Unit 2: Basics of Programming

Introduction to Algorithms

1. Idea of an Algorithm

- **Definition**: An algorithm is a step-by-step procedure to solve a problem or perform a task. It is a well-defined sequence of instructions to achieve a particular goal.
- **Purpose**: Algorithms are used to break down complex problems into manageable steps that can be systematically executed.

2. Steps to Solve Logical and Numerical Problems

- **Understand the Problem**: Clearly define the problem and understand the requirements.
- **Break Down the Problem**: Divide the problem into smaller, more manageable parts.
- **Develop an Algorithm**: Write down the steps needed to solve the problem logically.
- **Test the Algorithm**: Ensure the algorithm works by applying it to different scenarios.
- **Optimize the Algorithm**: Improve efficiency by minimizing time and space complexity.

3. Characterstics of an Algorithm

1. **Input:** An algorithm must have zero or more inputs.
2. **Output:** An algorithm must produce one or more outputs.
3. **Definiteness:** Each instruction in an algorithm must be clear and unambiguous.
4. **Finiteness:** An algorithm must terminate after a finite number of steps.
5. **Effectiveness:** Each instruction in an algorithm must be simple and basic.

Algorithms, Flowchart, and Pseudocode

1. Algorithm

- **Example**: Algorithm to find the sum of two numbers:
    1. Start
    2. Input two numbers, `a` and `b`
    3. Calculate the sum: `sum = a + b`
    4. Output the sum
    5. End

2. Flowchart

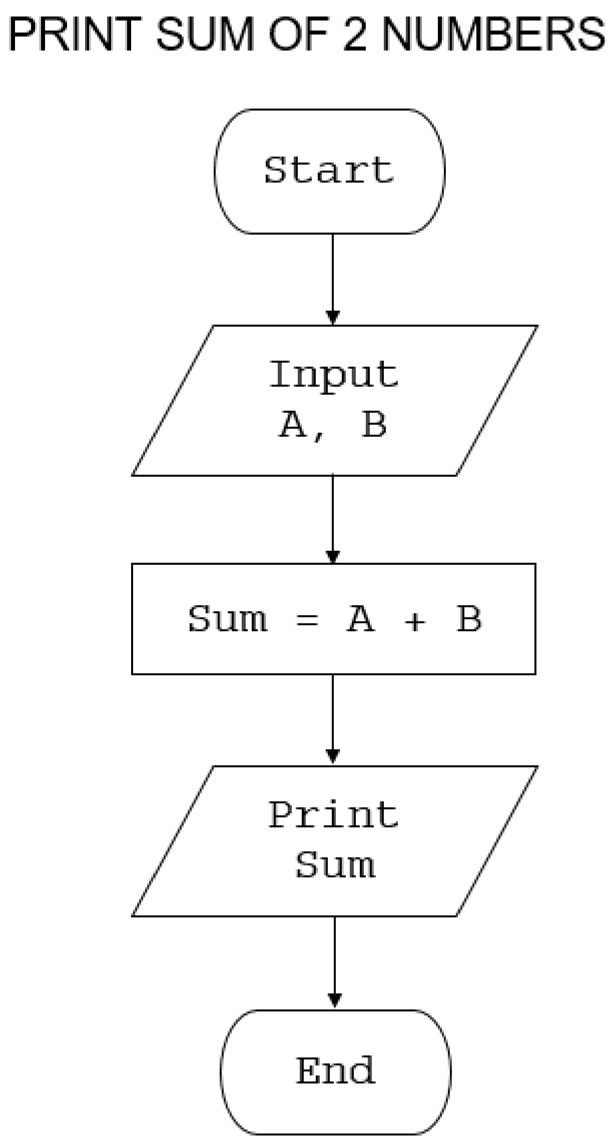
- **Definition**: A flowchart is a graphical representation of an algorithm using symbols to represent operations and arrows to show the flow of control.

- **Basic Symbols**:

    - **Oval**: Represents the start or end of the process.
    - **Rectangle**: Represents a process or operation.
    - **Parallelogram**: Represents input or output.
    - **Diamond**: Represents a decision or branching.

- **Example**: Flowchart for the above algorithm to find the sum of two numbers:

    - **Start → Input a, b → Process: sum = a + b → Output sum → End**



3. Pseudocode

- **Definition**: Pseudocode is an informal way of writing the algorithm in a way that resembles programming code but is written in plain English.
- **Example**: Pseudocode to find the sum of two numbers:

```
Start
Input a, b
sum = a + b
Print sum
End
```

Basic C++ Program

1. Basic Structure of a C++ Program

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

int main() {
    // Code to be executed
    return 0;
}
```

- `#include <iostream>` : This includes the standard input-output stream library.
- `using namespace std;` : This allows us to use the standard C++ library without prefixing `std::` .
- `int main()` : The main function is the entry point of the program.
- `return 0;` : Returns 0 to indicate the program ended successfully.

2. Compile and Execute a C++ Program

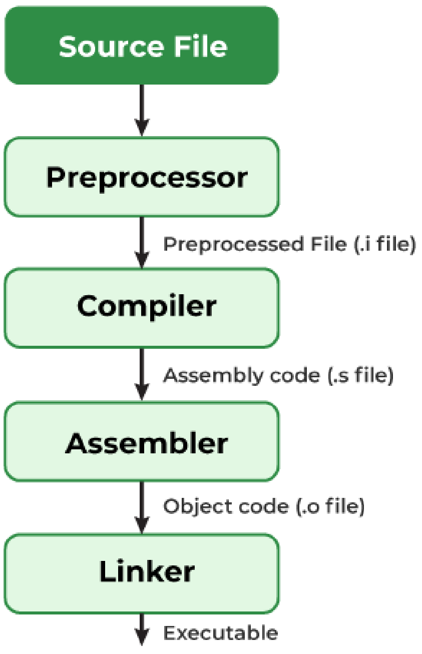
How do we compile and run a C++ program?

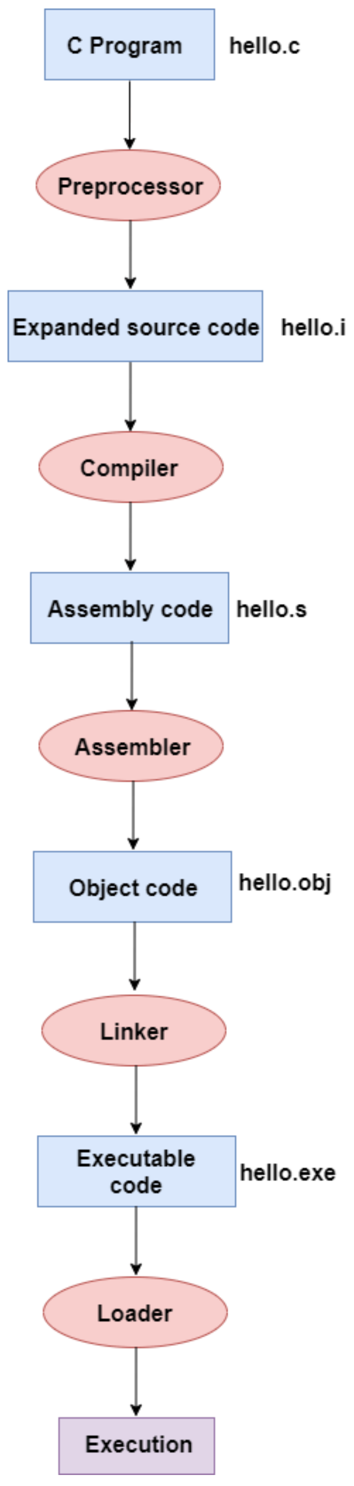
- **Compile**: Use a C++ compiler (e.g., `g++` on Linux) to convert the source code into an executable program.

  ```
  g++ program.cpp
  ```

- **Execute**: Run the compiled program.

  ```
  ./a.out
  ```



	 C Program hello.c Preprocessor Expanded source code hello.i Compiler Assembly code hello.s Assembler Object code hello.obj Linker Executable code hello.exe Loader Execution

There are four steps in the C++ compilation process that converts source code into machine-readable code:

1. Preprocessing
2. Compilation
3. Assembling
4. Linking

The following steps are taken to execute a program:

1. Firstly, the input file, i.e., **hello.cpp**, is passed to the preprocessor, and the preprocessor converts the source code into expanded source code. The extension of the expanded source code would be **hello.i**.
2. The expanded source code is passed to the compiler, and the compiler converts this expanded source code into assembly code. The extension of the assembly code would be **hello.s**.
3. This assembly code is then sent to the assembler, which converts the assembly code into object code.
4. After the creation of an object code, the linker creates the executable file. The loader will then load the executable file for the execution.

Step 1: Creating a C Source File: We first create a C program using an editor and save the file as filename.cpp

```
$ vi filename.cpp
```

We can write a simple hello world program and save it.

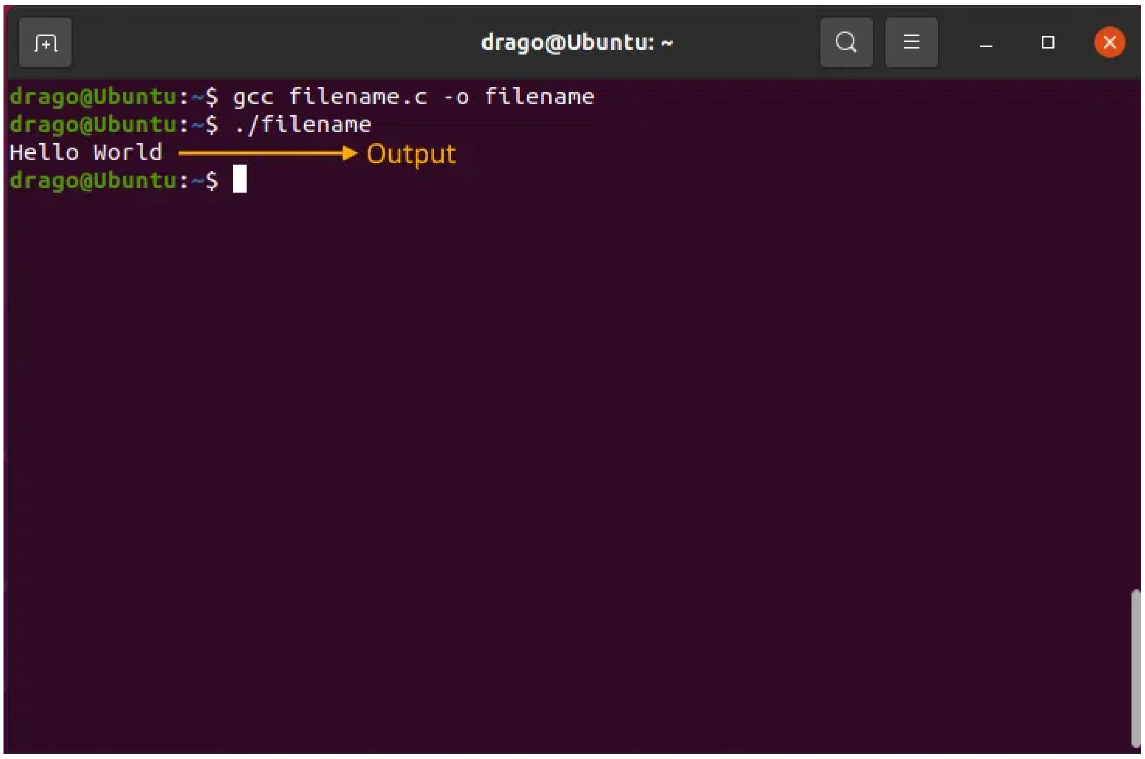
Step 2: Compiling using GCC compiler: We use the following command in the terminal for compiling our filename.cpp source file

```
$ gcc filename.cpp –o filename
```

Step 3: Executing the program: After compilation executable is generated and we run the generated executable using the below command.

```
$ ./filename
```

The program will be executed and the output will be shown in the terminal.



```
drago@Ubuntu:~$ gcc filename.c -o filename
drago@Ubuntu:~$ ./filename
Hello World ──────────► Output
drago@Ubuntu:~$ 
```

Character Set, Tokens, and Data Types

---

1. Character Set

- **Definition**: The set of valid characters that a programming language can recognize, including letters, digits, symbols, and control characters.
- Components of the Character Set:
  - Letters: A to Z (uppercase) and a to z (lowercase)
  - Digits: 0 to 9
  - Special Characters: Symbols like +, -, *, /, =, {, }, [,], (,), ;, :, @, #, $, &, etc.
  - Whitespace Characters: Space ( ), tab (\t), newline (\n)
  - Escape Sequences: Special characters starting with a backslash, like \n (newline), \t (tab), \ (backslash), ' (single quote), " (double quote)

2. Tokens

- **Definition**: The smallest units in a C++ program.
- **Types**:
  - **Identifiers**: Names given to variables, functions, etc. (e.g., `sum`, `main`).
  - **Keywords**: Reserved words with special meanings (e.g., `int`, `return`).
  - **Literals**: Constant values assigned to variables (e.g., `5`, `"Hello"`).
  - **Operators**: Symbols representing operations (e.g., `+`, `-`, `*`).
  - **Punctuation**: Symbols used for structuring the code (e.g., `;`, `{}`, `()`).

3. Variables

- **Definition**: Named storage locations in memory that can hold data.
- **Syntax**:

```
int a = 5;
```

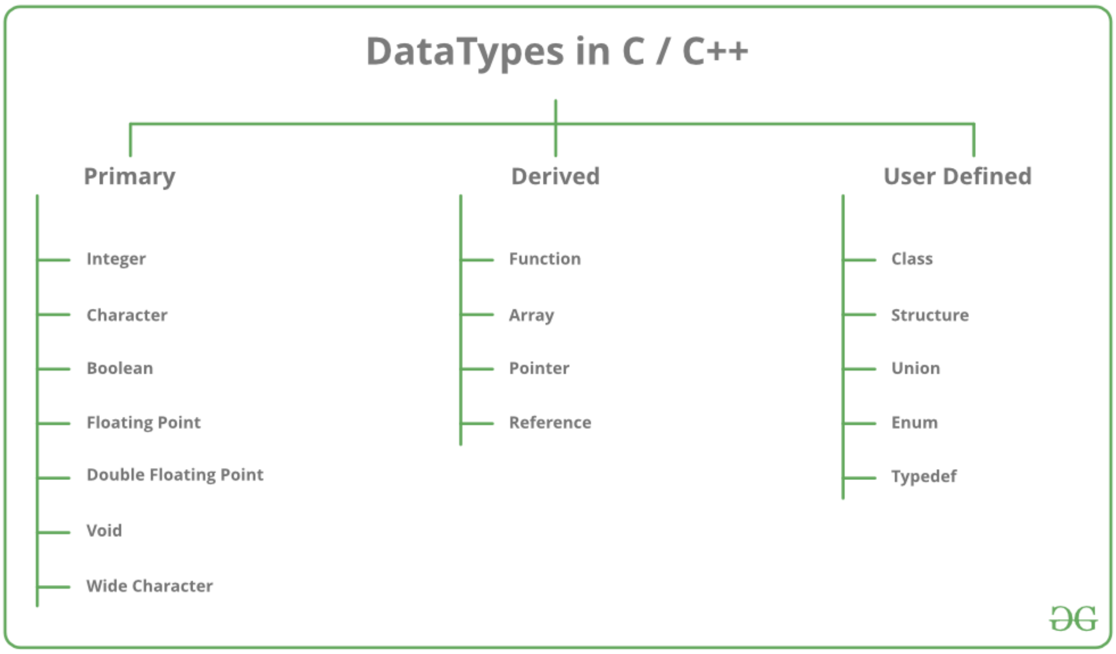
4. Constants

- **Definition**: Variables whose values cannot be changed during program execution.
- **Syntax**:

```
const int PI = 3;
```

5. Data Types:

- **Definition**: Specify the type of data a variable can hold.
- C++ supports the following data types:
  1. Primary or Built-in or Fundamental data type
  2. Derived data types
  3. User-defined data types
- **Basic Data Types/ Primary or Built-in or Fundamental data type**:
  - **int**: Integer type (e.g., `int x = 10;`)
  - **float**: Floating-point type (e.g., `float y = 3.14;`)
  - **char**: Character type (e.g., `char c = 'A';`)
  - **double**: Double-precision floating-point type (e.g., `double z = 3.14159;`)
  - **bool**: Boolean type (e.g., `bool isTrue = true;`)



C++ provides several basic (or fundamental) data types that allow you to define variables with different storage capacities and capabilities. These types include integers, floating-point numbers, characters, and boolean values.

Basic Data Types in C++

1. **Integer Types**:
   - Used to store whole numbers (positive, negative, or zero).

Data Type	Size (Bytes)	Range
`int`	4	-2,147,483,648 to 2,147,483,647
`short int`	2	-32,768 to 32,767
`long int`	4 or 8	At least -2,147,483,648 to 2,147,483,647
`long long int`	8	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
`unsigned int`	4	0 to 4,294,967,295
`unsigned short int`	2	0 to 65,535
`unsigned long int`	4 or 8	0 to 4,294,967,295 (or larger)
`unsigned long long int`	8	0 to 18,446,744,073,709,551,615

   - **Modifiers**: `signed`, `unsigned`, `short`, and `long` can modify integer types to adjust their size and range.
     - `signed` allows both positive and negative values.
     - `unsigned` allows only non-negative values, extending the range upwards.

2. **Floating-Point Types**:
   - Used to store real numbers (numbers with fractional parts).

Data Type	Size (Bytes)	Precision (Decimal Places)	Range
`float`	4	6-7	±3.4e-38 to ±3.4e+38
`double`	8	15	±1.7e-308 to ±1.7e+308
`long double`	8 or 16	More than `double`	Depends on system

3. **Character Type**:
   - Used to store individual characters.

Data Type	Size (Bytes)	Range
`char`	1	-128 to 127

`unsigned char`	1	0 to 255
`wchar_t`	2 or 4	Larger character set (wide characters), typically used for Unicode.

4. **Boolean Type**:
   - Used to store true or false values.

Data Type	Size (Bytes)	Values
`bool`	1	`true` or `false`

5. **Void Type**:
   - Does not store any data. Used mainly in functions to specify that they return nothing.

Data Type	Size	Purpose
`void`	N/A	Used in functions with no return.

Example Code Using Basic Data Types:

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

int main() {
    int a = 10;                 // Integer
    float b = 3.14f;            // Floating-point
    double c = 3.1415926535;    // Double precision floating-point
    char d = 'A';               // Character
    bool e = true;              // Boolean
    unsigned int f = 300;       // Unsigned integer

    cout << "Integer a: " << a << endl;
    cout << "Float b: " << b << endl;
    cout << "Double c: " << c << endl;
    cout << "Character d: " << d << endl;
    cout << "Boolean e: " << e << endl;
    cout << "Unsigned Integer f: " << f << endl;

    return 0;
}
```

Output:

```
Integer a: 10
Float b: 3.14
Double c: 3.14159
Character d: A
```

```
Boolean e: 1
Unsigned Integer f: 300
```

Summary of Basic Data Types:

Type	Description
`int`	Stores integers.
`float`	Stores single-precision floating-point numbers.
`double`	Stores double-precision floating-point numbers.
`char`	Stores single characters.
`bool`	Stores boolean values (`true` or `false`).
`void`	Used for functions that return no value.

Types of Errors

Programming Errors • Errors are the problems or the faults that occur in the program, which makes the behaviour of the program abnormal. • Programming errors are also known as the bugs or faults. • The process of removing these bugs is known as debugging. • These errors are detected either during the time of compilation or execution.

There are mainly five types of errors in programming:

1. Syntax Error
2. Semantic Error
3. Logical Error
4. Run-Time Error
5. Linker Error

1. Syntax Errors: Syntax errors are also known as the compilation errors as they occurred at the compilation time.

- **Definition**: Errors due to incorrect syntax, such as missing semicolons, mismatched parentheses, or misspelled keywords. These errors are mainly occurred due to the mistakes while typing or do not follow the syntax of the specified. programming language.

- **Example**:

```
1.  int a = 5  // Missing semicolon causes a syntax error
```

```
2. cout<<b; // "int b" is missing, variable without declaration
```

```
3. int main()
    {
        cout<<"Hello";  // missing close curly bracket }
```

```
4. a=10;
   cout<<a;   // a is undeclared
```

Commonly occurred syntax errors are:

```
          - If we miss the parenthesis (}) while writing the code.
          -  Displaying the value of a variable without its declaration.
          - If we miss the semicolon (;) at the end of the statement.
```

2. Logical Error: The logical error is an error that leads to an undesired output. These errors produce the incorrect output, but they are errorfree, known as logical errors.

```
          - The occurrence of these errors mainly depends upon the logical thinking of
the developer.
          - If the programmers sound logically good, then there will be fewer chances
of these errors.
```

- **Definition**: Errors where the program runs but produces incorrect results due to flawed logic.
- **Example**:

  ```
  for(int i=0; i<=10; i++); // logical error, as we put semicolon after loop
  {
      cout<< i;
  }
  ```

3. Runtime Error: Errors which occur during program execution(run-time) after successful compilation are called run-time errors. These errors are very difficult to find, as the compiler does not point to these errors.

- **Definition**: Errors that occur during the execution of the program, such as dividing by zero or accessing invalid memory.

- **Example**:

  ```
  int a= 10;
  int b= a / 0;
  cout<< b;
  ```

- *4. Linker Error:** These error occurs when after compilation we link the different object files with main’s object using Ctrl+F9 key(RUN).

  - Linker errors are mainly generated when the executable file of the program is not created. This can be happened either due to the wrong function prototyping or usage of the wrong header file.
  - The most common linker error that occurs is that we use Main() instead of main().
    - **Example**:

      ```
      #include<iostream>
      using namespace std;
      int Main()  // undefined reference to main
      {
         cout<<"Hello";
      ```

```
    return 0;
}
```

- *5. Semantic errors:** These errors that occurred when the statements are not understandable by the compiler.

1. **Use of a un-initialized variable.**

```
int i;
i=i+2;
```

2. **Type compatibility**

```
    int b = "javatpoint";
```

3. **Errors in expressions**

```
    int a, b, c;
    a + b = c;
```

4. **Array index out of bound**

```
  int a[10];
  a[10] = 34;
```

- **Example**:

```
int a= 2, b=3, c=1;
a+b=c; // semantic error
```