## GCC Compiler:

The GCC (GNU Compiler Collection) is a set of compilers provided by the GNU Project. It is one of the most widely used compilers in the Linux environment and supports various programming languages, including C, C++, Fortran, Ada, and others. GCC is the default compiler on many Linux distributions and is a key tool for software development on the platform.

Gcc package is generally pre-installed in Ubuntu but if its not there, then

apt-get update

apt-get install build-essential

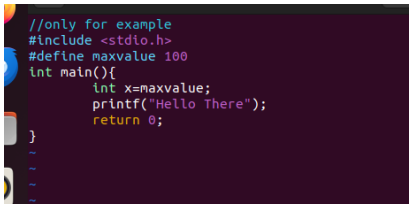
The build-essential package includes essential tools, including GCC, required for compiling software on Ubuntu.

## Compilation Process:

- Preprocessing
    - **Inclusion of header file :**During preprocessing of program if we have included header file e.g. stdio.h .Then it will include all the code of stdio.h
    - **Removal of commands:** If in program there is any command the it removes those commands
    - **Replacing Macro name by its value:** Third task of preprocessing is to replace the macro name by its value if used in the program

**Example:**

```
//this is just example
#include<stdio.h>
#define maxvalue 10
Int main()
    {
    Int x=maxvalue;
Printf(“hello There!”);
Return 0;
}
```



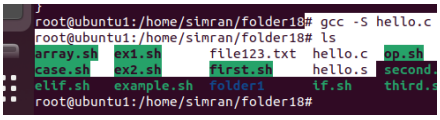

If you just want to run preprocessing state

gcc -E hello.c  (-E flag tells compiler to execute only preprocessing step)


Now you can see there is not comment, Macro got replaces by its definition

- Compiling
  - Compiler converts program into assembly language code
  - The following command tell to execute only preprocessing step and compilation step

gcc -S hello.c

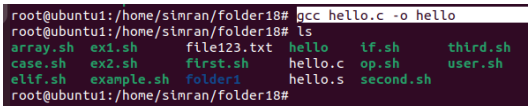



After running this command there is one more file hello.s

hello.s file will contains the assembly code which our machine can't understand.

Now we need assembler to covert this assembly code into binary code so that our machine can understand.

- Assembling
  - This is the third step and here it converts the assembly code into binary
    gcc hello.c -o hello
    With this command we are saving the binary code into this hello file

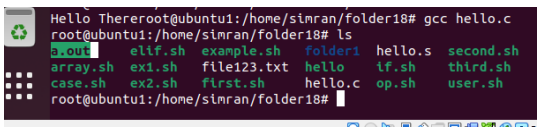



- Linking
  - In program if we have used some functions from another library then during linking the address of those function will get included in final executable file
  - In this step if there are 40 files containing the binary code. With linking we can generate one executable file for all.

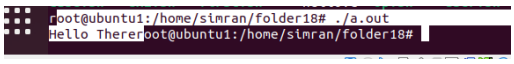
To execute all four steps then use command

gcc hello.c

bydefault it generates a.out file which is our executable file




To run this use ./a.out



If you want to name executable file differently then:

gcc hello.c -o hello

## Example2:

```c
//airthmetic operation
#include <stdio.h>
int main(){
        int a =5, b =3;
        int sum = a + b;
        int diff = a - b;
        int pro = a * b;
        float quotient = (float)a / b;

        printf("Sum: %d\n", sum);
        printf("Diff: %d\n", diff);
        printf("Prduct: %d\n", pro);
        printf("Quotient: %.2f\n", quotient);

        return 0;
}
~
~
```

```
root@ubuntu1:/home/simran/folder18# gcc airth.c -o airth
root@ubuntu1:/home/simran/folder18# ls
airth      case.sh   ex2.sh       first.sh   if.sh       third.sh
airth.c    elif.sh   example.sh   folder1    op.sh       user.sh
array.sh   ex1.sh    file123.txt  hello.c    second.sh
root@ubuntu1:/home/simran/folder18# ./airth
Sum: 8
Diff: 2
Prduct: 15
Quotient: 1.67
```

**The file extensions** commonly associated with the source code files for various programming languages when using the GNU Compiler Collection (GCC) are as follows:

1. **C:**

Source Code File Extension: .c

Example: example.c

2. **C++:**

Source Code File Extension: .cpp, .cxx, .cc

Example: example.cpp

3. **Objective-C:**

Source Code File Extension: .m
Example: example.m

4. **Objective-C++:**

Source Code File Extension: .mm
Example: example.mm

5. **Java (using GCJ):**

Source Code File Extension: .java

Example: Example.java

## Here are the typical compilation commands for different languages:

**1. C:**

```bash
gcc -o output_program input_file.c
```

**2. C++:**

```bash
g++ -o output_program input_file.cpp
```

**3. Fortran:**

```bash
gfortran -o output_program input_file.f90
```

**4. Objective-C:**

```bash
gcc -o output_program input_file.m -lobjc
```

**5.Objective-C++:**

```bash
g++ -o output_program input_file.mm -lobjc
```

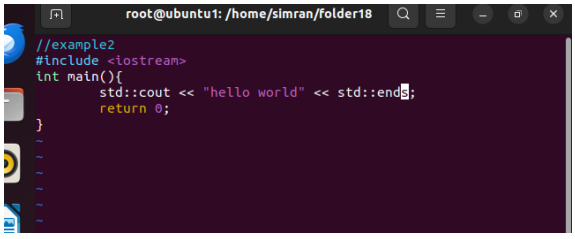
**6. Java (using GCJ):**

```bash
gcj -o output_program input_file.java
```

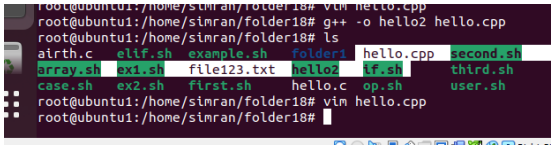
**7. Go:**

```bash
go build -o output_program input_file.go
```

## Example2:



Hello2 executable file




# Ansible:

Ansible is an open-source automation tool that is used for configuration management, application deployment, task automation, and infrastructure orchestration. It simplifies the process of automating IT tasks, allowing system administrators to manage and configure systems more efficiently.

**Example:**

To setup Apache on 5 servers of the company

Using Ansible we can make a single configuration file to automatically create Apache setup on all the 5 servers of the company instead of doing SSH

Here are some key details about Ansible on Ubuntu:

**Installation:**

To install Ansible on Ubuntu, you can use the following commands:

```bash
sudo apt update
sudo apt install ansible
```

This will install Ansible and its dependencies on your Ubuntu system.

**Configuration:**

Ansible uses a configuration file, typically located at `/etc/ansible/ansible.cfg`. You can customize this file to modify Ansible's behavior. Additionally, inventory files define the hosts on which Ansible will operate.

## Inventory:

The inventory file (usually located at `/etc/ansible/hosts`) lists the hosts or servers on which Ansible will execute tasks. You can define groups of hosts and assign variables to them.

## Playbooks:

The configuration  file which we made to setup Apache on 5 servers that file is playbook written in YAML.

Ansible uses **YAML-based files** called playbooks to define automation tasks. Playbooks can include multiple plays, and each play consists of tasks that define the desired state of the system.

## Modules:

Ansible modules are reusable units of code that perform specific tasks. There are modules for various purposes, such as managing packages, files, users, and services.

## Ad-Hoc Commands:

You can use Ansible for ad-hoc commands to perform quick tasks on remote hosts. For example:

```bash
**ansible all -m ping**
```

This command uses the `ping` module to check if hosts are reachable.

# Working:

- Module: small program  for a task
  Example:
  - name: Copy a file to remote host
    copy:
      src: /path/to/local/file.txt
      dest: /path/to/remote/

- Inventory: A file contains servers information. (inventory.yaml)
- Playbook: it contains multiple modules

  ---

  - name: Manage Configuration File

    hosts: app_servers

    become: true

    tasks:

      - name: Copy config file

        copy:

          src: files/app_config.conf

          dest: /etc/app_config.conf

  To install:

  apt-get install ansible

Ansible playbooks are YAML files that define a set of tasks to be executed on remote hosts. Below are a few examples of Ansible playbooks for common use cases:

### Example 1: Basic Playbook

This simple playbook installs the Apache web server on remote hosts:

```yaml
---
- name: Install Apache
  hosts: web_servers
  become: true  # Run tasks with sudo

  tasks:
    - name: Update package cache
      apt:
        update_cache: yes

    - name: Install Apache
      apt:
        name: apache2
        state: present
```

## Example 2: File Management

This playbook ensures a specific file exists on remote hosts:

```yaml
---
- name: Manage Configuration File
  hosts: app_servers
  become: true

  tasks:
```

```
  - name: Copy config file
    copy:
      src: files/app_config.conf
      dest: /etc/app_config.conf
    notify:
      - Restart App Service

handlers:
  - name: Restart App Service
    service:
      name: app_service
      state: restarted
```

## Example 3: User and Group Management

This playbook creates a user and a corresponding group:

```yaml
---
- name: Manage Users and Groups
  hosts: all
  become: true

  tasks:
    - name: Create group
      group:
        name: developers
        state: present

    - name: Create user
      user:
        name: john
```

```
    group: developers
    password: "{{ 'my_password' | password_hash('sha512') }}"
    state: present
```

## Example 4: Conditional Tasks

This playbook includes a task based on a condition:

```yaml
---
- name: Conditional Task
  hosts: all
  become: true

  tasks:
    - name: Install Apache on Ubuntu
      apt:
        name: apache2
        state: present
      when: "'ubuntu' in ansible_distribution.lower()"

    - name: Install Apache on CentOS
      yum:
        name: httpd
        state: present
      when: "'centos' in ansible_distribution.lower()"
```

## Example 5: Using Variables

This playbook uses variables to customize tasks:

```yaml
---
- name: Use Variables
  hosts: app_servers
  become: true

  vars:
    app_port: 8080

  tasks:
    - name: Ensure app is running
      service:
        name: my_app
        state: started
      vars:
        app_port: "{{ app_port }}"
```