Tokens in C++:

In C++, **tokens** are the smallest individual units in a program. They are the building blocks of a C++ program, and the compiler uses these tokens to understand and execute the code. Every C++ program is composed of tokens.

Types of Tokens in C++

1. **Keywords**
2. **Identifiers**
3. **Constants**
4. **Strings**
5. **Special Symbols**
6. **Operators**

---

1. Keywords

- **Definition**: Reserved words in C++ that have predefined meanings and cannot be used as names for variables, functions, or other identifiers.
- **Characteristics**:
    1. Written in lowercase.
    2. Cannot be modified or redefined.
    3. Serve specific purposes in the program.
- **Examples**:
    - **Data types**: int, float, char, bool, double, void
    - **Control statements**: if, else, for, while, do, switch, case, break, continue
    - **Access specifiers**: public, private, protected
    - **Others**: class, struct, return, sizeof, new, delete
- **Purpose**: Keywords help define the structure and behavior of the program by specifying data types, control flow, and access levels.

```
int main() {
   int x = 10;  // 'int' and 'return' are keywords
   return 0;
}
```

- **Rules**:
    1. Keywords must be written exactly as defined (case-sensitive).
    2. Cannot use keywords as variable names or identifiers.
    3. Improper usage results in compilation errors.

---

2. Identifiers

- **Definition**: Names used by programmers to identify variables, functions, arrays, or other user-defined elements in the program.
- **Characteristics**:
    1. Must begin with a letter (A-Z or a-z) or an underscore (_).
    2. Can contain letters, digits (0-9), and underscores.
    3. Case-sensitive (e.g., Variable and variable are different).
    4. Cannot use C++ keywords as identifiers.
- **Examples**:

```
int age = 25;          // 'age' is an identifier

float _salary = 45000;  // '_salary' is an identifier
```

- **Best Practices**:
    1. Use meaningful and descriptive names for identifiers (e.g., totalMarks, studentAge).
    2. Avoid starting identifiers with underscores, as these are often reserved for system or compiler usage.
    3. Follow a consistent naming convention like camelCase or snake_case for better readability.
- **Rules**:
    1. Identifiers must not contain special symbols (e.g., @, #, $).
    2. Avoid excessively long names to ensure readability and maintainability.

---

3. Constants

- **Definition**: Fixed values that do not change during program execution.
- **Types**:
    1. **Integer constants**: Whole numbers like 10, -5.
    2. **Floating-point constants**: Decimal values like 3.14, -0.001.
    3. **Character constants**: Single characters enclosed in single quotes, like 'a', '9'.

4. **Boolean constants**: true and false.
5. **Enumerations**: Defined constants using enum.

- **Purpose**: Constants provide values that remain consistent throughout the program. They improve code reliability and readability.
- **Examples**:

```
const int MAX = 100;       // MAX is a constant

#define PI 3.14159        // Preprocessor constant
```

- **Rules**:
    1. Use the const keyword to declare constants.
    2. Constants are immutable after their definition.
    3. Use meaningful names in uppercase for constants to distinguish them from variables.
    4. Avoid magic numbers (hardcoded values); use constants instead.

---

4. Strings

- **Definition**: A sequence of characters enclosed in double quotes ("), representing text.
- **Characteristics**:
    1. Strings are stored as arrays of characters ending with a null character (\0).
    2. Represented by the std::string class or character arrays.
- **Purpose**: Strings allow the handling of textual data in a program, including names, messages, or any textual information.
- **Examples**:

```
#include <iostream>

#include <string>


int main() {

   std::string name = "John Doe";  // String using std::string class

   char greeting[] = "Hello";      // String as a character array

   std::cout << name << std::endl;

   return 0;

}
```

- **Rules**:
    1. Strings must be enclosed in double quotes.
    2. Single quotes are used only for character literals (e.g., 'A').
    3. Ensure proper inclusion of the <string> header when using std::string.

---

5. Special Symbols

- **Definition**: Symbols with predefined meanings that are used for specific purposes in C++.
- **Examples**:
    - **Curly braces ({})**: Define the scope of functions, loops, or conditional blocks.
    - **Parentheses (()):** Enclose function arguments or control expressions.
    - **Square brackets ([])**: Used for arrays.
    - **Semicolon (;)**: Ends a statement.
    - **Comma (,)**: Separates multiple items.
    - **Pound sign (#)**: Used for preprocessor directives.
- **Purpose**: Special symbols structure the program, ensuring logical grouping, separation, and proper execution.
- **Examples**:

```
#include <iostream>   // '#' for preprocessor directive

int arr[5] = {1, 2, 3, 4, 5}; // '[]' for arrays, '{}' for initialization
```

- **Rules**:
    1. Always pair opening and closing symbols correctly (e.g., {} or []).
    2. Use semicolons to terminate statements.
    3. Ensure preprocessor directives start with #.

---

6. Operators

- **Definition**: Symbols used to perform operations on variables and values.
- **Types**:
    1. **Arithmetic operators**: +, -, *, /, %
    2. **Relational operators**: ==, !=, <, >, <=, >=
    3. **Logical operators**: &&, ||, !
    4. **Bitwise operators**: &, |, ^, ~, <<, >>

5. **Assignment operators**: =, +=, -=, *=, /=, %=
6. **Increment/Decrement operators**: ++, --

- **Purpose**: Operators allow manipulation of data, performing computations, and controlling program flow.
- **Examples**:

```
int a = 10, b = 5;

int sum = a + b;     // '+' is an arithmetic operator

if (a > b) {        // '>' is a relational operator

   std::cout << "a is greater";

}
```

- **Rules**:
    1. Operators must be used with compatible data types.
    2. Avoid ambiguous usage by using parentheses for precedence.
    3. Logical and bitwise operators are often used in control structures and low-level programming.

Comprehensive Example

```
#include <iostream>

using namespace std;

int main() {

   // Keywords and Identifiers

   const int MAX = 100;   // MAX is a constant

   int x = 10, y = 20;    // x and y are identifiers


   // Arithmetic Operators

   int sum = x + y;       // '+' is an operator


   // String and Output

   string greeting = "Hello, World!";

   cout << greeting << endl;  // '<<' is an operator


   // Conditional Block
```

```
    if (x < y) {          // 'if' is a keyword
        cout << "x is smaller than y" << endl;
    }

    return 0;
}
```