
# CSC2161 PROGRAMMING IN C++

Dieudonne U

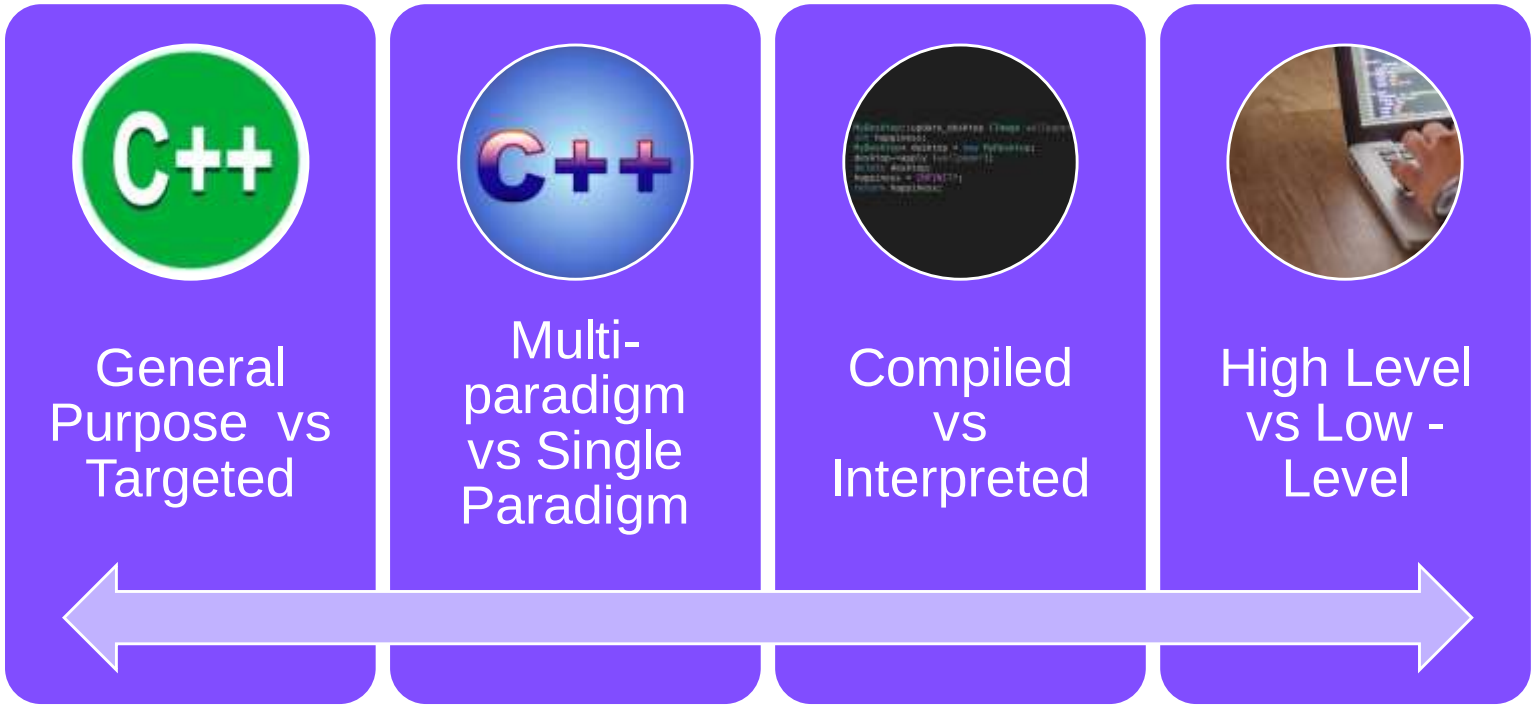
# AGENDA

C++ History

Variables, Operators, Expressions

Control Structures

# C++ Programming Language




# General Purpose

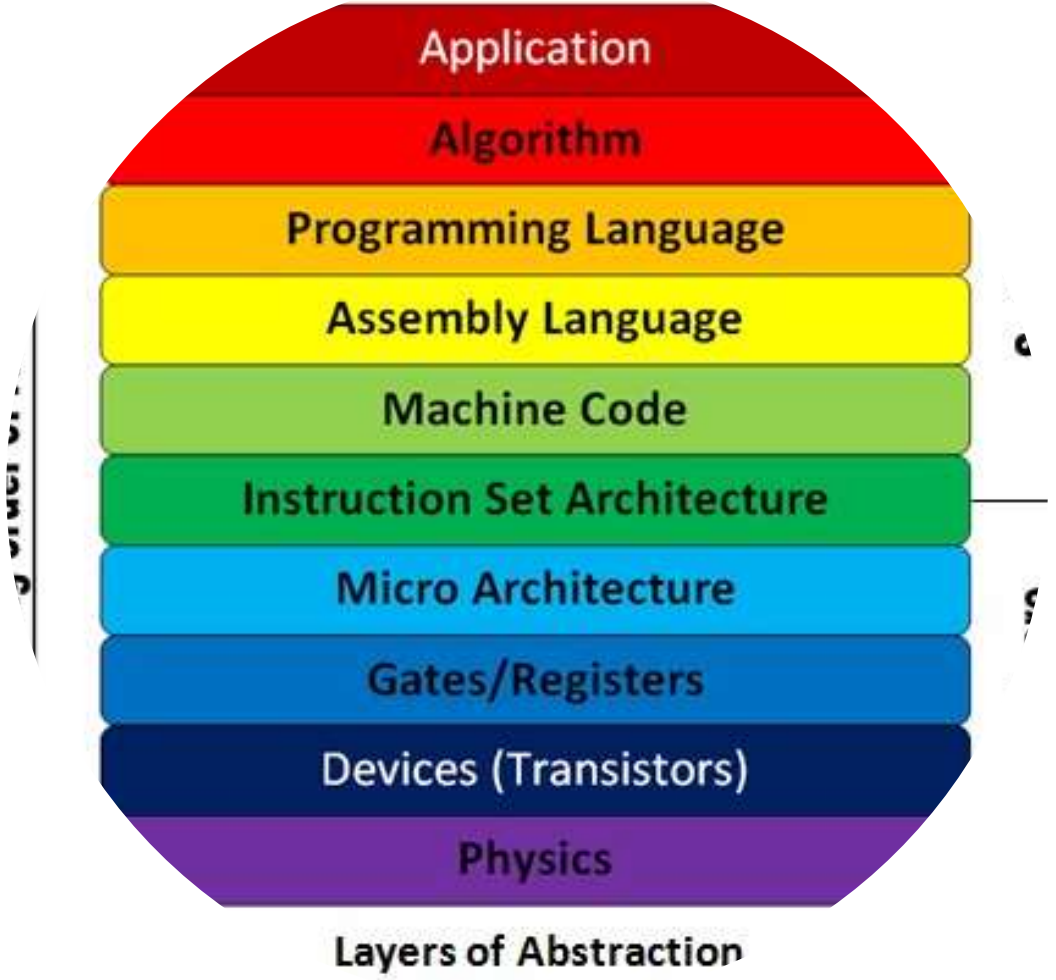
- Used to develop applications in various domains: Finance, Telecommunication, Games.
- As opposed to targeted programming platforms such as MATLAB used in Engineering and Science Applications, R used in Data Science Applications..

# MULTI -PARADIGM

- A programming paradigm is a methodology to develop computer applications.
- There are many programming paradigms: Structured (Fortran, C), Functional (Haskell, Scala, Lisp), Object-oriented (Java, Python, C#), Logic (Prolog)
- C++ supports many programming paradigms : Structured, Object-Oriented, Generic Programming

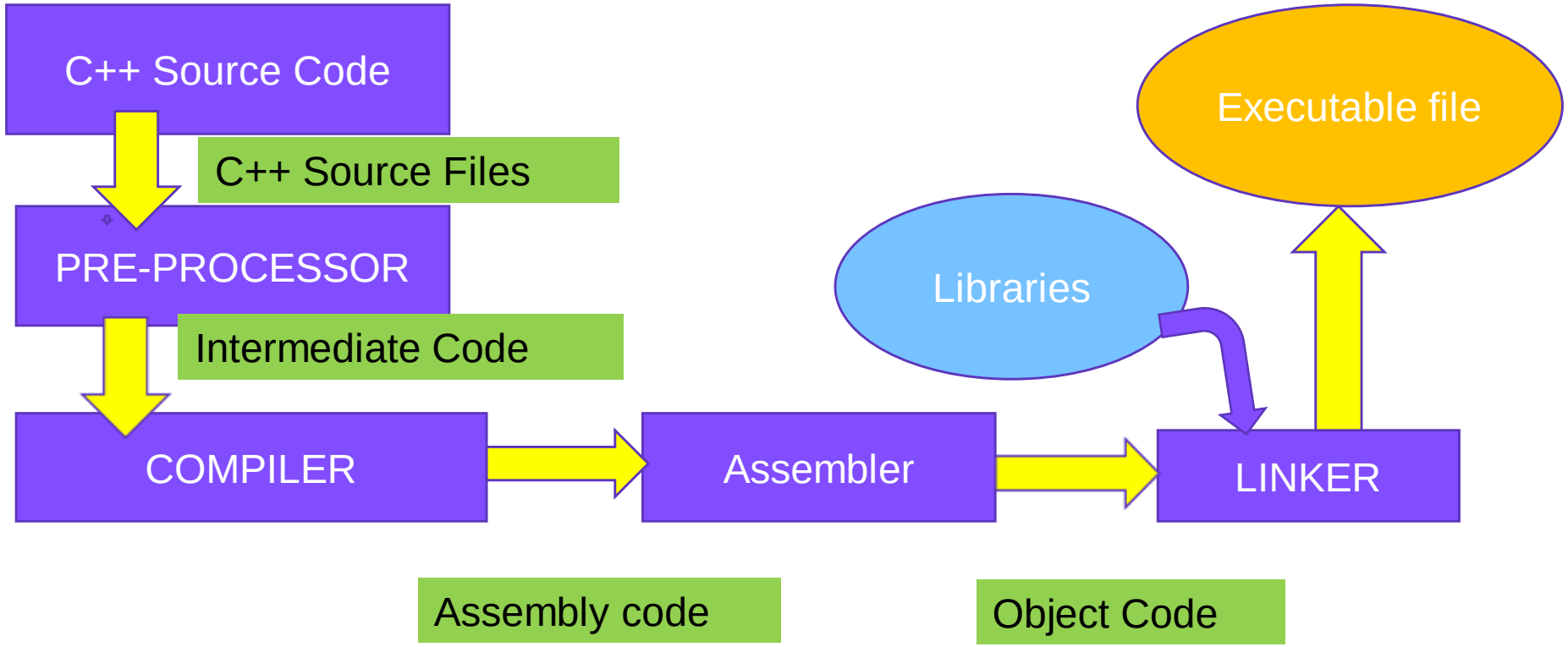
# HIGH LEVEL

- Machine code : Manipulated instructions in binary format, specific to each machine architecture – No abstraction
- Assembly language: Uses commands to manipulate the content of registers. Converted in machine code by and assembler. Very low abstraction
- High Level programing languages: programs are written in a language easier to understand by human, but not understood by the CPU. Has to be either compiled or interpreted

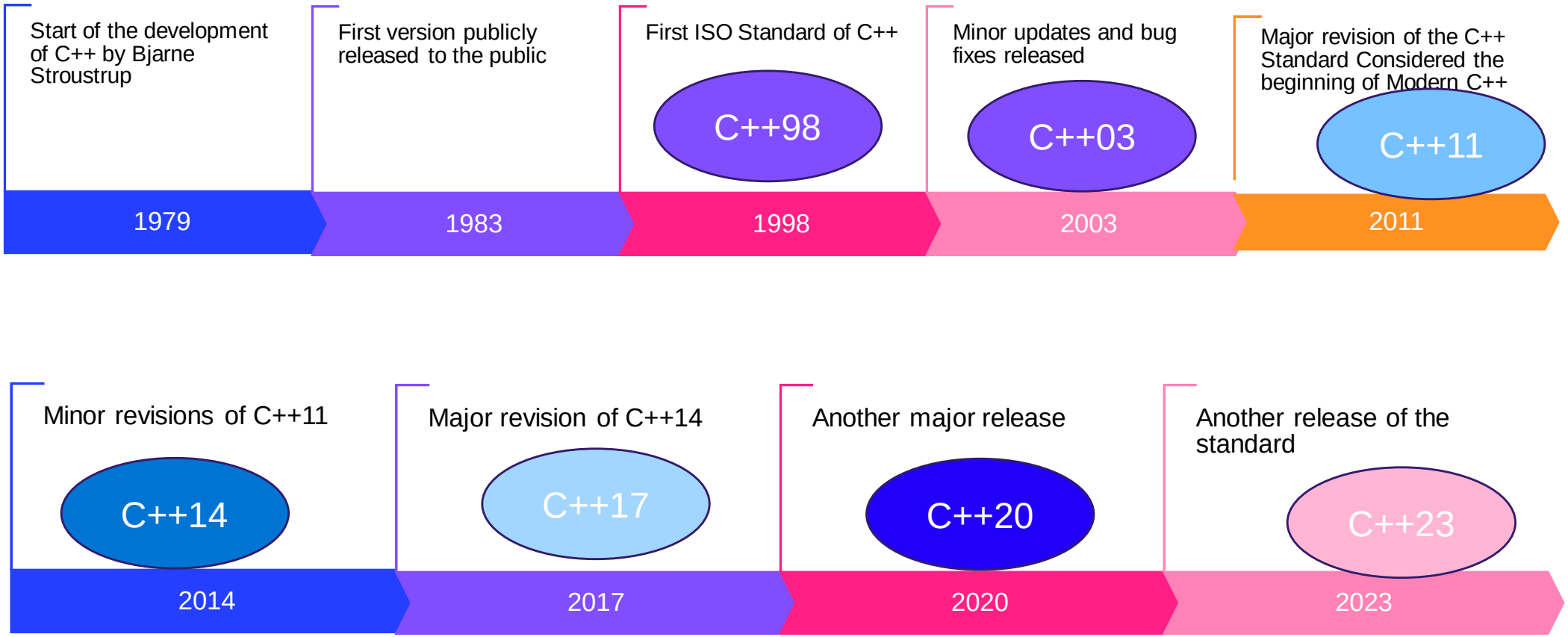


Layers of Abstraction

# C++ Build Process



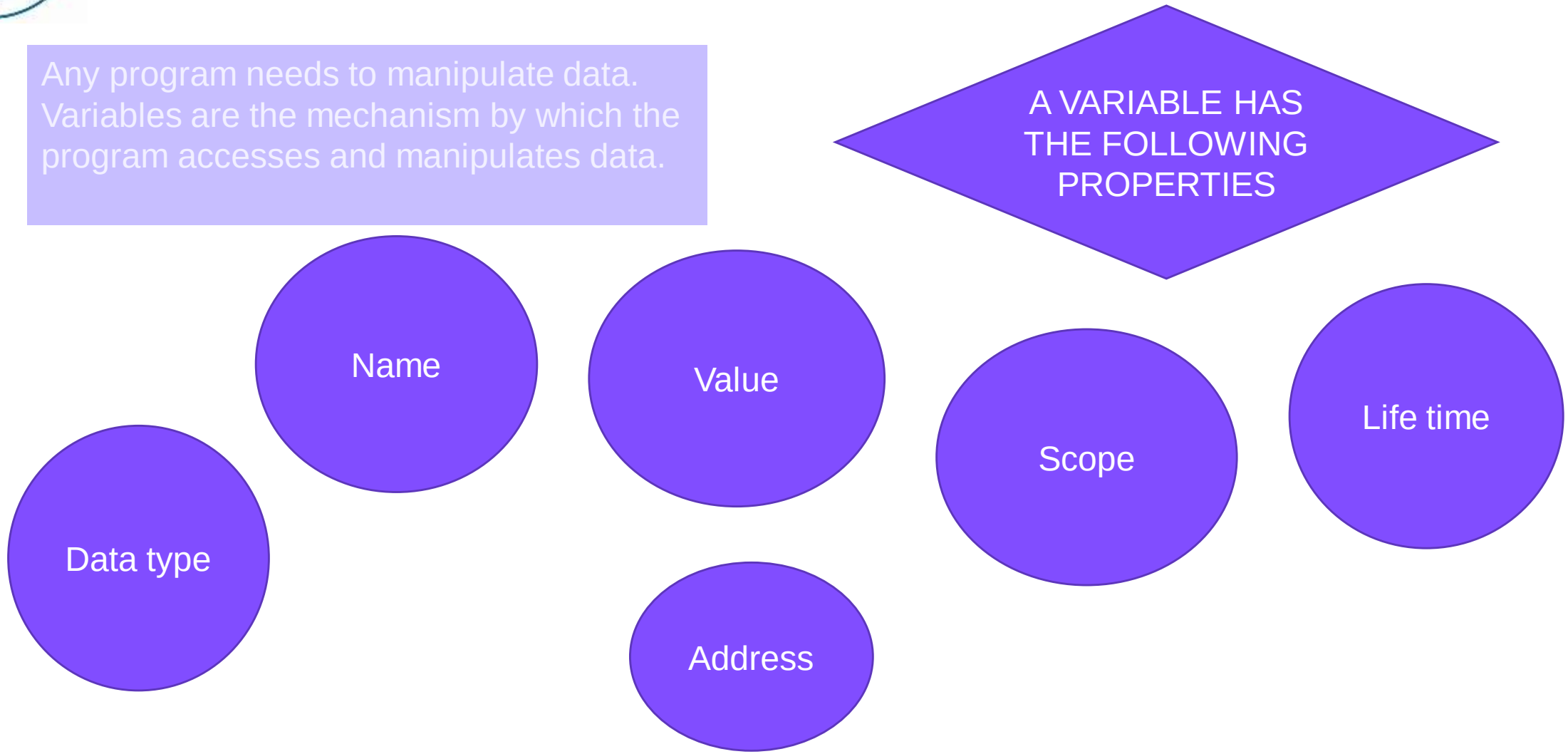
# History of C++


Start of the development of C++ by Bjarne Stroustrup
1979
First version publicly released to the public
1983
First ISO Standard of C++
C++98
1998
Minor updates and bug fixes released
C++03
2003
Major revision of the C++ Standard Considered the beginning of Modern C++
C++11
2011
Minor revisions of C++11
C++14
2014
Major revision of C++14
C++17
2017
Another major release
C++20
2020
Another release of the standard
C++23
2023


# VARIABLES

A variable is a memory location destined to hold a value

Any program needs to manipulate data. Variables are the mechanism by which the program accesses and manipulates data.



A DATA TYPE SPECIFIES THE RANGE OF VALUES A VARIABLE CAN HAVE AND THE OPERATIONS IT CAN BE INVOLVED IN.

Built-in data types are defined by the language

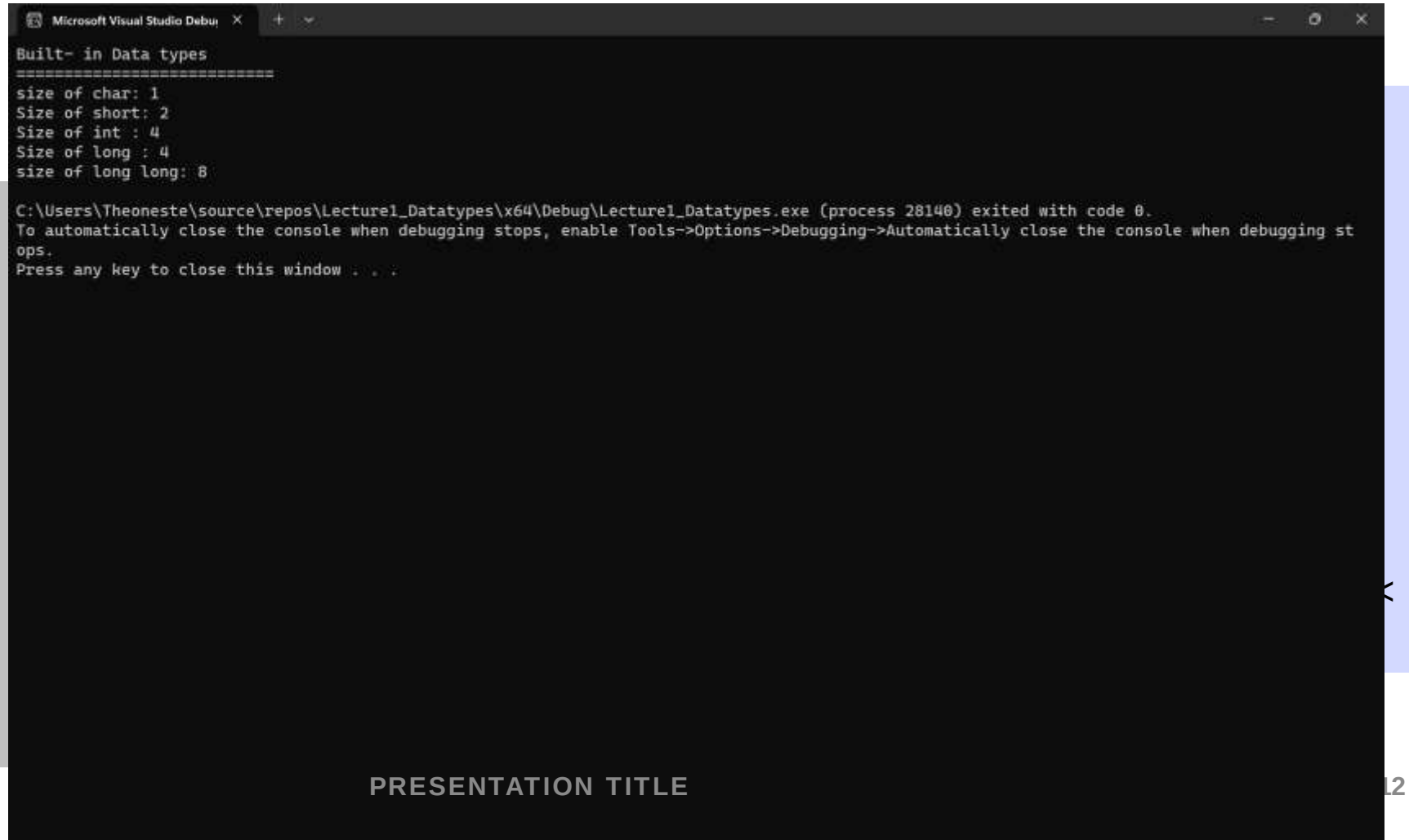
- Integers: (whole numbers)short, int , long, long long.
- Real numbers: (Decimal point numbers) float, double
- A Single character: char
- Boolean (true, false) : bool

```
#include <iostream>

int main()
{
std::cout << "Built- in Data types \n";
std::cout << "============================\n";
std::cout << "size of char: " << sizeof(char) << "\n";
std::cout << "Size of short: " << sizeof(short) << "\n";
std::cout << "Size of int : " << sizeof(int) << "\n";
std::cout << "Size of long : " << sizeof(long) << "\n";
std::cout << "size of long long: "<< sizeof(long long) << "\n";
}
```

A DATA TYPE SPECIFIES THE RANGE OF VALUES A VARIABLE CAN HAVE AND THE OPERATIONS IT CAN BE INVOLVED IN.

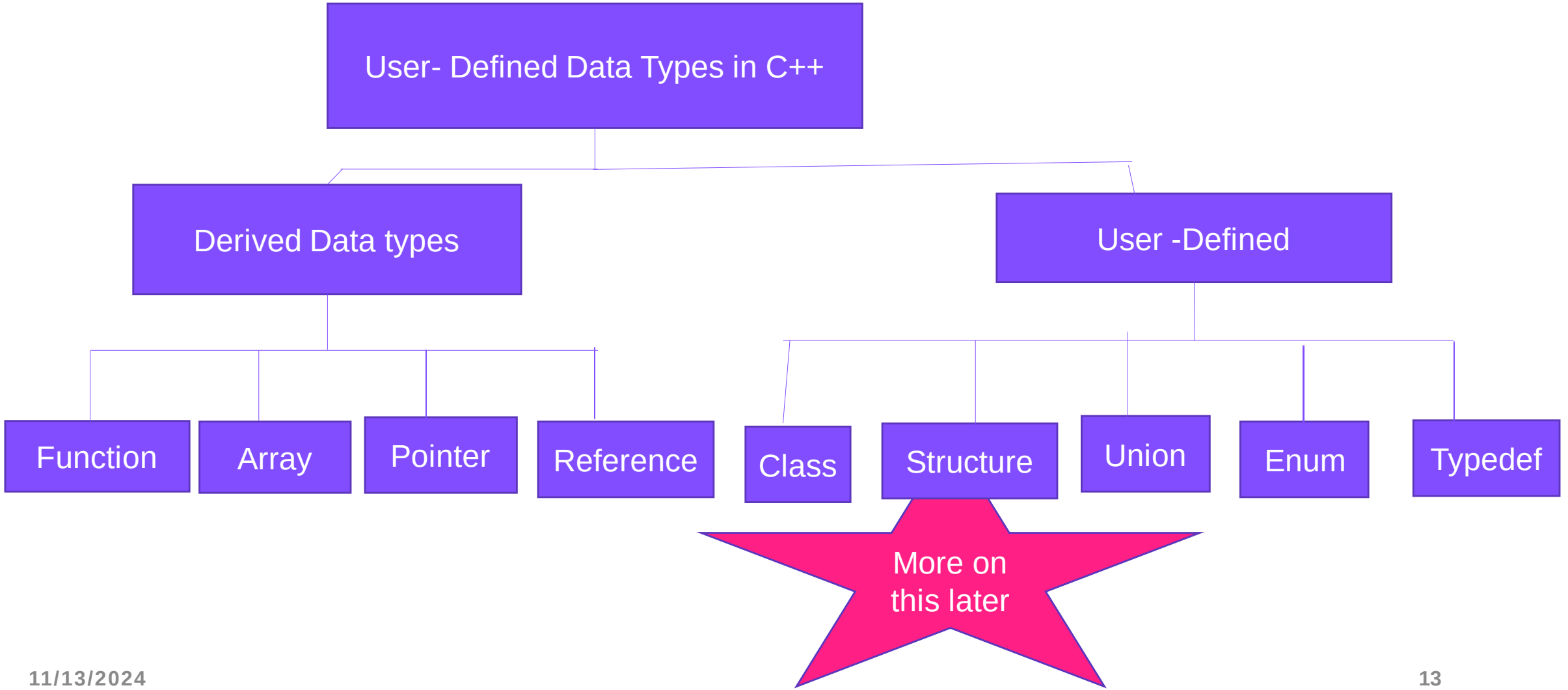
Built-in data types are defined by the language

- Integers: (whole numbers)short, int , long, long long.
- Real numbers: (Decimal point numbers) float, double
- A Single character: char
- Boolean (true, false) : bool




< "\n";

C++ ALLOWS USER TO DEFINE DATA TYPES => C++ IS AN EXTENSIBLE PROGRAMMING LANGUAGE.



IN C++ EVERY VARIABLE MUST BE DECLARED AND INITIALIZED BEFORE BEING USED

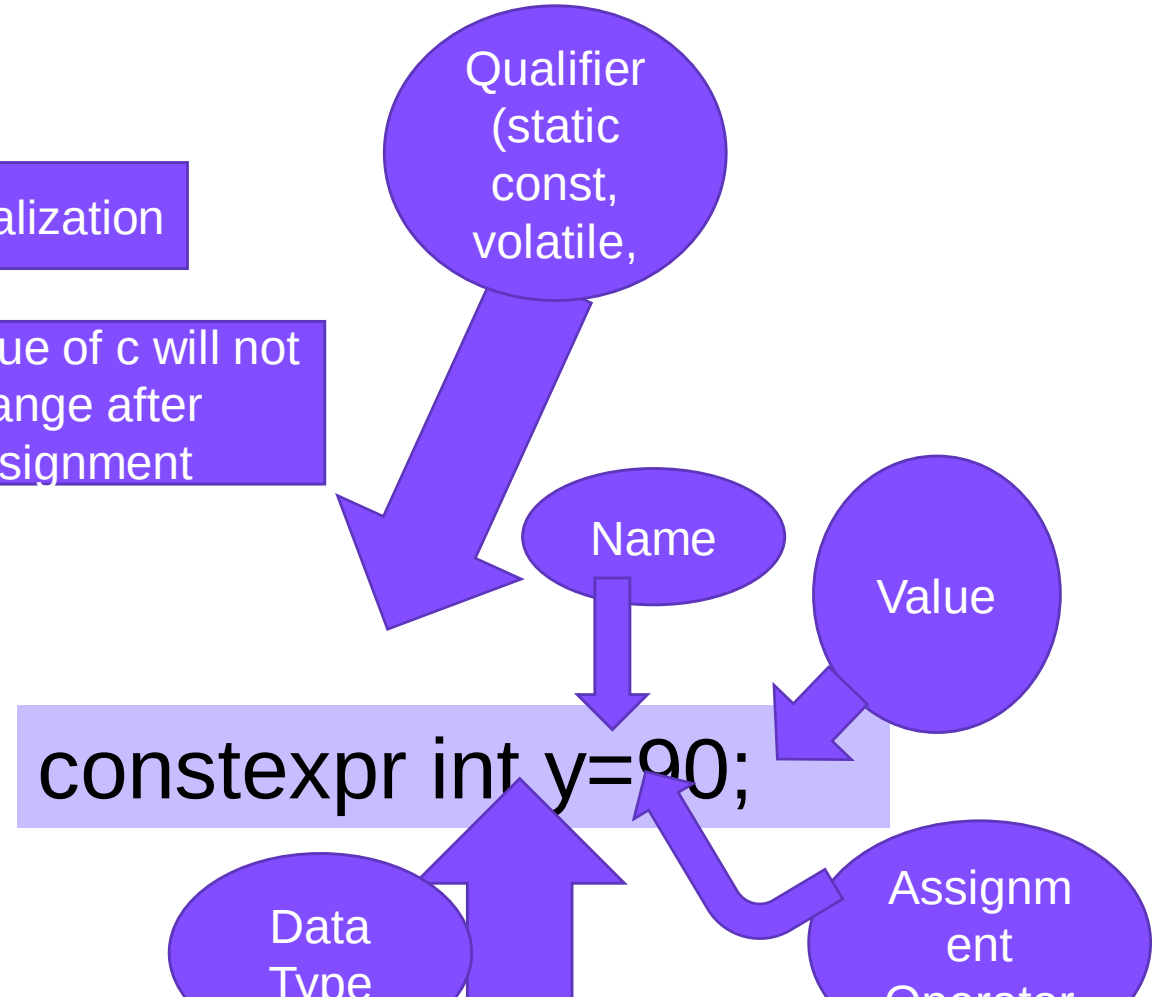
```
#include <iostream>

int a = 7;

int main()
{
int b;
std::cout <<" Enter a value for
b\n";
std::cin >> b;
const int c=b;
constexpr int y = 90;

return 0;
}
```

Declaration and intialization

Declaration without initialization

The value of c will not change after assignment

Constexpr introduced in C++11 Evaluated at compile time (Whenever possible )



IN C++ THE NAME OF A VARIABLE CONTAINS LETTERS, DIGITS AND UNDERSCORE. IT CANNOT START WITH A DIGIT AND IT CANNOT CONTAIN A BLANK SPACE. C++ IS CASE SENSITIVE.

Key words have a specific meaning in C++, and therefore cannot be used as identifiers (name of variables or other names entities)

The same rules apply when naming functions, classes, structures, enumerations, namespaces….

https://en.cppreference.com/w/cpp/keyword

# THE SCOPE OF A VARIABLE IS THE BLOCK IN WHICH THAT VARIABLE IS ACCESSIBLE, WITHIN A PROGRAM

```
#include <iost
int a = 7;
int main()
{
 int a = 8;
 std::cout << " a=     << a << "
 std::cout << "a= " <<  ::a << "\n
 while (a-- > 0)
  {
    static int c
    int d = 1;
    ++c;
    ++d;
    std::cout << "c=          c << "\t" << "d= " << d << "\n";
  }
 return 0;
}
```

Global variable (Declared outside any function)

The Local Variable hides the global one (8)

Global value printed (7)

Scope resolution operator

Even though, it is legal, it is highly discouraged to use the same variable in different scopes of the same program

# THE SCOPE OF A VARIABLE IS THE BLOCK IN WHICH THAT VARIABLE IS ACCESSIBLE, WITHIN A PROGRAM.

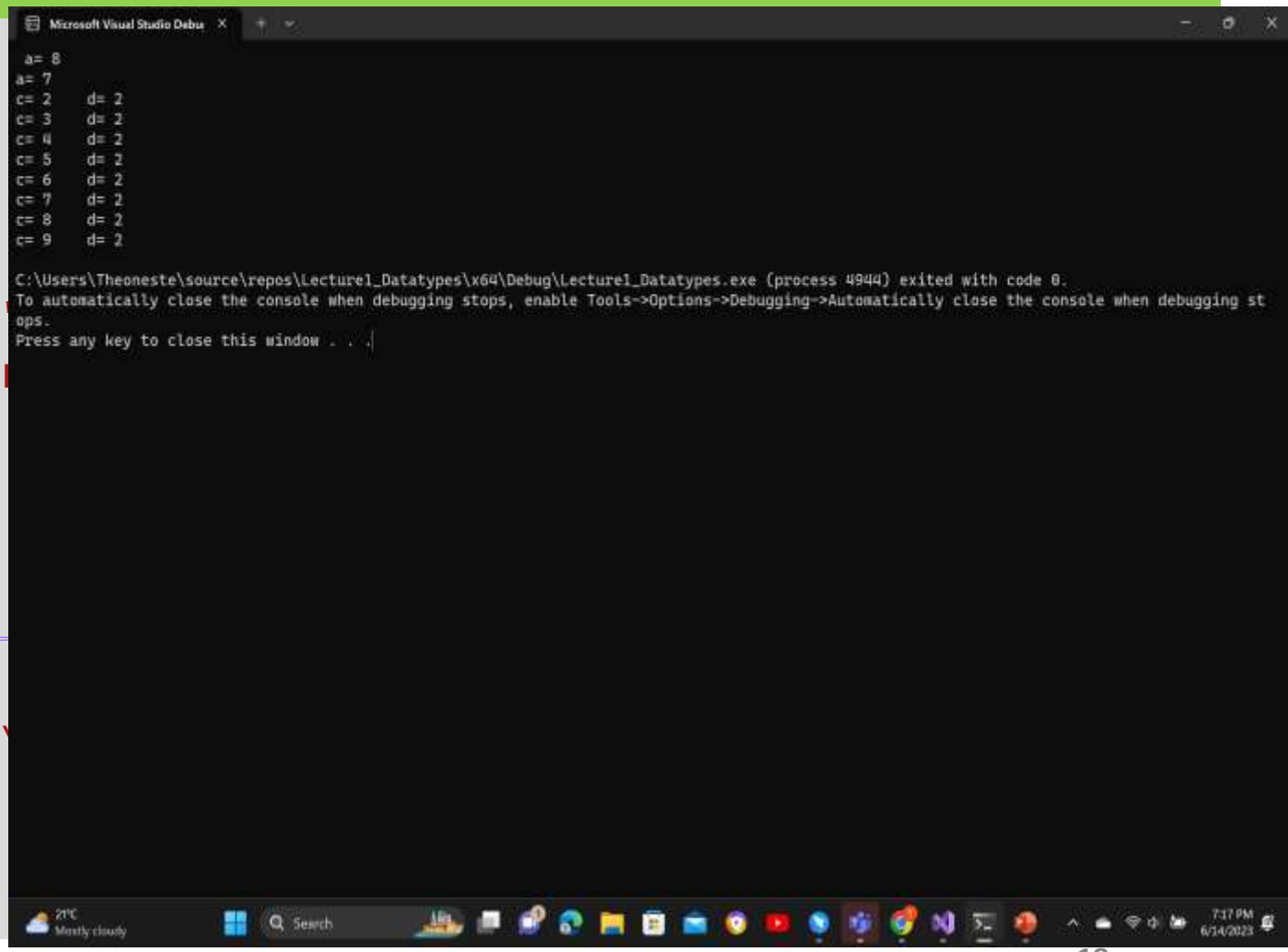
```
#include <iostream>
int a = 7;
int main()
{
 int a = 8;
 std::cout << " a=
 std::cout << "a=
 while (a-- > 0)
  {
    static int c = 1;
    int d = 1;
    ++c;
    ++d;
    std::cout << "c= " << c << "\t" << "d= " << d << "
  }
 return 0;
}
```

The variables c and d are local to the while loop. The variable c is static=> It is initialized only once. When execution leaves its scope, c is inaccessible, but retains its value.

The variable d is automatic => Every time execution leaves its scope, the variable is destroyed and recreated when execution comes back to the scope

# THE SCOPE OF A VARIABLE IS THE BLOCK IN WHICH THAT VARIABLE IS ACCESSIBLE, WITHIN A PROGRAM.

```
#include <iostream>
int a = 7;
int main()
{
 int a = 8;
 std::cout << " a= " << a << "\n
 std::cout << "a= " << ::a << "\
 while (a-- > 0)
  {
    static int c = 1;
    int d = 1;
    ++c;
    ++d;
    std::cout << "c= " << c << "
  }
 return 0;
}
```



# THE SCOPE OF A VARIABLE IS THE BLOCK IN WHICH THAT VARIABLE IS ACCESSIBLE, WITHIN A PROGRAM.

Life Time

```
#include <iostream>
int a = 7;
int main()
{
 int a = 8;
 std::cout << " a= " << a << "\n";
 std::cout << "a= " << ::a << "\n";
 while (a-- > 0)
  {
    static int c = 1;
    int d = 1;
    ++c;
    ++d;
    std::cout << "c= " << c << "\t" << "d= " << d << "
  }
 return 0;
}
```

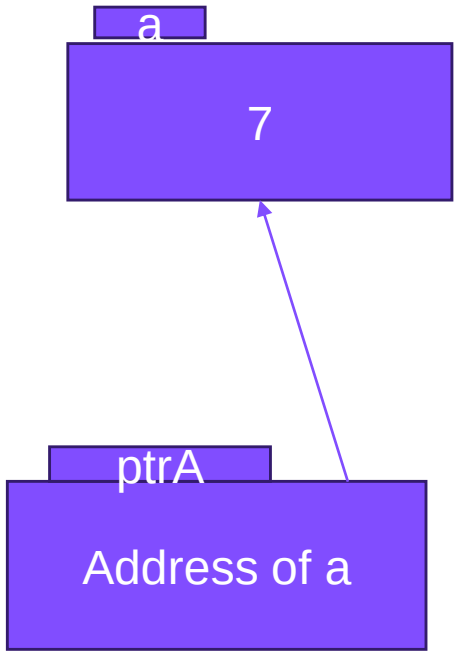
Global variables and live and are accessible from the point of their declaration up to the end of the program

Static variables live from their declaration up to the end of the program. They are not accessible outside of their scope

Automatic local variables exist when the program execution is in their scope

# EVERY VARIABLE HAS AN ADDRESS. C AND C++ GIVE THE PROGRAMMER DIRECT ACCESS TO THE ADDRESS OF A VARIABLE USING THE ADDRESSOF OPERATOR

```
#include <iostream>

int main()
{
int a = 7;
int* ptrA = &a;
std::cout << "Value of a : " << a << "\n";
std::cout << "Address of a : " << ptrA << "\n";

return 0;
}
```




# OPERATORS

# C++ SUPPORTS MULTIPLE OPERATORS OPERATORS ENABLE COMPUTATION

| Operator | Name | Description | Example |
|---|---|---|---|
| + | Addition | Adds together two values | x + y |
| - | Subtraction | Subtracts one value from another | x - y |
| * | Multiplication | Multiplies two values | x * y |
| / | Division | Divides one value by another | x / y |
| % | Modulus | Returns the division remainder | x % y |
| ++ | Increment | Increases the value of a variable by 1 | ++x |
| -- | Decrement | Decreases the value of a variable by 1 | --x |

ARITHMETIC OPERATORS

# RELATIONAL OPERATORS

| Operator | Name | Example |
|---|---|---|
| == | Equal to | x == y |
| != | Not equal | x != y |
| > | Greater than | x > y |
| < | Less than | x < y |
| >= | Greater than or equal to | x >= y |
| <= | Less than or equal to | x <= y |

# LOGICAL OPERATORS

| Operator | Name | Description | Example |
|---|---|---|---|
| && | Logical and | Returns true if both statements are true | x < 5 && x < 10 |
| \|\| | Logical or | Returns true if one of the statements is true | x < 5 \|\| x < 4 |
| ! | Logical not | Reverse the result, returns false if the result is true | !(x < 5 && x < 10) |
| | | | |

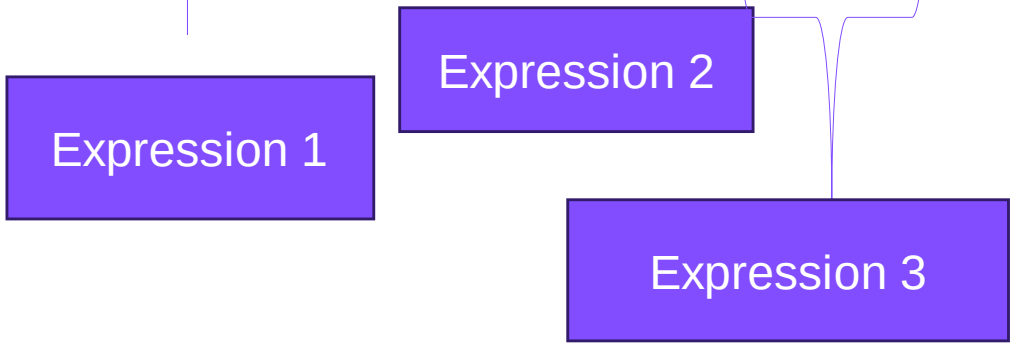
# ASSIGNMENT OPERATORS

| Operator | Example | Same As |
|---|---|---|
| = | x = 5 | x = 5 |
| += | x += 3 | x = x + 3 |
| -= | x -= 3 | x = x - 3 |
| *= | x *= 3 | x = x * 3 |
| /= | x /= 3 | x = x / 3 |
| %= | x %= 3 | x = x % 3 |
| &= | x &= 3 | x = x & 3 |
| \|= | x \|= 3 | x = x \| 3 |
| ^= | x ^= 3 | x = x ^ 3 |
| >>= | x >>= 3 | x = x >> 3 |
| <<= | x <<= 3 | x = x << 3 |

# INCREMENT (DECREMENT OPERATORS CAN BE USED AS PREFIX OR POSTFIX.

Conditional (Ternary Operator)

```
int c = a > b ? --a : b++;
```




```
#include <iostream>

int main()
{
int a = 8;
int b = 10;
int c = a > b ? --a : b++;
std::cout << "a= " << a << "\n";
std::cout << "b= " << b << "\n";
std::cout << "c= " << c << "\n";
return 0;
}
```

If expression 1 evaluates to true, then expression 2 is evaluated, otherwise expression 3 is evaluated

<
<< >>

PRECEDENCE OF OPERATORS
DETERMINES THE ORDER OF OPERATIONS.
WHEN OPERATORS HAVE THE SAME PRECEDENCE, ASSOCIATIVITY
DETERMINES THE ORDER OF OPERATIONS.

| * / % | + - | << >> | When in doubt use parentheses |
|---|---|---|---|

https://en.cppreference.com/w/cpp/language/operator_precedence

# C++ HAS MANY MORE OPERATORS

- Function call ()
- Array subscript []
- sizeof operator already seen
- new and delete operators
- Bitwise operators
- Scope resolution operator already seen
- And many more…

# TYPE CASTING CONSISTS IN CONVERTING A VARIABLE TO  A DIFFERENT TYPE. C++ HAS A NUMBER OF TYPE CAST OPERATORS.

```
#include <iostream>

int main()
{
int number1 = 7;
int number2 = 4;
std::cout << "7/4 = " << number1 / numb  << "\n";
std::cout << "7/4 = " << static_cast<double>(number1) / number2 << "\n";
int number3 = static_cast<double>(number1) / number2;
std::cout <<  ber3 << "\n";
return 0;
}
```

EXPLICIT TYPE CAST

IMPLCIT TYPE CAST

IN C++ ANY OPERATION  HAPPENS BETWEEN VARIABLES OF THE SAME TYPE

Expression is the combination of the constants, variables, and operators, which are arranged according to the syntax of C++ language and, after computation, return some values that may be a boolean, an integer, or any other data type of C++.

An expression is a sequence of *operators* and their *operands*, that specifies a computation.
Expression evaluation may produce a result (e.g., evaluation of `2 + 2` produces the result `4`) and may generate side-effects (e.g. evaluation of `std::printf("%d", 4)` prints the character `'4'` on the standard output).
Each C++ expression is characterized by two

https://en.cppreference.com/w/cpp/language/expressions

category.


# CONTROL STRUCTURES

## DECISION CONTROL STRUCTURES

DECISION CONTROL STRUCTURES ALLOW PROGRAMS TO MAKE DECISIONS BASED ON SOME CONDITION. BASED ON A CONDITION THE PROGRAM SELECTS WHAT TO DO, AMONG MANY ALTERNATIVES.

```
#include <iostream>

int main()
{
int number;
std::cout << "Enter a positive number \n";
std::cin >> number;
if (number > 0)
   std::cout << "Great you have followed  the
instruction\n";
return 0;
```

DECISION CONTROL STRUCTURES ALLOW PROGRAMS TO MAKE DECISIONS BASED ON SOME CONDITION. BASED ON A CONDITION THE PROGRAM SELECTS WHAT TO DO, AMONG MANY ALTERNATIVES.

```
#include <iostream>

int main()
{
int number;
std::cout << "Enter a positive number \n";
std::cin >> number;
if (number > 0)
    std::cout << "Great you have followed  the
instruction\n";
else
```

DECISION CONTROL STRUCTURES ALLOW PROGRAMS TO MAKE DECISIONS BASED ON SOME CONDITION. BASED ON A CONDITION THE PROGRAM SELECTS WHAT TO DO, AMONG MANY ALTERNATIVES.

```
#include <iostream>

int main()
{
int number;
std::cout << "Enter a positive number \n";
std::cin >> number;
if (number > 0)
    std::cout << "Great you have followed  the
instruction\n";
else
```

```cpp
#include <iostream>

int main()
{
 double cumulative_average;
 char letter_grade;
 std::cout << " Enter your average: \n";
 std::cin >> cumulative_average;
 if (cumulative_average >= 80)
    letter_grade = 'A';
 else if (cumulative_average >= 70)
    letter_grade = 'B';
 else if (cumulative_average >= 60)
    letter_grade = 'C';
 else if (cumulative_average >= 50)
    letter_grade = 'D';
 else
    letter_grade = 'F';

 std::cout << "Average : " << cumulative_average << "\n";
 std::cout << "Letter grade :" << letter_grade << "\n";

 return 0;
}
```

# SWITCH ….CASE STATEMENT WORKS IN A SIMILAR WAY AS IF ELSE IF….ELSE BUT WITH SOME RESTRICTIONS

```
#include <iostream>
int main()
{
int number;
std::cout << "Enter a positive integer\n";
std::cin >> number;
switch (number % 3)
{
    case 0:
    std::cout << number << " is divisible by 3\n";
    case 1:
    std::cout << "remainder is  1\n";
    case 2:
    std::cout << "remainder is 2 \n";
    default:
    std::cout << "Unkown result\n";
}
return 0;
}
```

```
#include <iostream>
int main()
{
int number;
std::cout << "Enter a positive integer\n";
std::cin >> number;
switch (number % 3)
{
   case 0:
   std::cout << number << " is divisible by 3\n";
   break;
   case 1:
   std::cout << "remainder is  1\n";
   break;
   case 2:
   std::cout << "remainder is 2 \n";
   break;
   default:
   std::cout << "Unkown result\n";
}
return 0;
}
```


# CONTROL STRUCTURES

## REPETITION CONTROL STRUCTURES

IN A LOOP (REPETITION CONTROL STRUCTURE) A GROUP OF STATEMENTS IS REPEATEDLY EXECUTED, UNTIL A CONDITION IS FALSE.

```
#include <iostream>
int main()
{
 int number;
 int i = 0;
 std::cout << "Enter a positive number : \n";
 std::cin >> number;
 while (i < number)
  {
    std::cout << "Hello World\n";
    i++;
  }
 std::cout << " From Kigali\n";
 return 0;
}
```

Assume number is 5. What will be the output of this program?

IN A LOOP (REPETITION CONTROL STRUCTURE) A GROUP OF STATEMENTS IS REPEATEDLY EXECUTED, UNTIL A CONDITION IS FALSE.

```
#include <iostream>
int main()
{
 int number;
 int i = 0;
 std::cout << "Enter a positive number : \n";
 std::cin >> number;
 while (i < number)
  {
    std::cout << "Hello World\n";
    i++;
  }
 std::cout << " From Kigali\n";
 return 0;
}
```

Assume number is -1. What will be the output of this program?

# A WHILE IS ENTRY CONTROLLED – A DO WHILE LOOP IS EXIT CONTROLLED

While loop executes o or more times

```
#include <iostream>

int main()
{
int number;
int i = 0;
std::cout << "Enter a number : \n";
std::cin >> number;
while (i < number)
{
std::cout << "Hello World\n";
i++;
}
std::cout << " From Kigali\n";


return 0;
}
```

DO While loop executes 1 or more times

```
#include <iostream>

int main()
{
 int number;
 int i = 0;
 std::cout << "Enter a number : \n";
 std::cin >> number;
 do
  {
     std::cout << "Hello World\n";
     i++;
   }while (i < number);
std::cout << " From Kigali\n";


return 0;
}
```

# A FOR LOOP HAS THREE EXPRESSIONS : INITIALIZATION , CONDITION, INCREMENT (DECREMENT)

```
#include <iostream>
int main()
{
 int number;
 std::cout << "Enter a positive number : \n";
 std::cin >> number;
 for(int i=0;i<number;++i)
    {
    std::cout << "Hello World\n";
    }
std::cout << " From Kigali\n";
return 0;
}
```

# A FOR LOOP HAS THREE EXPRESSIONS : INITIALIZATION , CONDITION, INCREMENT (DECREMENT)

```
#include <iostream>
int main()
{
 int number;
 int i=0;
 std::cout << "Enter a positive number : \n";
 std::cin >> number;
 for(;i<number;++i)
    {
    std::cout << "Hello World\n";
    }
std::cout << " From Kigali\n";
return 0;
}
```

# A FOR LOOP HAS THREE EXPRESSIONS : INITIALIZATION , CONDITION, INCREMENT (DECREMENT)

```
#include <iostream>
int main()
{
 int number;
 int i=0;
 std::cout << "Enter a positive number : \n";
 std::cin >> number;
 for(;i<number;)
    {
    std::cout << "Hello World\n";
    ++i;
    }
std::cout << " From Kigali\n";
return 0;
```

# A FOR LOOP HAS THREE EXPRESSIONS : INITIALIZATION , CONDITION, INCREMENT (DECREMENT)

```
#include <iostream>
int main()
{
 int number;
 int i=0
 std::cout << "Enter a positive number : \n";
 std::cin >> number;
 for(; ;)
    {
     std::cout << "Hello World\n";
     ++i;
     if(i>=number)
         break;
}
```

Break statement terminates the immediately enclosing loop

```
#include<iostream>
int main()
{
int number;
std::cout << "Enter a positive number : \n";
std::cin >> number;
for(int i=0;i<number;++i)
{
 if (number % 2)
    continue;
 std::cout << number<<"\n";
}
std::cout << " From Kigali\n";
return 0;
}
```

Continue statement cuts short the current iteration and moves to the next.

```
#include <iostream>
int main()
{
int number=3;
while (number >= 0)
{
    std::cout << number * number << "\n";
    --number;
}
std:: cout <<number <<"\n";
while (number < 4)
    std::cout << ++number << "\n";
std::cout << number<<"\n";
while (number >= 0)
    std::cout << (number /= 2) << "\n";
return 0;
}
```

WHAT IS THE OUTPUT ??

```cpp
#include <iostream>
int main()
{
int number=3;
while (number >= 0)
{
    std::cout << number * number << "\n";
    --number;
}
std:: cout <<number <<"\n";
while (number < 4)
    std::cout << ++number << "\n";
std::cout << number<<"\n";
while (number >= 0)
    std::cout << (number /= 2) << "\n";
return 0;
}
```

```
9
4
1
0
-1
0
1
2
3
4
4
2
1
0
```

```
#include <iostream>
int main()
{
int number=3;
while (number >= 0)
{
    std::cout << number * number <<
    --number;
}
std:: cout <<number <<"\n";
while (number < 4)
    std::cout << ++number << "\n";
std::cout << number<<"\n";
while (number >= 0)
    std::cout << (number /= 2) << "\n";
return 0;
}
```

9
4
1
0

-1

0
1
2
3
4

4

2
1
0

9
4
1
0
-1
0
1
2
3
4
4
2
1
0

```cpp
#include <iostream>
int main()
{
int x=10;
std::cout << --x + 1 << "\n";
std::cout << x++ << "\n";

return 0;
}
```

What is the ouput??

```cpp
#include <iostream>
int main()
{
int x=1,y=2;
std::cout << x - y + 3 * 4 / 5 << "\n";

return 0;
}
```

What is the output??

```
#include <iostream>
int main()
{
int grade = 100;
if (grade > 60 and grade < 100)
    std::cout << "Success \n";
else
    std::cout << "Fail\n";
return 0;
}
```

What is the output??

```
#include <iostream>
int main()
{
   int x = 1, y = 2;
   switch (y)
   {
   case 1:
   std::cout << "One\n";
   case 2:
   std::cout << "Two\n";
   default:
   std::cout << "Three\n";
}
return 0;
}
```

What is the output ??

```
#include <iostream>
int main()
{
int number_legs;
std::cout << "Enter number of legs\n";
std::cin >> number_legs;
if (number_legs == 1)
        std::cout << "Pirate\n";
else if (number_legs == 2)
        std::cout << "Bird\n";
else if (number_legs == 4)
        std::cout << "Cat\n";
else if (number_legs == 6)
        std::cout << "Spider\n";
else
        std::cout << "Idon't know who you are\n";
return 0;
}
```

Rewrite this program using switch –case instead of if –else

Rewrite this program using a while loop instead of the for loop!

```
#include <iostream>
int main()
{
int max_number;
std::cout << "Enter max number\n";
std::cin >> max_number;
for (int i = max_number; i >= 0; i--)
    std::cout << i << "\n";

return 0;
}
```

# ASSIGNMENT 1 – PROBLEM 1

You own a bicycle rental business and you want to write a program that will help you determine what your customers will have to pay you for a rental. You are going to provide two integers as input. The first integer represents the starting time (hour) and the second integer represent the ending time (hour). Decimal numbers are not accepted.

(1) The starting time will always be less than the ending time. Also no rental can span more than a day. The start time will be between 0 and 23 and the ending time will be between 1 and 24.

(2) The rate per hour depends on the range as follows :

0 - 7 and 21- 24 : the rent will be 500 RWF per hour

7-14: and 19h-21: rent will be 1000 RWF per hour

14-19 : the rent will be 1500 RWF per hour

Write a C++ Program that implements the above bicycle rental program

# ASSIGNMENT 1 PROBLEM 2

**Mushrooms**

- Agaric Jaunissant
- Amanite tue-mouche
- Cepe de bordeau
- Coprin chevelu
- Girolle
- Pied bleu

- The cepe bordeaux is the only one to have pores, the other mushrooms having gills.
- Both coprin chevelu and agaric jaunissant grow in meadows, other mushrooms grow in forests.

- The only mushrooms to have a convex cup are agaric jaunissant,amanite tue-mouches, and pied bleu.
- The only mushrooms to have a ring are agaric jaunissant,amanite tue-mouches, and coprin chevelu

# ASSIGNMENT 1 PROBLEM 2 (Cont…)

- Does your mushroom have gills ?
- Does your mushroom grow in a forest ?
- Does your mushroom have a ring?
- Does your mushroom have a convex cup?

- The User thinks of a mushroom (one of the six).
- The program asks at most three of the four questions
- The User answers truthfully by yes or no.
- Based on the user's answers, the program determines the mushroom

Write a C++ Program that implements this scenario.

# THANK YOU

Presenter name
Email address
Website

# CSC2161 C++ PROGRAMMING FUNCTIONS

Dieudonne U

Presentation Title


# AGENDA

Topic two

Topic three

Topic four

# What are functions of C++?

A function is a block of code that performs some operation. A function can optionally define input parameters that enable callers to pass arguments into the function. A function can optionally return a value as output.

# BENEFITS

- MODULARITY (DECMPOSITION)
- READABILITY
- MAINTANABILITY
- REUSABILITY

# FUNCTION DEFINITION

# C++ HAS ONE DEFINITION RULE (ODR): A FUNCTION CAN DECLARED OR INVOKED IN MANY DIFFERENT PLACES, BUT IT IS DEFINED IN ONE PLACE.

```
#include <iostream>
bool isPrime(int);
int main()
{
 int n;
 std::cout << "Enter a positive
number\n";
 std::cin >> n;
for (int i = 2; i < n; ++i)
{
  if (isPrime(i))
    std::cout << i << "\n";
}
return 0;
}
```

```
bool isPrime(int number)
{
  if (number <= 1)
       return false;
  if (number == 2)
       return true;
 for (int i = 2; i < number;
++i)
    if (number % i == 0)
          return false;
 return true;
}
```

# C++ HAS ONE DEFINITION RULE (ODR): A FUNCTION CAN DECLARED OR INVOKED IN MANY DIFFERENT PLACES, BUT IT IS DEFINED IN ONE PLACE.

```
#include <iostream>
bool isPrime(int);
int main()
{
 int n;
 std::cout << "Enter a positive
number\n";
 std::cin >> n;
for (int i = 2; i
{
  if (isPrime(i))
    std::cout << i << "\n";
}
return 0;
}
```

PROTOTYPE

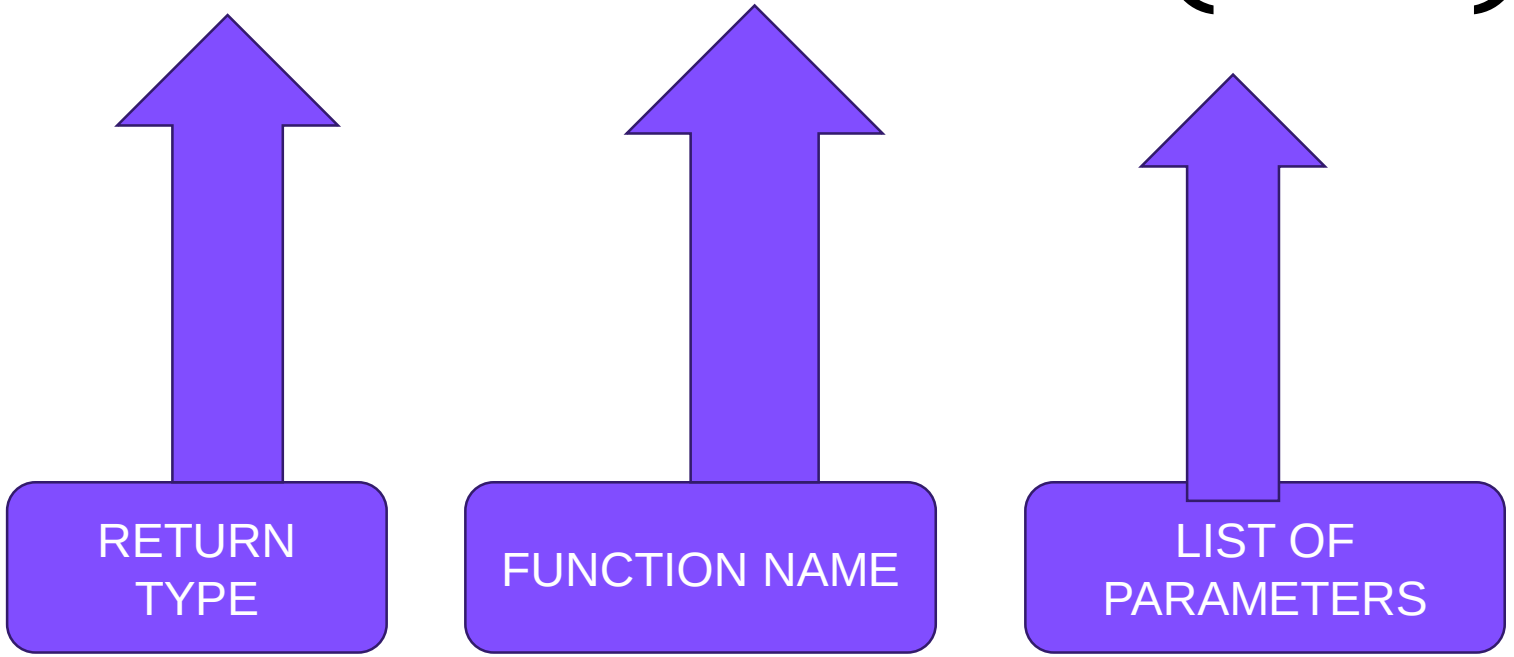
FUNCTION CALL

```
bool isPrime(int number)
{
  if (number <= 1)
      return false;
  if (number == 2)
      return true;
 for (int i = 2; i < number;
++i)
    if (number % i == 0)
        return false;
 return true;
}
```

FUNCTION DEFINITION

# FUNCTION PROTOTYPE : DECLARATION

```
bool isPrime(int);
```




Prototype is required when the function call comes before the compiler sees the function definition.

# FUNCTION DEFINITION

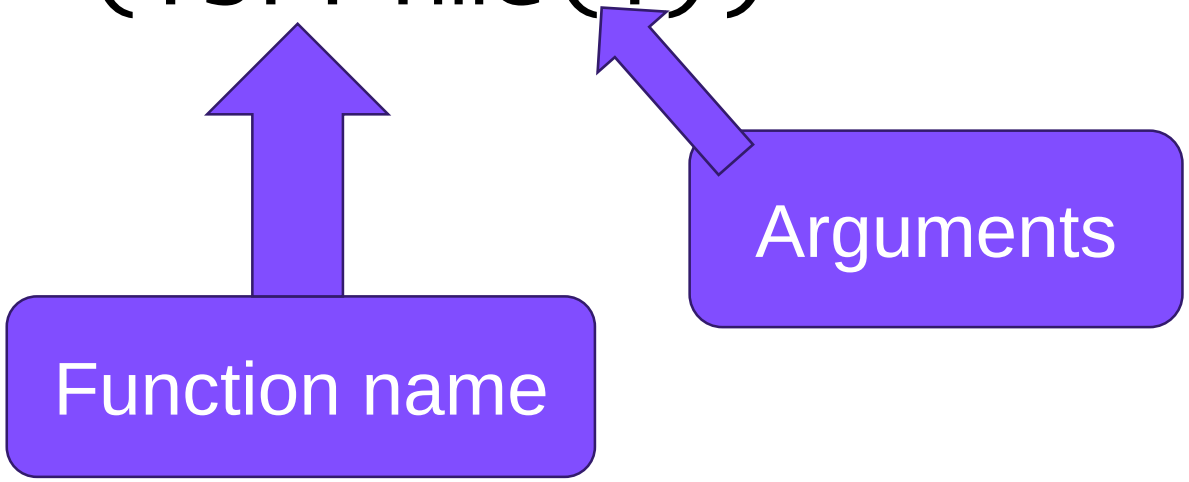
```
bool isPrime(int number)
{
    if (number <= 1)
        return false;
    if (number == 2)
        return true;
    for (int i = 2; i < number; ++i)
        if (number % i == 0)
            return false;
    return true;
}
```

FUNCTION HEADER

FUNCTION BODY

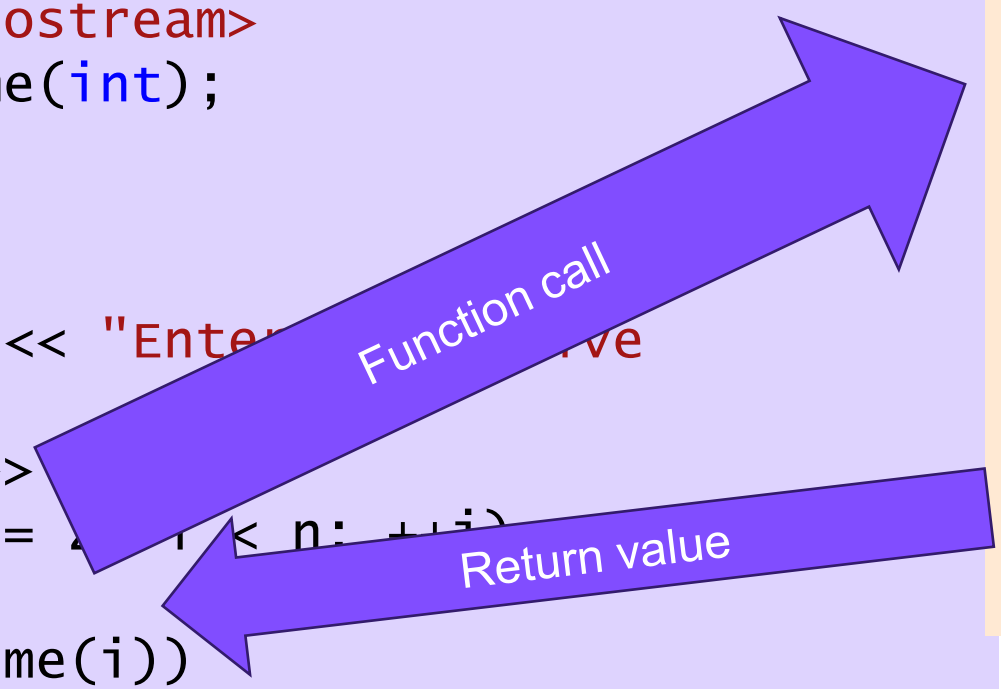
# FUNCTION CALL

```
if (isPrime(i))
```



# C++ HAS ONE DEFINITION RULE (ODR): A FUNCTION CAN DECLARED OR INVOKED IN MANY DIFFERENT PLACES, BUT IT IS DEFINED IN ONE PLACE.

```
#include <iostream>
bool isPrime(int);
int main()
{
 int n;
 std::cout << "Ente                ve
number\n";
 std::cin >>
for (int i =      < n; ++i)
{
  if (isPrime(i))
    std::cout << i << "\n";
}
return 0;
}
```




```
bool isPrime(int number)
{
  if (number <= 1)
      return false;
  if (number == 2)
      return true;
 for (int i = 2; i < number;
++i)
    if (number % i == 0)
        return false;
 return true;
}
```

# Inline function

Function calls involve performance overheads. If the function is simple enough, it might be advantageous to just lay the code of the function where the function is called. That is what happens when a function is inline.

```
inline bool isEven(int n)
{
    return n % 2 == 0;
}
```

Making a function inline is a request to the compiler, it decides to implement it or not.

Compilers are smart enough to make simple functions inline, without the request of the programmer.

# C++ HAS TWO MECHANISMS OF PASSING PARAMETERS TO FUNCTIONS

```
void SwapByValue(int a, int b)
{
   int temp = a;
   a = b;
   b = temp;
}
```

```
void SwapByRef(int& a, int& b)
{
  int temp = a;
  a = b;
  b = temp;
}
```

```
int main()
{
int a = 7, b = 9;
SwapByValue(a, b);
std::cout << "a= " << a << "\n";
std::cout << "b= " << b << "\n";
SwapByRef(a, b);
std::cout << "a= " << a << "\n";
std::cout << "b= " << b << "\n";
    return 0;
}
```

# C++ HAS TWO MECHANISMS OF PASSING PARAMETERS TO FUNCTIONS

```
void SwapByValue(int a, int b)
{
   int temp = a;
   a = b;
   b = temp;
}
```

```
void SwapByRef(int& a, int& b)
{
  int temp = a;
  a = b;
  b = temp;
}
```

```
int main()
{
int a = 7, b = 9;
SwapByValue(a, b);
std::cout << "a= " << a << "\n";
std::cout << "b= " << b << "\n";
SwapByRef(a, b);
std::cout << "a= " << a << "\n";
std::cout << "b= " << b << "\n";
    return 0;
}
```

a= 7
b= 9

a= 9
b= 7

# RECURSIVE FUNCTIONS

Recursion is the technique of making a function call itself. This technique provides a way to break complicated problems down into simple problems which are easier to solve.
Recursion may be a bit difficult to understand. The best way to figure out how it works is to experiment with it.

Example

**Write the output of this program**

```
int sum(int k) {
  if (k > 0) {
    return k + sum(k - 1);
  } else {
    return 0;
  }
}

int main() {
  int result = sum(10);
  cout << result;
  return 0;
}
```

Exercises

1. Write a function to calculate the factorial of a number
2. Write a function for Fibonacci series
3. Write a function to find the sum of elelents in an Array

Presentation Title

# THANK YOU

Presenter name

Email address

Website

# CSC2161
# C++ OOP

Dieudonne


# AGENDA

Topic two

Topic three

Topc four

# What are functions of C++?

The major purp
concept of obje

Object Orientec
concepts such a

The programmi
an object is kno
language.

•

# Class

Collection of objects is called class. It is a logical entity.

A Class in C++ is the foundational element that leads to Object-Oriented programming. A class instance must be created in order to access and use the user-defined data type's data members and member functions. An object's class acts as its blueprint. Take the class of cars as an example. Even if different names and brands may be used for different cars, all of them will have some characteristics in common, such as four wheels, a speed limit, a range of miles, etc. In this case, the class of car is represented by the wheels, the speed limitations, and the mileage.

# Object

Any entity that has state and behavior is known as an object. For example: chair, pen, table, keyboard, bike etc. It can be physical and logical.

# Inheritance

When one object acquires all the properties and behaviours of parent object i.e. known as inheritance. It provides code reusability. It is used to achieve runtime polymorphism.

Sub class - Subclass or Derived Class refers to a class that receives properties from another class.

Super class - The term "Base Class" or "Super Class" refers to the class from which a subclass inherits its properties.

Reusability - As a result, when we wish to create a new class, but an existing class already contains some of the code we need, we can generate our new class from the old class thanks to inheritance. This allows us to utilize the fields and methods of the pre-existing class.

# Polymorphism

When one task is performed by different ways i.e. known as polymorphism. For example: to convince the customer differently, to draw something e.g. shape or rectangle etc.

Different situations may cause an operation to behave differently. The type of data utilized in the operation determines the behavior.

# Abstraction

Hiding internal details and showing functionality is known as abstraction. Data abstraction is the process of exposing to the outside world only the information that is absolutely necessary while concealing implementation or background information.For example: phone call, we don't know the internal processing.

In C++, we use abstract class and interface to achieve abstraction.

# Encapsulation



Binding (or wrapping) code and data together into a single unit is known as encapsulation. For example: capsule, it is wrapped with different medicines.

Encapsulation is typically understood as the grouping of related pieces of information and data into a single entity. Encapsulation is the process of tying together data and the functions that work with it in object-oriented programming. Take a look at a practical illustration of encapsulation: at a company, there are various divisions, including the sales division, the finance division, and the accounts division. All financial transactions are handled by the finance sector, which also maintains records of all financial data. In a similar vein, the sales section is in charge of all tasks relating to sales and maintains a record of each sale.

Now, a scenario could occur when, for some reason, a financial official requires all the information on sales for a specific month. Under the umbrella term "sales section," all of the employees who can influence the sales section's data are grouped together. Data abstraction or concealing is another side effect of encapsulation. In the same way that encapsulation hides the data. In the aforementioned example,  any other area cannot access any of the data from any of the sections, such as sales, finance, or accounts.

# Advantages of OOPs over Procedure-oriented programming language



OOPs makes development and maintenance easier where as in Procedure-oriented programming language it is not easy to manage if code grows as project size grows.

OOPs provide data hiding whereas in Procedure-oriented programming language a global data can be accessed from anywhere.

OOPs provide ability to simulate real-world event much more effectively. We can provide the solution of real word problem if we are using the Object-Oriented Programming language.

# Overloading

Polymorphism also has a subset known as overloading.

An existing operator or function is said to be overloaded when it is forced to operate on a new data type.

# C++ Object and Class



Since C++ is an object-oriented language, program is designed using objects and classes in C++.

# Creating C++ Classes and Objects

In C++, class is a group of similar objects. It is a template from which objects are created. It can have fields, methods, constructors etc.

Let's see an example of C++ class that has three fields only.

```
class Student
 {
    public:
    int id;  //field or data member
    String name;//field or data member
 }
```

```
#include <iostream>
using namespace std;
class Student {
  public:
    int id;//data member (also instance variable)
    string name;//data member(also instance variable)
};
int main() {
   Student s1; //creating an object of Student
   s1.id = 201;
   s1.name = "Sonoo Jaiswal";
   cout<<s1.id<<endl;
   cout<<s1.name<<endl;
   return 0;
}
```

# Initialize and Display data through method

```
#include <iostream>
using namespace std;
class Student {
  public:
     int id;//data member (also instance variable)
     string name;//data member(also instance variable)
     void insert(int i, string n)
     {
        id = i;
        name = n;
     }
     void display()
     {
        cout<<id<<"  "<<name<<endl;
     }
};
```

```
int main(void) {
    Student s1; //creating an object of Student
    Student s2; //creating an object of Student
    s1.insert(201, "Sonoo");
    s2.insert(202, "Nakul");
    s1.display();
    s2.display();
    return 0;
}
```

# C++ Constructor

In C++, constructor is a special method which is invoked automatically at the time of object creation. It is used to initialize the data members of new object generally. The constructor in C++ has the same name as class or structure.

In brief, A particular procedure called a constructor is called automatically when an object is created in C++. In general, it is employed to create the data members of new things. In C++, the class or structure name also serves as the constructor name. When an object is completed, the constructor is called. Because it creates the values or gives data for the thing, it is known as a constructor.

The Constructors prototype looks like this:

<**class**-name> (list-of-parameters);

The following syntax is used to define the class's constructor:

<**class**-name> (list-of-parameters) { // constructor definition }

The following syntax is used to define a constructor outside of a class:

<**class**-name>: :<**class**-name> (list-of-parameters){ // constructor definition}

Constructors lack a return type since they don't have a return value.

There can be two types of constructors in C++.

Default constructor

Parameterized constructor

# C++ Default Constructor

A constructor which has no argument is known as default constructor. It is invoked at the time of creating object. Let's see the simple example of C++ default Constructor.

```
#include <iostream>
using namespace std;
class Employee
 {
   public:
        Employee()
        {
            cout<<"Default Constructor Invoked"<<endl;
        }
};
int main(void)
{
    Employee e1; //creating an object of Employee
    Employee e2;
    return 0;
}
```

# C++ Parameterized Constructor

Constructor which has parameters is called parameterized constructor. It is used to provide different values to distinct objects. Let's see the simple example of C++ Parameterized Constructor.

```
// part 1 of 2

#include <iostream>

using namespace std;

class Employee {

  public:

     int id;//data member (also instance variable)

     string name;//data member(also instance variable)

     float salary;

     Employee(int i, string n, float s)

     {

        id = i;

        name = n;

        salary = s;

     }
```

```
//Part 2 of 2

void display()
      {
         cout<<id<<"  "<<name<<"  "<<salary<<endl;
      }
};

int main(void) {
    Employee e1 =Employee(101, "Sonoo", 890000); //cr
eating an object of Employee
    Employee e2=Employee(102, "Nakul", 59000);
    e1.display();
    e2.display();
    return 0;
}
```

# What is a destructor in C++?

A destructor works opposite to constructor; it destructs the objects of classes. It can be defined only once in a class. Like constructors, it is invoked automatically.

A destructor is defined like constructor. It must have same name as class. But it is prefixed with a tilde sign (~).

# C++ Constructor and Destructor Example

```
include <iostream>

using namespace std;

class Employee

 {

   public:

       Employee()

       {

          cout<<"Constructor Invoked"<<endl;

       }

       ~Employee()

       {

          cout<<"Destructor Invoked"<<endl;

       }

};
```

```
int main(void)
{
    Employee e1; //creating an object of Employee
    Employee e2; //creating an object of Employee
    return 0;
}
```

Output

```
Constructor Invoked
Constructor Invoked
Destructor Invoked
Destructor Invoked
```

# C++ this Pointer

In C++ programming, **this** is a keyword that refers to the current instance of the class. There can be 3 main usage of this keyword in C++.

It can be used **to pass current object as a parameter to another method.**

It can be used **to refer current class instance variable.**

It can be used **to declare indexers.**

```
#include <iostream>
using namespace std;
class Employee {
  public:
     int id; //data member (also instance variable)
     string name; //data member(also instance variable)
     float salary;
     Employee(int id, string name, float salary)
     {
         this->id = id;
        this->name = name;
        this->salary = salary;
     }
     void display()
     {
        cout<<id<<"  "<<name<<"  "<<salary<<endl;
     }
};
```

```
int main(void) {
    Employee e1 =Employee(101, "Sonoo", 890000); //creating an object of Employee
    Employee e2=Employee(102, "Nakul", 59000); //creating an object of Employee
    e1.display();
    e2.display();
    return 0;
}
```

Output:

```
101  Sonoo  890000
102  Nakul  59000
```

# Static

In C++, static is a keyword or modifier that belongs to the type not instance. So instance is not required to access the static members. In C++, static can be field, method, constructor, class, properties, operator and event.

**Advantage of C++ static keyword**

**Memory efficient:** Now we don't need to create instance for accessing the static members, so it saves memory. Moreover, it belongs to the type, so it will not get memory each time when instance is created.

# Static Field

A field which is declared as static is called static field. Unlike instance field which gets memory each time whenever you create object, there is only one copy of static field created in the memory. It is shared to all the objects.

It is used to refer the common property of all objects such as rateOfInterest in case of Account, companyName in case of Employee etc.

```
#include <iostream>
using namespace std;
class Account {
  public:
     int accno; //data member (also instance variable)
     string name; //data member(also instance variable)
     static float rateOfInterest;
     Account(int accno, string name)
     {
         this->accno = accno;
        this->name = name;
     }
     void display()
     {
        cout<<accno<< "<<name<< " "<<rateOfInterest<<endl;
     }
};
```

```
float Account::rateOfInterest=6.5;
int main(void) {
    Account a1 =Account(201, "Sanjay"); //creating an object
of Employee
    Account a2=Account(202, "Nakul"); //creating an object of
 Employee
    a1.display();
    a2.display();
    return 0;
}
```

Output:
201 Sanjay 6.5 202 Nakul 6.5

# Access Specifiers



By now, you are quite familiar with the public keyword that appears in all of our class examples:

```
class MyClass {  // The class
  public:       // Access specifier
    // class members goes here
};
```

The public keyword is an access specifier. Access specifiers define how the members (attributes and methods) of a class can be accessed. In the example above, the members are public - which means that they can be accessed and modified from outside the code.

However, what if we want members to be private and hidden from the outside world?

In C++, there are three access specifiers:

public - members are accessible from outside the class

private - members cannot be accessed (or viewed) from outside the class

protected - members cannot be accessed from outside the class, however, they can be accessed in inherited classes. You will learn more about Inheritance later.

# In the following example, we demonstrate the differences between public and private members:

```
class MyClass {
  public:    // Public access specifier
    int x;   // Public attribute
  private:   // Private access specifier
    int y;   // Private attribute
};

int main() {
  MyClass myObj;
  myObj.x = 25;  // Allowed (public)
  myObj.y = 50;  // Not allowed (private)
  return 0;
}
```

If you try to access a private member, an error occurs:

error: y is private

Note: By default, all members of a class are private if you don't specify an access specifier:

Example

```
class MyClass {
  int x;   // Private attribute
  int y;   // Private attribute
};
```

# Struct

In C++, classes and structs are blueprints that are used to create the instance of a class. Structs are used for lightweight objects such as Rectangle, color, Point, etc.

Unlike class, structs in C++ are value type than reference type. It is useful if you have data that is not intended to be modified after creation of struct.

C++ Structure is a collection of different data types. It is similar to the class that holds different types of data.

The Syntax Of Structure

```
struct structure_name
{
    // member declarations.
}
```

In the above declaration, a structure is declared by preceding the **struct keyword** followed by the identifier(structure name). Inside the curly braces, we can declare the member variables of different types. **Consider the following situation:**

```
struct Student
{
   char name[20];
    int id;
    int age;
}
```

In the above case, Student is a structure contains three variables name, id, and age. When the structure is declared, no memory is allocated. When the variable of a structure is created, then the memory is allocated. Let's understand this scenario.

How to create the instance of Structure?

Structure variable can be defined as:

**Student s;**

# How to access the variable of Structure:

The variable of the structure can be accessed by simply using the instance of the structure followed by the dot (.) operator and then the field of the structure.

**For example:**

s.id = 4;

In the above statement, we are accessing the id field of the structure Student by using the **dot(.)** operator and assigns the value 4 to the id field.

**C++ Struct Example**

Let's see a simple example of struct Rectangle which has two data members width and height.

```
#include <iostream>
using namespace std;
 struct Rectangle
{
   int width, height;


 };
int main(void) {
    struct Rectangle rec;
    rec.width=8;
    rec.height=5;
   cout<<"Area of Rectangle is: "<<(rec.width * rec.height)<<endl;
 return 0;
}
```

# Structure v/s Class

| Structure | Class |
| --- | --- |
| If access specifier is not declared explicitly, then by default access specifier will be public. | If access specifier is not declared explicitly, then by default access specifier will be private. |
| Syntax of Structure:<br><br>struct structure_name<br>{<br>// body of the structure.<br>} | Syntax of Class:<br><br>class class_name<br>{<br>// body of the class.<br>} |
| The instance of the structure is known as "Structure variable". | The instance of the class is known as "Object of the class". |

# Inheritence

In C++, inheritance is a process in which one object acquires all the properties and behaviors of its parent object automatically. In such way, you can reuse, extend or modify the attributes and behaviors which are defined in other class.

In C++, the class which inherits the members of another class is called derived class and the class whose members are inherited is called base class. The derived class is the specialized class for the base class.

# Advantage of C++ Inheritance

Code reusability: Now you can reuse the members of your parent class. So, there is no need to define the member again. So less code is required in the class.

# Types Of Inheritance

**C++ supports five types of inheritance:**

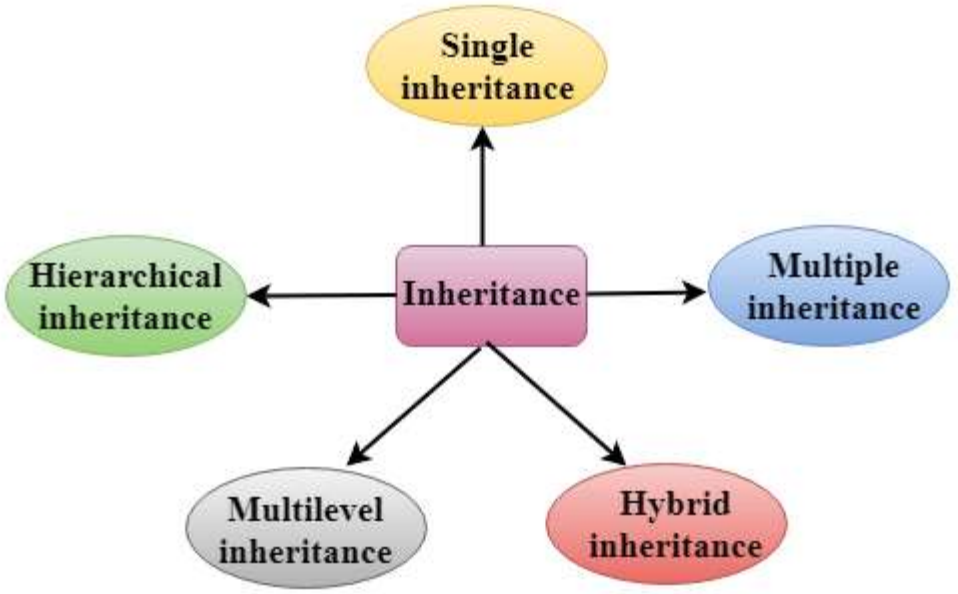
Single inheritance

Multiple inheritance

Hierarchical inheritance

Multilevel inheritance

Hybrid inheritance



# Derived class

A Derived class is defined as the class derived from the base class.

The Syntax of Derived class:

```
class derived_class_name :: visibility-mode base_class_name
{
   // body of the derived class.
}
```

**Where,**

**derived_class_name:** It is the name of the derived class.

**visibility mode:** The visibility mode specifies whether the features of the base class are publicly inherited or privately inherited. It can be public or private.

**base_class_name:** It is the name of the base class.

When the base class is privately inherited by the derived class, public members of the base class becomes the private members of the derived class. Therefore, the public members of the base class are not accessible by the objects of the derived class only by the member functions of the derived class.

When the base class is publicly inherited by the derived class, public members of the base class also become the public members of the derived class. Therefore, the public members of the base class are accessible by the objects of the derived class as well as by the member functions of the base class.

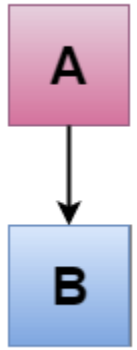
Note:

In C++, the default mode of visibility is private.

The private members of the base class are never inherited.

# C++ Single Inheritance

**Single inheritance** is defined as the inheritance in which a derived class is inherited from the only one base class. Where 'A' is the base class, and 'B' is the derived class.



# C++ Single Level Inheritance Example: Inheriting Fields

When one class inherits another class, it is known as single level inheritance. Let's see the example of single level inheritance which inherits the fields only.

```
#include <iostream>
using namespace std;
 class Employee {
   public:
   float salary = 60000;
 };
   class Programmer: public  Employee  {
   public:
   float bonus = 5000;
   };
int main(void) {
     Programmer p1;
     cout<<"Salary: "<<p1.salary<<endl;
     cout<<"Bonus: "<<p1.bonus<<endl;
    return 0;
}
```

Output:
Salary: 60000
Bonus: 5000

In the above example, Employee is the **base** class and Programmer is the **derived** class.

# C++ Single Level Inheritance Example: Inheriting Methods

```
#include <iostream>

using namespace std;

 class Animal {

   public:

 void eat() {

    cout<<"Eating..."<<endl;

 }

   };
```

```
 class Dog: public Animal
   {
      public:
     void bark(){
    cout<<"Barking...";
     }
    };
int main(void) {
    Dog d1;
    d1.eat();
    d1.bark();
    return 0;
}
```

Output:

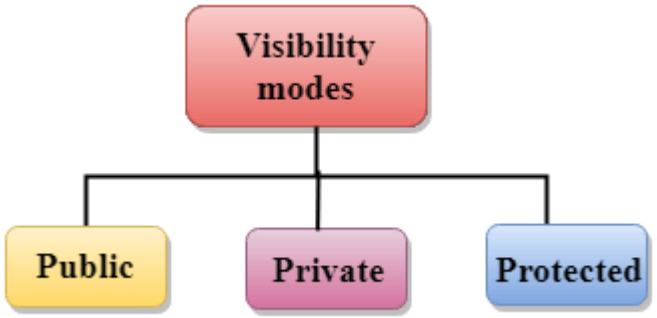
Eating... Barking...

# How to make a Private Member Inheritable

The private member is not inheritable. If we modify the visibility mode by making it public, but this takes away the advantage of data hiding.

C++ introduces a third visibility modifier, i.e., **protected**. The member which is declared as protected will be accessible to all the member functions within the class as well as the class immediately derived from it.

**Visibility modes can be classified into three categories:**




**Public**: When the member is declared as public, it is accessible to all the functions of the program.

**Private**: When the member is declared as private, it is accessible within the class only.

**Protected**: When the member is declared as protected, it is accessible within its own class as well as the class immediately derived from it.

# C++ Multilevel Inheritance

**Multilevel inheritance** is a process of deriving a class from another derived class.

# C++ Multi Level Inheritance Example

When one class inherits another class which is further inherited by another class, it is known as multi level inheritance in C++. Inheritance is transitive so the last derived class acquires all the members of all its base classes.

Let's see the example of multi level inheritance in C++.

```
#include <iostream>
using namespace std;
 class Animal {
   public:
 void eat() {
    cout<<"Eating..."<<en
dl;
 }
   };
```

```
class Dog: public Animal
 {
    public:
  void bark(){
 cout<<"Barking..."<<endl;
  }
};
```

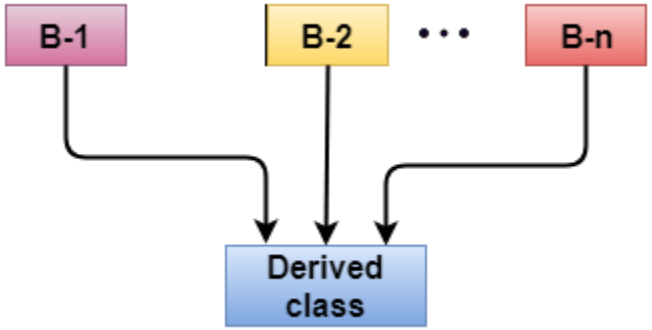
```
class BabyDog: public Dog
  {
     public:
    void weep() {
   cout<<"Weeping...";
    }
  };
 int main(void) {
    BabyDog d1;
    d1.eat();
    d1.bark();
     d1.weep();
     return 0;
}
```

Output:

```
Eating...
Barking...
Weeping...
```

# C++ Multiple Inheritance

**Multiple inheritance** is the process of deriving a new class that inherits the attributes from two or more classes.




**Syntax of the Derived class:**

```
class D : visibility B-1, visibility B-2, ?
{
    // Body of the class;
}
```

```
#include <iostream>
using namespace std;
class A
{
   protected:
    int a;
   public:
   void get_a(int n)
   {
      a = n;
   }
};
```

```
class B
{
   protected:
   int b;
   public:
   void get_b(int n)
   {
      b = n;
   }
};
```

```
class C : public A,public B
{
   public:
    void display()
    {
        std::cout << "The value of a is : " <<a<< std::endl;
        std::cout << "The value of b is : " <<b<< std::endl;
        cout<<"Addition of a and b is : "<<a+b;
    }
};
```

```
 int main()
 {
    C c;
    c.get_a(10);
    c.get_b(20);
    c.display();

     return 0;
 1.}
```

Output:
The value of a is : 10
The value of b is : 20
Addition of a and b is : 30

# Ambiguity Resolution in Inheritance

Ambiguity can be occurred in using the multiple inheritance when a function with the same name occurs in more than one base class.

Let's understand this through an example:

```cpp
#include <iostream>
using namespace std;
class A
{
    public:
    void display()
    {
        std::cout << "Class A" << std::endl;
    }
};
```

```cpp
class B
{
    public:
    void display()
    {
        std::cout << "Class B" << std::en
dl;
    }
};
```

```cpp
class C : public A, public B
{
    void view()
    {
        display();
    }
};
int main()
{
    C c;
    c.display();
    return 0;
}
```

Output:

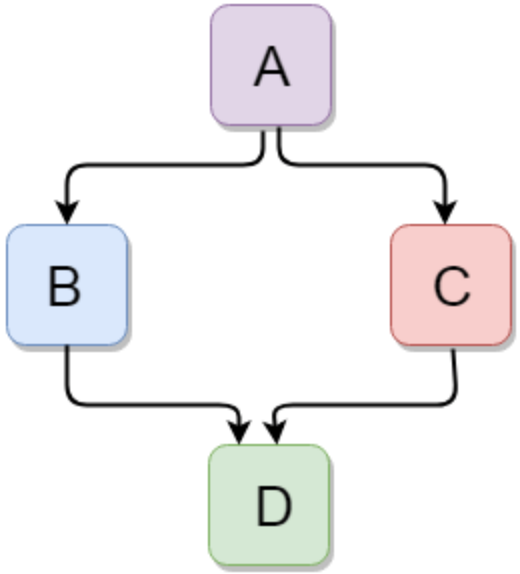
error: reference to 'display' is ambiguous display();

The above issue can be resolved by using the class resolution operator with the function. In the above example, the derived class code can be rewritten as:

```
class C : public A, public B
{
   void view()
   {
      A :: display();       // Calling the display() function of class A.
      B :: display();       // Calling the display() function of class B.

   }
};
```

# C++ Hybrid Inheritence

Hybrid inheritance is a combination of more than one type of inheritance.



# Example

```
#include <iostream>

using namespace std;

class A

{

   protected:

   int a;

   public:

   void get_a()

   {

     std::cout << "Enter the value of 'a' : " << std::endl;

     cin>>a;

   }

};
```

```
class B : public A
{
   protected:
   int b;
   public:
   void get_b()
   {
      std::cout << "Enter the value of 'b' : " << std::endl;
     cin>>b;
   }
};
```

```
class C
{
    protected:
    int c;
    public:
    void get_c()
    {
        std::cout << "Enter the value of c is : " << std::e
ndl;
       cin>>c;
    }
};
```

```
class D : public B, public C
{
    protected:
    int d;
    public:
    void mul()
    {
        get_a();
        get_b();
        get_c();
        std::cout << "Multiplication of a,b,c is : " <<a*b*c<< st
d::endl;
    }
};
```

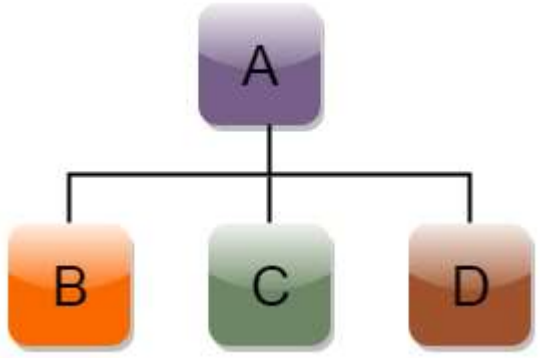
```
int main()
{
    D d;
    d.mul();
    return 0;
}
```

Output:

Enter the value of 'a' : 10 Enter the value of 'b' : 20 Enter the value of c is : 30 Multiplication of a,b,c is : 6000

# C++ Hierarchical Inheritance

Hierarchical inheritance is defined as the process of deriving more than one class from a base class.




**Syntax of Hierarchical inheritance:**

```
class A
{
    // body of the class A.
}
class B : public A
{
    // body of class B.
}
class C : public A
{
    // body of class C.
}
class D : public A
{
    // body of class D.
}
```

```
#include <iostream>
using namespace std;
class Shape                // Declaration of base class.
{
    public:
    int a;
    int b;
    void get_data(int n,int m)
    {
        a= n;
        b = m;
    }
};

class Rectangle : public Shape  // inheriting Shape class

{
    public:
    int rect_area()
    {
        int result = a*b;
        return result;
    }
};
```

```
class Triangle : public Shape    // inheriting Shape class
{
    public:
    int triangle_area()
    {
        float result = 0.5*a*b;
        return result;
    }
};
```

```
int main()
{
    Rectangle r;
    Triangle t;
    int length,breadth,base,height;
    std::cout << "Enter the length and breadth of a rectangle: " << std::endl;
    cin>>length>>breadth;
    r.get_data(length,breadth);
    int m = r.rect_area();
    std::cout << "Area of the rectangle is : " <<m<< std::endl;
    std::cout << "Enter the base and height of the triangle: " << std::endl;
    cin>>base>>height;
    t.get_data(base,height);
    float n = t.triangle_area();
    std::cout <<"Area of the triangle is : "  << n<<std::endl;
    return 0;
}
```

Output:

```
Enter the length and breadth of a rectangle:
23
20
 Area of the rectangle is : 460
Enter the base and height of the triangle:
2
5
Area of the triangle is : 5
```

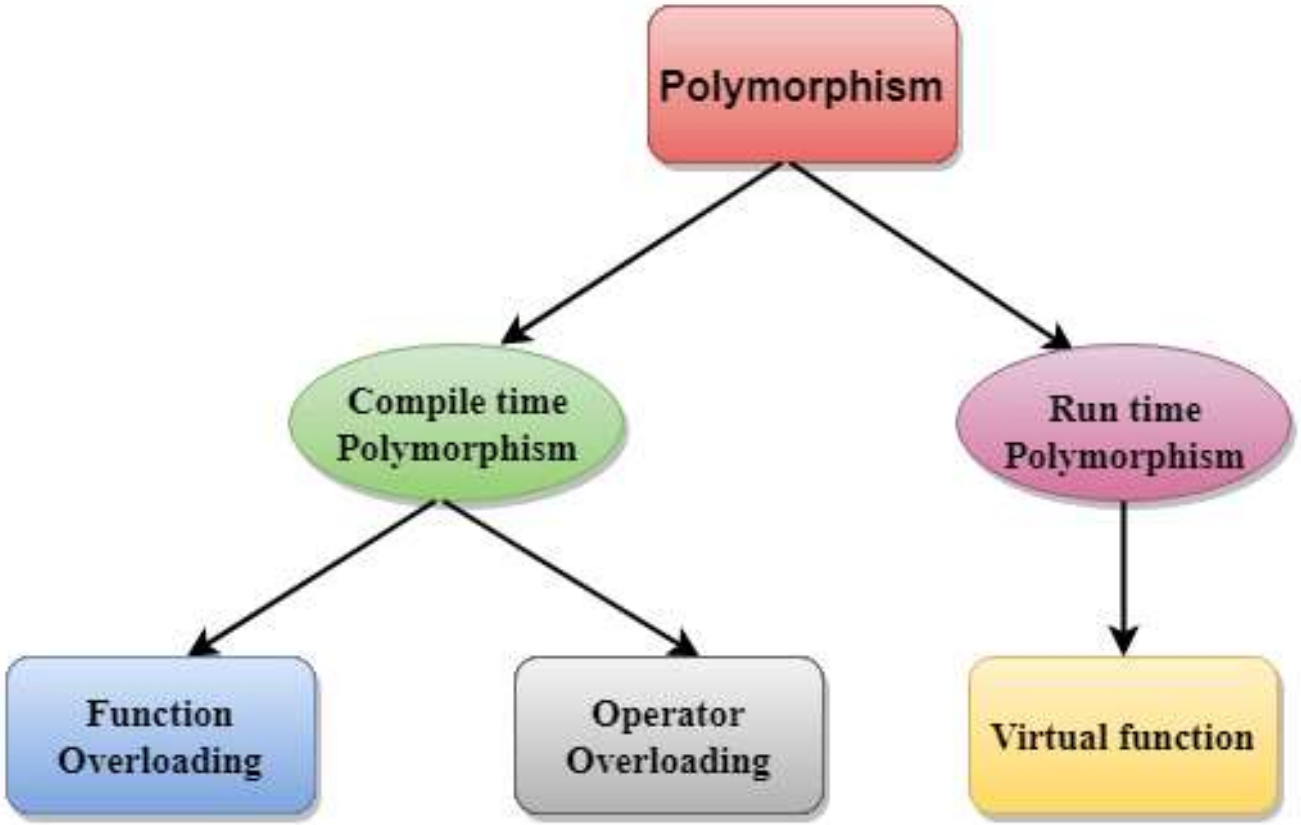
# C++ Polymorphism

The term "Polymorphism" is the combination of "poly" + "morphs" which means many forms. It is a greek word. In object-oriented programming, we use 3 main concepts: inheritance, encapsulation, and polymorphism.

## Real Life Example Of Polymorphism

Let's consider a real-life example of polymorphism. A lady behaves like a teacher in a classroom, mother or daughter in a home and customer in a market. Here, a single person is behaving differently according to the situations.

# There are two types of polymorphism in C++:



# Compile time polymorphism:

The overloaded functions are invoked by matching the type and number of arguments. This information is available at the compile time and, therefore, compiler selects the appropriate function at the compile time. It is achieved by function overloading and operator overloading which is also known as static binding or early binding. Now, let's consider the case where function name and prototype is same.

```
class A                          //  base class declaration.
 {
      int a;
      public:
      void display()
      {
           cout<< "Class A ";
       }
 };
class B : public A                    //  derived class declaration.
{
    int b;
    public:
   void display()
  {
       cout<<"Class B";
  }
};
```

In the above case, the prototype of display() function is the same in both the **base and derived class**. Therefore, the static binding cannot be applied. It would be great if the appropriate function is selected at the run time. This is known as **run time polymorphism**.

# Run time polymorphism

Run time polymorphism is achieved when the object's method is invoked at the run time instead of compile time. It is achieved by method overriding which is also known as dynamic binding or late binding.

# Differences b/w compile time and run time polymorphism.

| Compile time polymorphism | Run time polymorphism |
|---|---|
| The function to be invoked is known at the compile time. | The function to be invoked is known at the run time. |
| It is also known as overloading, early binding and static binding. | It is also known as overriding, Dynamic binding and late binding. |
| Overloading is a compile time polymorphism where more than one method is having the same name but with the different number of parameters or the type of the parameters. | Overriding is a run time polymorphism where more than one method is having the same name, number of parameters and the type of the parameters. |
| It is achieved by function overloading and operator overloading. | It is achieved by virtual functions and pointers. |
| It provides fast execution as it is known at the compile time. | It provides slow execution as it is known at the run time. |
| It is less flexible as mainly all the things execute at the compile time. | It is more flexible as all the things execute at the run time. |

# C++ Runtime Polymorphism Example

Let's see a simple example of run time polymorphism in C++.
// an example without the virtual keyword.

```
#include <iostream>
using namespace std;
class Animal {
   public:
void eat(){
cout<<"Eating...";
   }
};
```

```
class Dog: public Animal
{
 public:
 void eat()  {
          cout<<"Eating bread...";
    }
};
int main(void) {
   Dog d = Dog();
   d.eat();
   return 0;
```

**Output:**
Eating bread...

# C++ Run time Polymorphism Example: By using two derived class

Let's see another example of run time polymorphism in C++ where we are having two derived classes.

// an example with virtual keyword.

```
#include <iostream>
using namespace std;
class Shape {                    //  base class
   public:
virtual void draw(){             // virtual function

cout<<"drawing..."<<endl;
   }
};
```

```
class Rectangle: public Shape //  inherit
ing Shape class.
{
 public:
 void draw()
   {
       cout<<"drawing rectangle..."<<endl;

    }
};
```

```
class Circle: public Shape
 //  inheriting Shape class.

{
 public:
 void draw()
   {
     cout<<"drawing circle..."<<endl;
   }
};
```

```
int main(void) {
    Shape *s;                    //  base class pointer.
    Shape sh;                    // base class object.
    Rectangle rec;
    Circle cir;
      s=&sh;
     s->draw();
        s=&rec;
     s->draw();
    s=&cir
    s->draw();
}
```

**Output:**

drawing...
drawing rectangle...
drawing circle...

# Runtime Polymorphism with Data Members

Runtime Polymorphism can be achieved by data members in C++. Let's see an example where we are accessing the field by reference variable which refers to the instance of derived class.

```
#include <iostream>
using namespace std;
class Animal {       //  base class declarat
ion.
    public:
    string color = "Black";
};
```

```
class Dog: public Animal       // inheriting Animal
class.
{
 public:
    string color = "Grey";
};
int main(void) {
     Animal d= Dog();
    cout<<d.color;
}
```

**Output:**
Black

# Overloading

If we create two or more members having the same name but different in number or type of parameter, it is known as C++ overloading. In C++, we can overload:

methods,

constructors, and

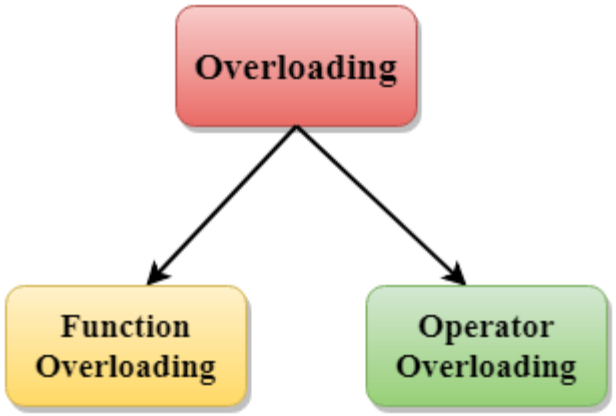
indexed properties

It is because these members have parameters only.

# Types of overloading in C++ are:

Function overloading

Operator overloading



# C++ Function Overloading

Function Overloading is defined as the process of having two or more function with the same name, but different in parameters is known as function overloading in C++. In function overloading, the function is redefined by using either different types of arguments or a different number of arguments. It is only through these differences compiler can differentiate between the functions.

The advantage of Function overloading is that it increases the readability of the program because you don't need to use different names for the same action.

# C++ Function Overloading Example

Let's see the simple example of function overloading where we are changing number of arguments of add() method.

// program of function overloading when number of arguments vary.

```
#include <iostream>
using namespace std;
class Cal {
    public:
static int add(int a,int b){
        return a + b;
    }
static int add(int a, int b, int c)
    {
        return a + b + c;
    }
};
```

```
int main(void) {
    Cal C;
        //     class object declaration.
    cout<<C.add(10, 20)<<endl;
    cout<<C.add(12, 20, 23);
   return 0;
}
```

**Output:**

```
30
55
```

Let's see the simple example when the type of the arguments vary.

// Program of function overloading with different types of arguments.

```
#include<iostream>
using namespace std;
int mul(int,int);
float mul(float,int);


int mul(int a,int b)
{
    return a*b;
}
float mul(double x, int y)
{
    return x*y;
}
```

```
int main()
{
    int r1 = mul(6,7);
    float r2 = mul(0.2,3);
    std::cout << "r1 is : " <<r1<< std::endl;
    std::cout <<"r2 is : "  <<r2<< std::endl;
    return 0;
}
```

**Output:**

```
r1 is : 42
r2 is : 0.6
```

# Function Overloading and Ambiguity

When the compiler is unable to decide which function is to be invoked among the overloaded function, this situation is known as **function overloading**.

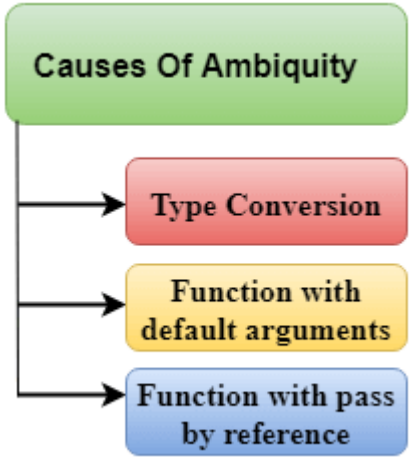
When the compiler shows the ambiguity error, the compiler does not run the program.

**Causes of Function Overloading:**

Type Conversion.

Function with default arguments.

Function with pass by reference.



# Type Conversion

```
#include<iostream>
using namespace std;
void fun(int);
void fun(float);
void fun(int i)
{
   std::cout << "Value of i is : " <<i<< std::endl;
}
void fun(float j)
{
   std::cout << "Value of j is : " <<j<< std::endl;
}
```

```
int main()
{
   fun(12);
   fun(1.2);
   return 0;
}
```

The above example shows an error "**call of overloaded 'fun(double)' is ambiguous**". The fun(10) will call the first function. The fun(1.2) calls the second function according to our prediction. But, this does not refer to any function as in C++, all the floating point constants are treated as double not as a float. If we replace float to double, the program works. Therefore, this is a type conversion from float to double.

Function with Default Arguments

```
#include<iostream>
using namespace std;
void fun(int);
void fun(int,int);
void fun(int i)
{
    std::cout << "Value of i is : " <<i<< std::endl;
}
void fun(int a,int b=9)
{
    std::cout << "Value of a is : " <<a<< std::endl;
    std::cout << "Value of b is : " <<b<< std::endl;
}
int main()
{
    fun(12);

    return 0;
}
```

The above example shows an error "call of overloaded 'fun(int)' is ambiguous". The fun(int a, int b=9) can be called in two ways: first is by calling the function with one argument, i.e., fun(12) and another way is calling the function with two arguments, i.e., fun(4,5). The fun(int i) function is invoked with one argument. Therefore, the compiler could not be able to select among fun(int i) and fun(int a,int b=9).

- Function with pass by reference

```
#include <iostream>
using namespace std;
void fun(int);
void fun(int &);
int main()
{
int a=10;
fun(a); // error, which f()?
return 0;
}
```

```
void fun(int x)
{
std::cout << "Value of x is : " <<x<< std::endl;
}
void fun(int &b)
{
std::cout << "Value of b is : " <<b<< std::endl;
}
```

The above example shows an error "**call of overloaded 'fun(int&)' is ambiguous**". The first function takes one integer argument and the second function takes a reference parameter as an argument. In this case, the compiler does not know which function is needed by the user as there is no syntactical difference between the fun(int) and fun(int &).

# C++ Operators Overloading

Operator overloading is a compile-time polymorphism in which the operator is overloaded to provide the special meaning to the user-defined data type. Operator overloading is used to overload or redefines most of the operators available in C++. It is used to perform the operation on the user-defined data type. For example, C++ provides the ability to add the variables of the user-defined data type that is applied to the built-in data types.

The advantage of Operators overloading is to perform different operations on the same operand.

# Operator that cannot be overloaded are as follows:

Scope operator (::)

Sizeof

member selector(.)

member pointer selector(*)

ternary operator(?:)

Syntax of Operator Overloading

```
return_type class_name  : : operator op(argument_list)
{
    // body of the function.
}
```

Where the **return type** is the type of value returned by the function.

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

**operator op** is an operator function where op is the operator being overloaded, and the operator is the keyword.

Rules for Operator Overloading

Existing operators can only be overloaded, but the new operators cannot be overloaded.

The overloaded operator contains atleast one operand of the user-defined data type.

We cannot use friend function to overload certain operators. However, the member function can be used to overload those operators.

When unary operators are overloaded through a member function take no explicit arguments, but, if they are overloaded by a friend function, takes one argument.

When binary operators are overloaded through a member function takes one explicit argument, and if they are overloaded through a friend function takes two explicit arguments.

```
#include <iostream>
using namespace std;
class Test
{
   private:
      int num;
   public:
      Test(): num(8){}
      void operator ++()        {
         num = num+2;
      }
      void Print() {
          cout<<"The Count is: "<<num;
      }
};
```

```
int main()
{
    Test tt;
    ++tt;  // calling of a function "void operator ++()"
    tt.Print();
    return 0;
}
```

Let's see a simple example of overloading the binary operators.

// program to overload the binary operators.

```
#include <iostream>
using namespace std;
class A
{

    int x;
     public:
     A(){}
    A(int i)
    {
       x=i;
    }
    void operator+(A);
    void display();
};

void A :: operator+(A a)
{

    int m = x+a.x;
     cout<<"The result of the addition of two objects is : "<<
m;

}
int main()
{
    A a1(5);
    A a2(4);
    a1+a2;
    return 0;
}
```

# Function Overriding

If derived class defines same function as defined in its base class, it is known as function overriding in C++. It is used to achieve runtime polymorphism. It enables you to provide specific implementation of the function which is already provided by its base class.

## C++ Function Overriding Example

Let's see a simple example of Function overriding in C++. In this example, we are overriding the eat() function.

```
#include <iostream>
using namespace std;
class Animal {
    public:
void eat(){
cout<<"Eating...";
    }
};
class Dog: public Animal
{
 public:
 void eat()
    {
       cout<<"Eating bread...";
    }
};
int main(void) {
   Dog d = Dog();
   d.eat();
   return 0;
}
```

Output:

Eating bread...

# C++ virtual function

A C++ virtual function is a member function in the base class that you redefine in a derived class. It is declared using the virtual keyword.

It is used to tell the compiler to perform dynamic linkage or late binding on the function.

There is a necessity to use the single pointer to refer to all the objects of the different classes. So, we create the pointer to the base class that refers to all the derived objects. But, when base class pointer contains the address of the derived class object, always executes the base class function. This issue can only be resolved by using the 'virtual' function.

A 'virtual' is a keyword preceding the normal declaration of a function.

When the function is made virtual, C++ determines which function is to be invoked at the runtime based on the type of the object pointed by the base class pointer.

# Late binding or Dynamic linkage

In late binding function call is resolved during runtime. Therefore compiler determines the type of object at runtime, and then binds the function call.

**Rules of Virtual Function**

Virtual functions must be members of some class.

Virtual functions cannot be static members.

They are accessed through object pointers.

They can be a friend of another class.

A virtual function must be defined in the base class, even though it is not used.

The prototypes of a virtual function of the base class and all the derived classes must be identical. If the two functions with the same name but different prototypes, C++ will consider them as the overloaded functions.

We cannot have a virtual constructor, but we can have a virtual destructor

Consider the situation when we don't use the virtual keyword.

```
#include <iostream>
using namespace std;
class A
{
   int x=5;
    public:
    void display()
    {
        std::cout << "Value of x is : " << x<<std::endl;
    }
};
class B: public A
{
    int y = 10;
    public:
    void display()
    {
        std::cout << "Value of y is : " <<y<< std::endl;
    }
};
int main()
{
    A *a;
    B b;
    a = &b;
   a->display();
    return 0;
}
```

**Output:**
Value of x is : 5

In the above example, * a is the base class pointer. The pointer can only access the base class members but not the members of the derived class. Although C++ permits the base pointer to point to any object derived from the base class, it cannot directly access the members of the derived class. Therefore, there is a need for virtual function which allows the base pointer to access the members of the derived class.

# C++ virtual function Example

Let's see the simple example of C++ virtual function used to invoked the derived class in a program

```
#include <iostream>
{
class A{
 public:
 virtual void display()
 {
  cout << "Base class is invoked"<<endl;
 }
};
class B:public A
{
 public:
 void display()
 {
  cout << "Derived Class is invoked"<<endl;
 }
};
int main()
{
 A* a;    //pointer of base class
 B b;     //object of derived class
 a = &b;
 a->display();   //Late Binding occurs
}
```

**Output:**
Derived Class is invoked

# Pure Virtual Function

A virtual function is not used for performing any task. It only serves as a placeholder.

When the function has no definition, such function is known as "**do-nothing**" function.

The "**do-nothing**" function is known as a **pure virtual function**. A pure virtual function is a function declared in the base class that has no definition relative to the base class.

A class containing the pure virtual function cannot be used to declare the objects of its own, such classes are known as abstract base classes.

The main objective of the base class is to provide the traits to the derived classes and to create the base pointer used for achieving the runtime polymorphism.

**Pure virtual function can be defined as:**

**virtual void** display() = 0;

```
#include <iostream>
using namespace std;
class Base
{
    public:
    virtual void show() = 0;
};
class Derived : public Base
{
    public:
    void show()
    {
        std::cout << "Derived class is derived from the base c
lass." << std::endl;
    }
1.};
```

```
int main()
{
    Base *bptr;
    //Base b;
    Derived d;
    bptr = &d;
    bptr->show();
    return 0;
}
```

**Output:**
Derived class is derived from the base class.

In the above example, the base class contains the pure virtual function. Therefore, the base class is an abstract base class. We cannot create the object of the base class.

# Interfaces in C++ (Abstract Classes)

Abstract classes are the way to achieve abstraction in C++. Abstraction in C++ is the process to hide the internal details and showing functionality only. Abstraction can be achieved by two ways:

**Abstract class**

**Interface**

Abstract class and interface both can have abstract methods which are necessary for abstraction.

# C++ Abstract class

In C++ class is made abstract by declaring at least one of its functions as <>strong>pure virtual function. A pure virtual function is specified by placing "= 0" in its declaration. Its implementation must be provided by derived classes.

Let's see an example of abstract class in C++ which has one abstract method draw(). Its implementation is provided by derived classes: Rectangle and Circle. Both classes have different implementation.

```
#include <iostream>
using namespace std;
 class Shape
{
   public:
   virtual void draw()=0;
};
 class Rectangle : Shape
{
   public:
    void draw()
   {
      cout < <"drawing rectangle..." < <endl;
   }
};
```

```
class Circle : Shape
{
    public:
     void draw()
    {
        cout <<"drawing circle..." < <endl;
    }
};
int main( ) {
    Rectangle rec;
    Circle cir;
    rec.draw();
    cir.draw();
   return 0;
}
```

Output:
drawing rectangle... drawing circle...

# Data Abstraction in C++

Data Abstraction is a process of providing only the essential details to the outside world and hiding the internal details, i.e., representing only the essential details in the program.

Data Abstraction is a programming technique that depends on the seperation of the interface and implementation details of the program.

Let's take a real life example of AC, which can be turned ON or OFF, change the temperature, change the mode, and other external components such as fan, swing. But, we don't know the internal details of the AC, i.e., how it works internally. Thus, we can say that AC seperates the implementation details from the external interface.

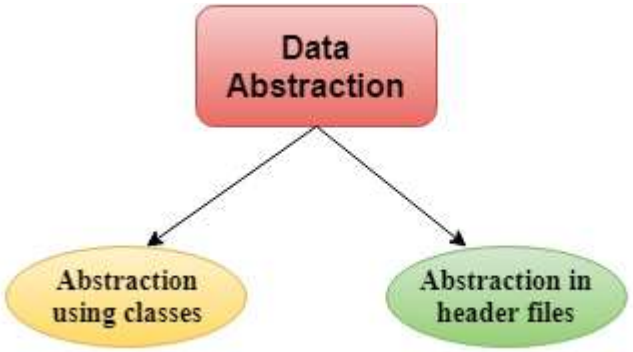
C++ provides a great level of abstraction. For example, pow() function is used to calculate the power of a number without knowing the algorithm the function follows.

In C++ program if we implement class with private and public members then it is an example of data abstraction.

# Data Abstraction can be achieved in two ways:

Abstraction using classes

Abstraction in header files.



**Abstraction using classes:** An abstraction can be achieved using classes. A class is used to group all the data members and member functions into a single unit by using the access specifiers. A class has the responsibility to determine which data member is to be visible outside and which is not.

**Abstraction in header files:** An another type of abstraction is header file. For example, pow() function available is used to calculate the power of a number without actually knowing which algorithm function uses to calculate the power. Thus, we can say that header files hides all the implementation details from the user.

# Access Specifiers Implement Abstraction:

**Public specifier:** When the members are declared as public, members can be accessed anywhere from the program.

**Private specifier:** When the members are declared as private, members can only be accessed only by the member functions of the class.

Let's see a simple example of abstraction in header files.

```
#include <iostream>
#include<math.h>
using namespace std;
int main()
{
 int n = 4;
   int power = 3;
   int result = pow(n,power);        // pow(n,power) is the  power function
   std::cout << "Cube of n is : " <<result<< std::endl;
   return 0;
}
```

In the above example, pow() function is used to calculate 4 raised to the power 3. The pow() function is present in the math.h header file in which all the implementation details of the pow() function is hidden.

```
#include <iostream>
using namespace std;
 class Sum
{
private: int x, y, z; // private variable
s
public:
void add()
{
cout<<"Enter two numbers: ";
cin>>x>>y;
z= x+y;
cout<<"Sum of two number is: "<<z<<endl;
}
};
int main()
{
Sum sm;
sm.add();
return 0;
}
```

Output:

Enter two numbers:

3

6

Sum of two number is: 9

In the above example, abstraction is achieved using classes. A class 'Sum' contains the private members x, y and z are only accessible by the member functions of the class.

# Advantages Of Abstraction:

Implementation details of the class are protected from the inadvertent user level errors.

A programmer does not need to write the low level code.

Data Abstraction avoids the code duplication, i.e., programmer does not have to undergo the same tasks every time to perform the similar operation.

The main aim of the data abstraction is to reuse the code and the proper partitioning of the code across the classes.

Internal implementation can be changed without affecting the user level code.

# Organizing C++ Classes Across Multiple Files: Best Practices for Header and Source Files

In C++, it's common practice to split classes into multiple files to improve code organization, readability, and maintainability. Typically, a class is divided into a header file (with a .h or .hpp extension) and a source file (with a .cpp extension).

# Steps to define and use C++ classes in different files

## 1. Header File (ClassName.h)

The header file contains the class declaration, which includes the class definition, member variables, and member function prototypes (but no actual implementation of the functions).

**MyClass.h**

```
#ifndef MYCLASS_H
#define MYCLASS_H

class MyClass {
public:
   MyClass();          // Constructor
   void display();     // Member function

private:
   int value;          // Private member variable
};

#endif
```

Include Guards (#ifndef, #define, #endif) are used to prevent multiple inclusions of the same header file.The class declaration in the header file defines what the class will look like without providing the actual logic.

Include Guards (#ifndef, #define, #endif) are used to prevent multiple inclusions of the same header file.The class declaration in the header file defines what the class will look like without providing the actual logic.

# 2. Source File (ClassName.cpp)

The source file contains the actual implementation of the class's member functions. You include the corresponding header file here so that the compiler knows the class structure.

```
#include "MyClass.h"
#include <iostream>

MyClass::MyClass() : value(0) {
    // Constructor implementation
}

void MyClass::display() {
    std::cout << "Value: " << value << std::endl;
}
```

Include the header file (#include "MyClass.h") to inform the compiler about the class declaration.Each function is prefixed with ClassName:: to indicate that the function belongs to the MyClass class.

# 3. Main Program File (main.cpp)

```
#include "MyClass.h"


int main() {
    MyClass obj;    // Create object of MyClass
    obj.display();  // Call the display function
    return 0;
}
```

The main program only needs to include the header file, not the .cpp file, because the implementation details are taken care of during the linking process after compilation.

# Compilation

If you're compiling these files manually, you can use a command like the following:

g++ main.cpp MyClass.cpp -o program

This compiles both the main program and the class implementation and links them into a single executable named program.

## Summary of File Structure

MyClass.h: Contains the class declaration (interface).
MyClass.cpp: Contains the implementation of the class methods.
main.cpp: The program that uses the class.

# Benefits of This Structure

**Separation of Concerns**: The header file describes the "what" (interface), while the source file describes the "how" (implementation).

**Reusability:** Header files can be reused in multiple projects or modules without copying code.Reduced

**Compilation Time:** By splitting files, changes to the implementation file don’t require recompiling files that only use the header file.

# C++ Pointers

# Pointers

The pointer in C++ language is a variable, it is also known as locator or indicator that points to an address of a value.

The symbol of an address is represented by a pointer. In addition to creating and modifying dynamic data structures, they allow programs to emulate call-by-reference. One of the principal applications of pointers is iterating through the components of arrays or other data structures. The pointer variable that refers to the same data type as the variable you're dealing with has the address of that variable set to it (such as an int or string).

- **Syntax**

datatype *var_name;

**int** *ptr;  // ptr can point to an address which holds int data

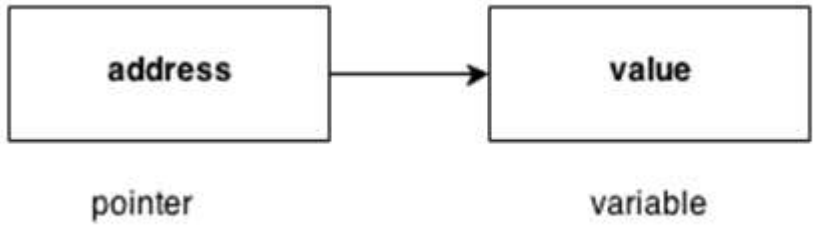
# How to use a pointer?

**Establish a pointer variable.**

employing the unary operator (&), which yields the address of the variable, to assign a pointer to a variable's address.

Using the unary operator (*), which gives the variable's value at the address provided by its argument, one can access the value stored in an address.

Since the data type knows how many bytes the information is held in, we associate it with a reference. The size of the data type to which a pointer points is added when we increment a pointer.



# Advantage of pointer

1) Pointer reduces the code and improves the performance, it is used to retrieving strings, trees etc. and used with arrays, structures and functions.

2) We can return multiple values from function using pointer.

3) It makes you able to access any memory location in the computer's memory.

# Usage of pointer

There are many usage of pointers in C++ language.

**1) Dynamic memory allocation**

In c language, we can dynamically allocate memory using malloc() and calloc() functions where pointer is used.

**2) Arrays, Functions and Structures**

Pointers in c language are widely used in arrays, functions and structures. It reduces the code and improves the performance.

# Symbols used in pointer

| Symbol | Name | Description |
|---|---|---|
| & (ampersand sign) | Address operator | Determine the address of a variable. |
| * (asterisk sign) | Indirection operator | Access the value of an address. |

# Declaring a pointer

The pointer in C++ language can be declared using ∗ (asterisk symbol).

**int** ∗ a; //pointer to int

**char** ∗ c; //pointer to char

# Pointer Example

- Let's see the simple example of using pointers printing the address and value.

```
#include <iostream>
using namespace std;
int main()
{
int number=30;
int * p;
p=&number;//stores the address of number variable
cout<<"Address of number variable is:"<<&number<<endl;
cout<<"Address of p variable is:"<<p<<endl;
cout<<"Value of p variable is:"<<*p<<endl;
  return 0;
}
```

Output:

Address of number variable is:0x7ffccc8724c4
Address of p variable is:0x7ffccc8724c4
Value of p variable is:30

# Pointer Program to swap 2 numbers without using 3rd variable

```
#include <iostream>
using namespace std;
int main()
{
int a=20,b=10,*p1=&a,*p2=&b;
cout<<"Before swap: *p1="<<*p1<<" *p2="<<*p2<<endl;
*p1=*p1+*p2;
*p2=*p1-*p2;
*p1=*p1-*p2;
cout<<"After swap: *p1="<<*p1<<" *p2="<<*p2<<endl;
   return 0;
}
```

Output:

Before swap: *p1=20 *p2=10

After swap: *p1=10 *p2=20

# What is a void pointer?

This unique type of pointer, which is available in C++, stands in for the lack of a kind. Pointers that point to a value that has no type are known as void pointers (and thus also an undetermined length and undetermined dereferencing properties).

This indicates that void pointers are very flexible because they can point to any data type. This flexibility has benefits. Direct dereference is not possible with these pointers.

Before they may be dereferenced, they must be converted into another pointer type that points to a specific data type.

# What is a invalid pointer?

A pointer must point to a valid address, not necessarily to useful items (like for arrays). We refer to these as incorrect pointers. Additionally, incorrect pointers are uninitialized pointers.

```
int *ptr1;
int arr[10];
int *ptr2 = arr+20;
```

Here, ptr1 is not initialized, making it invalid, and ptr2 is outside of the bounds of arr, making it likewise weak. (Take note that not all build failures are caused by faulty references.)

# What is a null pointer?

- A null pointer is not merely an incorrect address; it also points nowhere. Here are two ways to mark a pointer as NULL:
- **int** *ptr1 = 0;
- **int** *ptr2 = NULL;

# What is a pointer to a pointer?

In C++, we have the ability to build a pointer to another pointer, which might then point to data or another pointer. The unary operator (*) is all that is needed in the syntax for declaring the pointer for each level of indirection.

**char** a;

**char** *b;

**char** ** c;

a = 'g';

b = &a;

c = &b;

Here b points to a char that stores 'g', and c points to the pointer b.

# What are references and pointers?

- Call-By-Value
- Call-By-Reference with a Pointer Argument
- Call-By-Reference with a Reference Argument

```
#include
using namespace std;
// Pass-by-Value
int square1(int n)
{cout << "address of n1 in square1(): " << &n << "\n";
n *= n;
return n;
}
// Pass-by-Reference with Pointer Arguments
void square2(int* n)
{
cout << "address of n2 in square2(): " << n << "\n";
*n *= *n;
}
```

```
// Pass-by-Reference with Reference Arguments
void square3(int& n)
{

cout << "address of n3 in square3(): " << &n << "\n";
n *= n;
}
void example()
{
    // Call-by-Value
    int n1 = 8;
    cout << "address of n1 in main(): " << &n1 << "\n";
    cout << "Square of n1: " << square1(n1) << "\n";
    cout << "No change in n1: " << n1 << "\n";
```

```
 // Call-by-Reference with Pointer Arguments
    int n2 = 8;
    cout << "address of n2 in main(): " << &n2 << "\n";
    square2(&n2);
    cout << "Square of n2: " << n2 << "\n";
    cout << "Change reflected in n2: " << n2 << "\n";

    // Call-by-Reference with Reference Arguments
    int n3 = 8;
    cout << "address of n3 in main(): " << &n3 << "\n";
    square3(n3);
    cout << "Square of n3: " << n3 << "\n";
    cout << "Change reflected in n3: " << n3 << "\n";
}
// Driver program
int main() { example(); }
```

# Output

```
address of n1 in main(): 0x7fffa7e2de64
address of n1 in square1(): 0x7fffa7e2de4c
Square of n1: 64
No change in n1: 8
address of n2 in main(): 0x7fffa7e2de68
address of n2 in square2(): 0x7fffa7e2de68
Square of n2: 64
Change reflected in n2: 64
address of n3 in main(): 0x7fffa7e2de6c
address of n3 in square3(): 0x7fffa7e2de6c
Square of n3: 64
Change reflected in n3: 64
```

# The C++ Standard Template Library (STL) and File I/O

- The Standard Template Library (STL) is a set of C++ template classes to provide common programming data structures and functions such as lists, stacks, arrays, etc. It is a library of container classes, algorithms, and iterators. It is a generalized library and so, its components are parameterized. Working knowledge of template classes is a prerequisite for working with STL.
- The C++ Standard Template Library (STL) is a collection of algorithms, data structures, and other components that can be used to simplify the development of C++ programs. The STL provides a range of containers, such as vectors, lists, and maps, as well as algorithms for searching, sorting and manipulating data.
- One of the key benefits of the STL is that it provides a way to write generic, reusable code that can be applied to different data types. This means that you can write an algorithm once, and then use it with different types of data without having to write separate code for each type.
- The STL also provides a way to write efficient code. Many of the algorithms and data structures in the STL are implemented using optimized algorithms, which can result in faster execution times compared to custom code.

# Some of the key components of the STL include:

- **Containers:** The STL provides a range of containers, such as vector, list, map, set, and stack, which can be used to store and manipulate data.
- **Algorithms:** The STL provides a range of algorithms, such as sort, find, and binary_search, which can be used to manipulate data stored in containers.
- **Iterators:** Iterators are objects that provide a way to traverse the elements of a container. The STL provides a range of iterators, such as forward_iterator, bidirectional_iterator, and random_access_iterator, that can be used with different types of containers.
- **Function Objects:** Function objects, also known as functors, are objects that can be used as function arguments to algorithms. They provide a way to pass a function to an algorithm, allowing you to customize its behavior.
- **Adapters:** Adapters are components that modify the behavior of other components in the STL. For example, the reverse_iterator adapter can be used to reverse the order of elements in a container.

- By using the STL, you can simplify your code, reduce the likelihood of errors, and improve the performance of your programs.

**STL has 4 components:**

- **Algorithms**
- **Containers**
- **Functors**
- **Iterators**

# 1. Algorithms

- The header algorithm defines a collection of functions specially designed to be used on a range of elements. They act on containers and provide means for various operations for the contents of the containers.
- Algorithm
  - Sorting
  - Searching
  - Important STL Algorithms
  - Useful Array algorithms
  - Partition Operations

- Numeric
  - valarray class
- Containers or container classes store objects and data. There are in total seven standards “first-class” container classes and three container adaptor classes and only seven header files that provide access to these containers or container adaptors.
- Sequence Containers: implement data structures that can be accessed in a sequential manner.
  - vector
  - list
  - deque
  - arrays
  - forward_list( Introduced in C++11)

- Container Adaptors: provide a different interface for sequential containers.
  - queue
  - priority_queue
  - stack
- Associative Containers: implement sorted data structures that can be quickly searched (O(log n) complexity).
  - set
  - multiset
  - map
  - multimap

- Unordered Associative Containers: implement unordered data structures that can be quickly searched
  - unordered_set (Introduced in C++11)
  - unordered_multiset (Introduced in C++11)
  - unordered_map (Introduced in C++11)
  - unordered_multimap (Introduced in C++11)

- **3. Functors**
- The STL includes classes that overload the function call operator. Instances of such classes are called function objects or functors. Functors allow the working of the associated function to be customized with the help of parameters to be passed. **Must Read** – Functors

- **4. Iterators**
- As the name suggests, iterators are used for working on a sequence of values. They are the major feature that allows generality in STL. **Must Read** – Iterators

# Utility Library

- C++ includes a variety of utility libraries that provide functionality ranging from bit-counting to partial function application. These libraries can be broadly divided into two groups:
- language support libraries, and
- general-purpose libraries.

# Advantages of the C++ Standard Template Library (STL):

- **Reusability**: One of the key advantages of the STL is that it provides a way to write generic, reusable code that can be applied to different data types. This can lead to more efficient and maintainable code.
- **Efficient algorithms:** Many of the algorithms and data structures in the STL are implemented using optimized algorithms, which can result in faster execution times compared to custom code.
- **Improved code readability**: The STL provides a consistent and well-documented way of working with data, which can make your code easier to understand and maintain.
- **Large community of users:** The STL is widely used, which means that there is a large community of developers who can provide support and resources, such as tutorials and forums.

# Disadvantages of the C++ Standard Template Library (STL):

- **Learning curve:** The STL can be difficult to learn, especially for beginners, due to its complex syntax and use of advanced features like iterators and function objects.
- **Lack of control**: When using the STL, you have to rely on the implementation provided by the library, which can limit your control over certain aspects of your code.
- **Performance:** In some cases, using the STL can result in slower execution times compared to custom code, especially when dealing with small amounts of data.

# Files and Streams

- In C++ programming we are using the **iostream** standard library, it provides **cin** and **cout** methods for reading from input and writing to output respectively.
- To read and write from a file we are using the standard C++ library called **fstream**.

Let us see the data types define in fstream library is:

| Data Type | Description |
| --- | --- |
| fstream | It is used to create files, write information to files, and read information from files. |
| ifstream | It is used to read information from files. |
| ofstream | It is used to create files and write information to the files. |

# FileStream example: writing to a file

- Let's see the simple example of writing to a text file **testout.txt** using C++ FileStream programming.

```
#include <iostream>
#include <fstream>
using namespace std;
int main () {
  ofstream filestream("testout.txt");
  if (filestream.is_open())
  {
    filestream << "Welcome to CSE.\n";
    filestream << "C++ Course.\n";
    filestream.close();
  }
  else cout <<"File opening is fail.";
  return 0;
}
```

# Output:

The content of a text file testout.txt is set with the data:

Welcome to CSE.

C++  Course.

# C++ FileStream example: reading from a file

- Let's see the simple example of reading from a text file testout.txt using C++ FileStream programming.

```
#include <iostream>
#include <fstream>
using namespace std;
int main () {
  string srg;
  ifstream filestream("testout.txt");
  if (filestream.is_open())
  {
    while ( getline (filestream,srg) )
    {
      cout << srg <<endl;
    }
    filestream.close();
  }
  else {
      cout << "File opening is fail."<<endl;
    }
  return 0;
}
```

# Note

Note: Before running the code a text file named as "testout.txt" is need to be created and the content of a text file is given below:

Welcome to javaTpoint.

C++ Tutorial.

# C++ Read and Write Example

- Let's see the simple example of writing the data to a text file **testout.txt** and then reading the data from the file using C++ FileStream programming.

```
#include <fstream>
#include <iostream>
using namespace std;
int main () {
  char input[75];
  ofstream os;
  os.open("testout.txt");
  cout <<"Writing to a text file:" << endl;
  cout << "Please Enter your name: ";
  cin.getline(input, 100);
  os << input << endl;
  cout << "Please Enter your age: ";
  cin >> input;
  cin.ignore();
  os << input << endl;
  os.close();
  ifstream is;
  string line;
  is.open("testout.txt");
  cout << "Reading from a text file:" << endl;
  while (getline (is,line))
  {
  cout << line << endl;
  }
  is.close();
  return 0;
}
```

# Output

Writing to a text file:

Please Enter your name: Nakul Jain

Please Enter your age: 22

Reading from a text file:  Nakul Jain

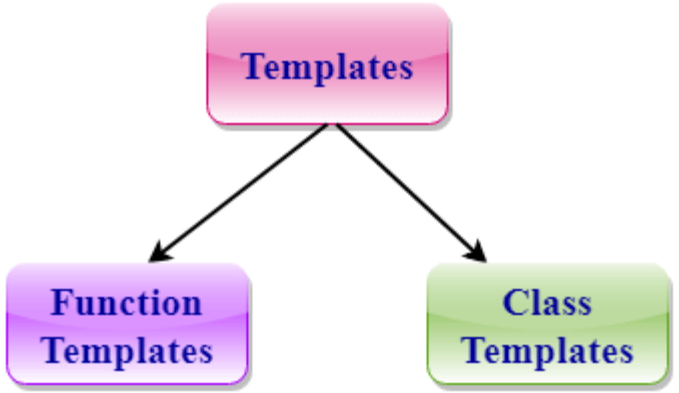
22

# Templates

- A C++ template is a powerful feature added to C++. It allows you to define the generic classes and generic functions and thus provides support for generic programming. Generic programming is a technique where generic types are used as parameters in algorithms so that they can work for a variety of data types.

# Templates can be represented in two ways:

- Function templates
- Class templates



# Function Templates:

- We can define a template for a function. For example, if we have an add() function, we can create versions of the add function for adding the int, float or double type values.

# Class Template:

- We can define a template for a class. For example, a class template can be created for the array class that can accept the array of various types such as int array, float array or double array.

# Function Template

- Generic functions use the concept of a function template. Generic functions define a set of operations that can be applied to the various types of data.
- The type of the data that the function will operate on depends on the type of the data passed as a parameter.
- For example, Quick sorting algorithm is implemented using a generic function, it can be implemented to an array of integers or array of floats.
- A Generic function is created by using the keyword template. The template defines what function will do.

# Syntax of Function Template

```
template < class Ttype> ret_type func_name(parameter_list)
{
    // body of function.
}
```

Where **Ttype**: It is a placeholder name for a data type used by the function. It is used within the function definition. It is only a placeholder that the compiler will automatically replace this placeholder with the actual data type.

# class: A class keyword is used to specify a generic type in a template declaration.

```
 #include <iostream>
using namespace std;
template<class T> T add(T &a,T &b)
{
    T result = a+b;
    return result;

}
```

```
int main()
{
  int i =2;
  int j =3;
  float m = 2.3;
  float n = 1.2;
  cout<<"Addition of i and j is :"<<add(i,j);
  cout<<'\n';
  cout<<"Addition of m and n is :"<<add(m,n);
  return 0;
}
```

**Output:**

Addition of i and j is :5
Addition of m and n is :3.5

- In the above example, we create the function template which can perform the addition operation on any type either it can be integer, float or double.

# Function Templates with Multiple Parameters

We can use more than one generic type in the template function by using the comma to separate the list.

Syntax

```
 template<class T1, class T2,.....>
return_type function_name (arguments of type T1, T2....)
{
    // body of function.
}
```

In the above syntax, we have seen that the template function can accept any number of arguments of a different type.

# Let's see a simple example:

```
#include <iostream>
using namespace std;
template<class X,class Y> void fun(X a,Y b)
{
    std::cout << "Value of a is : " <<a<< std::endl;
    std::cout << "Value of b is : " <<b<< std::endl;
}
int main()
{
   fun(15,12.3);
   return 0;
}
```

In the above example, we use two generic types in the template function, i.e., X and Y.

**Output:**

```
Value of a is : 15
Value of b is : 12.3
```

# Overloading a Function Template

We can overload the generic function means that the overloaded template functions can differ in the parameter list.

- **Let’s understand this through a simple example:**

```
#include <iostream>
using namespace std;
template<class X> void fun(X a)
{
   std::cout << "Value of a is : " <<a<< std::endl;
}
template<class X,class Y> void fun(X b ,Y c)
{
   std::cout << "Value of b is : " <<b<< std::endl;
   std::cout << "Value of c is : " <<c<< std::endl;
}
int main()
{
  fun(10);
  fun(20,30.5);
  return 0;
}
```

```
Value of a is : 10
Value of b is : 20
Value of c is : 30.5
```

- In the above example, template of fun() function is overloaded.

# Restrictions of Generic Functions

- Generic functions perform the same operation for all the versions of a function except the data type differs. Let's see a simple example of an overloaded function which cannot be replaced by the generic function as both the functions have different functionalities.

# Let's understand this through a simple example:

```
#include <iostream>
using namespace std;
void fun(double a)
{
   cout<<"value of a is : "<<a<<'\n';
}
void fun(int b)
{
   if(b%2==0)
   {
      cout<<"Number is even";
   }
   else
   {
      cout<<"Number is odd";
   }
}
int main()
{
   fun(4.6);
   fun(6);
   return 0;
}
```

**Output:**

value of a is : 4.6
Number is even

- In the above example, we overload the ordinary functions. We cannot overload the generic functions as both the functions have different functionalities. First one is displaying the value and the second one determines whether the number is even or not.

# CLASS TEMPLATE

- **Class Template** can also be defined similarly to the Function Template. When a class uses the concept of Template, then the class is known as generic class.

Syntax

```
template<class Ttype>
class class_name
{
  .
  .
}
```

- **Ttype** is a placeholder name which will be determined when the class is instantiated. We can define more than one generic data type using a comma-separated list. The Ttype can be used inside the class body.
- Now, we create an instance of a class
- class_name<type> ob;
- **where class_name**: It is the name of the class.
- **type**: It is the type of the data that the class is operating on.
- **ob**: It is the name of the object.

# Example

```
#include <iostream>
using namespace std;
template<class T>
class A
{
   public:
   T num1 = 5;
   T num2 = 6;
   void add()
   {
      std::cout << "Addition of num1 and num2 : " << num1+num2<<std::endl;
   }

};
```

```
int main()
{
   A<int> d;
   d.add();
   return 0;
}
```

**Output:**

Addition of num1 and num2 : 11

- In the above example, we create a template for class A. Inside the main() method, we create the instance of class A named as, 'd'.

# CLASS TEMPLATE WITH MULTIPLE PARAMETERS

- We can use more than one generic data type in a class template, and each generic data type is separated by the comma.

Syntax

```
template<class T1, class T2, ......>
class class_name
{
   // Body of the class.
}
```

# Let's see a simple example when class template contains two generic data types.

```
#include <iostream>
    using namespace std;
    template<class T1, class T2>
   class A
   {
       T1 a;
       T2 b;
       public:
      A(T1 x,T2 y)
     {
        a = x;
        b = y;
      }
      void display()
           {
                 std::cout << "Values of a and b are : " << a<<" ,"<<
b<<std::endl;
           }
     };

     int main()
    {
         A<int,float> d(5,6.5);
         d.display();
         return 0;
    }
```

**Output:**

Values of a and b are : 5,6.5

# Nontype Template Arguments

- The template can contain multiple arguments, and we can also use the non-type arguments In addition to the type T argument, we can also use other types of arguments such as strings, function names, constant expression and built-in types.

**Let' s see the following example:**

```
template<class T, int size>
class array
{
        T arr[size];          // automatic array initialization.
};
```

In the above case, the nontype template argument is size and therefore, template supplies the size of the array as an argument.

Arguments are specified when the objects of a class are created:

```
array<int, 15> t1;          // array of 15 integers.
array<float, 10> t2;        // array of 10 floats.
array<char, 4> t3;          // array of 4 chars.
```

- Let's see a simple example of nontype template arguments.

```
#include <iostream>
using namespace std;
template<class T, int size>
class A
{
   public:
   T arr[size];
   void insert()
   {
      int i =1;
      for (int j=0;j<size;j++)
      {
         arr[j] = i;
         i++;
      }
   }
   void display()
   {
      for(int i=0;i<size;i++)
      {
         std::cout << arr[i] << " ";
      }
   }
};
int main()
{
   A<int,10> t1;
   t1.insert();
   t1.display();
   return 0;
}
```

Output:

1 2 3 4 5 6 7 8 9 10

In the above example, the class template is created which contains the nontype template argument, i.e., size. It is specified when the object of class 'A' is created.

# Points to remember

- C++ supports a powerful feature known as a template to implement the concept of generic programming.
- A template allows us to create a family of classes or family of functions to handle different data types.
- Template classes and functions eliminate the code duplication of different data types and thus makes the development easier and faster.
- Multiple parameters can be used in both class and function template.
- Template functions can also be overloaded.
- We can also use nontype arguments such as built-in or derived data types as template arguments.