1.1 Basic concepts of OOPs

- **C++ programming language** was developed in 1980 by **Bjarne Stroustrup** at bell laboratories of AT&T (American Telephone & Telegraph), located in U.S.A.
- **Bjarne Stroustrup** is known as the **founder of C++ language.**
- It was develop for adding a feature of **OOP (Object Oriented Programming)** in C without significantly changing the C component.
- C++ programming is "relative" (called a superset) of C, it means any valid C program is also a valid C++ program.
- Let's see the programming languages that were developed before C++ language.

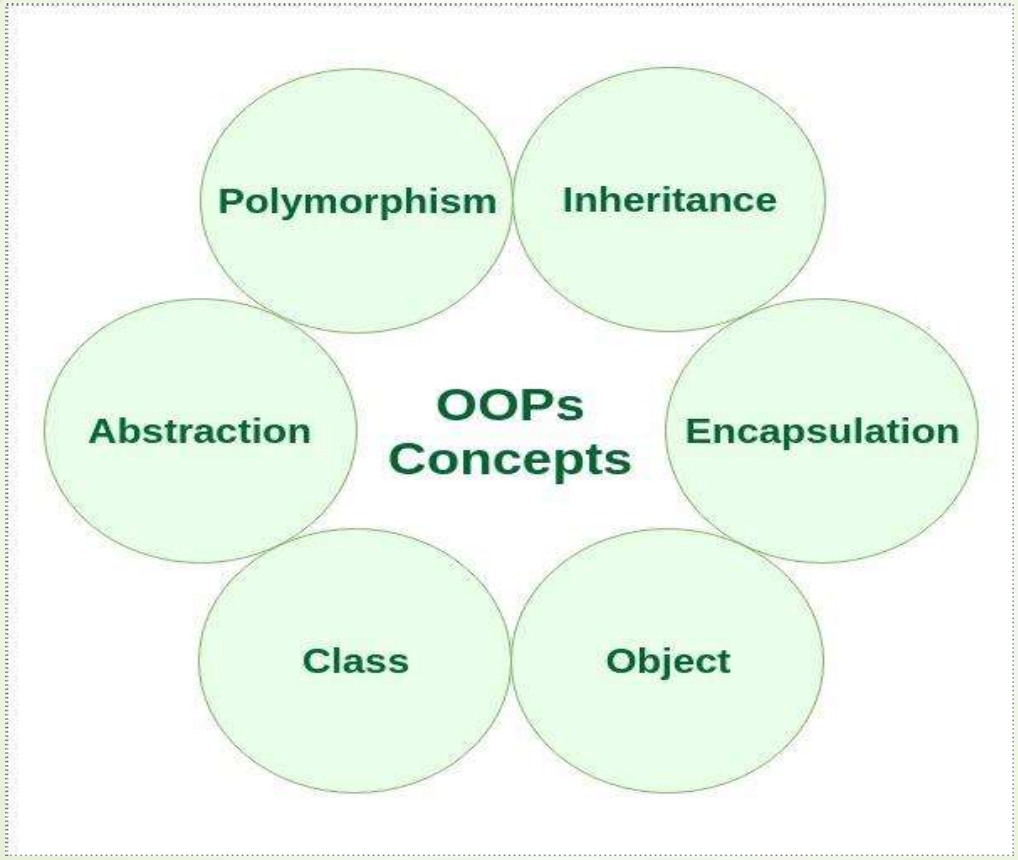
1.1 Basic concepts of OOPs

Object-oriented programming – As the name suggests uses objects in programming. Object-oriented programming aims to implement real-world entities like inheritance, hiding, polymorphism, etc. in programming. The main aim of OOP is to bind together the data and the functions that operate on them so that no other part of the code can access this data except that function.

There are some basic concepts that act as the building blocks of OOPs i.e.

1. **Class**
2. **Object**
3. **Encapsulation**
4. **Abstraction**
5. **Polymorphism**
6. **Inheritance**
7. **Dynamic Binding**
8. **Message Passing**

1.1 Basic concepts of OOPs



1.1 Basic concepts of OOPs

- **Class :**

The building block of C++ that leads to Object-Oriented programming is a Class. It is a user-defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A class is like a blueprint for an object. For Example: Consider the Class of Cars. There may be many cars with different names and brands but all of them will share some common properties like all of them will have 4 wheels, Speed Limit, Mileage range, etc. So here, the Car is the class, and wheels, speed limits, and mileage are their properties.

- A Class is a user-defined data type that has data members and member functions.
- Data members are the data variables and member functions are the functions used to manipulate these variables together these data members and member functions define the properties and behavior of the objects in a Class.
- In the above example of class Car, the data member will be speed limit, mileage, etc and member functions can apply brakes, increase speed, etc.
- We can say that a **Class in C++** is a blueprint representing a group of objects which shares some common properties and behaviors.

1.1 Basic concepts of OOPs

- **Object**
- An Object is an identifiable entity with some characteristics and behavior. An Object is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.

Objects take up space in memory and have an associated address like a record in pascal or structure or union. When a program is executed the objects interact by sending messages to one another. Each object contains data and code to manipulate the data. Objects can interact without having to know details of each other's data or code, it is sufficient to know the type of message accepted and the type of response returned by the objects.

1.1 Basic concepts of OOPs

- **Encapsulation**
- In normal terms, Encapsulation is defined as wrapping up data and information under a single unit. In Object-Oriented Programming, Encapsulation is defined as binding together the data and the functions that manipulate them. Consider a real-life example of encapsulation, in a company, there are different sections like the accounts section, finance section, sales section, etc. The finance section handles all the financial transactions and keeps records of all the data related to finance. Similarly, the sales section handles all the sales-related activities and keeps records of all the sales. Now there may arise a situation when for some reason an official from the finance section needs all the data about sales in a particular month. In this case, he is not allowed to directly access the data of the sales section. He will first have to contact some other officer in the sales section and then request him to give the particular data. This is what encapsulation is. Here the data of the sales section and the employees that can manipulate them are wrapped under a single name "sales section".
- *Encapsulation in C++*
- Encapsulation also leads to *data abstraction or data hiding*. Using encapsulation also hides the data. In the above example, the data of any of the sections like sales, finance, or accounts are hidden from any other section.

1.1 Basic concepts of OOPs

- **Abstraction**
- Data abstraction is one of the most essential and important features of object-oriented programming in C++. Abstraction means displaying only essential information and hiding the details. Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation. Consider a real-life example of a man driving a car. The man only knows that pressing the accelerator will increase the speed of the car or applying brakes will stop the car but he does not know how on pressing the accelerator the speed is actually increasing, he does not know about the inner mechanism of the car or the implementation of an accelerator, brakes, etc. in the car. This is what abstraction is.
- ***Abstraction using Classes***: We can implement Abstraction in C++ using classes. The class helps us to group data members and member functions using available access specifiers. A Class can decide which data member will be visible to the outside world and which is not.
- ***Abstraction in Header files***: One more type of abstraction in C++ can be header files. For example, consider the pow() method present in math.h header file. Whenever we need to calculate the power of a number, we simply call the function pow() present in the math.h header file and pass the numbers as arguments without knowing the underlying algorithm according to which the function is actually calculating the power of numbers.

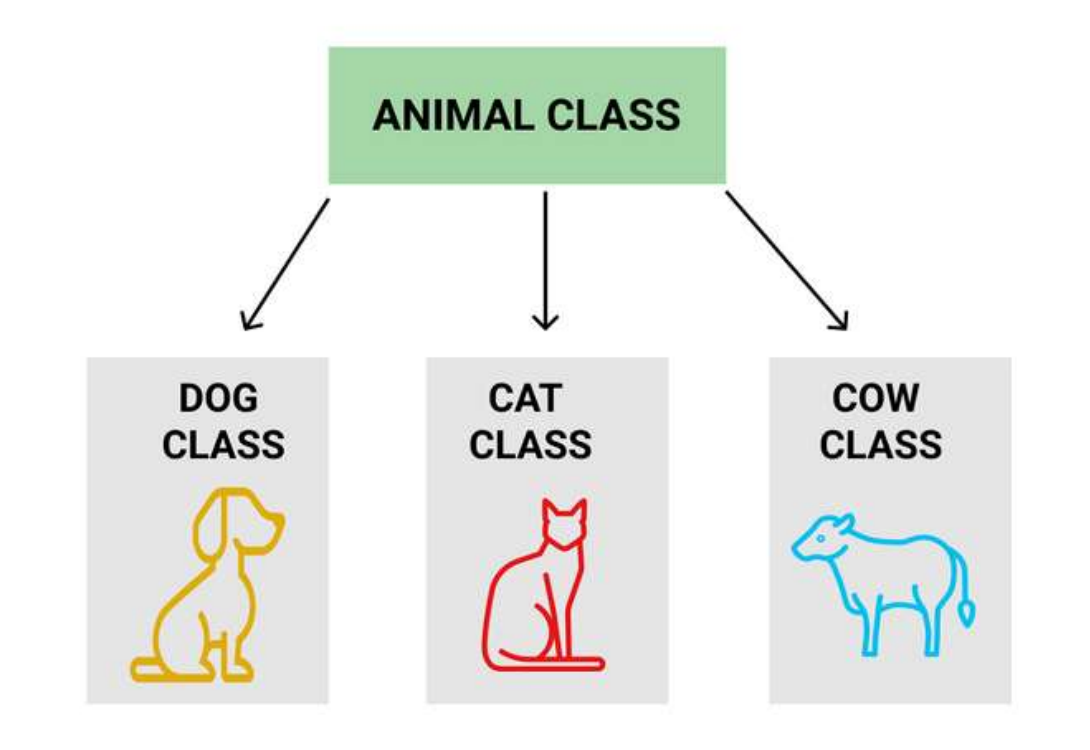
1.1 Basic concepts of OOPs

- **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 person at the same time can have different characteristics. A man at the same time is a father, a husband, and an employee. So the same person possesses different behavior in different situations. This is called polymorphism. An operation may exhibit different behaviors in different instances. The behavior depends upon the types of data used in the operation. C++ supports operator overloading and function overloading.
- ***Operator Overloading***: The process of making an operator exhibit different behaviors in different instances is known as operator overloading.
- ***Function Overloading***: Function overloading is using a single function name to perform different types of tasks. Polymorphism is extensively used in implementing inheritance.

1.1 Basic concepts of OOPs

- **Inheritance**
- The capability of a class to derive properties and characteristics from another class is called Inheritance. Inheritance is one of the most important features of Object-Oriented Programming.
- **Sub Class**: The class that inherits properties from another class is called Sub class or Derived Class.
- **Super Class**: The class whose properties are inherited by a sub-class is called Base Class or Superclass.
- **Reusability**: Inheritance supports the concept of "reusability", i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.
- **Example**: Dog, Cat, Cow can be Derived Class of Animal Base Class.

1.1 Basic concepts of OOPs



O O P S - CONCEPT

1.2 Benefits of OOPs

1. Modularity
2. Reusability
3. Encapsulation
4. Abstraction
5. Inheritance
6. Polymorphism
7. Flexibility and Scalability
8. Improved Maintainability
9. Real-World Modeling
10. Enhanced Collaboration
11. Ease of Troubleshooting

Advantages

1.2 Benefits of OOPs

- **1. Modularity**
- Breaks down large systems into smaller, manageable objects.
- Each object encapsulates its data and behavior, promoting independent development.
- Enhances code readability and simplifies understanding.
- Makes testing individual modules easier and more efficient.
- Facilitates code modifications without affecting unrelated components.
- Promotes better software organization and maintainability.

1.2 Benefits of OOPs

- **2. Reusability**
- Objects and classes can be reused across projects or application modules.
- Reduces the need to write repetitive code, saving time and effort.
- Encourages creating libraries of reusable components for common functionality.
- Enhances productivity with quicker development cycles.
- Promotes consistency in code design by reusing tried-and-tested implementations.
- Simplifies system extensions using existing reusable components.

1.2 Benefits of OOPs

- **3. Encapsulation**
- Groups related data and methods into a single unit (object).
- Restricts direct access to object data, protecting it from unintended modification.
- Provides controlled access through getter and setter methods.
- Encourages hiding complex implementation details from external entities.
- Increases security by ensuring that only intended operations are performed on data.
- Simplifies debugging by isolating object functionality.

1.2 Benefits of OOPs

- **4. Abstraction**
- Focuses on essential features while hiding unnecessary implementation details.
- Achieved through abstract classes and interfaces.
- Reduces system complexity, making it easier to understand and maintain.
- Promotes the design of adaptable and extendable systems.
- Encourages a separation of "what" a system does from "how" it does it.
- Helps in designing high-level, easy-to-understand blueprints.

1.2 Benefits of OOPs

- **5. Inheritance**
- Allows subclasses to inherit properties and behaviors from superclasses.
- Promotes code reuse, reducing redundancy in applications.
- Simplifies the process of adding new features by extending existing classes.
- Facilitates hierarchical classification for better code organization.
- Encourages consistent functionality across related classes.
- Supports polymorphism by allowing the same method to work differently in subclasses.

1.2 Benefits of OOPs

- **6. Polymorphism**
- Enables treating objects of different classes as instances of a common superclass.
- Provides flexibility by allowing method overriding for custom behavior in subclasses.
- Simplifies code by using a single interface to represent multiple data types.
- Encourages extensibility by allowing new classes to fit seamlessly into existing systems.
- Supports dynamic method invocation at runtime based on object type.
- Improves readability and maintainability with consistent method usage.

1.2 Benefits of OOPs

- **7. Flexibility and Scalability**
- Facilitates building systems that adapt easily to new requirements.
- Encourages extending functionality without modifying existing code.
- Reduces system downtime during upgrades or changes.
- Supports building large, complex projects with minimal code restructuring.
- Enhances modularity and simplifies collaboration among developers.
- Scales well for evolving software needs or increased application size.

1.2 Benefits of OOPs

- **8. Improved Maintainability**
- Encourages writing clean, modular code that is easy to update.
- Isolates changes within specific objects, minimizing the risk of errors.
- Simplifies fixing bugs and adding new features.
- Enhances debugging efficiency by reducing the scope of issues.
- Promotes long-term code sustainability with well-defined object boundaries.
- Helps in version management by ensuring better code organization.

1.2 Benefits of OOPs

- **9. Real-World Modeling**
- Mimics real-world entities through objects and their interactions.
- Enhances understanding of problems by bridging the problem and solution domains.
- Promotes intuitive mapping of real-world scenarios to code.
- Simplifies system analysis and design processes.
- Encourages creating reusable models for different applications.
- Makes it easier to communicate concepts among non-technical stakeholders.

1.2 Benefits of OOPs

- **10. Enhanced Collaboration**
- Divides responsibilities among developers using modular objects and classes.
- Simplifies integration of code written by multiple team members.
- Promotes consistency in the codebase with clear object interfaces.
- Encourages teamwork by isolating individual components for development.
- Reduces conflicts during development with independent modules.
- Enhances code readability, making it easier for team members to understand.

1.2 Benefits of OOPs

- **11. Ease of Troubleshooting**
- Isolates errors within specific objects, simplifying debugging.
- Promotes targeted testing of individual components.
- Reduces the complexity of diagnosing and fixing system-wide issues.
- Encourages logging and monitoring at the object level for better diagnostics.
- Improves software reliability by enabling quick resolution of issues.
- Supports iterative development by addressing issues in small, manageable modules.

1.3 C++ Tokens, Variables, Constants and data types

1.3 C++ Tokens, Variables, Constants and data types

1. Punctuators

- Punctuators are the token characters having specific meanings within the syntax of the programming language. These symbols are used in a variety of functions, including ending the statements, defining control statements, separating items, and more.
- **Below are the most common punctuators used in C++ programming:**
- **Semicolon (;):** It is used to terminate the statement.
- **Square brackets []:** They are used to store array elements.
- **Curly Braces {}:** They are used to define blocks of code.
- **Double-quote (“):** It is used to enclose string literals.
- **Single-quote (‘):** It is used to enclose character literals.

1.3 C++ Tokens, Variables, Constants and data types

2. Keywords

- **Keywords** in C++ are the tokens that are the reserved words in programming languages that have their specific meaning and functionalities within a program. Keywords cannot be used as an identifier to name any variables.
- For example, a variable or function cannot be named as 'int' because it is reserved for declaring integer data type.
- There are 95 keywords reserved in C++.
- Common Keywords Used in C++
- **auto, break, case, char, const, continue, default, do, double, else, enum, extern, float, for, goto, if, inline, int, long, register, return, short, signed, sizeof, static, struct, switch, typedef, union, unsigned, void, volatile, while**

1.3 C++ Tokens, Variables, Constants and data types

3. Strings

- In C++, a **string** is not a built-in data type like 'int', 'char', or 'float'. It is a class available in the STL library which provides the functionality to work with a sequence of characters, that represents a string of text.
- string variable_name;
  string name= "This is string";
- When we define any variable using the 'string' keyword we are actually defining an object that represents a sequence of characters

1.3 C++ Tokens, Variables, Constants and data types

4. Operators

- C++ operators are special symbols that are used to perform operations on operands such as variables, constants, or expressions. A wide range of operators is available in C++ to perform a specific type of operations which includes arithmetic operations, comparison operations, logical operations, and more.
- **For example,** (A+B), in which 'A' and 'B' are operands, and '+' is an arithmetic operator which is used to add two operands.
- There can be three types of operators:
- **1. Unary Operators**
- Unary operators are used with **single operands** only. They perform the operations on a single variable. For example, increment and decrement operators.
- **Increment operator (++):** It is used to increment the value of an operand by 1.
- **Decrement operator (—):** It is used to decrement the value of an operand by 1.

1.3 C++ Tokens, Variables, Constants and data types

- **2. Binary Operators**
- They are used with the **two operands** and they perform the operations between the two variables. For example (A<B), less than (<) operator compares A and B, returns true if A is less than B else returns false.
- **Arithmetic Operators:** These operators perform basic arithmetic operations on operands. They include **'+', '-', '*', '/',** and **'%'**
- **Comparison Operators:** These operators are used to compare two operands, and they include '==', '!=', '<', '>', '<=', and '>='.
- **Logical Operators:** These operators perform logical operations on boolean values. They include '&&', '||', and '!'.
- **Assignment Operators:** These operators are used to assign values to variables, and they include **'variables'**, '**-=**', '***=**', '**/=**', and '**%=**'.
- **Bitwise Operators:** These operators perform bitwise operations on integers. They include'&', '|', '^', '~', '<<', and '>>'.

1.3 C++ Tokens, Variables, Constants and data types

- **3. Ternary Operator**
- The **ternary operator** is the only operator that takes three operands. It is also known as a conditional operator that is used for conditional expressions.
- Expression1 ? Expression2 : Expression3;
- If**Expression1** became **true** then **Expression2** will be executed otherwise **Expression3** will be executed.

1.3 C++ Tokens, Variables, Constants and data types

5. Constants

- **Constants** are the tokens in C++ that are used to define variables at the time of initialization and the assigned value cannot be changed after that.We can define the constants in C++ in two ways that are using the **const**, **constexpr** keyword and **#define** preprocessor directive.
- const type name = val
  constexpr type name = val
  #define name val

1.3 C++ Tokens, Variables, Constants and data types

6. Identifiers

- In C++, entities like variables, functions, classes, or structs must be given unique names within the program so that they can be uniquely identified. The unique names given to these entities are known as **identifiers.**
- It is recommended to choose valid and relevant names of identifiers to write readable and maintainable programs. Keywords cannot be used as an identifier because they are reserved words to do specific tasks. In the below example, "**first_name**' is an identifier.
- string **first_name** = "Raju";

1.3 C++ Tokens, Variables, Constants and data types

- **We have to follow a set of rules to define the name of identifiers as follows:**
- An identifier can only begin with a **letter or an underscore(_).**
- An identifier can consist of **letters** (A-Z or a-z), **digits** (0-9), and **underscores (_)**. White spaces and Special characters can not be used as the name of an identifier.
- Keywords cannot be used as an identifier because they are reserved words to do specific tasks. For example, **string**, **int, class**, **struct**, etc.
- Identifier must be **unique** in its namespace.
- As C++ is a case-sensitive language so identifiers such as 'first_name' and 'First_name' are different entities.

1.3 C++ Tokens, Variables, Constants and data types

Variables :

Variables are containers for storing data values.

- Declaring (Creating) Variables
- To create a variable, specify the type and assign it a value:
- **Syntax**
- *datatype variableName = value;*
- **Example**
- int myNum = 15;

1.3 C++ Tokens, Variables, Constants and data types

- Constants
- When you don't want others (or yourself) to change existing variable values, use the **const** keyword (this will declare the variable as "constant", which means **unchangeable and read-only**):

Example :

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

Note : When you declare a constant variable, it must be assigned with a value

1.3 C++ Tokens, Variables, Constants and data types

Data Type :

- The data type specifies the size and type of information the variable will store:

Data Type	Size	Description
boolean	1 byte	Stores true or false values
char	1 byte	Stores a single character/letter/number, or ASCII values
int	2 or 4 bytes	Stores whole numbers, without decimals
float	4 bytes	Stores fractional numbers, containing one or more decimals. Sufficient for storing 6-7 decimal digits
double	8 bytes	Stores fractional numbers, containing one or more decimals. Sufficient for storing 15 decimal digits

1.3 C++ Tokens, Variables, Constants and data types

Example :

```
int myNum = 5;                  // Integer (whole number)
float myFloatNum = 5.99;        // Floating point number
double myDoubleNum = 9.98;      // Floating point number
char myLetter = 'D';            // Character
bool myBoolean = true;          // Boolean
string myText = "Hello";        // String
```

1.4 Basic Input / Output Statements

- In C++, input and output are performed in the form of a sequence of bytes or more commonly known as **streams**.
- **Input Stream:** If the direction of flow of bytes is from the device (for example, Keyboard) to the main memory then this process is called input.
- **Output Stream:** If the direction of flow of bytes is opposite, i.e. from main memory to device (display screen) then this process is called output.

All of these streams are defined inside the **<iostream>** header file which contains all the standard input and output tools of C++. The two instances **cout** and **cin** of iostream class are used very often for printing outputs and taking inputs respectively. These two are the most basic methods of taking input and printing output in C++.

1.4 Basic Input / Output Statements

Standard Output Stream – cout

- The C++ **cout** is the instance of the **ostream** class used to produce output on the standard output device which is usually the display screen. The data needed to be displayed on the screen is inserted in the standard output stream (cout) using the **insertion operator(<<)**.
- **Syntax**
- **cout <<** value/variable;
- **Example**
- **cout<<**"Hello World";

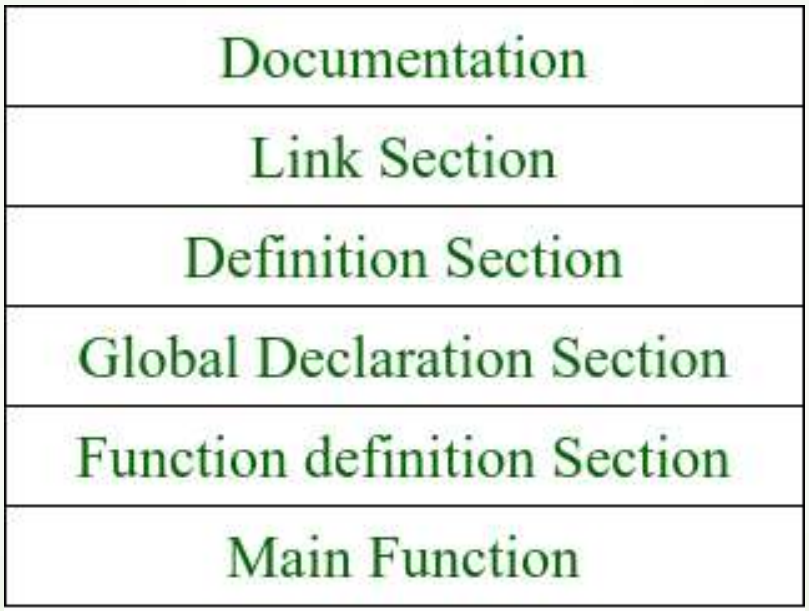
1.4 Basic Input / Output Statements

Standard Input Stream – cin

- The C++ **cin** statement is the instance of the class **istream** and is used to read input from the standard input device which is usually a keyboard. The **extraction operator (>>)** is used along with the object **cin** for extracting the data from the input stream and store it in some variable in the program.
- **Syntax**
- **cin >>** variable;
- **Example**
- **cin>>**num;

1.5 Structure of C++ Program

The C++ program is written using a specific template structure. The structure of the program written in C++ language is as follows:



Skeleton of C Program

1.5 Structure of C++ Program

- **Documentation Section**:
- This section comes first and is used to document the logic of the program that the programmer going to code.
- It can be also used to write for purpose of the program.
- Whatever written in the documentation section is the comment and is not compiled by the compiler.
- Documentation Section is optional since the program can execute without them.

1.5 Structure of C++ Program

- **Linking Section:**
- The linking section contains two parts:
- **Header Files:**
- Generally, a program includes various programming elements like built-in functions, classes, keywords, constants, operators, etc. that are already defined in the standard C++ library.
- In order to use such pre-defined elements in a program, an appropriate header must be included in the program.
- Standard headers are specified in a program through the preprocessor directive #include. In Figure, the iostream header is used. When the compiler processes the instruction #include<iostream>, it includes the contents of the stream in the program. This enables the programmer to use standard input, output, and error facilities that are provided only through the standard streams defined in <iostream>. These standard streams process data as a stream of characters, that is, data is read and displayed in a continuous flow. The standard streams defined in <iostream> are listed here.
- #include<iostream>

1.5 Structure of C++ Program

- **Namespaces:**
- A namespace permits grouping of various entities like classes, objects, functions, and various C++ tokens, etc. under a single name.
- Any user can create separate namespaces of its own and can use them in any other program.
- In the below snippets, **namespace std** contains declarations for cout, cin, endl, etc. statements.
- using namespace std;
  Namespaces can be accessed in multiple ways:
  - using namespace std;
  - using std :: cout;

1.5 Structure of C++ Program

- **Definition Section**:
- It is used to declare some constants and assign them some value.
- In this section, anyone can define your own datatype using primitive data types.
- In **#define** is a compiler directive which tells the compiler whenever the message is found
- **typedef int INTEGER;** this statement tells the compiler that whenever you will encounter INTEGER replace it by int and as you have declared INTEGER as datatype you cannot use it as an identifier.

1.5 Structure of C++ Program

- **Global Declaration Section:**
- Here, the variables and the class definitions which are going to be used in the program are declared to make them global.
- The scope of the variable declared in this section lasts until the entire program terminates.
- These variables are accessible within the user-defined functions also.
- **Function Declaration Section:**
- It contains all the functions which our main functions need.
- Usually, this section contains the User-defined functions.
- This part of the program can be written after the main function but for this, write the function prototype in this section for the function which for you are going to write code after the main function.

1.5 Structure of C++ Program

Main Function :

- The main function tells the compiler where to start the execution of the program. The execution of the program starts with the main function.
- All the statements that are to be executed are written in the main function.
- The compiler executes all the instructions which are written in the curly braces {} which encloses the body of the main function.
- Once all instructions from the main function are executed, control comes out of the main function and the program terminates and no further execution occur.

1.6 Scope Resolution Operator in C++

In C++, the **scope resolution operator (::)** is used to access the identifiers such as variable names and function names defined inside some other scope in the current scope.

- **Syntax of Scope Resolution Operator**
- The scope resolution operator follows this general syntax:
- scope_name **::** identifier
- where ***scope_name*** is the name of the scope where ***identifier*** is defined.

1.6 Scope Resolution Operator in C++

- **Applications of Scope Resolution Operator**
- Following are the main applications of scope resolution operator illustrated with an example:
- **Accessing Global Variables**
- When a local variable shadows a global variable, we can use :: to access the global variable.

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

// Global x
int x = 3;

int main() {

        // Local x
    int x = 10;

        // Printing the global x
    cout << ::x;

    return 0;
}
```

1.6 Scope Resolution Operator in C++

- **Namespace Resolution**
- It is also used to resolve the identifier declared inside different namespaces.

```
#include <bits/stdc++.h>
using namespace std;

// A sample namespace with a variable
namespace N {
      int val = 10;
};

int main() {

      // Accessing val from namespace N
    cout << N::val;

      return 0;
}
```

1.6 Scope Resolution Operator in C++

- **Define Class Member Function Outside Class**
- It is also used to define the member function of the class outside the class template.

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

// A sample class
class A {
public:

    // Only declaration of member function
    void fun();
};

// Definition outside class by referring to it
// using ::
void A::fun() {
      cout << "fun() called";
}

int main() {
    A a;
    a.fun();
    return 0;
}
```

1.6 Scope Resolution Operator in C++

- **Define Class Member Function Outside Class**
- It is also used to define the member function of the class outside the class template.

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

// A sample class
class A {
public:

    // Only declaration of member function
    void fun();
};

// Definition outside class by referring to it
// using ::
void A::fun() {
      cout << "fun() called";
}

int main() {
    A a;
    a.fun();
    return 0;
}
```

1.6 Scope Resolution Operator in C++

- **Access Class's Static Members**
- Static members of a class can be accessed without creating the object of the class. It is possible to access them using scope resolution operator.

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

class A {
public:
    static int x;
};

// In C++, static members must be explicitly defined
// like this
int A::x = 1;

int main() {

      // Accessing static data member
      cout << A::x;

    return 0;
}
```

1.6 Scope Resolution Operator in C++

- **Refer to Base Class Member in Derived Class**
- The scope resolution operator can also be used to refer to the members of base class in a derived class especially if they have the same name.
-

```cpp
#include <bits/stdc++.h>
using namespace std;

class Base {
public:
    void func() {
        cout << "Base class func()" << endl;
    }
};

class Derived : public Base {
public:

    // Overridden function
    void func() {
        cout << "Derived class func()" << endl;
    }
};

int main() {
    Derived obj;

    // Calling base class's func() from the object of
    // derived class
    obj.Base::func();

    obj.func();
    return 0;
}
```

1.7 Control Structure : Conditional Statements, Looping Statements, Jumping Statements

C++ has the following conditional statements:

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

The if Statement

- Use the if statement to specify a block of C++ code to be executed if a condition is true.
- **Syntax**
- **if (*condition*)**
- **{**
  // block of code to be executed if the condition is true
  }

1.7 Control Structure : Conditional Statements, Looping Statements, Jumping Statements

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

int main() {
  int x = 20;
  int y = 18;
  if (x > y) {
    cout << "x is greater than y";
  }
  return 0;
}
```

1.7 Control Structure : Conditional Statements, Looping Statements, Jumping Statements

- The else Statement
- Use the else statement to specify a block of code to be executed if the condition is false.
- **Syntax**
- **if (*condition*)**
- **{**
  // block of code to be executed if the condition is true
  }
- **else**
- **{**
  // block of code to be executed if the condition is false
  }

1.7 Control Structure : Conditional Statements, Looping Statements, Jumping Statements

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

int main() {
  int time = 20;
  if (time < 18) {
    cout << "Good day.";
  } else {
    cout << "Good evening.";
  }
  return 0;
}
```

1.7 Control Structure : Conditional Statements, Looping Statements, Jumping Statements

- The else if Statement
- Use the else if statement to specify a new condition if the first condition is false.
- Syntax
- if (*condition1*) {
  // block of code to be executed if condition1 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 condition1 is false and condition2 is false
  }

1.7 Control Structure : Conditional Statements, Looping Statements, Jumping Statements

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

int main() {
  int time = 22;
  if (time < 10) {
    cout << "Good morning.";
  } else if (time < 20) {
    cout << "Good day.";
  } else {
    cout << "Good evening.";
  }
  return 0;
}
```

1.7 Control Structure : Conditional Statements, Looping Statements, Jumping Statements

- Switch Statements
- Use the switch statement to select one of many code blocks to be executed.
- Syntax
- switch(*expression*) {

```
  case x:
    // code block
    break;
  case y:
    // code block
    break;
  default:
    // code block
}
```

1.7 Control Structure : Conditional Statements, Looping Statements, Jumping Statements

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

int main() {
  int day = 4;
  switch (day) {
    case 6:
      cout << "Today is Saturday";
      break;
    case 7:
      cout << "Today is Sunday";
      break;
    default:
      cout << "Looking forward to the Weekend";
  }
  return 0;
}
```

1.7 Control Structure : Conditional Statements, Looping Statements, Jumping Statements

- In C++, looping statements are used to execute a block of code repeatedly based on a condition. There are three types of looping statements in C++:
- **for loop**
- **while loop**
- **do-while loop**

1.7 Control Structure : Conditional Statements, Looping Statements, Jumping Statements

- **For Loop**
- The for loop is used when you know beforehand how many times you want to execute a statement or a block of statements.
- **Syntax:**
- for(initialization; condition; increment/decrement) { // Code to be executed }
- **Initialization**: Executed once, before the loop starts.
- **Condition**: Checked before every iteration, and if true, the loop continues.
- **Increment/Decrement**: Executed after each iteration.

1.7 Control Structure : Conditional Statements, Looping Statements, Jumping Statements

Example:

cpp Copy

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

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

Output:

yaml Copy

```yaml
Iteration number: 1
Iteration number: 2
Iteration number: 3
Iteration number: 4
Iteration number: 5
```

1.7 Control Structure : Conditional Statements, Looping Statements, Jumping Statements

- **2. While Loop**
- The while loop is used when the number of iterations is not necessarily known, and you want to repeat a block of code as long as a condition is true.
- **Syntax:**
- while (condition) { // Code to be executed }
- **Condition:** Checked before every iteration. If it's true, the loop continues

1.7 Control Structure : Conditional Statements, Looping Statements, Jumping Statements

Example:

cpp Copy

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

int main() {
    int i = 1;
    while (i <= 5) {
        cout << "Iteration number: " << i << endl;
        i++;
    }
    return 0;
}
```

Output:

yaml Copy

```yaml
Iteration number: 1
Iteration number: 2
Iteration number: 3
Iteration number: 4
Iteration number: 5
```

1.7 Control Structure : Conditional Statements, Looping Statements, Jumping Statements

- **3. Do-While Loop**
- The do-while loop is similar to the while loop, but it guarantees at least one iteration, as the condition is checked after the loop's execution.
- **Syntax:**
- do { // Code to be executed } while (condition);
- **Condition:** Checked after each iteration. The loop runs at least once, even if the condition is false at the first check.

1.7 Control Structure : Conditional Statements, Looping Statements, Jumping Statements

Example:

cpp Copy

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

int main() {
    int i = 1;
    do {
        cout << "Iteration number: " << i << endl;
        i++;
    } while (i <= 5);
    return 0;
}
```

Output:

yaml Copy

```yaml
Iteration number: 1
Iteration number: 2
Iteration number: 3
Iteration number: 4
Iteration number: 5
```

1.7 Control Structure : Conditional Statements, Looping Statements, Jumping Statements

- **Summary:**
- **For Loop**: Best when you know the number of iterations.
- **While Loop**: Useful when the condition may change during the loop execution.
- **Do-While Loop**: Ensures at least one iteration, regardless of the condition.

1.7 Control Structure : Conditional Statements, Looping Statements, Jumping Statements

- Jumping statements in C++ allow you to alter the flow of execution within loops or functions. These statements include:
- **break**
- **continue**
- **return**

1.7 Control Structure : Conditional Statements, Looping Statements, Jumping Statements

- **1. break Statement**
- The break statement is used to exit from a loop or a switch statement immediately, regardless of whether the loop condition is true or not.
- **Syntax:**
- break;

Example:

cpp Copy

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

int main() {
    for (int i = 1; i <= 10; i++) {
        if (i == 6) {
            break;  // Exit the loop when i equals 6
        }
        cout << "Iteration number: " << i << endl;
    }
    cout << "Loop ended." << endl;
    return 0;
}
```

Output:

yaml Copy

```yaml
Iteration number: 1
Iteration number: 2
Iteration number: 3
Iteration number: 4
Iteration number: 5
Loop ended.
```

In this example, the loop stops when `i` becomes 6 because of the `break` statement.

1.7 Control Structure : Conditional Statements, Looping Statements, Jumping Statements

- **2. continue Statement**
- The continue statement skips the current iteration of a loop and proceeds to the next iteration of the loop. It only affects the loop in which it is called.
- **Syntax:**
- continue;

Example:

cpp Copy

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

int main() {
    for (int i = 1; i <= 10; i++) {
        if (i == 6) {
            continue;  // Skip iteration when i equals 6
        }
        cout << "Iteration number: " << i << endl;
    }
    return 0;
}
```

Output:

yaml Copy

```yaml
Iteration number: 1
Iteration number: 2
Iteration number: 3
Iteration number: 4
Iteration number: 5
Iteration number: 7
Iteration number: 8
Iteration number: 9
Iteration number: 10
```

Here, when `i` equals 6, the `continue` statement skips that iteration, and the loop moves on to the next value.

1.7 Control Structure : Conditional Statements, Looping Statements, Jumping Statements

- **3. return Statement**
- The return statement is used to exit a function and optionally return a value. When used in a loop or inside a function, it terminates the current execution and returns control to the calling function.
- **Syntax:**
- return; // If returning void or no value return value; // If returning a value

Example:

cpp Copy

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

void printNumbers(int limit) {
    for (int i = 1; i <= limit; i++) {
        if (i == 6) {
            return;  // Exit the function when i equals 6
        }
        cout << "Iteration number: " << i << endl;
    }
}

int main() {
    printNumbers(10);  // Function call
    cout << "Function call ended." << endl;
    return 0;
}
```

Output:

yaml Copy

```yaml
Iteration number: 1
Iteration number: 2
Iteration number: 3
Iteration number: 4
Iteration number: 5
Function call ended.
```

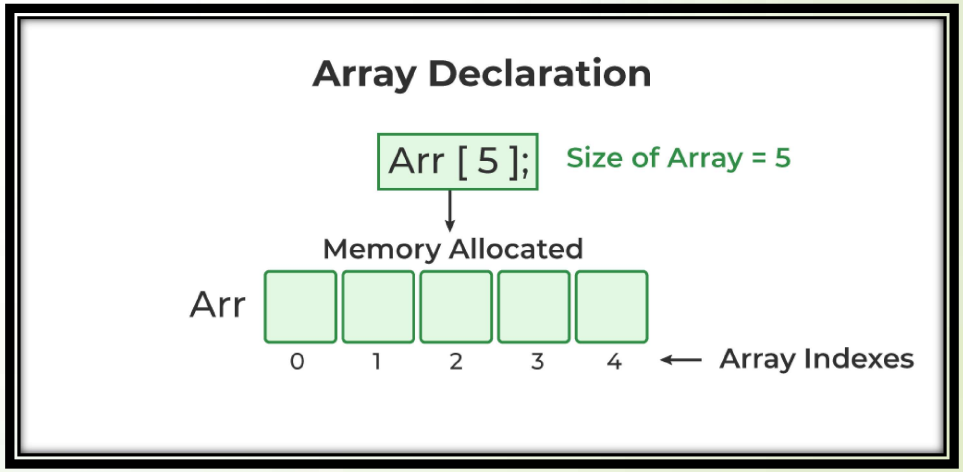
In this case, the function `printNumbers` exits when `i` equals 6 due to the `return` statement, and the message `"Function call ended."` is printed afterward.

1.7 Control Structure : Conditional Statements, Looping Statements, Jumping Statements

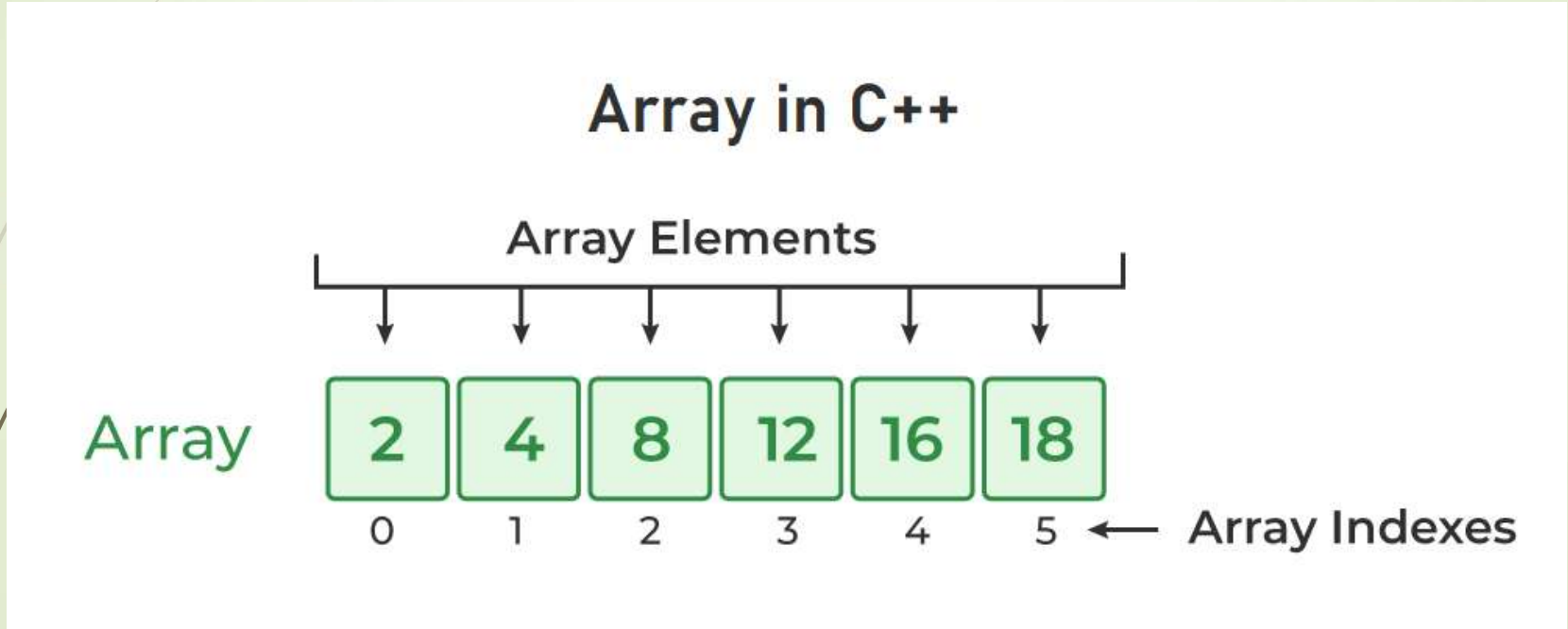
- **Summary of Jumping Statements:**
- **break**: Exits from the nearest loop or switch statement.
- **continue**: Skips the current iteration of a loop and continues to the next iteration.
- **return**: Exits from the current function and optionally returns a value to the caller.

1.8 Arrays

- C++ Arrays
- In C++, an **array** is a collection of elements of the same data type stored in contiguous memory locations. Arrays allow you to store multiple values in a single variable, which is useful for working with a large set of data.
- To declare an array, define the variable type, specify the name of the array followed by **square brackets** and specify the number of elements it should store:
- **Syntax :**
- data_type array_name[Size_of_array];
- **Example :**
- int arr[5];
- int myNum[3] = {10, 20, 30};



1.8 Arrays


Array in C++
Array Elements
Array
2
4
8
12
16
18
0
1
2
3
4
5
Array Indexes

1.8 Arrays

- **1. Initialize Array with Values in C++**
- int arr[5] = {1, 2, 3, 4, 5};
- **2. Initialize Array with Values and without Size in C++**
- int arr[] = {1, 2, 3, 4, 5};

1.8 Arrays

- **Properties of Arrays in C++**
- An Array is a collection of data of the same data type, stored at a contiguous memory location.
- Indexing of an array starts from **0.** It means the first element is stored at the 0th index, the second at 1st, and so on.
- Elements of an array can be accessed using their indices.
- Once an array is declared its size remains constant throughout the program.
- An array can have multiple dimensions.
- The size of the array in bytes can be determined by the sizeof operator using which we can also find the number of elements in the array.
- We can find the size of the type of elements stored in an array by subtracting adjacent addresses.

```cpp
//  C++ Program to Illustrate How to Access Array Elements
#include <iostream>
using namespace std;

int main()
{

    int arr[3];

    // Inserting elements in an array
    arr[0] = 10;
    arr[1] = 20;
    arr[2] = 30;

    // Accessing and printing elements of the array
    cout << "arr[0]: " << arr[0] << endl;
    cout << "arr[1]: " << arr[1] << endl;
    cout << "arr[2]: " << arr[2] << endl;

    return 0;
}
```

Output

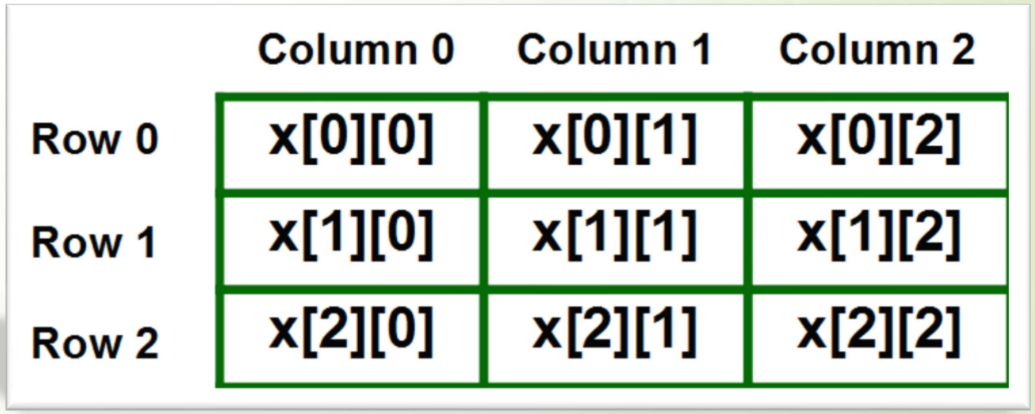
```
arr[0]: 10
arr[1]: 20
arr[2]: 30
```

1.8 Arrays

- **Multidimensional Arrays in C++**
- Arrays declared with more than one dimension are called multidimensional arrays. The most widely used multidimensional arrays are 2D arrays and 3D arrays. These arrays are generally represented in the form of rows and columns.
- **Multidimensional Array Declaration**
- Data_Type Array_Name[Size1][Size2]...[SizeN];
- **Data_Type:** Type of data to be stored in the array.
- **Array_Name:** Name of the array.
- **Size1, Size2,..., SizeN:** Size of each dimension.

1.8 Arrays

- **Two Dimensional Array in C++**
- In C++, a two-dimensional array is a grouping of elements arranged in rows and columns. Each element is accessed using two indices: one for the row and one for the column, which makes it easy to visualize as a table or grid.
- **Syntax of 2D array**
- data_Type array_name[n][m];
- Where,
- **n:** Number of rows.
- **m:** Number of columns



```cpp
// c++ program to illustrate the two dimensional array
#include <iostream>
using namespace std;

int main()
{
    // Declaring 2D array
    int arr[4][4];

    // Initialize 2D array using loop
    for (int i = 0; i < 4; i++) {
        for (int j = 0; j < 4; j++) {
            arr[i][j] = i + j;
        }
    }

    // Printing the element of 2D array
    for (int i = 0; i < 4; i++) {
        for (int j = 0; j < 4; j++) {
            cout << arr[i][j] << " ";
        }
        cout << endl;
    }

    return 0;
}
```

Output

```
0 1 2 3
1 2 3 4
2 3 4 5
3 4 5 6
```

References

- https://www.geeksforgeeks.org/c-plus-plus/
- https://www.w3schools.com/cpp/default.asp
- https://cplusplus.com/doc/tutorial/