

GNU Build Tools

LinuXpert Systems

WhatsApp: 98841 65649, 90808 01382

E-mail: baskar@linuxpert.in

TABLE OF CONTENTS

1. GNU Build and Binary Tools 1

2. gcc 2

3. g++ 8

4. make 16

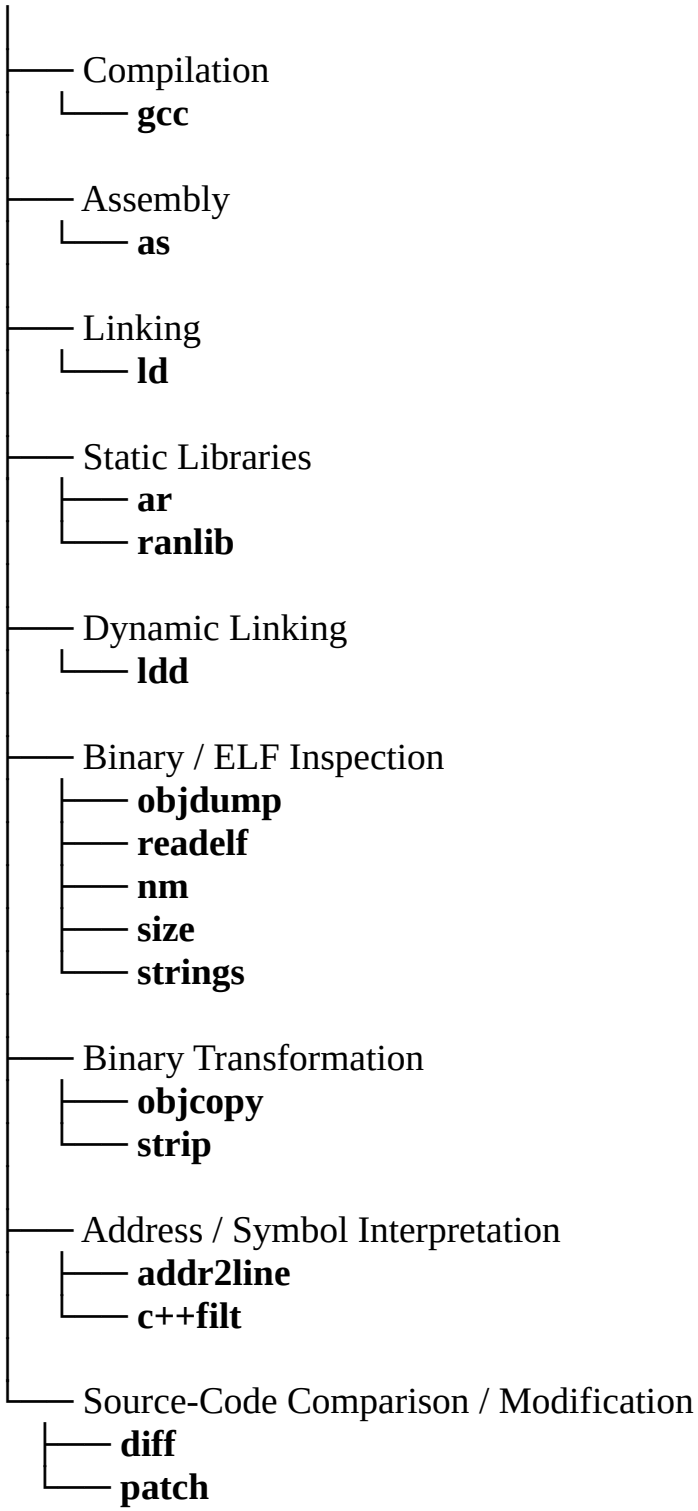
5. cmake 22

6. as 29

7. gdb 75

8. git 83

GNU Build and Binary Tools



2. gcc

gcc (GNU Compiler Collection) is the standard compiler used on Linux and Unix-like systems to compile C, C++, Objective-C, Fortran, Ada, Go, and several other programming languages.

Although the name is **GNU C Compiler**, today it refers to the **GNU Compiler Collection**.

Syntax

```
gcc [options] file...
```

Basic Example

Suppose you have a C source file named **`hello.c`**

```
#include <stdio.h>

int main() {
    printf("Hello, World!\n");
    return 0;
}
```

Compile it:

```
gcc hello.c
```

This creates an executable named:

```
a.out
```

Run it:

```
./a.out
```

Output:

```
Hello, World!
```

Specify Output File

Instead of `a.out`, use the `-o` option.

```
gcc hello.c -o hello
```

Run:

```
./hello
```

Compile Multiple Files

```
gcc main.c math.c util.c -o program
```

This compiles all source files and links them into a single executable.

Compile Without Linking

The `-c` option compiles source code into an object (`.o`) file without creating an executable.

```
gcc -c hello.c
```

Output:

```
hello.o
```

Compile with Debug Information

The `-g` option includes debugging symbols.

```
gcc -g hello.c -o hello
```

Useful with:

```
gdb hello
```

Enable Compiler Warnings

```
gcc -Wall hello.c
```

Common warning options:

```
-Wall
-Wextra
-Wpedantic
```

Example:

```
gcc -Wall -Wextra hello.c
```

Optimise the Program

```
gcc -O2 hello.c
```

Optimization levels:

Option	Meaning
`-O0`	No optimization (default for debugging)
`-O1`	Basic optimization
`-O2`	Recommended optimization
`-O3`	Maximum optimization
`-Os`	Optimize for size
`-Ofast`	Aggressive optimization (may ignore strict standards)

Include Header Directories

```
gcc -Iinclude hello.c
```

Looks for header files in:

```
include/
```

Link Libraries

Link the math library:

```
gcc program.c -lm
```

Example:

```
gcc sqrt.c -lm
```

Define Preprocessor Macros

```
gcc -DDEBUG program.c
```

Equivalent to:

```
#define DEBUG
```

Produce Assembly Code

```
gcc -S hello.c
```

Output:

```
hello.s
```

Generate Object File Only

```
gcc -c hello.c
```

Produces:

```
hello.o
```

View Preprocessed Output

```
gcc -E hello.c
```

Shows the source after preprocessing (header expansion, macro substitution, etc.).

Specify C Standard

```
gcc -std=c11 hello.c
```

Common standards:

Option	Standard
`-std=c89`	ANSI C
`-std=c99`	C99
`-std=c11`	C11
`-std=c17`	C17
`-std=c23`	C23

Commonly Used Options

Option	Description
`-o file`	Output executable name
`-c`	Compile only
`-g`	Include debugging information
`-Wall`	Enable common warnings
`-Wextra`	Enable additional warnings
`-O2`	Optimize generated code
`-I dir`	Add header search path
`-L dir`	Add library search path
`-l library`	Link with a library
`-E`	Run only the preprocessor
`-S`	Generate assembly code
`-std=c11`	Use the C11 language standard
`-DNAME`	Define a preprocessor macro

Compilation Stages

When you run:

```
gcc hello.c -o hello
```

GCC performs these stages:

```
hello.c
    │
    ▼
Preprocessor
    │
    ▼
Compiler
    │
    ▼
Assembler
    │
    ▼
Object File (.o)
    │
    ▼
Linker
    │
    ▼
Executable (hello)
```

Practical Examples

Compile normally:

```
gcc main.c -o main
```

Compile with warnings:

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

Compile with debugging:

```
gcc -g main.c -o main
```

Compile with optimization:

```
gcc -O2 main.c -o main
```

Compile multiple files:

```
gcc main.c utils.c math.c -o app
```

Link the math library:

```
gcc trig.c -lm -o trig
```

Compile to an object file:

```
gcc -c main.c
```

Generate assembly:

```
gcc -S main.c
```

When is `gcc` Used ?

- Developing C programs
- Building Linux software
- Compiling open-source projects
- Creating executables from source code
- Generating object and assembly files
- Debugging applications with `gdb`
- Optimizing programs for performance or size

`gcc` is one of the most important development tools on Linux and forms the foundation of many software build systems, such as **`make`, `cmake`**, and **`autotools`.**

3. g++

g++ is the **GNU C++ compiler**. It is used to **compile, assemble, and link C++ source code** into an executable program.

It is part of the **GNU Compiler Collection (GCC)**.

Syntax

```
g++ [options] source_file(s)
```

Example:

```
g++ hello.cpp
```

This compiles `hello.cpp` and creates an executable named:

```
a.out
```

Run it using:

```
./a.out
```

Basic Examples

1. Compile a C++ program

```
g++ hello.cpp
```

Creates:

```
a.out
```

2. Specify the output filename

```
g++ hello.cpp -o hello
```

Output:

```
hello
```

Run:

```
./hello
```

---

3. Compile multiple source files

```
g++ main.cpp math.cpp utils.cpp -o program
```

All files are compiled and linked into one executable.

---

4. Compile without linking

```
g++ -c hello.cpp
```

Produces:

```
hello.o
```

```
(Object file)
```

---

5. Link object files

```
g++ main.o math.o -o app
```

Creates the executable `app`.

Common Options

Option	Description
`-o file`	Specify output filename
`-c`	Compile only (no linking)
`-g`	Include debugging information
`-O1`, `-O2`, `-O3`	Enable optimization
`-Wall`	Enable most compiler warnings
`-Wextra`	Enable additional warnings
`-Werror`	Treat warnings as errors
`-std=c++11`	Use C++11 standard
`-std=c++14`	Use C++14 standard
`-std=c++17`	Use C++17 standard
`-std=c++20`	Use C++20 standard
`-std=c++23`	Use C++23 standard
`-I directory`	Add header search directory
`-L directory`	Add library search directory
`-l library`	Link against a library
`-shared`	Build a shared library
`-static`	Link statically
`-E`	Preprocess only
`-S`	Compile to assembly
`-v`	Display compilation details
`--version`	Show compiler version

Compilation Stages

When you run:

```
g++ hello.cpp -o hello
```

The compiler performs these stages:

```
Source Code (.cpp)
        │
        ▼
Preprocessing
        │
        ▼
Compilation
        │
        ▼
Assembly
        │
        ▼
Object File (.o)
        │
        ▼
Linking
        │
        ▼
Executable
```

Debug Build

```
g++ -g hello.cpp -o hello
```

Includes debugging symbols for use with debuggers like `gdb`.

Optimized Build

```
g++ -O2 hello.cpp -o hello
```

Optimizes the program for better performance.

Enable Warnings

```
g++ -Wall -Wextra hello.cpp
```

Helps detect common programming mistakes.

Compile with a Specific C++ Standard

C++11

```
g++ -std=c++11 hello.cpp
```

C++17

```
g++ -std=c++17 hello.cpp
```

C++20

```
g++ -std=c++20 hello.cpp
```

Using Header Files

Project structure:

```
project/
│
├── main.cpp
├── math.cpp
└── math.h
```

Compile:

```
g++ main.cpp math.cpp -o calculator
```

Include Additional Header Directories

```
g++ main.cpp -I include/
```

Searches for header files in the `include/` directory.

Link a Library

Example using the math library:

```
g++ main.cpp -lm
```

Note: For C++ programs, linking `libm` explicitly is often unnecessary because `g++` automatically links the required standard C++ libraries. It is still common when linking C libraries or in mixed-language projects.

Generate Assembly Code

```
g++ -S hello.cpp
```

Produces:

```
hello.s
```

Generate an Object File

```
g++ -c hello.cpp
```

Produces:

```
hello.o
```

Show Compiler Version

```
g++ --version
```

Example output:

```
g++ (Ubuntu 13.3.0) 13.3.0
```

Difference Between `gcc` and `g++`

Feature	gcc	g++
Primary language	C	C++
Compiles C	Yes	Yes (treats files appropriately by extension)
Compiles C++	Yes (with correct options/extensions)	Yes
Links C++ standard library automatically	No	Yes
Default language for .cpp files	Requires C++ mode if needed	C++
Best choice for C++ programs	Possible, but less convenient	**Recommended**

For **C++** projects, **g++** is generally preferred because it automatically links the **C++** standard library (`libstdc++`), whereas **gcc** may require additional linker options.

Practical Example

Create `hello.cpp`:

```
#include <iostream>

int main() {
    std::cout << "Hello, World!\n";
    return 0;
}
```

Compile:

```
g++ hello.cpp -o hello
```

Run:

```
./hello
```

Output:

```
Hello, World!
```

Summary

* `g++` is the GNU C++ compiler.

- It compiles, assembles, and links C++ programs into executables.
- It automatically links the C++ standard library.
- Common options include `-o`, `-c`, `-g`, `-O2`, `-Wall`, and `-std=c++17` (or newer standards like `c++20`/`c++23`).
- It is the recommended compiler command for C++ development on Linux and other systems using GCC.

4. make

`make` is a **build automation tool** used to automatically compile, link, and build software projects. It reads instructions from a file (usually named **Makefile**) and executes the required commands in the correct order.

It is one of the most commonly used tools for building C and C++ programs, although it can automate almost any repetitive task.

Syntax

```
make [options] [target]
```

- **options** – Modify how `make` behaves.
- **target** – The specific task or build rule to execute.

How `make` Works

1. Reads the **Makefile** in the current directory.
2. Determines which files need updating.
3. Executes only the commands required to rebuild outdated files.
4. Stops if an error occurs (unless instructed otherwise).

For example:

```
main.c
utils.c
utils.h
Makefile
```

Running:

```
make
```

may compile only the source files that have changed since the last build.

Basic Example

Makefile

```
hello: hello.o
       gcc hello.o -o hello

hello.o: hello.c
         gcc -c hello.c

clean:
        rm -f hello hello.o
```

Run:

```
make
```

Output:

```
gcc -c hello.c
gcc hello.o -o hello
```

Common Examples

Build the default target

```
make
```

Builds the first target in the Makefile.

Build a specific target

```
make clean
```

Runs the **`clean`** target.

Compile a program

```
make hello
```

Builds only the **`hello`** target.

Use a different Makefile

```
make -f MyMakefile
```

Uses `MyMakefile` instead of the default `Makefile`.

Run multiple jobs in parallel

```
make -j4
```

Uses 4 CPU cores for faster compilation.

Or use all available cores:

```
make -j
```

Force rebuilding

```
make -B
```

Rebuilds everything regardless of timestamps.

Dry run (show commands only)

```
make -n
```

Displays the commands without executing them.

Show debugging information

```
make --debug
```

Displays detailed information about dependency checking.

Ignore errors

```
make -i
```

Continues even if some commands fail.

Print variables

```
make -p
```

Displays all variables and rules.

Common Options

Option	Description
`-f FILE`	Use a specific Makefile
`-j N`	Run N jobs in parallel
`-k`	Continue after errors when possible
`-B`	Force rebuilding
`-n`	Show commands without executing
`-C DIR`	Change to directory before building
`-i`	Ignore command errors
`-s`	Silent mode (hide command output)
`-p`	Print variables and rules
`-v`	Show version

Typical Workflow

Suppose your project contains:

```
project/
    Makefile
    main.c
    math.c
    math.h
```

Build:

```
make
```

Clean build files:

```
make clean
```

Install program:

```
make install
```

Run tests:

```
make test
```

These targets (`clean`, `install`, `test`) exist only if they are defined in the Makefile.

Advantages

- Automates compilation and other repetitive tasks.
- Rebuilds only files that have changed, saving time.
- Handles complex file dependencies automatically.
- Supports parallel builds to speed up compilation.
- Widely used with C, C++, kernel development, embedded systems, and many open-source projects.

Exit Status

- **0** – Build completed successfully.
- **Non-zero** – An error occurred during the build.

Related Commands

- `gcc` – GNU C/C++ compiler
- `g++` – GNU C++ compiler
- `cmake` – Generates Makefiles or other build-system files
- `ninja` – High-performance build system
- `ld` – Linker
- `ar` – Creates static libraries

Summary

Feature	Description
Command	make
Purpose	Automates software builds and other repetitive tasks
Input File	Makefile (or makefile)
Default Action	Builds the first target in the Makefile
Common Use	Compiling C/C++ projects, cleaning builds, running tests, installing software
Typical User	Software developers, system programmers, DevOps engineers

5. cmake

`cmake` is a **cross-platform build system generator** used to configure, generate, and manage the build process of software projects. It reads project configuration files (typically `CMakeLists.txt`) and generates native build files for tools such as **Make**, **Ninja**, **Visual Studio**, or **Xcode**.

Unlike **`gcc`**, which directly compiles source code, `cmake` prepares the build environment by generating the appropriate build files.

Syntax

```
cmake [options] <source_directory>
```

or

```
cmake [options] -S <source_directory> -B <build_directory>
```

Purpose

- Configure software projects
- Generate platform-specific build files
- Manage compiler options and dependencies
- Support cross-platform development
- Simplify complex software builds

Common Workflow

```
Source Code
     │
     ▼
CMakeLists.txt
     │
     ▼
cmake
     │
     ▼
Generate Makefiles / Ninja files / Visual Studio project
     │
     ▼
make / ninja / IDE
     │
     ▼
Executable Program
```

Basic Examples

1. Configure a project

```
cmake .
```

Uses the current directory as the source directory.

---

2. Configure using a separate build directory (recommended)

```
cmake -S . -B build
```

- Source directory: `.`
- Build directory: `build`

Creates:

```
project/
│
├── CMakeLists.txt
├── main.cpp
└── build/
```

3. Build the project

```
cmake --build build
```

Instead of running `make` directly, this uses the configured build tool automatically.

---

4. Install the program

```
cmake --install build
```

Installs files according to the project's installation rules.

---

5. Specify the compiler

```
cmake -DCMAKE_C_COMPILER=gcc \
      -DCMAKE_CXX_COMPILER=g++
```

---

6. Build in Release mode

```
cmake -S . -B build -DCMAKE_BUILD_TYPE=Release
```

Possible build types:

- Debug
- Release
- RelWithDebInfo
- MinSizeRel

---

7. Build in Debug mode

```
cmake -S . -B build -DCMAKE_BUILD_TYPE=Debug
```

Includes debugging symbols.

8. Generate Ninja build files

```
cmake -G Ninja -S . -B build
```

---

9. Generate Unix Makefiles

```
cmake -G "Unix Makefiles" -S . -B build
```

---

10. List available generators

```
cmake --help
```

Look for the "Generators" section.

Frequently Used Options

Option	Description
`-S`	Source directory
`-B`	Build directory
`-G`	Select build system generator
`-DVAR=value`	Set a CMake variable
`--build`	Build a configured project
`--install`	Install built files
`--version`	Display CMake version
`--help`	Show help

Typical Project Structure

```
MyProject/
│
├── CMakeLists.txt
├── src/
│   ├── main.cpp
│   └── utils.cpp
│
├── include/
│   └── utils.h
│
└── build/
```

`build/` contains generated files and is usually not committed to version control.

Simple `CMakeLists.txt`

```
cmake_minimum_required(VERSION 3.10)

project(Hello)

add_executable(hello main.cpp)
```

Compile using:

```
cmake -S . -B build
cmake --build build
```

Run:

```
./build/hello
```

Build Process

```
Source Files
     │
     ▼
CMakeLists.txt
     │
     ▼
cmake
     │
     ▼
Configuration
     │
     ▼
Makefiles / Ninja Files
     │
     ▼
make / ninja
     │
     ▼
Executable
```

Common Commands

Command	Purpose
`cmake -S . -B build`	Configure project
`cmake --build build`	Build project
`cmake --install build`	Install project
`cmake --build build --clean-first`	Clean then build
`cmake --build build --target install`	Build install target
`cmake --version`	Show CMake version

Advantages

- Cross-platform
- Supports many compilers
- Generates Makefiles, Ninja files, Visual Studio solutions, Xcode projects, and more
- Handles large multi-directory projects
- Integrates with external libraries and package managers
- Supports testing (CTest) and packaging (CPack)

Common Use Cases

- Building C and C++ applications
- Embedded systems development
- Open-source projects
- Cross-platform software
- Large modular codebases
- Library development

Difference Between `cmake`, `gcc`, and `make`

Tool	Purpose
`cmake`	Configures a project and generates native build files
`gcc`	Compiles C source code into object files or executables
`g++`	Compiles C++ source code
`make`	Executes build instructions from a Makefile
`ninja`	Fast build tool that executes build instructions generated by CMake

---

A common build sequence is:

```
cmake -S . -B build      # Configure and generate build files

cmake --build build      # Compile the project

./build/my_program       # Run the executable
```

This out-of-source build approach keeps generated files separate from your source code, making the project easier to manage and clean.

6. as

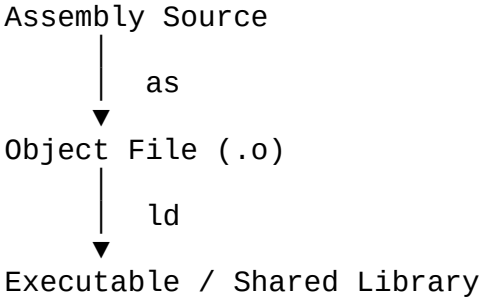
`as` is the **GNU assembler** used on Linux to convert **assembly language source code into machine-code object files**.

It is part of the **GNU Binutils** package and is a fundamental component of the Linux software-development toolchain.

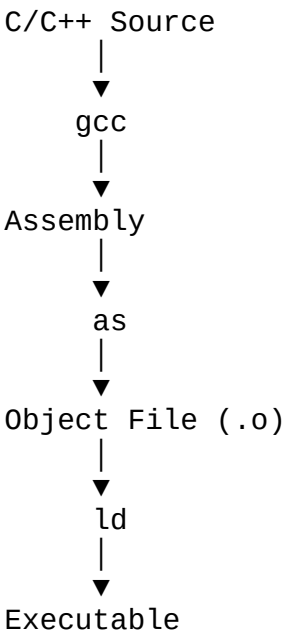
In simple terms:

`as` converts assembly language into an object file that can later be linked by **`ld`** into an executable or shared library.

The basic compilation flow is:



When C/C++ is compiled with GCC, GCC may invoke the assembler automatically:



2. Basic Syntax

```
as [OPTIONS] SOURCE_FILE -o OUTPUT_FILE
```

For example:

```
as program.s -o program.o
```

This converts:

```
program.s
```

into:

```
program.o
```

The resulting **`.o`** file is normally an **ELF relocatable object file** on Linux.

---

3. Check Whether ‘`as`’ Is Installed

Use:

```
which as
```

Typical output:

```
/usr/bin/as
```

Check its version:

```
as --version
```

You may see:

```
GNU assembler (GNU Binutils for Ubuntu)
```

On Ubuntu/Debian, it is normally installed as part of:

```
binutils
```

Install it with:

```
sudo apt install binutils
```

4. Why Is the Command Called `as`?

The command name is simply:

```
as
```

meaning:

```
Assembler
```

It is part of the GNU Binary Utilities:

```
GNU Binutils
│
├── as       → assembler
├── ld       → linker
├── nm       → symbol viewer
├── objdump  → binary/assembly inspector
├── readelf  → ELF inspector
├── ar       → archive/library manager
└── strip    → remove symbols/debug information
```

Together these tools form an important part of the Linux development environment.

---

5. First Assembly Program

Create:

```
nano hello.s
```

For x86-64 Linux using GNU assembler syntax:

```
.section .text

.global _start

_start:
    mov $60, %rax
    mov $0, %rdi
    syscall
```

This program directly invokes the Linux `exit` system call.

6. Assemble the Program

Run:

```
as hello.s -o hello.o
```

Now:

```
ls -l hello.o
```

You should have:

```
hello.o
```

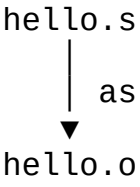
Check its type:

```
file hello.o
```

Typical output:

```
hello.o: ELF 64-bit LSB relocatable, x86-64, ...
```

This is an important concept:



The assembler has **not yet produced the final executable**.

---

7. Link the Object File

Now use `ld`:

```
ld hello.o -o hello
```

Then:

```
file hello
```

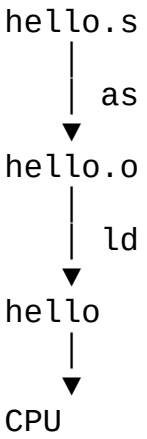
You should see an ELF executable.

Run it:

```
./hello
```

The program exits immediately.

The complete process is:



---

8. `as` vs `gcc`

You can assemble directly with:

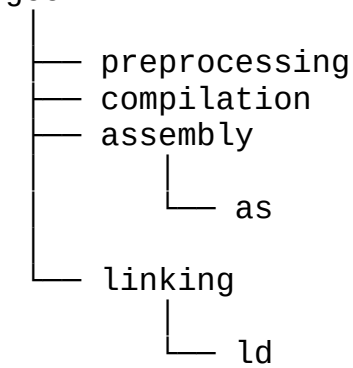
```
as hello.s -o hello.o
```

But GCC can also invoke the assembler.

For example:

```
gcc -c hello.s -o hello.o
```

Conceptually:



When learning the toolchain, using **`as`** directly helps students understand what GCC is doing behind the scenes.

9. Assembly Source File Extensions

Common extensions include:

`.s`
`.S`

There is an important distinction.
`.s`

Normally treated as assembly source without C preprocessing.

```
as program.s -o program.o
```
`.S`

Typically passed through the C preprocessor when using GCC:

```
gcc -c program.S
```

This allows assembly source to use:

```
#define
#include
#if
#ifdef
```

and other preprocessing facilities.

---

10. GNU Assembly Syntax

GNU assembler on x86 commonly uses **AT&T syntax**.

Example:

```
mov $10, %rax
```

The general pattern is:

```
instruction source, destination
```

For example:

```
mov %rax, %rbx
```

means:
`RBX ← RAX`

GNU `as` also supports architecture-dependent syntax options.

11. Intel Syntax and GNU `as`

When using GNU assembler directly, x86 assembly is commonly written in AT&T syntax.

For example:

```
mov $10, %rax
```

By contrast, Intel syntax would normally be:

```
mov rax, 10
```

This distinction is important when moving between:

- GNU `as`
- `objdump`
- NASM
- Intel manuals
- compiler-generated assembly

For example, `objdump` can display GNU-assembled code using Intel syntax:

```
objdump -d -M intel hello.o
```

---

12. `.section`

The assembler uses directives to organize assembly source.

For example:

```
.section .text
```

means that subsequent instructions belong to the `.text` section.

A typical assembly file may contain:

```
.section .text

.global main

main:
    ...

.section .data
```

```
message:
    .asciz "Hello"
```

Important sections include:

```
.text
    Machine instructions

.data
    Initialized writable data

.rodata
    Read-only data

.bss
    Uninitialized data
```

---

13. `.global`

The `.global` directive makes a symbol visible to the linker.

Example:

```
.global _start
```

Then:

```
_start:
```

This allows the linker to find `_start`.

Without appropriate symbol visibility, another object file may not be able to reference the symbol.

Conceptually:

```
.global _start
       |
       ▼
Global Symbol
       |
       ▼
Linker can resolve it
```

14. `.globl`

GNU assembler also accepts:

```
.globl _start
```

This is equivalent to:

```
.global _start
```

For example:

```
.section .text
.globl _start

_start:
    mov $60, %rax
    mov $0, %rdi
    syscall
```

---

15. `.text`

The `.text` directive selects the code section:

```
.section .text
```

or commonly:

```
.text
```

Example:

```
.text

.global add

add:
    mov %edi, %eax
    add %esi, %eax
    ret
```

This creates machine instructions in `.text`.

16. `.data`

The `.data` directive selects initialized writable data.

Example:

```
.data
```

```
number:
    .long 100
```

The resulting object file will contain a `.data` section.

You can inspect it using:

```
objdump -h program.o
```

and:

```
objdump -s -j .data program.o
```

---

17. `.rodata`

Read-only data can be placed in:

```
.section .rodata
```

For example:

```
.section .rodata
```

```
message:
    .asciz "Hello World"
```

This is useful for constants and strings.

---

18. `.bss`

The BSS section is used for uninitialized storage.

For example:

```
.section .bss
```

```
.lcomm buffer, 1024
```

This reserves storage without storing 1024 bytes of initial data in the executable.

This is conceptually similar to:

```
char buffer[1024];
```

for static/global storage.

---

19. Define Data With `.byte`

The `.byte` directive stores one-byte values.

```
.byte 10
```

Multiple values:

```
.byte 10, 20, 30, 40
```

Example:

```
.data

values:
    .byte 10, 20, 30, 40
```

---

20. Define 16-bit Data With `.word`

```
.word 1000
```

Example:

```
.data

value:
    .word 1000
```

21. Define 32-bit Data With `.long`

```
.long 100000
```

Example:

```
.data

number:
    .long 100000
```

---

22. Define 64-bit Data With `.quad`

```
.quad 123456789
```

Example:

```
.data

large_number:
    .quad 123456789
```

---

23. Define Strings With `.ascii`

```
.ascii "Hello"
```

This does **not** automatically add a null terminator.

For example:

```
.ascii "Hello"
```

stores:

```
H e l l o
```

---

24. Define Strings With `.asciz`

Use:

```
.asciz "Hello"
```

This adds a null character at the end.

Conceptually:

```
H e l l o \0
```

This is particularly useful when calling C library functions such as `printf`.

---

25. Comments

GNU assembler comments on x86 commonly begin with:

```
#
```

Example:

```
mov $60, %rax      # exit system call
mov $0, %rdi       # status = 0
syscall
```

Comments are ignored by the assembler.

---

26. Labels

Labels identify locations in the assembly program.

Example:

```
_start:
```

Another:

```
loop:
```

Instructions can reference them:

```
jmp loop
```

The assembler converts these symbolic references into appropriate relocations or addresses as required.

27. Assemble a Function for Use From C

Create:

```
add.s
```

with:

```
.text

.global add

add:
    mov %edi, %eax
    add %esi, %eax
    ret
```

This implements:

```
int add(int a, int b);
```

Assemble:

```
as add.s -o add.o
```

Now create:

```
#include <stdio.h>
int add(int, int);

int main()
{
    printf("%d\n", add(10, 20));
    return 0;
}
```

Compile:

```
gcc -c main.c -o main.o
```

Link:

```
gcc main.o add.o -o program
```

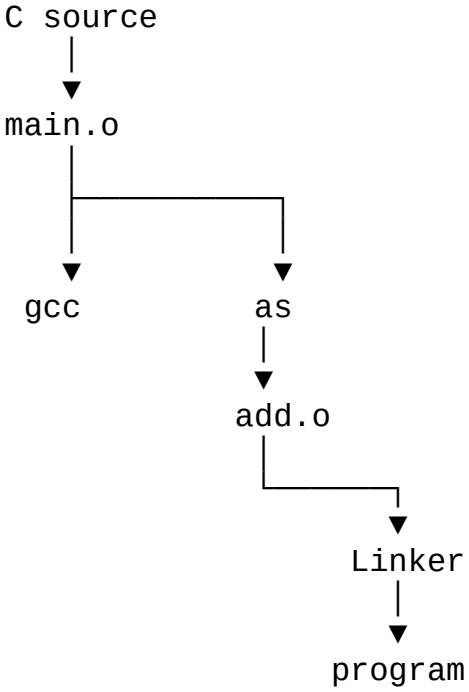
Run:

```
./program
```

Output:

```
30
```

This demonstrates:



28. Inspect the Assembled Object File

After:

```
as add.s -o add.o
```

run:

```
file add.o
```

Then:

```
objdump -h add.o
```

Then:

```
nm add.o
```

You may see:

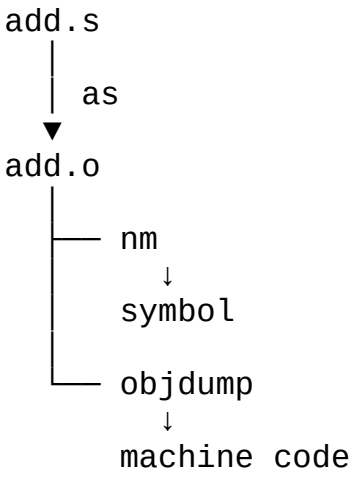
```
0000000000000000 T add
```

Then:

```
objdump -d -M intel add.o
```

You can observe the machine instructions generated by `as`.

Thus:



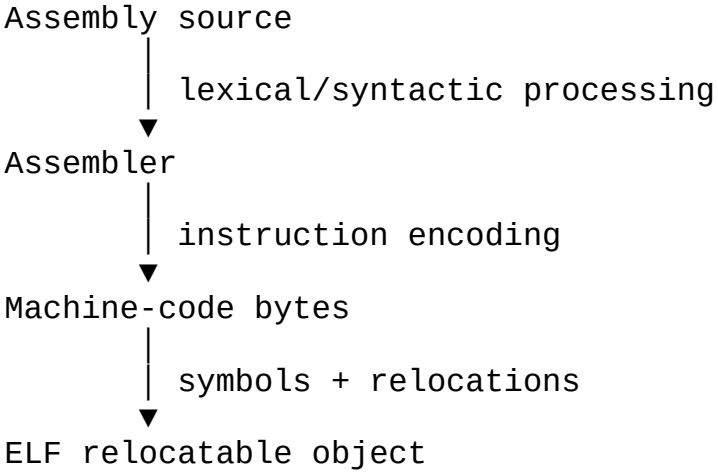
---

29. Assembly Source → Object File

The fundamental operation of `as` is:

```
as source.s -o source.o
```

Conceptually:



The object file contains more than just machine code.

It also contains information needed by the linker.

30. Relocations

Suppose assembly references an external function:

```
call external_function
```

At assembly time, the final address may not yet be known.

The assembler therefore creates a **relocation entry**.

You can inspect it with:

```
objdump -r program.o
```

You might see something conceptually similar to:

```
RELOCATION RECORDS FOR [.text]:
OFFSET           TYPE              VALUE
000000000005     R_X86_64_PLT32    external_function-0x4
```

This demonstrates an important relationship:

```
as
 │
 ├── machine code
 ├── symbols
 └── relocation information
              │
              ▼
             ld
              │
              ▼
       Final addresses
```

---

31. Undefined Symbols Created by `as`

Consider:

```
.text
.global main

.extern external_function

main:
    call external_function
    ret
```

Assemble:

```
as program.s -o program.o
```

Now:

```
nm program.o
```

may show:

```
                 U external_function
0000000000000000 T main
```

This is a direct demonstration of the relationship between the assembler and linker.

---

32. `.extern`

The `.extern` directive can declare an external symbol.

Example:

```
.extern printf
```

Then the assembly code can refer to it:

```
call printf
```

The actual implementation will normally be supplied during linking.

---

33. Inspect Relocations

After assembling:

```
as program.s -o program.o
```

run:

```
objdump -r program.o
```

You can also use:

```
readelf -r program.o
```

This provides a useful learning sequence:

```
Assembly
   ↓
as
   ↓
Object file
   ↓
readelf -r
   ↓
Relocations
```

---

34. Inspect Symbols

Use:

```
nm program.o
```

or:

```
readelf -s program.o
```

The assembler creates and records symbols such as:

```
main
add
external_function
```

The linker later uses these symbols.

---

35. Generate Assembly From C With GCC

Although `as` assembles assembly code, GCC can generate assembly from C:

```
gcc -S program.c
```

This creates:

```
program.s
```

Then assemble it manually:

```
as program.s -o program.o
```

This gives students a very useful experiment:

```
program.c
    │
    │ gcc -S
    ▼
program.s
    │
    │ as
    ▼
program.o
```

This separates the **compiler** from the **assembler**.

---

36. Complete GCC-to-Assembler Workflow

For:

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

GCC performs several stages.

Conceptually:

```
program.c
    │
    ▼
Preprocessor
    │
    ▼
Compiler
    │
    ▼
Assembly (.s)
    │
    ▼
Assembler (as)
    │
    ▼
Object File (.o)
    │
    ▼
Linker (ld)
    │
    ▼
Executable
```

This is why understanding `as` is important when teaching students how GCC actually works.

37. See GCC Invoke the Assembler

Use:

```
gcc -v -c program.c
```

The verbose output can show the compilation stages and commands invoked by GCC.

You can also use:

```
gcc -### -c program.c
```

to show the commands GCC would execute without actually executing them.

This can help students discover:

```
gcc
↓
cc1
↓
as
```

and during linking:

```
ld
```

or a compiler-driver-mediated linker invocation.

---

38. Generate Assembly and Assemble It

This is one of the best experiments for understanding the toolchain.

Start with:

```
gcc -S -O0 program.c -o program.s
```

Inspect:

```
less program.s
```

Then:

```
as program.s -o program.o
```

Inspect:

```
nm program.o
```

and:

```
objdump -d program.o
```

Finally link:

```
gcc program.o -o program
```

Thus:

```
C
│
│ gcc -S
▼
Assembly
│
│ as
▼
Object
│
│ gcc / linker
▼
Executable
```

---

39. `as` and `objdump`

These commands are almost natural partners.

`as`:

```
as program.s -o program.o
```

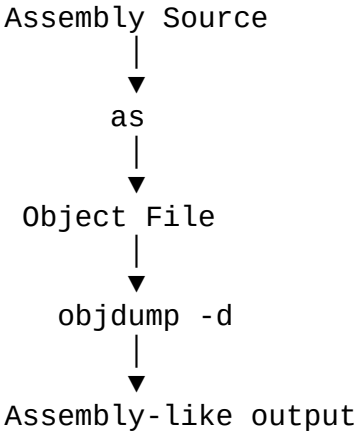
creates the object file.

`objdump`:

```
objdump -d program.o
```

disassembles the machine code.

Therefore:



This allows students to verify what the assembler actually produced.

---

40. `as` and `nm`

`as` creates symbols.

`nm` displays them.

For example:

```
.global add
```

add:

```
    ...
```

Assemble:

```
as add.s -o add.o
```

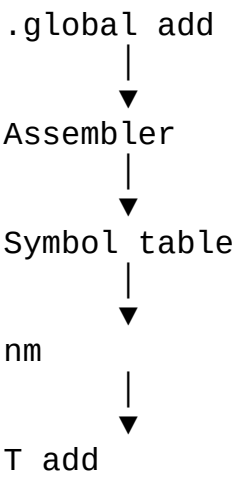
Then:

```
nm add.o
```

Output:

```
0000000000000000 T add
```

Thus:



---

41. `as` and `ld`

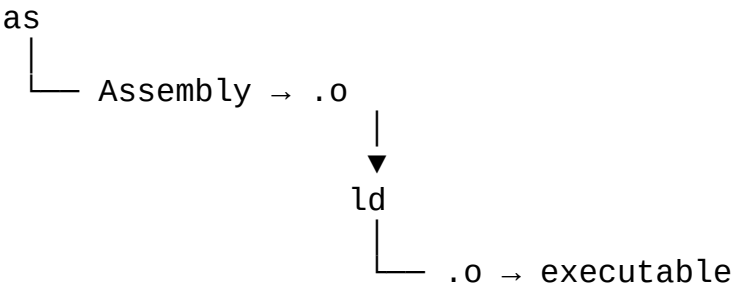
`as` creates relocatable object files:

```
as program.s -o program.o
```

`ld` links them:

```
ld program.o -o program
```

Therefore:



This distinction is fundamental.

`as` assembles; `ld` links.

42. `as` vs `ld`

Feature	as	ld
Reads assembly source	✓	No
Produces object file	✓	Can produce linked output
Encodes instructions	✓	No
Creates symbols	✓	Resolves symbols
Creates relocations	✓	Applies/resolves them
Links libraries	No	✓
Produces final executable	No, normally	✓

The distinction should be emphasized to students:

```
as = assembler
```

```
ld = linker
```

---

43. `as` vs NASM

Both are assemblers, but they use different syntax and toolchain conventions.

Feature	GNU as	NASM
Common syntax	AT&T	Intel
GNU Binutils	✓	No
Linux toolchain integration	**Excellent**	Excellent
Used by GCC	**Yes**	Not normally
ELF support	✓	✓
Intel-style syntax	Via appropriate mechanisms	**Native**

For a GNU/Linux engineering curriculum, **`as`** is especially important because it integrates directly with:

```
gcc
ld
objdump
nm
readelf
```

44. Architecture Matters

as is architecture-dependent.

The assembler must know the target instruction set.

For example:

```
x86-64
ARM64
ARM
RISC-V
PowerPC
MIPS
```

An assembly program written for x86-64 cannot normally be assembled as ARM64 code without changing the source.

The assembler is therefore closely tied to:

```
Target Architecture
        │
        ▼
Instruction Set
        │
        ▼
Assembler
        │
        ▼
Machine Code
```

---

45. Cross-Assembler

Cross-development environments provide architecture-specific assembler tools.

For example, an ARM64 cross-toolchain may contain:

```
aarch64-linux-gnu-as
```

You can check:

```
which aarch64-linux-gnu-as
```

and assemble ARM64 source using the appropriate assembler.

This is particularly important in:

- Embedded Systems
- IoT
- ARM development
- Raspberry Pi development
- Automotive systems
- Mobile platforms
- RISC-V development

---

46. Display `as` Help

Use:

```
as --help
```

For version information:

```
as --version
```

For supported architecture options, the help output is particularly useful.

Because assembler options vary substantially by target architecture, students should treat:

```
as --help
```

as an important reference.

47. Useful `as` Options

Some commonly encountered options include:

Option	Purpose
`-o FILE`	Specify output object file
`-g`	Generate debugging information
`--gdwarf-2`	Generate DWARF-2 debug information
`--gdwarf-4`	Generate DWARF-4 debug information where supported
`--gdwarf-5`	Generate DWARF-5 debug information where supported
`--32`	Generate 32-bit code where supported
`--64`	Generate 64-bit code where supported
`-I DIR`	Add directory to include search path
`-a`	Produce assembly listing
`--listing`	Generate listing output
`--defsym SYMBOL=VALUE`	Define a symbol
`-march=CPU`	Select architecture where supported
`-mcpu=CPU`	Select processor where supported
`-m32` / `-m64`	Target word size where supported

Note: Architecture-specific options differ between x86, ARM, RISC-V, and other targets, so `as --help` should be consulted for the particular toolchain.

---

48. Specify the Output File

The most commonly used option is:

`-o`

Example:

```
as program.s -o program.o
```

Without `-o`, the assembler uses its default output filename behavior, which is generally not what you want in a build system.

Best practice:

```
as program.s -o program.o
```

49. Generate Debug Information

You can assemble with debugging information using:

```
as -g program.s -o program.o
```

This allows debugging tools to associate machine instructions with source-level information when sufficient debugging information is present.

Then:

```
gdb program
```

can make use of the resulting debug information after linking.

---

50. Generate an Assembly Listing

The assembler can generate listing information.

For example:

```
as -al program.s -o program.o
```

Depending on the assembler version/options, the listing can show source lines alongside addresses and generated bytes.

This is useful for studying:

```
Assembly instruction
        ↓
Machine-code bytes
        ↓
Object-file address
```

---

51. `.include`

Assembly source can include another assembly file.

For example:

```
.include "definitions.inc"
```

This is useful for:

- constants
- macros
- reusable definitions
- architecture-specific declarations

The include search path can be extended with:

```
as -I/path/to/includes program.s -o program.o
```

---

52. Assembly Macros

GNU assembler supports macros.

For example:

```
.macro EXIT
    mov $60, %rax
    mov $0, %rdi
    syscall
.endm
```

Then:

```
.text

.global _start

_start:
    EXIT
```

The assembler expands the macro during assembly.

This introduces students to an important idea:

```
Assembly Source
      │
      ▼
Assembler processing
      │
      ├── directives
      ├── macros
      ├── symbols
      └── instruction encoding
      │
      ▼
Object file
```

53. Conditional Assembly

GNU assembler supports conditional directives.

For example:

```
.if 1
    nop
.endif
```

This becomes useful for architecture-specific or configuration-specific assembly code.

Combined with `.S` files and preprocessing, this becomes a powerful mechanism for low-level software development.

---

54. Common Assembler Directives

Some important GNU assembler directives include:

Directive	Purpose
`.text`	Select code section
`.data`	Select initialized data
`.bss`	Select BSS section
`.section`	Select/create section
`.global` / `.globl`	Export symbol
`.extern`	Declare external symbol
`.byte`	Define 8-bit data
`.word`	Define 16-bit data
`.long`	Define 32-bit data
`.quad`	Define 64-bit data
`.ascii`	Store string
`.asciz`	Store null-terminated string
`.align`	Align data/instructions
`.equ`	Define a symbolic constant
`.set`	Set a symbol value
`.macro`	Define macro
`.endm`	End macro
`.include`	Include another assembly file
`.comm`	Define common symbol
`.lcomm`	Define local common storage
`.type`	Specify symbol type
`.size`	Specify symbol size

55. `.equ`

Define a symbolic constant:

```
.equ SYS_EXIT, 60
```

Then:

```
mov $SYS_EXIT, %rax
```

This makes assembly programs easier to maintain.

---

56. `.type`

The `.type` directive can describe the type of a symbol.

For example:

```
.type add, @function

add:
    mov %edi, %eax
    add %esi, %eax
   ret
```

This helps tools such as:

```
nm
readelf
objdump
```

interpret the symbol correctly.

---

57. `.size`

A symbol's size can be specified:

```
.size add, .-add
```

For example:

```
.global add
.type add, @function
```

```
add:
    mov %edi, %eax
    add %esi, %eax
    ret

.size add, .-add
```

This provides useful metadata to the linker and binary-analysis tools.

You can then inspect it with:

```
nm -S add.o
```

---

58. Alignment

The assembler supports alignment directives.

For example:

```
.align 16
```

or architecture-specific variants such as:

```
.p2align 4
```

Alignment can matter for:

- performance
- ABI requirements
- data structures
- SIMD instructions
- cache behavior

This gives students a practical connection between assembly and computer architecture.

---

59. Examine the Result With `readelf`

After:

```
as add.s -o add.o
```

run:

```
readelf -h add.o
```

Then:

```
readelf -S add.o
```

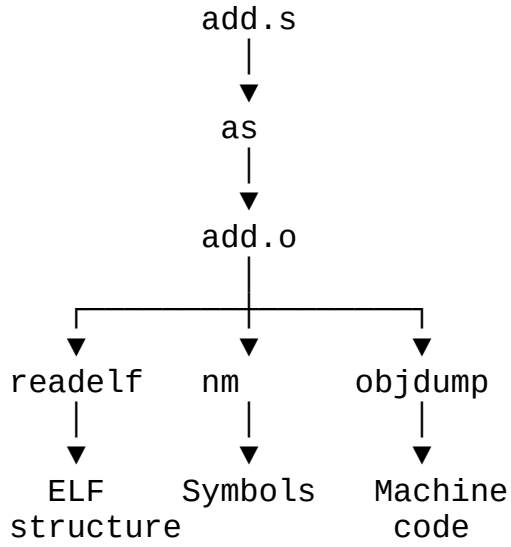
Then:

```
readelf -s add.o
```

Then:

```
readelf -r add.o
```

This demonstrates:



60. Examine the Result With `objdump`

After assembly:

```
objdump -h add.o
```

shows sections.

```
objdump -t add.o
```

shows symbols.

```
objdump -r add.o
```

shows relocations.

```
objdump -d add.o
```

shows machine instructions.

This is an excellent way to demonstrate exactly what the assembler has produced.

---

61. Complete Assembly Investigation

Suppose:

```
program.s
```

contains:

```
.text

.global add
.type add, @function

add:
    mov %edi, %eax
    add %esi, %eax
    ret

.size add, .-add
```

Assemble:

```
as program.s -o program.o
```

Then investigate:

File type

```
file program.o
```

Sections

```
objdump -h program.o
```

Symbols

```
nm program.o
```

Symbol sizes

```
nm -S program.o
```

Relocations

```
objdump -r program.o
```

Disassembly

```
objdump -d -M intel program.o
```

ELF structure

```
readelf -h program.o
readelf -S program.o
readelf -s program.o
```

This gives students a complete view of the output of `as`.

---

62. Practical Laboratory Exercise

Experiment: Assembly → Object → Executable

Step 1 — Create assembly source

```
nano add.s
```

Enter:

```
.text

.global add
.type add, @function

add:
    mov %edi, %eax
    add %esi, %eax
    ret

.size add, .-add
```

Step 2 — Assemble

```
as add.s -o add.o
```

Step 3 — Verify

```
file add.o
```

Step 4 — Inspect symbols

```
nm add.o
```

Expected:

```
0000000000000000 T add
```

Step 5 — Inspect machine code

```
objdump -d -M intel add.o
```

Step 6 — Create a C program

```
#include <stdio.h>

int add(int, int);

int main()
{
    printf("10 + 20 = %d\n", add(10, 20));
    return 0;
}
```

Save as:

```
main.c
```

Step 7 — Compile

```
gcc -c main.c -o main.o
```

Step 8 — Examine the undefined reference

```
nm main.o
```

Look for:

```
U add
```

Step 9 — Link

```
gcc main.o add.o -o program
```

Step 10 — Execute

```
./program
```

Expected:

```
10 + 20 = 30
```

This single exercise demonstrates:

```
Assembly
   ↓
   as
   ↓
Object file
   ↓
   nm
   ↓
Symbol
   ↓
   ld
   ↓
Executable
```

63. Practical Experiment — C to Assembly to Machine Code

Start with:

```
int add(int a, int b)
{
    return a + b;
}
```

Generate assembly:

```
gcc -S -O0 add.c -o add.s
```

Examine:

```
less add.s
```

```
as add.s -o add.o
```

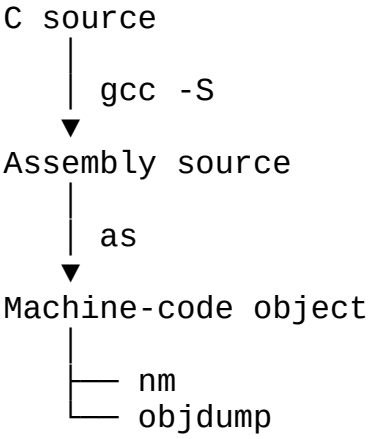
Inspect:

```
objdump -d -M intel add.o
```

Finally inspect symbols:

```
nm add.o
```

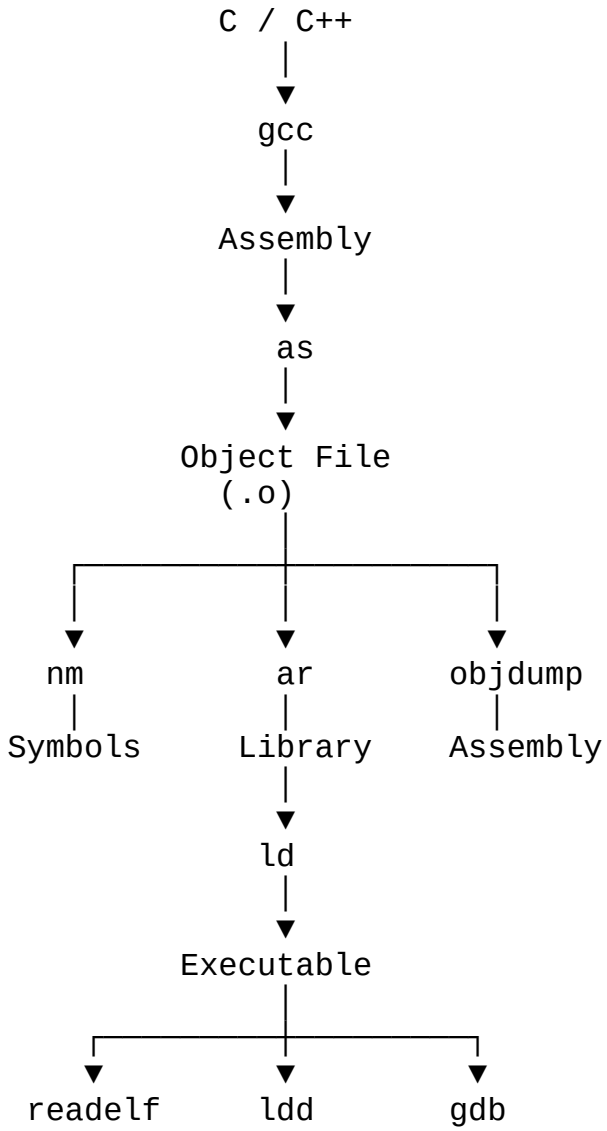
Students can observe:



This is one of the most effective practical exercises for introducing compiler internals.

64. `as` in the Complete GNU Toolchain

The relationship between the commands can be summarized as:



This makes `as` the critical bridge between **assembly language and object code**.

65. `as` and the Compiler

A compiler such as GCC can generate assembly:

```
gcc -S program.c
```

Then:

```
as program.s -o program.o
```

So:

```
gcc -S
   ↓
Assembly
   ↓
as
   ↓
Object
```

This separation is pedagogically valuable because students can see that **compilation and assembly are distinct stages**.

---

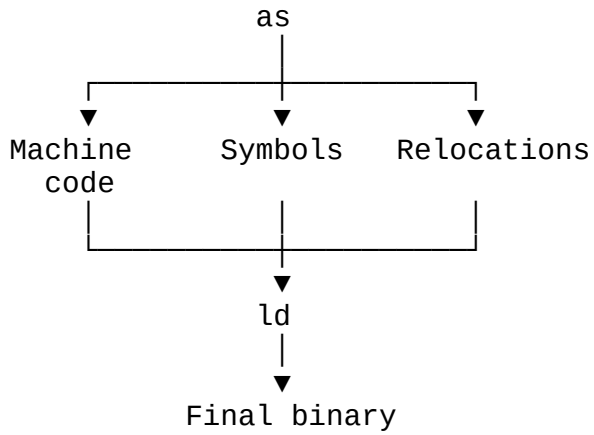
66. `as` and the Linker

The assembler does not normally resolve all external addresses.

For example:

```
call printf
```
may require information that can only be resolved later.

Therefore:



This is one of the most important concepts students should learn.

67. Why `as` Is Important for Engineering Students

Learning `as` gives students direct exposure to the lower levels of software execution.

It connects:

Programming

```
C/C++
   ↓
Assembly
```

Computer Architecture

```
Assembly
   ↓
Instruction Set
   ↓
Machine Code
```

Compiler Design

```
Compiler output
    ↓
Assembly
    ↓
Assembler
```

System Programming

```
Object files
Symbols
Relocations
ELF
```

Operating Systems

```
System calls
Startup code
Kernel interfaces
```

Embedded Systems

```
C
  ↓
Assembly
  ↓
Machine code
  ↓
Microcontroller / CPU
```

---

68. Recommended Position in a Linux Engineering Curriculum

If students are learning Linux command-line and system-programming tools, a very effective sequence is:

```
 1. Linux Shell
       ↓
 2. gcc
       ↓
 3. gcc -S
       ↓
 4. as
       ↓
 5. Object Files
       ↓
 6. nm
       ↓
 7. ar
       ↓
 8. ld
       ↓
 9. readelf
       ↓
10. objdump
       ↓
11. ldd
       ↓
12. gdb
```

This creates a coherent progression:

```
Source Code
    ↓
Compilation
    ↓
Assembly
    ↓
Machine Code
    ↓
Object File
    ↓
Symbols
    ↓
Libraries
    ↓
Linking
    ↓
ELF Executable
    ↓
Disassembly
    ↓
Runtime Dependencies
    ↓
Debugging
```

69. One-Page as Cheat Sheet

```
# Check assembler
which as

# Version
as --version

# Help
as --help

# Assemble source
as program.s -o program.o

# Assemble with debug information
as -g program.s -o program.o

# Generate listing
as -al program.s -o program.o

# Add include directory
as -I/path/to/include program.s -o program.o

# Inspect resulting object
file program.o

# Display sections
objdump -h program.o

# Display symbols
nm program.o

# Display relocations
objdump -r program.o

# Disassemble
objdump -d program.o

# Intel-style disassembly
objdump -d -M intel program.o

# Display ELF information
readelf -h program.o
readelf -S program.o
readelf -s program.o
```

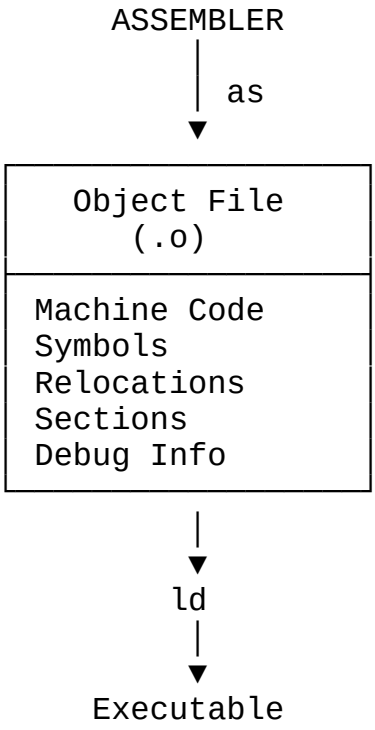
```
# Link object
ld program.o -o program

# Or link through GCC
gcc program.o -o program
```

---

70. The Key Concept

The most important thing for students to remember is:



---

as is the assembler that transforms human-readable assembly instructions into machine-code object files, while preserving the symbols, sections, and relocation information required by the linker.

For an engineering curriculum, **as** is particularly valuable because it provides the missing link between compiler-generated assembly and the actual machine code executed by the processor. It fits naturally alongside **gcc, nm, ar, ld, readelf, objdump, ldd**, and **gdb** as a practical Linux systems-programming toolchain.

7. gdb

`gdb` (GNU Debugger) is a command-line debugging tool used to find and fix errors (bugs) in programs. It allows you to run a program under controlled conditions, pause execution, inspect variables, examine memory, and step through the source code line by line.

It is one of the most widely used debugging tools for programs written in **C, C++, Fortran, Ada, Go, Rust**, and several other languages.

Syntax

```
gdb [options] program
```

or

```
gdb program corefile
```

or

```
gdb --args program arguments
```

Purpose

- Debug program crashes
- Find segmentation faults
- Trace program execution
- Inspect variable values
- Set breakpoints
- Step through code
- Examine memory and registers
- Analyze core dump files

Basic Example

Suppose you have a C program:

```
#include <stdio.h>

int main() {
    int x = 5;
    int y = 0;

    printf("%d\n", x / y);

    return 0;
}
```

Compile with debugging information:

```
gcc -g test.c -o test
```

Start GDB:

```
gdb ./test
```

Run the program:

```
(gdb) run
```

If the program crashes, GDB shows the exact line where it happened.

Common Options

Option	Description
`-q`	Quiet mode (don't print startup messages)
`-tui`	Text User Interface (source code window)
`--args`	Pass command-line arguments to the program
`-batch`	Run commands non-interactively
`-ex command`	Execute a command on startup
`-x file`	Execute commands from a file

Example:

```
gdb -q ./program
```

Frequently Used GDB Commands

Command	Description
`run`	Start the program
`start`	Run until main()
`break`	Set breakpoint
`continue`	Continue execution
`next`	Execute next line (don't enter functions)
`step`	Execute next line (enter functions)
`finish`	Finish current function
`list`	Show source code
`print`	Display variable value
`display`	Automatically display a variable after each stop
`watch`	Stop when a variable changes
`backtrace` (`bt`)	Show function call stack
`frame`	Select a stack frame
`info locals`	Show local variables
`info args`	Show function arguments
`info breakpoints`	List breakpoints
`delete`	Delete breakpoints
`quit`	Exit GDB

Example 1: Set a Breakpoint

```
(gdb) break main
```

or

```
(gdb) break 25
```

Break at line 25.

Example 2: Run Program

```
(gdb) run
```

If the program needs arguments:

```
(gdb) run input.txt output.txt
```

Example 3: Step Through Code

Step into functions:

```
(gdb) step
```

Execute current line only:

```
(gdb) next
```

Example 4: Print Variable

```
(gdb) print count
```

Output:

```
$1 = 42
```

Example 5: Watch Variable

```
(gdb) watch counter
```

Execution stops whenever `counter` changes.

Example 6: Display Variables Continuously

```
(gdb) display x
```

Each step automatically shows:

```
1: x = 15
```

Example 7: View Backtrace

After a crash:

```
(gdb) backtrace
```

or

```
(gdb) bt
```

Example:

```
#0 divide()
#1 calculate()
#2 main()
```

Example 8: Inspect Memory

Display memory in hexadecimal:

```
(gdb) x/16x address
```

Display as characters:

```
(gdb) x/20c address
```

Display as strings:

```
(gdb) x/s address
```

Example 9: Examine Registers

```
(gdb) info registers
```

Useful for low-level debugging.

Example 10: Debug a Core Dump

If a program generated a core file:

```
gdb ./program core
```

View the crash location:

```
(gdb) bt
```

Example 11: Debug a Running Process

Find the process ID:

```
ps aux | grep program
```

Attach GDB:

```
gdb -p 12345
```

Detach:

```
detach
```

Example 12: Pass Arguments

```
gdb --args ./program file1.txt file2.txt
```

Inside GDB:

```
run
```

Example 13: Using TUI Mode

```
gdb -tui ./program
```

Displays source code, registers, and command prompt in a split-screen terminal interface.

Compile for Debugging

Always compile with debugging symbols:

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

For easier debugging (disable optimizations):

```
gcc -g -O0 program.c -o program
```

Advantages

- Powerful interactive debugger
- Free and open source
- Supports multiple programming languages
- Debugs local and remote programs
- Supports multi-threaded applications
- Can analyze core dumps
- Extensive scripting support (Python)

Limitations

- Command-line interface can be challenging for beginners
- Optimized builds (`-O2`, `-O3`) can make debugging harder because variables may be optimized away or code reordered
- Requires debug symbols (`-g`) for the best debugging experience

Typical Workflow

```
# Compile with debug information

gcc -g -O0 hello.c -o hello

# Start debugger

gdb ./hello

# Inside GDB

(gdb) break main
(gdb) run
(gdb) next
(gdb) print variable
(gdb) continue
(gdb) quit
```

Summary

gdb is the standard GNU debugger used to diagnose and fix bugs in programs. It lets you pause execution, inspect variables, step through source code, examine memory and registers, debug crashes, attach to running processes, and analyze core dump files, making it an essential tool for software development and troubleshooting.

8. git

`git` is a **distributed version control system (DVCS)** used to track changes in source code and other files. It allows developers to manage project history, collaborate with others, create branches for new features, and merge changes safely.

It is one of the most widely used tools in software development.

Syntax

```
git <command> [options]
```

Examples:

```
git status
git add file.txt
git commit -m "Initial commit"
git push origin main
```

What Git Does

Git helps you:

- Track every change made to files
- Restore previous versions
- Work on multiple features using branche
- Collaborate with multiple developer
- Merge changes from different people
- Maintain a complete project history

How Git Works

A Git project consists of three main areas:

```
Working Directory
        │
        ▼
Staging Area (Index)
        │
        ▼
Git Repository (.git)
```

1. Working Directory

The files you are currently editing.

Example:

```
main.c
README.md
style.css
```

---

2. Staging Area

Files prepared for the next commit.

Example:

```
git add main.c
```

---

3. Repository

The permanent history of commits.

Example:

```
git commit -m "Added login feature"
```

Basic Workflow

```
Create/Edit File
        |
        ▼
git add
        |
        ▼
git commit
        |
        ▼
git push (optional)
```

Common Git Commands

Command	Purpose
`git init`	Create a new Git repository
`git clone`	Copy an existing repository
`git status`	Show repository status
`git add`	Stage files
`git commit`	Save staged changes
`git log`	Show commit history
`git diff`	Show file differences
`git branch`	Manage branches
`git checkout`	Switch branches or restore files (older workflow)
`git switch`	Switch branches (newer command)
`git restore`	Restore files
`git merge`	Merge branches
`git rebase`	Reapply commits onto another branch
`git fetch`	Download updates from a remote
`git pull`	Fetch and merge updates
`git push`	Upload commits to a remote repository
`git remote`	Manage remote repositories
`git tag`	Create version tags

Frequently Used Commands

Initialize a repository

```
git init
```

Creates a hidden `.git` directory.

Check repository status

```
git status
```

Shows:

- Modified files
- Untracked files
- Staged files
- Current branch

Example:

```
On branch main

Modified:
    app.py

Untracked:
    notes.txt
```

Add files

Single file:

```
git add main.c
```

All files:

```
git add .
```

Commit changes

```
git commit -m "Fixed login bug"
```

Creates a snapshot of the staged changes.

View commit history

```
git log
```

Compact version:

```
git log --oneline
```

Example:

```
5f6d7a2 Fixed login bug
e21ab19 Added authentication
c42d110 Initial commit
```

Compare changes

```
git diff
```

Shows modifications that haven't been staged.

Create a branch

```
git branch feature-login
```

Switch branches

```
git switch feature-login
```

Older equivalent:

```
git checkout feature-login
```

Merge a branch

```
git merge feature-login
```

Clone a repository

```
git clone https://example.com/project.git
```

Downloads a repository to your computer.

Download updates

```
git pull
```

Upload changes

```
git push
```

Example Workflow

Create a repository:

```
mkdir project
cd project
git init
```

Create a file:

```
echo "Hello Git" > hello.txt
```

Check status:

```
git status
```

Stage the file:

```
git add hello.txt
```

Commit:

```
git commit -m "Added hello.txt"
```

View history:

```
git log –oneline
```

Output:

```
8f3ab21 Added hello.txt
```

Common Options

Option	Description
`--help`	Show help for a command
`--version`	Display the installed Git version
`-m`	Provide a commit message
`--oneline`	Compact commit history
`-a`	Automatically stage tracked file changes before committing
`-v`	Verbose output

Example:

```
git --version
```

Output:

```
git version 2.45.1
```

Advantages of Git

- Fast and efficient
- Complete project history
- Supports offline work
- Easy branching and merging
- Widely used in open-source and enterprise projects
- Excellent collaboration support
- Helps recover previous versions of files

Limitations

- Has a learning curve, especially for branching and rebasing
- Merge conflicts may require manual resolution
- Large binary files are not handled as efficiently as text files

When to Use Git

Use Git when you need to:

- Develop software projects
- Track changes to documents or code
- Collaborate with a team
- Maintain version history
- Experiment with new features without affecting the main codebase
- Roll back to earlier versions if needed

Summary

- **Command:** `git`
- **Purpose:** Manage version control for files and projects.
- **Type:** Distributed Version Control System (DVCS).
- **Key Features:** Change tracking, commits, branching, merging, collaboration, and history management.
- **Common Workflow:** Edit files → `git add` → `git commit` → `git push` (if using a remote repository).