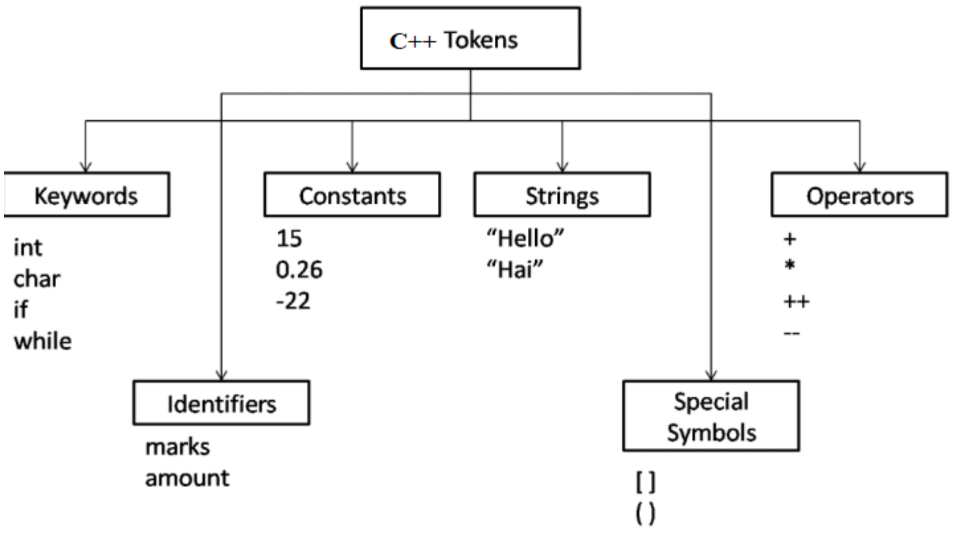
1.3. TOKENS

Tokens are the **smallest individual units** (building blocks) of a C++ program that the compiler recognizes. Every C++ program is made up of tokens.

Types of tokens:

C++ tokens are classified into **six main categories**:

1. **Keywords**
2. **Identifiers**
3. **Constants**
4. **Strings**
5. **Operators**
6. **Punctuation / Special symbols**



1.3.1 Keywords

- ✓ **Keywords are reserved words** in C++ that have specific, predefined meanings and purposes within the language.
- ✓ Because they serve special roles in the compiler’s interpretation, **keywords cannot be used as variable names, identifiers, or function names** in a program.

- ✓ Common examples of C++ keywords include **int, float, char, if, else, for, while, class, public, private, return,** and **switch**.

1.3.2 Identifiers

- Identifiers are **names given by the programmer** to variables, functions, classes, objects, arrays, etc.
- Must follow rules:
    - Can contain letters, digits, and underscore.
    - No special characters except _
    - Cannot start with a digit
    - Cannot be a keyword
    - Case-sensitive
- Examples: total, marks, Student1, sum_of_numbers

1.3.3 Constants (Literals)

- Constants are **fixed values** that do not change during program execution.

Types:

- ✓ **Integer constants**, such as *10* or *−5*, which represent whole numbers.
- ✓ **Floating-point constants**, such as *3.14* or *0.005*, which represent real numbers with decimal points.
- ✓ **Character constants**, such as 'A' or '5', which are enclosed in single quotes and represent a single character.
- ✓ **String constants**, such as "Hello", which consist of a sequence of characters enclosed in double quotes.
- ✓ **Boolean constants**, such as *true* and *false*, which represent logical truth values.

1.3.4 Strings

- ✓ A **string** is a sequence of characters enclosed in **double quotes**, used to represent text in C++.
- ✓ For example, "C++ Programming" is a valid string.
- ✓ In memory, every string is automatically **terminated with a null character ('\0')**, which indicates the end of the string.

1.3.5 Operators

Operators are **special symbols** that perform specific operations on variables and constants. They allow manipulation of data and form expressions in C++.

Types:

- **Arithmetic operators**: +, -, *, /, % used for mathematical operations.
- **Relational operators**: ==, !=, >, <, >=, <=
- **Logical operators**: &&, ||, !
- **Assignment operators**: =, +=, -=, *=, /=
- **Increment/Decrement**: ++, --
- **Ternary operator**: ? :

1.3.6 Punctuation / Special Symbols

Punctuation marks and special symbols are used to **structure C++ programs** and define the syntax of the language.

Examples:

- ✓ **{ }** which define a block of code, typically used in functions, loops, and conditional statements.
- ✓ **( )** which are used for function parameters and grouping expressions.

- ✓ **[ ]** which denote array subscripts and are used to access array elements.
- ✓ **;** which acts as a **statement terminator**, marking the end of a command.
- ✓ **,** which is used as a **separator** for multiple variables or function arguments.
- ✓ **:** which may be used for labels, the conditional operator, or public/private specifiers in classes.
- ✓ **#** which denotes a **pre-processor directive**, such as #include or #define.

Example :

```
int sum = a + b;
```

Tokens in this statement:

- int → Keyword
- sum → Identifier
- = → Operator
- a → Identifier
- + → Operator
- b → Identifier
- ; → Special symbol

1.4. EXPRESSIONS

An expression is a **combination of variables, constants, and operators** that produces a value.

1.4.1 Arithmetic Expressions

- Arithmetic expressions use arithmetic operators such as **+**, **−**, *****, **/**, and **%** to perform mathematical calculations.

- Example:

  a + b * 2,

  x % 10

1.4.2 Relational Expressions

- Relational expressions are used to **compare two values**, and they always produce a result that is either **true (1)** or **false (0)**.
- Example:

  a > b, x == y

1.4.3 Logical Expressions

- Logical expressions combine two or more relational expressions using logical operators
- Operators such as && (AND), || (OR), ! (NOT)
- Example:

  (a > b) && (x < y)

  where the final result depends on the truth values of the combined conditions.

1.4.4 Assignment Expressions

- Assignment expressions are used to **assign values to variables** using the assignment operator or compound assignment operators.
- Examples:

  x = 10

  a += 5 (same as a = a + 5)

1.5 **CONTROL STRUCTURES IN C++**

✓ Control structures determine **how the flow of execution** proceeds in a C++ program.

- ✓ They allow the programmer to **control decision-making, repetition, and jumps** in program flow.
- ✓ Control structures make programs **logical, efficient, and modular**.

C++ provides three main categories:

1. **Decision-Making Statements**
2. **Looping Statements**
3. **Jump Statements**

Each category is explained below with **definitions, features, syntax, and examples**.

1.5.1 Decision-Making Statements

Decision-making statements allow the program to **choose** between different actions based on a condition.

- ✓ Decision-making statements are **used for conditional execution**, allowing a program to choose whether a specific block of code should run or not.
- ✓ The conditions used in these statements are usually **relational or logical expressions** that compare values or evaluate logical relationships.
- ✓ These conditions always **produce a result of either true (1) or false (0)**, which determines the flow of execution.
- ✓ They **help implement logical decisions in real-world situations**, enabling programs to react differently based on varying inputs or conditions.

Types of Decision-Making Statements

1. **if Statement**

2. **if–else Statement**
3. **Nested if–else**
4. **switch Statement**

1. if Statement

The **if statement** executes a block of code **only when the specified condition evaluates to true**.

Features

- It is the **simplest form of decision-making control** in C++.
- It does **not provide an alternative block**, meaning the statements inside it run only if the condition is true.
- It is primarily used for **testing a single condition** before executing an action.

Syntax

```
if (condition)
{
    statements;
}
```

Example

```
if (marks >= 50)
{
    cout << "Pass";
}
```

- If the condition evaluates to **false**, the entire block of statements inside the *if* statement is simply **skipped**.
- It is most useful for **simple checks or validations**, such as testing user input, checking limits, or verifying conditions before execution.

2. if–else Statement

The **if–else statement** executes one block of code when the condition is true and executes a different block when the condition is false.

Features

- It provides **two-way decision control**, allowing the program to choose between two possible sets of actions.
- It ensures that **one of the two blocks always executes**, regardless of the condition.

Syntax

```
if (condition)
{
   true-block;
}
else
{
   false-block;
}
```

Example

```
if (mark >= 50)
{
   cout << "Pass";
}
else
{
   cout << "Fail";
}
```

- The if–else structure is particularly useful in **binary choice situations**, such as yes/no decisions, pass/fail conditions, or true/false outcomes.

3. Nested if–else

A **nested if–else** occurs when an *if* or *else* block contains another *if–else* statement inside it. This creates a multi-level decision structure, allowing more complex decision-making.

Example

```
if (Mark >= 50)
    cout << "Pass";
else if (Mark < 50)
    cout << "Fail";
else
    cout << "Absent";
```

- Nested if–else statements are used when **multiple conditions** must be checked in sequence.
- They work well for tasks like assigning grades, categorizing values, or making multi-stage decisions.

4. switch Statement

Switch Statement is used when there are **multiple possible choices** for a variable or expression.

- ✓ The **switch statement replaces long else-if ladders**, making the program easier to read and maintain when multiple conditions depend on the same variable.
- ✓ It **supports only integral data types**, such as int, char, and enum, for its expression and case labels.
- ✓ The switch statement **uses case labels to match specific values, and the break statement is required to stop fall-through** and prevent execution from continuing into the next case.

Syntax

```
switch(expression)
```

```
{
    case value1:
        statements;
        break;

    case value2:
        statements;
        break;

    default:
        statements;
}
```

Example

```
int day = 3;
switch(day)
{
    case 1:
          cout << "Monday";
          break;
    case 2:
        cout << "Tuesday";
        break;
    case 3:
        cout << "Wednesday";
        break;
default:
        cout << "Invalid day";
}
```

- ✓ The **break statement is required to prevent the execution of the next case**, ensuring that the program exits the switch block once the matching case has been executed.
- ✓ The **default case is optional but recommended**, as it handles unexpected or unmatched values and provides a safe fallback in the switch structure.

1.5.2. Looping (Iterative) Statements

- ✓ Looping statements execute a block of code **repeatedly until a specified condition becomes false**, allowing a program to perform repeated actions efficiently.
- ✓ They **help reduce code repetition** by avoiding the need to write the same statements multiple times.
  Looping structures are commonly **used for tasks such as counting, summing values, and searching through data**.
- ✓ Every loop generally consists of three main parts: **initialization**, where the loop control variable is set; **condition checking**, which determines whether the loop should continue; and **an update step** (increment or decrement), which modifies the control variable during each iteration.

- ✓ Loop consists of:
  - Initialization
  - Condition checking
  - Update (increment/decrement)

1. Looping Statements

- ✓ Looping statements in C++ are used to **execute a block of code repeatedly** as long as a specified condition remains true.
- ✓ They help **reduce code repetition** and are commonly used for **counting, summing, searching, or processing arrays**.

i) for Loop

The **for loop** is used when the number of iterations is **known in advance**, making it ideal for count-controlled loops.

Syntax

```
for (initialization; condition; update)
{
        statements;
}
```

Example: Print numbers 1 to 5

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

- All loop control elements such as **initialization, condition, and update**—are written in a single line.
- Preferred for **count-controlled loops**, where the exact number of repetitions is known beforehand.

ii) while Loop

The **while loop** checks the condition first and then executes the block. It is used when the number of iterations is **not known beforehand**.

Syntax

```
while (condition)
 {
   statements;
}
```

Example

```
int i = 1;
while(i <= 5)
{
        cout << i << " ";
        i++;
}
```

- If the condition is **false initially**, the loop **will not execute even once**.
- Useful for **condition-controlled loops** where the number of repetitions depends on dynamic data.

iii) do–while Loop

The **do–while loop** executes the block **at least once** because the condition is checked **after the block**.

Syntax

```
do
{
   statements;
} while(condition);
```

Example

```
int i = 1;
do
{
        cout << i << " ";
        i++;
} while(i <= 5);
```

When to Use

- Creating **menus** in console programs
- Loops that depend on **user input**
- **Validation tasks** where the code must run at least once

1.5.3. Jump Statements

Jump statements provide an **unconditional change in the flow of execution** in a program. They are often used to **control loops or exit blocks early**.

a) break

Definition

The **break statement** terminates the **nearest enclosing loop or switch block** immediately.

Example

```
for(int i = 1; i <= 10; i++)
 {
          if(i == 5) break;
         cout << i << " ";
}
```

- Prevents **infinite loops**.
- Can only be used **inside loops or switch statements**.

b) continue

The **continue statement skips the current iteration** of a loop and moves to the **next cycle**.

Example

```
for(int i = 1; i <= 5; i++)
 {
     if(i == 3) continue;
    cout << i << " ";
}
```

- Useful when you want to **skip specific values or conditions** without terminating the loop.

c) goto

The **goto statement** transfers control **unconditionally** to a labelled statement elsewhere in the program.

Syntax

```
goto label;
// some statements
label:
   statements;
```

Example

```
int i = 1;
start:
cout << i << " ";
i++;
if(i <= 5) goto start;
```

- Not recommended in modern C++ programming because it can make the code **hard to read and debug**.
- Prefer structured loops over goto for clarity and maintainability.