Comprehensive Guide to GCC Compilation, Libraries, and Makefiles

Linux Shell and Basic Commands

Shell Variables

Shell variables are maintained by the shell and stored as strings. Key environment variables include:

Viewing and Setting Variables:

```
echo $VARIABLE_NAME          # View variable value
set                          # View all shell variables
set | more                   # View page by page
MYVAR="MY_VALUE"            # Set variable (no spaces around =)
export VARIABLE_NAME=value   # Make variable available to subshells
```

Important Environment Variables:

- `PATH` - Directories searched for commands (colon-separated list)
- `C_INCLUDE_PATH` - Header file search directories
- `LIBRARY_PATH` - Library search directories for linking
- `LD_LIBRARY_PATH` - Runtime library search directories
- `HOME` - User's home directory

PATH Management:

```
echo $PATH                           # Display current PATH
PATH=$PATH:$HOME/bin                # Add directory to PATH
export C_INCLUDE_PATH=$C_INCLUDE_PATH:/path/to/headers:.
```

File and Directory Operations

Navigation Commands:

```
pwd                     # Show current directory
cd <dirname>           # Change to directory (absolute or relative)
```

```
cd ~                    # Go to home directory
cd ..                   # Go to parent directory
```

File/Directory Creation:

```
mkdir <dirname>         # Create directory
mkdir -p path/to/dir    # Create directory with parent directories
rmdir <dirname>         # Remove empty directory
touch <filename>        # Create empty file or update timestamp
```

File Operations:

```
cp <file1> <file2>      # Copy file
cp -r <dir1> <dir2>     # Copy directory recursively
cp -f <file1> <file2>   # Force copy (overwrite)
mv <file1> <file2>      # Move/rename file
rm <file>               # Remove file
rm -i <file>            # Interactive removal (prompt)
rm -r <dir>             # Remove directory recursively
rm -f <file>            # Force removal (no prompt)
```

File Listing and Viewing:

```
ls                      # List files
ls -l                   # Long listing (detailed)
ls -a                   # Show hidden files
ls -R                   # Recursive listing
ls -t                   # Sort by modification time
ls -r                   # Reverse sort order
ls -d                   # Don't expand directory contents

cat <file>              # Display file contents
head <file>             # Show first 10 lines
head -n 5 <file>        # Show first 5 lines
head -c 100 <file>      # Show first 100 characters
tail <file>             # Show last 10 lines
tail -n 5 <file>        # Show last 5 lines
more <file>             # Page-by-page display
```

File Permissions:

```
chmod u+x <file>        # Add execute permission for user
chmod g-w <file>        # Remove write permission for group
chmod a+r <file>        # Add read permission for all
chmod 755 <file>        # Set permissions: rwxr-xr-x
chmod 644 <file>        # Set permissions: rw-r--r--
```

Other Useful Commands:

```
wc <file>              # Count lines, words, characters
wc -l <file>           # Count lines only
wc -w <file>           # Count words only
wc -c <file>           # Count characters only
diff <file1> <file2>   # Compare files
diff -y <file1> <file2> # Side-by-side comparison
```

Wildcards and Redirection:

```
ls *.c                 # List all .c files
ls ?.txt               # List files with single character + .txt
./program < input.txt  # Input redirection
ls -l > output.txt     # Output redirection
ls -l >> output.txt    # Append output
ls -l | wc -l          # Pipe output to another command
```

GCC Compilation Process

Four Stages of Compilation

1. Preprocessing (-E flag):

```
gcc -E hello.c > hello.i    # Generate preprocessed file
cpp hello.c > hello.i       # Alternative using cpp directly
```

- Expands `#include` directives
- Processes `#define` macros
- Removes comments
- Handles conditional compilation (`#ifdef`, `#ifndef`)

2. Compilation (-S flag):

```
gcc -S hello.c              # Generate assembly file (hello.s)
```

- Converts C code to assembly language
- Performs syntax and semantic checking
- Target architecture specific

3. Assembly (-c flag):

```
gcc -c hello.c              # Generate object file (hello.o)
```

- Converts assembly to machine code
- Creates relocatable object file

- Symbol references remain unresolved

4. Linking (default):

```
gcc hello.o -o hello        # Link object files to create executable
```

- Resolves symbol references
- Combines object files and libraries
- Creates final executable

Complete Compilation Examples

Single File Compilation:

```
gcc -Wall hello.c                    # Compile with warnings to a.out
gcc -Wall -o myprogram hello.c       # Compile to named executable
gcc -Wall -g -o myprogram hello.c    # Include debug information
```

Multi-file Compilation:

```
# Method 1: One command
gcc -Wall file1.c file2.c file3.c -o program

# Method 2: Separate compilation
gcc -Wall -c file1.c      # Creates file1.o
gcc -Wall -c file2.c      # Creates file2.o
gcc -Wall -c file3.c      # Creates file3.o
gcc file1.o file2.o file3.o -o program
```

GCC Flags and Options

Warning Flags:

```
-Wall              # Enable most warnings
-Wextra            # Enable extra warnings not in -Wall
-Wcomment          # Warn about nested comments
-Wformat           # Warn about printf/scanf format mismatches
-Wunused           # Warn about unused variables
-Wimplicit         # Warn about functions used before declaration
-Wconversion       # Warn about implicit type conversions
-Wshadow           # Warn about shadowed variables
-Werror            # Treat warnings as errors
```

Optimization Flags:

```
-O0                # No optimization (default, good for debugging)
-O1                # Basic optimization
-O2                # Standard optimization (recommended for production)
```

```
-O3                # Aggressive optimization
-Os                # Optimize for size
-Og                # Optimize for debugging experience
```

Debug and Development Flags:

```
-g                 # Generate debug information
-ggdb              # Generate debug info optimized for GDB
-v                 # Verbose compilation
-static            # Force static linking
```

Library and Include Flags:

```
-I<directory>      # Add include directory
-L<directory>      # Add library search directory
-l<library>        # Link with library (e.g., -lm for math)
-fPIC              # Generate position-independent code (for shared libs)
-shared            # Create shared library
```

Header File Management

Include Syntax:

```
#include <stdio.h>     // System headers (search in default paths)
#include "myheader.h"  // User headers (search in current dir first)
```

Include Path Options:

```
# Method 1: Command line flag
gcc -I/path/to/headers -I. program.c

# Method 2: Environment variable
export C_INCLUDE_PATH="/path/to/headers:."
gcc program.c
```

Header Guard Example:

```
#ifndef MYHEADER_H
#define MYHEADER_H
// Header content here
#endif
```

Macros and Preprocessor

Defining Macros:

```
#define MAX_SIZE 100
#define SQUARE(x) ((x)*(x))
#define DEBUG_PRINT(x) printf("DEBUG: %s\n", x)
```

Command-line Macro Definition:

```
gcc -DDEBUG program.c              # Define DEBUG macro
gcc -DMAX_SIZE=200 program.c       # Define with value
gcc -DMYSTRING="hello" program.c   # Define string macro
```

Conditional Compilation:

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

#ifndef RELEASE
    // Development code
#else
    // Release code
#endif

#if defined(DEBUG) && DEBUG > 1
    // Detailed debugging
#endif
```

Multi-file C Programming Structure

File Organization Best Practices

Header Files (.h):

- Type definitions (`typedef`, `struct`, `enum`)
- Function prototypes
- Macro definitions (`#define`)
- External variable declarations (`extern`)
- Should NOT contain function implementations
- Should NOT contain variable definitions

Source Files (.c):

- Function implementations
- Global variable definitions

- Local helper functions
- Include necessary header files

Example Multi-file Project (Staque Library)

defs.h:

```
#ifndef DEFS_H
#define DEFS_H

typedef struct _node {
    int data;
    struct _node *next;
} node;

typedef node *nodep;

#endif
```

stack.h:

```
#ifndef STACK_H
#define STACK_H

#include "defs.h"

typedef nodep stack;

stack initstack(void);
int emptystack(stack s);
int top(stack s);
stack push(stack s, int data);
stack pop(stack s);
void printstack(stack s);
stack destroystack(stack s);

#endif
```

queue.h:

```
#ifndef QUEUE_H
#define QUEUE_H

#include "defs.h"

typedef struct {
    nodep front;
    nodep back;
} queue;

queue initqueue(void);
int emptyqueue(queue q);
```

```
int front(queue q);
queue enqueue(queue q, int data);
queue dequeue(queue q);
void printqueue(queue q);
queue destroyqueue(queue q);

#endif
```

Compilation Commands:

```
# All at once
gcc -Wall staquecheck.c stack.c queue.c -o program

# Separate compilation
gcc -Wall -c stack.c
gcc -Wall -c queue.c
gcc -Wall -o program staquecheck.c stack.o queue.o

# With include path
gcc -Wall -I. -c stack.c
gcc -Wall -I. -c queue.c
gcc -Wall -I. -o program staquecheck.c stack.o queue.o
```

Static and Dynamic Libraries

Static Libraries (.a files)

Creating Static Libraries:

```
# Step 1: Compile source files to object files
gcc -Wall -c stack.c        # Creates stack.o
gcc -Wall -c queue.c        # Creates queue.o

# Step 2: Create static library using ar command
ar rcs libstaque.a stack.o queue.o

# Verify library contents
nm libstaque.a              # Show symbols in library
```

Using Static Libraries:

```
# Method 1: Link with library
gcc -Wall staquecheck.c -L. -lstaque -o program

# Method 2: Direct linking
gcc -Wall staquecheck.c libstaque.a -o program

# Set library path environment variable
export LIBRARY_PATH=$LIBRARY_PATH:.
gcc -Wall staquecheck.c -lstaque -o program
```

Static Library Characteristics:

- Library code included in final executable
- Larger executable size
- No runtime dependencies
- Self-contained executables
- Updates require recompilation

Dynamic/Shared Libraries (.so files)

Creating Shared Libraries:

```
# Step 1: Compile with position-independent code
gcc -Wall -fPIC -c stack.c   # Creates stack.o
gcc -Wall -fPIC -c queue.c   # Creates queue.o

# Step 2: Create shared library
gcc -shared -o libstaque.so stack.o queue.o
```

Using Shared Libraries:

```
# Compile (same as static)
gcc -Wall staquecheck.c -L. -lstaque -o program

# Check dependencies
ldd program                  # Show shared library dependencies

# Set runtime library path
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:.
./program

# Alternative: embed library path during compilation
gcc -Wall -Wl,-rpath,. staquecheck.c -L. -lstaque -o program
```

Shared Library Characteristics:

- Smaller executable size
- Runtime dependencies required
- Shared among multiple programs
- Can update library without recompiling programs
- Requires library to be present at runtime

Library Debugging Commands

```
# Static library analysis
ar t libname.a              # List files in archive
nm libname.a                # Show symbols
ar x libname.a              # Extract object files

# Shared library analysis
ldd executable              # Show dependencies
nm -D libname.so            # Show dynamic symbols
objdump -T libname.so       # Show symbol table

# General debugging
file filename               # Show file type
size executable             # Show section sizes
strip executable            # Remove debug symbols
```

Makefiles

Basic Makefile Structure

Rule Format:

```
target: dependencies
        command1
        command2
        ...
```

CRITICAL: Commands must be indented with TAB characters, not spaces!

Simple Makefile Example

```
# Variables
CC = gcc
CFLAGS = -Wall -g
TARGET = program
SOURCES = main.c utils.c calculate.c
OBJECTS = $(SOURCES:.c=.o)

# Phony targets
.PHONY: all clean

# Default target
all: $(TARGET)

# Main target
$(TARGET): $(OBJECTS)
        $(CC) $(OBJECTS) -o $(TARGET)

# Pattern rule for object files
%.o: %.c
```

```
	$(CC) $(CFLAGS) -c $< -o $@

# Specific dependencies
main.o: main.c main.h utils.h
utils.o: utils.c utils.h
calculate.o: calculate.c calculate.h utils.h

# Clean target
clean:
	rm -f $(OBJECTS) $(TARGET)
```

Makefile Variables

Standard Variables:

```
CC = gcc                    # C compiler
CXX = g++                   # C++ compiler
CFLAGS = -Wall -g           # C compiler flags
CPPFLAGS = -DDEBUG          # Preprocessor flags
LDFLAGS = -L/usr/local/lib  # Linker flags (paths)
LDLIBS = -lm -lpthread      # Libraries to link
AR = ar rcs                 # Archive command
RM = rm -f                  # Remove command
```

Variable Types:

```
# Recursive assignment (=)
VAR1 = $(VAR2) file.o
VAR2 = main.o

# Simple assignment (:=)
VAR3 := $(VAR1)            # Evaluated immediately

# Append (+=)
CFLAGS += -O2
```

Automatic Variables:

```
$@          # Target name
$<          # First prerequisite
$^          # All prerequisites
$?          # Prerequisites newer than target
$*          # Stem of pattern rule
```

Advanced Makefile Features

Pattern Rules:

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

lib%.a: %.o
        $(AR) $@ $<
```

Conditional Statements:

```
ifdef DEBUG
    CFLAGS += -g -DDEBUG
else
    CFLAGS += -O2 -DNDEBUG
endif

ifeq ($(CC),gcc)
    CFLAGS += -Wall
endif
```

Functions:

```
SOURCES := $(wildcard *.c)           # Get all .c files
OBJECTS := $(patsubst %.c,%.o,$(SOURCES))  # Convert .c to .o
DIRS := $(dir $(SOURCES))            # Get directories
```

Multi-directory Projects

Recursive Make:

```
SUBDIRS = lib src tests

all:
        for dir in $(SUBDIRS); do \
                $(MAKE) -C $$dir; \
        done

clean:
        for dir in $(SUBDIRS); do \
                $(MAKE) -C $$dir clean; \
        done
```

Including Makefiles:

```
include common.mk
include $(SUBDIR)/Makefile
-include optional.mk         # Don't error if missing
```

Library-specific Makefile

```
# Static and shared library makefile
CC = gcc
CFLAGS = -Wall -g -fPIC
AR = ar rcs
SOURCES = stack.c queue.c
OBJECTS = $(SOURCES:.c=.o)
STATIC_LIB = libstaque.a
SHARED_LIB = libstaque.so

.PHONY: all static shared clean

all: static shared

static: $(STATIC_LIB)

shared: $(SHARED_LIB)

$(STATIC_LIB): $(OBJECTS)
        $(AR) $@ $^

$(SHARED_LIB): $(OBJECTS)
        $(CC) -shared -o $@ $^

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

# Dependencies
stack.o: stack.c stack.h defs.h
queue.o: queue.c queue.h defs.h

clean:
        $(RM) $(OBJECTS) $(STATIC_LIB) $(SHARED_LIB)

install: $(STATIC_LIB) $(SHARED_LIB)
        cp $(STATIC_LIB) /usr/local/lib/
        cp $(SHARED_LIB) /usr/local/lib/
        ldconfig
```

Make Command Usage

```
make                    # Build default target
make target             # Build specific target
make -j4                # Parallel build (4 jobs)
make -n                 # Dry run (show commands)
make -f Makefile.alt    # Use specific makefile
make clean              # Build clean target
make -d                 # Debug mode
make -s                 # Silent mode
make VAR=value target   # Override variable
```

Complete Project Example

Project Structure

```
project/
├── include/
│   ├── defs.h
│   ├── stack.h
│   └── queue.h
├── src/
│   ├── stack.c
│   └── queue.c
├── test/
│   └── staquecheck.c
├── lib/
├── bin/
└── Makefile
```

Complete Makefile

```
# Project configuration
PROJECT = staque
VERSION = 1.0

# Directories
SRCDIR = src
INCDIR = include
TESTDIR = test
LIBDIR = lib
BINDIR = bin

# Compiler and flags
CC = gcc
CFLAGS = -Wall -Wextra -g -std=c99
CPPFLAGS = -I$(INCDIR)
LDFLAGS = -L$(LIBDIR)
LDLIBS = -l$(PROJECT)

# Files
SOURCES = $(wildcard $(SRCDIR)/*.c)
OBJECTS = $(SOURCES:$(SRCDIR)/%.c=%.o)
HEADERS = $(wildcard $(INCDIR)/*.h)

# Libraries
STATIC_LIB = $(LIBDIR)/lib$(PROJECT).a
SHARED_LIB = $(LIBDIR)/lib$(PROJECT).so

# Test program
TEST_SRC = $(TESTDIR)/staquecheck.c
TEST_BIN = $(BINDIR)/test_$(PROJECT)

# Phony targets
.PHONY: all static shared test clean install dirs
```

```
# Default target
all: dirs static shared test

# Create directories
dirs:
	mkdir -p $(LIBDIR) $(BINDIR)

# Static library
static: $(STATIC_LIB)

$(STATIC_LIB): $(OBJECTS) | $(LIBDIR)
	$(AR) rcs $@ $^

# Shared library
shared: $(SHARED_LIB)

$(SHARED_LIB): CFLAGS += -fPIC
$(SHARED_LIB): $(OBJECTS) | $(LIBDIR)
	$(CC) -shared -o $@ $^

# Object files
%.o: $(SRCDIR)/%.c $(HEADERS)
	$(CC) $(CFLAGS) $(CPPFLAGS) -c $< -o $@

# Test program
test: $(TEST_BIN)

$(TEST_BIN): $(TEST_SRC) $(STATIC_LIB) | $(BINDIR)
	$(CC) $(CFLAGS) $(CPPFLAGS) $(LDFLAGS) $< $(LDLIBS) -o $@

# Clean
clean:
	$(RM) *.o $(STATIC_LIB) $(SHARED_LIB) $(TEST_BIN)

# Install
install: $(STATIC_LIB) $(SHARED_LIB)
	install -d /usr/local/lib /usr/local/include/$(PROJECT)
	install -m 644 $(STATIC_LIB) /usr/local/lib/
	install -m 755 $(SHARED_LIB) /usr/local/lib/
	install -m 644 $(HEADERS) /usr/local/include/$(PROJECT)/
	ldconfig

# Uninstall
uninstall:
	$(RM) /usr/local/lib/lib$(PROJECT).*
	$(RM) -r /usr/local/include/$(PROJECT)
	ldconfig

# Debug info
debug:
	@echo "Sources: $(SOURCES)"
	@echo "Objects: $(OBJECTS)"
	@echo "Headers: $(HEADERS)"
	@echo "CFLAGS: $(CFLAGS)"
```

Common Pitfalls and Troubleshooting

Makefile Issues

1. **TAB vs Spaces:** Commands MUST use TAB indentation
2. **Missing Dependencies:** Files not rebuilding when headers change
3. **Wrong Variable Types:** Using `=` vs `:=` incorrectly
4. **Phony Targets:** Not marking targets like `clean` as phony
5. **Library Order:** Libraries must come after object files in link line

Compilation Issues

1. **Include Path Problems:** Headers not found
2. **Library Linking:** Wrong order, missing -L flags
3. **Undefined References:** Missing function definitions or libraries
4. **Symbol Conflicts:** Multiple definitions of same symbol

Environment Issues

1. **Path Problems:** Commands not found
2. **Library Path:** Shared libraries not found at runtime
3. **Permission Issues:** Cannot write to directories
4. **Shell Differences:** Commands behaving differently in different shells

Debugging Commands

```
# Compilation debugging
gcc -v file.c                    # Verbose compilation
gcc -E file.c | less             # Check preprocessor output
gcc -S file.c; cat file.s        # Check assembly output

# Library debugging
nm -g library.a                  # Show global symbols
objdump -t file.o                # Show symbol table
ldd executable                   # Show shared library dependencies

# Makefile debugging
make -n                          # Show commands without executing
make -d                          # Debug make decisions
make -p                          # Print make database
make --trace                     # Trace rule execution
```

This comprehensive guide covers all the essential commands, concepts, and best practices for GCC compilation, library management, and makefile creation as presented in the provided documentation files.

⁂

1. Example-Staque.pdf
2. gcc.pdf
3. library.pdf
4. Linux-Commands.pdf
5. makefile-1.pdf
6. makefile.pdf