Unit 4

File Systems, Command-Line Tools, and Build Automation

Learning Objectives

- To perform text and binary file operations in C
- To utilize command-line arguments for flexible program input
- To understand and apply preprocessor directives and macros
- To modularize code using custom header files
- To build, compile, and manage C projects using automation tools like Makefiles
- To use version control systems (Git) and debugging tools (gdb)
- To explore foundational CI/CD practices in modern C development

Structure

4.14 Case Study
4.15 References

4.1 File Handling in C

File handling allows a program to **store data permanently** on disk and access it later. It is essential for creating, reading, updating, and deleting files such as logs, databases, or user records.
C provides a standard library <stdio.h> that includes all file handling functions like fopen(), fclose(), fprintf(), fscanf(), etc.

4.1.1 Introduction to File Operations

In C, files are handled using **file pointers** of type FILE *. Files must be opened before accessing and closed after finishing operations.
Basic File Operations:

- Create and open a file
- Read from the file
- Write to the file
- Close the file

Declaration:

```
FILE *fp;
```

4.1.2 Opening and Closing Files (fopen(), fclose())

fopen()
Used to open a file in a specific mode (read, write, append, etc.)

```
fp = fopen("data.txt", "r");    // Open file in read mode
```

Common Modes:

Mode	Meaning
"r"	Read existing file
"w"	Write (create or overwrite)
"a"	Append
"r+"	Read and write
"w+"	Write and read
"a+"	Append and read

fclose()
Closes the opened file to release memory and save data.

```
fclose(fp);
```

Always close files to avoid **data corruption or memory leaks**.

4.1.3 Reading from Files (fscanf(), fgets(), fread())

fscanf()
Reads formatted data from a file, similar to scanf().

```
FILE *fp = fopen("data.txt", "r");
int age;
```

fscanf(fp, "%d", &age);

fgets()

Reads a line (including spaces) from a file.

char line[100];

fgets(line, 100, fp);

fread()

Used for reading **binary data** from a file.

fread(buffer, size, count, fp);

- Reads count blocks of size bytes each from the file.

4.1.4 Writing to Files (fprintf(), fputs(), fwrite())

fprintf()

Writes formatted data to a file (like printf()).

fprintf(fp, "Name: %s, Age: %d\n", name, age);

fputs()

Writes a string to a file.

fputs("Hello, File!", fp);

fwrite()

Used to write binary data.

fwrite(buffer, size, count, fp);

Example:

int arr[5] = {1, 2, 3, 4, 5};

fwrite(arr, sizeof(int), 5, fp);

4.1.5 File Modes, Error Handling and feof(), ferror()

File Modes Recap:

- "r": Open for reading
- "w": Create or overwrite
- "a": Append to end of file

Error Handling:

Check if file opened successfully:

```
FILE *fp = fopen("data.txt", "r");
if (fp == NULL) {
    printf("File could not be opened.\n");
}
```

feof()

Returns true if **end of file** is reached.

```
while (!feof(fp)) {
    // read data
}
```

ferror()

Checks if an error occurred during file operations.

```
if (ferror(fp)) {
    printf("Error reading the file.\n");
```

}

4.2 Text vs Binary File Handling

C supports two types of files:

- **Text Files**: Human-readable, stored as characters (ASCII)
- **Binary Files**: Machine-readable, stored in raw bytes

Understanding both is essential for efficient data storage and retrieval in real-world applications.

4.2.1 Differences Between Text and Binary Files

Feature	Text File	Binary File
Format	Human-readable characters	Raw byte format
Size	Larger (extra characters like '\n')	Smaller (compact)
Read/Write	Using fprintf(), fscanf()	Using fwrite(), fread()
Use Case	Logs, config files, simple data	Images, databases, performance apps
Portability	More portable	May vary across platforms
Performance	Slower (requires parsing)	Faster (direct memory access)

4.2.2 Binary File Operations using fread() and fwrite()

Binary operations read and write raw memory blocks. Useful for structs, arrays, and large datasets.

Writing to a Binary File:

```
FILE *fp = fopen("data.bin", "wb");
int arr[] = {1, 2, 3, 4, 5};
fwrite(arr, sizeof(int), 5, fp);
fclose(fp);
```

Reading from a Binary File:

```
FILE *fp = fopen("data.bin", "rb");
int arr[5];
fread(arr, sizeof(int), 5, fp);
fclose(fp);
```

- "wb" = write binary
- "rb" = read binary
- Binary files do not store characters like \n or \0 unless explicitly written

4.2.3 Use Cases and Efficiency of Binary Files

When to use Binary Files:

- Handling large datasets (arrays, images, multimedia)
- Reading and writing entire structures or blocks
- Applications where **speed and space efficiency** are critical

Example: Storing Structure Data

```
struct Student {
    int id;
    float marks;
};

struct Student s1 = {101, 89.5};
FILE *fp = fopen("student.bin", "wb");
fwrite(&s1, sizeof(struct Student), 1, fp);
fclose(fp);
```

Why more efficient?

- No need to convert to/from characters
- Direct memory-level operations

4.2.4 File Structures for Data Storage

In binary files, **structures** can be directly stored and retrieved without converting to strings.

Example: Read Structure from Binary File

```
struct Student s2;
FILE *fp = fopen("student.bin", "rb");
fread(&s2, sizeof(struct Student), 1, fp);
fclose(fp);

printf("ID: %d, Marks: %.2f", s2.id, s2.marks);
```

Advantages:

- Maintains structure layout
- Faster read/write
- Avoids manual parsing

Note: Use caution when transferring binary files between different architectures (endianness, padding may differ)

4.2.5 Handling Large Data Files

When working with **large datasets**, binary files are preferred because:

- They are **smaller in size**
- **I/O operations are faster**
- Random access is possible using fseek() and ftell()

Example: Moving the File Pointer

```
fseek(fp, sizeof(struct Student) * 10, SEEK_SET);    // Move to 11th record
```

Tips:

- Use fread() and fwrite() in blocks
- Keep memory usage low by loading only what you need

- Validate file operations with feof() and ferror()

4.3 Command-Line Arguments

Command-line arguments allow users to **pass inputs to a program when it starts** — directly from the terminal. They are commonly used in **Unix/Linux environments** to create flexible and interactive programs.

Example command:

./program file.txt 5

Here, file.txt and 5 are passed to the program as arguments.

4.3.1 main(int argc, char *argv[]) Format

To use command-line arguments, modify the main function as:

int main(int argc, char *argv[])

- **argc (Argument Count)**: Total number of arguments including the program name
- **argv[] (Argument Vector)**: Array of strings representing arguments

Example:

```
#include <stdio.h>

int main(int argc, char *argv[]) {
	printf("Total arguments: %d\n", argc);
	printf("Program name: %s\n", argv[0]);
	for (int i = 1; i < argc; i++) {
		printf("Arg %d: %s\n", i, argv[i]);
	}
	return 0;
}
```

If you run: ./demo hello 123

Output:

Total arguments: 3

Program name: ./demo

Arg 1: hello

Arg 2: 123

4.3.2 Accessing and Parsing Command-Line Inputs

All arguments are received as **strings** (character arrays). If a numeric value is needed, use atoi(), atof(), etc.

Example:

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
	int num = atoi(argv[1]);
	printf("Square: %d\n", num * num);
	return 0;
```

}
If run as: ./square 5 → Output: Square: 25
Note: Always check if argc is sufficient before accessing argv[].

4.3.3 Practical Use Cases: File Names, Flags, Arguments
Command-line arguments are commonly used for:

- **Passing file names**

./myprog input.txt output.txt

- **Setting configurations**

./compress image.jpg -q 80

- **Flags and options**

./search -v -case file.txt
This allows building tools like gcc, ls, and cp that accept dynamic input.

4.3.4 Error Checking for Argument Inputs
Always validate arguments before use to prevent crashes or incorrect behavior.
Example:

```
int main(int argc, char *argv[]) {
    if (argc != 2) {
        printf("Usage: %s <number>\n", argv[0]);
        return 1;
    }

    int num = atoi(argv[1]);
    printf("Double: %d\n", num * 2);
    return 0;
}
```

- Prevents program from crashing when no argument is passed
- Improves **user experience and debugging**

4.3.5 Creating Flexible Command-Line Programs
By using command-line inputs effectively, you can:

- Build **menu-driven utilities**
- Accept **file paths or parameters** from users
- Develop **testable programs** for automation
- Enable **script integration** in Unix/Linux systems

Example: Basic Copy Program

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    if (argc != 3) {
```

```
        printf("Usage: %s <source> <destination>\n", argv[0]);
        return 1;
    }

    FILE *src = fopen(argv[1], "r");
    FILE *dest = fopen(argv[2], "w");

    if (src == NULL || dest == NULL) {
        printf("File error.\n");
        return 1;
    }

    char ch;
    while ((ch = fgetc(src)) != EOF)
        fputc(ch, dest);

    fclose(src);
    fclose(dest);
    printf("File copied successfully.\n");
    return 0;
}
```

4.4 Preprocessor Directives

Preprocessor directives are instructions that are **executed before the actual compilation** of code begins. These directives start with the # symbol and help manage **macros, constants, file inclusion, and conditional compilation**.

They are not C statements — they are handled by the **preprocessor**, not the compiler.

4.4.1 Role of the Preprocessor in Compilation

The **C preprocessor** performs tasks like:

- File inclusion
- Macro expansion
- Conditional compilation
- Constant substitution

These are done **before** the actual compilation phase.

Example of preprocessing:

```
#include <stdio.h>      // File inclusion
#define PI 3.14         // Constant substitution
```

The preprocessor replaces PI with 3.14 everywhere before compiling.

4.4.2 #define, #include, and Constant Declarations

#define

Used to define symbolic constants or macros.

```
#define MAX 100
```

#define AREA(r) (3.14 * r * r)

- No semicolon at the end
- No memory is allocated — it's text replacement

#include

Used to include **header files** or external libraries.

```
#include <stdio.h>      // Standard header
#include "myfile.h"     // User-defined header
```

- Angle brackets for system headers
- Double quotes for custom files

Constant Declaration

Using const keyword:

```
const int x = 10;
```

Unlike #define, const is type-safe and allows debugging.

4.4.3 Conditional Compilation (#ifdef, #ifndef, #endif)

Conditional compilation helps to **include or exclude code** based on defined conditions.

Syntax:

```
#ifdef DEBUG
    printf("Debug mode on\n");
#endif
```

Common directives:

Directive	Meaning
#ifdef	If macro is defined
#ifndef	If macro is not defined
#if	If condition is true
#else	If condition is false
#endif	End of condition block

Example:

```
#define DEBUG

#ifdef DEBUG
    printf("This is debug info.\n");
#else
    printf("Release build.\n");
#endif
```

4.4.4 Creating and Using Macros

Macros are defined using #define and can accept arguments like functions.

Example:

```
#define SQUARE(x) ((x) * (x))

int result = SQUARE(5);    // Expands to (5 * 5)
```

- Use parentheses to prevent unexpected behavior due to operator precedence.

Bad Example:
```
#define SQUARE(x) x * x
int a = SQUARE(1 + 2);    // Becomes 1 + 2 * 1 + 2 = 5 (Wrong!)
```

4.4.5 Best Practices and Macro Pitfalls

Best Practices:

- Prefer const, inline, or enum over #define when possible
- Use uppercase for macro names (#define MAX_SIZE 100)
- Always use parentheses in macro expressions
- Use macros for **constants or simple code snippets** only

Common Pitfalls:

- **No type checking** for macros
- Can cause **unexpected results** due to lack of scope or brackets
- Can make debugging harder
- Avoid using macros for complex logic — use functions instead

4.5 Header Files and Code Modularity

Header files in C are used to **organize code into reusable modules**, promoting better code structure, reusability, and maintainability — especially in **large projects**.
They usually contain:

- Function declarations
- Macros
- Constants
- Structure definitions

4.5.1 Purpose and Benefits of Header Files

Purpose:

- Separate declarations from definitions
- Avoid code duplication
- Promote cleaner main programs

Benefits:

- Code **modularity**: organize code logically

- **Reusability**: share functions/definitions across files
- **Ease of maintenance**: change in one header reflects in all dependent files
- Supports **team development** by dividing work across modules

Example:
Suppose you have a math utility header:
```
// mathutils.h
int square(int x);
#define PI 3.14
```
Used in:
```
#include "mathutils.h"
```

4.5.2 Declaring Functions and Macros in Headers

Header files typically **declare** functions and **define** macros/constants, but do **not include function definitions**.

Example:
```
// calc.h
#ifndef CALC_H
#define CALC_H

int add(int a, int b);
int multiply(int a, int b);
#define MAX(x, y) ((x) > (y) ? (x) : (y))

#endif
```
Then implement in calc.c:
```
int add(int a, int b) {
	return a + b;
}
```

4.5.3 #include Usage with Custom Headers

To use your own header files:
```
#include "filename.h"      // for user-defined headers
#include <stdio.h>         // for standard headers
```
- Quotes "" tell the compiler to **look in the current directory**
- Angle brackets <> search in **system directories**

4.5.4 Header Guards and Avoiding Redundancy

To **prevent multiple inclusions** of the same header, use **header guards**.

Syntax:
```
#ifndef HEADER_NAME_H
#define HEADER_NAME_H

// Declarations
```

#endif
Example:
```
// config.h
#ifndef CONFIG_H
#define CONFIG_H

#define VERSION 1.0

#endif
```
Without header guards, the compiler may encounter **redefinition errors**.

4.5.5 Managing Large Projects with Modular Headers
In large projects:

- Group related functions into **modules**
- Create one header file for each module
- Include only required headers in each source file
- Use a **master header** to include commonly used headers

Project Structure Example:
```
project/
├── main.c
├── input.c
├── output.c
├── input.h
├── output.h
└── utils/
    ├── fileops.c
    └── fileops.h
```
Benefits:

- Easier debugging and testing
- Clean separation of logic
- Better team collaboration

4.6 Build Automation using Makefile
In large C projects with multiple source files, manually compiling each file can be tedious and error-prone. A **Makefile** automates this process, ensuring only updated files are compiled, saving time and reducing mistakes.

4.6.1 Introduction to Build Process
The **build process** in C includes:

1. **Preprocessing** (.c files to expanded .c)

2. **Compilation** (source to object .o)
3. **Linking** (object files to final executable)

In multi-file projects:

- Each .c file is compiled separately
- All object files are linked to form the executable

Makefiles automate this process using simple rules.

4.6.2 Structure of a Makefile: Targets, Dependencies, Commands

A Makefile is a set of **rules** that define:

- **Target**: the file to be created (e.g., main.o, app)
- **Dependencies**: files the target depends on (e.g., .c, .h)
- **Commands**: shell commands (usually gcc)

Syntax:

target: dependencies
<TAB> command
Note: The command line **must begin with a tab**, not spaces.
Example:

```
main.o: main.c utils.h
    gcc -c main.c
```

This rule says: "To build main.o, compile main.c if main.c or utils.h has changed."

4.6.3 Automating Compilation of Multi-File Projects

Let's assume a project with:

- main.c
- maths.c
- maths.h

Makefile:

```
app: main.o maths.o
    gcc -o app main.o maths.o

main.o: main.c maths.h
    gcc -c main.c

maths.o: maths.c maths.h
    gcc -c maths.c

clean:
    rm *.o app
```

How it works:

- Only recompiles .c files that have changed

- Links the object files to create app
- make clean removes temporary files

To build, just run:
make

4.6.4 Using Variables and Comments in Makefiles
You can use **variables** for readability and reusability.
Example:

```
CC = gcc
CFLAGS = -Wall

app: main.o maths.o
	$(CC) -o app main.o maths.o

main.o: main.c
	$(CC) $(CFLAGS) -c main.c
```

- $(CC) = Compiler
- $(CFLAGS) = Compiler options

Comments:

```
# This is a comment
```

4.6.5 Practical Example: Makefile for a C Project
Project Files:

- main.c
- calc.c
- calc.h

Makefile:

```
CC = gcc
CFLAGS = -Wall -g
OBJ = main.o calc.o
TARGET = calculator

$(TARGET): $(OBJ)
	$(CC) $(CFLAGS) -o $(TARGET) $(OBJ)

main.o: main.c calc.h
	$(CC) $(CFLAGS) -c main.c

calc.o: calc.c calc.h
	$(CC) $(CFLAGS) -c calc.c

clean:
```

```
	rm -f *.o $(TARGET)
```
To compile:
make
To remove build files:
make clean

4.7 Compiler Flags and Optimization

Compiler flags are command-line options provided to the **GCC (GNU Compiler Collection)** or other compilers to control how source code is compiled. These flags are used for:

- Enabling warnings
- Debugging
- Optimizing code
- Setting language standards
- Cross-compilation

4.7.1 Common GCC Flags: -Wall, -O, -g, -std

-Wall

- Enables **most important warnings**
- Helps catch common bugs like unused variables, uninitialized values

gcc -Wall program.c

-O (optimization default)

- Enables basic optimizations (same as -O1)

gcc -O program.c

-g

- Includes **debug information** for tools like gdb
- Does not affect performance

gcc -g program.c

-std

- Specifies **C language standard** to use

Examples:
gcc -std=c89 program.c
gcc -std=c99 program.c
gcc -std=c11 program.c

4.7.2 Optimization Flags and Use Cases (-O1, -O2, -O3)

GCC provides different levels of optimization:

Flag Optimization Level **Use Case**

Flag	Optimization Level	Use Case
-O0	No optimization (default)	Debugging and development
-O1	Basic optimizations	Balance between speed and compilation
-O2	Further optimizations	Recommended for most production builds
-O3	Aggressive optimizations	High-performance applications (e.g., math)
-Os	Optimize for size	Embedded or size-constrained systems

Example:

```
gcc -O2 program.c -o program
```

4.7.3 Debugging Flags and Warning Management

Debugging Flags:

- -g: Add debug symbols
- -ggdb: Add symbols specifically for gdb

Warning Flags:

- -Wall: All common warnings
- -Wextra: Additional warnings
- -Werror: Treat all warnings as **errors**

Example:

```
gcc -Wall -Wextra -Werror -g program.c
```

This helps enforce good coding practices and aids in debugging.

4.7.4 Cross-Platform Compilation Considerations

When compiling code for different platforms (e.g., compiling Linux code on Windows), use:

- **Cross-compilers**: like arm-linux-gcc, x86_64-w64-mingw32-gcc
- Flags like -m32 or -m64 for 32-bit/64-bit builds

Example:

```
gcc -m32 -o program program.c        # Compile for 32-bit architecture
```

Considerations:

- Check for endian differences
- Avoid platform-specific headers
- Use conditional compilation with #ifdef

4.7.5 Combining Flags in Build Scripts

Compiler flags are often combined in **Makefiles** or build scripts for automation.

Example Makefile:

```
CC = gcc
CFLAGS = -Wall -Wextra -O2 -std=c11 -g
```

```
program: main.o util.o
    $(CC) $(CFLAGS) -o program main.o util.o
```
Flags can also be customized per file:
```
main.o: main.c
    $(CC) -O0 -g -c main.c
```

4.8 Debugging with gdb

gdb (GNU Debugger) is a command-line tool used to **find and fix bugs in C programs**. It allows you to:

- Run your program **line-by-line**
- Set breakpoints
- Inspect variables and memory
- Analyze segmentation faults and crashes

4.8.1 Introduction to gdb and Debugging Concepts

gdb lets developers:

- **Pause execution** at specific lines (breakpoints)
- **Step through code** line-by-line or function-by-function
- **Inspect values** of variables at runtime
- Understand what causes **logic errors or crashes**

Common use cases:

- Tracking down **segmentation faults**
- Watching variables to catch incorrect assignments
- Debugging **loops, conditions, and function calls**

4.8.2 Compiling with -g for Debug Info

To use gdb, compile the program with the **-g flag**, which includes **debug symbols** in the executable.

Example:

```
gcc -g program.c -o program
```

Without -g, gdb will not show line numbers or variable names.

4.8.3 Running gdb and Setting Breakpoints

To start debugging:

```
gdb ./program
```

Inside gdb, you can run commands like:

Common gdb Commands:

Command	**Description**

Command	Description
break main	Set breakpoint at main()
break 15	Set breakpoint at line 15
break func_name	Set breakpoint at a function
run	Start the program
list	Show source code
delete	Remove breakpoints
quit	Exit gdb

4.8.4 Stepping Through Code and Inspecting Variables

Once your program hits a breakpoint, you can **step through it**.

Commands:

Command	Description
next or n	Execute next line (step over function calls)
step or s	Step into functions
continue	Continue until next breakpoint
print x	Print value of variable x
info locals	List all local variables
backtrace	Show function call stack

Example:

(gdb) print x
$1 = 5

You can also **modify variable values** during execution:

(gdb) set var x = 10

4.8.5 Using gdb to Debug Segmentation Faults and Logic Errors

When a program crashes (segmentation fault), gdb helps identify the **exact location** and cause.

Steps to debug a crash:

1. Compile with -g

2. Run program using gdb:

gdb ./program

3. Start execution:

run

4. After crash:

backtrace

This shows the **call stack** at the time of the crash.

Example:

Program received signal SIGSEGV, Segmentation fault.
0x000000000040115a in faulty_function (x=0) at program.c:20

You can then inspect variables using print to understand the error.

4.9 Introduction to Version Control (Git)

Version control is essential in software development to **track changes, collaborate safely**, and **manage different versions** of a project over time. **Git** is the most widely used version control system, and **GitHub** is a popular platform for hosting Git repositories online.

4.9.1 Need for Version Control in Development

Without version control:

- Code changes can be lost or overwritten
- Collaboration becomes chaotic
- Tracking who changed what, and when, is difficult

Git solves this by:

- Maintaining a **history of all changes**
- Supporting **collaboration across teams**
- Allowing **rollback** to previous versions
- Providing **branching** for testing features safely

Use cases:

- Undo mistakes
- Track code over time
- Develop features in isolation
- Merge work from multiple developers

4.9.2 Basic Git Commands: init, add, commit, status, log

git init

Initializes a Git repository in the current folder.

```
git init
```

git add

Stages changes to be committed.

```
git add file.c
git add .                # Adds all files
```

git commit

Saves the staged changes to the repository history with a message.

```
git commit -m "Initial commit"
```

git status

Shows the current state of the working directory (tracked/untracked files, staged changes).

```
git status
```

git log
Shows the commit history.
git log

4.9.3 Branching and Merging Basics
Branching allows working on features or fixes **without affecting the main code**.
Creating a branch:
git branch feature-1
Switching branches:
git checkout feature-1
Merging a branch into main:
git checkout main
git merge feature-1
Benefits of branching:

- Isolated development
- Easier testing and experimentation
- Parallel work in teams

4.9.4 Using .gitignore and Remote Repositories
.gitignore
Specifies files/folders Git should **not track** (e.g., compiled files, temporary files).
Example .gitignore:
*.o
*.exe
*.log
build/
This avoids cluttering the repository with generated or irrelevant files.
Remote Repositories
Used to **collaborate online**, usually hosted on **GitHub, GitLab, or Bitbucket**.
Example commands:
git remote add origin https://github.com/user/project.git
git push -u origin main
git pull origin main

4.9.5 GitHub and Collaborative Workflows
GitHub allows:

- Hosting repositories
- Reviewing and commenting on code
- Pull requests for merging code
- Issue tracking

Typical GitHub Workflow:

1. Fork repository

2. Clone to local system
3. Create a branch for your feature
4. Make changes and commit
5. Push to GitHub
6. Create a pull request for review

Key GitHub features:

- Pull Requests (PRs)
- Issues and discussions
- Actions for CI/CD
- Repository Insights (activity, contributors)

4.10 CI/CD Concepts for C Projects

CI/CD stands for **Continuous Integration** and **Continuous Deployment/Delivery**. It is a **modern software practice** that automates the **building, testing, and deployment** of applications.

CI/CD helps ensure that:

- Code is always **working and testable**
- Changes are **integrated and verified** regularly
- Teams can **deliver features faster and with fewer bugs**

4.10.1 Overview of CI/CD in Software Development

- **Continuous Integration (CI)**: Automatically integrates code from multiple developers and runs tests on each change.
- **Continuous Delivery (CD)**: Ensures that the code is always ready to be deployed.
- **Continuous Deployment (CD)**: Automatically deploys every change that passes tests to production.

In C projects, CI/CD can:

- Run make automatically
- Compile multiple source files
- Detect compile-time and logic errors
- Run unit tests for verification

4.10.2 Setting Up a Basic CI Pipeline (e.g., GitHub Actions)

CI tools like **GitHub Actions**, **GitLab CI**, or **Jenkins** allow setting up a pipeline using YAML configuration files.

Example: GitHub Actions for C Build

```
# .github/workflows/c-build.yml
name: C Build

on: [push, pull_request]

jobs:
  build:
    runs-on: ubuntu-latest

    steps:
    - name: Checkout Code
      uses: actions/checkout@v2

    - name: Install GCC
      run: sudo apt-get install gcc

    - name: Compile C Program
      run: gcc main.c -o main

    - name: Run the Program
      run: ./main
```

This pipeline:

- Runs on each **push** or **pull request**
- Checks out the repository
- Installs GCC
- Compiles the code
- Runs the binary

4.10.3 Automating Builds and Tests on Code Push

In CI/CD:

- Every push triggers a **build**
- Code is compiled and **unit tests** are executed
- If tests fail, the CI pipeline **fails** and warns developers

This:

- Avoids broken code being merged
- Keeps the **main branch always working**

- Saves time in manual testing

Example Test Step:
- name: Run Unit Tests
  run: ./test_suite
You can include test runners like **CUnit**, **Check**, or custom scripts.

4.10.4 Introduction to Docker for C Applications

Docker helps create a **portable environment** that includes your app, compiler, libraries, and build tools.
Dockerfile Example for a C Project:
```
FROM gcc:latest
COPY . /app
WORKDIR /app
RUN gcc main.c -o main
CMD ["./main"]
```
Benefits:

- No "works on my machine" issues
- Ensures **consistent builds**
- Can be integrated into CI pipelines

To build and run:
```
docker build -t my-c-app .
docker run my-c-app
```

4.10.5 Benefits of CI/CD in Team Environments

In team-based C development:

- CI/CD ensures **every developer's code is tested**
- Reduces **merge conflicts and integration issues**
- Builds **trust** in automatic deployment
- Encourages **small, frequent updates**
- Supports **collaborative, test-driven development**

Key Benefits:

- **Faster feedback** for code changes
- **Early bug detection**
- **Improved code quality**
- Supports **agile workflows**

4.11 Summary

This module focused on essential tools and concepts that bridge the gap between C programming and professional development environments. Learners explored how to perform file I/O operations for text and binary data, how to utilize command-line arguments, and how to manage code efficiently using preprocessor directives and header files. The importance of build automation through Makefiles was covered, alongside optimizing compilation with GCC flags. Debugging using gdb, understanding version control with Git, and setting up CI/CD pipelines (e.g., GitHub Actions) were introduced as part of modern development practices. These skills are essential for creating, testing, and deploying maintainable, collaborative C software projects.

4.12 Keywords

Term	Definition
fopen()	Function used to open a file in C.
Makefile	A script used to automate the build process in C projects.
gcc	GNU Compiler Collection used to compile C programs.
gdb	GNU Debugger used to analyze and fix runtime errors.
.gitignore	File that tells Git which files/folders to ignore.
-Wall	GCC flag that enables all common compiler warnings.
Command-Line Args	Inputs passed to main() via argc and argv[].
Header File	File containing declarations shared between multiple source files.
GitHub Actions	Tool for automating workflows like testing and deployment.
Docker	Containerization tool used for consistent environments across systems.

4.13 Self-Assessment Questions (Subjective & Case-Based)

Subjective Questions

1. Explain the difference between text and binary file handling in C with examples.
2. Describe the structure and purpose of a Makefile. What are targets and dependencies?
3. Discuss the role of preprocessor directives and macros in modular C programming.
4. How can gdb help in resolving segmentation faults? Explain with an example.
5. What is CI/CD? How can GitHub Actions be used in a C project?

Case-Based Questions

1. **Scenario:** A team is working on a student management system in C using multiple .c and .h files. How can they:

- Organize the code using header files?
- Use a Makefile to compile all files?
- Integrate GitHub Actions to test builds automatically?

2. **Scenario:** You have a program that crashes at runtime. Describe how you would use gcc and gdb to identify and fix the issue.

4.14 Case Study

Title: ***Automated Compilation and Testing Pipeline for a Library Management System in C***

Objective:

To demonstrate how a medium-scale C project (e.g., a Library Management System) can be compiled, debugged, versioned, and tested using professional tools.

Key Elements:

- Use of multiple .c and .h files for modularity
- Build management using a custom Makefile
- Handling user data using text and binary file I/O
- Version tracking using Git and GitHub
- CI setup using GitHub Actions to compile and run basic test cases
- Use of debugging tools like gdb for runtime checks

Outcome:

Students will develop, test, debug, and version-control a complete working C project using tools relevant in real-world software development.

4.15 References

1. **The C Programming Language** by Brian W. Kernighan and Dennis M. Ritchie
2. **GNU Compiler Collection (GCC) Manual** – https://gcc.gnu.org/onlinedocs
3. **GNU Make Manual** – https://www.gnu.org/software/make/manual
4. **Git Documentation** – https://git-scm.com/doc
5. **GitHub Actions Docs** – https://docs.github.com/en/actions
6. **GDB Debugger Guide** – https://www.gnu.org/software/gdb/documentation
7. **Docker for Beginners** – https://docs.docker.com/get-started/