Compiling, Makefiles, and Debugging in C

Shiza Ali

1 Compilation Process in C

C programs are translated into machine code in multiple stages:

1. **Preprocessing** (`gcc -E`): Handles #include, #define, and macros.
2. **Compilation** (`gcc -S`): Converts source code into assembly code.
3. **Assembling** (`gcc -c`): Translates assembly into object files (`.o`).
4. **Linking** (`gcc`): Combines object files and libraries into an executable.

Example:

```
gcc -Wall -o program main.c utils.c
```

—

2 Makefiles

A Makefile automates the compilation process by defining how source files depend on each other and how to build them. This saves time and avoids recompiling everything when only one file changes.

2.1 Why Use Makefiles?

- Manages large projects with many source files.
- Ensures only modified files are recompiled.
- Standardizes the build process with simple commands.

2.2 Basic Syntax

```
target: prerequisites
    recipe (command)
```

- **Target:** What you want to build (executable or object file).

- **Prerequisites:** Files the target depends on.
- **Recipe:** Commands (usually compiler calls) to build the target.
- The recipe must be indented with a **TAB**, not spaces.

—

2.3 Step-by-Step Example

Suppose we have:

- `main.c`
- `utils.c`
- `utils.h`

Naive compilation:

```
gcc -Wall -o program main.c utils.c
```

Using a Makefile:

```
program: main.o utils.o
    gcc -Wall -o program main.o utils.o

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

utils.o: utils.c utils.h
    gcc -Wall -c utils.c

clean:
    rm -f *.o program
```

Explanation:

- If `main.c` changes, only `main.o` is recompiled.
- If `utils.c` changes, only `utils.o` is rebuilt.
- The final linking step runs only if needed.

—

2.4 Using Variables

Makefiles allow variables for flexibility:

```
CC = gcc
CFLAGS = -Wall -g

program: main.o utils.o
    $(CC) $(CFLAGS) -o program main.o utils.o
```

Variables avoid repeating options and make the Makefile easier to maintain.

—

2.5 Automatic Variables

- `$@` = name of the target
- `$<` = first prerequisite
- `$^` = all prerequisites

Example:

```
%.o: %.c
    $(CC) $(CFLAGS) -c $< -o $@
```

This rule tells Make how to build any `.o` file from a `.c` file.

—

2.6 Full Example with Pattern Rules

```
CC = gcc
CFLAGS = -Wall -g
OBJS = main.o utils.o math.o

all: program

program: $(OBJS)
    $(CC) $(CFLAGS) -o $@ $^

%.o: %.c
    $(CC) $(CFLAGS) -c $< -o $@

clean:
    rm -f $(OBJS) program
```

This example:

- Defines objects in `OBJS`.
- Uses pattern rules to compile any `.c` into a `.o`.
- Links all objects into the final `program`.

—

3 Debugging in C

3.1 Using `gdb`

1. Compile with debug info: `gcc -g main.c -o program`
2. Run debugger: `gdb ./program`
3. Useful commands:

- `run` – start program
- `break main` – set breakpoint at `main`
- `next` – step over lines
- `step` – step into functions
- `print var` – inspect variables
- `backtrace` – view call stack

3.2 Debugging with Print Statements

Simple but effective: use `printf()` to trace values.

3.3 Common Errors

- **Segmentation Faults**: invalid memory access.
- **Uninitialized Variables**: always initialize.
- **Memory Leaks**: free allocated memory.
- **Mismatched Format Specifiers**: using wrong printf placeholders.

—

4 Summary

- Compilation = preprocessing → compiling → assembling → linking.
- Makefiles automate builds and reduce recompilation.
- Debugging tools: `gdb`, `printf`, static analyzers.