How GCC Works

A source-level guide to the GNU Compiler Collection, for programmers who have used GCC for years and never read it.

Fadli

Contents

1 Introduction

You have used GCC for years. You know what `-O2` does, roughly. You have fought `cannot find crt1.o` at two in the morning. You have built a cross toolchain by copying somebody's shell script and changing the triple, and it worked, and you never found out why.

This book is about what is actually in there.

It is not a manual. GCC already has excellent manuals — several of them — and where they cover a mechanism this book tells you which node to read. It is also not a wiki: there is an order to the chapters, each one assumes the ones before it, and each one is about a single mechanism named in its title.

What the book does that the manuals do not is show you the code. Every concrete claim here carries a link to the line of GCC source that proves it, so that when you disagree with a sentence you are one click from settling the argument. GCC's own comments are unusually good, and where a comment explains a decision better than prose could, the comment is quoted rather than paraphrased.

1.1 Who this is for

Someone comfortable with C, ELF, `readelf`, and the general shape of a linker. You do not need to have read a compiler book. You do not need to know what SSA is; when it matters, you will be told.

You do not need to build GCC to use most of this book. Nearly every mechanism described has a command that lets you watch it happen on a compiler you already have installed — `-###`, `-v`, `-print-search-dirs`, `-dumpspecs`, `-fdump-tree-all`. Chapters that describe something you cannot observe from outside say so explicitly.

1.2 How the source is cited

Every source reference in this book points at an **immutable upstream release tag** on the official read-only mirror at `github.com/gcc-mirror/gcc`. At the time of writing that tag is `releases/gcc-15.2.0`, and every line number was read out of that tag rather than out of anybody's working tree.

Each chapter opens with a note naming its pin and ends with a reminder of it. That matters because line numbers rot: the code around them almost never does, but the numbers themselves move with every patch. If you are reading a GCC 12 tree, the file names alone will differ — see below.

When a mechanism genuinely *changed* between major releases, rather than merely shifting a few lines down, that is flagged inline, at the point of the claim, so a reader on an older compiler is warned by the sentence they are reading rather than by an appendix they will never reach.

> **If you are on GCC 11 or earlier, every filename in this book is wrong.** GCC 12 renamed nearly every `.c` file in the compiler to `.cc` in a single commit, `5c69acb3` (Martin Liska, 2022-01-14). So `gcc.c` became `gcc.cc`, `gimplify.c` became `gimplify.cc`, `aarch64.c` became `aarch64.cc`. Every pre-12 blog post, mailing-list thread and Stack Overflow answer on the internet uses the old names. Mentally add or drop the second `c` as needed.

1.3 The shape of the book

Part I, "The driver is not the compiler", is about the program you actually type. `gcc` compiles nothing at all; it is a program whose entire job is to decide which other programs to run and to build each one's command line. That single fact explains most of what looks arbitrary about GCC from outside, and Part I is the longest part of the book because almost every path, prefix, library and header decision lives there.

Part II is about configuring and building GCC itself, and **Part III** about the specific problem of bootstrapping a cross toolchain, where the compiler and the C library each need the other to exist first.

Part IV goes inside `cc1` — the three intermediate representations, the pass manager, and how a machine description becomes an instruction. **Part V** covers the runtime libraries GCC ships. **Part VI** is about working on GCC as a project: the testsuite, the conventions, and how a patch gets proposed.

1.4 Four things to keep straight from the start

These recur in every part and they are where most published explanations of GCC go wrong.

The driver never opens a library file. It emits `-l` and `-L` and lets `ld` resolve them. A "library search path" in GCC's sense is a list of strings the driver hands to the linker, nothing more.

Link-time search and run-time search are unrelated systems. A successful link says nothing whatever about whether the resulting binary will start. That is `ld.so`'s business — `DT_RUNPATH`, `ld.so.conf`, `LD_LIBRARY_PATH` — and no chapter in Part I touches it.

Header search and library search are separate mechanisms. They happen to be relocated by the same prefix, which makes them look like one thing. They are implemented in different files by different code with different sysroot rules.

Anything under `libexec/gcc/` is a host program; anything under `<target>/lib/` is a target artifact. A host program runs on your machine. A target artifact never runs at all on your machine; it only gets linked into what your compiler produces. Blurring these two makes cross-compilation incoherent, and Chapter 1.2 is largely about keeping them apart.

2 What actually runs when you type `gcc foo.c`

Source references in this chapter are to GCC 15.2.0 (`releases/gcc-15.2.0`).

You type `gcc hello.c`, and an `a.out` appears. Somewhere in between, your C was turned into machine code. The obvious assumption is that `gcc` did it.

It did not. **`gcc` compiles nothing.** It is a *driver*: a program whose entire job is to work out which other programs to run, build each one's command line, and execute them in order. Almost every "GCC can't find X" problem is a driver problem, and the driver will tell you exactly what it did if you ask it.

2.1 Watching it happen

Start with the one flag worth learning before any other:

```
$ gcc -### hello.c
```

`-###` prints every command line the driver *would* run, fully expanded and shell-quoted, and then runs none of them. Its sibling `-v` prints the same command lines and does execute them. If you only ever remember one thing from this book, remember `-###`: it turns "GCC is doing something weird" into a concrete argument list you can read, diff against a working toolchain, and paste back into a shell.

What you will see is three or four separate programs, not one:

```
gcc hello.c
  │
  ├─► cc1        hello.c  →  /tmp/ccXXXXXX.s     the actual C compiler
  │                                              (cc1plus for C++, lto1 for LTO)
  ├─► as         .s       →  hello.o             the assembler, from binutils
  │
  └─► collect2   hello.o  →  a.out               a wrapper that runs ld
        └─► ld
```

Every arrow there is a real `fork` and `exec`. The temporary `.s` file in the middle is real too, and you can see the driver invent its name:

```
$ gcc -### hello.c 2>&1 | grep -o '/tmp/cc[A-Za-z0-9]*\.s'
```

Nobody named that file. Two separate programs — `cc1` and `as` — nevertheless agree on it, because the driver generated one name and substituted it into both command lines. You will meet the mechanism that does this in Chapter 1.4; for now, note only that the intermediate files between stages are the driver's bookkeeping, not yours.

You can stop the chain early at each boundary, which is a useful way to convince yourself the stages are really separate:

Flag	Stops after
`-E`	the preprocessor
`-S`	`cc1`, leaving a `.s`
`-c`	`as`, leaving a `.o`
(none)	the link

2.2 Where those programs live

The driver is not searching your `$PATH` for `cc1`. It could not: `cc1` is not on your `$PATH`, and deliberately so.

Program	What it is	Installed in
`gcc`, `g++`	the driver	`$prefix/bin/`
`cc1`, `cc1plus`, `lto1`	the real compilers	`$prefix/libexec/gcc/<target>/<version>/`
`collect2`, `lto-wrapper`	GCC's own link-time helpers	same `libexec` directory
`as`, `ld`	binutils	binutils' install, or the tool directory

All four rows are **host** programs — they run on your machine, as processes. That is the point of `libexec`: those are private implementation binaries that happen to need to be `exec`'d, and versioning them by target and release is what lets several GCCs coexist.

The driver finds them through a dedicated search list, distinct from the one it uses for libraries. Both are printed by one command:

```
$ gcc -print-search-dirs
```

The `programs:` line is the executable list; the `libraries:` line is the one that resolves `crt1.o` and `-lfoo`. They are separate lists built by separate code, and Chapter 1.3 shows you how to read them. Do not go looking for a third line for headers — there isn't one, for a reason covered in that chapter.

2.3 The driver has no command lines in it

Here is the part that is genuinely surprising: the argument lists you just looked at are not written in C anywhere. The driver holds *templates* — strings in a small `%`-escape language — and expands them at run time.

Listing 1-1 is the real entry that handles a `.c` file. This is the whole of GCC's knowledge about how to compile C, and it is a string constant.

```
  {".c", "@c", 0, 0, 1},
  {"@c",
   /* cc1 has an integrated ISO C preprocessor.  We should invoke the
      external preprocessor if -save-temps is given.  */
     "%{E|M|MM:%(trad_capable_cpp) %(cpp_options) %(cpp_debug_options)}\
      %{!E:%{!M:%{!MM:\
      ...
      %{!save-temps*:%{!traditional-cpp:%{!no-integrated-cpp:\
      cc1 %(cpp_unique_options) %(cc1_options)}}}\
      %{!fsyntax-only:%(invoke_as)}}}}", 0, 0, 1},
```

Listing 1-1: the `.c` entry in `default_compilers[]` (`gcc/gcc.cc:1454-1468`). Abridged; the omitted middle handles `-save-temps` and rejects `-traditional`.

Two things fall out of that string immediately.

The literal word `cc1` appears in it. That is how the driver knows what to run: not from a variable, but from the first word of an expanded template. And `%{!fsyntax-only:%(invoke_as)}` is why the assembler runs at all. `invoke_as` is another template:

```
static const char *invoke_as =
"%{!fwpa*:\
   %{fcompare-debug=*|fdump-final-insns=*:%:compare-debug-dump-opt()}\
   %{!S:-o %|.s |\n as %(asm_options) %m.s %A }\
  }";
```

Listing 1-2: `invoke_as` (`gcc/gcc.cc:1313-1322`).

Look at `|\n as`. A newline inside a spec template means *"that was the end of one program; what follows is another program to run."* The pipe before it means what a pipe usually means. So the "chain" in the diagram at the top of this chapter is not implemented by a loop over stages in the driver's C code — it is implemented by a newline in a string.

Three consequences matter enough to state now, even though the language itself waits until Chapter 1.4:

1. **A target customises GCC by replacing a template, not by patching the driver.** Two GCCs built from identical source for different targets can behave completely differently, because their target headers `#define` different templates.
2. **`-nostdlib`, `-nostartfiles` and `-nodefaultlibs` are not implemented in C at all.** They are conditionals inside one template string. That is exactly why they compose the way they do.
3. **You can replace those templates yourself**, per-invocation with `-specs=FILE`, without rebuilding anything.

> There is no `gcc/specs.cc`, and there never has been. The spec language lives entirely inside `gcc/gcc.cc`, where the interpreter is a function called `do_spec_1`. `gcc/specs.cc` nevertheless appears in a great deal of secondhand documentation. If you go looking for it you will waste an afternoon.

2.4 Why `collect2` and not `ld`

You might have expected the last program in the chain to be `ld`. On most targets it isn't; it is a GCC program called `collect2`, which then runs `ld`.

The default is set by one macro:

```
#ifndef LINKER_NAME
#define LINKER_NAME "collect2"
#endif
```

Listing 1-3: `LINKER_NAME` (`gcc/gcc.cc:902-904`).

`collect2`'s own opening comment says what it is for: *"Collect static initialization info into data structures that can be traversed by C++ initialization and finalization routines"* (`gcc/collect2.cc:1-2`). On platforms whose object format has no native mechanism for running constructors before `main`, somebody has to scan the objects, build a table of constructor and destructor pointers, and get that table into the image. `collect2` does that scan, generates a small C file containing the tables, and compiles it.

Which means `collect2` needs a compiler. So does `lto-wrapper`, GCC's other link-time helper, which re-runs code generation at link time for link-time optimisation and therefore needs both the compiler *and* your original options.

This is the recursion that trips people up. If either helper grabbed whatever `gcc` happened to be first on `$PATH`, the second pass would run with a different prefix, a different sysroot and different search paths than your original command — and on a cross toolchain, quite possibly a different *target*.

The driver prevents that by putting the answers in the environment before it spawns anything. `COLLECT_GCC` holds the full pathname of the running driver, taken from `argv[0]`:

```
  obstack_grow (&collect_obstack, "COLLECT_GCC=", sizeof ("COLLECT_GCC=") - 1);
```

Listing 1-4: exporting `COLLECT_GCC` (`gcc/gcc.cc:8718`).

and `COLLECT_GCC_OPTIONS` holds every switch, each individually single-quoted, so the recursion reconstructs the same paths. This is the whole reason `-B`, `--sysroot` and `-L` survive into the link step. It is also why `gcc -v` output is littered with `COLLECT_*` lines:

```
$ gcc -v hello.c 2>&1 | grep '^COLLECT_'
```

Those lines are already shell-quoted, so they paste straight into a terminal. They are the fastest way to reproduce exactly what `collect2` was handed.

The failure mode is worth knowing. `lto-wrapper` refuses to run without them:

```
  collect_gcc = getenv ("COLLECT_GCC");
  if (!collect_gcc)
    fatal_error (input_location,
         "environment variable %<COLLECT_GCC%> must be set");
```

Listing 1-5: `lto-wrapper` hard-errors (`gcc/lto-wrapper.cc:1444-1448`).

`collect2` is more forgiving, and that is the dangerous half:

```
  /* Determine the full path name of the C compiler to use.  */
  c_file_name = getenv ("COLLECT_GCC");
  if (c_file_name == 0)
    {
#ifdef CROSS_DIRECTORY_STRUCTURE
      c_file_name = concat (target_machine, "-gcc", NULL);
#else
      c_file_name = "gcc";
#endif
    }
```

Listing 1-6: `collect2`'s fallback (`gcc/collect2.cc:1142-1151`).

A `collect2` invoked without `COLLECT_GCC` will silently fall back to `<target>-gcc`, and then to plain `gcc`. On a cross toolchain that can pick up a *different* compiler with different startfile and sysroot paths, and the only symptom is a link that mysteriously resolves the wrong `crtbegin.o`. Chapter 1.6 covers the whole environment channel.

Two smaller details, both of which surprise people:

The driver will fall back to `ld` if `collect2` is missing, quietly:

```
      /* We'll use ld if we can't find collect2.  */
      if (! strcmp (linker_name_spec, "collect2"))
        {
          char *s = find_a_program ("collect2");
          if (s == NULL)
        set_static_spec_shared (&linker_name_spec, "ld");
        }
```

Listing 1-7: the `collect2`-to-`ld` fallback (`gcc/gcc.cc:9243-9249`).

And when you grep `-###` output for the link line, grep for `collect2`, not for `ld` — on a normal Linux toolchain `ld` never appears on the driver's output at all, because it is `collect2` that runs it.

2.5 `gcc` and `g++` are the same program

They are not two compilers. They are the same driver binary's worth of code with one extra hook compiled in, and the hook runs before anything else has been acted upon:

```
  /* Do language-specific adjustment/addition of flags.  */
  lang_specific_driver (&decoded_options, &decoded_options_count,
            &added_libraries);
```

Listing 1-8: the language hook (`gcc/gcc.cc:4860-4862`).

At that point the command line has been decoded into an array of options but nothing has been done with it, and the hook is free to insert, delete and reorder entries. `g++`'s implementation does exactly two things: it makes `.c` files compile as C++, and it appends the C++ runtime libraries to your link — while silently moving your own `-lm` and `-lc` in the process. That is the entire practical difference between the two commands at link time. Link C++ objects with `gcc` and you get undefined `std::` symbols; nothing else changes.

Chapter 1.11 takes that hook apart, because the details are surprising and are documented nowhere in the manuals.

2.6 A debugging loop that actually works

When something links wrong, run these in order and stop at the first surprise.

```
$ gcc -### foo.c                     # is the flag you passed even reaching the linker?
$ gcc -print-search-dirs             # is the directory you expect in `libraries:`?
$ gcc -print-file-name=libfoo.a      # which copy of it wins?
```

```
$ gcc -print-sysroot                  # is a sysroot in play at all?
$ gcc -print-multi-directory          # are you getting the right ABI variant?
```

The third one is the honest one: `-print-file-name` walks the same list the linker is given, so its answer is exactly what the link would pick — and when the file is not found at all, it echoes your argument straight back. That single behaviour is the fastest diagnosis in this book, and Chapter 1.3 explains why it works that way.

2.7 What to take from this

`gcc` is a program that builds command lines. Everything in Part I follows from that: the search paths exist so the driver can fill in filenames, the spec language exists so targets can rewrite the command lines, and the environment variables exist so the driver's own recursive invocations agree with it.

The manuals cover the observation tools well — `-###`, `-v` and the `-print-*` family are all in `invoke.texi` under Overall Options (`gcc/doc/invoke.texi:1549`), and the spec language has its own node, Spec Files (`:37290`). What they do not describe is the `COLLECT_GCC` channel from the driver's side, or `collect2`'s fallback when it is missing. For those, the source above is the documentation.

Next: the vocabulary the rest of Part I is written in — three machine names, and the difference between a program that runs and an object that only ever gets linked.

All source references in this chapter are to **GCC 15.2.0** (`releases/gcc-15.2.0`). Line numbers in other releases will differ; the surrounding code rarely does. Where behaviour itself changed across a major version, it is flagged inline.

3 Three machines, two worlds

Source references in this chapter are to GCC 15.2.0 (`releases/gcc-15.2.0`).

You have an ARM cross toolchain. You point `--sysroot` at your root filesystem and `crt1.o` is found. You point it somewhere else and `crtbegin.o` is still found, from the old place, and you cannot see why one moved and the other didn't.

The reason is that a GCC installation contains artifacts belonging to **two different machines at once**, and every configure flag and every command-line option belongs to one world or the other. Once you can look at a path and say which world it is in, most of GCC's path behaviour stops being arbitrary.

This chapter is the vocabulary the rest of Part I is written in. It is short, and everything after it depends on it.

3.1 Three machine names

Almost every autoconf package has two machine names. GCC has three, because a compiler is the one program whose *output* is for a machine other than the one it runs on. The manual defines them in one sentence:

> The **build** machine is the system which you are using, the **host** machine is the system where you want to run the resulting compiler (normally the build machine), and the **target** machine is the system for which you want the compiler to generate code.
>
> — `gcc/doc/install.texi:841-844`

Flavour	build	host	target
native	`x86_64-linux-gnu`	same	same
cross	`x86_64-linux-gnu`	same	`arm-linux-gnueabihf`
Canadian cross	`x86_64-linux-gnu`	`x86_64-w64-mingw32`	`arm-none-eabi`

The practical consequence is the one to hold on to: **a single `make` produces artifacts for two different machines at once.** Almost everything confusing about GCC's configure flags comes from that one fact, and the useful question to ask of any flag is always "is this about the machine running the compiler, or the machine running its output?"

You can ask a built compiler which target it is for:

```
$ arm-linux-gnueabihf-gcc -dumpmachine
```

That prints the target triple and exits — the quickest check that you are running the compiler you think you are.

3.2 Host code and target code

Now the split that matters when you are reading paths.

	Host code	Target code
Examples	`gcc`, `cc1`, `cc1plus`, `lto1`, `collect2`, `lto-wrapper`	`libgcc.a`, `libstdc++.so`, `crt1.o`, `crtbegin.o`
Runs	on your machine, as a process	never — it is *linked into* what your compiler produces
Built by	the build machine's system compiler	the compiler this build just produced
Lives in	`bin/`, `libexec/`	`lib/gcc/<t>/<v>/`, `<t>/lib/`, and the sysroot

Target code is the part people misjudge. `libgcc.a` is not a library GCC uses; it is a library GCC *emits calls into*, compiled for a machine that may not be able to run a single instruction of the compiler that built it. When a cross toolchain "finds the wrong `libgcc.a`", nothing has gone wrong with GCC's own installation — it has picked the wrong artifact for a foreign machine.

The clean rule, which holds without exception:

> `libexec/` and `bin/` are **host**. `<target>/lib/` and the sysroot are **target**. `lib/gcc/<target>/<version>/` is a **mix** — host data such as the `specs` file and GCC's own headers, alongside target objects such as `libgcc.a` and `crtbegin.o`.

Most flags move one world or the other. `-B` is the notable exception: it is a single prefix applied to three separate search lists at once, and those lists span both worlds. That is why `-B` feels blunt, and it is covered in Chapter 1.5.

3.3 What a sysroot actually is

A sysroot is a directory that **stands in for `/` on the target machine**. Inside it, `usr/include/stdio.h` plays the role that `/usr/include/stdio.h` plays on the target.

It has to exist because your own `/usr/include/stdio.h` describes *your* libc: wrong `sizeof(long)`, wrong struct layouts, wrong syscall numbers, wrong `#define`s. Reading it while cross-compiling would be silently catastrophic — you would get an object file that compiles, links, and misbehaves on the device. So the compiler needs a second, target-flavoured `/`. That is all a sysroot is.

Mechanically it is unglamorous. It is a string glued onto the front of certain absolute paths, by a function whose entire job is that gluing:

```
add_sysrooted_prefix (struct path_prefix *pprefix, const char *prefix,
              const char *component,
              /* enum prefix_priority */ int priority,
              int require_machine_suffix, int os_multilib)
{
  if (!IS_ABSOLUTE_PATH (prefix))
    fatal_error (input_location, "system path %qs is not absolute", prefix);

  if (target_system_root)
    {
      ...
      prefix = concat (sysroot_no_trailing_dir_separator, prefix, NULL);
```

Listing 2-1: `add_sysrooted_prefix` (`gcc/gcc.cc:3177-3200`). Abridged; the omitted lines strip a trailing separator and splice in a per-multilib sysroot suffix.

The important thing about Listing 2-1 is what it implies about everything else: there is a sibling function, `add_prefix` (`gcc/gcc.cc:3144`), which does the same job *without* the sysroot. Whether a given directory follows your `--sysroot` comes down to which of the two functions registered it. That is the whole mechanism. There is no post-processing pass that rewrites paths, and no flag that can retroactively sysroot a prefix that was registered with `add_prefix`.

3.4 The rule that explains most surprising path results

> **The sysroot belongs to the C library. The prefix belongs to GCC.**
>
> GCC never installs anything into the sysroot, and the C library never installs anything into the prefix.

Almost every "why didn't `--sysroot` move that?" has the same answer: because that thing is GCC's, not libc's. Sorting the pieces of a normal link by owner:

The C library's, in the sysroot — this is the half a sysroot moves:

- `<stdio.h>`, `<pthread.h>`, `<fcntl.h>` and the rest of the system headers
- `libc.so`, `libm.so`
- `crt1.o`, `crti.o`, `crtn.o`

GCC's, under the prefix — a sysroot never touches these:

- `libgcc.a`, `libgcc_eh.a`, `libgcc_s.so.1`
- `crtbegin.o`, `crtend.o` (they are part of libgcc)

- `libstdc++` and its headers, which live in the tool directory
- `<stddef.h>`, `<stdint.h>`, `<limits.h>`, `<stdarg.h>` — surprisingly, GCC's; the C standard requires the *compiler* to supply these, and Chapter 1.9 is about why

That last group is the one that catches everyone. You will not find `stdint.h` in your sysroot, you are not supposed to, and copying a sysroot from one machine to another does not bring it along. The compiler's own headers travel with the *compiler*.

3.5 The directory map

Three autoconf variables underlie every path formula in GCC:

Variable	What it is	Default
`$prefix`	root of the entire install	`--prefix=`, else `/usr/local`
`$exec_prefix`	root for machine-dependent files	`--exec-prefix=`, else `$prefix`
`$libdir`	host libraries	`$exec_prefix/lib`

`$prefix` and `$exec_prefix` are *different variables that merely default to the same value* — `gcc/Makefile.in:662` and `:673`. Pass `--exec-prefix` and paths built from one diverge from paths built from the other. This is not hypothetical: libstdc++ installs its headers under `${prefix}` and its libraries under `${exec_prefix}`, so passing `--exec-prefix` puts the C++ headers and the C++ library in different trees. That is a Chapter 1.12 problem, but the cause is here.

Then the directories themselves. Two have names you will meet constantly:

libsubdir — GCC's private per-version directory:

```
libsubdir = $(libdir)/gcc/$(real_target_noncanonical)/$(version)$(accel_dir_suffix)
```

Listing 2-2: `libsubdir` (`gcc/Makefile.in:682`).

That expands to something like `/opt/gcc-arm/lib/gcc/arm-linux-gnueabihf/15.2.0/`, and it is where `specs`, GCC's own `include/`, `libgcc.a` and `crtbegin.o` live. Note the `$(version)` component: two GCC releases have entirely separate libsubdirs, which is what lets them coexist and what makes mixing GCC 12's headers with GCC 15's `cc1` a broken toolchain rather than a merely unwise one.

tooldir — the target's own subtree, defined at the top level:

```
tooldir='${exec_prefix}'/${target_noncanonical}
```

Listing 2-3: `tooldir` (`configure.ac:3121`).

That is `/opt/gcc-arm/arm-linux-gnueabihf/`, and inside it are `bin/`, `lib/`, `include/` and `sys-include/` — a little pretend `/usr` for the target. A cross libstdc++ installs there, and so does newlib.

Rounding it out:

Name	Default	Contains	Sysrooted?
prefix	`--prefix`	everything GCC installs	no
libsubdir	`$libdir/gcc/<target>/<ver>/`	GCC's target objects **and** host data	no
tooldir	`$exec_prefix/<target>/`	`bin/ lib/ include/ sys-include/`	no
toolexeclibdir	`$tooldir/lib` (cross), `$libdir` (native)	target libraries such as `libstdc++.so`	no
sysroot	`--with-sysroot`, else `$tooldir/sys-root`	the C library's entire world	it *is* the sysroot

The manual documents `--with-toolexeclibdir`'s default as `${gcc_tooldir}/lib` (`install.texi:2705-2707`), and it is worth noticing that the bare `--with-sysroot` default sits *inside* the tooldir, hence inside `$exec_prefix`. That is exactly the condition under which a sysroot relocates with a moved toolchain, and the manual says so:

> If the specified directory is a subdirectory of `${exec_prefix}`, then it will be found relative to the GCC binaries if the installation tree is moved.
>
> — `gcc/doc/install.texi:2721-2723`

The code that implements that sentence is guarded by a macro that configure only defines when the condition holds (`gcc/configure.ac:176-182`, acted on at `gcc/gcc.cc:5542-5559`). It is covered in Chapter 1.5.

3.6 What a real install looks like

Under `--prefix=/opt/gcc-arm --target=arm-linux-gnueabihf`:

Path	Contains	World
`/opt/gcc-arm/bin/`	`arm-linux-gnueabihf-gcc`, `-g++`, `gcc-ar`	host
`/opt/gcc-arm/libexec/gcc/<target>/<ver>/`	`cc1`, `cc1plus`, `lto1`, `collect2`	host
`/opt/gcc-arm/lib/gcc/<target>/<ver>/`	`specs`, `include/`, `crtbegin*.o`, `libgcc.a`	mixed
`/opt/gcc-arm/<target>/lib/`	`libstdc++.so`, newlib's `libc.a`	target
`/opt/gcc-arm/share/`	docs, man pages	host data
the sysroot, wherever it is	`libc.so`, `crt1.o`, `<stdio.h>`	target

Check your own understanding against a real compiler:

```
$ arm-linux-gnueabihf-gcc -dumpmachine       # the target triple
$ arm-linux-gnueabihf-gcc -print-sysroot     # the C library's world; empty = none
$ arm-linux-gnueabihf-gcc -print-search-dirs # install root, programs, target libraries
$ arm-linux-gnueabihf-gcc -v                 # the configure line it was built with
```

An empty `-print-sysroot` is meaningful rather than broken, and the next chapter explains what it implies.

3.7 Why this makes cross-toolchain builds circular

One more consequence, because it drives all of Part III.

`libgcc` and `libstdc++` are **target libraries that GCC itself ships**. That is exactly why they sit awkwardly between the two worlds, and it creates a dependency cycle:

1. Target libraries need a target compiler. Fine — the build makes one first, an uninstalled `xgcc`.
2. But libgcc's own *source* includes libc headers. The manual is blunt about it: *"When crossing to GNU/Linux, you need the headers so GCC can build the exception handling for libgcc."* (`install.texi:2765-2766`)
3. And libstdc++ needs a libc it can actually **link** against, not merely headers — its configure runs hundreds of link probes.
4. But the libc was itself built by a compiler.

The conventional way out is to build a deliberately crippled GCC first:

```
binutils → libc headers → minimal GCC (--without-headers)
         → full libc    → full GCC (libgcc + libstdc++)
```

The mechanism that makes the minimal GCC possible is an internal configure variable called `inhibit_libc`, which builds a stripped libgcc needing no C library at all. It has one non-obvious extra condition that people trip over constantly, and Part III opens with it.

3.8 An archaeological note

Before sysroots existed you pointed GCC at the target headers and libraries with `--with-headers` and `--with-libs`, and configure **copied them into the install tree** (`configure.ac:2931`). The manual now marks both *"Deprecated in favor of `--with-sysroot`"* (`install.texi:2751-2753`, `:2769-2770`).

Two reasons to know they existed. You will meet them in old build scripts. And they explain why `$tooldir/include` and `$tooldir/sys-include` are *still* header search paths today, on a pure-sysroot toolchain that never asked for them — the surviving path is compiled in as `TOOL_INCLUDE_DIR` (`gcc/Makefile.in:2613`), and you will see it again in Chapter 1.9.

3.9 What to take from this

Two machines, one install tree. Look at any path and ask which machine's code is in it. `bin/` and `libexec/` are yours; `<target>/lib/` and the sysroot are the device's; `lib/gcc/<t>/<v>/` is both, which is why it is the confusing one. And a sysroot is a string prepended by `add_sysrooted_prefix`, applied to libc's half of the world and to nothing else.

Next: how to make a built compiler tell you every decision it has made.

All source references in this chapter are to **GCC 15.2.0** (`releases/gcc-15.2.0`). Line numbers in other releases will differ; the surrounding code rarely does. Where behaviour itself changed across a major version, it is flagged inline.

4 Asking a built compiler what it decided

Source references in this chapter are to GCC 15.2.0 (`releases/gcc-15.2.0`).

Something is wrong. A header resolves to the wrong file, a `-l` picks up a library from a directory you have never heard of, a flag you passed appears to do nothing. The temptation is to read documentation and reason about what *should* happen.

Don't. A built compiler will tell you what it *did*, and this chapter is the reference for how to ask. Every chapter after this one leans on these commands, so they come before the mechanisms they observe.

The whole family shares one property, visible in the source: they are handled in a single function that prints an answer and returns before any real work begins.

```
driver::maybe_print_and_exit () const
{
  if (print_search_dirs)
    ...
  if (print_file_name)
    ...
```

Listing 3-1: `driver::maybe_print_and_exit` (`gcc/gcc.cc:8793`), called from `gcc.cc:8301`.

That is why the manual gives them all the same "and don't do anything else" wording, and why passing two of them at once gets you only the first.

4.1 Pick your question

You want to know	Command
What commands would this actually run?	`gcc -### foo.c`
Which file would `-lfoo` or `crt1.o` resolve to?	`gcc -print-file-name=libfoo.a`
Where does it look for programs and libraries?	`gcc -print-search-dirs`
Where does it look for **headers**?	`gcc -v -E - < /dev/null`
Is a sysroot in play?	`gcc -print-sysroot`
What was this compiler configured with?	`gcc -v`
Which target is this?	`gcc -dumpmachine`
Which ABI variant am I getting?	`gcc -m32 -print-multi-directory`
Which variants exist at all?	`gcc -print-multi-lib`
Every predefined macro?	`gcc -dM -E - < /dev/null`
The driver's command-line templates?	`gcc -dumpspecs`

4.2 `-print-file-name=`: the honest one

```
$ gcc -print-file-name=libc.a
$ gcc -print-file-name=crt1.o
$ gcc -print-libgcc-file-name        # exactly -print-file-name=libgcc.a
```

This resolves a name through the driver's link-time search path and prints the absolute result. It is not an approximation of what the linker would find; it walks the same list the linker is handed as `-L` flags. `-print-libgcc-file-name` is not even a separate mechanism — it is one assignment, `print_file_name = "libgcc.a"` (`gcc/gcc.cc:4298`).

What makes it a diagnostic rather than merely informative is its behaviour on failure, and that behaviour is the entire body of the function:

```
find_file (const char *name)
{
```

```
  char *newname = find_a_file (&startfile_prefixes, name, R_OK, true);
  return newname ? newname : name;
}
```

Listing 3-2: `find_file` (`gcc/gcc.cc:8068-8072`).

If the file is not found, you get your argument back verbatim. No error, no non-zero exit — just the string you passed in. Which turns one command into a one-line diagnosis, because different files come from different owners:

Echoed back verbatim	What is wrong
`crt1.o`	your **sysroot** — that file belongs to libc
`crtbegin.o`	your **prefix or `-B`** — that file belongs to libgcc
`libstdc++.so`	libstdc++ was never installed where the driver looks

If you internalise one thing from this chapter, make it that table. Chapter 1.8 covers who provides which startup file and why they are interleaved.

4.3 `-print-search-dirs`: exactly three lines

```
$ gcc -print-search-dirs
```

The output has three lines and no more, because that is literally all the code prints:

```
      printf (_("install: %s%s\n"),
          gcc_exec_prefix ? gcc_exec_prefix : standard_exec_prefix,
          gcc_exec_prefix ? "" : machine_suffix);
      printf (_("programs: %s\n"),
          build_search_list (&exec_prefixes, "", false, false));
      printf (_("libraries: %s\n"),
          build_search_list (&startfile_prefixes, "", false, true));
```

Listing 3-3: `-print-search-dirs` (`gcc/gcc.cc:8795-8805`).

Line	Backed by	Used to find
`install:`	`gcc_exec_prefix` or `standard_exec_prefix`	— (the resolved install root)
`programs:`	`exec_prefixes`	`cc1`, `as`, `ld`, `collect2` — **host** executables
`libraries:`	`startfile_prefixes`	`crt*.o` and `-l` libraries — **target** files

> **There is no `headers:` line, and no driver-side option prints header paths.** This trips up everyone at least once. The reason is structural, not an oversight: the header search list is built and printed inside `cc1`, not inside the driver, so no `-print-*` option in `gcc` has access to it.

Two more things about Listing 3-3 worth knowing before you read the output.

The sysroot is already folded into the `libraries:` entries. It is prepended as the list is *built*, by `add_sysrooted_prefix` — you saw that in Chapter 1.2 — not applied to the output afterwards. So an entry that does not begin with your sysroot is not sysrooted and never will be.

And the multilib suffixes are *not* expanded here. What you see is the raw prefix list; each entry gets expanded four ways at lookup time, which is Chapter 1.7's subject. So a directory can appear in `libraries:` and still not be the one a particular `-m` flag actually searches.

The fastest way to find out whether your environment is interfering:

```
$ gcc -print-search-dirs
$ env -u GCC_EXEC_PREFIX -u COMPILER_PATH -u LIBRARY_PATH gcc -print-search-dirs
```

Diffing those two settles it in one step. Chapter 1.6 explains what each of those variables does.

4.4 The header search list, from `cc1`

```
$ gcc -v -E - < /dev/null
```

Three flags doing three jobs: `-` reads the program from standard input (which is empty), `-E` stops after preprocessing, and `-v` lets you see `cc1`'s stderr. The output includes a block like:

```
#include "..." search starts here:
#include <...> search starts here:
 /opt/gcc-arm/lib/gcc/arm-none-eabi/15.2.0/include
 ...
End of search list.
```

Those exact strings come from `cc1`, not the driver:

```
  /* If verbose, print the list of dirs to search.  */
  if (verbose)
    {
      fprintf (stderr, _("#include \"...\" search starts here:\n"));
      ...
      fprintf (stderr, _("End of search list.\n"));
```

Listing 3-4: the verbose printout (`gcc/incpath.cc:391-405`).

For C++ you must say so, because standard input has no extension to infer a language from:

```
$ g++ -v -E -x c++ /dev/null
```

Check whether the `include/c++/...` entries begin with your sysroot path. If they do, this toolchain was configured for sysroot-relative C++ headers; if they do not, they are prefix-relative and `--sysroot` will never move them. That distinction is Chapter 1.12's subject.

> **If you are on GCC 14 or earlier, this printout has one section fewer.** GCC 15 added a fifth include chain for C23's `#embed`, so `merge_include_chains` now sysroot-expands quote, bracket, system, after **and** embed chains (`gcc/incpath.cc:359-363`) and the verbose output gained an `#embed <...> search starts here:` block (`:408`). Confirmed by checking for `INC_EMBED` at each release tag: absent through 14.3.0, present at 15.2.0.

When a header resolves to a file you did not expect, the list is not enough — you want the nesting. `-H` prints the include tree, one dot of indentation per level:

```
$ echo '#include <limits.h>' | gcc -H -E -x c - > /dev/null
```

That is the command that makes `#include_next` visible, and Chapter 1.9 uses it.

Two related one-liners:

```
$ gcc -print-file-name=include              # GCC's own header directory
$ ls "$(gcc -print-file-name=include)"      # what your compiler actually ships
```

Note what the first one is doing: `include` is not a library, but `-print-file-name` does not care — it looks up whatever name you give it in the library search list, and GCC's own header directory happens to sit inside a directory that is on it.

4.5 `-print-sysroot`

```
$ gcc -print-sysroot
```

Empty output means no sysroot was configured. That is a meaningful answer, not a failure, and for a cross compiler it has a further consequence: such a compiler gets no `/lib` or `/usr/lib` search entries at all. Chapter 1.5 shows the single line of code that decides this, and explains why it is correct rather than unhelpful.

The manual warns that the printed path comes "possibly with an extra suffix that depends on compilation options", and that suffix is real:

```
  if (print_sysroot)
    {
      if (target_system_root)
    {
          if (target_sysroot_suffix)
        printf ("%s%s\n", target_system_root, target_sysroot_suffix);
          else
        printf ("%s\n", target_system_root);
    }
```

Listing 3-5: `-print-sysroot` (`gcc/gcc.cc:8876-8886`).

On targets that define `SYSROOT_SUFFIX_SPEC` — MIPS is the canonical case — the answer changes with `-EL` or `-msoft-float`. **Do not cache this value** in a build script. Chapter 1.7 covers the mechanism.

4.6 `-v`: what this compiler was built from

With no source file, `gcc -v` prints the version, the thread model, and the complete `configure` line the compiler was built with. That line is baked in at build time as a string constant in a generated header (`gcc/gcc.cc:225`, printed at `:7710`) — it is not re-derived, and it cannot be wrong.

Worth looking for in it:

- `--with-sysroot` — Chapter 1.5
- `--with-specs` — see below; it does not show up in `-dumpspecs`
- `--enable-languages`, and any `--with-arch` / `--with-cpu` defaults
- `--disable-multilib`, which explains a suspiciously short `-print-multi-lib`

With a source file, `-v` additionally prints every subprocess command line as it runs, plus every `COLLECT_*` variable the driver exports.

4.7 `-dumpspecs`, and the three spec pipelines

```
$ gcc -dumpspecs
```

This prints the driver's spec table and exits. Every line beginning with `*` is a spec name, and the following lines are its current value:

```
    case OPT_dumpspecs:
      {
    struct spec_list *sl;
    init_spec ();
    for (sl = specs; sl; sl = sl->next)
      printf ("*%s:\n%s\n\n", sl->name, *(sl->ptr_spec));
    if (link_command_spec)
      printf ("*link_command:\n%s\n\n", link_command_spec);
    exit (0);
      }
```

Listing 3-6: `-dumpspecs` (`gcc/gcc.cc:4216-4226`).

Note `init_spec ()` on the second line: the table is initialised, including any startup rewriting a target does, before being printed. So `-dumpspecs` shows you live values, not the macro text in the source. Chapter 1.4 makes a great deal of that distinction.

What `-dumpspecs` cannot show you is `--with-specs=`, and that is by design rather than an omission. Three different mechanisms are involved:

Mechanism	What it does	In `-dumpspecs`?
built-in specs	the compiled-in table	yes
`--with-specs=…` at configure time	rewrites the **command line**	**no**
`-specs=FILE` at run time	reads a file into the table at startup	not by a plain `-dumpspecs`

`--with-specs` becomes `CONFIGURE_SPECS` (`gcc/configure.ac:1084-1089`), which lands in an array of *self specs*:

```
static const char *const driver_self_specs[] = {
  "%{fdump-final-insns:-fdump-final-insns=.} %<fdump-final-insns",
  DRIVER_SELF_SPECS, CONFIGURE_SPECS, GOMP_SELF_SPECS, GTM_SELF_SPECS,
```

Listing 3-7: `driver_self_specs[]` (`gcc/gcc.cc:1361-1364`), applied at `:8528-8529`.

Those rewrite your argument list; they never touch the spec table. So `-dumpspecs` genuinely has nothing to show. Use `-###` to see the effect instead.

> **Check for a stray `specs` file when a toolchain misbehaves inexplicably.** The driver looks for a file literally named `specs` on its own library search path (`gcc/gcc.cc:8496-8499`) and, if it finds one, reads it into the table — silently rewriting the toolchain's behaviour. A default install does not ship one. Find out with:
>
> ```
> $ ls "$(dirname "$(gcc -print-libgcc-file-name)")"/specs
> ```

4.8 `-dM -E`: every predefined macro

```
$ echo | gcc -dM -E -x c - | sort
```

`-E` stops after preprocessing (required for `-dM`), and `-dM` replaces the preprocessed text with a `#define` for every macro live at the end of the run. It is parsed one character at a time in the C-family front end:

```
      case 'M':         /* Dump macros only.  */
```

Listing 3-8: `-dM` (`gcc/c-family/c-opts.cc:2064`, set into `flag_dump_macros` at `:2068`, consumed at `:1547`).

Useful narrowings:

```
$ gcc -dM -E - < /dev/null | grep '__SIZEOF'          # every type's size
$ gcc -dM -E - < /dev/null | grep '__GNUC'            # compiler version
$ gcc -dM -E - < /dev/null | grep -i 'arm\|thumb'     # what your -m flags did
$ gcc -dM -E - < /dev/null | grep __STDC_HOSTED__     # 1 hosted, 0 freestanding
```

Where those values come from is Part IV's subject.

4.9 The multilib family

All four of these resolve against the target's variant table, using the actual `-m*` flags on the command line you give them:

Option	Prints
`-print-multi-directory`	GCC-side subdirectory for these flags, `.` if none
`-print-multi-os-directory`	OS-side subdirectory — `.`, `../lib64`, `../lib32`, …
`-print-multi-lib`	the whole `dir;@flag@flag` mapping table
`-print-multiarch`	the multiarch subdirectory, e.g. `x86_64-linux-gnu`

The first two default to `.` rather than to nothing when there is no variant (`gcc/gcc.cc:8858-8865` and `:8887-8894`), whereas `-print-multiarch` prints an empty line (`:8866-8873`) — a small asymmetry that matters if you are consuming the output from a script.

The useful pairing is a directory question and a file question:

```
$ gcc -m32 -print-multi-os-directory     # which subdirectory
$ gcc -m32 -print-file-name=libc.a       # what is actually in it
```

Because of a fallback covered in Chapter 1.7, those two can disagree in a way that is worth understanding.

4.10 `-###` and `-v`, one more time

```
$ gcc -v   hello.c      # run everything, printing each command line
$ gcc -### hello.c      # print the command lines, quoted, run nothing
```

`-###` is the one to reach for, because it is safe to run on a broken toolchain and its output diffs cleanly against a working one. In practice you almost always want it piped:

```
$ gcc -### hello.c 2>&1 | grep collect2          # just the link line
$ gcc -### -nostdlib hello.c                     # confirm what a flag removed
$ gcc -specs=my.specs -### hello.c               # confirm what an override did
```

4.11 Documentation coverage

This is one area where the manuals are good. The whole `-print-*` family is documented under Overall Options (`gcc/doc/invoke.texi:1549`), and `-B`, `--sysroot` and the include and library search options under Directory Options (`:19484`).

Three things in this chapter are *not* documented, and are worth knowing you will not find:

- That `-print-search-dirs` has no `headers:` line, and that no driver option prints header paths at all. You have to notice that `cc1` prints them instead.
- That an installed `specs` file silently overrides the built-in table.
- That `-print-search-dirs` shows unexpanded prefixes, so its output is not the set of directories a given `-m` flag actually searches.

Next: the language those command-line templates are written in.

All source references in this chapter are to **GCC 15.2.0** (`releases/gcc-15.2.0`). Line numbers in other releases will differ; the surrounding code rarely does. Where behaviour itself changed across a major version, it is flagged inline.

5 Specs: how the driver writes a command line

Source references in this chapter are to GCC 15.2.0 (`releases/gcc-15.2.0`).

You pass `-static-libgcc` and the link line changes. You go looking for the code that implements it and there isn't any. There is no `if (static_libgcc)` anywhere in the link path, no function that consults a flag and appends a library.

What there is instead is a string:

```
%{static|static-libgcc|static-pie:-lgcc -lgcc_eh}%{!static:%{!static-libgcc: …
```

GCC's driver builds every command line by expanding templates written in a small `%`-escape language. Roughly: printf for command lines, with conditionals and function calls bolted on. This chapter is about reading and writing that language, because once you can, you can change a toolchain's link behaviour without rebuilding it — and you can explain why several otherwise baffling options behave the way they do.

The order here is deliberate. The *names* come first, then the `%` directives that reference them. The shorthands are meaningless until you know what they point at.

5.1 The table

A **spec** is a named entry in a table the driver holds in memory: a name, and the template text currently bound to it. `-dumpspecs` prints the whole table, and every line starting with `*` is a name:

```
$ gcc -dumpspecs | grep '^\*'                    # every name your driver has
$ gcc -dumpspecs | sed -n '/^\*libgcc:/,/^$/p'   # one name and its value
```

The table is a linked list of a struct whose first two comment lines tell you something important:

```
struct spec_list
{
                /* The following 2 fields must be first */
                /* to allow EXTRA_SPECS to be initialized */
  const char *name;     /* name of the spec.  */
  const char *ptr;      /* available ptr if no static pointer */
  ...
  const char **ptr_spec;    /* pointer to the spec itself.  */
```

Listing 4-1: `struct spec_list` (`gcc/gcc.cc:1685-1700`).

Note `ptr_spec`: an entry does not hold the text, it holds a **pointer to the C variable** that holds the text. The table is seeded from a static array:

```
  INIT_STATIC_SPEC ("libgcc",           &libgcc_spec),
  INIT_STATIC_SPEC ("startfile",        &startfile_spec),
```

Listing 4-2: two entries from `static_specs[]` (`gcc/gcc.cc:1729-1730`; the array begins at `:1707`).

Everything that mutates a spec — a target's override, a `-specs=` file, the driver's own startup rewriting — goes through one function and writes through that pointer (`set_spec`, `gcc/gcc.cc:2066`). Nothing else caches the text. That indirection is the single most useful fact in this chapter, and the reason is coming in a moment.

The names you will actually meet:

Name	Holds
`cpp`	options for the preprocessor
`cc1` / `cc1plus`	options for the C / C++ compiler proper
`asm`	options for the assembler
`link`	options for the linker
`lib`	the default system libraries — usually `-lc`
`libgcc`	GCC's helper library — `-lgcc` and friends
`startfile`	objects linked **before** yours
`endfile`	objects linked **after** yours
`link_command`	the complete template for the whole link

That set is hardcoded into the driver executable. There is no external file or database declaring which names exist; it is fixed when GCC is built, from three sources:

Source	Contributes
`static_specs[]` in `gcc/gcc.cc:1707`	the universal names above, on every target
target OS headers, e.g. `gcc/config/gnu-user.h`	`#define STARTFILE_SPEC …` — **replacements** for the defaults
target CPU headers, e.g. `gcc/config/aarch64/aarch64.h`	`#define EXTRA_SPECS …` — **additional** names such as `asm_cpu_spec`

This is why two GCCs built from identical source for different targets behave completely differently. Same code, different `#define`s. And it is why `-dumpspecs` output is target-specific: an `aarch64-none-elf-gcc` and an `x86_64-linux-gnu-gcc` list different names. Always dump *your* compiler rather than trusting a transcription, including the ones in this book.

There are two ways to use a name. You can **reference** it from inside another spec — `%(link)` expands the `link` spec right there, and the hot-path names have one-letter aliases so `%l` means the same thing. Or you can **override** it from a specs file. Both matter later.

5.2 Reading a conditional

Take the `libgcc` spec on a target with a shared libgcc. Its value is roughly:

```
%{static|static-libgcc|static-pie:-lgcc -lgcc_eh}
%{!static:%{!static-libgcc:%{!static-pie:
  %{!shared-libgcc:-lgcc --as-needed -lgcc_s --no-as-needed}
  %{shared-libgcc:-lgcc_s%{!shared: -lgcc}}}}}
```

There are only two constructs in there. `%{FLAG:text}` emits `text` if the flag was given; `%{!FLAG:text}` emits it if the flag was not. Everything else is nesting, and three details make the nesting readable:

- `|` is **or**. The first line fires if any of `-static`, `-static-libgcc` or `-static-pie` was given.
- There is no `&&`. Nesting negations is how you say and-not: `%{!a:%{!b:X}}`.
- The output half can itself contain conditionals. `X` in `%{S:X}` is more spec text, expanded recursively — which is why `%{shared-libgcc:-lgcc_s%{!shared: -lgcc}}` appends `-lgcc` only when you are not building a shared object.

Work it out and you get:

Your command	`libgcc` expands to
`gcc foo.o`	`-lgcc --as-needed -lgcc_s --no-as-needed`
`gcc -static foo.o`	`-lgcc -lgcc_eh`
`gcc -shared-libgcc foo.o`	`-lgcc_s -lgcc`
`gcc -shared -shared-libgcc foo.o`	`-lgcc_s`

Watch it happen:

```
$ gcc -### -static-libgcc foo.o 2>&1 | tr ' ' '\n' | grep lgcc
```

So `-static-libgcc` is not implemented by any C code in the link path. It is a switch tested by a conditional inside one string, which is exactly why replacing the string changes the behaviour with no rebuild.

5.3 Why `%G` names a slot, not a value

Here is where the pointer indirection from Listing 4-1 earns its keep, and where most secondhand explanations of GCC go wrong.

You will see the arrow `%G → LIBGCC_SPEC` written as though it were an equality. It is a provenance note. Four distinct things share that name:

#	Thing	Kind	Where
1	`LIBGCC_SPEC`	a **C preprocessor macro**, overridable by the target	`gcc.cc:878-889`
2	`libgcc_spec`	a **C variable**, initialised from the macro	`gcc.cc:1217`
3	`"libgcc"`	the **name** that variable is registered under	`gcc.cc:1729`
4	`%G`	a **directive** meaning "expand whatever is registered as `libgcc`"	`gcc.cc:6813`

The implementation of the directive is two lines and settles it:

```
      case 'G':
        value = do_spec_1 (libgcc_spec, 0, NULL);
```

Listing 4-3: `%G` (`gcc/gcc.cc:6813-6814`). `do_spec_1` is the expander.

So `%G` does not stand for the text of `LIBGCC_SPEC`. It **recursively expands the current value of the `libgcc` spec**, which merely started life as that macro. It is exactly equivalent to `%(libgcc)`, whose handler resolves the name through the same table (`gcc.cc:6931`).

That distinction is not pedantry, because the value gets rewritten at startup. The macro's own text is:

```
#define LIBGCC_SPEC "-lgcc"
```

Listing 4-4: the default `LIBGCC_SPEC` (`gcc/gcc.cc:887`).

Three words. But on a target configured with `ENABLE_SHARED_LIBGCC`, the driver walks that string at startup looking for the literal `-lgcc` and replaces it wholesale:

```
    const char *p = libgcc_spec;
    int in_sep = 1;

    /* Transform the extant libgcc_spec into one that uses the shared libgcc
```

```
       when given the proper command line arguments.  */
    while (*p)
      {
    if (in_sep && *p == '-' && startswith (p, "-lgcc"))
      {
        init_gcc_specs (&obstack,
                "-lgcc_s"
```

Listing 4-5: the shared-libgcc rewrite (`gcc/gcc.cc:1922-1931`).

The replacement text is assembled by `init_gcc_specs` (`gcc/gcc.cc:1816-1851`) and depends on `USE_LD_AS_NEEDED` and `LINK_EH_SPEC`, so it differs between targets. That is why the conditional in the previous section is "roughly" and not exactly, and why you should run `-dumpspecs` on your own toolchain.

The string a beginner reads in the driver source is not the string that expands.

5.4 Position is the point

The second worked example is the whole link, in one template. Trimmed to what matters:

```
%{!fsyntax-only:%{!c:%{!M:%{!MM:%{!E:%{!S:      ← only when actually linking
    %(linker)  %l                                ← collect2/ld, then the link spec
    %X %{o*} %{e*}                               ← your -Wl, options; -o; -e
    %{!nostdlib:%{!r:%{!nostartfiles:%S}}}       ← startfile: crt1 crti crtbegin
    %o                                           ← YOUR object files
    %{!nostdlib:%{!r:%{!nodefaultlibs:
        %(link_gcc_c_sequence)}}}                ← expands to  %G %L %G
    %{!nostdlib:%{!r:%{!nostartfiles:%E}}}       ← endfile: crtend crtn
    %{T*} }}}}}}
```

Listing 4-6: `LINK_COMMAND_SPEC`, abridged (`gcc/gcc.cc:1159-1178`).

`%l %S %E %G %L` are one-letter aliases for the names `link`, `startfile`, `endfile`, `libgcc` and `lib`. So Listing 4-6 **contains the previous section by reference**: each `%G` expands that entire nested conditional.

Three things to take from it.

Position in the template is position on the command line. `%S` … `%o` … libraries … `%E` *is* the link order. The whole startfile/endfile split exists because `%S` and `%E` sit on opposite sides of your object files. That is the subject of Chapter 1.8.

`-nostdlib`, `-nostartfiles`, `-nodefaultlibs` and `-r` are pure conditionals. No C code implements them; they fail a `%{!…:}` guard and the text disappears. The guards are on three specific lines — `gcc.cc:1168`, `:1176` and `:1177` — and reading the nesting gives you the truth table directly. Chapter 1.10 does that.

A target customises one small name, not this whole string. The `%G %L %G` ordering lives in its own spec precisely so a port can change it, and the comment above it says so:

```
/* This is overridable by the target in case they need to specify the
   -lgcc and -lc order specially, yet not require them to override all
   of LINK_COMMAND_SPEC.  */
#ifndef LINK_GCC_C_SEQUENCE_SPEC
#define LINK_GCC_C_SEQUENCE_SPEC "%G %{!nolibc:%L %G}"
#endif
```

Listing 4-7: `LINK_GCC_C_SEQUENCE_SPEC` (`gcc/gcc.cc:987-992`).

Yes, `%G` really is there twice. Chapter 1.8 explains why.

5.5 Directives that expand state, not text

The third family reads the driver's own state or names files. You cannot see their values by looking at a spec — but every one of them is printable with a command from Chapter 1.3.

Directive	Reads	Emits	Print it with
`%D`	the library search list	one `-L` per directory	`-print-search-dirs`
`%R`	the sysroot	the bare path	`-print-sysroot`
`%M`	the multilib OS directory	`.`, `../lib32`, `../lib64`	`-print-multi-os-directory`
`%I`	GCC's own header directories	`-isystem …/include`, `…/include-fixed`	`-print-file-name=include`
`%X` `%Y` `%Z`	your `-Wl,` / `-Wa,` / `-Wp,` options	them, prefix stripped	`-###`

The driver's own documentation of `%D` and `%R`, from the long comment that defines the language:

```
%D Dump out a -L option for each directory in startfile_prefixes.
   If multilib_dir is set, extra entries are generated with it affixed.
...
%R Output the concatenation of target_system_root and
       target_sysroot_suffix.
```

Listing 4-8: two directives, self-documented (`gcc/gcc.cc:592-593` and `:599-600`).

`%R` is *the* sysroot mechanism. When Chapter 1.2 said a sysroot is "a string glued onto the front of certain paths", `%R` is where the gluing is written for spec-level consumers. A target that wants sysroot-relative link behaviour puts `%R` in its spec; one that does not, does not. Its implementation is exactly the concatenation the comment describes (`gcc.cc:6819-6830`).

Then the file-naming family:

Directive	Names
`%i`	the input file for this step
`%o`	all output files — in the link, where *your* `.o` files land
`%O`	the object suffix, `.o`
`%s`	**look this name up in the library search list**
`%g…` `%u…` `%U…` `%j…`	temporary files, differing in whether the name is shared or unique

You met `%g` in Chapter 1.1 without being told its name. The scratch `.s` file that carries assembly from `cc1` to `as` is `%g.s` expanding — *shared*, so two separate programs agree on a file nobody named:

```
$ gcc -### hello.c 2>&1 | grep -o '/tmp/cc[A-Za-z0-9]*\.s'
```

5.5.1 `%s`, which causes the most confusion of any directive

Consider a startfile spec containing:

```
crt1.o%s   crti.o%s   crtbegin.o%s
└ from libc ──────┘   └ from libgcc ┘
```

All three are written identically. Yet `crt1.o` comes from the C library's sysroot and `crtbegin.o` from GCC's own prefix. There is no hint of that asymmetry in the spec, because `%s` means nothing more than "search for this name", and the driver's comment is explicit that it is one list:

```
%s     current argument is the name of a library or startup file of some sort.
       Search for that file in a standard list of directories
   and substitute the full name found.
```

Listing 4-9: `%s` (`gcc/gcc.cc:571-573`).

One search list; two providers; the difference invisible in the spec and visible in one command:

```
$ gcc -print-file-name=crt1.o        # the libc half
$ gcc -print-file-name=crtbegin.o    # the libgcc half
```

5.6 The rest of the language

The language documents itself in a 230-line comment at `gcc/gcc.cc:471-700`, which is worth reading once end to end. The summary:

Name references. `%(name)` for any name; `%G`, `%L`, `%S`, `%E`, `%l`, `%a`, `%1`, `%2`, `%C` as one-letter aliases for `libgcc`, `lib`, `startfile`, `endfile`, `link`, `asm`, `cc1`, `cc1plus`, `cpp`.

Conditionals, the `%{...}` family:

Form	Meaning
`%{S}`	emit `-S` if it was given
`%{S*}`	emit every switch starting `-S`, arguments included
`%{S:X}`	emit `X` if `-S` was given
`%{!S:X}`	emit `X` if `-S` was not given
`%{S|T:X}`	emit `X` if `-S` or `-T`
`%{.S:X}`	emit `X` if the input file has suffix `.S`
`%{S:X;T:Y;:D}`	if / else-if / else, with `D` as the default arm

That last form is worth knowing because real target specs use it heavily — `%{msoft-float:-soft;:-hard}` emits one or the other, and Chapter 1.8 shows a glibc startfile spec that selects among five different `crt1` variants with it. Conditionals are dispatched to one function ([`case '{'` at `gcc.cc:6840`, `handle_braces` at `:7267`).

Escaping is a backslash, which is why you will see `%{std=iso9899\:1999:X}` — without it, that colon would be read as the `:X` separator.

Function calls, `%:name(args)`, dispatched through a registry:

```
static const struct spec_function static_spec_functions[] =
{
  { "getenv",                   getenv_spec_function },
  { "if-exists",        if_exists_spec_function },
  ...
```

Listing 4-10: the spec-function table (`gcc/gcc.cc:1775-1801`), dispatched at `case ':'`, `:6846`.

The useful ones are `%:getenv(VAR SUFFIX)`, `%:if-exists(f)` and `%:if-exists-else(a b)` for picking a file that is actually present, `%:include(libgomp.spec)` for pulling in another specs file mid-expansion, `%:version-compare(...)` for Darwin-style OS gating, `%:sanitize(address)` to test which sanitizer is on, and the numeric predicates `%:gt`, `%:debug-level-gt`, `%:dwarf-version-gt`. The return string is re-processed as spec text. Used as a predicate, `%{%:function(args):X}` emits `X` when the function returns something non-empty.

Note that the table ends with `EXTRA_SPEC_FUNCTIONS`, so a port can add its own.

5.7 Writing a specs file

A specs file overrides entries in the table. This is how you change a toolchain's behaviour without rebuilding it, and it is the mechanism behind, for example, ARM's `nano.specs`.

```
*libgcc:
-lgcc -lgcc_eh
```

```
*startfile:
+ my-extra-crt.o
```

Listing 4-11: a two-entry specs file.

The syntax is: `*name:` alone on a line, the value on the following lines, terminated by a **blank line**. A value beginning with `+` — plus, then a space — appends to the existing value instead of replacing it, and that is one line of C:

```
  *(sl->ptr_spec) = ((spec[0] == '+' && ISSPACE ((unsigned char)spec[1]))
             ? concat (old_spec, spec + 1, NULL)
             : xstrdup (spec));
```

Listing 4-12: replace or append, in `set_spec` (`gcc/gcc.cc:2106-2108`).

Two directives work only in a file, not in a spec: `%rename old new` moves a spec's text to a new name (`gcc.cc:2454`), and `%include FILE` / `%include_noerr FILE` splice in another file (`:2411`).

`%rename` exists for the idiom you will actually want, which is *adding to* a spec rather than replacing it. Rename the original out of the way, then reference it:

```
%rename link old_link

*link:
%(old_link) -extra-flag
```

Listing 4-13: the wrapping idiom.

A specs file may also introduce an entirely new name — `set_spec` creates the entry if it does not exist (`gcc.cc:2066`) — which is only useful if something references it with `%(name)`. That is exactly what Listing 4-13 is doing.

Then:

```
$ gcc -specs=my.specs -### hello.c    # see exactly what your override did
```

Multiple `-specs=` are applied in the order given.

5.8 Where specs come from, in order

Later entries win:

1. **Built-in defaults**, compiled into the driver, each overridable by a target.
2. **Target overrides**, from your target's config headers.
3. **Driver startup rewrites** — the `libgcc` rewrite of Listing 4-5 is the one you are most likely to meet, and it is what makes `-static-libgcc` work at all.
4. **A `specs` file next to the driver** (`gcc.cc:8496-8499`). Not shipped by default, but if one is present it silently rewrites the toolchain.
5. **`-specs=FILE`** on the command line (`gcc.cc:8633-8640`).

5.9 Debugging

```
$ gcc -dumpspecs                      # the whole table, current values
$ gcc -### hello.c                    # every expanded command line, nothing run
$ gcc -specs=my.specs -### hello.c    # what your override actually did
```

For the truly stuck, the driver has a compile-time `DEBUG_SPECS` that narrates every `%(name)` expansion and every `set_spec` (`gcc.cc:2110-2113`). It is `#ifdef`-only: you have to rebuild the driver with it defined, which puts it firmly in the last-resort category.

5.10 Documentation coverage

The spec language has its own manual node, Spec Files (`gcc/doc/invoke.texi:37290`), and it is a decent reference for the directives. What it does not cover, and what this chapter exists for:

- That `%G` expands a *mutable slot* rather than the macro's text, and that the slot is rewritten at startup on shared-libgcc targets. The manual describes `%G` as processing `LIBGCC_SPEC`, which is true only before startup.
- That an installed `specs` file overrides the built-in table.
- That `--with-specs` rewrites your command line via `driver_self_specs` rather than editing the table, and therefore cannot appear in `-dumpspecs`.

Next: the search lists that `%D`, `%I`, `%R` and `%s` all draw on.

All source references in this chapter are to **GCC 15.2.0** (`releases/gcc-15.2.0`). Line numbers in other releases will differ; the surrounding code rarely does. Where behaviour itself changed across a major version, it is flagged inline.

6 How the driver finds anything

Source references in this chapter are to GCC 15.2.0 (`releases/gcc-15.2.0`).

Your cross compiler cannot find `crt1.o`. You add `--sysroot=/arm-rootfs`, and now it can. So you try the same trick on `crtbegin.o`, which is also missing, and nothing happens — it is still missing, and `--sysroot` has no effect on it whatever.

Both files arrive on the link line through the same directive, `%s`. Both are looked up in the same list. And yet one follows your sysroot and the other ignores it.

The answer is that the list is built from entries registered by two different functions, and only one of them prepends the sysroot. This chapter is about how that list gets built, in what order, and which flag touches which part of it.

6.1 Two lists, not one

The driver keeps two search lists (three, counting headers, which get a chapter of their own):

List	Finds	Printed as
`exec_prefixes`	`cc1`, `as`, `ld`, `collect2` — **host** programs	`programs:`
`startfile_prefixes`	`crt*.o`, and every `-l` — **target** files	`libraries:`

```
$ gcc -print-search-dirs
```

They are genuinely separate lists in separate variables, and a flag can feed one, the other, or both. `-L` feeds only the second. `-B` feeds all three, which is why it is the blunt instrument.

Everything the linker eventually searches comes out of `startfile_prefixes`, via one directive:

```
%D Dump out a -L option for each directory in startfile_prefixes.
   If multilib_dir is set, extra entries are generated with it affixed.
```

Listing 5-1: `%D` (`gcc/gcc.cc:592-593`; handler at `:6250`).

So the driver's "library search path" is not a path it searches. It is a list it converts into `-L` flags. **The driver never opens a library file.** It emits `-l` and `-L` and lets `ld` do the resolving. The one exception is `-print-file-name`, which walks the list itself precisely so that it can tell you what `ld` would conclude.

6.2 The order the list is built in

`startfile_prefixes` is assembled during startup, and the order of the calls is the search order. In sequence:

1. **`-B` prefixes**, from the command line (`gcc.cc:4590`).
2. **`GCC_EXEC_PREFIX`**, from the environment (`gcc.cc:4784`).
3. **`LIBRARY_PATH`**, from the environment — *native compilers only* (`gcc.cc:4923`).
4. **The relocated install prefix**, computed from `argv[0]` (`gcc.cc:4818-4835`).
5. **The tool directory**, `$prefix/<target>/lib/` (`gcc.cc:5537-5539`).
6. **Either** `STARTFILE_PREFIX_SPEC` (`gcc.cc:8579-8587`) **or** the `STANDARD_STARTFILE_PREFIX*` chain (`gcc.cc:8590-8630`) — never both.

That ordering is documented, in the internals manual rather than the user manual: `gcc/doc/tm.texi:570` begins *"Here is the order of prefixes tried for startfiles"* and enumerates them, including the conditions. If you only remember one manual reference from Part I, `info gccint 'Target Macros' Driver` is a good candidate.

Let us take the interesting steps in turn.

6.3 `-B` wins, and nothing can sort before it

```
    case OPT_B:
      {
```

```
    size_t len = strlen (arg);

    /* Catch the case where the user has forgotten to append a
       directory separator to the path.  Note, they may be using
       -B to add an executable name prefix, eg "i386-elf-", in
       order to distinguish between multiple installations of
       GCC in the same directory.  Hence we must check to see
       if appending a directory separator actually makes a
       valid directory name.  */
    if (!IS_DIR_SEPARATOR (arg[len - 1])
        && is_directory (arg))
      {
        ...
      }

    add_prefix (&exec_prefixes, arg, NULL,
            PREFIX_PRIORITY_B_OPT, 0, 0);
    add_prefix (&startfile_prefixes, arg, NULL,
            PREFIX_PRIORITY_B_OPT, 0, 0);
    add_prefix (&include_prefixes, arg, NULL,
            PREFIX_PRIORITY_B_OPT, 0, 0);
      }
```

Listing 5-2: `-B` (`gcc/gcc.cc:4590-4619`).

Three things in one listing.

`-B` feeds all three lists. That is the whole of its definition. It is not a library path option or an executable path option; it is a prefix applied to programs, link inputs and headers simultaneously, which is exactly why it fixes problems you did not know you had and breaks things you did not expect.

The trailing `/` is added for you — but only conditionally, and the comment explains why. A `-B` argument without a trailing separator is a legitimate *executable name* prefix, so the driver has to check whether appending a separator would name a real directory before doing it. If you write `-B/opt/alt` and `/opt/alt` exists, you get `/opt/alt/`. If it does not exist, you have just told the driver that your programs are called `/opt/altcc1` and so on.

`PREFIX_PRIORITY_B_OPT` is the highest priority there is. And "highest" is stronger than it sounds, because the enum has exactly two members:

```
/* Ranking of prefixes in the sort list. -B prefixes are put before
   all others.  */

enum path_prefix_priority
{
  PREFIX_PRIORITY_B_OPT,
  PREFIX_PRIORITY_LAST
};
```

Listing 5-3: the priority enum (`gcc/gcc.cc:3120-3127`).

Two values. So **`-B` beats the environment, and the environment beats the compiled-in defaults**, and there is no way to express "before `-B`". Every environment contribution and every compiled-in default lands at `PREFIX_PRIORITY_LAST` and is ordered only by when it was added.

One thing `-B` does *not* do: it does not move C++ header search. It adds GCC's *own* header directories, which is enough to change which `<stdint.h>` you compile against but not which `<vector>`. That asymmetry is Chapter 1.9's and Chapter 1.12's territory. And it has nothing to do with the linker's `-Bstatic`.

6.4 Relocation: how a moved toolchain still works

You can move an installed GCC tree to a completely different directory and it keeps working. That is not luck, and it is not RPATH. The driver recomputes its own prefix from `argv[0]` on every single run:

```
  gcc_libexec_prefix = standard_libexec_prefix;
#ifndef VMS
  /* FIXME: make_relative_prefix doesn't yet work for VMS.  */
  if (!gcc_exec_prefix)
    {
      gcc_exec_prefix = get_relative_prefix (decoded_options[0].arg,
                         standard_bindir_prefix,
                         standard_exec_prefix);
      gcc_libexec_prefix = get_relative_prefix (decoded_options[0].arg,
                         standard_bindir_prefix,
                         standard_libexec_prefix);
      if (gcc_exec_prefix)
    xputenv (concat ("GCC_EXEC_PREFIX=", gcc_exec_prefix, NULL));
    }
```

Listing 5-4: relocation (`gcc/gcc.cc:4822-4835`). `get_relative_prefix` is `make_relative_prefix` from libiberty, assigned at `:4816`.

Read the three arguments. It knows where it was *configured* to be installed (`standard_bindir_prefix`, `standard_exec_prefix`), it knows where it *actually is* (`decoded_options[0].arg`, which is `argv[0]`), and it computes the same relative relationship against the new location. If you configured with `--prefix=/opt/gcc-arm` and the binary is now at `/home/me/toolchain/bin/gcc`, then `standard_exec_prefix` is recomputed as `/home/me/toolchain/lib/gcc/`.

Notice the `xputenv` on the last line. The driver **exports the answer** as `GCC_EXEC_PREFIX`, whether or not you set that variable yourself. So relocation and the environment variable are two faces of one mechanism, and the driver's own comment a few lines later says so:

```
  /* From this point onward, gcc_exec_prefix is non-null if the toolchain
     is relocated. The toolchain was either relocated using GCC_EXEC_PREFIX
     or an automatically created GCC_EXEC_PREFIX from
     decoded_options[0].arg.  */
```

Listing 5-5: the driver's summary of its own state (`gcc/gcc.cc:4855-4858`).

That variable, and the fact that setting it merely pre-empts this computation, is Chapter 1.6's subject.

There is a caveat the manual states and that people get wrong. Relocation moves GCC's own tree. It does **not** in general move your sysroot — that only happens when the sysroot was configured to live inside the exec prefix, in which case configure defines an extra macro and the driver relocates it too (`gcc/configure.ac:176-182`, acted on at `gcc/gcc.cc:5542-5559`). The manual's wording:

> If the specified directory is a subdirectory of `${exec_prefix}`, then it will be found relative to the GCC binaries if the installation tree is moved.
>
> — `gcc/doc/install.texi:2720-2722`

Bare `--with-sysroot` with no argument defaults to `$exec_prefix/<target>/sys-root` (`gcc/configure.ac:169-170`), which satisfies that condition — so the default is relocatable and an explicit `--with-sysroot=/arm-rootfs` is not. That is a good default and an easy surprise.

6.5 The tool directory

Two lines put the target's own subtree on the lists:

```
  add_prefix (&exec_prefixes,
          concat (tooldir_prefix, "bin", dir_separator_str, NULL),
          "BINUTILS", PREFIX_PRIORITY_LAST, 0, 0);
  add_prefix (&startfile_prefixes,
          concat (tooldir_prefix, "lib", dir_separator_str, NULL),
          "BINUTILS", PREFIX_PRIORITY_LAST, 0, 1);
```

Listing 5-6: the tool directory (`gcc/gcc.cc:5534-5539`).

`$prefix/<target>/bin/` for programs, `$prefix/<target>/lib/` for libraries. Note `add_prefix`, not `add_sysrooted_prefix` — **the tool directory is never sysrooted.** This is the single most important line for bare-metal toolchains: it is why, after you install newlib into `$prefix/<target>/lib/`, `-lc` simply works with no `-L` and no sysroot. And it is why, in a cross install, `libstdc++.so` is found there rather than in your sysroot no matter what `--sysroot` you pass.

The final argument, `1`, marks this prefix as an *OS multilib* prefix, which changes how it gets expanded. That is Chapter 1.7.

6.6 The startfile chain, and the cross-compiler gate

The last step is the one with a branch in it. Either the target defines `STARTFILE_PREFIX_SPEC`, in which case it wins outright:

```
  /* Look for startfiles in the standard places.  */
  if (*startfile_prefix_spec != 0
      && do_spec_2 (startfile_prefix_spec, NULL) == 0
      && do_spec_1 (" ", 0, NULL) == 0)
    {
      for (const char *arg : argbuf)
    add_sysrooted_prefix (&startfile_prefixes, arg, "BINUTILS",
                  PREFIX_PRIORITY_LAST, 0, 1);
    }
```

Listing 5-7: the `STARTFILE_PREFIX_SPEC` branch (`gcc/gcc.cc:8579-8587`).

or the standard chain runs — and it is guarded:

```
  /* We should eventually get rid of all these and stick to
     startfile_prefix_spec exclusively.  */
  else if (*cross_compile == '0' || target_system_root)
```

Listing 5-8: the gate (`gcc/gcc.cc:8588-8590`).

That one line is worth staring at. It says: add the standard system library directories **only if** this is a native compiler, **or** a sysroot is configured.

> **A cross compiler with no sysroot gets no `/lib` and no `/usr/lib` at all.**
>
> This is deliberate and correct. Searching the *host's* `/usr/lib` for target libraries is how you get baffling `file in wrong format` errors from the linker, or worse, a link that succeeds against completely wrong objects. If your cross compiler seems to have suspiciously few entries in `libraries:`, this line is why, and it is protecting you.

Inside the gate, the entries that make it in:

```
      /* Sysrooted prefixes are relocated because target_system_root is
     also relocated by gcc_exec_prefix.  */
      if (*standard_startfile_prefix_1)
    add_sysrooted_prefix (&startfile_prefixes,
                  standard_startfile_prefix_1, "BINUTILS",
                  PREFIX_PRIORITY_LAST, 0, 1);
      if (*standard_startfile_prefix_2)
    add_sysrooted_prefix (&startfile_prefixes,
                  standard_startfile_prefix_2, "BINUTILS",
                  PREFIX_PRIORITY_LAST, 0, 1);
```

Listing 5-9: where the sysroot gets glued on (`gcc/gcc.cc:8621-8630`).

And those two prefixes are exactly what you would guess:

```
#ifndef STANDARD_STARTFILE_PREFIX_1
#define STANDARD_STARTFILE_PREFIX_1 "/lib/"
#endif
#ifndef STANDARD_STARTFILE_PREFIX_2
#define STANDARD_STARTFILE_PREFIX_2 "/usr/lib/"
#endif
```

Listing 5-10: the two standard prefixes (`gcc/gcc.cc:1613-1618`).

`add_sysrooted_prefix`, from Chapter 1.2, is where `/lib/` becomes `/arm-rootfs/lib/`. **That call is the entire sysroot mechanism for libraries.** Nothing else applies it, nothing can retroactively apply it, and the entries registered with plain `add_prefix` — the tool directory, `-B`, `GCC_EXEC_PREFIX`, the relocated install prefix — are never sysrooted at all.

Which answers the question this chapter opened with. `crt1.o` lives in the sysroot's `/usr/lib`, reached through Listing 5-9. `crtbegin.o` lives in libsubdir, reached through the relocated install prefix registered with `add_prefix`. Same `%s`, same list, different registration function, different answer to `--sysroot`.

There is one more piece of the machine-dependent chain worth noting, because it explains why native compilers behave differently: `MD_EXEC_PREFIX`, `MD_STARTFILE_PREFIX` and `MD_STARTFILE_PREFIX_1` are `#undef`'d outright for a cross compiler, with the comment *"Don't use these prefixes for a cross compiler"* (`gcc/gcc.cc:1620-1624`).

6.7 The linker gets told about the sysroot too

Prefixes are not the only route. If `ld` supports it, the driver passes the sysroot straight through, so that `ld`'s own built-in search directories and any `=`-prefixed path inside a linker script become sysroot-relative:

```
#ifdef HAVE_LD_SYSROOT
  /* Pass the --sysroot option to the linker, if it supports that.  If
     there is a sysroot_suffix_spec, it has already been processed by
     this point, so target_system_root really is the system root we
     should be using.  */
  if (target_system_root)
    {
      obstack_grow (&obstack, "%(sysroot_spec) ", strlen ("%(sysroot_spec) "));
      obstack_grow0 (&obstack, link_spec, strlen (link_spec));
      set_spec ("link", XOBFINISH (&obstack, const char *), false);
    }
#endif
```

Listing 5-11: prepending the sysroot to the `link` spec (`gcc/gcc.cc:8554-8566`).

Note what that does: it *rewrites the `link` spec* at startup, prepending a reference to another spec. And that other spec is one line:

```
#ifndef SYSROOT_SPEC
# define SYSROOT_SPEC "--sysroot=%R"
#endif
```

Listing 5-12: `SYSROOT_SPEC` (`gcc/gcc.cc:1189-1191`).

`%R` expands to the sysroot, from Chapter 1.4. So `--sysroot=/arm-rootfs` on your command line becomes `--sysroot=/arm-rootfs` on `ld`'s command line, and you can see it:

```
$ gcc --sysroot=/arm-rootfs -### hello.c 2>&1 | grep -o '\-\-sysroot=[^ "]*'
```

Listing 5-11 is also a nice demonstration of Chapter 1.4's claim that `%G` and friends name mutable slots. Here the driver is mutating the `link` slot at startup, by name, through `set_spec`.

6.8 Which knob do I want?

Four flags look like "where GCC looks for things" and are four different mechanisms. Two are configure-time and two are usage-time.

Flag	When	Moves	Whose world
`--prefix`	configure	where GCC installs itself	GCC's own tree
`--with-sysroot`	configure	the default target root for libc	C library's tree
`--with-build-sysroot`	configure	libc's location *during the build only*	C library's tree
`--sysroot`	usage	overrides the configured sysroot for one run	C library's tree
`-B`	usage	three search lists at once	both

The two usage-time ones are what this chapter has been about. `--sysroot` overwrites `target_system_root` (`gcc.cc:4559`), which every `add_sysrooted_prefix` call and every `%R` then picks up.

The configure-time half belongs to Part II, but one thing is worth saying now because it explains a category of bug. `--with-build-sysroot` sets the sysroot used **only while building the target libraries**, and it is refreshingly literal:

```
AC_ARG_WITH(build-sysroot,
  [...],
  [if test x"$withval" != x ; then
     SYSROOT_CFLAGS_FOR_TARGET="--sysroot=$withval"
   fi], ...)
```

Listing 5-13: `--with-build-sysroot` *(*`gcc/configure.ac:140-146`*).*

It is an extra `--sysroot=` appended to the command line the build uses to compile libgcc and libstdc++. It leaves no trace in the installed compiler. The reason you need it is that libstdc++'s configure runs several hundred compile-and-link probes against the target C library, and if those cannot link, features configure themselves to *not available* — so you get a quietly crippled libstdc++ rather than a build failure. Part III returns to this.

6.9 Diagnosing it

The whole chapter reduces to four commands and one table.

```
$ arm-linux-gnueabihf-gcc -print-sysroot              # the baked-in sysroot
$ arm-linux-gnueabihf-gcc -print-search-dirs          # the two lists
$ arm-linux-gnueabihf-gcc -print-file-name=crt1.o     # the libc half
$ arm-linux-gnueabihf-gcc -print-file-name=crtbegin.o # the libgcc half
```

Symptom	Diagnosis
`crt1.o` echoed back verbatim	your **sysroot** is wrong
`crtbegin.o` echoed back verbatim	your **prefix or** `-B` is wrong
`libraries:` suspiciously short on a cross	no sysroot configured — Listing 5-8
`-print-search-dirs` looks right, link still fails	multilib expansion — Chapter 1.7

And do not forget that three environment variables feed these same lists invisibly. Nothing on your command line reveals them, so when a toolchain behaves differently in CI than on your desk:

```
$ gcc -print-search-dirs
$ env -u GCC_EXEC_PREFIX -u COMPILER_PATH -u LIBRARY_PATH gcc -print-search-dirs
```

That is the next chapter.

6.10 Documentation coverage

The startfile prefix order is genuinely well documented, in the internals manual at `gcc/doc/tm.texi:570`. `-B` and `--sysroot` are in Directory Options (`gcc/doc/invoke.texi:19484`), and the configure-time flags in `install.texi`.

Two things are not documented:

- That `add_prefix` versus `add_sysrooted_prefix` is what decides whether a directory follows `--sysroot`. The manual lists which prefixes get "any sysroot modifications" per entry, which is the same information, but it never says the mechanism is per-entry and irreversible.
- That `-print-search-dirs` shows the list *before* multilib expansion.

Next: the same lists, moved invisibly from the environment.

All source references in this chapter are to **GCC 15.2.0** (`releases/gcc-15.2.0`). Line numbers in other releases will differ; the surrounding code rarely does. Where behaviour itself changed across a major version, it is flagged inline.

7 The environment that moves your search paths

Source references in this chapter are to GCC 15.2.0 (`releases/gcc-15.2.0`).

The build works on your machine and fails in CI. Same source, same compiler version, same command line — you have checked the command line three times.

Then check the environment. Five variables steer the GCC driver, and none of them appears on any command line you can read. They split cleanly into two groups: ones **you set** to move its search paths, and ones **the driver sets** so that its own subprocesses agree with it.

Direction	Variables
In — you set them, the driver reads them	`GCC_EXEC_PREFIX`, `COMPILER_PATH`, `LIBRARY_PATH`, `LPATH`
Out — the driver sets them, its children read them	`COLLECT_GCC`, `COLLECT_GCC_OPTIONS`, `COLLECT_LTO_WRAPPER`, `COLLECT_AS_OPTIONS`

This chapter assumes Chapter 1.5, because the first group feeds exactly the lists that chapter built.

7.1 The three you might set

Variable	Feeds	Works on a cross compiler?
`GCC_EXEC_PREFIX`	programs and libraries, plus header search	yes
`COMPILER_PATH`	programs **and** headers	yes
`LIBRARY_PATH`	libraries	**no — native only**
`LPATH`	as `LIBRARY_PATH`, legacy	**no — native only**

Their priority is settled by Listing 5-3 in the previous chapter: the enum has two values, and all three of these land at `PREFIX_PRIORITY_LAST`. So **`-B` beats the environment, and the environment beats the compiled-in defaults**, with no way to sort anything before a `-B`.

7.1.1 `LIBRARY_PATH` does nothing on a cross compiler

This one wastes a lot of people's afternoons, and the reason is one clause:

```
  temp = env.get (LIBRARY_PATH_ENV);
  if (temp && *cross_compile == '0')
```

Listing 6-1: the native-only guard (`gcc/gcc.cc:4923-4924`; `LPATH` gets the same treatment at `:4956-4957`).

`*cross_compile == '0'` means "this is a native compiler". So if you are debugging why `LIBRARY_PATH` has no effect on your `arm-linux-gnueabihf-gcc`, **that is the entire answer**. Use `-B` or `-L` instead.

`GCC_EXEC_PREFIX` and `COMPILER_PATH` have no such guard and work on a cross (`gcc.cc:4889`).

Note also `LIBRARY_PATH_ENV`, which is a macro rather than a literal (`gcc.cc:209-211`) — a port can rename the variable, and some do.

7.1.2 `GCC_EXEC_PREFIX` is the relocation mechanism, seen from outside

You already met this variable in Chapter 1.5 without setting it. Its documented default is `$prefix/lib/gcc/`, and what makes it interesting is that **when you don't set it, the driver derives it from `argv[0]` and exports it anyway** — Listing 5-4. Setting it by hand merely pre-empts that computation.

When it is set, by you or by the driver, it does three things:

```
      set_std_prefix (gcc_exec_prefix, len);
      add_prefix (&exec_prefixes, gcc_libexec_prefix, "GCC",
          PREFIX_PRIORITY_LAST, 0, 0);
      add_prefix (&startfile_prefixes, gcc_exec_prefix, "GCC",
          PREFIX_PRIORITY_LAST, 0, 0);
```

Listing 6-2: what `GCC_EXEC_PREFIX` feeds (`gcc/gcc.cc:4879-4884`).

The two `add_prefix` calls are the search lists. The `set_std_prefix` call is the one people miss: it perturbs **header** search too, by telling `gcc/prefix.cc` what to substitute for the configure-time prefix in compiled-in directory names. That is how a relocated toolchain finds its own `<stdint.h>`, and Chapter 1.9 shows the consuming code.

Two smaller behaviours worth knowing:

A trailing `/lib/gcc/` is stripped. There is a small block that recognises that suffix and shortens the length before use (`gcc.cc:4864-4877`), so pointing the variable at either `$prefix/` or `$prefix/lib/gcc/` behaves the same. That is a kindness, not a coincidence.

It relocates the sysroot too, but only if the toolchain was configured with the sysroot inside `$exec_prefix` (`gcc.cc:5542-5559`). Chapter 1.5 covers the condition.

And one related flag, which is really about this mechanism: `-no-canonical-prefixes` changes *which* relative-prefix function gets used, so a symlinked driver resolves to the symlink's directory rather than the real one:

```
      if (decoded_options[j].opt_index == OPT_no_canonical_prefixes)
    {
      get_relative_prefix = make_relative_prefix_ignore_links;
      break;
    }
```

Listing 6-3: `-no-canonical-prefixes` (`gcc/gcc.cc:4807-4816`).

Note the comment above it: this has to be handled *early*, before normal option processing, because the prefixes it affects are needed to build the default search paths. It is the only option in the driver treated that way.

7.1.3 Proving whether the environment is to blame

```
$ gcc -print-search-dirs

$ env -u GCC_EXEC_PREFIX -u COMPILER_PATH \
      -u LIBRARY_PATH gcc -print-search-dirs
```

Diffing those two outputs is the fastest way to settle it, and it is a good first move whenever a toolchain behaves differently in two places.

7.1.4 The trap: the driver rewrites them

Before running `collect2` or `ld`, the driver overwrites two of the variables you may have set, from its own final prefix lists:

```
      /* Rebuild the COMPILER_PATH and LIBRARY_PATH environment variables
     for collect.  */
      putenv_from_prefixes (&exec_prefixes, "COMPILER_PATH", false);
      putenv_from_prefixes (&startfile_prefixes, LIBRARY_PATH_ENV, true);
```

Listing 6-4: propagation to subprocesses (`gcc/gcc.cc:9275-9278`).

So a child process sees the **resolved** lists — including everything contributed by `-B` and by the compiled-in defaults — not whatever you originally exported.

> **Reading `LIBRARY_PATH` inside a build script that GCC invoked tells you about the driver, not about your shell.** This bites in configure scripts and in `Makefile`s driven from a compiler wrapper, where the value looks like something you set and is not.

7.2 The `COLLECT_*` family: the driver talking to itself

Same channel, flowing the other way. You do not set these; you read them when debugging.

They exist because two of the driver's children have to **re-enter the driver**, as Chapter 1.1 described: `collect2` must compile the constructor tables it generates, and `lto-wrapper` must re-run code generation at link time with your original options.

7.2.1 `COLLECT_GCC` — which driver

The full pathname of the running driver. The comment above the code explains the one non-obvious choice:

```
/* Set up to remember the pathname of gcc and any options
   needed for collect.  We use argv[0] instead of progname because
   we need the complete pathname.  */

void
driver::putenv_COLLECT_GCC (const char *argv0) const
{
  obstack_init (&collect_obstack);
  obstack_grow (&collect_obstack, "COLLECT_GCC=", sizeof ("COLLECT_GCC=") - 1);
  obstack_grow (&collect_obstack, argv0, strlen (argv0) + 1);
```

Listing 6-5: exporting `COLLECT_GCC` (`gcc/gcc.cc:8710-8721`).

`progname` would give you `gcc`. `argv[0]` gives you the path you actually invoked, which is the only thing that identifies *this* installation out of the several you may have.

The two consumers disagree about what to do when it is missing, and the difference matters:

Consumer	Behaviour when unset
`lto-wrapper`	hard error: *"environment variable `COLLECT_GCC` must be set"*
`collect2`	falls back to `<target>-gcc`, then to plain `gcc`

Chapter 1.1 shows both code paths (`lto-wrapper.cc:1444-1448` and `collect2.cc:1142-1151`).

> **That `collect2` fallback is a genuine footgun on a cross toolchain.** A `collect2` invoked without `COLLECT_GCC` can silently pick up a *different* compiler — and therefore a different sysroot, different startfile prefixes, and a different `crtbegin.o`. The link succeeds. The binary is wrong. You will not get a diagnostic.

7.2.2 `COLLECT_GCC_OPTIONS` — with which switches

Every switch, and each of its arguments, emitted individually single-quoted:

```
COLLECT_GCC_OPTIONS='-B' '/opt/alt/' '--sysroot=/arm-rootfs' '-O2' '-dumpdir' 'a-'
```

The function that builds it names its own contract in its comment:

```
/* Store switches not filtered out by %<S in spec in COLLECT_GCC_OPTIONS
   and place that in the environment.  */

static void
set_collect_gcc_options (void)
```

Listing 6-6: `set_collect_gcc_options` (`gcc/gcc.cc:5610-5614`).

Three details follow from that, and all three surprise people.

It is not your command line. Switches removed by `%<S` in the specs are dropped from it — the filtering is explicit:

```
      /* Ignore elided switches.  */
      if ((switches[i].live_cond
       & (SWITCH_IGNORE | SWITCH_KEEP_FOR_GCC))
```

```
      == SWITCH_IGNORE)
    continue;
```

Listing 6-7: `%<S` filtering (`gcc/gcc.cc:5634-5638`).

It is rebuilt before every subprocess, not snapshotted at startup (`gcc.cc:5841` and `:6169`), so it reflects spec state at that moment.

`-dumpdir` appears even though you never typed it. So do other options synthesised along the way. If you are diffing two `COLLECT_GCC_OPTIONS` lines, expect content you did not write.

Embedded single quotes are escaped as `'\''`, in the loop just below Listing 6-6 — which is what makes the whole line safe to paste into a shell.

7.2.3 Why this belongs in a chapter about search paths

This is the mechanism that makes path flags **survive the recursion**. Put `-B`, `--sysroot` or `-L` on your command line and it lands in `COLLECT_GCC_OPTIONS`; when `collect2` re-invokes the driver with those options, the inner driver rebuilds the same prefix lists and the same sysroot as the outer one. Combine that with the `COMPILER_PATH` and `LIBRARY_PATH` rewrite of Listing 6-4, and the entire subprocess tree agrees about where things live.

Without it, a `-B` would apply to the compile and silently not to the link.

7.2.4 The rest of the family

Variable	Purpose	Set at
`COLLECT_LTO_WRAPPER`	path to `lto-wrapper`	`gcc.cc:8723-8726`
`COLLECT_AS_OPTIONS`	accumulated `-Xassembler` / `-Wa,` options	`gcc.cc:6094-6112`
`COLLECT_NO_DEMANGLE`	suppresses symbol demangling in `collect2`	**you**, by hand

`COLLECT_NO_DEMANGLE` is the odd one out: it is an input, read by `collect2` (`collect2.cc:895`) and then re-exported so that a nested invocation inherits it (`:898`). It is the only member of the family you would ever set deliberately. `COLLECT_LTO_WRAPPER` is likewise checked by `collect2` and turned into a hard error if absent (`collect2.cc:604-614`).

7.3 Reading them

`gcc -v` prints every variable the driver sets, because the setter itself echoes under `-v`:

```
void
env_manager::xput (const char *string)
{
  if (m_debug)
    fprintf (stderr, "env_manager::xput (%s)\n", string);
  if (verbose_flag)
    fnotice (stderr, "%s\n", string);
```

Listing 6-8: why `-v` echoes the environment (`gcc/gcc.cc:126-132`).

That is why `-v` output opens with `COLLECT_GCC=` and `COLLECT_LTO_WRAPPER=`, and prints a fresh `COLLECT_GCC_OPTIONS=` line immediately before every subprocess — one per rebuild, per Listing 6-7's neighbours.

```
$ gcc -v hello.c 2>&1 | grep '^COLLECT_'
```

Those lines are already shell-quoted, so they paste straight into a terminal. It is the fastest way to reproduce exactly what `collect2` was handed.

Setting them yourself is meaningful only if you are invoking `collect2` or `lto-wrapper` **directly**, which is occasionally a useful thing to do when debugging a link. Exporting them before a normal `gcc` run accomplishes nothing: the driver overwrites both before spawning anything.

7.4 Three things to carry away

`-B` outranks all of them. Two priority levels, and `-B` has the higher one.

`LIBRARY_PATH` does nothing on a cross compiler. One `*cross_compile == '0'` guard, and no diagnostic when it silently does not apply.

The `COLLECT_*` pair is why path flags survive the recursion into `collect2` and back — and why a `collect2` without them can quietly use the wrong compiler.

7.5 Documentation coverage

`GCC_EXEC_PREFIX`, `COMPILER_PATH` and `LIBRARY_PATH` are documented, in the Environment Variables node (`gcc/doc/invoke.texi:37984`, with the individual items at `:38057`, `:38092` and `:38099`).

What is not documented:

- The `COLLECT_*` family the driver *exports*. The Environment Variables node covers what you may set; the outward channel is source-only.
- `collect2`'s fallback to `<target>-gcc` and then `gcc` when `COLLECT_GCC` is unset — the one behaviour in this chapter most likely to cost you a day.
- That the driver overwrites `COMPILER_PATH` and `LIBRARY_PATH` before spawning the linker, so their values inside a child process are the driver's, not yours.
- That switches removed by `%<S` are absent from `COLLECT_GCC_OPTIONS`.

Next: why the directory in `libraries:` is not necessarily the directory that gets searched.

All source references in this chapter are to **GCC 15.2.0** (`releases/gcc-15.2.0`). Line numbers in other releases will differ; the surrounding code rarely does. Where behaviour itself changed across a major version, it is flagged inline.

8 One triple, many ABIs: multilib and multiarch

Source references in this chapter are to GCC 15.2.0 (`releases/gcc-15.2.0`).

You pass `-m32`. You supply no extra flags, no extra `-L`, nothing. And somehow the link picks up a 32-bit `libgcc.a` and a 32-bit `libc.so`, out of an installation whose default is 64-bit.

Then you pass `-mcpu=cortex-m4 -mfloat-abi=hard` on a bare-metal toolchain and the same magic fails: the link succeeds, and the resulting binary is built against the wrong float ABI.

One target triple can have several mutually incompatible ABIs, and multilib is how one toolchain ships a separate copy of every target library for each of them. The mechanism is a four-way expansion of every search prefix, plus one fallback pass that is the reason the second case fails quietly rather than loudly.

8.1 The problem, in the manual's words

> Write options that are mutually incompatible side by side, separated by a slash. Write options that may be used together separated by a space. The build procedure will build all combinations of compatible options.
>
> For example, if you set `MULTILIB_OPTIONS` to `m68000/m68020 msoft-float`, `Makefile` will build special versions of `libgcc.a` using the following sets of options: `-m68000`, `-m68020`, `-msoft-float`, `-m68000 -msoft-float`, and `-m68020 -msoft-float`.
>
> — `gcc/doc/fragments.texi:63-83`

Five variants of `libgcc.a` from two lines of makefile. And not just libgcc: `libstdc++`, `libatomic`, newlib's `libc.a`, every `crt*.o` — each has to be built and installed once per variant, and the driver has to pick the matching set on every single invocation.

You mostly do not notice this working. You notice when it does not.

8.2 Which variant am I getting?

Three commands, and the distinction between them is the whole chapter:

```
$ gcc -print-multi-lib                  # every variant this toolchain HAS
$ gcc -m32 -print-multi-directory       # GCC-side dir for THIS invocation
$ gcc -m32 -print-multi-os-directory    # OS-side dir for THIS invocation
$ gcc -m32 -print-file-name=libgcc.a    # which variant actually won
```

`-print-multi-lib` prints one line per variant, in a format the driver's own comment describes:

> The format of `multilib_select` is a list of elements. Each element is a subdirectory name followed by a list of options followed by a semicolon. [...] A subdirectory name is optionally followed by a colon and the corresponding multiarch name.
>
> — `gcc/gcc.cc:9750-9761`

so you get something shaped like:

```
.;
32;@m32
x32;@mx32
```

First field the directory, then the options that select it, `@`-separated.

> **If the variant you expect is missing from `-print-multi-lib`, the library was never built.** No amount of `-B` or `--sysroot` will conjure it into existence. You need a toolchain built with that variant, or you need to build it. This is the single most useful check in this chapter, and it is much faster than reasoning about paths.

A short `-print-multi-lib` usually means `--disable-multilib` at configure time, which `gcc -v` will show you.

8.3 Two directory names, not one

Here is the asymmetry that causes most of the confusion.

	GCC-side	OS-side
Applies to	directories inside GCC's install — `lib/gcc/<t>/<v>/<dir>/`	directories following OS convention — the sysroot's `/lib`, `/usr/lib`
Typical value	`32`	`../lib32`
Fragment variable	`MULTILIB_DIRNAMES`	`MULTILIB_OSDIRNAMES`
Driver variable	`multilib_dir`	`multilib_os_dir`
Print with	`-print-multi-directory`	`-print-multi-os-directory`

The driver keeps them as three separate variables with three separate comments:

```
/* Subdirectory to use for locating libraries.  Set by
   set_multilib_dir based on the compilation options.  */

static const char *multilib_dir;

/* Subdirectory to use for locating libraries in OS conventions.  Set by
   set_multilib_dir based on the compilation options.  */

static const char *multilib_os_dir;

/* Subdirectory to use for locating libraries in multiarch conventions.  Set by
   set_multilib_dir based on the compilation options.  */

static const char *multiarch_dir;
```

Listing 7-1: the three subdirectory variables (`gcc/gcc.cc:1666-1679`).

Why there are two — three, with multiarch — is worth stating plainly, because it is not arbitrary. GCC's own tree may name its variants however it likes; it owns that directory. But the *sysroot's* layout is dictated by whoever built the distribution: `/usr/lib32` on one, Debian's `/usr/lib/i386-linux-gnu` on another, `/usr/lib64` on a third. The target's build configuration is the mapping between GCC's convention and the OS's.

The practical consequence is that when you go hunting for a missing library by hand, you have to look in **both** shapes. `lib/gcc/<t>/<v>/32/` and `<sysroot>/usr/lib32/` are the same variant.

All three are chosen once per invocation by one function, `set_multilib_dir` (`gcc/gcc.cc:9764`), which matches your command-line switches against the `multilib` spec. And it is a spec — the variant table is a string in the spec table like any other (`gcc.cc:1733`), built at run time because it can be too long for some compilers' string limits (`gcc.cc:1326-1330`).

In spec text, `%M` expands to the OS-side name, with `.` as the no-variant case:

```
      case 'M':
        if (multilib_os_dir == NULL)
          obstack_1grow (&obstack, '.');
        else
          obstack_grow (&obstack, multilib_os_dir,
                strlen (multilib_os_dir));
```

Listing 7-2: `%M` (`gcc/gcc.cc:6805-6811`).

8.4 The four-way expansion

Now the mechanism. Every prefix on the list Chapter 1.5 built is tried up to four ways, by one function, in a fixed order. This is `for_each_path` (`gcc/gcc.cc:2778`), and each attempt is a labelled block:

```
      /* Look first in MACHINE/VERSION subdirectory.  */
      if (!skip_multi_dir)
        {
          memcpy (path + len, multi_suffix, suffix_len + 1);
```

Listing 7-3: pass one, machine and version plus `multilib_dir` (`gcc/gcc.cc:2838-2841`).

```
      /* Some paths are tried with just the machine (ie. target)
         subdir.  This is used for finding as, ld, etc.  */
      if (!skip_multi_dir
          && pl->require_machine_suffix == 2)
```

Listing 7-4: pass two, just the machine (`gcc/gcc.cc:2847-2850`).

```
      /* Now try the multiarch path.  */
      if (!skip_multi_dir
          && !pl->require_machine_suffix && multiarch_dir)
```

Listing 7-5: pass three, multiarch (`gcc/gcc.cc:2858-2860`).

```
      /* Now try the base path.  */
      if (!pl->require_machine_suffix
          && !(pl->os_multilib ? skip_multi_os_dir : skip_multi_dir))
        {
          const char *this_multi;
          ...
          if (pl->os_multilib)
        {
          this_multi = multi_os_dir;
```

Listing 7-6: pass four, the bare prefix plus `multilib_os_dir` (`gcc/gcc.cc:2868-2884`).

Read Listing 7-6 carefully, because it is where the two directory names finally diverge. The choice between `multi_os_dir` and `multi_dir` is made by `pl->os_multilib`, a per-prefix flag — the last argument to `add_prefix`. Look back at Listing 5-6 in Chapter 1.5: the tool directory was registered with a final argument of `1`. That is what marks it as an OS-convention prefix, and therefore what makes `../lib32` rather than `32` get appended to it.

So the expansion, concretely:

```
$prefix/lib/gcc/<target>/<ver>/32/     ← multilib_dir, on a GCC-convention prefix
<sysroot>/usr/lib/../lib32/            ← multilib_os_dir, on an OS-convention prefix
```

Yes, `/usr/lib/../lib32/` really is the shape of the path. `..` is not normalised away, which is why `-print-multi-os-directory` prints things like `../lib32` and why a strace of a link shows paths that look wrong and are not.

`require_machine_suffix == 2` in Listing 7-4 is the "just the machine" case, used for finding `as` and `ld` — which is how a toolchain finds `$prefix/<target>/bin/as` without a version component. It applies to `exec_prefixes`, not to libraries.

8.5 The fallback pass, and why a wrong link succeeds

The whole prefix list is walked **twice**: once with the multilib subdirectories, and once without.

```
      /* Run through the paths again, this time without multilibs.
     Don't repeat any we have already seen.  */
      if (multi_dir)
    {
      free (CONST_CAST (char *, multi_dir));
      multi_dir = NULL;
      ...
    }
```

```
      else
    skip_multi_dir = true;
```

Listing 7-7: the second pass (`gcc/gcc.cc:2902-2913`).

That second pass is a deliberate fallback. It lets a target ship only the default variant of some library rather than every combination — which is genuinely useful, because the combinatorial explosion of ARM's `-mcpu` and float-ABI axes is enormous.

It is also **why a link can succeed while silently picking up the wrong ABI's library.** The variant directory was empty, so the search fell through to the default one, and `ld` was handed an archive it was perfectly willing to read. Sometimes you get a diagnostic about incompatible attributes; on many targets you do not.

This is documented, and the manual states both halves:

> If the *osdir* part begins with a `!`, GCC will not search in the non-multilib directory and use exclusively the multilib directory. Otherwise, the compiler will examine the search path for libraries and crt files twice; the first time it will add *multilib* to each directory in the search path, the second it will not.
>
> — `gcc/doc/fragments.texi:205-210`

So a target that wants the fallback *off* — because falling through is worse than failing — prefixes its `MULTILIB_OSDIRNAMES` entry with `!`.

When you suspect this has happened, do not reason about directories. Ask which file:

```
$ gcc -mcpu=cortex-m4 -mfloat-abi=hard -print-file-name=libgcc.a
$ gcc -mcpu=cortex-m4 -mfloat-abi=hard -print-multi-directory
```

If the second prints a variant directory and the first returns a path that does not contain it, you have just watched the fallback happen.

8.6 Multiarch is a different thing

Debian's `/usr/lib/x86_64-linux-gnu` scheme puts one directory per triple so that several architectures' libraries coexist in one filesystem. It is **orthogonal to multilib** — a configuration can be multiarch without being multilibbed — but it shares the plumbing, and it gets its own pass in `for_each_path` (Listing 7-5).

The ordering is fixed and documented:

> Each multiarch subdirectory will be searched before the corresponding OS multilib directory, for example `/lib/i386-linux-gnu` before `/lib/../lib32`.
>
> — `gcc/doc/fragments.texi:218-220`

```
$ gcc -print-multiarch          # e.g. x86_64-linux-gnu
```

Note that this one prints an empty line rather than `.` when there is no multiarch directory (`gcc.cc:8867-8874`), unlike its two siblings.

8.7 Multilib and the sysroot: two independent schemes

Knowing which one your target uses matters, because they behave completely differently under `-print-sysroot`.

Scheme one, subdirectories inside one sysroot. The common case, and everything above. The OS-side directory is appended to paths within a single sysroot: one `/arm-rootfs`, containing `usr/lib` and `usr/lib32`.

Scheme two, a different sysroot per variant. Some targets compute a suffix on **the sysroot itself**, through `SYSROOT_SUFFIX_SPEC` (declared at `gcc.cc:1229`, with the default empty at `:1193-1195`). MIPS is the canonical user:

```
```

```
#undef SYSROOT_SUFFIX_SPEC
#define SYSROOT_SUFFIX_SPEC                        \
    "/%{mmicromips:micro}mips%{mel|EL:el}-" MIPS_SYSVERSION_SPEC    \
    "%{msoft-float:-soft;:-hard}"                    \
    "%{!mips32r6:%{!mips64r6:%{mnan=2008:-nan2008}}}%{muclibc:-uclibc}"
```

Listing 7-8: MIPS computing a sysroot suffix from your flags (`gcc/config/mips/mti-linux.h:29-33`).

That is ordinary spec text, using the if/else form from Chapter 1.4. So `--sysroot=/opt/rootfs -EL -msoft-float` really searches `/opt/rootfs/mipsel-r2-soft/…`. The suffix is evaluated once at startup (`gcc.cc:8542-8552`) and glued on inside `add_sysrooted_prefix` (`gcc.cc:3194-3196`) — the same function from Chapter 1.2, which turns out to concatenate three pieces rather than two.

Three consequences if you are on such a target:

- **`-print-sysroot` prints the suffixed path**, so its output changes with `-EL` or `-msoft-float`. Listing 3-5 in Chapter 1.3 is the code. **Do not cache it.**
- Your "sysroot" is not one root filesystem but a **family** of them, and `--with-sysroot` only gives the base. The suffix is invisible in the configure line, so `gcc -v` will not warn you.
- The same target also makes the *startfile prefix* multilib-conditional, through the `STARTFILE_PREFIX_SPEC` branch of Chapter 1.5:

```
#undef STARTFILE_PREFIX_SPEC
#define STARTFILE_PREFIX_SPEC                          \
  "%{mabi=32: /usr/local/lib/ /lib/ /usr/lib/}         \
   %{mabi=n32: /usr/local/lib32/ /lib32/ /usr/lib32/}  \
   %{mabi=64: /usr/local/lib64/ /lib64/ /usr/lib64/}"
```

Listing 7-9: MIPS replacing the standard prefix chain entirely (`gcc/config/mips/mti-linux.h:37-41`).

Header search gets the same treatment through `SYSROOT_HEADERS_SUFFIX_SPEC`, which MIPS simply defines as the same spec (`mti-linux.h:35`), processed at `gcc.cc:8568-8577`.

8.8 Declaring the variant set

If you are building a toolchain rather than using one, the variant set comes from your target's `t-*` makefile fragment. The variables, all documented in `fragments.texi`:

Variable	Declares	Documented
`MULTILIB_OPTIONS`	the option axes; `/` for mutually exclusive, space for combinable	`:64`
`MULTILIB_DIRNAMES`	the GCC-side directory name per variant	`:86`
`MULTILIB_MATCHES`	option spellings that map to the same variant	`:106`
`MULTILIB_EXCEPTIONS`	combinations not to build	`:114`
`MULTILIB_REQUIRED`	the whitelist form of the same idea	`:129`
`MULTILIB_REUSE`	serve an unbuilt option set from an existing variant	`:152`
`MULTILIB_EXTRA_OPTS`	options forced into every variant build	`:184`
`MULTILIB_OSDIRNAMES`	the OS-side name, optionally `:multiarch` after a colon	`:192`

A real one, and it is worth seeing the real text rather than a tidied version:

```
comma=,
MULTILIB_OPTIONS    = $(subst $(comma),/,$(TM_MULTILIB_CONFIG))
MULTILIB_DIRNAMES   = $(patsubst m%, %, $(subst /, ,$(MULTILIB_OPTIONS)))
MULTILIB_OSDIRNAMES = m64=../lib64$(call if_multiarch,:x86_64-linux-gnu)
MULTILIB_OSDIRNAMES+= m32=$(if $(wildcard $(shell echo
    $(SYSTEM_HEADER_DIR))/../../usr/lib32),../lib32,../lib)$(call if_multiarch,:i386-linux-gnu)
MULTILIB_OSDIRNAMES+= mx32=../libx32$(call if_multiarch,:x86_64-linux-gnux32)
```

Listing 7-10: x86-64 GNU/Linux (`gcc/config/i386/t-linux64:33-38`).

Three pieces of indirection in six lines, and each is instructive. The option list comes from `TM_MULTILIB_CONFIG`, which `config.gcc` sets from the triple — so `x86_64-linux-gnu` and `i686-linux-gnu` get different axes from the same fragment. `if_multiarch` emits the `:name` part only on a multiarch-enabled configuration, so the same line serves Debian and Fedora. And the `m32` line **probes the filesystem** at build time for `usr/lib32`, falling back to `../lib` when it is absent, with the reasoning in a comment above:

> On Debian, Ubuntu and other derivative distributions, the 32bit libraries are found in /lib32 and /usr/lib32, /lib64 and /usr/lib64 are symlinks to /lib and /usr/lib, while other distributions install libraries into /lib64 and /usr/lib64. The LSB does not enforce the use of /lib64 and /usr/lib64, it doesn't tell anything about the 32bit libraries on those systems.
>
> — `gcc/config/i386/t-linux64:19-24`

That fragment is compiled into the driver by a shell script, `genmultilib`, which generates `multilib.h` (`gcc/Makefile.in:2476-2483`) — and the driver includes it with a pointed comment about ordering:

```
#include "multilib.h" /* before tm.h */
```

Listing 7-11: (`gcc/gcc.cc:37`).

> **Prune multilibs in GCC and you prune them everywhere.** newlib, and anything else built in a combined tree, takes its variant matrix straight from `xgcc -print-multi-lib`. A `--disable-multilib` GCC produces a single-variant newlib whether you asked for that or not. Part III returns to this.

8.9 What to take from this

`multilib_dir` and `multilib_os_dir` are different things with different roles, and which one a prefix gets depends on a flag set when the prefix was registered. Every prefix is tried four ways, then the whole list is tried again with no variant at all — and that last fallback is why a wrong-ABI link can succeed silently.

Before debugging paths, check that the variant exists:

```
$ gcc -print-multi-lib
```

8.10 Documentation coverage

This is the best-documented mechanism in Part I. `fragments.texi` covers the fragment variables thoroughly, including the two-pass walk (`:205-210`), the `!` prefix that disables it, the multiarch ordering (`:218-220`) and the fact that `MULTILIB_OSDIRNAMES` subsumes `MULTIARCH_DIRNAME` (`:212-216`). The `-print-multi-*` options are in Overall Options.

What is not documented is the consequence rather than the mechanism: that the fallback pass turns a missing variant into a wrong-library link rather than an error, and that `-print-search-dirs` shows the list before any of this expansion happens, so its output is not the set of directories actually searched.

Next: what all these paths are searched *for*.

All source references in this chapter are to **GCC 15.2.0** (`releases/gcc-15.2.0`). Line numbers in other releases will differ; the surrounding code rarely does. Where behaviour itself changed across a major version, it is flagged inline.

9 What actually reaches the linker

Source references in this chapter are to GCC 15.2.0 (`releases/gcc-15.2.0`).

You have looked at a `-v` link line before, and you have probably wondered about two things in it. There are five `crt`-something object files, interleaved around your own objects in an order nobody explained. And `-lgcc` appears twice.

Neither is an accident, and neither is arbitrary. This chapter is the canonical account of what reaches the linker, in what order, and who supplies each piece.

9.1 The shape of a link

For a glibc target, the objects come out in this order:

```
crt1.o   crti.o   crtbegin.o   …your objects…   crtend.o   crtn.o
└──────  libc  ─────┘└ libgcc ┘                 └ libgcc ┘  └ libc ┘
```

and then the libraries:

```
-lgcc   -lc   -lgcc
```

Two facts about that picture explain most link problems you will ever have with GCC:

1. **The startup files come from two unrelated providers**, interleaved.
2. **libgcc appears twice**, on purpose.

See it for yourself, on your own toolchain:

```
$ gcc -### hello.c 2>&1 | grep collect2
```

Grep for `collect2`, not `ld` — Chapter 1.1 explains why.

9.2 It is one template

Before the pieces, the thing they sit in. The entire link is a single spec, and its structure *is* the link order:

```
%{!fsyntax-only:%{!c:%{!M:%{!MM:%{!E:%{!S:          ← only when actually linking
    %(linker)                                        ← collect2 (or ld)
    %{flto} %{fno-lto} %{flto=*} %l                  ← target link flags
    %X %{o*} %{e*} …                                 ← -Wl, options, -o, -e …
    %{!nostdlib:%{!r:%{!nostartfiles:%S}}}           ← crt1.o crti.o crtbegin.o
    %{static|no-pie|static-pie:} %@{L*} %(link_libgcc)
    %o                                               ← YOUR object files
    %{!nostdlib:%{!r:%{!nodefaultlibs:
        %(link_ssp) %(link_gcc_c_sequence)}}}        ← libgcc + libc + libgcc
    %{!nostdlib:%{!r:%{!nostartfiles:%E}}}           ← crtend.o crtn.o
    %{T*}  \n%(post_link) }}}}}}
```

Listing 8-1: `LINK_COMMAND_SPEC`, abridged (`gcc/gcc.cc:1159-1178`).

Three observations, one of which is the whole reason this chapter can be short.

Position in the template is position on the command line. `%S` … `%o` … libraries … `%E` *is* the link order. The startfile/endfile split exists for no other reason than that `%S` and `%E` sit on opposite sides of `%o`.

`-nostdlib`, `-nostartfiles`, `-nodefaultlibs` and `-r` are pure conditionals. No C code implements them; they fail a `%{!…:}` guard and the text simply is not emitted. The three guards live at `gcc.cc:1168`, `:1176` and `:1177`, and Chapter 1.10 reads the truth table straight off the nesting.

A target customises one small sub-spec, not this string. Nobody copies Listing 8-1.

The `%(link_libgcc)` reference just before your objects is worth a note, since it is where all of Chapter 1.5's work shows up:

```
#ifndef LINK_LIBGCC_SPEC
/* Generate -L options for startfile prefix list.  */
# define LINK_LIBGCC_SPEC "%D"
#endif
```

Listing 8-2: where the `-L` flags come from (`gcc/gcc.cc:1180-1183`).

One `%D`. That is the entire connection between the driver's prefix list and the linker's search path — one directive, expanding to one `-L` per directory.

9.3 Two providers, one search list

Files	Provider	Installed under	Moved by
`crt1.o`, `Scrt1.o`, `gcrt1.o`, `crti.o`, `crtn.o`	**the C library**	the sysroot's `/usr/lib`	`--with-sysroot`, `--sysroot`
`crtbegin*.o`, `crtend*.o`	**GCC's libgcc**	`$prefix/lib/gcc/<t>/<v>/`	`--prefix`, `-B`

Both halves are written identically in the spec, because `%s` means nothing more than "look this name up in the library search list" (Listing 4-9). One list, two origins, and the difference invisible in the spec text.

The libc half is found because `/lib/` and `/usr/lib/` are registered with `add_sysrooted_prefix` — Listing 5-9 in Chapter 1.5. **That call is where the sysroot gets glued on.** The libgcc half is found through prefixes derived from the driver's own location and is never sysrooted.

So the diagnosis is one pair of commands:

```
$ gcc -print-file-name=crt1.o       # the libc half
$ gcc -print-file-name=crtbegin.o   # the libgcc half
```

Echoed back verbatim	Means
`crt1.o`	your **sysroot** is wrong
`crtbegin.o`	your **prefix or `-B`** is wrong

9.3.1 What a real startfile spec looks like

The *default* `STARTFILE_SPEC` in the driver is not the crt1/crti/crtbegin shape at all:

```
/* config.h can define STARTFILE_SPEC to override the default crt0 files.  */
#ifndef STARTFILE_SPEC
#define STARTFILE_SPEC  \
  "%{!shared:%{pg:gcrt0%O%s}%{!pg:%{p:mcrt0%O%s}%{!p:crt0%O%s}}}"
#endif
```

Listing 8-3: the default `STARTFILE_SPEC` (`gcc/gcc.cc:891-895`). Note `crt0`, and note that `ENDFILE_SPEC` defaults to the empty string at `:897-900`.

That default is a historical fallback almost nobody uses. **Do not repeat the claim that `STARTFILE_SPEC` supplies `crt1.o crti.o crtbegin.o`** — that is one target family's override. Here is the real one, for GNU userspace:

```
#define GNU_USER_TARGET_STARTFILE_SPEC \
  "%{shared:; \
     pg|p|profile:%{static-pie:grcrt1.o%s;:gcrt1.o%s}; \
     static:crt1.o%s; \
     static-pie:rcrt1.o%s; \
     " PIE_SPEC ":Scrt1.o%s; \
     :crt1.o%s} " \
   GNU_USER_TARGET_CRTI " \
   %{static:crtbeginT.o%s; \
```

```
    shared|static-pie|" PIE_SPEC ":crtbeginS.o%s; \
    :crtbegin.o%s} \
  ...
```

Listing 8-4: the glibc startfile spec (`gcc/config/gnu-user.h:51-65`, installed over the default at `:66-67`).

This is worth reading slowly, because it is the best example in the tree of the if/else-if/else form from Chapter 1.4. Six arms select *one* `crt1` variant:

You passed	You get
`-shared`	nothing at all (the empty first arm)
`-pg`, `-p` or `-profile`	`gcrt1.o`, or `grcrt1.o` with `-static-pie`
`-static`	`crt1.o`
`-static-pie`	`rcrt1.o`
whatever `PIE_SPEC` matches	`Scrt1.o`
otherwise	`crt1.o`

Then a second, independent selection picks `crtbeginT.o`, `crtbeginS.o` or `crtbegin.o`. Which is why `gcc -static -### hello.c` and `gcc -shared -### hello.c` produce link lines that share almost no startup objects, and why "the crt1 file" is not a well-defined phrase.

The `crti.o` and `crtn.o` in the middle come from two one-line macros:

```
#define GNU_USER_TARGET_CRTI "crti.o%s"
#define GNU_USER_TARGET_CRTN "crtn.o%s"
```

Listing 8-5: (`gcc/config/gnu-user.h:43-44`).

and the matching `ENDFILE_SPEC` mirrors the whole structure at `gnu-user.h:75-85`.

> **Do not assume `crti.o` comes from the C library.** On many bare-metal targets libgcc supplies it. Which `crt*` files a given target's libgcc provides is declared in `libgcc/config.host` as `extra_parts` — for example, `arm*-*-eabi*` gets `extra_parts="crtbegin.o crtend.o crti.o crtn.o"` (`libgcc/config.host:584`). On such a target, a missing `crti.o` means your *prefix* is wrong, not your sysroot, and the table above does not apply. Check what your libgcc installed before blaming anything.

Those `extra_parts` become `EXTRA_PARTS` in libgcc's makefile (`libgcc/Makefile.in:59-61`) and get installed into

```
inst_libdir = $(libsubdir)$(MULTISUBDIR)
```

Listing 8-6: where libgcc's objects land (`libgcc/Makefile.in:315`; the install rule is `install-leaf` at `:1188`).

which is libsubdir plus the multilib subdirectory from Chapter 1.7. Note that libgcc computes `libsubdir` from `real_host_noncanonical` (`libgcc/Makefile.in:204`) rather than from a target variable — because libgcc is configured with `--host=<GCC's target>`. Whenever a path formula in a *target library* says "host", read "GCC's target". That convention will bite you again in Chapter 1.12.

9.4 Why `-lgcc` is there twice

This is the one everybody eventually has to debug. The sequence is:

```
-lgcc   -lc   -lgcc
 |       |     └── libgcc AGAIN, after libc
 |       └──────── libc
 └──────────────── libgcc
```

And it is one spec:

```
/* This is overridable by the target in case they need to specify the
   -lgcc and -lc order specially, yet not require them to override all
   of LINK_COMMAND_SPEC.  */
#ifndef LINK_GCC_C_SEQUENCE_SPEC
#define LINK_GCC_C_SEQUENCE_SPEC "%G %{!nolibc:%L %G}"
#endif
```

Listing 8-7: the double libgcc (`gcc/gcc.cc:987-992`). `%G` is the `libgcc` spec, `%L` the `lib` spec.

Why: libc can call *into* libgcc — `__divdi3` on a target without a division instruction, the unwinder, soft-float helpers. With static archives the linker resolves left to right and only pulls in the members it needs *at the moment it reads the archive*. So a reference created by `libc.a` after `libgcc.a` had already been scanned would go unresolved. Repeating libgcc after libc closes that.

`-nolibc` drops the middle pair and keeps the leading `-lgcc`, which you can read directly off the spec: the `%{!nolibc:…}` wraps `%L %G` and not the first `%G`.

The `lib` spec on the other side of it has a default worth knowing:

```
/* config.h can define LIB_SPEC to override the default libraries.  */
#ifndef LIB_SPEC
#define LIB_SPEC "%{!shared:%{g*:-lg} %{!p:%{!pg:-lc}}%{p:-lc_p}%{pg:-lc_p}}"
#endif
```

Listing 8-8: the default `LIB_SPEC` (`gcc/gcc.cc:754-757`).

So `-lc` is the default, `-lc_p` under profiling, and nothing at all when building a shared object. Most targets override this in their `config/*.h`.

9.4.1 What libgcc actually is

GCC's own runtime support library. The compiler emits calls into it whether you asked for it or not:

- integer division and modulo helpers on targets without the instruction — `__divdi3`, `__udivsi3`
- ARM EABI helpers — the `__aeabi_*` family
- soft-float emulation
- the stack unwinder used by C++ exceptions and by `__attribute__((cleanup))`

> **This is why you still need `-lgcc` under `-nostdlib`.** Dropping the standard libraries does not stop the compiler emitting a call to `__aeabi_idiv`. A bare-metal link failing on an undefined `__aeabi_*` or `__udivsi3` is the single most common consequence, and Chapter 1.10 covers what you owe once you have taken the runtime away.

And note what `%G` expands to is not simply `-lgcc`. On a shared-libgcc target the slot was rewritten at startup — Listing 4-5 — into the nested conditional that chooses between `-lgcc -lgcc_eh`, `-lgcc --as-needed -lgcc_s --no-as-needed` and `-lgcc_s -lgcc` depending on `-static`, `-static-libgcc` and `-shared-libgcc`. Which is why:

```
$ gcc -### foo.o 2>&1 | tr ' ' '\n' | grep lgcc
$ gcc -### -static foo.o 2>&1 | tr ' ' '\n' | grep lgcc
```

give you different answers on the same toolchain.

9.5 Watching a flag take effect

The fastest way to internalise Listing 8-1 is to diff its output against itself:

```
$ gcc -### hello.c                    # baseline
$ gcc -### -nostdlib hello.c          # crt*.o and -lc/-lgcc gone
$ gcc -### -nostartfiles hello.c      # crt*.o gone, -lc/-lgcc kept
$ gcc -### -nodefaultlibs hello.c     # crt*.o kept, -lc/-lgcc gone
$ gcc -dumpspecs | grep -A2 '^\*link_command'
```

Each of those four differences is one `%{!…:}` guard failing.

9.6 Things that surprise people

`g++` reorders your `-lm` and `-lc`. They are hoisted out of wherever you put them and re-emitted *after* `-lstdc++`. This is a real, silent reordering of your command line, and on a target where everything is a static archive it changes symbol resolution. Chapter 1.11.

`-r` behaves exactly like `-nostdlib` as far as these guards are concerned. A relocatable link never gets the runtime — read the `%{!r:` in each of the three guards.

`-static-libstdc++` does not appear in the link line on most targets. It is consumed by the C++ driver and turned into `-Wl,-Bstatic` … `-Wl,-Bdynamic` around one `-l`. Grep for `Bstatic`, not for the option. Chapter 1.11.

`ENDFILE_SPEC` is empty by default (Listing 8-3). If `crtend.o` is not on your link line, that may be entirely correct for your target.

A successful link says nothing about run time. Finding `libstdc++.so.6` or `libc.so.6` when the program starts is the *target loader's* job — `DT_RUNPATH`, `ld.so.conf`, `LD_LIBRARY_PATH` — and nothing in this chapter, or in this whole part of the book, touches it. Link-time search and run-time search are unrelated systems that happen to involve some of the same filenames.

9.7 Documentation coverage

`-nostdlib` and friends are documented under Link Options (`gcc/doc/invoke.texi:19067`), and libgcc has its own chapter in the internals manual.

What is not documented:

- **Why `-lgcc` appears twice.** The reason exists only as reasoning you have to reconstruct; the spec is there, the rationale is not. The nearest thing is the comment in Listing 8-7, which explains why the ordering is *overridable* rather than why it is what it is.
- **That `STARTFILE_SPEC`'s default is crt0-based** and that the familiar crt1/crti/crtbegin shape is a per-target-family override. Every secondhand account of GCC's link order presents one target's spec as universal.
- **That `crti.o` may come from libgcc rather than libc**, which inverts the diagnosis table above on bare-metal targets.

Next: the same question for headers.

All source references in this chapter are to **GCC 15.2.0** (`releases/gcc-15.2.0`). Line numbers in other releases will differ; the surrounding code rarely does. Where behaviour itself changed across a major version, it is flagged inline.

10 Where `<stdint.h>` actually comes from

Source references in this chapter are to GCC 15.2.0 (`releases/gcc-15.2.0`).

You are building a sysroot for a cross target. You have copied in glibc's headers, you have `stdio.h` and `pthread.h` and `fcntl.h`, and you check for `stdint.h` and it is not there. You go looking for the glibc build step that produces it, and there isn't one.

`<stdint.h>` is not the C library's. It is the compiler's, and it lives under GCC's own prefix, in `$prefix/lib/gcc/<target>/<version>/include/`. So do `<stddef.h>`, `<limits.h>`, `<stdarg.h>` and about a dozen others.

```
$ gcc -print-file-name=include
$ ls "$(gcc -print-file-name=include)"
```

This chapter is the compile-time twin of Chapter 1.8: one chapter for what reaches the linker, one for where headers come from. The sysroot rule from Chapter 1.2 — the sysroot belongs to the C library, the prefix belongs to GCC — applies here without exception, and this is the case where it surprises people most.

10.1 Why the compiler *has* to provide them

The C standard defines two kinds of execution environment:

Environment	Means	`__STDC_HOSTED__`
Hosted	a full OS with a complete C library — files, threads, `<stdio.h>`	`1`
Freestanding	bare metal, kernels, early boot — no OS, maybe no libc at all	`0`

A conforming implementation must work in a freestanding environment, where by definition there is no C library to supply headers. So the standard requires the *compiler itself* to provide a small core set — the freestanding headers.

Which is why these headers exist at all as a separate category: they must be available when nothing else is.

The sources are in the tree, under `gcc/ginclude/`, and the list of which ones get installed is a makefile variable:

```
USER_H = $(srcdir)/ginclude/float.h \
     $(srcdir)/ginclude/iso646.h \
     $(srcdir)/ginclude/stdarg.h \
     $(srcdir)/ginclude/stdbool.h \
     $(srcdir)/ginclude/stddef.h \
     $(srcdir)/ginclude/varargs.h \
     $(srcdir)/ginclude/stdfix.h \
     $(srcdir)/ginclude/stdnoreturn.h \
     $(srcdir)/ginclude/stdalign.h \
     $(srcdir)/ginclude/stdatomic.h \
     $(srcdir)/ginclude/stdckdint.h \
     $(EXTRA_HEADERS)
```

Listing 9-1: the headers GCC installs (`gcc/Makefile.in:473-484`).

Note `$(EXTRA_HEADERS)` on the last line: that is where a target adds its own, which is how `<arm_neon.h>` and `<immintrin.h>` get there.

What each of them contributes, and why a libc could not:

Header	Provides	Why it must be the compiler's
`<stddef.h>`	`size_t`, `ptrdiff_t`, `NULL`, `offsetof`, `wchar_t`	These *are* compiler properties; `offsetof` is `__builtin_offsetof`
`<stdarg.h>`	`va_list`, `va_start`, `va_arg`, `va_end`	Expands to `__builtin_va_*`; the varargs ABI is generated code, not a library
`<stdint.h>`	`int32_t`, `uint64_t`, `INTPTR_MAX`, …	Must match the target data model exactly — ILP32, LP64, LLP64
`<limits.h>`	`INT_MAX`, `CHAR_BIT`, `LONG_MIN`	Derived from the target's integer widths
`<float.h>`	`FLT_MANT_DIG`, `DBL_MAX`, …	Derived from the target's floating-point formats
`<stdbool.h>`	`bool`, `true`, `false`	Maps to the built-in `_Bool`
`<stdalign.h>`	`alignas`, `alignof`	Maps to `_Alignas` / `_Alignof`
`<stdatomic.h>`	`atomic_int`, `atomic_load`, …	Atomics are codegen plus libatomic, not libc
`<iso646.h>`	`and`, `or`, `not`, …	Pure macros; nothing for a libc to contribute
arch intrinsics	`<arm_neon.h>`, `<immintrin.h>`, …	One-to-one mappings onto instructions the back end knows

The pattern is consistent: **every one of these encodes something only the compiler knows** — type widths, alignment rules, calling conventions, instruction sets. A libc physically cannot get them right for an arbitrary compiler.

> **If you are on GCC 13 or earlier, `<stdckdint.h>` is not there.** C23's checked integer arithmetic header arrived in GCC 14; `gcc/ginclude/stdckdint.h` does not exist at `releases/gcc-13.3.0` and does at `releases/gcc-14.3.0`.
>
> While you are checking that list against your own compiler, note that `<stdbit.h>` is **not** one of GCC's headers, despite being C23 and despite what you may read. At the pin it exists only as a libstdc++ C-compatibility header (`libstdc++-v3/include/c_compatibility/stdbit.h`), not in `gcc/ginclude/`.

In C23, `bool`, `alignas`, `alignof` and `noreturn` became keywords, so `<stdbool.h>`, `<stdalign.h>` and `<stdnoreturn.h>` are now largely vestigial. They still exist, and still come from GCC. As always: run `ls` on your own `-print-file-name=include` rather than trusting any list, including Listing 9-1.

10.2 `include/` versus `include-fixed/`

Both sit in libsubdir, and both come before the sysroot in the search order:

Directory	Contains
`include/`	GCC's **own** headers — everything above
`include-fixed/`	**patched copies of the system's headers**

`include-fixed/` is the output of **fixincludes**, a build-time pass that copies system headers found at build time and mechanically repairs constructs GCC dislikes — ancient K&R declarations, macros that break under a modern preprocessor. The machinery lives in `fixincludes/`, and its README describes the workflow, which is a rule database compiled by AutoGen:

> If you are having some problem with a system header that is either broken by the manufacturer, or is broken by the fixinclude process, then you will need to alter or add information to the include fix definitions file, `inclhack.def`.

— `fixincludes/README:5-8`

Two consequences worth knowing.

`include-fixed/` is a snapshot taken when GCC was built. If your sysroot's headers were updated afterwards, the fixed copies can be stale, and you will be compiling against a patched version of a header that no longer exists.

On a modern glibc or musl target it is usually near-empty. The machinery is mostly historical; on older or unusual targets it can matter a great deal.

The driver knows about it as a separate directory in the compiled-in list, and the entry has its own multilib behaviour:

```
#ifdef FIXED_INCLUDE_DIR
    /* This is the dir for fixincludes.  */
#ifndef SYSROOT_HEADERS_SUFFIX_SPEC
    { FIXED_INCLUDE_DIR, "GCC", 0, 0, 0, 2 },
#endif
    { FIXED_INCLUDE_DIR, "GCC", 0, 0, 0,
      /* A multilib suffix needs adding if different multilibs use
     different headers.  */
```

Listing 9-2: the `include-fixed` entries (`gcc/cppdefault.cc:75-83`).

The trailing `2` is the `multilib` field, meaning "append the *multiarch* path" — the same three-way distinction from Chapter 1.7, reaching into header search.

10.3 The search order, and why GCC's copy wins

```
$ gcc -v -E - < /dev/null
```

The `<...>` section looks like this, in this order:

```
#include <...> search starts here:
 /opt/gcc-arm/lib/gcc/arm-none-eabi/15.2.0/include        ← GCC's own
 /opt/gcc-arm/lib/gcc/arm-none-eabi/15.2.0/include-fixed  ← fixincludes output
 /opt/gcc-arm/arm-none-eabi/include                       ← tooldir
 <sysroot>/usr/include                                    ← the C library's
End of search list.
```

The order comes from one compiled-in array, `cpp_include_defaults` (`gcc/cppdefault.cc:38`), and it is *literally* an array — the search order is the declaration order, from the C++ header directories at the top down to a final entry the source comments on with admirable brevity:

```
#ifdef NATIVE_SYSTEM_HEADER_DIR
    /* /usr/include comes dead last.  */
    { NATIVE_SYSTEM_HEADER_DIR, NATIVE_SYSTEM_HEADER_COMPONENT, 0, 0, 1, 2 },
    { NATIVE_SYSTEM_HEADER_DIR, NATIVE_SYSTEM_HEADER_COMPONENT, 0, 0, 1, 0 },
#endif
```

Listing 9-3: the last entries in the list (`gcc/cppdefault.cc:98-101`).

The macros that populate the interesting entries are `#define`d into the compiler by its makefile:

Macro	Directory
`GCC_INCLUDE_DIR`	`$libsubdir/include` — GCC's own
`FIXED_INCLUDE_DIR`	`$libsubdir/include-fixed`
`TOOL_INCLUDE_DIR`	`$tooldir/include` — `gcc/Makefile.in:2613`
`NATIVE_SYSTEM_HEADER_DIR`	`/usr/include`, sysrooted
`GPLUSPLUS_INCLUDE_DIR`	the C++ headers — Chapter 1.12

`TOOL_INCLUDE_DIR` is a survivor of the pre-sysroot `--with-headers` mechanism from Chapter 1.2. It is still searched on a pure-sysroot toolchain that never asked for it.

GCC's directories come first, and that ordering is the whole basis of the next section: when you write `#include <limits.h>`, you get GCC's copy, and GCC's copy then decides whether to bring libc's in as well.

10.3.1 The sysroot decision is per entry

Each entry in the array carries its own flag saying whether the sysroot applies:

```
  const char add_sysroot;   /* FNAME should be prefixed by
                               cpp_SYSROOT.  */
```

Listing 9-4: the field (`gcc/cppdefault.h:48-49`).

and the consuming code branches on it:

```
      /* Should this directory start with the sysroot?  */
      if (sysroot && p->add_sysroot)
        {
          ...
          str = concat (sysroot_no_trailing_dir_separator, p->fname, NULL);
          ...
        }
      else if (!p->add_sysroot && relocated
           && !filename_ncmp (p->fname, cpp_PREFIX, cpp_PREFIX_len))
        {
```

Listing 9-5: sysroot or relocate, per directory (`gcc/incpath.cc:181-193`).

Read the `else if`. **GCC's own header directories take that branch**: not sysrooted, but *relocatable* — if the path starts with the configure-time prefix and the compiler has been relocated, the run-time prefix is substituted instead. That is the mechanism behind "they are GCC's, not libc's", stated in code, and it is the consumer of the `set_std_prefix` call you saw in Listing 6-2.

This is exactly parallel to `add_prefix` versus `add_sysrooted_prefix` for libraries, and it is a completely separate implementation. Header search and library search are two mechanisms that happen to be relocated by the same prefix.

10.4 `#include_next`, the wrapper trick

Some of these headers are not replacements but **wrappers**. They need to add compiler knowledge *and* let the C library contribute its own definitions. The GNU extension that makes that possible:

> `#include <foo.h>` searches the path **from the beginning**. `#include_next <foo.h>` searches **from just after the directory the current file was found in**.

So a header can include the next file of the same name further down the search path — itself, one layer down — without knowing where that layer is and without risk of including itself.

`<stdint.h>` is the clearest case, because the whole hosted/freestanding split is right there in the shipped source:

```
#ifndef _GCC_WRAP_STDINT_H
#if __STDC_HOSTED__
...
# include_next <stdint.h>
...
#else
# include "stdint-gcc.h"
#endif
#define _GCC_WRAP_STDINT_H
#endif
```

Listing 9-6: `stdint-wrap.h` (`gcc/ginclude/stdint-wrap.h`, abridged — the omitted lines define `__STDC_LIMIT_MACROS` for C++11 and silence a `-Wpedantic` warning about `include_next`).

In a **hosted** build GCC's copy defers to the system's. In a **freestanding** build there may be no system copy at all, so it falls back to a self-contained version GCC ships alongside, `stdint-gcc.h`. That is precisely the freestanding requirement being honoured, in eight lines.

Which of the two gets installed as `include/stdint.h` is a build-time decision:

```
    if [ $(USE_GCC_STDINT) = wrap ]; then \
      cp $(srcdir)/ginclude/stdint-wrap.h include/stdint.h; \
      ...
    elif [ $(USE_GCC_STDINT) = provide ]; then \
      cp $(T_STDINT_GCC_H) include/stdint.h; \
```

Listing 9-7: wrap or provide (`gcc/Makefile.in:3514-3524`; `USE_GCC_STDINT` comes from configure, `:816`).

10.4.1 `<limits.h>` is assembled, not shipped

`<limits.h>` works the same way but is stranger: there is no single `gcc/limits.h` in the tree to read, because the installed file is **concatenated at build time from three fragments**:

```
      if $(LIMITS_H_TEST) ; then \
        cat $(srcdir)/limitx.h $(T_GLIMITS_H) $(srcdir)/limity.h > tmp-xlimits.h; \
      else \
        cat $(T_GLIMITS_H) > tmp-xlimits.h; \
      fi; \
```

Listing 9-8: building `limits.h` (`gcc/Makefile.in:3528-3532`).

and the test is one line:

```
LIMITS_H_TEST = [ -f $(BUILD_SYSTEM_HEADER_DIR)/limits.h ]
```

Listing 9-9: (`gcc/Makefile.in:591`).

So: if the system has its own `limits.h`, you get the wrapper form — prologue, GCC's own limits, epilogue. If it does not, you get GCC's limits alone with no `#include_next` at all. The fragments say so themselves:

```
/* This administrivia gets added to the beginning of limits.h
   if the system has its own version of limits.h.  */
```

Listing 9-10: (`gcc/limitx.h:24-25`; the matching `gcc/limity.h:1-2` says "the end of limits.h".)

There is one more hop, and it is a nice piece of indirection. `limitx.h` does not use `#include_next` directly:

```
/* Use "..." so that we find syslimits.h only in this same directory.  */
#include "syslimits.h"
```

Listing 9-11: (`gcc/limitx.h:33-34`).

and `syslimits.h` is a copy of `gsyslimits.h` (`gcc/Makefile.in:3540`) whose entire body is:

```
#define _GCC_NEXT_LIMITS_H      /* tell gcc's limits.h to recurse */
#pragma GCC diagnostic push
#pragma GCC diagnostic ignored "-Wpedantic" // include_next
#include_next <limits.h>
#pragma GCC diagnostic pop
#undef _GCC_NEXT_LIMITS_H
```

Listing 9-12: `gsyslimits.h` (`gcc/gsyslimits.h:6-11`).

The reason for the extra file is that `include-fixed/` may contain a *patched* `syslimits.h` instead — that is one of fixincludes' jobs — so the hop gives the build a place to intervene without touching `limits.h` itself.

10.4.2 Watching it happen

```
$ echo '#include <limits.h>' | gcc -H -E -x c - > /dev/null
```

`-H` prints the include tree, one dot of indentation per nesting level. You will see GCC's `limits.h` at depth one, `syslimits.h` below it, and the sysroot's `limits.h` below that — the hop, visible.

Which headers are wrappers and which are standalone varies by target, so run `-H` rather than trusting a table. Roughly: `<stddef.h>`, `<stdarg.h>`, `<stdbool.h>`, `<iso646.h>` and the arch intrinsics are standalone; `<limits.h>` and `<stdint.h>` are wrappers when the system has its own.

10.5 Gotchas

Do not go looking in the sysroot for `stdint.h`. It is not there and it is not supposed to be. `gcc -print-file-name=include` is where to look.

Copying a sysroot does not copy these. A sysroot is complete for libc and useless on its own; the compiler's own headers travel with the *compiler*.

They are version-locked. The path contains `<version>` (Listing 2-2), so two GCC releases have separate copies. Mixing GCC 12's headers with GCC 15's `cc1` is a broken toolchain, not merely an unwise one.

`-nostdinc` removes these too, not only the system directories (`gcc/c-family/c.opt:2523`). On bare metal you almost always want them back:

```
$ gcc -nostdinc -isystem "$(gcc -print-file-name=include)" …
```

`-ffreestanding` changes nothing about header search. This is the pairing people most reliably get backwards. Its whole help text is:

```
ffreestanding
C ObjC C++ ObjC++
Do not assume that standard C libraries and "main" exist.
```

Listing 9-13: (`gcc/c-family/c.opt:1986-1988`).

It changes what the compiler may *assume* — it stops `printf("x\n")` becoming `puts("x")`, relaxes the requirements on `main`, and sets `__STDC_HOSTED__` to `0`. It moves no directory. Chapter 1.10 puts it side by side with `-nostdlib` and `-nostdinc`, which are the two options it is most often confused with.

```
$ gcc -dM -E - < /dev/null | grep __STDC_HOSTED__
$ gcc -ffreestanding -dM -E - < /dev/null | grep __STDC_HOSTED__
```

`-B` moves them. A `-B<dir>` adds `<dir>/<target>/<version>/include` and `.../include-fixed` to the header search, because `-B` feeds `include_prefixes` (Listing 5-2) and `%I` expands that list:

```
          info.option = "-isystem";
          info.append = "include";
          ...
          for_each_path (&include_prefixes, false, info.append_len,
                 spec_path, &info);
```

Listing 9-14: `%I` emitting GCC's header directories (`gcc/gcc.cc:6604-6612`).

That is why `-B` can change which `<stdint.h>` you compile against. It is also exactly why it cannot change which `<vector>` you get: only GCC's own two directories are in `include_prefixes`.

`include-fixed/` can be stale relative to an updated sysroot.

10.6 Checking

```
$ gcc -print-file-name=include                                  # GCC's own header dir
$ ls "$(gcc -print-file-name=include)"                           # what it actually ships
$ gcc -v -E - < /dev/null                                        # the full search list
$ echo '#include <limits.h>' | gcc -H -E -x c - >/dev/null    # the include_next chain
$ echo '#include <stdint.h>' | gcc -E -x c - | grep -m1 stdint
```

The last one tells you which file a specific include actually resolved to, which is occasionally the only question you have.

10.7 Documentation coverage

`-nostdinc`, `-ffreestanding` and `-I` / `-isystem` / `-idirafter` are documented under Directory Options (`gcc/doc/invoke.texi:19484`) and in the preprocessor manual, which also covers `#include_next` and `-H`.

What is not documented:

- **That `<limits.h>` is assembled from three fragments at build time**, and that which form you get depends on a filesystem test. The `syslimits.h` hop in particular exists only in the source.
- **That the sysroot decision is per include-directory, carried in a struct field**, and therefore not something a flag can change after the fact.
- **That GCC's own header directories are relocated rather than sysrooted** — the `else if` in Listing 9-5, which is the precise statement of why `--sysroot` cannot move them.

That closes Part I's account of search paths. The remaining three chapters of Part I are about deliberately taking pieces of the runtime away, and about the one mechanism in the driver that rewrites your command line behind your back.

All source references in this chapter are to **GCC 15.2.0** (`releases/gcc-15.2.0`). Line numbers in other releases will differ; the surrounding code rarely does. Where behaviour itself changed across a major version, it is flagged inline.

11 Turning the runtime off

Source references in this chapter are to GCC 15.2.0 (`releases/gcc-15.2.0`).

You are bringing up firmware on a Cortex-M part. There is no operating system, no glibc, and certainly no `crt1.o` that knows how to call `main`. So you reach for `-nostdlib`, the link succeeds in the sense that `ld` produces a file, and then it fails on an undefined reference to `__aeabi_idiv` — a symbol you have never typed, from a library you thought you had just switched off.

`-nostdlib` is not one mechanism. It is three `%{!…:}` guards in one spec string, plus one `case` label in the C++ driver, and it does not stop the compiler emitting calls into libgcc. Everything it removes, you now owe. This chapter reads the guards, derives the truth table from the nesting rather than from the manual, and separates `-nostdlib` from the three options that sound like it and are not.

Chapter 1.8 built the full link line. This chapter takes it apart.

11.1 The four options, and where they act

Four options in the family, all declared in `gcc/common.opt`, all carrying the single word `Driver` and nothing else:

```
nodefaultlibs
Driver

nostartfiles
Driver

nolibc
Driver

nostdlib
Driver

nostdlib++
Driver
```

Listing 10-1: the whole declaration of every option in this chapter (`gcc/common.opt:3782-3795`).

Read what is missing: there is no `Var(...)`, no help text, no `Init(...)`. An option with `Driver` and no `Var` has **no C variable anywhere in the compiler**. It cannot be tested with an `if`. Its entire existence is as a string that a `%{!nostdlib:…}` conditional can match against. That is why these options compose so cleanly, and it is also why grepping the source for `flag_nostdlib` finds nothing.

> **If you are on GCC 12 or earlier, `-nostdlib++` does not exist.** It was added in **GCC 13**; the option is absent from `gcc/common.opt` at `releases/gcc-11.4.0` and `releases/gcc-12.3.0`, and present from `releases/gcc-13.3.0` onward. `-nolibc` and `-nostdlib` go back much further.

11.2 The truth table is the nesting

Three lines of `LINK_COMMAND_SPEC` carry all of it. You met the whole template as Listing 8-1; here are just the guarded slots:

```
%{s} %{t} %{u*} %{z} %{Z} %{!nostdlib:%{!r:%{!nostartfiles:%S}}} \
...
%{!nostdlib:%{!r:%{!nodefaultlibs:%(link_ssp) %(link_gcc_c_sequence)}}}\
%{!nostdlib:%{!r:%{!nostartfiles:%E}}} %{T*}
```

Listing 10-2: the three guarded slots, at `gcc/gcc.cc:1168`, `:1176` and `:1177`.

Each slot is guarded three deep, and the outer two guards are the same on all three lines. Read them from the outside in and the behaviour falls out without consulting any documentation:

Option	Suppresses `%S` (startfiles)	Suppresses `%(link_gcc_c_sequence)`	Suppresses `%E` (endfiles)
`-nostartfiles`	yes	no	yes
`-nodefaultlibs`	no	yes	no
`-nostdlib`	yes	yes	yes
`-r`	yes	yes	yes

The two rows that catch people are the last two, and for the same reason: the outer guards are `%{!nostdlib:%{!r:` on every line, so **`-r` and `-nostdlib` are indistinguishable here**. A relocatable link never gets the runtime. If you have been passing `-r` and wondering why `crtbegin.o` is missing, that is the whole answer, and it is correct — a partial link must not drag in startup code that will be linked again later.

The distinction between `-nostartfiles` and `-nodefaultlibs` is just which inner guard sits on which line. `-nostartfiles` appears on the `%S` and `%E` lines; `-nodefaultlibs` appears on the library line. Nothing enforces the symmetry — it is literally which word was typed inside which `%{!…:}`.

11.3 Watching the guards fail

Every claim above is one `-###` away. `-###` prints the command lines the driver would run and then stops, so you can diff its output against itself:

```
$ gcc -### hello.c 2>&1 | tr ' ' '\n' | grep -E 'crt|^-l' | sort > full
$ gcc -### -nostdlib      hello.c 2>&1 | tr ' ' '\n' | grep -E 'crt|^-l' | sort > nostdlib
$ gcc -### -nostartfiles  hello.c 2>&1 | tr ' ' '\n' | grep -E 'crt|^-l' | sort > nostart
$ gcc -### -nodefaultlibs hello.c 2>&1 | tr ' ' '\n' | grep -E 'crt|^-l' | sort > nodeflibs
$ diff full nostart      # only crt* lines disappear
$ diff full nodeflibs    # only -l* lines disappear
$ diff full nostdlib     # both
$ gcc -### -r hello.c 2>&1 | tr ' ' '\n' | grep -cE 'crt|^-l'    # 0, same as -nostdlib
```

Listing 10-3: deriving the truth table experimentally.

The `sort` matters, because `-nodefaultlibs` also changes the order of what is left. What it does not do is change any search path: `-print-search-dirs` and `-print-file-name=crt1.o` give identical answers with and without every option in this chapter. These options remove *text from a command line*. They never touch where the driver looks.

11.4 What each guard is actually gating

`%S` and `%E` are the startfile and endfile specs from Chapter 1.8. Recall that their defaults are crt0-based and empty respectively (`gcc/gcc.cc:891-895` and `:897-900`), and the familiar `crt1.o crti.o crtbegin.o` shape is the GNU-userspace override. So what `-nostartfiles` costs you is target-specific — on glibc it is five objects, on a target that never defined `STARTFILE_SPEC` it may be one.

The library slot is more interesting, because it is not `-lc`:

```
%G %{!nolibc:%L %G}
```

Listing 10-4: `LINK_GCC_C_SEQUENCE_SPEC` (`gcc/gcc.cc:987-992`).

`%G` is libgcc and `%L` is libc, so the sequence is `-lgcc -lc -lgcc` — the double libgcc from Chapter 1.8. `-nodefaultlibs` removes the whole expression; `-nolibc` removes only the middle, leaving `-lgcc` on both sides. That is the option you want when you have your own libc but still want the compiler's helper routines, and it is the one almost nobody knows about.

This text is byte-identical at every release from GCC 11 to 15.2.0, which makes it one of the safer things to memorise in the driver.

11.5 The `-lgcc` you cannot switch off

Here is the failure the chapter opened with, stated precisely.

`-nostdlib` removes `-lgcc` from the link line. It does **not** remove the calls to libgcc that the compiler has already emitted into your object files. Those calls appear because the target has no instruction for what you wrote: a 64-bit divide on a 32-bit machine, a soft-float multiply, an unaligned wide load, a `switch` lowered to a table helper. On Arm they are the `__aeabi_*` family; elsewhere `__divdi3`, `__udivsi3`, `__muldf3`.

```
$ arm-none-eabi-gcc -c -O2 -mcpu=cortex-m0 div.c
$ arm-none-eabi-nm -u div.o | grep aeabi
```

Listing 10-5: the calls survive `-nostdlib` because they were emitted at compile time, long before the link line existed.

So the working incantation on bare metal is almost never bare `-nostdlib`. It is `-nostdlib` plus libgcc back by hand:

```
$ arm-none-eabi-gcc -nostdlib -T link.ld start.o main.o -lgcc
```

and `-print-libgcc-file-name` tells you which one you are getting:

```
$ arm-none-eabi-gcc -mcpu=cortex-m4 -mfloat-abi=hard -print-libgcc-file-name
```

That path runs through the whole multilib expansion of Chapter 1.7, so the answer changes with your `-mcpu` and `-mfloat-abi`. A `libgcc.a` from the wrong variant links and then misbehaves.

An undefined `__aeabi_idiv` or `__udivsi3` after a `-nostdlib` link is the single most common consequence of this chapter, and it is always the same fix.

11.6 The `-lstdc++` column has a different mechanism

`-lstdc++` is not in any spec. It is injected by the C++ driver before the specs run at all, so it cannot be removed by a `%{!…:}` guard. The C++ driver checks the options itself:

```
    case OPT_nostdlib__:
      args[i] |= SKIPOPT;
      /* FALLTHRU */
    case OPT_nostdlib:
    case OPT_nodefaultlibs:
      library = -1;
      break;
```

Listing 10-6: the C++ side of the family (`gcc/cp/g++spec.cc:170-176`).

`library = -1` means "never link the C++ runtime", per the state comment at `g++spec.cc:92-97`. Three consequences that do not follow from the spec guards:

`-nostartfiles` does not suppress `-lstdc++`. It is not in that `case` list. The startfiles and the C++ runtime are decided by two unrelated pieces of code, and this is where the symmetry of the truth table breaks.

`-r` still ends up at `-1`, by a different route. It is absent from Listing 10-6 but present a few lines further down, in the list alongside `-c`, `-S` and `-E` (`:215-225`), whose comment is *"Don't specify libraries if we won't link, since that would cause a warning."*

`-nostdlib++` disappears from the command line. Note the `SKIPOPT` on the first line, and the `/* FALLTHRU */` — it takes the same `library = -1` as `-nostdlib`, and is then deleted from the option array. Unlike `-nostdlib`, it never reaches the specs, so it drops the C++ runtime while leaving `-lc` and `-lgcc` untouched. Chapter 1.11 reads that state machine in full.

11.7 Three options that sound related and are not

This is the confusion worth spending a page on, because the names invite it.

Option	Stage	What it changes
`-nostdlib`	link	text on the linker command line
`-ffreestanding`	compile	what the compiler may assume about libc and `main`
`-nostdinc` / `-nostdinc++`	preprocess	the header search list

They are fully independent, and each is implemented in a different part of the compiler. You can `#include <stdio.h>` perfectly happily under `-nostdlib` and fail only at link. You can pass `-ffreestanding` and watch the link line not change at all.

11.7.1 `-ffreestanding` sets two flags, not one

Every secondhand account says `-ffreestanding` sets `flag_hosted = 0`. It sets two:

```
    case OPT_ffreestanding:
      value = !value;
      /* Fall through.  */
    case OPT_fhosted:
      flag_hosted = value;
      flag_no_builtin = !value;
      break;
```

Listing 10-7: `gcc/c-family/c-opts.cc:498-504`.

`flag_hosted` is what `__STDC_HOSTED__` is defined from and what relaxes the requirements on `main`. But `flag_no_builtin` is the flag that does the thing people actually notice: it stops the compiler turning `printf("x\n")` into `puts("x")` or `memcpy` into inline moves. Two further effects follow from it — loop pattern recognition is disabled (`:935-939`) and the `-Wmain` default flips (`:947-954`).

So `-ffreestanding` is closer to `-fno-builtin` than to `-nostdlib`, and if you were expecting it to change your link line you were reading the name, not the code.

11.7.2 `-nostdinc` removes GCC's own headers too

`-nostdinc` sets a file-static `bool`:

```
    case OPT_nostdinc:
      std_inc = false;
      break;
```

Listing 10-8: `gcc/c-family/c-opts.cc:639-641`.

which is passed through as the `stdinc` parameter of `register_include_chains` (`c-opts.cc:867-868`), where it gates one call:

```
  /* Finally chain on the standard directories.  */
  if (stdinc)
    add_standard_paths (sysroot, iprefix, imultilib, cxx_stdinc);
```

Listing 10-9: the entire implementation of `-nostdinc` (`gcc/incpath.cc:512-514`).

`add_standard_paths` is the function that walks `cpp_include_defaults` — the table from Chapter 1.9 that contains **both** `/usr/include` and GCC's own `lib/gcc/<target>/<version>/include`. One `if` skips the whole table. So `-nostdinc` does not remove "the system headers"; it removes *every* standard directory, including the one holding `<stddef.h>`, `<stdint.h>` and `<stdarg.h>`, which Chapter 1.9 showed the compiler is required to provide.

Which is why the useful form is almost always this pair:

```
$ gcc -nostdinc -isystem "$(gcc -print-file-name=include)" -c foo.c
```

Listing 10-10: drop the system headers, keep the compiler's own.

`-isystem` adds to a chain that is merged after `add_standard_paths` would have run, so it survives. Confirm with `-v`:

```
$ echo 'int main(void){return 0;}' | gcc -nostdinc -E -v -x c - 2>&1 | sed -n '/search starts
    here/,/End of search/p'
```

`-nostdinc++` is the narrow one. It sets `std_cxx_inc = false`, which arrives as `cxx_stdinc` and only suppresses entries whose `cplusplus` field is non-zero (`gcc/incpath.cc:176-177`) — the C++ directories from Chapter 1.9's table, and nothing else. The C headers stay.

11.8 What you owe once you have taken it away

Under `-nostdlib` you are responsible for four things that were previously invisible:

1. **An entry point.** The linker's default is `_start`, not `main`. Nothing now calls `main`, sets up the stack, or zeroes `.bss`.
2. **Static initialisation.** `crtbegin.o` / `crtend.o` are what walk the `.init_array` / `.fini_array` lists. Without them, C++ constructors for file-scope objects and C `__attribute__((constructor))` functions never run, silently.
3. **libgcc**, by hand, as above.
4. **A libc, or the discipline not to need one.** `-nolibc` is the middle ground: keep `-lgcc`, drop `-lc`.

Points 1 and 2 are the ones that produce a program that links, runs, and does the wrong thing rather than one that fails to build.

11.9 Documentation coverage

The options themselves are documented, in the Link Options node (`gcc/doc/invoke.texi:19067`, or `info gcc 'Link Options'`). What is documented is *what each one suppresses.*

What is not documented anywhere:

- **That they are pure spec conditionals with no C variable.** The manual describes the effect; nothing tells you the implementation is three `%{!…:}` guards, which is the fact that lets you predict the interactions instead of memorising them.
- **That `-r` is equivalent to `-nostdlib`** for all three slots. You can only get this by reading the `%{!r:` in Listing 10-2.
- **That `-nostdinc` removes GCC's own headers as well as the system's.** The help text says "standard system include directories", which reads as `/usr/include`. Listing 10-9 is the only place the truth is stated.
- **That `-ffreestanding` sets `flag_no_builtin`.** The documented effect is about `main` and the standard library; the built-in-recognition half is only in Listing 10-7.
- **That `-lstdc++` is removed by a different mechanism entirely**, and therefore that the truth table has an asymmetric fourth column. Nothing in the manual connects the two families.

11.10 Things that surprise people

These options change no search path. Removing `-lc` does not remove `<sysroot>/usr/lib` from the search list. Chapter 1.8's rule holds: the driver emits text, `ld` resolves it.

`-nostdlib` and `-ffreestanding` are frequently used together and do not overlap at all. For a genuinely hermetic build you want all three families — `-nostdlib -ffreestanding -nostdinc` — plus explicit `-I`, `-L` and `--sysroot`.

A missing constructor is a `-nostartfiles` bug. If your C++ globals are uninitialised and nothing crashed, look for `crtbegin.o` on the link line before looking anywhere else.

`-nolibc` exists. People reach for `-nodefaultlibs` and then add `-lgcc` back by hand, which is exactly what `-nolibc` does for you in one option.

All source references in this chapter are to **GCC 15.2.0** (`releases/gcc-15.2.0`). Line numbers in other releases will differ; the surrounding code rarely does. Where behaviour itself changed across a major version, it is flagged inline.

12 How `g++` injects `-lstdc++`

Source references in this chapter are to GCC 15.2.0 (`releases/gcc-15.2.0`).

You have a project of C object files. Nothing in it is C++. You link it with `g++` because that is what the build system happens to use, and you add `-lz` because you need zlib. The link line that comes out has `-lstdc++` on it, and — more disconcertingly — your `-lm` has moved. You typed it before `-lz`; it comes out after `-lstdc++`.

`g++` is the same binary logic as `gcc` with one extra function called before any option is acted on. That function may rewrite the entire option array: insert options, delete them, and reorder them. Everything surprising about C++ linking happens in those few hundred lines, and none of it is in the manual.

This chapter reads `gcc/cp/g++spec.cc` end to end. Chapter 1.8 showed what the generic driver does with a link line; this is what has already happened to that line before the driver sees it.

12.1 One hook, called before anything

The generic driver decodes `argv` into an array of `struct cl_decoded_option`, and then — before acting on a single one of them — calls into the front end:

```
  /* Do language-specific adjustment/addition of flags.  */
  lang_specific_driver (&decoded_options, &decoded_options_count,
            &added_libraries);
```

Listing 11-1: the only hook this chapter is about (`gcc/gcc.cc:4860-4862`).

Note the addresses-of. All three arguments are out-parameters: the callee may replace the array wholesale, change its length, and report how many libraries it added. Every front end links a `lang_specific_driver`; C's is a no-op, and C++'s is `gcc/cp/g++spec.cc`. **That is the entire difference between `gcc` and `g++`.**

The consequence is worth stating before the mechanism: by the time the specs run, `-lstdc++` is an ordinary `-l`, indistinguishable from one you typed. No spec mentions it. Grepping `-dumpspecs` for `stdc++` finds nothing.

12.2 One variable with four values

The whole decision collapses into a single `int`, and its own comment is the clearest documentation that exists:

```
  /* What action to take for the c++ runtime library:
    -1  means we should not link it in.
     0  means we should link it if it is needed.
     1  means it is needed and should be linked in.
     2  means it is needed but should be linked statically.  */
  int library = 0;
```

Listing 11-2: `gcc/cp/g++spec.cc:92-97`.

The default is `0` — *link it if needed* — and the whole first pass over the options exists to move it off zero. Two things follow from the initial value being the "maybe" state rather than "no": most transitions are `0 → 1`, and they are guarded by `if (library == 0)` so that an earlier `-1` cannot be overwritten.

Alongside `library` there is a parallel array of per-argument bits, one `int` per option:

```
/* This bit is set if we saw a `-xfoo' language specification.  */
#define LANGSPEC    (1<<1)
/* This bit is set if they did `-lm' or `-lmath'.  */
#define MATHLIB     (1<<2)
/* This bit is set if they did `-lc'.  */
#define WITHLIBC    (1<<3)
/* Skip this option.  */
#define SKIPOPT     (1<<4)
/* Add -lstdc++exp for experimental features that need library support.  */
#define EXPERIMENTAL    (1<<5)
```

Listing 11-3: the per-argument bits (`gcc/cp/g++spec.cc:26-35`).

Four of those five are set and tested. **`EXPERIMENTAL` is never used** — the `#define` is its only occurrence in the file, at every release from 13.3.0 through 15.2.0. `-fcontracts` is tracked by a separate `bool need_experimental` instead. It is the first of two pieces of vestigial state you will meet in this chapter; both are worth knowing about only so that you stop looking for what sets them.

12.3 What turns injection on

Two paths reach `library = 1`, and both are broader than people expect.

12.3.1 Any `-l` you do not recognise

```
    case OPT_l:
      if (strcmp (arg, MATH_LIBRARY) == 0)
        {
          args[i] |= MATHLIB;
          need_math = 0;
        }
      else if (strcmp (arg, "c") == 0)
        args[i] |= WITHLIBC;
      else
        /* Unrecognized libraries (e.g. -lfoo) may require libstdc++.  */
        library = (library == 0) ? 1 : library;
      break;
```

Listing 11-4: `gcc/cp/g++spec.cc:178-189`.

There are exactly two recognised libraries — `MATH_LIBRARY` (`"m"` by default) and `"c"`. Everything else is an unknown, and an unknown might need the C++ runtime. So `g++ -lz foo.o` links `libstdc++` because of `-lz`, on the strength of the source comment quoted above and nothing more.

`-Xlinker` and `-Wl,` do the same thing, with a comment that spells out the reasoning (`:207-213`): *"Arguments that go directly to the linker might be .o files, or something, and so might cause libstdc++ to be needed."*

12.3.2 Any input file that is not a header

This is the rule that folklore gets wrong. It is not "a C++ source file on the command line". It is a *negative* test on the filename:

```
        /* If we don't know that this is a header file, we might
           need to be linking in the libraries.  */
        if (library == 0)
          {
        if ((len <= 2 || strcmp (arg + (len - 2), ".H") != 0)
            && (len <= 2 || strcmp (arg + (len - 2), ".h") != 0)
            && (len <= 4 || strcmp (arg + (len - 4), ".hpp") != 0)
            ...
            && (len <= 3 || strcmp (arg + (len - 3), ".hh") != 0))
          library = 1;
          }
```

Listing 11-5: abridged from `gcc/cp/g++spec.cc:277-291`.

Nine header suffixes are checked — `.H`, `.h`, `.hpp`, `.hp`, `.hxx`, `.h++`, `.HPP`, `.tcc`, `.hh` — and **anything else sets `library = 1`**. There is no test for a C++ extension anywhere in the function. `g++ foo.o`, `g++ foo.a`, `g++ foo.s` all link the C++ runtime, because none of those is a header.

So the accurate rule is: *`g++` links `libstdc++` unless you gave it nothing but headers, or explicitly told it not to.* The driver is deliberately conservative, and the trade is sound — a missing `-lstdc++` is a confusing undefined-symbol error, a spurious one costs a `DT_NEEDED` entry.

You can watch the two paths independently:

```
$ g++ -### foo.o        2>&1 | tr ' ' '\n' | grep -c stdc      # 1 — input file rule
$ g++ -### foo.hh       2>&1 | tr ' ' '\n' | grep -c stdc      # 0 — header, no link anyway
$ gcc -### foo.o -lz    2>&1 | tr ' ' '\n' | grep -c stdc      # 0 — gcc injects nothing
```

Listing 11-6: the injection rules, one command each.

12.4 What turns injection off

Chapter 1.10 covered the link-line half of `-nostdlib`. Here is its C++ half:

```
    case OPT_nostdlib__:
      args[i] |= SKIPOPT;
      /* FALLTHRU */
    case OPT_nostdlib:
    case OPT_nodefaultlibs:
      library = -1;
      break;
```

Listing 11-7: `gcc/cp/g++spec.cc:170-176`.

And separately, the options under which no link happens at all:

```
    case OPT_c:
    case OPT_r:
    case OPT_S:
    case OPT_E:
    case OPT_M:
    case OPT_MM:
    case OPT_fsyntax_only:
      /* Don't specify libraries if we won't link, since that would
         cause a warning.  */
      library = -1;
      break;
```

Listing 11-8: `gcc/cp/g++spec.cc:215-225`.

Listing 11-8 is why `g++ -c foo.cc` produces no unused-argument noise, and why `-r` lands at `-1` even though it is absent from Listing 11-7.

Note what is **not** in either list: `-nostartfiles`. The truth table from Chapter 1.10 is asymmetric in its fourth column precisely because that column is decided here, by two `case` labels, rather than by the `%{!…:}` guards in `LINK_COMMAND_SPEC`.

`-nostdlib++` is the one that vanishes. It takes `SKIPOPT` before falling through, which means the option is deleted from the array at `:368-369` and never reaches the specs. So it removes the C++ runtime and leaves `-lc` and `-lgcc` completely alone — which is exactly what its name promises and what `-nostdlib` cannot do.

> **If you are on GCC 12 or earlier, `-nostdlib++` does not exist.** It arrived in **GCC 13**, along with `-lstdc++exp`. The same commit took this file from 436 to 460 lines, so every line number in this chapter differs before GCC 13 — and before GCC 12 the file is `g++spec.c`, not `.cc`.

12.5 The reordering nobody documents

Now the part that changed your command line. In the copy loop, two options are plucked out as they go past:

```
      /* Make sure -lstdc++ is before the math library, since libstdc++
     itself uses those math routines.  */
      if (!saw_math && (args[i] & MATHLIB) && library > 0)
    {
      --j;
      saw_math = &decoded_options[i];
    }
```

```
      if (!saw_libc && (args[i] & WITHLIBC) && library > 0)
    {
      --j;
      saw_libc = &decoded_options[i];
    }
```

Listing 11-9: `gcc/cp/g++spec.cc:326-338`.

The `--j` is the removal. The option was already written to `new_decoded_options[j]` at the top of the loop; decrementing `j` means the next iteration overwrites it. The option is stashed in a pointer and re-emitted at the end, after the runtime:

```
  if (saw_math)
    new_decoded_options[j++] = *saw_math;
  else if (library > 0 && need_math)
    { ... generate_option (OPT_l, MATH_LIBRARY, ...) ... }
  if (saw_time)
    new_decoded_options[j++] = *saw_time;
  if (saw_libc)
    new_decoded_options[j++] = *saw_libc;
```

Listing 11-10: the re-emission (`gcc/cp/g++spec.cc:430-443`).

Three things to take from this.

The reordering is real and silent. `g++ -lm foo.o` links `foo.o -lstdc++ -lm`. On a target where everything is a static archive, `ld` resolves left to right and order determines which symbols get pulled in, so this is not cosmetic. The reason is correct — `libstdc++.a` has undefined references into `libm` — but the consequence is that **you cannot put `-lm` before `-lstdc++` through the `g++` driver at all.** If you need that, drive the link with `gcc` or `ld` and supply the C++ runtime yourself.

A missing `-lm` is synthesised. The `else if` branch adds one when you did not. It is gated on `need_math`, which is `(MATH_LIBRARY[0] != '\0')` at `:129` — so a port that folds the maths routines into libc defines `MATH_LIBRARY ""` and no `-lm` appears. Check this before concluding that a new target's link line is too short.

The `-lrt` slot is dead code. `saw_time` is initialised to `NULL` at `:118-119`, read at Listing 11-10, and **assigned nowhere in the file** — there is no `TIMELIB` bit to go with `MATHLIB` and `WITHLIBC`. Verified identical at every release from 11.4.0 to 15.2.0. Exactly two of your options ever get hoisted: `-lm` and `-lc`. This is the second piece of vestigial state promised earlier.

```
$ g++ -### -lm -lz foo.o 2>&1 | tr ' ' '\n' | grep -E '^-l|stdc'
```

Listing 11-11: watch `-lm` come out after `-lstdc++`, and `-lz` stay where you put it.

12.6 `-static-libstdc++` changes no path and no filename

The option does not affect the library name, the search order, or any `-L`. It wraps the one `-l` in a pair of linker state changes:

```
… -Wl,-Bstatic -lstdc++ -Wl,-Bdynamic -lm -lc
```

That is all it is: flip `ld` into archive-preference mode for exactly one library, then flip it back. The two halves are emitted around the `-l`:

```
#ifdef HAVE_LD_STATIC_DYNAMIC
      if (library > 1 && !static_link)
    {
      generate_option (OPT_Wl_, LD_STATIC_OPTION, 1, CL_DRIVER,
               &new_decoded_options[j]);
      j++;
    }
#endif
```

Listing 11-12: the opening half (`gcc/cp/g++spec.cc:384-391`); the closing `LD_DYNAMIC_OPTION` is the mirror image at `:421-428`.

`library > 1` is the `2` state, and `!static_link` skips the wrap under plain `-static`, where everything is archive-only already. The option that sets the state also deletes itself:

```
    case OPT_static_libstdc__:
      library = library >= 0 ? 2 : library;
#ifdef HAVE_LD_STATIC_DYNAMIC
      /* Remove -static-libstdc++ from the command only if target supports
         LD_STATIC_DYNAMIC.  When not supported, it is left in so that a
         back-end target can use outfile substitution.  */
      args[i] |= SKIPOPT;
#endif
      break;
```

Listing 11-13: `gcc/cp/g++spec.cc:235-243`.

Read the `library >= 0` guard: `g++ -c -static-libstdc++` stays at `-1`, because `-c` got there first. And read the `#ifdef`: on a target *without* `HAVE_LD_STATIC_DYNAMIC` the option is **not** deleted, so it survives into the specs for the port to act on. That conditional is why two targets disagree about whether the option appears in `-v` output.

Two consequences that bite in practice:

`libstdc++.a` has to exist in the same search path. Nothing about the search changed — only `ld`'s preference between `.a` and `.so`. If you ship a cross toolchain with a shared-only libstdc++, this option silently gives you the shared one and the link succeeds.

The option does not appear in the link line on most targets. Do not grep `-v` output for `-static-libstdc++`; grep for `Bstatic`:

```
$ g++ -### -static-libstdc++ foo.o 2>&1 | tr ' ' '\n' | grep -E 'Bstatic|Bdynamic|stdc'
```

Listing 11-14: the option is gone; its effect is two `-Wl,` options.

The exact spelling is configure-probed, in a `case "$target"` at `gcc/configure.ac:4231-4250`:

Value	Target
`-Bstatic` / `-Bdynamic`	GNU ld, Solaris — the default
`-bstatic` / `-bdynamic`	AIX
`-aarchive_shared` / `-adefault`	HP-UX (non-GNU ld)

with the three `AC_DEFINE`s at `:4253-4259`.

12.7 Which library, exactly

The name is a macro, so a port can change it:

```
#ifndef LIBSTDCXX
#define LIBSTDCXX "stdc++"
#endif
#ifndef LIBSTDCXX_PROFILE
#define LIBSTDCXX_PROFILE LIBSTDCXX
#endif
#ifndef LIBSTDCXX_STATIC
#define LIBSTDCXX_STATIC NULL
#endif
```

Listing 11-15: `gcc/cp/g++spec.cc:44-52`.

`LIBSTDCXX_PROFILE` is chosen when `saw_profile_flag` is set by `-p` or `-pg` (`:407-410`). `LIBSTDCXX_STATIC` is not a replacement but an **extra** library appended after the main one when linking statically (`:413-419`) — the hook an RTOS

or bare-metal port uses to add its own support archive. If a link line on an unusual target does not say `-lstdc++`, one of these three macros is why.

`-fcontracts` adds `-lstdc++exp` **ahead of** the main library (`:378-383`), and — read the line numbers against Listing 11-12 — that happens *before* the `-Wl,-Bstatic`. So **`-static-libstdc++` does not cover `libstdc++exp`.**

`-stdlib=libc++` switches to clang's library, emitting `-lc++` and `-lc++abi` (`:392-406`). The ABI library is skipped when the port sets `LIBCXXABI` to `NULL`, whose comment explains the case: a platform may forward the ABI library from libc++ or combine it some other way. The option only exists if GCC was configured for it — `ENABLE_STDLIB_OPTION`, which Chapter 1.12 covers.

12.8 The bookkeeping that stops `g++` linking nothing

`added_libraries` is incremented for each synthesised `-l` and handed back through the third out-parameter of Listing 11-1. The generic driver uses it once:

```
  if (n_infiles == added_libraries)
    fatal_error (input_location, "no input files");
```

Listing 11-16: `gcc/gcc.cc:8970-8971`.

Without that count, plain `g++` with no arguments would see the libraries it had just added itself, conclude it had inputs, and attempt a link. This is the only place the return value is used, and it is the reason `lang_specific_driver` has to report what it did rather than just doing it.

12.9 Documentation coverage

The options are documented — `-static-libstdc++`, `-nostdlib++` and `-stdlib=` are all in the manual. The mechanism is not, anywhere:

- **The injection itself has no texi node.** No document states that `g++` adds `-lstdc++`, let alone under what conditions. The `library` state machine exists only as the source comment in Listing 11-2.
- **The `-lfoo` rule is undocumented**, and it is the one that catches people: linking any third-party library through `g++` guarantees the C++ runtime.
- **The input-file rule is undocumented and is a header-suffix blacklist**, not a C++-source test. Listing 11-5 is the only statement of it.
- **The `-lm` / `-lc` hoist is undocumented.** Nothing warns that `g++` silently reorders your command line.
- **`-static-libstdc++`'s mechanism is undocumented.** The option is described; that it becomes `-Wl,-Bstatic` … `-Wl,-Bdynamic` and vanishes from the command line is not, and neither is the `HAVE_LD_STATIC_DYNAMIC` conditional that decides whether it vanishes.

This is the largest documentation gap in Part I, which is why the chapter quotes so much source: the comments in `g++spec.cc` are the specification.

12.10 Things that surprise people

`gcc` never injects anything. Link C++ objects with `gcc` and you get undefined `std::` symbols. That is the whole practical difference between the two drivers at link time — not a different compiler, not different code generation.

`g++ -lz foo.o` links the C++ runtime even though nothing in it is C++.

Your `-lm` and `-lc` positions are not preserved. Everything else is.

`-nostartfiles` does not suppress `-lstdc++`. Two mechanisms, two lists.

A successful link says nothing about whether the program starts. Finding `libstdc++.so.6` at run time is the dynamic loader's job, driven by `DT_NEEDED`, `DT_RUNPATH` and `ld.so.conf` — an entirely separate system from anything in this chapter. Which `libstdc++` the `-l` resolved to at link time is Chapter 1.12.

All source references in this chapter are to **GCC 15.2.0** (`releases/gcc-15.2.0`). Line numbers in other releases will differ; the surrounding code rarely does. Where behaviour itself changed across a major version, it is flagged inline.

13 Which `libstdc++`, and which `<vector>`

Source references in this chapter are to GCC 15.2.0 (`releases/gcc-15.2.0`).

You have a cross toolchain and a target root filesystem. You pass `--sysroot=/arm-rootfs`, you point it at a rootfs that definitely contains `/usr/include/c++/15` and `/usr/lib/libstdc++.so.6`, and `g++ -v -E -x c++ /dev/null` shows you a `<vector>` from somewhere else entirely — a path under your compiler's install prefix. Then you try to fix it with `-B` and nothing changes at all.

Neither half of libstdc++ is in the sysroot, by design. The headers and the library both live relative to where the compiler is installed, and `--sysroot` is about the C library. There is exactly one configure-time route by which the headers become sysroot-relative, and you have to ask for it.

Chapter 1.11 got `-lstdc++` onto the link line. This chapter answers which file it resolves to, and which `<vector>` you compiled against — two questions with two completely different mechanisms that happen to be moved by the same prefix.

13.1 Three questions people conflate

Question	Answered by	Sysroot-aware?
Should `-lstdc++` be on the link line?	the `g++` driver — Chapter 1.11	no — pure option rewriting
Where does `#include <vector>` come from?	GCC's compiled-in include table	**only if configured for it**
Which `libstdc++.so` does that `-l` find?	the driver's prefix list → `-L` flags → `ld`	yes, but the sysroot usually loses

The third row carries Chapter 1.8's rule: the driver never opens `libstdc++.so`. It emits one `-l` and a pile of `-L`, and `ld` resolves. So "which library" is really "which directory comes first".

13.2 Where a cross install actually puts it

For a cross — `host != target` — libstdc++ installs into the **tool directory**, not the sysroot and not `$prefix/lib`:

Artefact	Install location
Headers	`$prefix/$target_alias/include/c++/$version`
`libstdc++.{a,so}`	`$exec_prefix/$target_alias/lib[/$multi_os_dir]`

Both come out of one autoconf macro, `GLIBCXX_EXPORT_INSTALL_INFO` (`libstdc++-v3/acinclude.m4:729`). The headers first:

```
# Default case for install directory for include files.
if test $version_specific_libs = no && test $gxx_include_dir = no; then
  gxx_include_dir='include/c++/${gcc_version}'
  if test -n "$with_cross_host" &&
     test x"$with_cross_host" != x"no"; then
    gxx_include_dir='${prefix}/${target_alias}/'"$gxx_include_dir"
  else
    gxx_include_dir='${prefix}/'"$gxx_include_dir"
  fi
fi
```

Listing 12-1: the header install directory (`libstdc++-v3/acinclude.m4:760-769`).

Then the library, whose comment states the policy outright:

```
# Calculate glibcxx_toolexecdir, glibcxx_toolexeclibdir
# Install a library built with a cross compiler in tooldir, not libdir.
if test x"$glibcxx_toolexecdir" = x"no"; then
  if test -n "$with_cross_host" &&
     test x"$with_cross_host" != x"no"; then
```

```
      glibcxx_toolexecdir='${exec_prefix}/${host_alias}'
      case ${with_toolexeclibdir} in
    no)
      glibcxx_toolexeclibdir='${toolexecdir}/lib'
      ;;
```

Listing 12-2: the library install directory (`libstdc++-v3/acinclude.m4:784-793`).

> **Read this file at the tag, not in a checkout.** `libstdc++-v3/acinclude.m4` differs by **+70/-5** between `releases/gcc-15.2.0` and the AdaCore working branch in the trees this book was written from, and by more between point releases. Every line number in this section was re-derived from `git show releases/gcc-15.2.0:libstdc++-v3/acinclude.m4`; taking them from a worktree puts them roughly two dozen lines out.

13.3 Two traps in those two listings

Both are the kind of thing you only notice after a build has gone wrong.

`$prefix` and `$exec_prefix` are different variables. They default to the same value, which is why nobody notices. But Listing 12-1 builds the header path from `${prefix}` and Listing 12-2 builds the library path from `${exec_prefix}`. Pass `--exec-prefix` and your C++ headers and your C++ library land in **different trees**, from one configure run, with no warning.

`${host_alias}` in a target library means GCC's *target*. Target libraries are configured with `--host=<target triple>`, because from libstdc++'s own point of view the machine it will run on *is* GCC's target. So `${exec_prefix}/${host_alias}` in Listing 12-2 is `$exec_prefix/<target>`, the tooldir. libgcc plays the same trick with `real_host_noncanonical`. Whenever a path formula inside a target library says "host", read "GCC's target" — Chapter 1.2's three machines, seen from the other end.

13.4 The driver's matching view

The install rules would be useless if the driver looked elsewhere, so the same two paths are computed a second time, in `gcc/configure.ac`. That file says so:

```
# This logic must match libstdc++-v3/acinclude.m4:GLIBCXX_EXPORT_INSTALL_INFO.
if test x${gcc_gxx_include_dir} = x; then
  if test x${enable_version_specific_runtime_libs} = xyes; then
    gcc_gxx_include_dir='${libsubdir}/include/c++'
  else
    libstdcxx_incdir='include/c++/$(version)'
    if test x$host != x$target; then
       libstdcxx_incdir="$target_alias/$libstdcxx_incdir"
    fi
    gcc_gxx_include_dir="\$(libsubdir)/\$(libsubdir_to_prefix)$libstdcxx_incdir"
  fi
```

Listing 12-3: the driver's half (`gcc/configure.ac:213-223`).

It is worse than a duplicated formula: libstdc++'s copy of the comment (`acinclude.m4:727-728`) names a **third** file, `config/gxx-include-dir.m4`, that must be kept consistent too. Change one, change all three, or the headers install where the driver will not look.

For the library side the driver does not compute a path at all — it adds the whole tooldir to its prefix list:

```
  add_prefix (&startfile_prefixes,
          concat (tooldir_prefix, "lib", dir_separator_str, NULL),
          "BINUTILS", PREFIX_PRIORITY_LAST, 0, 1);
```

Listing 12-4: `gcc/gcc.cc:5537-5539`.

That is `add_prefix`, **not** `add_sysrooted_prefix`. One function call is the whole reason `--sysroot` does not move your C++ library. Compare Chapter 1.5's Listing 5-9: the sysrooted variant prepends `target_system_root` (`:3177-3205`); this one does not.

13.5 Does `--sysroot` move the headers? Only if you asked

Each entry in the compiled-in include table carries its own sysroot flag — not a global setting, a per-directory one:

```
  const char add_sysroot;   /* FNAME should be prefixed by
                               cpp_SYSROOT.  */
```

Listing 12-5: `gcc/cppdefault.h:48-49`.

and for the C++ directories the flag is a configure-time macro:

```
#ifdef GPLUSPLUS_INCLUDE_DIR
    /* Pick up GNU C++ generic include files.  */
    { GPLUSPLUS_INCLUDE_DIR, "G++", 1, 1,
      GPLUSPLUS_INCLUDE_DIR_ADD_SYSROOT, 0 },
#endif
```

Listing 12-6: `gcc/cppdefault.cc:43-47`.

Because the decision is baked into the table per entry, **no command-line flag can change it after the fact**. `--sysroot` supplies the string; whether a given directory uses it was decided when GCC was configured.

That macro comes from `gcc_gxx_include_dir_add_sysroot`, which is initialised to zero at `gcc/configure.ac:211` and raised in exactly one place:

```
elif test "${with_sysroot+set}" = set; then
  gcc_gxx_without_sysroot=`expr "${gcc_gxx_include_dir}" : "${with_sysroot}"'\(.*\)'`
  if test "${gcc_gxx_without_sysroot}"; then
    gcc_gxx_include_dir="${gcc_gxx_without_sysroot}"
    gcc_gxx_include_dir_add_sysroot=1
  fi
fi
```

Listing 12-7: `gcc/configure.ac:224-230`.

Read the `elif`. It is the else-branch of `if test x${gcc_gxx_include_dir} = x` from Listing 12-3, so it is reachable **only when you passed `--with-gxx-include-dir`**. If you did not, the default branch runs, the path is prefix-relative, and the flag stays `0` forever. Even when you did pass it, the sysroot is only factored out if your path is *literally string-prefixed* by `--with-sysroot`'s value — that `expr` is a string match, not a path comparison.

Which gives the whole case table:

Configuration	`include/c++/15` resolves to	Follows `--sysroot`?
plain cross, no `--with-gxx-include-dir`	`$prefix/$target/include/c++/$ver`, relocatable	no
`--with-sysroot=S` only	same as above	no
`--with-sysroot=S --with-gxx-include-dir=S/usr/include/c++/15`	`<sysroot>/usr/include/c++/15`	**yes**
`--with-gxx-include-dir=/opt/foo`, no overlap with the sysroot	`/opt/foo`, absolute	no

Only the third row gives you the sysroot-native layout, and it needs both options with one path textually inside the other. The default buys you something else that is usually worth more: **relocatability**.

13.6 What the default gives you instead

If a standard directory is not sysrooted, `add_standard_paths` takes a different branch — it relocates the path against the *installed* prefix:

```
      else if (!p->add_sysroot && relocated
           && !filename_ncmp (p->fname, cpp_PREFIX, cpp_PREFIX_len))
        {
          static const char *relocated_prefix;
          char *ostr;
          /* If this path starts with the configure-time prefix,
         but the compiler has been relocated, replace it
         with the run-time prefix.  The run-time exec prefix
         is GCC_EXEC_PREFIX.  Compute the path from there back
         to the toplevel prefix.  */
```

Listing 12-8: `gcc/incpath.cc:192-201`.

This is the branch a normal cross takes: not sysrooted, but relocated — move the installed toolchain and the C++ headers follow it, because the path is recomputed from `GCC_EXEC_PREFIX` (Chapter 1.6) rather than being an absolute string. The two branches are mutually exclusive by construction, which is the precise reason `--sysroot` cannot move GCC's own header directories: they are not sysroot-relative, they are prefix-relative, and the prefix is discovered at run time.

`-B` cannot help either. `-B` feeds `include_prefixes`, and `%I` turns that into `-isystem <B>/<target>/<version>/include` and `.../include-fixed` — GCC's *own* header directories, from Chapter 1.9. The C++ directories are not in that set. So `-B` changes which `<stdint.h>` you get and not which `<vector>`.

13.6.1 The per-invocation escape hatch

Independent of all the configure-time machinery, a leading `=` or `$SYSROOT` on any `-I`, `-isystem` or `-idirafter` is expanded to the sysroot:

```
      if (p->user_supplied_p)
    {
      if (p->name[0] == '=')
        p->name = concat (sysroot, p->name + 1, NULL);
      if (startswith (p->name, "$SYSROOT"))
        p->name = concat (sysroot, p->name + strlen ("$SYSROOT"), NULL);
    }
```

Listing 12-9: `gcc/incpath.cc:333-339`; applied to the quote, bracket, system, after and `#embed` chains at `:356-364`.

Note `user_supplied_p`: this only ever touches paths you passed on the command line, never the compiled-in table. Which makes it the right tool for testing a sysroot-resident header set against a compiler that was not configured for one:

```
$ arm-linux-gnueabihf-g++ --sysroot=/arm-rootfs \
    -nostdinc++ -isystem =/usr/include/c++/15 -E -v -x c++ /dev/null
```

Listing 12-10: `-nostdinc++` drops the compiled-in C++ directories (Chapter 1.10), `=` supplies sysroot-relative ones.

13.7 Does `--sysroot` move the library? Yes, and it usually loses

Two mechanisms put the sysroot into the link-time search:

Sysrooted startfile prefixes. `/lib/` and `/usr/lib/` (`gcc/gcc.cc:1613-1617`) are added through `add_sysrooted_prefix` (`:8621-8630`), which prepends `target_system_root`.

The linker is told. If `ld` supports it, `%(sysroot_spec)` — that is `--sysroot=%R` (`:1189-1191`) — is prepended to the whole `link` spec (`:8554-8564`), which makes `ld`'s own built-in search directories and any `=`-prefixed path in a linker script sysroot-relative too.

So the sysroot genuinely is searched. It loses because Listing 12-4's tooldir prefix is not sysrooted and *is* where the file actually is. A sysroot copy only wins if you put one there and it sorts earlier — at which point you have two candidates and a search-order question you now have to answer. Usually the wrong move.

There is one gate worth knowing about here, which explains a different confusion:

```
  else if (*cross_compile == '0' || target_system_root)
```

Listing 12-11: `gcc/gcc.cc:8590`.

A cross compiler configured with no sysroot at all gets no `/lib` or `/usr/lib` prefixes whatsoever. That is not a bug: searching the *host's* `/usr/lib` for target libraries is how you get "file in wrong format" errors. If a cross seems to have suspiciously few library paths, this one line is why, and it is correct. Ports defining `STARTFILE_PREFIX_SPEC` take the earlier branch instead (`:8579-8587`), which wins outright over this whole chain.

13.8 The four configure flags that move it

Each has to be honoured twice — by libstdc++'s install rules and by the driver's compiled-in defaults.

Flag	Moves	Default	Follows `--sysroot`?
`--with-gxx-include-dir=DIR`	libstdc++ headers	`$prefix/$target_alias/include/c++/$ver`	only via Listing 12-7
`--with-gxx-libcxx-include-dir=DIR`	libc++ headers, **and gates `-stdlib=`**	`$prefix/$target_alias/include/c++/v1`	same mechanism
`--enable-version-specific-runtime-libs`	headers **and** libraries into `libsubdir`	off	never
`--with-toolexeclibdir=DIR`	where cross-built libraries install	`$tooldir/lib`	never

`--with-gxx-libcxx-include-dir` does double duty. Besides naming the libc++ header directory, it decides whether `-stdlib=` exists at all: `=no` disables the option, a path enables it, and *unset* enables it only on recent Darwin (`gcc/configure.ac:255-274`). If `-stdlib=libc++` is rejected as an unknown option, that is why — and Chapter 1.11's `-lc++`/`-lc++abi` emission is unreachable without it. The libc++ header entry is a separate row in the same table, distinguished by `cplusplus == 2` (`cppdefault.cc:58-62`) and selected by `flag_stdlib_kind` at `incpath.cc:176-177`.

`--enable-version-specific-runtime-libs` moves both halves out of the tooldir into GCC's versioned directory — headers to `${libsubdir}/include/c++`, libraries to `${libdir}/gcc/${host_alias}/${gcc_version}$(MULTISUBDIR)` (`acinclude.m4:771-782`). Two things follow: it applies only if you did *not* pass `--with-gxx-include-dir` (read the `&&` in Listing 12-1), and a version-specific layout can never be sysroot-relative, because Listing 12-7 is in the branch it does not take.

`--with-toolexeclibdir` moves the install, not the search. This is the one that produces a working build and a broken compiler. The flag is consumed at `acinclude.m4:790-797`, so the library installs where you said. But Listing 12-4 builds the driver's prefix from `tooldir_prefix` and the target triple — it is **not** parameterised by this flag. Point it somewhere unusual and `g++` will install `libstdc++.so` there and then fail to find it, unless you also supply `-L` or `-B`.

13.9 Asking your own compiler

Every claim above is checkable on an installed toolchain, and this is the sequence to run when a link picks up the wrong C++ runtime:

```
$ arm-linux-gnueabihf-g++ -print-sysroot                  # empty means: none configured
$ arm-linux-gnueabihf-g++ -print-search-dirs              # raw prefix lists, pre-expansion
$ arm-linux-gnueabihf-g++ -print-file-name=libstdc++.so   # which file the -l resolves to
$ arm-linux-gnueabihf-g++ -print-file-name=libstdc++.a    # ditto for -static-libstdc++
$ arm-linux-gnueabihf-g++ -mcpu=cortex-a9 -print-multi-os-directory
$ arm-linux-gnueabihf-g++ -v -E -x c++ /dev/null          # the C++ header search list
```

Listing 12-12: the six questions, in the order you should ask them.

Two readings to remember. `-print-file-name` echoes its argument back unchanged when the file was not found (Chapter 1.3), so `libstdc++.so` coming back verbatim means "not found", not "found in the current directory". And in the last command, **check whether the `include/c++/…` lines begin with your sysroot path**: if they do you have

the third row of the case table, and if they do not, `--sysroot` will never move them and no amount of retrying will change that.

`-print-search-dirs` is the one to be careful with. It prints prefixes *before* multilib expansion, so its output is not the set of directories any particular `-mcpu` actually searches — Chapter 1.7's four-way expansion happens later, inside `for_each_path` (`gcc/gcc.cc:2778`) with the variant chosen by `set_multilib_dir` (`:9764`). `-print-file-name` is the honest answer, because it performs the whole search.

13.10 Documentation coverage

The four configure flags are all in `gcc/doc/install.texi`, and the search-path ordering is in the internals manual (`info gccint 'Target Macros' Driver`, or `gcc/doc/tm.texi`, "Here is the order of prefixes tried for startfiles").

Not documented anywhere:

- **`--with-gxx-include-dir`'s interaction with `--with-sysroot`.** The entire case table above exists only as the shell in Listing 12-7. `configure --help` does not hint at it.
- **That the sysroot decision is per include-directory**, carried in the `add_sysroot` field (Listing 12-5) and therefore unchangeable by any flag. The corollary — that GCC's own directories are *relocated* rather than sysrooted (Listing 12-8) — is the precise reason `--sysroot` cannot move them, and it is stated nowhere but the source.
- **That `--with-toolexeclibdir` moves the install without moving the search.** The option is documented; the asymmetry is not.
- **That the path logic is duplicated across three files.** Only the in-tree comments say so, and they are the only warning you get.
- **That `-print-search-dirs` shows prefixes before multilib expansion.**

13.11 Things that surprise people

`--sysroot` does not move your C++ headers unless you configured `--with-gxx-include-dir` with a path inside `--with-sysroot`.

`$prefix/$target/lib` is not sysrooted, is where a cross libstdc++ actually lives, and is searched regardless of `--sysroot`.

`-B` does not move C++ header search. It moves GCC's own headers, which is enough to change `<stdint.h>` and not `<vector>`.

A cross ignores `LIBRARY_PATH`. The guard is `*cross_compile == '0'` (Chapter 1.6). Do not debug a cross with it.

A successful link says nothing about whether the program runs. Link-time search and run-time search are unrelated systems: `-L` and the prefix lists resolve the `-l`; `DT_NEEDED`, `DT_RUNPATH`, `ld.so.conf` and `LD_LIBRARY_PATH` decide whether the target finds `libstdc++.so.6` at exec time. Getting the first right tells you nothing about the second.

All source references in this chapter are to **GCC 15.2.0** (`releases/gcc-15.2.0`). Line numbers in other releases will differ; the surrounding code rarely does. `libstdc++-v3/acinclude.m4` in particular moves between point releases and in vendor forks — read it at the tag. Where behaviour itself changed across a major version, it is flagged inline.