the results to verify that the overloaded operator works as intended