Basics and Concepts of CPP

iamneo

CPP Programming Language

C++ is a cross-platform language that can be used to create high-performance application.

- C++ was developed by Bjarne Stroustrup, as an extension to the C language
- C++ is often viewed as a superset of C
- C++ is also known as a "C with class"
- C++ gives programmers a high level of control over system resources and memory

iamneo

Why CPP?

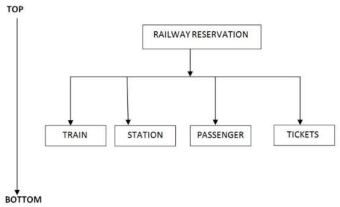
- C++ is an object-oriented programming language which gives a clear structure to programs and allows code to be reused, lowering development costs.
- C++ is portable and can be used to develop applications that can be adapted to multiple platforms.

iamneo

Difference between C and CPP:

C Programming Language	CPP Programming Language
C is (mostly) a subset of C++.	C++ is (mostly) a superset of C.
C supports procedural programming.	C++ supports both procedural and object-oriented programming paradigms.
C does not support information hiding.	Data is hidden by the Encapsulation
C is a function driven language.	C++ is an object driven language.
C follows the top-down approach.	C++ follows the Bottom-up approach.

iamneo



Differences between procedural and object-oriented programming

Procedural Oriented Programming	Object Oriented Programming
The program is divided into small parts called functions.	The program is divided into small parts called objects.
No access specifier in this procedural oriented.	Has access specifiers like private, public and protected.
Adding new data and functions is not easy.	Adding new data and function is easy.
It does not have any proper way of hiding data. so it is less secure.	It provides data hiding so it is more secure.
Procedural programming is based on the unreal world.	Object-oriented programming is based on the real world.
Code reusability absent in procedural programming.	Code reusability present in object-oriented programming.
Examples: C, FORTRAN, Pascal, Basic, etc.	Examples: C++, Java, Python, C#, etc.

Applications of CPP

C++ is used in developing browsers, operating systems, and applications, as well as in-game programming, embedded systems, banking applications, software engineering, compilers and so.

Advantages of C++ Over Other Languages:

- Object-oriented
- Speed compared to other general-purpose language
- Rich library support
- Pointer support
- Support low level system

Reading and Writing data using cin and cout

CPP Programming Language

C++ I/O operation is using the stream concept. Stream is the sequence of bytes or flow of data. It makes the performance fast.

- If bytes flow from main memory to device like printer, display screen, or a network connection, etc., this is called as output operation.
- If bytes flow from device like printer, display screen, or a network connection, etc., to main memory, this is called as input operation.

I/O Library Header Files

Header File	Function description
<iostream>	It is used to define the **cout**, **cin** and **cerr** objects, which correspond to standard output stream, standard input stream and standard error stream, respectively.
<iomanip>	It is used to declare services useful for performing formatted I/O, such as **setprecision** and **setw**.

Standard Output Stream (cout)

- The **cout** is a predefined object of ostream class. It is connected with the standard output device, which is usually a display screen.
- The cout is used in conjunction with stream insertion operator (<<) to display the output on a console

Standard Output Stream (cout)

- Let's see the simple example of standard output stream (cout)

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

int main() {
    cout << "Hello World!";
    return 0;
}
```

Output

```
Hello World!
```

iamneo

Standard Input Stream (cin)

- The cin is a predefined object of istream class. It is connected with the standard input device, which is usually a keyboard.
- The cin is used in conjunction with stream extraction operator (>>) to read the input from a console.

iamneo

Standard Input Stream (cin)

- Let's see the simple example of standard input stream (cin):

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

int main() {
    int height;
    cin >> height;
    cout << "Your height is: " << height;
    return 0;
}
```

Input

```
170
```

Output

```
Your height is: 170
```

iamneo

"using namespace std" in C++ program

- **Need of namespace:** As the same name can't be given to multiple variables, functions, classes, etc. in the same scope. So, to overcome this situation namespace is introduced.
- It is known that "**std**" (abbreviation for the standard) is a namespace whose members are used in the program.

iamneo

"using namespace std" in C++ program

- So, the members of the "**std**" namespace are cout, cin, endl, etc.
- This namespace is present in the iostream.h header file.

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

int main() {
    cout << "Hello World!";
    return 0;
}
```

iamneo

Omitting Namespace

- Some C++ programs run without the standard namespace library. The using namespace stdline can be omitted and replaced with the std keyword, followed by the :: operator for some objects:

```
#include <iostream>

int main() {
    std::cout << "Hello World!";
    return 0;
}
```

iamneo

Omitting Namespace

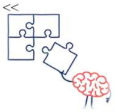
- Some C++ programs run without the standard namespace library. The using namespace stdline can be omitted and replaced with the std keyword, followed by the :: operator for some objects:

```
#include <iostream>

int main() {
    std::cout << "Hello World!";
    return 0;
}
```

iamneo

Recall

- Input and output are the basic effect and cause of programming
- Library functions define the input and output streams
- Basic input stream will be cin followed by >>
- Basic output stream will be cout followed by <<



iamneo

Data Types

iamneo

C++ Data Types

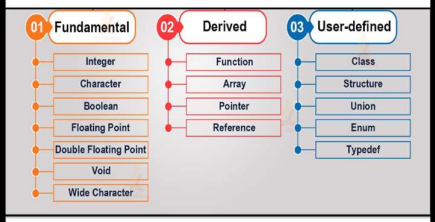
- Data types are used to tell the variables the type of data they can store.
- Whenever a variable is defined in C++, the compiler allocates some memory for that variable based on the data type with which it is declared.
- Every data type requires a different amount of memory.

C++ supports the following data types:

- Primary, built-in, or fundamental data type
- Derived data types
- User-defined data types

iamneo

C++ Data types



iamneo

C++ Data types

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

int main() {
    cout << "Size of char : " << sizeof(char) << endl;
    cout << "Size of int : " << sizeof(int) << endl;
    cout << "Size of long : " << sizeof(long) << endl;
    cout << "Size of float : " << sizeof(float) << endl;
    cout << "Size of double : " << sizeof(double) << endl;
    return 0;
}
```

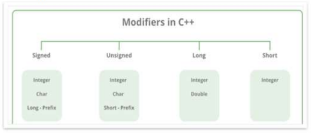
Output

```
Size of char : 1
Size of int : 4
Size of long : 8
Size of float : 4
Size of double : 8
```

iamneo

Datatype Modifiers in C++

- In C++, data type modifiers are used to modify the behavior and storage characteristics of basic data types.
- In C++, there are four modifiers. int, double, and char are the data types that can be modified using these modifiers. They are as follows:



iamneo

Datatype Modifiers in C++

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

int main() {
  cout << "unsigned int : " << sizeof(unsigned int) << endl;
  cout << "short int : " << sizeof(short int) << endl;
  cout << "long int : " << sizeof(long int) << endl;
  cout << "unsigned short int : " << sizeof(unsigned short int) << endl;
  cout << "unsigned long int : " << sizeof(unsigned long int) << endl;
  cout << "long long int : " << sizeof(long long int) << endl;
  cout << "unsigned long long int : " << sizeof(unsigned long long int) << endl;
  cout << "signed char : " << sizeof(signed char) << endl;
  cout << "unsigned char : " << sizeof(unsigned char) << endl;
  return 0;
}
```

Output

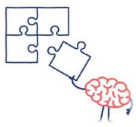
```
unsigned int : 4
short int : 2
long int : 8
unsigned short int : 2
unsigned long int : 8
long long int : 8
unsigned long long int : 8
signed char : 1
unsigned char : 1
```

iamneo

Data Type	Size (In Bytes)	Range
short int	2	-32,768 to 32,767
unsigned short int	2	0 to 65,535
unsigned int	4	0 to 4,294,967,295
int	4	-2,147,483,648 to 2,147,483,647
long int	4	-2,147,483,648 to 2,147,483,647
unsigned long int	4	0 to 4,294,967,295
long long int	8	-(2^63) to (2^63)-1
unsigned long long int	8	0 to 18,446,744,073,709,551,615
signed char	1	-128 to 127
unsigned char	1	0 to 255
float	4	
double	8	
long double	12	
wchar_t	2 to 4	1 wide character

Recall

- Datatypes
- Datatype Modifiers
- Examples



iamneo

A + B

Operators

iamneo

C++ Operators

Operators are used to perform operations on variables and values.

	Operator	Type
Unary Operator	+, --	Unary Operator
Binary Operator	+, -, *, /, %	Arithmetic Operator
	<, <=, >=, ==, !=	Relational Operator
	&&, ||, !	Logical Operator
	&, |, <<, >>, ~, ^	Bitwise Operator
	=, +=, -=, *=, /=, %=	Assignment Operator
Ternary Operator	?:	Ternary or Conditional Operator

iamneo

Unary Operators

C++ also provides increment and decrement operators: ++ and -- respectively.

- ++ increases the value of the operand by 1
- -- decreases it by 1

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

int main() {
    int var1 = 5, var2 = 5;
    cout << var1++ << endl;
    cout << ++var2 << endl;
    return 0;
}
```

Output

```
5
6
```

iamneo

Unary Operators

Let's see the simple example of pre and post decrement operator

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

int main() {
    int x = 10, y = 100, a, b;
    a = --x;
    cout << "a = " << a << endl;
    b = y--;
    cout << "b = " << b << endl;
    return 0;
}
```

Output

```
a = 9
b = 100
```

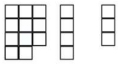
iamneo

Arithmetic Operators

- Arithmetic operators are used to perform common mathematical operations.

Operator	Function
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus (remainder)

Modulo operation



$11 \ mod \ 4 = 3$

iamneo

Arithmetic Operators

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

int main() {
    int a, b;
    a = 7;
    b = 2;
    cout << "a + b = " << (a + b) << endl;
    cout << "a - b = " << (a - b) << endl;
    cout << "a * b = " << (a * b) << endl;
    cout << "a / b = " << (a / b) << endl;
    cout << "a % b = " << (a % b) << endl;
    return 0;
}
```

Output

```
a + b = 9
a - b = 5
a * b = 14
a / b = 3
a % b = 1
```

iamneo

Relational Operator

- A relational operator is used to check the relationship between two operands.

Operator	Description
==	equal to
!=	not equal to
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to

iamneo

Relational Operator

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

int main() {
    int a, b;
    a = 3;
    b = 5;
    bool result;
    result = (a == b);
    cout << "3 == 5 is " << result << endl;
    result = (a != b);
    cout << "3 != 5 is " << result << endl;
    result = a > b;
    cout << "3 > 5 is " << result << endl;
    result = a < b;
    cout << "3 < 5 is " << result << endl;
    result = a >= b;
    cout << "3 >= 5 is " << result << endl;
    result = a <= b;
    cout << "3 <= 5 is " << result << endl;
    return 0;
}
```

Output

```
3 == 5 is 0
3 != 5 is 1
3 > 5 is 0
3 < 5 is 1
3 >= 5 is 0
3 <= 5 is 1
```

iamneo

Logical Operator

- Logical operators are used to check whether an expression is true or false. If the expression is true, it returns 1 whereas if the expression is false, it returns 0.

Operator	Meaning
&&	Logical and
||	Logical or
!	Logical not

iamneo

Logical Operator

INPUT A	OUTPUT (Y)
0	1
1	0

Logical NOT

INPUT A	INPUT B	OUTPUT (Y)
0	0	0
0	1	0
1	0	0
1	1	1

Logical AND

INPUT A	INPUT B	OUTPUT (Y)
0	0	0
0	1	1
1	0	1
1	1	1

Logical OR

iamneo

Logical Operator

- In C++, logical operators are commonly used in decision-making. To further understand the logical operators, let's see the following examples.

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

int main() {
    bool result;
    result = (3 != 5) && (3 < 5);
    cout << "(3 != 5) && (3 < 5) is " << result << endl;

    result = (3 == 5) && (3 < 5);
    cout << "(3 == 5) && (3 < 5) is " << result << endl;

    result = (3 == 5) && (3 > 5);
    cout << "(3 == 5) && (3 > 5) is " << result << endl;

    result = (3 != 5) || (3 < 5);
    cout << "(3 != 5) || (3 < 5) is " << result << endl;

    result = (3 != 5) || (3 > 5);
    cout << "(3 != 5) || (3 > 5) is " << result << endl;

    result = (3 == 5) || (3 > 5);
    cout << "(3 == 5) || (3 > 5) is " << result << endl;

    result = !(5 == 2);
    cout << "!(5 == 2) is " << result << endl;

    result = !(5 == 5);
    cout << "!(5 == 5) is " << result << endl;

    return 0;
}
```

Output

```
(3 != 5) && (3 < 5) is 1
(3 == 5) && (3 < 5) is 0
(3 == 5) && (3 > 5) is 0
(3 != 5) || (3 < 5) is 1
(3 != 5) || (3 > 5) is 1
(3 == 5) || (3 > 5) is 0
!(5 == 2) is 1
!(5 == 5) is 0
```

iamneo

Bitwise Operator

- In C++, bitwise operators are used to perform operations on individual bits. They can only be used alongside char and int data types.

Operator	Description
&	bitwise AND
|	bitwise OR
^	bitwise exclusive OR
<<	shift left
>>	shift right
~	one's complement

iamneo

Bitwise Operator

If a = 65, b = 15 , Equivalent binary values of 65 = 0100 0001; 15 = 0000 1111

Operator	Operation	Result
&	a & b	a: 0 1 0 0 0 0 0 1 b: 0 0 0 0 1 1 1 1 a & b: 0 0 0 0 0 0 0 1 (a&b) = 0000 0001$_2$= 1$_{10}$
|	a | b	a: 0 1 0 0 0 0 0 1 b: 0 0 0 0 1 1 1 1 a | b: 0 1 0 0 1 1 1 1 (a|b) = 01001111$_2$= 79$_{10}$
^	a ^ b	a: 0 1 0 0 0 0 0 1 b: 0 0 0 0 1 1 1 1 a ^ b: 0 1 0 0 1 1 1 0 (a^b) = 0100 1110$_2$= 78$_{10}$

iamneo

Bitwise Operator

If a = 15; the Equivalent binary value of a is 0000 1111

Operator	Operation	Result
<<	a << 3	a: 0 0 0 0 1 1 1 1 a << 3: 0 1 1 1 1 0 0 0 (a<< 3) = 01111000$_2$= 120$_{10}$
>>	a >> 2	a: 0 0 0 0 1 1 1 1 a>> 2: 0 0 0 0 1 1 0 0 (a>> 2) = 0000 0011$_2$= 3$_{10}$

Operator	Operation	Result
~	(~a)	a: 0 0 0 0 1 1 1 1 (~a): 1 1 1 1 0 0 0 0 (~a) = 1111 0000$_2$= -16$_{10}$

iamneo

Bitwise Operator

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

main() {
   unsigned int a = 50;
   unsigned int b = 12;
   int c = 0;

   c = a & b;
   cout << "c = " << c << endl ;
   c = a | b;
   cout << "c = " << c << endl ;
   c = a ^ b;
   cout << "c = " << c << endl ;
   c = ~a;
   cout << "c = " << c << endl ;
   c = a << 2;
   cout << "c = " << c << endl ;
   c = a >> 2;
   cout << "c = " << c << endl ;
   return 0;
}
```

Output

```
c = 0
c = 62
c = 62
c = -51
c = 200
c = 12
```

iamneo

Assignment Operator

- In C++, assignment operators are used to assign values to variables. For example, the expression C+=2 simply means that "C" is the variable, the "+=" is the assignment operator, and lastly the "2" is the value assigned to the variable.

Operator	Description
+=	Addition and assignment
-=	Subtraction and assignment
*=	Multiplication and assignment
/=	Division and assignment
%=	Remainder and assignment

iamneo

Assignment Operator

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

int main() {
    int a = 21;
    int c;

    c = a;
    cout << "c = " << c << endl;

    c += a;
    cout << "c = " << c << endl;

    c -= a;
    cout << "c = " << c << endl;

    c *= a;
    cout << "c = " << c << endl;

    c /= a;
    cout << "c = " << c << endl;

    c %= a;
    cout << "c = " << c << endl;

    return 0;
}
```

Output

```
c = 21
c = 42
c = 21
c = 441
c = 21
c = 0
```

iamneo

Ternary Operator

- A ternary operator evaluates the test condition and executes an expression out of two based on the result of the condition.

Syntax

```
condition ? expression1 : expression2;
```

Here, the *condition* is evaluated and

- if *condition* is true, *expression1* is executed.
- if *condition* is false, *expression2* is executed.

The ternary operator takes 3 operands (condition, expression1 and expression2). Hence, the name ternary operator.

iamneo

Ternary Operator

```cpp
#include <iostream>
using namespace std;
int main() {
    int a = 10;
    int b = 20;
    int max = a > b ? a : b;
    cout << "Maximum value = " << max;
    return 0;
}
```

Output

```
Maximum value = 20
```

iamneo

Other Operators in C++

The Comma operator	Comma (,) is an operator in C++ used to string together several expressions. The group of expression separated by comma is evaluated from left to right.
Sizeof	This is called as compile time operator. It returns the size of a variable in bytes.
Pointer	* Pointer to a variable & Address of
Component selection	. Direct component selector -> Indirect component selector
Class member operators	:: Scope access / resolution .* Dereference ->* Dereference pointer to class member

iamneo

Operator Precedence

- Precedence is the order in operators with different precedence are evaluated. In precedence, the operator with a higher precedence will be evaluated first.
- Then, the lower precedence operator is evaluated. To understand this better, let's look at the code snippet below.

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

int main() {
  cout << 8 + (6 * 2) / 2 - 10 ;
  return 0;
}
```

Output

```
4
```

iamneo

Operator Precedence

() [] . ->	
++ -- + - ! ~ (type) * & sizeof	**Unary Operator**
* / %	**Arithmetic Operator**
+ -	**Arithmetic Operator**
<< >>	**Shift Operator**
< <= > >=	**Relational Operator**
== !=	**Relational Operator**
&	**Bitwise AND Operator**
^	**Bitwise EX-OR Operator**
|	**Bitwise OR Operator**
&&	**Logical AND Operator**
||	**Logical OR Operator**
? :	**Ternary Conditional Operator**
= += -= *= /= %= &= ^= |= <<= >>=	**Assignment Operator**
,	**Comma**

iamneo

Associativity

- Associativity is the order in which operators with the same precedence are evaluated.
- For example, if we have an addition and subtraction expression, the compiler will evaluate from left to right since they both have the same precedence.

This can be done in two ways:

- Left to right
- Right to left

iamneo

Associativity

- Below is the list of operators in decreasing order of their precedence. The operators in the same row have the same precedence.

Category	Operators	Associativity
Multiplicative	*,/, %	Left to right
Additive	+, -	Left to right
AND	&	Left to right
OR	|	Left to right
Shift	<<, >>	Left to right
Equality	==, |=	Left to right
Conditional	?, :	Right to left
Assignment	+=, -=, *=, /=, >>=, <<=, &=, |=, ^=, ==	Right to left

iamneo

Example

- To understand this better, take a look at the code snippet below.

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

int main() {
  cout << 12*6/8*3*2/6;
  return 0;
}
```

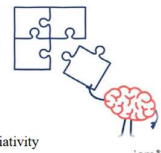
Output

9

iamneo

Recall

- Operators
- Unary Operators
- Arithmetic Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Assignment Operators
- Ternary Operators
- Other Operators in C++
- Operator Precedence and Associativity



iamneo

Conditional and Loops

iamneo

C++ Conditions and If Statements

C++ supports the usual logical conditions from mathematics:

- Less than: **a < b**
- Less than or equal to: **a <= b**
- Greater than: **a > b**
- Greater than or equal to: **a >= b**
- Equal to **a == b**
- Not Equal to: **a != b**

You can use these conditions to perform different actions for different decisions.

iamneo

C++ Conditions and If Statements

C++ has the following conditional statements:

- Use **if** to specify a block of code to be executed, if a specified condition is true.
- Use **else** to specify a block of code to be executed, if the same condition is false.
- Use **else if** to specify a new condition to test, if the first condition is false.
- Use **switch** to specify many alternative blocks of code to be executed.

iamneo

Example program – If & If-else

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

int main() {
    int marks = 60;

    if (marks > 40) {
        cout << "Passed";
    }
    cout << "\nHappy";

    return 0;
}
```

Output

```
Passed
Happy
```

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

int main() {
    int marks = 28;

    // Initiating if-else statement
    if (marks > 40)
        cout << "Student has passed";
    else
        cout << "Student has failed";

    return 0;
}
```

Output

```
Student has failed
```

iamneo

Example program - If-Else-If Ladder

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

int main() {
    int marks = 80;

    if (marks >= 90)
        cout << "Grade A";
    else if (marks >= 80)
        cout << "Grade B";
    else if (marks >= 60)
        cout << "Grade C";
    else if (marks >= 40)
        cout << "Grade D";
    else
        cout << "Grade F";

    return 0;
}
```

Output

```
Grade B
```

iamneo

Example program – Nested If

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

int main() {
    int n1 = 12;
    int n2 = 16;
    int n3 = 20;

    if (n1 > n2) {
        if (n1 > n3) {
            cout << "n1 is the largest number";
        } else {
            cout << "n3 is the largest number";
        }
    } else {
        if (n2 > n3) {
            cout << "n2 is the largest number";
        } else {
            cout << "n3 is the largest number";
        }
    }
    return 0;
}
```

Output

```
n3 is the largest number
```

iamneo

Example program – Switch Case

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

int main() {
    int x = 2;

    switch (x) {
        case 1:
            cout << "Option 1";
            break;
        case 2:
            cout << "Option 2";
            break;
        default:
            cout << "No options";
            break;
    }

    return 0;
}
```

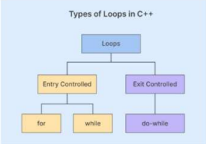
Output

```
Option 2
```

iamneo

Loops in C++

- Loops can execute a block of code as long as a specified condition is reached.
- Loops are handy because they save time, reduce errors, and they make code more readable.



iamneo

for Loop

- *for* loop is used to execute a set of statement repeatedly until a particular condition is satisfied. we can say it an open ended loop. General format is,

```cpp
for(initialization; condition; increment/decrement)
{
    statement block;
}
```

- In for loop we have exactly two semicolons, one after initialization and second after condition. In this loop we can have more than one initialization or increment/decrement, separated using comma operator. for loop can have only one condition.

iamneo

Example Program – for loop

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

int main() {
        for (int i = 1; i <= 5; ++i) {
        cout << i << " ";
    }
    return 0;
}
```

Output

```
1 2 3 4 5
```

iamneo

Example Program – Nested for loop

It is any type of loop inside a for loop.

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

int main() {
    for (int i = 1; i <= 5; i++) {
        for (int j = 1; j <= i; j++) {
            cout << j << " ";
        }
        cout << endl;
    }
    return 0;
}
```

Output

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```


while loop

while loop can be address as an entry control loop. It is completed in 3 steps.

- Variable initialization.(e.g int x=0;)
- condition(e.g while(x<=10))
- Variable increment or decrement (x++ or x-- or x=x+2)

The syntax of the while loop is:

```cpp
variable initialization;
while (condition)
{
    statements;
    variable increment or decrement;
}
```


Example Program – while loop

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

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

Output

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


do while loop

- In some situations it is necessary to execute body of the loop before testing the condition. Such situations can be handled with the help of do-while loop.
- do statement evaluates the body of the loop first and at the end, the condition is checked using while statement. General format of do-while loop is,

```cpp
do
{
    // a couple of statements
}
while(condition);
```

Example Program – do while loop

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

int main () {
   int a = 10;
   do {
      cout << "value of a: " << a << endl;
      a = a + 1;
   } while( a < 15 );
   return 0;
}
```

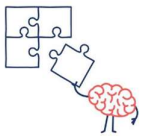
Output

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
```

iamneo

Recall:

- If statement
- If and If else
- If-Else-If Ladder
- Nested If
- Switch Case
- For loop and Nested for
- While
- Do while



iamneo

Classes & Objects

iamneo

OOPs Concept

- 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.
- The main aim of OOP is to bind together the data and the functions that operate on them so that no other part of the code can access this data except that function.

iamneo

Classes

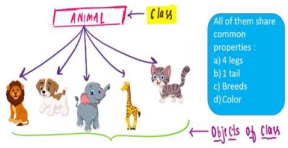
Class in C++ is the building block that leads to object oriented programming.

- Class is a user defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class.
- Data members are the data variables and member functions are the functions used to manipulate these variables together, these data members and member functions define the properties and behavior of the objects in a Class.

iamneo

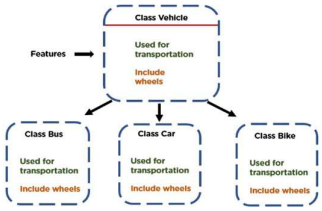
Exploring CLASS

- A C++ class is like a blueprint for an object. Let us consider an example class MOBILE.



iamneo

Features



iamneo

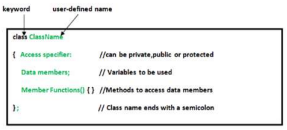
Class & Objects

- An Object is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.
- Here, consider Object as the "key" for accessing anything inside a classroom like chair, projector (data members) and accessing functions inside the classroom like watching lectures, taking a seminar etc.

iamneo

Defining class and Declaring objects

- A class is defined in C++ using the keyword class followed by the name of the class.
- The body of the class is defined inside the curly brackets and terminated by a semicolon at the end.

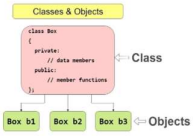


iamneo

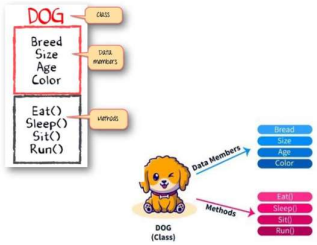
Declaring objects

- When a class is defined, only the specification for the object is defined; no memory or storage is allocated. To use the data and access functions defined in the class, you need to create objects.
- Syntax

```
ClassName ObjectName;
```



iamneo



iamneo

Accessing Data Members and Member Function

- Accessing data members and member functions: The data members and member functions of the class can be accessed using the dot('.') operator with the object.
- For example, if the name of the object is obj and you want to access the member function with the name printName() then you will have to write obj.printName().

iamneo

Accessing Data Members

- The public data members are also accessed in the same way given however the private data members are not allowed to be accessed directly by the object.
- Accessing a data member depends solely on the access control of that data member. This access control is given by Access modifiers in C++.
- There are three access modifiers: public, private, and protected.

iamneo

Example Program using Public

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

class Circle
{
    public:
        double radius;

        double compute_area()
        {
            return 3.14*radius*radius;
        }
};

int main()
{
    Circle obj;

    // accessing public datamember outside class
    obj.radius = 5.5;

    cout << "Radius is: " << obj.radius << "\n";
    cout << "Area is: " << obj.compute_area();
    return 0;
}
```

Output

```
Radius is: 5.5
Area is: 94.985
```

iamneo

Example Programs using Private

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

class Circle
{
    private:
        double radius;
    public:
        void compute_area(double r)
        { // member function can access private
            // data member radius
            radius = r;

            double area = 3.14*radius*radius;

            cout << "Radius is: " << radius << endl;
            cout << "Area is: " << area;
        }
};

int main()
{
    Circle obj;
    obj.compute_area(1.5);
    return 0;
}
```

Output

```
Radius is: 1.5
Area is: 7.065
```

iamneo

Example Program using Protected

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

class Parent
{
    protected:
    int id_protected;
};

class Child : public Parent
{
    public:
    void setId(int id)
    { // child class is able to access the inherited
      // protected data members of base class
        id_protected = id;
    }

    void displayId()
    {
        cout << "id_protected is: " << id_protected << endl;
    }
};

int main() {

    Child obj1; // member function of the derived class can
    // access the protected data members of the base class
    obj1.setId(81);
    obj1.displayId();
    return 0;
}
```

Output

```
id_protected is: 81
```

iamneo

Example Program using Protected

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

class Parent
{
    protected:
    int id_protected;
};

class Child : public Parent
{
    public:
    void setId(int id)
    { // Child class is able to access the inherited
        // protected data members of base class
        id_protected = id;
    }
    void displayId()
    {
        cout << "id_protected is: " << id_protected << endl;
    }
};

int main() {

    Child obj1; // member function of the derived class can
    // access the protected data members of the base class
    obj1.setId(81);
    obj1.displayId();
    return 0;
}
```

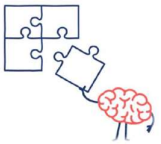
Output

```
id_protected is: 81
```

iamneo

Recall

- OOPs Concept
- Class & Objects
- Accessing Data members
- Accessing member function
- Access specifiers



iamneo

Structure and Union

iamneo

Structures

- The structure is a user-defined data type that is available in C++.
- Structures are used to combine different types of data types, just like an array is used to combine the same type of data types.
- A structure is declared by using the keyword "struct". When we declare a variable of the structure we need to write the keyword "struct in C language but for C++ the keyword is not mandatory.

Structure Syn

```cpp
struct structure_name
{
    member data_type1   member_name1;
    .
    .
    .
    member data_typeN   member_nameN;
};
```

iamneo

Example

- In the above syntax, we have used the struct keyword. The struct_name is the name of the structure.
- The struct members are added within curly braces. These members probably belong to different data types.
- For example:

```
struct Person
{
    char name[30];
    int citizenship;
    int age;
}
```

iamneo

Creating Struct Instances

- In the above example, we have created a struct named Person. We can create a struct variable as follows:

```
Person p;
```

- The p is a struct variable of type Person. We can use this variable to access the members of the struct.

iamneo

Accessing Struct Members

- To access the struct members, we use the instance of the struct and the dot (.) operator. For example, to access the member age of struct Person:

```
p.age = 27;
```

- We have accessed the member age of struct Person using the struct's instance, p. We have then set the value of the member age to 27.

iamneo

Example Program

```
#include <iostream>
using namespace std;
struct Person
{
    int citizenship;
    int age;
};
int main(void) {
    struct Person p;
    p.citizenship = 1;
    p.age = 27;
    cout << "Person citizenship: " << p.citizenship << endl;
    cout << "Person age: " << p.age << endl;

    return 0;
}
```

Output

```
Person citizenship: 1
Person age: 27
```

iamneo

Union

- A union is a type of structure that can be used where the amount of memory used is a key factor.
- Similarly to the structure, the union can contain different types of data types.
- Each time a new variable is initialized from the union it overwrites the previous in C language but in C++ we also don't need this keyword and uses that memory location.

iamneo

Declaration of Union

- This is most useful when the type of data being passed through functions is unknown, using a union which contains all possible data types can remedy this problem.
- It is declared by using the keyword "union".

```
union union_name {
    datatype member1;
    datatype member2;
    ...
};
```

- There are two methods using which we can define a union variable.
    1. With Union Declaration
    2. After Union Declaration

iamneo

Defining Union

- Defining union variable with declaration:

  Syntax:

```
union union_name {
    datatype member1;
    datatype member2;
    ...
} var1, var2,...,
```

- Defining union variable after declaration:

  Syntax:

```
union union_name var1, var2, var3,...,
```

iamneo

Example Program

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

union Number {
    int integer;
    float floating;
};

int main() {
    Number num;
    num.integer = 42;
    cout << "Integer: " << num.integer << endl;
    num.floating = 3.12f;
    cout << "Float: " << num.floating;
    return 0;
}
```

Output

```
Integer: 42
Float: 3.12
```

iamneo

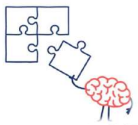
Different between Structure Vs Union

Structure	Union
The keyword 'struct' is used to define a structure.	The keyword 'union' is used to define a union.
A structure's members do not share memory.	A union's members share memory space.
You can retrieve any member at any time in a structure.	In a union, you can access only one member at a time.
A structure's members can all be initialized at the same time.	You can initialize only the first member.
The structure's size is equal to the sum of the sizes of its members.	The union is the same size as its largest member.
Changing the value of one member does not affect the value of another.	A change in one member's values will affect the other members' values.
Stores various values for all members.	Stores the same value for all members.

iamneo

Recall

- Structure
- Union
- Example programs
- Difference between struct and union.



iamneo

Enumeration

iamneo

Enumeration in C++

- An enumeration is a user-defined data type that consists of integral constants. To define an enumeration, keyword *enum* is used.

```
enum season { spring, summer, autumn, winter };
```

- Here, the name of the enumeration is season.
- And, spring, summer and winter are values of type season.
- By default, spring is 0, summer is 1 and so on. You can change the default value of an enum element during declaration (if necessary).

iamneo

Enumerated Type Declaration

- When you create an enumerated type, only a blueprint for the variable is created. Here's how you can create variables of enum type.

```
enum boolean { false, true };
// inside function
enum boolean check;
```

- Here, a variable check of type enum boolean is created.
- Here is another way to declare the same check variable using different syntax.

```
enum boolean
{
    false, true
} check;
```

iamneo

Example: Enumeration Type

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

enum week { Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday };

int main()
{
    week today;
    today = Wednesday;
    cout << "Day " << today+1;
    return 0;
}
```

Output

```
Day 4
```

iamneo

Example: Changing Default Value of Enums

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

enum seasons { spring = 34, summer = 4, autumn = 9, winter = 32};

int main() {
    seasons s;
    s = summer;
    cout << "Summer = " << s << endl;
    return 0;
}
```

Output

```
Summer = 4
```

iamneo

Why are enums used in C++ programming?

- An enum variable takes only one value out of many possible values. Let us look at an example.

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

enum suit {
    club = 0,
    diamonds = 10,
    hearts = 20,
    spades = 3
} card;

int main()
{
    card = club;
    cout << "Size of enum variable " << sizeof(card) << " bytes.";
    return 0;
}
```

Output

```
Size of enum variable 4 bytes.
```

iamneo

Need for Enum Class over Enum Type:

- Below are some of the reasons as to what are the limitations of Enum Type and why we need Enum Class to cover them.
- Enum is a collection of named integer constant means it's each element is assigned by integer value.
- It is declared with enum keyword.

iamneo

Enum Class

- C++ has introduced enum classes (also called scoped enumerations), that makes enumerations both strongly typed and strongly scoped. Class enum doesn't allow implicit conversion to int, and also doesn't compare enumerators from different enumerations.
- To define enum class we use class keyword after enum keyword.

iamneo

Syntax and Example

```
// Declaration
enum class EnumName{ Value1, Value2, ... ValueN};

// Initialisation
EnumName ObjectName = EnumName::Value;
```

```
// Declaration
enum class Color{ Red, Green, Blue};

// Initialisation
Color col = Color::Red;
```

iamneo

Example Program

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

enum rainbow{
    violet,
    indigo,
    blue,
    green,yellow,orange,red
}colors;

enum class eyecolor:char{
    blue,green,brown
}eye;

int main() {

    cout<<"size of enum rainbow variable: "<<sizeof(colors)<<endl;
    cout<<"size of enum class eyecolor variable:"<<sizeof(eye)<<endl;
    return 0;
}
```

Output

```
size of enum rainbow variable: 4
size of enum class eyecolor variable:1
```

iamneo

Example Program

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

enum rainbow{
    violet,
    indigo,
    blue,
    green,yellow,orange,red
}colors;

enum class eyecolor:char{
    blue,green,brown
}eye;

int main() {

    cout<<"size of enum rainbow variable: "<<sizeof(colors)<<endl;
    cout<<"size of enum class eyecolor variable:"<<sizeof(eye)<<endl;
    return 0;
}
```

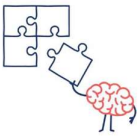
Output

```
size of enum rainbow variable: 4
size of enum class eyecolor variable:1
```

iamneo

Recall:

- Enumeration
- Why are enums used in C++
- Need for enum class over enum
- Enum class
- Example programs



iamneo

Inline & Non-Inline Member Function

iamneo

Function

- A function is a block of code which only runs when it is called.
- You can pass data, known as parameters, into a function.
- Functions are used to perform certain actions, and they are important for reusing code: Define the code once, and use it many times.

iamneo

Create a Function

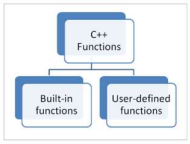
- C++ provides some pre-defined functions, such as main(), which is used to execute code. But you can also create your own functions to perform certain actions.
- To create (often referred to as declare) a function, specify the name of the function, followed by parentheses ():

Syntax:

```
void myFunction() {
  // code to be executed
}
```

iamnco

Type of Functions



Declaration of a function:

The syntax of creating function in C++ language is given below:

```
return_type function_name(data_type parameter...) {
    //code to be executed
}
```

iamnco

Example Code

creative coding

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

// declaring a function
void greet() {
    cout << "Good to see you";
}

int main() {
    // calling the function
    greet();
    return 0;
}
```

Output

```
Good to see you
```

iamnco

Inline Function

- The inline functions help to decrease the execution time of a program.
- The programmer can make a request to the compiler to make the function inline.
- Making inline means that the compiler can replace those function definitions in the place where they are called.
- The compiler replaces the definition of inline functions at compile time instead of referring function definition at runtime.

iamnco

Inline Function

Why do we need Inline function!!

- When performance is needed.
- Use inline function over macros.
- Prefer to use inline keyword outside the class with the function definition to hide implementation details.

iamneo

Syntax of an Inline Function

- The syntax of an Inline function is given below:

```
inline return_type function_name(parameters)
{
    // function code
}
```

- Remember, inlining is only a request to the compiler, not a command.
- The compiler can ignore the request for inlining.

iamneo

How to use Inline Function

- An example program to understand how an inline function is defined and used.

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

inline int square(int s) {
    return s * s;
}

int main() {
    cout << "The square of 3 is: " << square(3);
    return 0;
}
```

Output

```
The square of 3 is: 9
```

iamneo

Advantages and Disadvantages of Inline functions

Advantages	Disadvantages
Function call overhead doesn't occur.	Function inlining variable numbers increase drastically then it would surely cause overhead on register utilization.
It also saves the overhead of push/pop variables on the stack when a function is called.	If you use too many inline functions then the size of the binary executable file will be large, because of the duplication of the same code.
It also saves the overhead of a return call from a function.	Too much inlining can also reduce your instruction cache hit rate, thus reducing the speed of instruction fetch from that of cache memory to that of primary memory.
When you inline a function, you may enable the compiler to perform context-specific optimization on the body of the function. Such optimizations are not possible for normal function calls.	Inline functions might cause thrashing because inlining might increase the size of the binary executable file

iamneo

How to use Inline Member Function

- An example program to understand how an inline member function is defined and used.

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

class A {
public:
    inline int add(int a, int b) {
        return a + b;
    }
};

int main() {
    A obj;
    cout << "The sum of 3 and 4 is: " << obj.add(3, 4);
    return 0;
}
```

Output

```
The sum of 3 and 4 is: 7
```

iamneo

Non-Inline member Function

- A class member function is a function that, like any other variable, is defined or prototyped within the class declaration.
- It has access to all the members of the class and can operate on any object of that class.

There are two ways to define a procedure or function that belongs to a class:

- Inside Class Definition
- Outside Class Definition

iamneo

Non-Inline member Function

Inside Class Definition

- The member function is defined inside the class definition it can be defined directly.
- Similar to accessing a data member in the class we can also access the public member functions through the class object using the **dot operator (.).**

Syntax

```cpp
class ClassName {
public:
    return_type MethodName() {  // Method inside class definition
        // Body of member function
    }
};
```

iamneo

Non-Inline member Function

Outside Class Definition

- The member function is defined outside the class definition it can be defined using the **Scope resolution operator.**
- Similar to accessing a data member in the class we can also access the public member functions through the class object using the **dot operator (.).**

Syntax

```cpp
class ClassName {
public:
    return_type MethodName();  // Method declaration outside class definition
};

// Outside the class using the scope resolution operator
return_type ClassName::MethodName() {
    // Body of member function
}
```

iamneo

Difference Between Inline and Non-Inline Member Function

Inline member function

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

// Class definition for Multiplier
class Multiplier {
public:
    // Inline member function to multiply two integers
    int multiply(int a, int b) {
        return a * b;
    }
};

int main() {
    Multiplier multiplier; // Create an instance of Multiplier
    int a = 5, b = 10; // Define two integers
    cout << multiplier.multiply(a, b); // product a and b
    return 0;
}
```

Non-Inline member function

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

// Class definition for Multiplier
class Multiplier {
public:
    // Non-inline member function declaration
    int multiply(int a, int b);
};

// Non-inline member function definition
int Multiplier::multiply(int a, int b) {
    return a * b;
}

int main() {
    Multiplier multiplier; // Create an instance of Multiplier
    int a = 5, b = 10; // Define two integers
    cout << multiplier.multiply(a, b); // product of a and b
    return 0;
}
```

iamneo

Recall

- Basics of Functions
- Inline functions: Need, usage, syntaxes
- Advantanges and disadvantages of Inline and non Inline function
- Inline member functions
- Non-Inline member functions
- Difference between inline and non inline member functions.

iamneo

Static Data Member and Member Functions

iamneo

Static Data member

- Static data members are class members that are declared using the **static** keyword.
- There is only one copy of the static data member in the class, even if there are many class objects.
- This is because all the objects share the static data member.
- The static data member is always initialized to zero when the first-class object is created.

Syntax

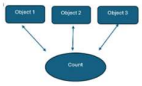
The syntax of the static data members is given as follows –

```
static data_type data_member_name;
```

iamneo

Static Data member

- A static variable is normally used to maintain value common to the entire class. For e.g., to hold the count of objects created.
- Note that the type and scope of each static member variable must be declared outside the class definition.
- This is necessary because the static data members are stored separately rather than as a part of an object.



Example Code

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

class MyClass {
public:
    static int count; // Declaration of static data member
    void increaseCount() {
        count++; // Incrementing the static count variable
    }
    int getCount() {
        return count; // Returning the value of the static count variable
    }
};
// Definition of static data member
int MyClass::count = 0;

int main() {
    MyClass obj1, obj2;
    obj1.increaseCount(); // Increase count via obj1
    obj2.increaseCount(); // Increase count via obj2

    cout << obj1.getCount() << endl; // Access count via obj1
    cout << obj2.getCount(); // Access count via obj2

    return 0;
}
```

Output

```
2
2
```

Static Member Function

- The static member functions are special functions used to access the static data members or other static member functions.
- A member function is defined using the static keyword.
- A static member function shares the single copy of the member function to any number of the class objects.
- We can access the static member function using the class name or class objects.
- If the static member function accesses any non-static data member or non-static member function, it throws an error.

Static Member Function

Syntax

```
class_name::function_name (parameter);
```

Here, the **class_name** is the name of the class.

function_name: The function name is the name of the static member function.

parameter: It defines the name of the pass arguments to the static member function.

Example Code

Let's see program to access the static member function using the class name in the C++ programming language.

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

class MyClass {
public:
    // Static member function
    static void staticFunc() {
        cout << "Hi, John";
    }
};

int main() {
    // Accessing the static member function using the class name
    MyClass::staticFunc();

    return 0;
}
```

iamneo

Example Code

Let's see program to access the static member function using the class object in the C++ programming language.

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

class App {
    // Declare a static data member
    static int num;

public:
    // Create a static member function
    static int func() {
        cout << "The value of num is: " << num << endl;
    }
};

// Initialize the static data member using the class name
int App::num = 15;

int main() {
    // Create an object of the class App
    App n;
    // Access the static member function using the object
    n.func();

    return 0;
}
```

iamneo

Example Code

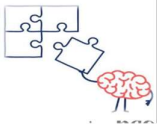
Let's consider an example to access the static member function using the object and class in the C++ programming

```cpp
#include <iostream>
using namespace std;
class MyClass {
private:
    // Declare static data members
    static int varA;
    static int varB;
public:
    // Define the static member function
    static void display() {
        cout << "The value of varA is: " << varA << endl;
        cout << "The value of varB is: " << varB << endl;
    }
};
// Initialize the static data members
int MyClass::varA = 20;
int MyClass::varB = 30;
int main() {
    // Create an object of the class MyClass
    MyClass obj;
    // Access the static member function using the object
    cout << "Print the static member through object name:" << endl;
    obj.display();
    // Access the static member function using the class name
    cout << "Print the static member through the class name:" << endl;
    MyClass::display();
    return 0;
}
```

iamneo

Recall

- Static Data member – Definition, Syntax, Example
- Static member Functions – Syntax and Examples



Function with Default arguments

iamneo

Functions

- A function is a set of statements that are executed together when the function is called.
- Every function has a name, which is used to call the respective function.
- A function **declaration** tells the compiler about a function's name, return type, and parameters. A function **definition** provides the actual body of the function.

iamneo

Default

- In a function, arguments are defined as the values passed when a function is called.
- Values passed are the source, and the receiving function is the destination.
- A default argument is a value in the function declaration automatically assigned by the compiler if the calling function does not pass any value to that argument.

iamneo

Characteristics for defining the default arguments

Following are the rules of declaring default arguments –

- The values passed in the default arguments are not constant. These values can be overwritten if the value is passed to the function. If not, the previously declared value retains.
- During the calling of function, the values are copied from left to right.
- All the values that will be given default value will be on the right.

iamneo

Example Code

```cpp
#include <iostream>
using namespace std;
// Function declaration with default arguments
// Default value for 'b' is 20 and for 'c' is 30
int add(int a, int b = 20, int c = 30) {
  return a + b + c;
}
int main() {
  // Call the function without default arguments
  cout << add(10) << endl; // Uses default values for b and c
  // Call the function with one argument (b takes the default value)
  cout << add(10, 40) << endl; // Uses default value for c
  return 0;
}
```

Output

```
60
80
```

iamneo

Advantages and Disadvantages of Default arguments

Advantages	Disadvantages
Default arguments are useful when we want to increase the capabilities of an existing function as we can do it just by adding another default argument to the function.	It increases the execution time as the compiler needs to replace the omitted arguments by their default values in the function call.
It helps in reducing the size of a program.	If the function call doesn't work properly, then default arguments can never be used.

iamneo

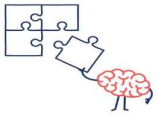
Advantages and Disadvantages of Default arguments

Advantages	Disadvantages
Default arguments are useful when we want to increase the capabilities of an existing function as we can do it just by adding another default argument to the function.	It increases the execution time as the compiler needs to replace the omitted arguments by their default values in the function call.
It helps in reducing the size of a program.	If the function call doesn't work properly, then default arguments can never be used.

iamneo

Recall

- Functions
- Default arguments
- Characteristics for defining the default arguments
- Advantages and disadvantages of default arguments



iamneo

Manipulator Functions in CPP

Introduction to Manipulators

What are Manipulators?

- Manipulators are functions that modify the input/output stream.
- They don't change variable values but alter the format of the I/O stream.
- Used with insertion (<<) and extraction (>>) operators.
- Commonly accessed via **<iomanip>** and **<ios>** header files.

Advantages and Purpose of Manipulators

- It is mainly used to make up the program structure.
- Manipulators functions are special stream function that changes certain format and characteristics of the input and output.
- Manipulators are used to changing the format of parameters on streams and to insert or extract certain special characters.

Types of Manipulators in C++

Non-Argument Manipulators(without Parameters)

- Non-argument manipulators are also known as "Parameterized manipulators".
- These manipulators require **iomanip** header.
- Examples are setprecision, setw and setfill.

Argumented manipulators (With parameters)

- Argument manipulators are also known as "Non parameterized manipulators".
- These manipulators require **iostream** header.
- Examples are endl, fixed, showpoint, left and flush.

Standard input/output Manipulators in C++

Function	Description
setw(int n)	Set field width to n
setbase	Set the base of the number system
setprecision(int p)	Set the precision to p
setfill(Char f)	Set the character to be filled
endl	Output a new line
skipws	Omit white space in input
noskipws	Do not omit white space in the input
ends	Add null character to close an output string
flush	Flush the buffer stream
lock	Lock the file associated with the file handle
ws	Omit leading white spaces before the first field
hex, oct, dec	Display the number in hexadecimal, octal, or decimal format

iamneo

Example Program - 1

Inserting Newline and Flushing the Stream for Integer:

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

int main() {
    int num = 42;
    double n = 45.4;
    cout << num << endl; // endl: inserts a newline and flushes the stream
    cout << n;
    return 0;
}
```

Explanation: Integer printed, followed by newline, then double printed.

Output

```
42
45.4
```

iamneo

Example Program - 2

Setting Field Width and Precision for Integer:

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

int main() {
    int num = 12345;
    cout << setw(10) << num << endl; // setw: sets field width to 10
    return 0;
}
```

Output

```
12345
```

Explanation: Number aligned right with leading spaces to width 10.

iamneo

Example Program - 3

Setting Precision for Float:

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

int main() {
    float num = 3.14159;
    cout << fixed << setprecision(2) << num << endl; // setprecision: sets precision to 2 decimal places
    return 0;
}
```

Output

```
3.14
```

Explanation: Float value fixed to 2 decimal places.

iamneo

Example Program - 4

Filling Unused Space with Character for Double:

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

int main() {
    double num = 123.456;
    cout << setfill('*') << setw(10) << num << endl; // setfill: fills unused space with '*'
    return 0;
}
```

Output

```
***123.456
```

Explanation: Double value with width 10, filled with '*' characters.

iamneo

Recall

- Introduction of Manipulators
- Advantages and Purpose of Manipulators
- Types of Manipulators in C++
- Standard input and output manipulators in C++
- Example programs

Snipping Tool

Screenshot copied to clipboard and saved
Select here to mark up and share.

Function Overloading

iamneo

Function Overloading

- In C++, two functions can have the same name if the number and/or type of arguments passed is different.
- These functions having the same name but different arguments are known as overloaded functions. For example:

```cpp
//same name different arguments
int test() { }
int test(int a) { }
float test(double a) { }
int test(int a, double b) { }
```

- Here, all 4 functions are overloaded functions.

iamneo

Function Overloading

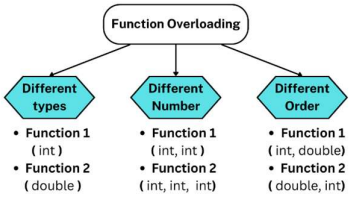
- Notice that the return types of all these 4 functions are not the same.
- Overloaded functions may or may not have different return types but they must have different arguments. For example,

```
//Error code
int test(int a) { }
double test(int b){ }
```

- Here, both functions have the same name, the same type, and the same number of arguments. Hence, the compiler will throw an error.

iamneo

Ways Of Function Overloading In C++



iamneo

Function Overloading Using Different Types Of Parameters

- You can define functions with the same name but multiple parameter types in C++ by overloading various types of parameters.
- This implies that you can design many functions that carry out comparable tasks but accept various data types as input.
- Based on the supplied argument types, the compiler determines which version of the function to call.

Syntax:

```
returnType functionName(parameterType1 parameter1) {
// Function implementation
}
returnType functionName(parameterType2 parameter2) {
// Function implementation
}
```

iamneo

Example - Different Types Of Parameters

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

void printValue(int A) {
    cout << endl << "Value of A : " << A;
}

void printValue(char A) {
    cout << endl << "Value of A: " << A;
}

void printValue(float A) {
    cout << endl << "Value of A : " << A;
}

int main() {
    printValue(10);
    printValue('@');
    printValue(3.14f);

    return 0;
}
```

Output

```
Value of A : 10
Value of A: @
Value of A : 3.14
```

iamneo

Function Overloading With Different Number Of Parameters

- Overloading using a different number of parameters allows you to define multiple functions with the same name but a different number of parameters.
- Here's the syntax for overloading functions with different numbers of parameters:

Syntax:

```
return_type function_name(parameter_type1 parameter1,
parameter_type2 parameter2, ...){
//function implementation
};

return_type function_name(parameter_type1 parameter1,
parameter_type2 parameter2, ..., parameter_typeN parameterN){
//function implementation
};
```

iamneo

Example - Different Number Of Parameters

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

// Function to calculate the sum of two integers
int add(int a, int b) {
    return a + b;
}

// Overloaded function to calculate the sum of three integers
int add(int a, int b, int c) {
    return a + b + c;
}

int main() {
    int sum1 = add(10, 20);
    int sum2 = add(10, 20, 30);
    cout << "Sum 1: " << sum1 << endl;
    cout << "Sum 2: " << sum2 << endl;
    return 0;
}
```

Output

```
Sum 1: 30
Sum 2: 60
```

iamneo

Function Overloading Using Different Sequence Of Parameters

- A function can be overloaded in C++ by altering the order of its parameters. As a result, you are able to define several functions with the same name but various parameter combinations.
- The types and order of the parameters help the compiler distinguish between these overloaded functions.
- The syntax for overloading a function with various parameter sequences is as follows:

```
return_type function_name(type1 parameter, type2 parameter, etc.){
//Function Implementation
};

return_type function_name(type2 parameter, type1 parameter, etc.){
//Function Implementation
};
```

iamneo

Example - Different Sequence Of Parameters

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

void print(int a, double b) {
    cout << "Printing int and double: " << a << ", " << b << endl;
}

void print(double b, int a) {
    cout << "Printing double and int: " << b << ", " << a << endl;
}

int main() {
    print(10, 3.14);
    print(3.14, 10);
    return 0;
}
```

Output

```
Printing int and double: 10, 3.14
Printing double and int: 3.14, 10
```

iamneo

Example - Different Sequence Of Parameters

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

void print(int a, double b) {
    cout << "Printing int and double: " << a << ", " << b << endl;
}

void print(double b, int a) {
    cout << "Printing double and int: " << b << ", " << a << endl;
}

int main() {
    print(10, 3.14);
    print(3.14, 10);
    return 0;
}
```

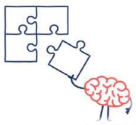
Output

```
Printing int and double: 10, 3.14
Printing double and int: 3.14, 10
```

iamneo

Recall

- Function overloading
- Different types of parameters
- Different number of parameters
- Different sequences of parameters (i.e., different order)
- Example programs



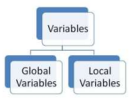
iamneo

Scope Rules

iamneo

Scope Rules

- Scope is a region of a program. Variable Scope is a region in a program where a variable is declared and used.
- Variables are thus of two types depending on the region where these are declared and used.



iamneo

Local

Local Variable

- Variables that are declared inside a function or a block are called local variables and are said to have local scope. These local variables can only be used within the function or block in which these are declared.
- For functions, local variable can either be a variable which is declared in the body of that function or can be defined as function parameters in the function definition
- Now let's see an example where a local variable is declared in the function definition.

iamneo

Local

Example Code - Local Variable

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

void func1(){
    int x = 4;
    cout << x << endl;
}

void func2(){
    int x = 5;
    cout << x << endl;
}

int main(){
    func1();
    func2();
    return 0;
}
```

Output

```
4
5
```

iamneo

Global

Global Variables

- Variables that are defined outside of all the functions and are accessible throughout the program are global variables and are said to have global scope. Once declared, these can be accessed by any function in the program.
- Let's see an example of a global variable.

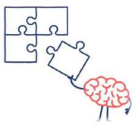
Output

```
20
30
```

iamneo

Recall

- Scope rules
- Local variable
- Global variable
- Example programs



Friend function &
Friend class

iamneo

Friend Function & Friend Class

- Data hiding is a fundamental concept of object-oriented programming. It restricts the access of private members from outside of the class.
- Similarly, protected members can only be accessed by derived classes and are inaccessible from outside. For example,

```
class MyClass {
    private:
        int member1;
}

int main() {
    MyClass obj;
    // Error! Cannot access private members from here.
    obj.member1 = 5;
}
```

iamneo

Friend Function & Friend Class

- However, there is a feature in C++ called friend functions that break this rule and allow us to access member functions from outside the class.
- Similarly, there is a friend class as well.

iamneo

Friend Function

- A friend function can access the private and protected data of a class. We declare a friend function using the *friend* keyword inside the body of the class.

```
class className {
    ... .. ...
    friend returnType functionName(arguments);
    ... .. ...
}
```

iamneo

Example Code - Friend Function

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

class Distance {
    private:
        int meter;
        friend int addFive(Distance);
    public:
        Distance() : meter(0) {}
};

// friend function definition
int addFive(Distance d) {
    //accessing private members from the friend function
    d.meter += 5;
    return d.meter;
}

int main() {
    Distance D;
    cout << "Distance: " << addFive(D);
    return 0;
}
```

Output

```
Distance: 5
```

iamneo

Friend Class

- We can also use a friend Class in C++ using the friend keyword.

  For example,

- When a class is declared a friend class, all the member functions of the friend class become friend functions. Since ClassB is a friend class, we can access all members of ClassA from inside ClassB.
- However, we cannot access members of ClassB from inside ClassA. It is because friend relation in C++ is only granted, not taken.

iamneo

Example Code - Friend Class

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

class ClassB;
class ClassA {
    private:
        int numA;
        // friend class declaration
        friend class ClassB;
    public:
        ClassA() : numA(12) {}
};

class ClassB {
    private:
        int numB;
    public:
        ClassB() : numB(1) {}
        int add() {
            ClassA objectA;
            return objectA.numA + numB;
        }
};

int main() {
    ClassB objectB;
    cout << "Sum: " << objectB.add();
    return 0;
}
```

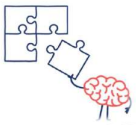
Output

```
Sum: 13
```

iamneo

Recall

- Friend Function
- Friend Class
- Example programs

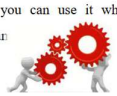


iamneo

Functions and It's Types

Function

- A function is defined as a group of statements or a block of code that carries out specific tasks. You need to give a particular name to the function and write some logic or group of information inside it.
- And then you can invoke that function from the main function. For example, you can name a function as factorial, and inside that function, you can write the logic to find the factorial of a number. And then, you can use it whenever you need that function in the progran



Why Use Functions in C++?

- Functions are used to minimize the repetition of code, as a function allows you to write the code inside the block. And you can call that block whenever you need that code segment, rather than writing the code repeatedly. It also helps in dividing the program into well-organized segments.
- Now, have a look at the syntax of C++ functions.

Function Syntax

- The syntax for creating a function is as follows:

```
return_type function_name(data_type parameter...) {
    //code to be executed
}
```

- Here, the return type is the data type of the value that the function will return. Then there is the function name, followed by the parameters which are not mandatory, which means a function may or may not contain parameters.

Function Declaration

- A function can be declared by writing its return type, the name of the function, and the arguments inside brackets. It informs the compiler that this particular function is present.
- In C++, if you want to define the function after the main function, then you have to declare that function first.

```
int avg(int s1, int s2, int s3);
```

iamneo

Function Definition

- A function definition specifies the body of the function. The declaration and definition of a function can be made together, but it should be done before calling it.

```
int avg(int s1, int s2, int s3) {
    return (s1 + s2 + s3) / 3;
}
```

iamneo

Calling a Function

- When you define a function, you tell the function what to do and to use that function; you have to call or invoke the function.
- When a function is called from the main function, then the control of the function is transferred to the function that is called. And then that function performs its task. When the task is finished, it returns the control to the main function.

```
int main(){
    int a1=3, a2=8, a3=10;
    int average = avg(a1,a2,a3);
    cout << "Average is: " <<average;
}
```

iamneo

Function Call Methods

- You can invoke or call a function in two ways: call by value and call by function.

iamneo

Call By Value

- In this calling method, you only pass the copies of the variable and not the actual argument to the function. As the copies of the variable or arguments are being passed, any changes made to the variable in the function don't affect the actual argument.
- For example:

Output

```
Value of a 100
Value of b 200
```

iamneo

Call By Reference without Using Pointer

- In this calling technique, you pass the address or reference of the argument, and the function receives the memory address of the argument. In this case, the actual value of the variable changes, or you can say that it reflects the changes back to the actual variable.
- For example:

Output

```
Value of a 200
Value of b 100
```

iamneo

Pointer

- In C++, every variable has its unique address or location in the computer's memory, and this special address is called memory address. You can define a pointer as the variable that holds the memory address of some other variable. It allows the developer to deal with the memory.

- Here 2000 is the address of a variable stored by the pointer, and 22 is the variable's value.

iamneo

Declaration and Initialization of a Pointer

- The basic syntax for the pointer in C++ is:

```
data_type *pointer_name;
```

- Here, the data type can be int, char, double, etc. The pointer name can be anything with the * sign. The * operator declares the variable is a pointer.
- Example:

```
int *ptr;
or
char *name;
```

iamneo

Declaration and Initialization of a Pointer

- Here ‘a’ is the variable of data type int, and 30 is the value that is assigned to this variable a.
- Pointer ptr is referring directly to the value a, so here ptr will store the address of variable a. &(address-of operator) is used to acquire the address of data stored in variable a.

Example:

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

int main(){
    int a = 50;
    int *ptr;
    ptr = &a;
    cout << "*ptr value is " << *ptr;
    return 0;
}
```

Output

```
*ptr value is 50
```

iamneo

Call By Reference Using Pointer

- A call by the pointer is a method in C++ to pass the values to the function arguments. In the case of call by pointer, the address of actual parameters is sent to the formal parameter, which means that if we change the values inside the function that will affect the actual values.
- For example:

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

void assignValue(int * a) {
  * a = 4;
}
int main() {
  int x = 3;
  assignValue( & x);
  cout << x;
  return 0;
}
```

Output

```
4
```

iamneo

Call By Reference Using Pointer

- A call by the pointer is a method in C++ to pass the values to the function arguments. In the case of call by pointer, the address of actual parameters is sent to the formal parameter, which means that if we change the values inside the function that will affect the actual values.
- For example:

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

void assignValue(int * a) {
  * a = 4;
}
int main() {
  int x = 3;
  assignValue( & x);
  cout << x;
  return 0;
}
```

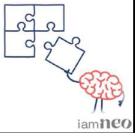
Output

```
4
```

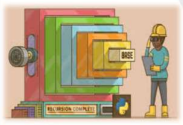
iamneo

Recall

- Function
- Why use function
- Function declaration
- Function definition
- Function Calling
- Calling Methods
- Call by value
- Call by reference with and without using pointer



iamneo



Recursion & Member Function

iamneo

Recursion

- A function that calls itself is known as a recursive function. And, this technique is known as recursion.

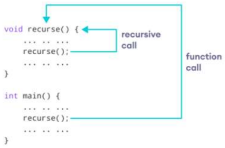
Syntax:

```
void recurse()
{
    ... .. ...
    recurse();
    ... .. ...
}

int main()
{
    ... .. ...
    recurse();
    ... .. ...
}
```

iamneo

Working of Recursion in C++

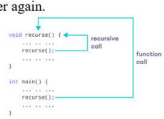
- The figure below shows how recursion works by calling itself over and over again.



iamneo

Working of Recursion in C++

- The figure below shows how recursion works by calling itself over and over again.



- The recursion continues until some condition is met.
- To prevent infinite recursion, if...else statement (or similar approach) can be used where one branch makes the recursive call and the other doesn't.

iamneo

Factorial of a Number Using Recursion

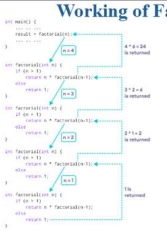
```cpp
#include <iostream>
using namespace std;

int factorial(int);

int main() {
    int n = 4, result;

    result = factorial(n);
    cout << "Factorial of " << n << " = " << result;
    return 0;
}

int factorial(int n) {
    if (n > 1) {
        return n * factorial(n - 1);
    } else {
        return 1;
    }
}
```

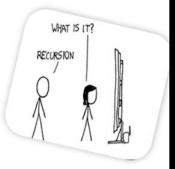
Output

```
Factorial of 4 = 24
```

iamneo

Working of Factorial Program





As we can see, the factorial() function is calling itself. However, during each call, we have decreased the value of n by 1. When n is less than 1, the factorial() function ultimately returns the

iamneo

Recursion in Member Functions Using C++

- Recursion in a member function occurs when a function within a class calls itself to solve smaller instances of the same problem, typically with a base case to terminate the recursive calls and avoid infinite loops.
- For Example,

```cpp
class Factorial {
public:
    int compute(int n) {
        if (n <= 1) return 1; // Base case
        else return n * compute(n - 1); // Recursive case
    }
};
```

iamneo

Recursion in Member Functions Using C++

- Recursion in a member function occurs when a function within a class calls itself to solve smaller instances of the same problem, typically with a base case to terminate the recursive calls and avoid infinite loops.
- For Example,


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

class Fibonacci {
public:
    // Member function
    int compute(int n) {
        if (n <= 1) return n;
        else return compute(n - 1) + compute(n - 2); // Recursive case
    }
};

int main() {
    Fibonacci fib; // Create an instance of the Fibonacci class
    int number = 6;
    int result = fib.compute(number); // Call the recursive member function
    cout << "Fibonacci number " << number << " is " << result;
    return 0;
}
```

Output

```
Fibonacci number 6 is 8
```

iamneo

Recursion in Member Functions Using C++

- Recursion in a member function occurs when a function within a class calls itself to solve smaller instances of the same problem, typically with a base case to terminate the recursive calls and avoid infinite loops.
- For Example,


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

class Fibonacci {
public:
    // Member function to compute Fibonacci number
    int compute(int n) {
        if (n <= 1) // Base case
            return n;
        else // Recursive case
            return compute(n - 1) + compute(n - 2);
    }
};

int main() {
    Fibonacci fib; // Create an instance of the Fibonacci class
    int number = 6;
    int result = fib.compute(number); // Call the recursive member function
    cout << "Fibonacci number " << number << " is " << result;
    return 0;
}
```

Output

```
Fibonacci number 6 is 8
```

iamneo

Recall

- Recursion
- Working flow of recursion
- Recursion in Member Function in C++
- Example program

iamneo