C++ CRASH COURSE

ACM – DBIT

Made By,

Grejo Joby

Introduction

What is C++

C++ was developed by Bjarne Stroustrup, as an extension to the C language.

++ stands for the increment operator in C. C++ is an updated version of C.

C++ gives programmers a high level of control over system resources and memory.

The language was updated 3 major times in 2011, 2014, and 2017 to C++11, C++14, and C++17.

Why Use C++

One of the world's most popular programming languages.

Speed - Like C programming, the performance of optimized C++ code is exceptional.

You can use C++ to develop games, desktop apps, operating systems, and so on.

An object-oriented programming language which gives a clear structure to programs and allows code to be reused, lowering development costs.

Portable and can be used to develop applications that can be adapted to multiple platforms.

Fun and easy to learn!

As C++ is close to C# and Java, it makes it easy for programmers to switch to C++ or vice versa

IDE

Anything! Even Notepad will do..

Other IDEs:

Code::Blocks

VS Code

Turbo C++

1.

Basic Syntax

Let’s start with the codes.

Hello World

```
#include <iostream>                              -> Header File Library
using namespace std;   //Optional                -> Will understand later

int main() {                                     -> Main Function
  cout << ”Hello World!”;                        -> Output Statement
  cout << ”This is not a new line!”;
  cout << ”\n This is a new line!”;
  return 0;                                      -> Return
}
```

Print Statement

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

int main() {
      cout << ”My Name is: ”<<“Grejo”<<endl;
      cout << ”This is a new line!\n”;
      cout << ”This is also new line!”;
      return 0;
}
```

Comments

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

int main() {
        cout << ”My Name is: ”<<“Grejo”<<endl;
        //Single Line Comment
        /* Multi
        Line
        Comment */
        return 0;
}
```

Escape Sequences

Escape Sequences	**Characters**
\b	Backspace
\f	Form feed
\n	**Newline**
\r	Return
\t	**Horizontal tab**
\v	Vertical tab
\	**Backslash**
\'	**Single quotation mark**
\"	**Double quotation mark**
\?	Question mark
\0	**Null Character**

Variables

• int	- stores integers (whole numbers), without decimals, such as 123 or -123
• double	- stores floating point numbers, with decimals, such as 19.99 or -19.99
• char	- stores single characters, such as 'a' or 'B'. Char values are surrounded by single quotes
• string	- stores text, such as "Hello World". String values are surrounded by double quotes
• bool	- stores values with two states: true or false

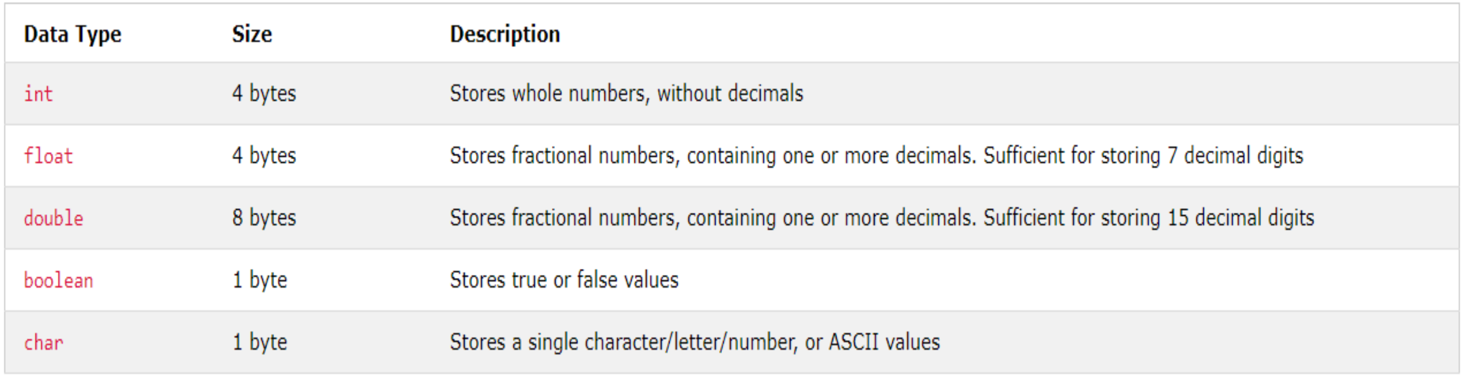
```
int myNum = 15;
cout << myNum;
```

```
int myNum;
myNum = 15;
cout << myNum;
```

```
int myAge = 35;
cout << "I am " << myAge << " years old.";
```

```
int myNum = 5;               // Integer
float myFloatNum = 5.99;    // Floating point number
double myFloatNum = 5.99;    // Floating point number
char myLetter = 'D';         // Character
string myText = "Hello";     // String (text)
bool myBoolean = true;       // Boolean (true or false)
```

DataTypes



Data Type	Size	Description
`int`	4 bytes	Stores whole numbers, without decimals
`float`	4 bytes	Stores fractional numbers, containing one or more decimals. Sufficient for storing 7 decimal digits
`double`	8 bytes	Stores fractional numbers, containing one or more decimals. Sufficient for storing 15 decimal digits
`boolean`	1 byte	Stores true or false values
`char`	1 byte	Stores a single character/letter/number, or ASCII values

Identifiers

All C++ **variables** must be **identified** with **unique names**. These unique names are called **identifiers**.

Rules of Identifiers:

- Names can contain letters, digits and underscores.
- Names must begin with a letter or an underscore (_)
- Names are case sensitive (myVar and myvar are different variables)
- Names cannot contain whitespaces or special characters like !, #, %, etc.
- Reserved words (like C++ keywords, such as int) cannot be used as names

Constants

Constants cannot be changed or modified.

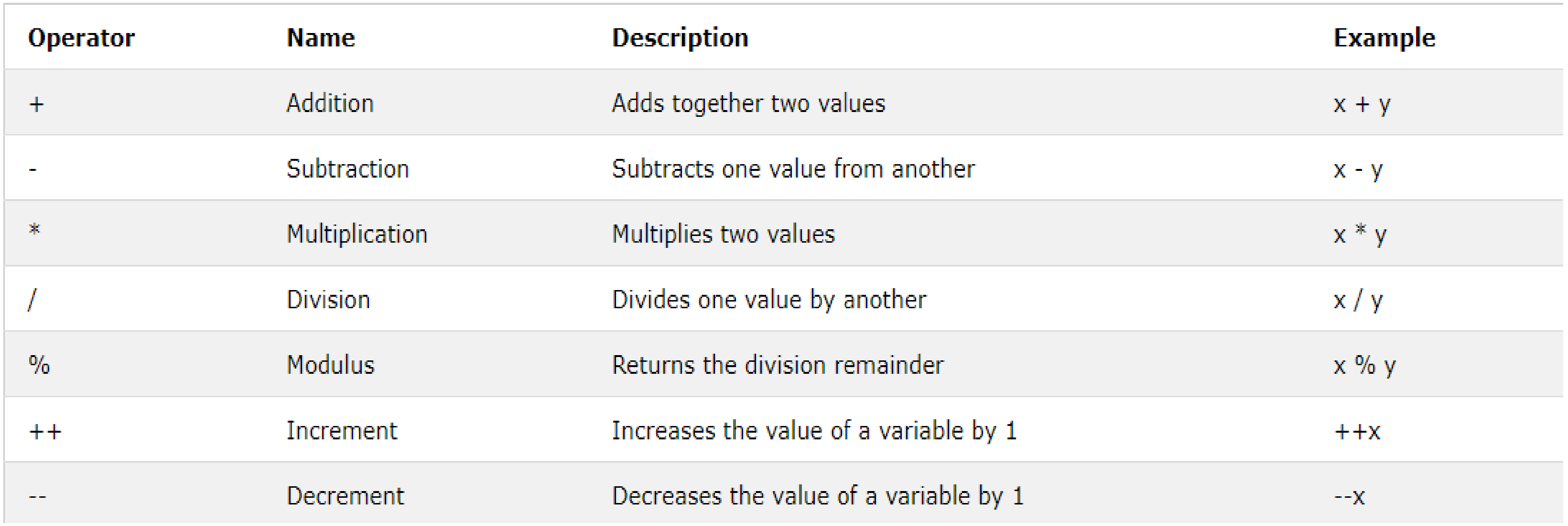
```
const int myNum = 15;  // myNum will always be 15
myNum = 10;  // error: assignment of read-only variable 'myNum’

const int minutesPerHour = 60;
const float PI = 3.14;
```

Input

```
int x;
cout << "Type a number: "; // Type a number and press enter
cin >> x; // Get user input from the keyboard
cout << "Your number is: " << x; // Display the input value
```

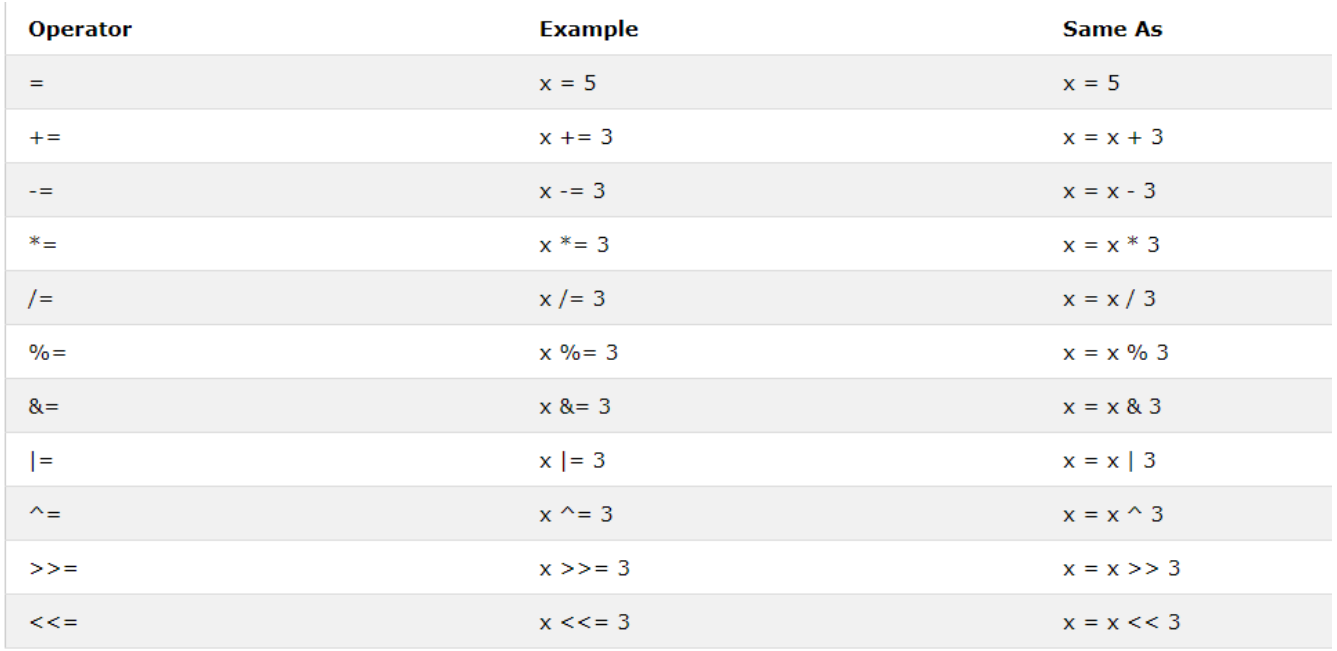
```
int x,y,z;
cout << "Type 3 numbers: ";
cin >> x >> y >> z;
cout << "Your numbers are: " << x << “,”<< y << “,” << z;
```

Operators – Arithmetic



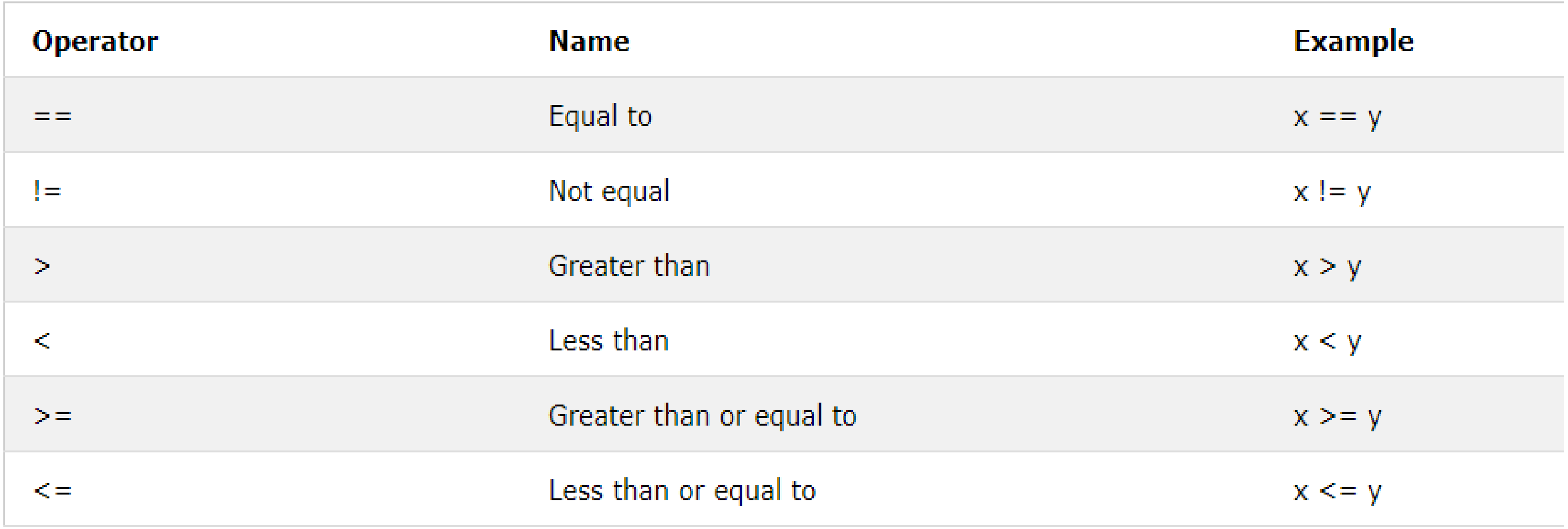
Operator	Name	Description	Example
+	Addition	Adds together two values	x + y
-	Subtraction	Subtracts one value from another	x - y
*	Multiplication	Multiplies two values	x * y
/	Division	Divides one value by another	x / y
%	Modulus	Returns the division remainder	x % y
++	Increment	Increases the value of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

Operators - Assignment



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

Operators – Comparison



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

Operators - Logical

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)

Strings

What is String? It is just an array of characters.

```
string greeting = "Hello";
```

```
char greeting[25] = "Hello";
```

```
// Include the string library
#include <string>

// Create a string variable
string greeting = "Hello";
```

Strings - Operations

```
string firstName = "John";
string lastName = "Doe";
string fullName = firstName + " " + lastName;
cout << fullName;
```

```
string lastName = "Doe";
string fullName = firstName.append(lastName);
cout << fullName;
```

```
string txt = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
cout << "The length of the txt string is: " << txt.length();
```

```
string myString = "Hello";
cout << myString[0];
// Outputs H
```

```
string myString = "Hello";
myString[0] = 'J';
cout << myString;
// Outputs Jello instead of Hello
```

ERROR!

```
string x = "10";
int y = 20;
string z = x + y;
```

Strings – User Input

```
string firstName;
cout << "Type your first name: ";
cin >> firstName; // get user input from the keyboard
cout << "Your name is: " << firstName;

// Type your first name: John
// Your name is: John
```

VS

```
string fullName;
cout << "Type your full name: ";
cin >> fullName;
cout << "Your name is: " << fullName;

// Type your full name: John Doe
// Your name is: John
```

```
string fullName;
cout << "Type your full name: ";
getline (cin, fullName);
cout << "Your name is: " << fullName;

// Type your full name: John Doe
// Your name is: John Doe
```

Math

```
cout << min(5, 10);

cout << max(5, 10);

#include <cmath>

cout << sqrt(64);
cout << round(2.6);
cout << log(2);
```

Some functions in cmath.h

Function	**Description**
abs(x)	Returns the absolute value of x
ceil(x)	Returns the value of x rounded up to its nearest integer
exp(x)	Returns the value of E^x
floor(x)	Returns the value of x rounded down to its nearest integer
pow(x, y)	Returns the value of x to the power of y

Header Files

Below are some inbuilt header files in C/C++:

1.#include<stdio.h> : It is used to perform input and output operations using functions **scanf()** and **printf()**.

2.#include<iostream>: It is used as a stream of Input and Output using cin and cout.

3.#include<string.h>: It is used to perform various functionalities related to string manupulation like strlen(), strcmp(), strcpy(), size(), etc.

4.#include<math.h>: It is used to perform mathematical operations like sqrt(), log2(), pow(), etc.

5.#include<iomanip.h>: It is used to access set() and setprecision() function to limit the decimal places in variables.

6.#include<fstream.h>: It is used to control the data to read from a file as an input and data to write into the file as an output.

7.#include<time.h>: It is used to perform functions related to date() and time() like setdate() and getdate(). It is also used to modify the system date and get the CPU time respectively.

8.#include<ctype.h> : The ctype. h header file of the C Standard Library declares several functions that are useful for testing and mapping characters.

Conditions

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

Short Hand If...Else (Ternary Operator)

variable = (condition) ? expressionTrue : expressionFalse;

```
if (condition) {
  // block of code to be executed if
the condition is true
} else if (condition2) {
  // block of code to be executed if
the condition1 is false and condition2
is true
} else {
  // block of code to be executed if
the condition is false
}
```

Ternary Operator

```
variable = (condition) ? expressionTrue : expressionFalse;
```

```
int time = 20;
if (time < 18) {
  cout << "Good day.";
} else {
  cout << "Good evening.";
}
```

vs

```
int time = 20;
string result = (time < 18) ? "Good day." : "Good evening.";
cout << result;
```

Switch Case

```
switch(expression) {
  case x:
    // code block
    break;
  case y:
    // code block
    break;
  default:
    // code block
}
```

The **break** Keyword

When C++ reaches a break keyword, it breaks out of the switch block.
This will stop the execution of more code and case testing inside the block.

The **default** Keyword

The default keyword specifies some code to run if there is no case match:

Loops

- While
- Do-While
- For

While Loop

```
int i = 0;
while (i < 5) {
  cout << i << "\n";
  i++;
}
```

Checks for condition and then iterates

Do-While Loop

```
int i = 0;
do {
  cout << i << "\n";
  i++;
}
while (i < 5);
```

Itereates then Checks for condition

For Loop

```
for (statement 1; statement 2; statement 3)
{
  // code block to be executed
}
```

Statement 1 is executed (one time) before the execution of the code block.
Statement 2 defines the condition for executing the code block.
Statement 3 is executed (every time) after the code block has been executed.

```
for (int i = 0; i < 5; i++) {
  cout << i << "\n";
}
```

Break & Continue

The break statement can also be used to jump out of a **loop**.

```
for (int i = 0; i < 10; i++) {
  if (i == 4) {
    break;
  }
  cout << i << "\n";
}
```

The continue statement breaks one iteration (in the loop), if a specified condition occurs, and continues with the next iteration in the loop.

```
for (int i = 0; i < 10; i++) {
  if (i == 4) {
    continue;
  }
  cout << i << "\n";
}
```

Goto & Label

When goto label; is encountered, the control of program jumps to label: and executes the code below it.

```
goto label;
... .. ...
... .. ...
... .. ...
label:
statement;
... .. ...
```

Arrays

Array stores multiples values of similar data type.

```
string cars[4];
```

```
string cars[] = {"Volvo", "BMW", "Ford"};
```

```
string cars[4] = {"Volvo", "BMW", "Ford", "Mazda"};
```

```
int myNum[3] = {10, 20, 30};
```

```
string cars[4] = {"Volvo", "BMW", "Ford", "Mazda"};
cout << cars[0];
// Outputs Volvo
```

```
cars[0] = "Opel";
```

```
string cars[4] =
{"Volvo", "BMW", "Ford", "Mazda"};
for(int i = 0; i < 4; i++) {
  cout << cars[i] << "\n";
}
```

```
string cars[4] ;
for(int i = 0; i < 4; i++) {
  cin >> cars[i];
}
```

Multi-D Arrays

```
int x[3][4];
```

	Col 1	Col 2	Col 3	Col 4
Row 1	x[0][0]	x[0][1]	x[0][2]	x[0][3]
Row 2	x[1][0]	x[1][1]	x[1][2]	x[1][3]
Row 3	x[2][0]	x[2][1]	x[2][2]	x[2][3]

```
float x[2][4][3];
```

```
int  test[2][3] = { {2, 4, 5}, {9, 0,
19}};
```

```
// Storing user input in the array
   for (int i = 0; i < 2; ++i) {
      for (int j = 0; j < 3; ++j) {
         cin >> numbers[i][j];
      }
   }
```

References

A reference variable is a "reference" to an existing variable, and it is created with the & operator. It links the variable by memory address.

```
string food = "Pizza";
string &meal = food;

cout << food << "\n";  // Outputs Pizza
cout << meal << "\n";  // Outputs Pizza
```

```
string food = "Pizza";

cout << &food; // Outputs 0x6dfed4
```

Pointers

A **pointer**, is a variable that **stores the memory address as its value**. It can point to any datatype.

```
string food = "Pizza";
string* ptr = &food;    // pointer variable, that
stores the address of food

// Output the value of food (Pizza)
cout << food << "\n";

// Output the memory address of food (0x6dfed4)
cout << &food << "\n";

// Output the memory address of food with the
pointer (0x6dfed4)
cout << ptr << "\n";
```

```
Declarations

string* mystring; // Preferred
string *mystring;
string * mystring;
```

Pointers - Derefrencing

Using pointer (*) to a pointer, dereferences it.

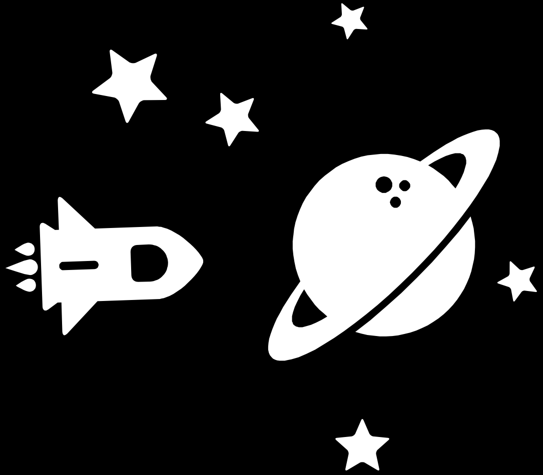
```
string food = "Pizza";  // Variable declaration
string* ptr = &food;    // Pointer declaration

// Reference: Output the memory address of food with the
pointer (0x6dfed4)
cout << ptr << "\n";

// Dereference: Output the value of food with the pointer
(Pizza)
cout << *ptr << "\n";
```

Pointers – Modify Value

```
string food = "Pizza";
string* ptr = &food;

// Output the value of food (Pizza)
cout << food << "\n";

// Output the memory address of food (0x6dfed4)
cout << &food << "\n";

// Access the memory address of food and output its value (Pizza)
cout << *ptr << "\n";

// Change the value of the pointer
*ptr = "Hamburger";

// Output the new value of the pointer (Hamburger)
cout << *ptr << "\n";

// Output the new value of the food variable (Hamburger)
cout << food << "\n";
```

C++ Functions

A function is a block of code which only runs when it is called.



Functions

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

```
// Create a function
void myFunction() {
  cout << "I just got executed!";
}

int main() {
  myFunction(); // call the function
  myFunction();
  return 0;
}

// Outputs "I just got executed!“ 2 times
```

Functions

Declaration vs Definition

Note: If a user-defined function, such as myFunction() is declared after the main() function, **an error will occur**.
It is because C++ works from top to bottom; which means that if the function is not declared above main(), the program is unaware of it:

```cpp
int main() {
  myFunction();
  return 0;
}

void myFunction() {
  cout << "I just got executed!";
}

// Error
```

```cpp
void myFunction();

int main() {
  myFunction();  // call the function
  return 0;
}

// Function definition
void myFunction() {
  cout << "I just got executed!";
}
```

Functions

Parameters and Arguments

```
void functionName(parameter1, parameter2, parameter3) {
  // code to be executed
}
```

```
void myFunction(string fname) {
  cout << fname << " Refsnes\n";
}

int main() {
  myFunction("Liam");
  myFunction("Jenny");
  myFunction("Anja");
  return 0;
}

// Liam Refsnes
// Jenny Refsnes
// Anja Refsnes
```

When a **parameter** is passed to the function,
it is called an **argument**. So, from the example above:
fname is a **parameter**, while Liam, Jenny and Anja are **arguments**.

Functions

Default Parameter

```
void myFunction(string country = "Norway") {
  cout << country << "\n";
}

int main() {
  myFunction("Sweden");
  myFunction("India");
  myFunction();
  myFunction("USA");
  return 0;
}

// Sweden
// India
// Norway
// USA
```

Functions

Multiple Parameters

```
void myFunction(string fname, int age) {
  cout << fname << " Refsnes. " << age << " years old. \n";
}

int main() {
  myFunction("Liam", 3);
  myFunction("Jenny", 14);
  myFunction("Anja", 30);
  return 0;
}

// Liam Refsnes. 3 years old.
// Jenny Refsnes. 14 years old.
// Anja Refsnes. 30 years old.
```

Note that when you are working with multiple parameters, the function call must have the same number of arguments as there are parameters, and the arguments must be passed in the same order.

Try : Combine default parameter with multiple parameters

Functions

Return Statement

```
int myFunction(int x, int y) {
  return x + y;
}

int main() {
  cout << myFunction(5, 3);
  return 0;
}

// Outputs 8 (5 + 3)
```

Functions

Pass by reference

```
void swapNums(int &x, int &y) {
  int z = x;
  x = y;
  y = z;
}
int main() {
  int firstNum = 10;
  int secondNum = 20;
  cout << "Before swap: " << "\n";
  cout << firstNum << secondNum << "\n";

  // Call the function, which will change the values of firstNum and secondNum
  swapNums(firstNum, secondNum);
  cout << "After swap: " << "\n";
  cout << firstNum << secondNum << "\n";

  return 0;
}
```

Arrays are always passed as reference. Only the first location is passed.

Functions

Function Overloading

```
int plusFuncInt(int x, int y) {
  return x + y;
}

double plusFuncDouble(double x, double y) {
  return x + y;
}

int main() {
  int myNum1 = plusFuncInt(8, 5);
  double myNum2 =
plusFuncDouble(4.3, 6.26);
  cout << "Int: " << myNum1 << "\n";
  cout << "Double: " << myNum2;
  return 0;
}
```

```
int plusFunc(int x, int y) {
  return x + y;
}

double plusFunc(double x, double y) {
  return x + y;
}

int main() {
  int myNum1 = plusFunc(8, 5);
  double myNum2 = plusFunc(4.3, 6.26);
  cout << "Int: " << myNum1 << "\n";
  cout << "Double: " << myNum2;
  return 0;
}
```

Multiple functions can have the same name as long as the number and/or type of parameters are different.

Functions

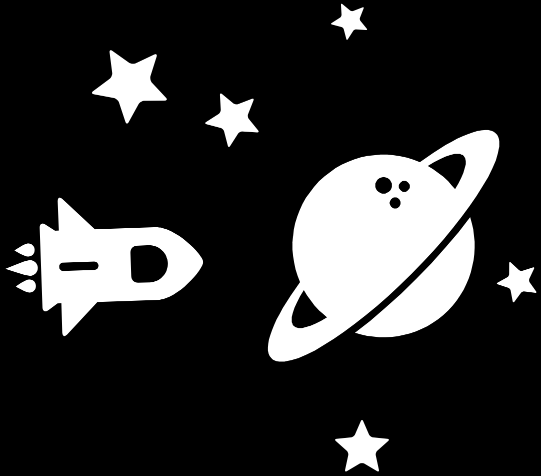
Recursion

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

int factorial(int);

int main() {
    int n, result;

    cout << "Enter a non-negative number: ";
    cin >> n;

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

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

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

C++ Structures



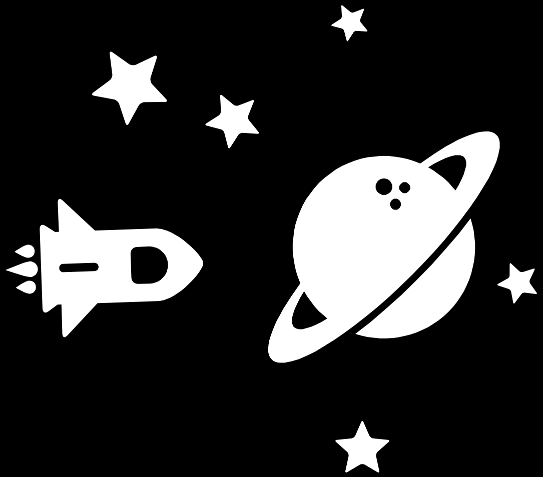
Structures

Structure is a collection of variables of different data types under a single name.
It is similar to a class, but class can have functions and structure cannot.
The struct keyword defines a structure type followed by an identifier (name of the structure).

```
struct Person
{
   char name[50];
   int age;
   float salary;
};
```

```
#include <iostream>
using namespace std;
struct Person
{
   char name[50];
   int age;
   float salary;
};
int main()
{
   Person p1;

   cout << "Enter Full name: ";
   cin.get(p1.name, 50);
   cout << "Enter age: ";
   cin >> p1.age;
   cout << "Enter salary: ";
   cin >> p1.salary;

   cout << "\nDisplaying Information." << endl;
   cout << "Name: " << p1.name << endl;
   cout <<"Age: " << p1.age << endl;
   cout << "Salary: " << p1.salary;

   return 0;
}
```

C++
OOPS



Object-Oriented Programming.

Procedural programming is about writing procedures or functions that perform operations on the data, while object-oriented programming is about creating objects that contain both data and functions.

OOPS

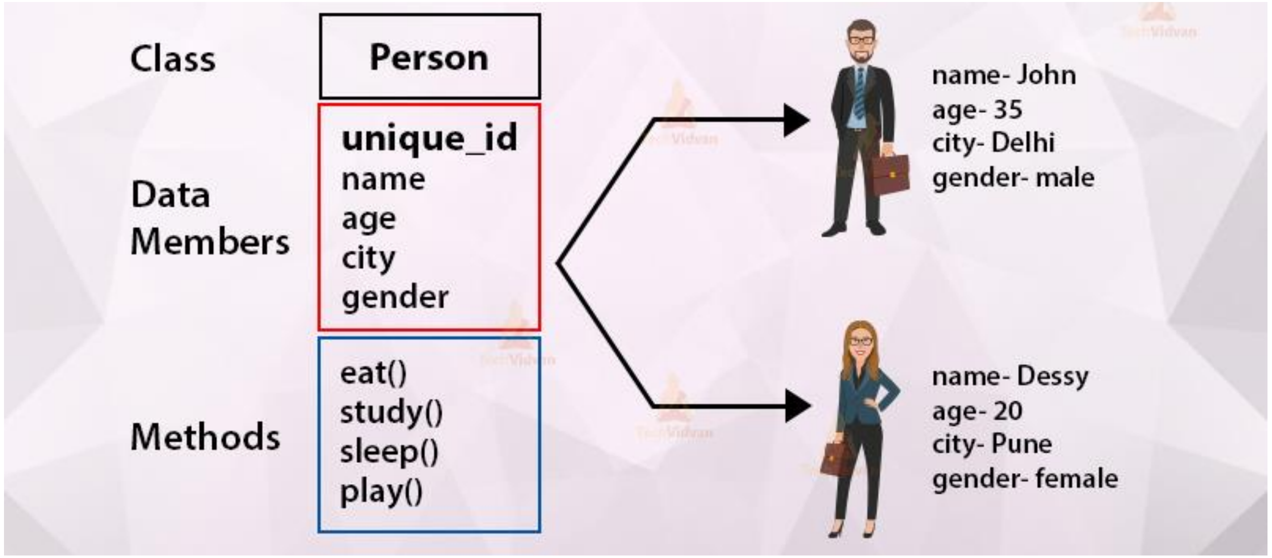
Advantages

- Faster and easier to execute
- Provides a clear structure for the programs
- Reduces Repeated Codes
- Reusability of code hence reduces development time
- Security (via Abstraction)
- Better productivity
- Easier Maintenance of Code
- Easy to expand your code

OOPS

Classes and Objects

Most Important Part of OOPS!!!



class is a template for objects, and an object is an instance of a class.

OOPS

Classes and Objects

The 'this' pointer is a constant pointer and it holds the memory address of the current object.

```
class Car {
  public:
    string brand;
    string model;
    int year;
};

int main() {
  // Create an object of Car
  Car carObj1;
  carObj1.brand = "BMW";
  carObj1.model = "X5";
  carObj1.year = 1999;

  // Create another object of Car
  Car carObj2;
  carObj2.brand = "Ford";
  carObj2.model = "Mustang";
  carObj2.year = 1969;

  // Print attribute values
  cout << carObj1.brand << " " << carObj1.model << " " <<
carObj1.year << "\n";
  cout << carObj2.brand << " " << carObj2.model << " " <<
carObj2.year << "\n";
  return 0;
}
```

class is a template for objects, and an object is an instance of a class.

OOPS

Class Methods

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

class Car {
  public:
    int speed(int maxSpeed);
};

int Car::speed(int maxSpeed) {
  return maxSpeed;
}

int main() {
  Car myObj; // Create an object of Car
  cout << myObj.speed(200); // Call the method with an argument
  return 0;
}
```

class is a template for objects, and an object is an instance of a class.

OOPS

Constructor

```
class MyClass {      // The class
  public:            // Access specifier
    MyClass() {      // Constructor
      cout << "Hello World!";
    }
};

int main() {
  MyClass myObj;     // Create an object
of MyClass (this will call the
constructor)
  return 0;
}
```

The constructor has the same name as the class, it is always public, and it does not have any return value.

```
class Car {          // The class
  public:            // Access specifier
    string brand;    // Attribute
    string model;    // Attribute
    int year;        // Attribute
    Car(string x, string y, int z) {
// Constructor with parameters
      brand = x;
      model = y;
      year = z;
    }
};

int main() {
  // Create Car objects and call the constructor with
different values
  Car carObj1("BMW", "X5", 1999);
  Car carObj2("Ford", "Mustang", 1969);

  // Print values
  cout << carObj1.brand << " " << carObj1.model << " " <<
carObj1.year << "\n";
  cout << carObj2.brand << " " << carObj2.model << " " <<
carObj2.year << "\n";
  return 0;
}
```

OOPS

Copy Constructor

A copy constructor is a member function which initializes an object using another object of the same class. A copy constructor has the following general function prototype:

```
ClassName (const ClassName &old_obj);
```

Syntax:

Point p2 = p1;
Point p2(p1);

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

class Point
{
private:
    int x, y;
public:
    Point(int x1, int y1) { x = x1; y = y1; }

    // Copy constructor
    Point(const Point &p2) {x = p2.x; y = p2.y; }

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

int main()
{
    Point p1(10, 15); // Normal constructor is called here
    Point p2 = p1; // Copy constructor is called here

    // Let us access values assigned by constructors
    cout << "p1.x = " << p1.getX() << ", p1.y = " << p1.getY();
    cout << "\np2.x = " << p2.getX() << ", p2.y = " << p2.getY();

    return 0;
}
```

OOPS

Destructor

- Destructor is a member function which destructs or deletes an object.

When is destructor called?
A destructor function is called automatically when the object goes out of scope:
(1) the function ends
(2) the program ends
(3) a block containing local variables ends
(4) a delete operator is called

How destructors are different from a normal member function?

- Destructors have same name as the class preceded by a tilde (~)
- Destructors don't take any argument and don't return anything

```
class String
{
private:
    char *s;
    int size;
public:
    String(char *); // constructor
    ~String();      // destructor
};

String::String(char *c)
{
    size = strlen(c);
    s = new char[size+1];
    strcpy(s,c);
}

String::~String()
{
    delete []s;
}
```

When do we need to write a user-defined destructor?
Compiler creates a default destructor for us. The default destructor works fine unless we have dynamically allocated memory or pointer in class. When a class contains a pointer to memory allocated in class, we should write a destructor to release memory before the class instance is destroyed. This must be done to avoid memory leak.

OOPS

Access Specifiers

```cpp
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;
}
```

In C++, there are three access specifiers:

- **public** - members are accessible from outside the class
- **private** - members cannot be accessed outside the class
- **protected** - members cannot be accessed from outside the class, however, they can be accessed in inherited classes.

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

OOPS

Nested Classes

A nested class is a class which is declared in another enclosing class.
A nested class is a member and as such has the same access rights as any other member.
The members of an enclosing class have no special access to members of a nested class; the usual access rules shall be obeyed.

```
#include<iostream>

using namespace std;

 /* start of Enclosing class declaration */
class Enclosing {
   private:
       int x;

   /* start of Nested class declaration */
   class Nested {
      int y;
      void NestedFun(Enclosing *e) {
        cout<<e->x;  // works fine: nested class can access
                     // private members of Enclosing class
      }
   }; // declaration Nested class ends here
}; // declaration Enclosing class ends here

int main()
{

}
```

OOPS - Concepts

- Encapsulation
- Inheritance
- Polymorphism
- Abstraction

OOPS

Encapsulation

Encapsulation refers to the bundling of data with the methods that operate on that data, or the restricting of direct access to some of an object's components.

Encapsulation is used to hide the values or state of a structured data object inside a class, preventing unauthorized parties' direct access to them.

Publicly accessible methods are generally provided in the class to access the values, and other client classes call these methods to retrieve and modify the values within the object.

OOPS

Encapsulation

- The salary attribute is private, which have restricted access.
- The public setSalary() method takes a parameter (s) and assigns it to the salary attribute (salary = s).
- The public getSalary() method returns the value of the private salary attribute.
- Inside main(), we create an object of the Employee class. Now we can use the setSalary() method to set the value of the private attribute to 50000.
- Then we call the getSalary() method on the object to return the value.

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

class Employee {
  private:
    // Private attribute
    int salary;

  public:
    // Setter
    void setSalary(int s) {
      salary = s;
    }
    // Getter
    int getSalary() {
      return salary;
    }
};

int main() {
  Employee myObj;
  myObj.setSalary(50000);
  cout << myObj.getSalary();
  return 0;
}
```

OOPS

Inheritance

Inheritance is a mechanism in which one class acquires the property of another class. For example, a child inherits the traits of his/her parents. With inheritance, we can reuse the fields and methods of the existing class. Hence, inheritance facilitates Reusability and is an important concept of OOPs.

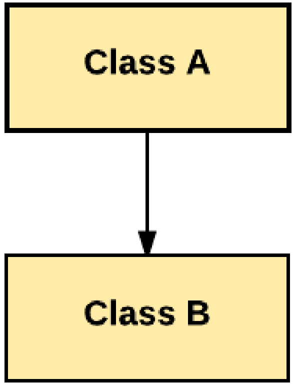
Types of Inheritance:

- Single Inheritance
- Multiple Inheritance
- Multilevel Inheritance
- Hierarchical Inheritance
- Hybrid Inheritance

OOPS

Inheritance – Single Inheritance

In Single Inheritance one class extends another class (one class only).

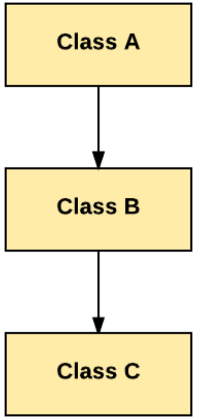


```
// Base class
class Vehicle {
  public:
    string brand = "Ford";
    void honk() {
      cout << "Tuut, tuut! \n" ;
    }
};
// Derived class
class Car: public Vehicle {
  public:
    string model = "Mustang";
};
int main() {
  Car myCar;
  myCar.honk();
  cout << myCar.brand + " " + myCar.model;
  return 0;
}
```

OOPS

Inheritance – Multilevel Inheritance

A class can also be derived from one class, which is already derived from another class.

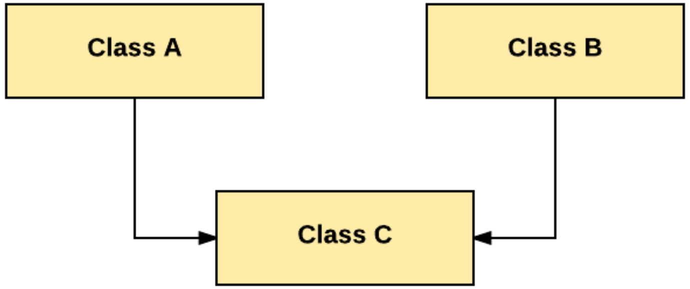


```cpp
// Base class (parent)
class MyClass {
  public:
    void myFunction() {
      cout << "Some content in parent class." ;
    }
};
// Derived class (child)
class MyChild: public MyClass {
};
// Derived class (grandchild)
class MyGrandChild: public MyChild {
};

int main() {
  MyGrandChild myObj;
  myObj.myFunction();
  return 0;
}
```

OOPS

Inheritance – Multiple Inheritance

A class can also be derived from more than one base class, using a **comma-separated list:**



```cpp
// Base class
class MyClass {
  public:
    void myFunction() {
      cout << "Some content in parent class." ;
    }
};
// Another base class
class MyOtherClass {
  public:
    void myOtherFunction() {
      cout << "Some content in another class." ;
    }
};
// Derived class
class MyChildClass: public MyClass, public MyOtherClass {
};
int main() {
  MyChildClass myObj;
  myObj.myFunction();
  myObj.myOtherFunction();
  return 0;
}
```

OOPS

Inheritance – Access Specifier

Protected access modifier is similar to that of private access modifiers, the difference is that the class member declared as Protected are inaccessible outside the class but they can be accessed by any subclass(derived class) of that class.

```cpp
// Base class
class Employee {
  protected: // Protected access specifier
    int salary;
};

// Derived class
class Programmer: public Employee {
  public:
    int bonus;
    void setSalary(int s) {
      salary = s;
    }
    int getSalary() {
      return salary;
    }
};

int main() {
  Programmer myObj;
  myObj.setSalary(50000);
  myObj.bonus = 15000;
  cout << "Salary: " << myObj.getSalary() << "\n";
  cout << "Bonus: " << myObj.bonus << "\n";
  return 0;
}
```

OOPS

Polymorphism

The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form. A real-life example of polymorphism, a person at the same time can have different characteristics. Like a man at the same time is a father, a husband, an employee.

1. Compile Time Polymorphism
2. Runtime Polymorphism

OOPS

Compile Time Polymorphism – Function Overloading

```
// Base class
class Animal {
  public:
    void animalSound() {
    cout << "The animal makes a sound \n" ;
  }
};
// Derived class
class Pig : public Animal {
  public:
    void animalSound() {
    cout << "The pig says: wee wee \n" ;
   }
};
// Derived class
class Dog : public Animal {
  public:
    void animalSound() {
    cout << "The dog says: bow wow \n" ;
  }
};
```

```
int main() {
  Animal myAnimal;
  Pig myPig;
  Dog myDog;

  myAnimal.animalSound();
  myPig.animalSound();
  myDog.animalSound();
  return 0;
}
```

OOPS

Run Time Polymorphism – Function Overriding

```
#include <iostream>
using namespace std;
class BaseClass {
  public: void disp(){
    cout<<"Function of Parent Class";
  }
};
class DerivedClass: public BaseClass{
  public:
    void disp() {
      cout<<"Function of Child Class";
    }
  };
int main() {
  DerivedClass obj = DerivedClass();
  obj.disp();
  return 0;
}
```

Output:

```
Function of Child Class
```

OOPS

Abstraction

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.

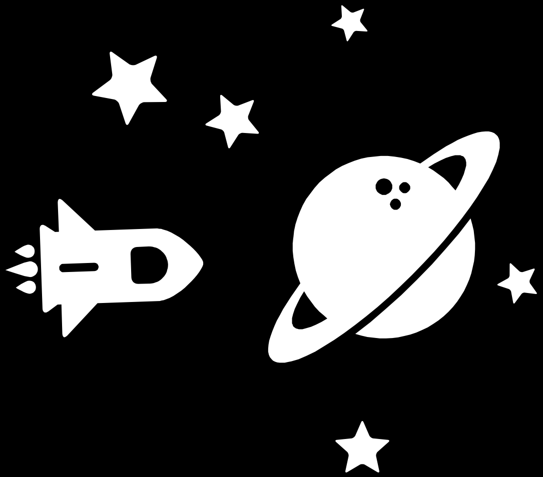
Data Abstraction is a programming technique that depends on the seperation of the interface and implementation details of the program.

Data Abstraction can be achieved in two ways:

- Abstraction using classes
- Abstraction in header files.

C++ Exception

Everything has some exceptional cases.



Exception Handling

Exception handling in C++ consist of three keywords: try, throw and catch:

- The try statement allows you to define a block of code to be tested for errors while it is being executed.
- The throw keyword throws an exception when a problem is detected, which lets us create a custom error.
- The catch statement allows you to define a block of code to be executed, if an error occurs in the try block.

The try and catch keywords come in pairs:

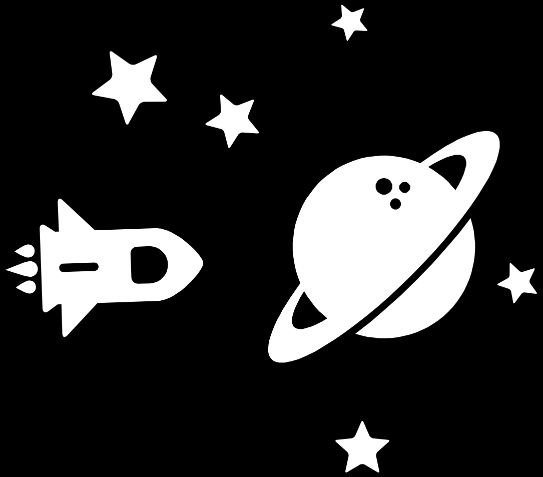
```
try {
  int age = 15;
  if (age > 18) {
    cout << "Access granted - you are old enough.";
  } else {
    throw (age);
  }
}
catch (int myNum) {
  cout << "Access denied - You must be at least 18 years 
old.\n";
  cout << "Age is: " << myNum;
}
```

Exception Handling

If you do not know the throw **type** used in the try block, you can use the "three dots" syntax (...) inside the catch block, which will handle any type of exception:

```
try {
  int age = 15;
  if (age > 18) {
    cout << "Access granted - you are old
enough.";
  } else {
    throw 505;
  }
}
catch (...) {
  cout << "Access denied - You must be at least
18 years old.\n";
}
```

You can create your own header file also.



Thanks!

Compiled by :

Grejo Joby