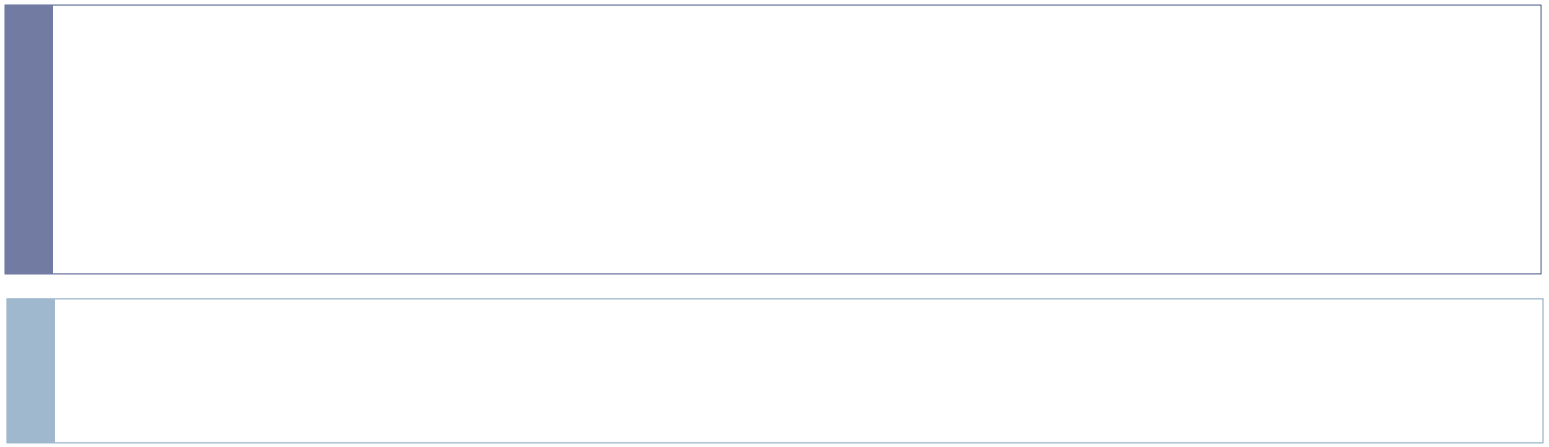
Unit-1Principles of ObjectOriented Programming Tokens, and Control Statements



Index

- Procedure – oriented programming
- Object oriented programming paradigm
- Basic concepts of object-oriented Programming
- Benefits of object-oriented programming
- Application of object-oriented programming
- What is C++?
- Application of C++
- Input/output operators
- Structure of C++ program
- Introduction of namespace

Index

- Tokens:
- keywords,
- identifiers,
- basic data types,
- user- defined types,
- derived data types,
- symbolic constants,
- type compatibility,
- declaration of variables,
- dynamic initialization of variables,
- reference variables

Index

- Operators in C++:
- ▪ scope resolution operator,
- ▪ member referencing operator,
- ▪ memory management operator,
- ▪ manipulators

Procedure – oriented programming

- Procedure programming can be defined as a programming model which is derived from structured programming, based upon the concept of calling procedure.
- Procedures, also known as routines, subroutines or functions, simply consist of a series of computational steps to be carried out.
- During a program's execution, any given procedure might be called at any point, including by other procedures or itself.
- *BASIC, Pascal and C.*

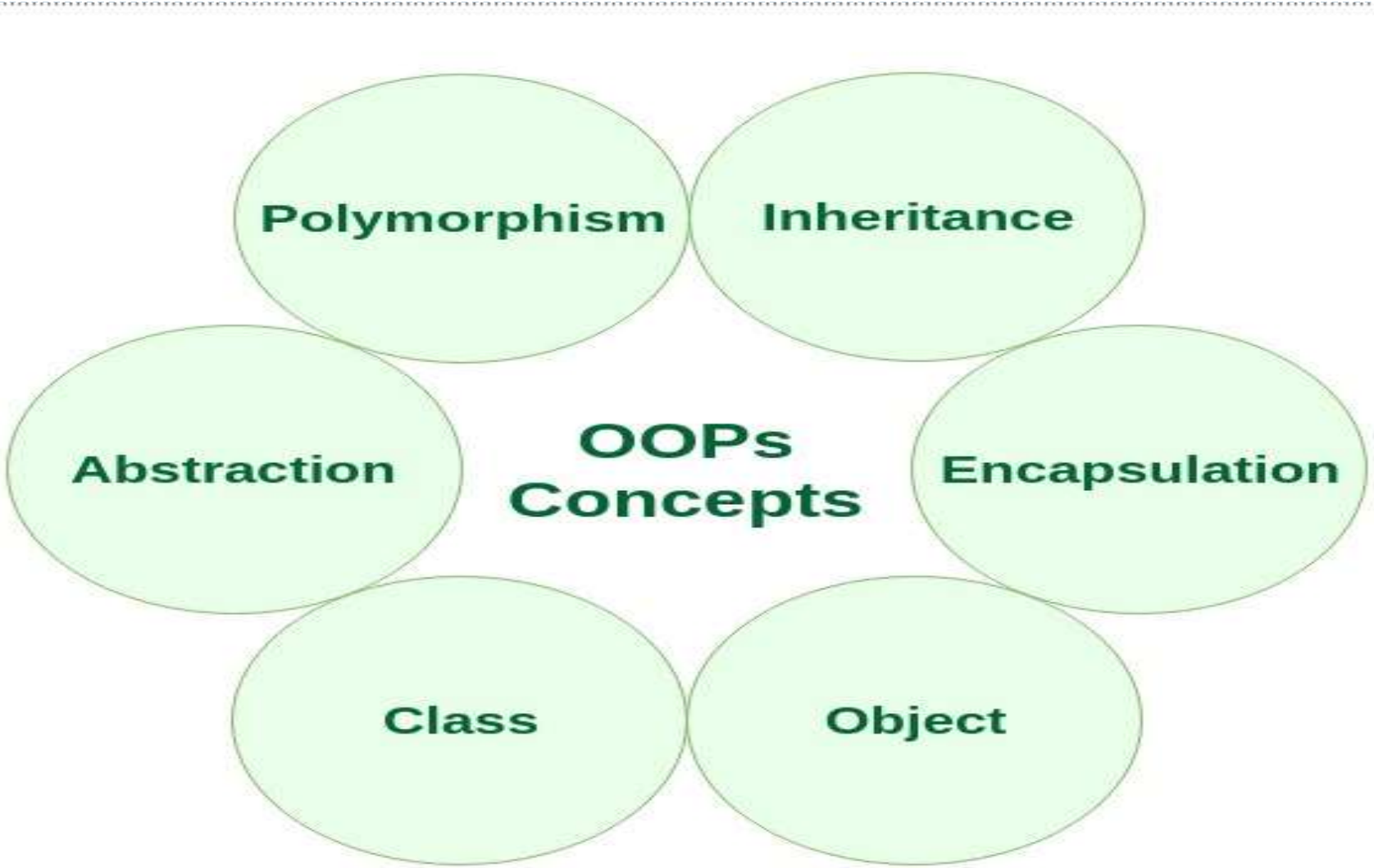
Object oriented programming paradigm

- Object oriented programming can be defined as a programming model which is based upon the concept of objects.
- Objects contain data in the form of attributes and code in the form of methods.
- In object-oriented programming, computer programs are designed using the concept of objects that interact with the real world.
- Object-oriented programming languages are various but the most popular ones are class-based, meaning that objects are instances of classes, which also determine their types.
- *Java, C++, C#, Python,*

Basic concepts of object-oriented Programming

- Object-Oriented Programming (OOP) is a programming paradigm that organizes software design around data, or objects, rather than functions and logic.
- The core idea is to model real-world entities as objects that have both **attributes** (data) and **behaviors** (methods).

1. Class
2. Objects
3. Encapsulation
4. Abstraction
5. Polymorphism
6. Inheritance


Polymorphism
Inheritance
Abstraction
OOPs Concepts
Encapsulation
Class
Object

Class

- It is a user-defined data type that act as a blueprint representing a group of objects which share some common properties and behaviors.
- These properties are stored as data members, and the behavior is represented by member functions.
- For example, consider the class of Animals.
  An Animal class could have common properties like name, age, and species as **data members**, and behaviors like eat, sleep, and makesound as **member functions.**

Example of Class

```
class Animal {
 public:
     string species;
     int age;
     int name;
// Member functions
     void eat()
{
      // eat something
 }
     void sleep()
{
    // sleep for few hrs
 }
void makeSound ()
{
     // make sound;
 } };
```

Object

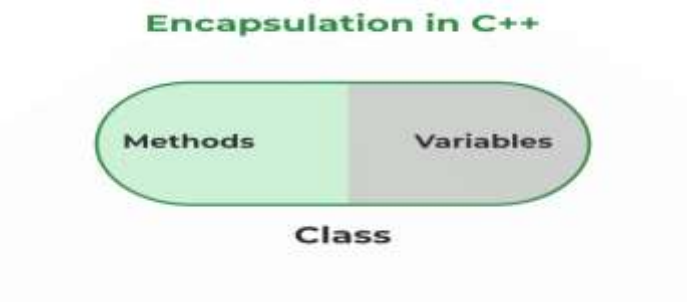
- An object is an identifiable actual entity with some characteristics and behavior. In C++, it is an instance of a class.
- For Example, the Animal class is just a concept or category, not an actual entity.
- But a black cat named jaguar is actual animal that exists. Similarly, classes are just the concepts and objects are the actual entity that belongs to that concept.

Example of Object

```
class Animal {
public:
    string species;
    int age;
    int name;

    // Member functions
    void eat() {
        // eat something
    }
    void sleep() {
        // sleep for few hrs
    }
    void makeSound () {
        // make sound;
    }
};

int main() {
   Animal  jaguar;
}
```

Encapsulation

- In simple 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 together in a class.
- Consider an example of the Animal class, the data members species, age and name are encapsulated with the member functions like eat(), sleep(), etc.
- They can be protected by the access specifier which hides the data of the class from outside.



Abstraction

- Abstraction is one of the most essential and important features of object-oriented programming in C++.
- Abstraction means displaying only essential information and ignoring the other details.
- Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation.
- In our case, when we call the makeSound() method, we don't need to know how the sound is produced internally, only that the method makes the animal sound.

Polymerphism

- The word polymorphism means having many forms.
- In simple words, we can define polymorphism as the ability of an entity to behave different in different scenarios.
- person at the same time can have different characteristics.
- For example, the same make Sound() method, the output will vary depending on the type of animal.
- So, this is an example of polymorphism where the make Sound() method behaves differently depending on the Animal type (Dog or Cat).

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.
- 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.

Benefits of OOPs

- **Modularity and Reusability**: OOP promotes modularity through classes and objects, allowing for code reusability.
- **Data Encapsulation**: OOP encapsulates data within objects, enhancing data security and integrity.
- **Inheritance**: OOP supports inheritance, reducing redundancy by reusing existing code.
- **Polymorphism**: OOP allows polymorphism, enabling flexible and dynamic code through method overriding.
- **Abstraction**: OOP enables abstraction, hiding complex details and exposing only essential features

Application of Object Oriented Programming

- **Real-Time System design:** Real-time system inherits complexities and makes it difficult to build them. OOP techniques make it easier to handle those complexities.
- **Hypertext and Hypermedia:** Hypertext is similar to regular text as it can be stored, searched, and edited easily. Hypermedia on the other hand is a superset of hypertext. OOP also helps in laying the framework for hypertext and hypermedia.
- **AI Expert System:** These are computer application that is developed to solve complex problems which are far beyond the human brain. OOP helps to develop such an AI expert System

Application of Object Oriented Programming

- **Stimulation and modeling system:** It is difficult to model complex systems due to varying specifications of variables. Stimulating complex systems require modeling and understanding interaction explicitly.
- **Object-oriented database:** The databases try to maintain a direct correspondence between the real world and database object in order to let the object retain it identity and integrity.

What is C++?

- C++ is a intermediate programming language that extends the C language by adding object-oriented features.
- It's known for its performance and versatility, making it suitable for systems programming, game development, and more.
- C++ is a cross-platform language that can be used to create high-performance applications.
- C++ was developed by Bjarne Stroustrup, as an extension to the C language.
- C++ gives programmers a high level of control over system resources and memory.

APPLICATION OF C++

- **1. Game Engines**

Real-time rendering for immersive 3D graphics.

Powerful tools for physics simulations and effects.

Cross-platform compatibility for a wider reach.

Companies: Epic Games, Unity, Crytek

2.Operating Systems

Manages hardware resources efficiently.

Provides fast and stable system-level performance.

Enables multitasking and robust memory management.

Companies: Microsoft, Apple, Canonical

APPLICATION OF C++

3. Embedded Systems

Lightweight and optimized for resource-constrained environments.

Enables real-time control for IoT and automotive systems.

Secure and reliable code for critical applications.

Companies: Bosch, Siemens, ARM

4 .Web Browsers

Fast webpage loading and smooth performance due to C++'s low-level control.

Optimized memory management and high efficiency for handling dynamic content.

Cross-platform compatibility ensures consistent user experiences.

Companies: Google (Chromium), Mozilla (Firefox), Microsoft (Edge)

Input output operators

- 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.

Input output operators

- **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;

Input output operators

Example of cout:

```
int main() {

   // Printing the given text using cout
    cout << “BCA";
    return 0;
}
```

Input output operators

- 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;

Input output operators

- Example of cin

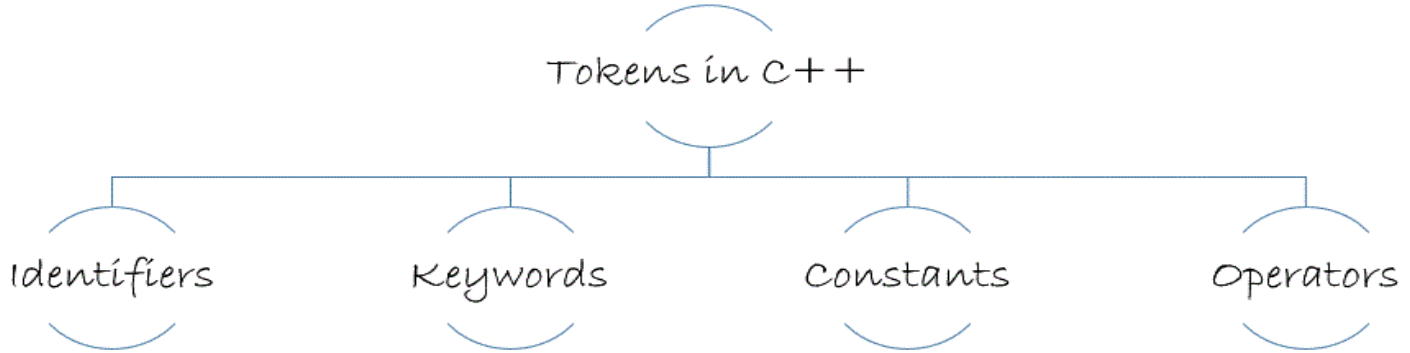
```
int main() {
    int age;

   // Taking input from user and store
   // it in variable
    cin >> age;

   // Output the entered age
    cout << "Age entered: " << age;
    return 0;
}
```

Introduction to namespace

- Namespace is a feature that provides a way to group related identifiers such as variables, functions, and classes under a single name.
- It provides the space where we can define or declare identifier i.e. variable, method, classes. In essence, a namespace defines a scope.
- A namespace definition begins with the keyword **namespace** followed by the namespace name as follows:
- **namespace name** { *// type1 mamber1 // type2 mamber2 // type3 mamber3 . . . . . }*

Token

- In C++, **tokens** can be defined as the smallest building block of C++ programs that the compiler understands.
- It is the smallest unit of a program into which the compiler divides the code for further processing.
- In this article, we will learn about different types of tokens in C++.



Token : Keyword

- Keywords 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++.

Tokens : Keyword

C and C++ Common Keywords			
auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

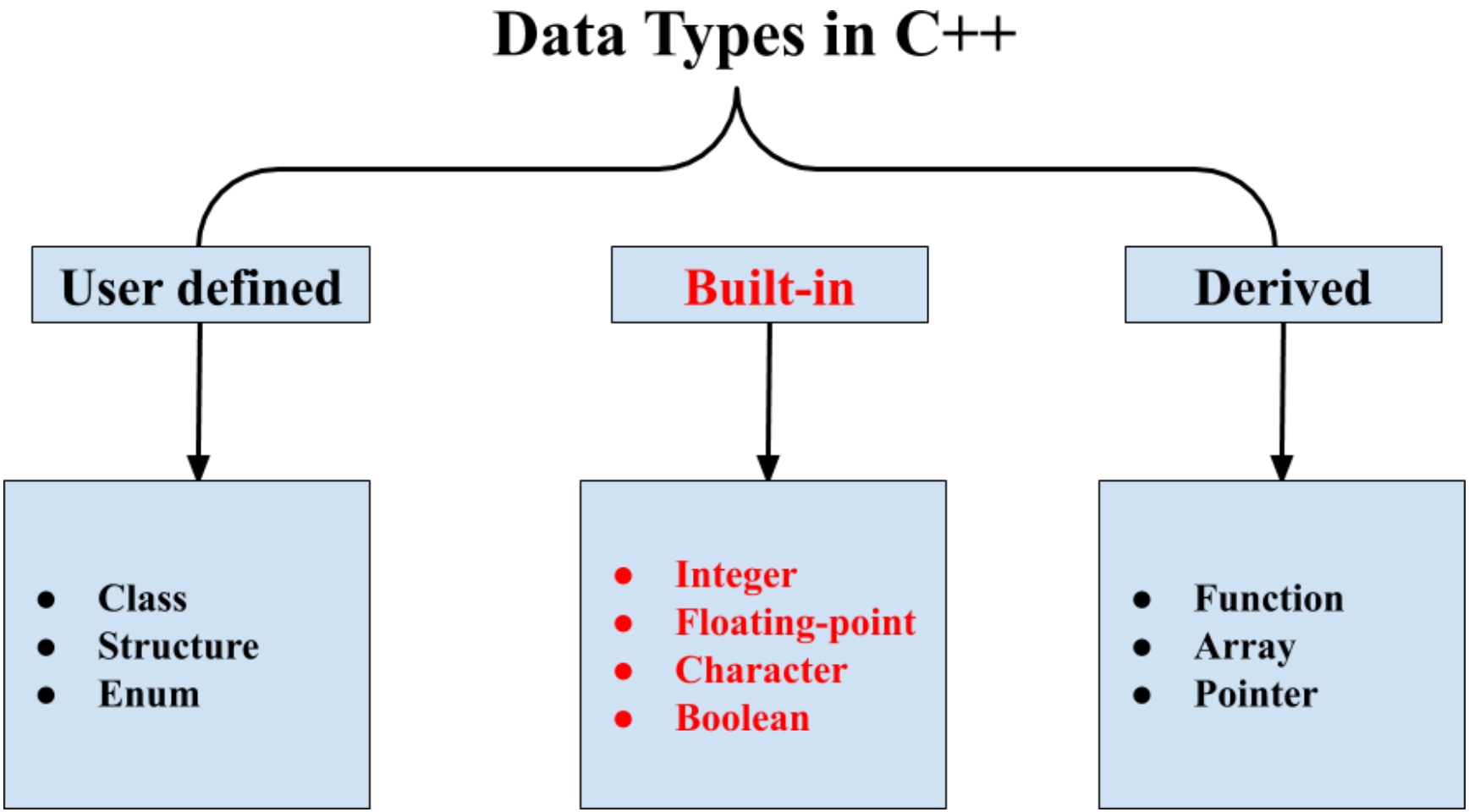
Tokens :Identifiers

- C++ allows the programmer to assign names of his own choice to variables, arrays, functions, structures, classes, and various other data structures called identifiers.
- The programmer may use the mixture of different types of character sets available in C++ to name an identifier.

Tokens : Identifiers

- **Rules for C++ Identifiers**
- First character should not begin with numbers.
- No special characters
- No keywords.
- No white spaces.
- Word limit.
- Case sensitive.

Tokens:Basic Datatypes



Tokens:Basic Datatype

- These data types are built-in or predefined data types and can be used directly by the user to declare variables example: int, char, float, bool, etc.

Integer Data type :

- Integer data types represent whole numbers without a fractional or decimal part. They can be signed (positive, negative, or zero) or unsigned (only positive or zero).
- **Example :**
  - int signedInt = -42;
  - unsigned int unsignedInt = 123;

Tokens :Basic datatypes

- **Character :**
  - Character data types represent individual characters from a character set, like ASCII or Unicode. In C++, ‘char’ is commonly used to represent characters.
  - **Example :**
    - char myChar = 'A';
- **Boolean :**
  - Boolean data types represent binary values, typically used for true (1) or false (0) conditions. In C++, `bool` is used for Boolean data.
  - **Example :**
    - bool isTrue = true; bool isFalse = false;

Tokens: Boolean Datatypes

- **Floating Point :**
  - Floating-point data types represent numbers with a fractional part.
  - In C++, ‘float’ is a single-precision floating-point type.
  - **Example :**
  - float myFloat = 3.14159;

Tokens: Basic Data type

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

Tokens:User-defined Data type

- In a programming language, **User Defined Data Types** are defined by the users in the program as per their needs in order to store data either of the same or different types as per the requirement.
- **User Defined Data types** or Composite Data types are derived from more than one built-in data type which is used to store complex data

Tokens:User-defined Data type

- **Class** – A User-defined data type that holds data members and functions whose access can be specified as private, public, or protected.

- It uses the '**class**' keyword for defining the data structure.

- class <classname>
- {
  - private:
    - Data_members;
    - Member_functions;
- };

Tokens:User-defined Data type

- **Structure** – A structured data type is used to group data items of different types into a single type. For Eg. Structure can be Address, in which it further contains information such as Flat number, Building name, street, city, state, pin code, etc.
- It is defined by using the '**struct**'
- **Example :**
- struct stud_id
- {
- Char name[10];
- Int roll_number;
- Char address[30];
- };

Tokens:User-defined Data type

- **Enumeration –** It helps assign names to integer constants in the program.
- The keyword '**enum**' is used.
- It is used to increase the readability of the code.
- Example :
- enum fruits{apple, mango, grapes};
- Here,apple will be assigned with **0** mango **1** and grapes **2**
- We can also assign specified numer to particular item to…
- Enum week{mon=1, tue, wed, thu, fri, sat, sun};

Tokens:Derived data type

- The data types that are derived from the primitive or built-in data types are referred to as Derived Data Types.
  - Function
  - Array
  - Pointer

Tokens : Derived data types

Functions

- A Function is a block of code or program segment that is defined to perform a specific well-defined task.
- A function is generally defined to save the user from writing the same lines of code again and again for the same input.
- All the lines of code are put together inside a single function and this can be called anywhere required. main() is a default function that is defined in every program of C++.

- **Syntax**
- ReturnType FunctionName (parameters);
- **Example :**
- void fun(int a,int b)
- {
  - cout << “sum is “<<a+b;
- }

Tokens : Derived data types

- ARRAY
- An Array is a collection of items stored at continuous memory locations.
- The idea of array is to represent many instances in one variable.
- **Syntax :**
- DataType ArrayName[size_of_array];
- **Example :**

  int arr[5];

  arr[0] = 5;
- arr[2] = -10;

Tokens : Derived data types

- Pointer
- Pointers are symbolic representation of addresses. They enable programs to simulate call-by-reference as well as to create and manipulate dynamic data structures.
- **Syntax :**

datatype *var_name;

- **Example :**

```
int var = 20;
int* ptr;
ptr = &var;
cout << "Value at ptr = " << ptr << "\n";
cout << "Value at var = " << var << "\n";
cout << "Value at *ptr = " << *ptr << "\n";
```

Tokens:Symbolic Constant

- Symbolic constants are nothing more than a label or name that is used to represent some fixed value that never changes throughout the course of a program.
- For example, one might define PI as a constant to represent the value 3.14159.
- In C++ we can create symbolic constant by many ways :
- Using the **const keyword**
  - const int size=15;
  - char name[size];
- Using **enum keyword**
  - enum{x=100,y,z};//value of x is 100 value of y is 101 value of z is 102
- Using **#define directive...**
  - #define var 100
- Constant cannot redefine during program execution.

Token: Type Compatible

- Type compatibility refers to whether the types of all variables are compatible to one another or not when written in an expression.
- **Example :**
- 3 variable of different data types in an expression must be compatible to each other.
- Example :
  - int a=10;
  - float b=20;

    double c;
  - c=a+b;
- Here, in expression c=a+b, a(int) and b(float) and c(double) all are different but it executes program without error because of compatibility.

Token: Declaration of variable

- Variables are used to store values.
- variable name is the name of memory location where value is stored.
- It must be alphanumeric, only underscore is allowed in a variable name.
- It is composed of letters, digits and only underscore. It must begin with alphabet or underscore.
- It can not be begin with numeric.
- Declaration will allocate memory for specified variable with garbage value.
- **Syntax :**
  - Data-Type Variable-name;
- **Example :**
  - int a;
    char c;

Token :Dynamic Initialization Variable

- As the declaration of variables, all the variables must be
- initialized before using them.
- In C++ we can initialize your variables dynamically (runtime) also.
- So the variable will get its initial value when you run the program.
- **Example :**

```
void main()
{
    int number;
    cout<<"Enter Number :";
    cin>>number;
    double range=number+1;//Dynamic initialization
    for(int x=1;x<=number;x++)
    {
        cout<<x<<endl;
    }
```

Token : Reference Variable

- An ordinary variable is a variable that contains the value of some type. (int a=10)
- A pointer is a variable that stores the address of another variable. It can be dereferenced to retrieve the value to which this pointer points to. (int *b=&a)
- There is another variable that C++ supports, i.e., references. It is a variable that behaves as an alias for another variable.(int &b=a)

- When a variable is declared as a reference, it becomes an alternative name for an existing variable.
- A variable can be declared as a reference by putting '&' in the declaration.
- Also, we can define a reference variable as a type of variable that can act as a reference to another variable.

Operators: Arithematic Operators

Name	Symbol	Description
Addition	+	Adds two operands.
Subtraction	-	Subtracts second operand from the first.
Multiplication	*	Multiplies two operands.
Division	/	Divides first operand by the second operand.
Modulo Operation	%	Returns the remainder an integer division.
Increment	++	Increase the value of operand by 1.
Decrement	--	Decrease the value of operand by 1.

Operators: Arithematic Operators

```
int main() {
    int a = 8, b = 3;

    // Addition
    cout << "a + b = " << (a + b) << endl;

    // Subtraction
    cout << "a - b = " << (a - b) << endl;

    // Multiplication
    cout << "a * b = " << (a * b) << endl;

    // Division
    cout << "a / b = " << (a / b) << endl;

    // Modulo
    cout << "a % b = " << (a % b) << endl;

    // Increament
    cout << "++a = " << ++a << endl;

    // Decrement
    cout << "b-- = " << b--;

    return 0;
}
```

Operators: Relational Operators

Name	Symbol	Description
Is Equal To	==	Checks both operands are equal
Greater Than	>	Checks first operand is greater than the second operand
Greater Than or Equal To	>=	Checks first operand is greater than equal to the second operand
Less Than	<	Checks first operand is lesser than the second operand
Less Than or Equal To	<=	Checks first operand is lesser than equal to the second operand
Not Equal To	!=	Checks both operands are not equal

Operators: Relational Operators

```
int main() {
    int a = 6, b = 4;

    // Equal operator
    cout << "a == b is " << (a == b) << endl;

    // Greater than operator
    cout << "a > b is " << (a > b) << endl;

    // Greater than Equal to operator
    cout << "a >= b is " << (a >= b) << endl;

    //  Lesser than operator
    cout << "a < b is " << (a < b) << endl;

    // Lesser than Equal to operator
    cout << "a <= b is " << (a <= b) << endl;

    // Not equal to operator
    cout << "a != b is " << (a != b);

    return 0;
}
```

Operators:Logical Operators

Name	Symbol	Description
Logical AND	&&	Returns true only if all the operands are true or non-zero.
Logical OR	||	Returns true if either of the operands is true or non-zero.
Logical NOT	!	Returns true if the operand is false or zero.

Operators:Logical Operators

```
int main() {
    int a = 6, b = 4;

    // Logical AND operator
    cout << "a && b is " << (a && b) << endl;

    // Logical OR operator
    cout << "a || b is " << (a || b) << endl;

    // Logical NOT operator
    cout << "!b is " << (!b);

    return 0;
}
```

Operators:Bitwise Operators

Name	Symbol	Description
Binary AND	&	Copies a bit to the evaluated result if it exists in both operands
Binary OR	|	Copies a bit to the evaluated result if it exists in any of the operand
Binary XOR	^	Copies the bit to the evaluated result if it is present in either of the operands but not both
Left Shift	<<	Shifts the value to left by the number of bits specified by the right operand.
Right Shift	>>	Shifts the value to right by the number of bits specified by the right operand.
One's Complement	~	Changes binary digits 1 to 0 and 0 to 1

Operators:Bitwise Operators

```
int main() {
    int a = 6, b = 4;

    // Binary AND operator
    cout << "a & b is " << (a & b) << endl;

    // Binary OR operator
    cout << "a | b is " << (a | b) << endl;

    // Binary XOR operator
    cout << "a ^ b is " << (a ^ b) << endl;

    // Left Shift operator
    cout << "a << 1 is " << (a << 1) << endl;

    // Right Shift operator
    cout << "a >> 1 is " << (a >> 1) << endl;

    // One's Complement operator
    cout << "~(a) is " << ~(a);

    return 0;
}
```

Operator:Assignment Operator

Name	Symbol	Description
Assignment	=	Assigns the value on the right to the variable on the left.
Add and Assignment	+=	First add right operand value into left operand then assign that value into left operand.
Subtract and Assignment	-=	First subtract right operand value into left operand then assign that value into left operand.
Multiply and Assignment	*=	First multiply right operand value into left operand then assign that value into left operand.
Divide and Assignment	/=	First divide right operand value into left operand then assign that value into left operand.

Operator:Assignment Operator

```
int main() {
    int a = 6, b = 4;

    // Assignment Operator.
    cout << "a = " << a << endl;

    //  Add and Assignment Operator.
    cout << "a += b is " << (a += b) << endl;

    // Subtract and Assignment Operator.
    cout << "a -= b is " << (a -= b) << endl;

    //  Multiply and Assignment Operator.
    cout << "a *= b is " << (a *= b) << endl;

    //  Divide and Assignment Operator.
    cout << "a /= b is " << (a /= b);

    return 0;
}
```

Operators:Ternary Operators

Syntax:

```
Expression1 ? Expression2 : Expression3
```

In the above statement:

- The ternary operator **?** determines the answer on the basis of the evaluation of **Expression1**.
- If **Expression1** is true, then **Expression2** gets evaluated.
- If **Expression1** is false, then **Expression3** gets evaluated.

Operators:Ternary Operators

```
int main() {
    int a = 3, b = 4;

    // Conditional Operator
    int result = (a < b) ? b : a;
    cout << "The greatest number "
         "is " << result;

    return 0;
}
```

Special Operators

- **Special Operators :**
- C++ language supports many special operators too. They are used for some special processes in C++ programming.
- The special operators used in C++ programming language are,
- Comma (,)
- Pointer Operators (& and *)
- Member Selection (. and ->)
- sizeof( ) operator

Operators

- Let's Have Look Some Other important Operators in C++ :
    1. scope resolution operator
    2. member referencing operator
    3. memory management operator
    4. manipulators

Operators

1. Scope Resolution Operators :

- In a program, we can have same variable names in different blocks.
- The variable of outer block can be access by inner block.
- The variables declared in outer block are known as the global variable and the variables declared in the inner block are referred as local variables.
- Thus the local and global variables define their scopes regarding how long they are visible.
- You can use scope resolution operator to access the global variable in C++.

Operators

- **Syntax :**
- ::variable_name;
- **Example :**
- int a=111; void main( )
- {
  - int a=10;
  - cout<<"Local A ="<<a;
  - **cout<<"Global A ="<<::a;**
- }

- **Uses of the scope resolution Operator :**
- It is used to access the hidden variables or member functions of a program.
- It defines the member function outside of the class using the scope resolution.
- It is used to access the static variable and static function of a class.
- The scope resolution operator is used to override function in the Inheritance.

➢ Program to define the member function outside of the class using the scope resolution (::) operator

- **Example :**

```
#include<conio.h>
#include<iostream.h>
   class operate
   {
         public:
         void fun();
   };
   void operate::fun()
   {
         cout<<"this is fun()";
   }
   void main()
   {
         clrscr();
         operate op;
         op.fun();
         getch();
   }
```

2.Member Referencing Operator

- In C++, a **reference** works as an alias for an existing variable, providing an alternative name for it and allowing you to work with the original data directly.
- **Example:**
- #include <iostream>
- int main() {
- int x = 10;
- // ref is a reference to x
- int& ref = x;
- // printing value using ref
- cout << ref << endl; // Changing the value and printing again
- ref = 22;
- cout << ref;
- return 0;
- }

Member Referencing Operator

- **Explanation**: In this program, **ref** is a reference to the variable **x**, meaning **ref** is just another name for **x**. When the value of **ref** is modified, it directly changes the value of **x**, since both **ref** and **x** refer to the same memory location.

3.Memory Management Operator

- Dynamic memory allocation in C/C++ refers to performing memory allocation manually by a programmer. Dynamically allocated memory is allocated on **Heap,** and non-static and local variables get memory allocated on **Stack.**

- C++ also define two Unary Operators :
    1. **new**
    2. **delete**

- New and Delete Operators performs the task of allocating and freeing the memory. Since, these Operators manipulates memory on the Free Store, They are also known as **Free Store Operators.**

3.Memory Management Operator

- **new operator :**
- The new operator denotes a request for memory allocation on the Free Store.
- If sufficient memory is available, a new operator initializes the memory and returns the address of the newly allocated and initialized memory to the pointer variable.
- **Syntax :**
  - pointer-variable = **new** data-type;
  - pointer-variable = **new** data-type[size]; //if array
- **Example :**

```
#include
<iostream.h>
void main()
{
  int* ptr=new int;
*ptr = 10;
cout << "Address: " << ptr << endl;
cout << "Value: " << *ptr;
}
```

3.Memory Management Operator

- Delete operator :
- When memory is no longer required, then it needs to be deallocated so that the memory can be used for another purpose. This can be achieved by using the delete operator.
- **Syntax:**
  - **delete** pointer-variable;
- **Example :**
- Delete with pointer :
- delete p;
- delete q;
- delete with an array :
- delete [ ]arr;

4.Manipulators

- **Manipulators** are helping functions that can modify the input or output stream. They can be included in the I/O statement to alter the format parameters of a stream. They are defined inside <iomanip> and some are also defined inside <iostream> header file.

Manipulators :Input Stream Manipulators

Manipulator	Description	Header File
ws	Skips leading whitespaces in the input stream.	iostream
noskipws	Disables skipping of leading whitespaces.	iostream

Manipulators :Input Stream Manipulators

```
int main() {
    char c1, c2;

    // Input skips whitespace by default
    cin >> c1;

    // Input the next character without skipping whitespace
    cin >> noskipws >> c2;

    cout << "c1: " << c1 << ", c2: " << c2;
    return 0;
}
```

Manipulators:Base manipulators

Manipulator	Description	Header File
hex	Formats output in hexadecimal base.	iostream
dec	Formats output in decimal base.	iostream
oct	Formats output in octal base.	iostream

Manipulators:Base manipulators

```
int main() {
    int n = 42;

    // Output in hexadecimal base
    cout << hex << n << endl;

    // Output in decimal base
    cout << dec << n << endl;

    // Output in octal base
    cout << oct << n;

    return 0;
}
```

Type casting

- Type casting, or type conversion, *is a fundamental concept in programming that involves* ***converting one data type into another****. This process is crucial for ensuring compatibility and flexibility within a program.*

Type Casting

Difference between Implicit and Explicit Type Casting:

Type Casting	Implicit (Automatic)	Explicit (Manual)
Definition	Conversion performed by the compiler without programmer's intervention.	Conversion explicitly specified by the programmer.
Occurrence	Happens automatically during certain operations or assignments.	Programmer needs to explicitly request the conversion.
Risk	Generally safe but may lead to loss of precision or unexpected results.	Requires careful handling by the programmer to prevent data loss or errors.
Syntax	No explicit syntax needed; the conversion is done automatically.	Requires explicit syntax, such as type casting functions or operators.

Examples:	int a = 4.5; // float to int conversion
	
	char ch = 97; // int to char conversion
		int a = 5, b = 6;
		float x = (float) a/b; // int to float
		
		int a = 1000000, b = 1000000;
		long long x = (long long)a * b; // int to long long