

C++ A Visual Guide

Sample Book — Chapters 1 to 2

Introduction

This book is designed for readers who are taking their first steps with C++, including systems engineers who want to build the right foundation from the start.

Rather than asking you to memorize more than you need, the book focuses on the essential concepts that will support your future study and day-to-day work. More advanced topics are included only when they are likely to become useful to many readers, and even then they are introduced only far enough to explain their purpose, context, and overall role.

Processes, relationships, and structures that can be difficult to picture from text alone are supported by diagrams. By moving between explanations, visuals, and examples, you can build a clearer and more practical understanding of the subject.

Product names, service names, logos, icons, and other identifying marks mentioned in this book are trademarks or registered trademarks of their respective owners. This book is not an official publication, certified training resource, or endorsed learning material of any company or organization.

C++ systems code becomes manageable when you reason explicitly about types, object lifetimes, ownership, and interfaces. The syntax matters, but these four ideas determine whether a program remains correct when it grows, handles failures, or interacts with operating-system and hardware resources.

This guide uses **C++20** as its language baseline. Facilities that require C++23 are labeled explicitly. Its examples favor portable standard C++; compiler- or platform-specific commands are labeled as examples rather than universal rules. The material progresses from language and lifetime fundamentals to library use, builds, diagnostics, concurrency, interoperability, performance, security, and project work.

How to Use This Book

This is a beginner-friendly foundation, and the chapters are arranged so that each one supplies knowledge used by the chapters that follow.

- Read the chapters in order on your first pass. The progression moves from how C++ programs are built and executed to types, lifetime, ownership, libraries, tooling, concurrency, interoperability, and project work.
- At the beginning of each chapter, review **What You'll Be Able to Do**. These outcomes identify the chapter's practical learning goals and help you distinguish essential material from supporting detail.
- Use the diagrams to understand relationships, processes, structures, state changes, and boundaries that are harder to picture from prose alone. The diagrams support the explanation; they do not replace it.
- Run and modify the code and hands-on examples whenever possible. Typing, compiling, testing, and deliberately changing a small example will reveal assumptions that passive reading can hide.
- Use each chapter's summary, checkpoint, review questions, or exercise to confirm that you can explain and apply the fundamentals without relying on the preceding page.

- Treat advanced-topic overviews as signposts rather than material to memorize now. Return to them when a project, code review, performance investigation, or platform boundary makes the topic relevant.

Table of Contents

[Introduction](#)

[How to Use This Book](#)

[Part I: How C++ Programs Work](#)

[Chapter 1: The C++ Mental Model for Systems Work](#)

[What C++ is](#)

[Where a program sits](#)

[Abstraction without avoidable cost](#)

[Three questions to ask about every piece of code](#)

[1. What is the type and lifetime of each object?](#)

[2. Who owns each resource?](#)

[3. What does each interface read, change, borrow, or own?](#)

[Modern C++ is not “C with classes”](#)

[Portability has boundaries](#)

[Reading a small program by responsibilities](#)

[Misconceptions that cause trouble](#)

[What to retain](#)

[Chapter 2: From Source File to Running Program](#)

[The build pipeline](#)

[The smallest complete program](#)

[Build and run with GCC or Clang](#)

[One MSVC example](#)

[What the compiler does](#)

[What the linker does](#)

[Four distinct failure classes](#)

[A compiler error](#)

[A linker error](#)

[A runtime failure](#)

[A wrong result without an error message](#)

[main, arguments, and exit status](#)

[Warnings are engineering information](#)

[Debug information and optimization levels](#)

[A practical diagnosis sequence](#)

[Checkpoint](#)

Part I: How C++ Programs Work

Chapter 1: The C++ Mental Model for Systems Work

What You'll Be Able to Do

By the end of this chapter, you will be able to:

- Explain where C++ source, the compiler, the linker, the operating system, and hardware fit in one execution model.
- Read unfamiliar code through the four lenses of type, lifetime, ownership, and interface responsibility.
- Distinguish portable language rules from implementation, platform, and project-specific assumptions.

C++ is a language for building useful abstractions without giving up explicit control over data

representation, lifetime, and resource use. A beginner does not need to start by memorizing every feature. Start by learning what objects exist, how long they remain valid, who owns each resource, and what each interface promises.

What C++ is

C++ is a **compiled language**, which means a compiler translates source code into instructions that a processor can execute. It is also **statically typed**, which means the type of each expression is checked primarily before the program runs.

C++ is commonly used for software where one or more of the following matter:

- Direct interaction with operating-system or hardware facilities.
- Predictable use of memory and processor time.
- Native libraries that must work with existing C or C++ interfaces.
- Long-running services, agents, command-line tools, and infrastructure components.
- Software that must run on several operating systems or processor families.

In this guide, **systems programming** means building software that works close to resources such as files, processes, sockets, devices, memory, and threads. It does not mean that every program must be an operating-system kernel or device driver.

Where a program sits

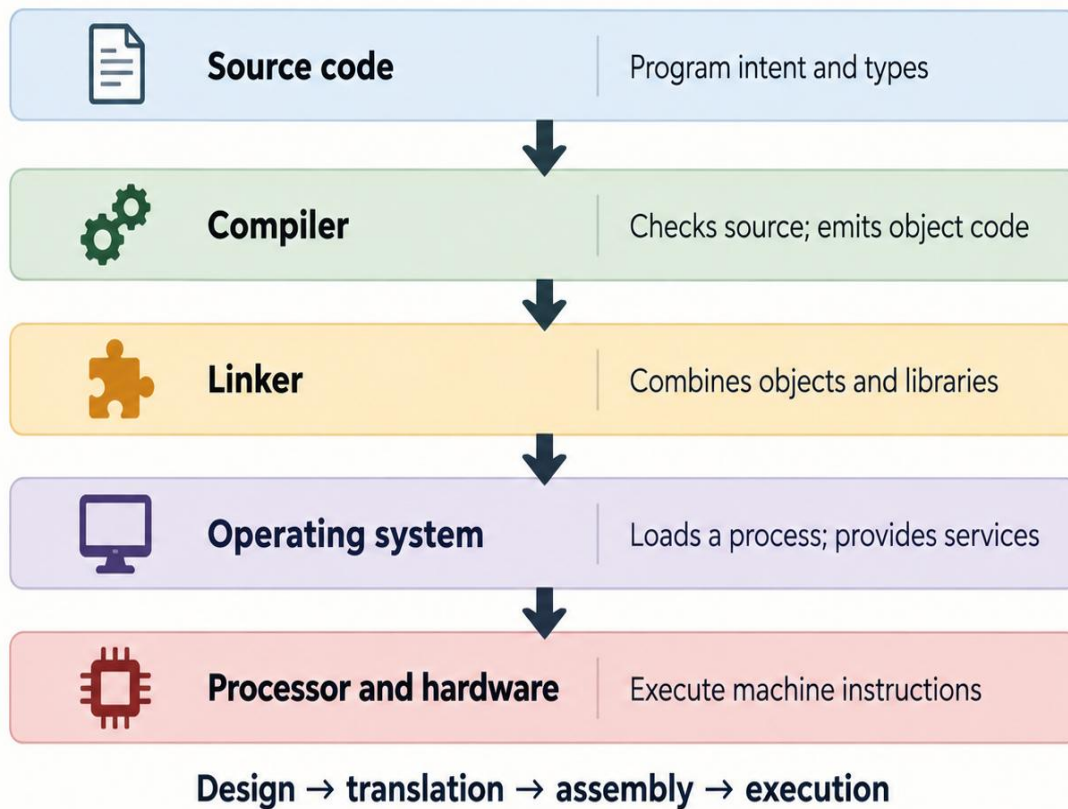
A running C++ program participates in several layers:

1. **Source code** expresses the program in C++.
2. A **compiler** checks the source and converts it into machine-oriented object code.
3. A **linker** combines object code and libraries into an executable program or library.
4. The **operating system** loads the executable, creates a process, and provides services such as files and network access.

5. The **processor and hardware** execute the resulting instructions.

Chapter 1: The C++ Mental Model for Systems Work

1-2 Where a C++ program sits from source to hardware



001

Figure 1-1: Where a C++ program sits from source to hardware

Think of these layers as a delivery chain. The C++ source is the design, the compiler turns the design into manufactured parts, the linker assembles the parts, and the operating system puts the assembled product into operation.

C++ gives you access to both ends of this chain. You can write a high-level type such as a validated network port or a reusable collection, while still controlling the bytes passed to a low-level interface.

Abstraction without avoidable cost

An **abstraction** is an interface that hides details that callers should not need to manage directly. A standard container, for example, can manage dynamically sized storage while presenting operations such as adding an element or asking for the current size.

C++ is strongly influenced by the goal of **zero-overhead abstraction**: a well-designed abstraction should not impose avoidable runtime cost compared with a correct hand-written lower-level implementation. This is a design goal, not a promise that every class or library call is free. You still need to understand what an abstraction does.

A useful comparison is a well-designed power tool. It hides the motor's internal construction, but it does not prevent the operator from choosing the correct bit, speed, or safety procedure. C++ abstractions can remove repetitive mechanics without removing engineering responsibility.

Three questions to ask about every piece of code

1. *What is the type and lifetime of each object?*

An **object** is a region of storage that has a type and a period during which it exists. Its **lifetime** is the interval during which the object may be used according to the language rules.

When reading code, ask:

- What values can this object represent?
- When is it initialized?
- When is it destroyed?
- Can another part of the program still refer to it after destruction?

Many serious C++ defects occur because code uses an object before proper initialization or after its lifetime has ended.

2. *Who owns each resource?*

A **resource** is something that must eventually be released or returned, such as allocated memory, a file, a lock, a socket, or an operating-system handle. **Ownership** is the responsibility for releasing that resource exactly once.

A clear design answers these questions:

- Which object owns the resource?
- Can ownership move?
- Is ownership shared?
- Which parts of the program merely borrow access?
- What event releases the resource?

When nobody owns a resource, it can leak. When several unrelated parts believe they own it, it can be released more than once.

3. *What does each interface read, change, borrow, or own?*

An **interface** is the visible contract through which one part of a program uses another. In C++, parameter and return types should communicate data flow.

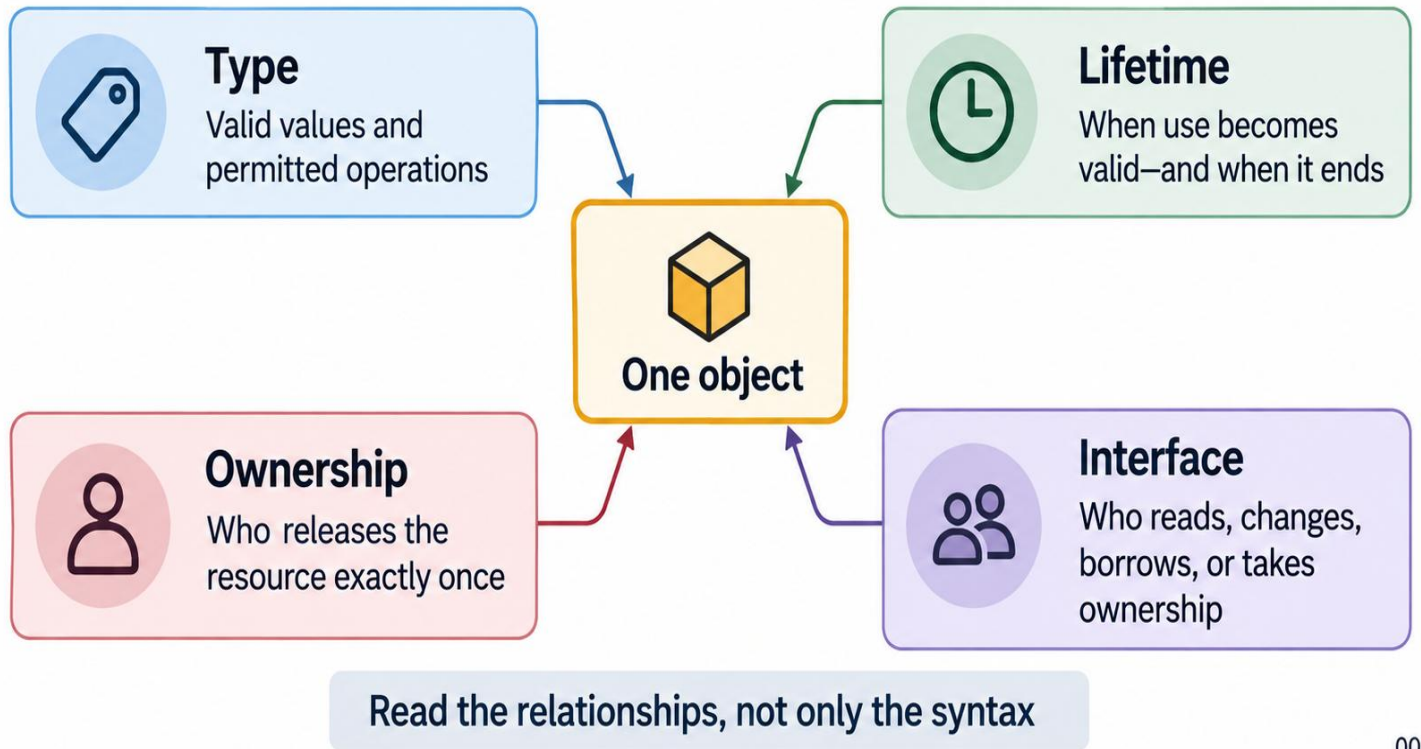
A function should make it reasonably clear whether it:

- Receives its own independent value.
- Reads an existing object without changing it.
- Changes an object supplied by the caller.
- Accepts an optional object that may be absent.
- Takes ownership of a resource.
- Returns a new value or transfers ownership.

Good interfaces reduce the amount of hidden knowledge required to call a function correctly.

Chapter 1: The C++ Mental Model for Systems Work

1-4 Four lenses for reading C++: type, lifetime, ownership, and interface



002

Figure 1-2: Four lenses for reading C++: type, lifetime, ownership, and interface

Modern C++ is not “C with classes”

C and C++ can interoperate, and C++ retains many low-level facilities inherited from C. Modern C++ practice, however, usually favors:

- Values and standard containers instead of manually managed arrays.
- Objects whose destructors release resources automatically.
- Clear ownership types instead of undocumented owning pointers.
- Scoped enumerations instead of unrelated integer constants.
- Library facilities and typed functions instead of unsafe macros.
- Small types that preserve valid state instead of exposing writable internals.

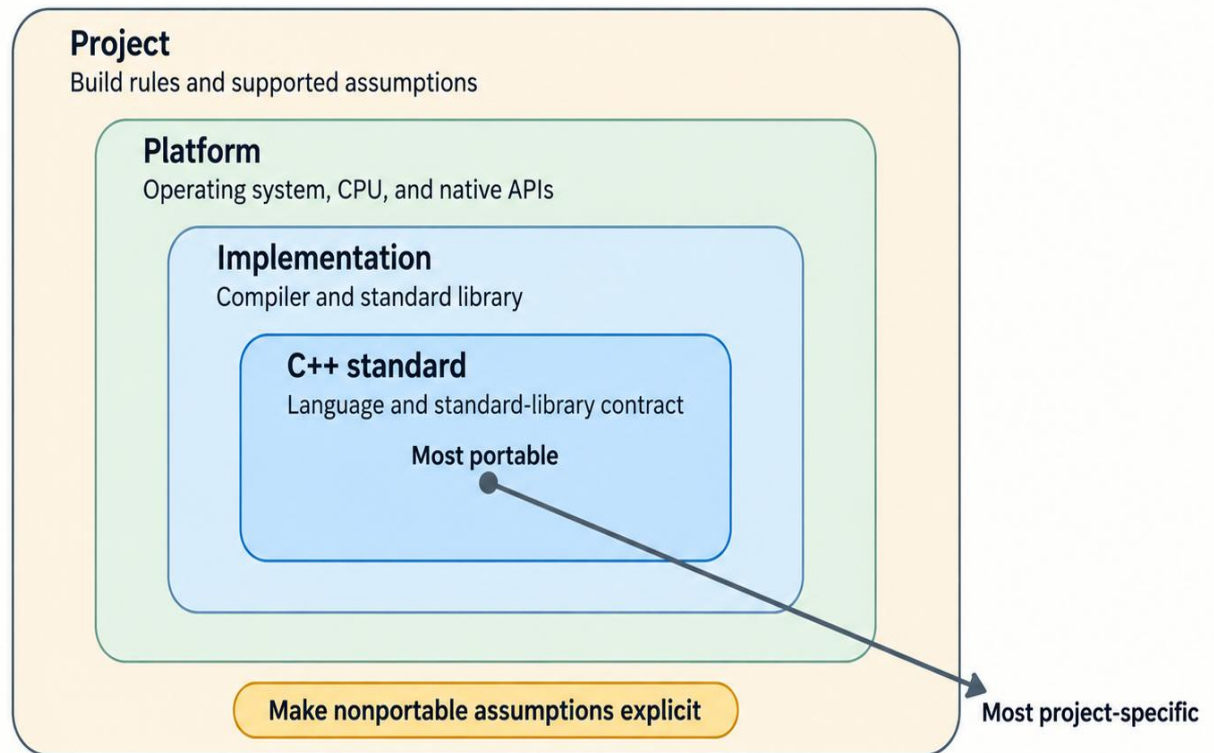
This does not mean that raw pointers, arrays, or C interfaces disappear. Systems engineers encounter them frequently. The important distinction is that they should normally be isolated at boundaries and wrapped in safer C++ interfaces.

Portability has boundaries

The **C++ standard** defines the language and standard-library behavior that conforming implementations must provide. A **C++ implementation** is the combination of compiler, standard library, and platform facilities used to build and run a program.

Chapter 1: The C++ Mental Model for Systems Work

1-6 Standard, implementation, platform, and project boundaries



003

Figure 1-3: Standard, implementation, platform, and project boundaries

Some behavior is portable across conforming implementations. Other behavior depends on the environment:

- The exact size of some integer types.
- File-path and character-encoding conventions.
- Operating-system APIs and handle types.
- Compiler command-line options.
- Processor byte order and alignment details.

A systems engineer should make such assumptions visible rather than silently treating one machine as the universal model.

Three categories of behavior are worth recognizing early:

- **Undefined behavior** occurs when the language places no requirements on what happens, often because a rule such as staying within array bounds was violated.
- **Implementation-defined behavior** allows each implementation to choose and document one permitted behavior.
- A **runtime error** is a failure detected while a validly built program is running, such as being unable to open a requested file.

These categories are different. A program may compile successfully and still contain any of them.

Reading a small program by responsibilities

```
#include <iostream>

int main() {
    int active_workers{3};
    int queued_jobs{8};

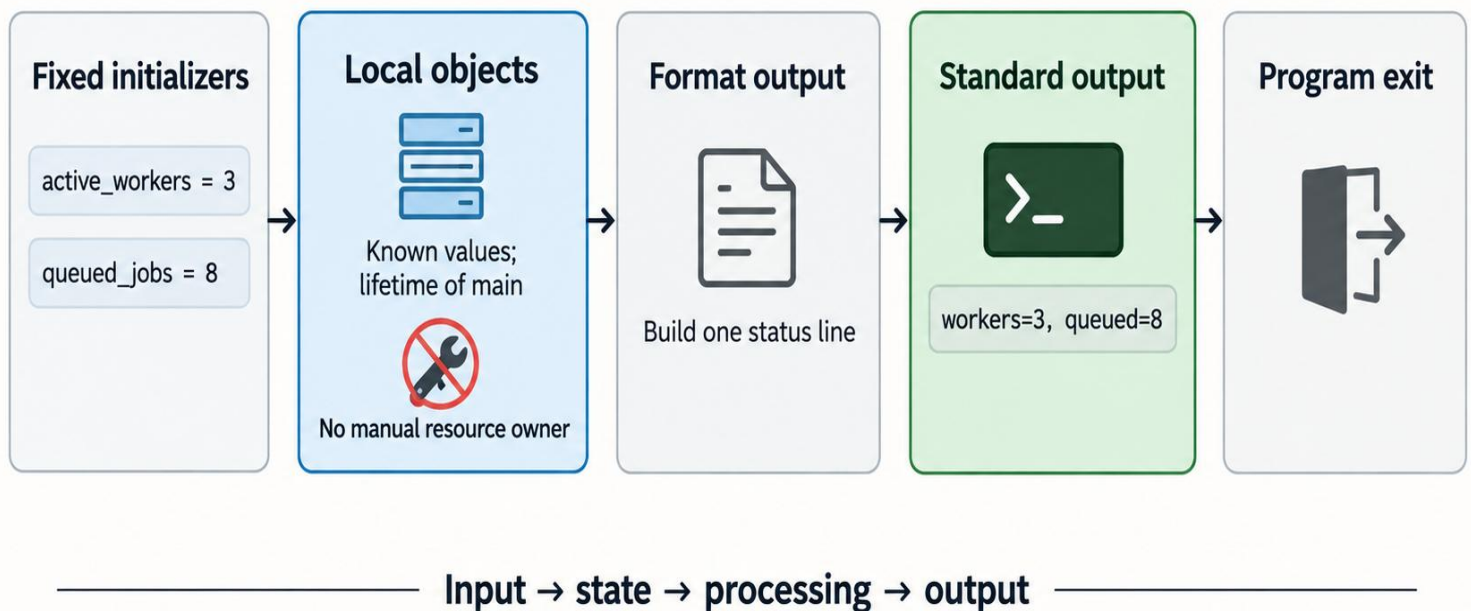
    std::cout << "workers=" << active_workers
               << ", queued=" << queued_jobs << '\n';
}
```

Without knowing every token, you can already identify the important responsibilities:

- The program has no external input; the two initializers provide fixed data.
- `active_workers` and `queued_jobs` are objects that hold integer values.
- Both objects are created when execution enters `main` and destroyed when it leaves.
- The program writes text to **standard output**, the process's normal output stream.
- No manually managed resource appears in this example.
- The program's observable result is one line of text.

Chapter 1: The C++ Mental Model for Systems Work

1-7 Reading a small program by responsibilities



004

Figure 1-4: Reading a small program by responsibilities

The expected output is:

```
workers=3, queued=8
```

This responsibility-first reading style scales. In a larger program, begin by finding inputs, outputs, important objects, resource owners, and lifetime relationships. Explain individual syntax only after you understand the shape of the work.

Misconceptions that cause trouble

“C++ is simply C plus object-oriented syntax.” C++ supports several programming styles, including value-oriented, generic, procedural, and object-oriented design. Modern resource management and standard-library use differ significantly from manual C patterns.

“A higher-level C++ construct must be slower.” A well-designed abstraction may compile to the same or better machine code than a hand-written alternative. Cost depends on the operation, not on whether the source looks abstract.

“If it compiles, it is correct.” Compilation proves only that the program satisfies the checks performed by the compiler for that build. It does not prove correct requirements, valid lifetime relationships, safe input handling, or correct results.

What to retain

Every object has a type and a lifetime. Every resource needs a clear owner. Every interface should reveal how data and ownership flow. These are the main questions that turn unfamiliar C++ from a collection of syntax into an understandable system.

A useful first reading exercise is to take a short program from an existing project and mark only four things: inputs, outputs, resource owners, and borrowed references. Do not try to explain every line on the first pass.

Chapter 2: From Source File to Running Program

What You'll Be Able to Do

By the end of this chapter, you will be able to:

- Trace a source file through preprocessing, compilation, linking, process startup, and program termination.
- Distinguish compiler errors, linker errors, runtime failures, warnings, and silently incorrect results.
- Build and run small C++ programs while checking output, diagnostics, and exit status.

A C++ program becomes executable through separate compilation and linking, so compiler errors, linker errors, runtime failures, and incorrect results must be diagnosed as different classes of problem. Knowing the stage that failed often tells you where to look.

The build pipeline

A typical project contains several kinds of artifact:

Artifact	Purpose
Source file, commonly .cpp	Contains function and object definitions that are compiled.
Header file, commonly .h or .hpp	Contains declarations and definitions that other source files need to see.
Object file, such as .o or .obj	Contains compiled code and metadata from one compiler invocation.
Static or shared library	Packages code for reuse by executables or other libraries.
Executable	Contains a program that the operating system can load and run.

Chapter 2: From Source File to Running Program

2-1 Build pipeline and generated artifacts

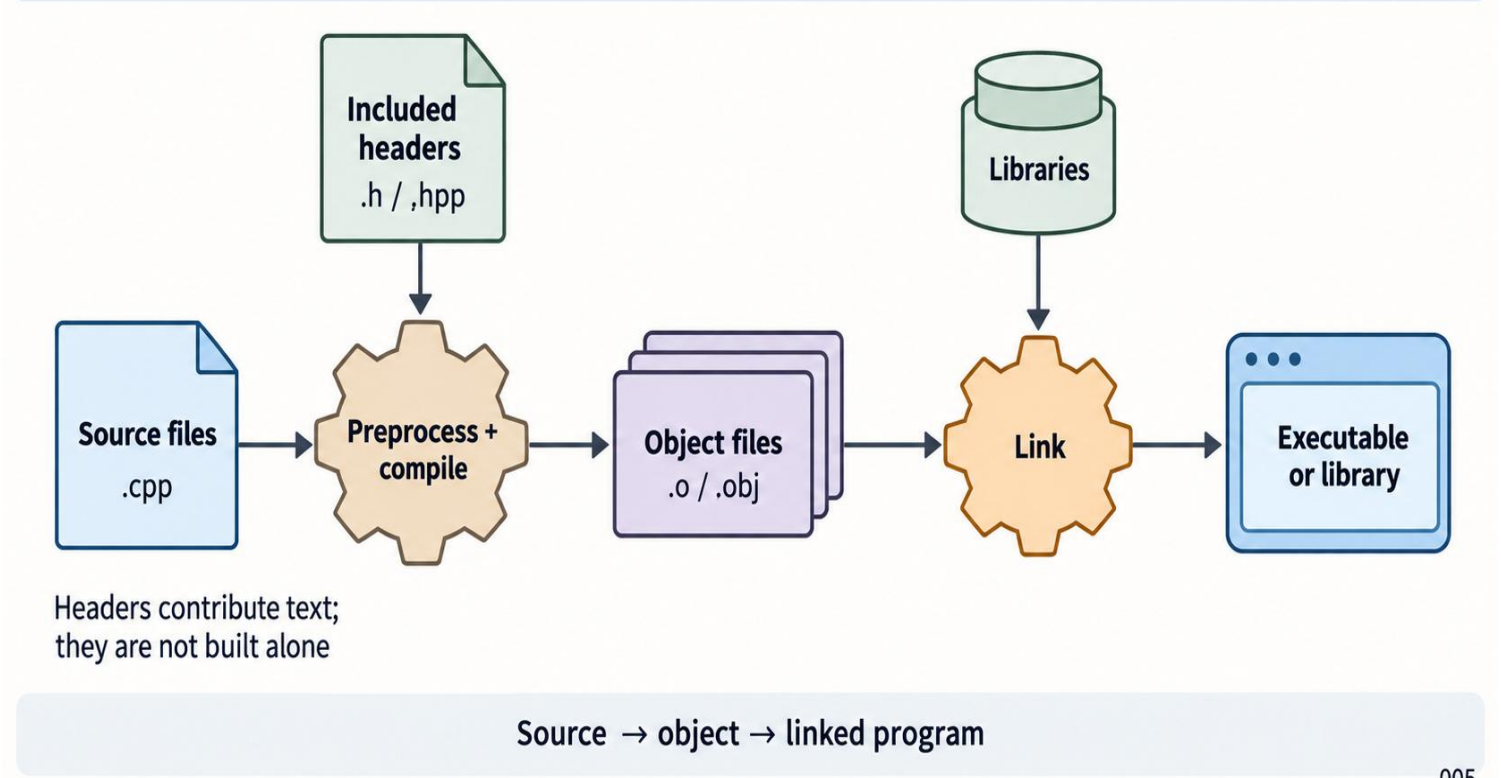


Figure 2-1: Build pipeline and generated artifacts

A **declaration** tells the compiler that a name and type exist. A **definition** supplies the entity itself, such as the body of a function or storage for an object. The distinction becomes important when code is split across files.

The build proceeds conceptually as follows:

1. The preprocessor handles directives such as `#include`.
2. The compiler parses and type-checks one translation unit.
3. The compiler generates an object file.
4. The linker combines object files and libraries.
5. The operating system loads the executable as a process.

A **translation unit** is the source text processed by one compiler invocation after included headers and preprocessing directives have been handled. A project with five `.cpp` files normally has at least five translation units.

Separate compilation matters because each source file can be compiled independently. A change to one implementation file does not necessarily require recompiling every other implementation file, although header changes often affect all source files that include that header.

The smallest complete program

```
#include <iostream>

int main() {
    std::cout << "C++ is running\n";
    return 0;
}
```

`main` is the program's normal entry function in a hosted C++ environment. The operating system starts the process, and the C++ runtime calls `main`.

Chapter 2: From Source File to Running Program

2-2 Process startup, C++ runtime, main, and exit status

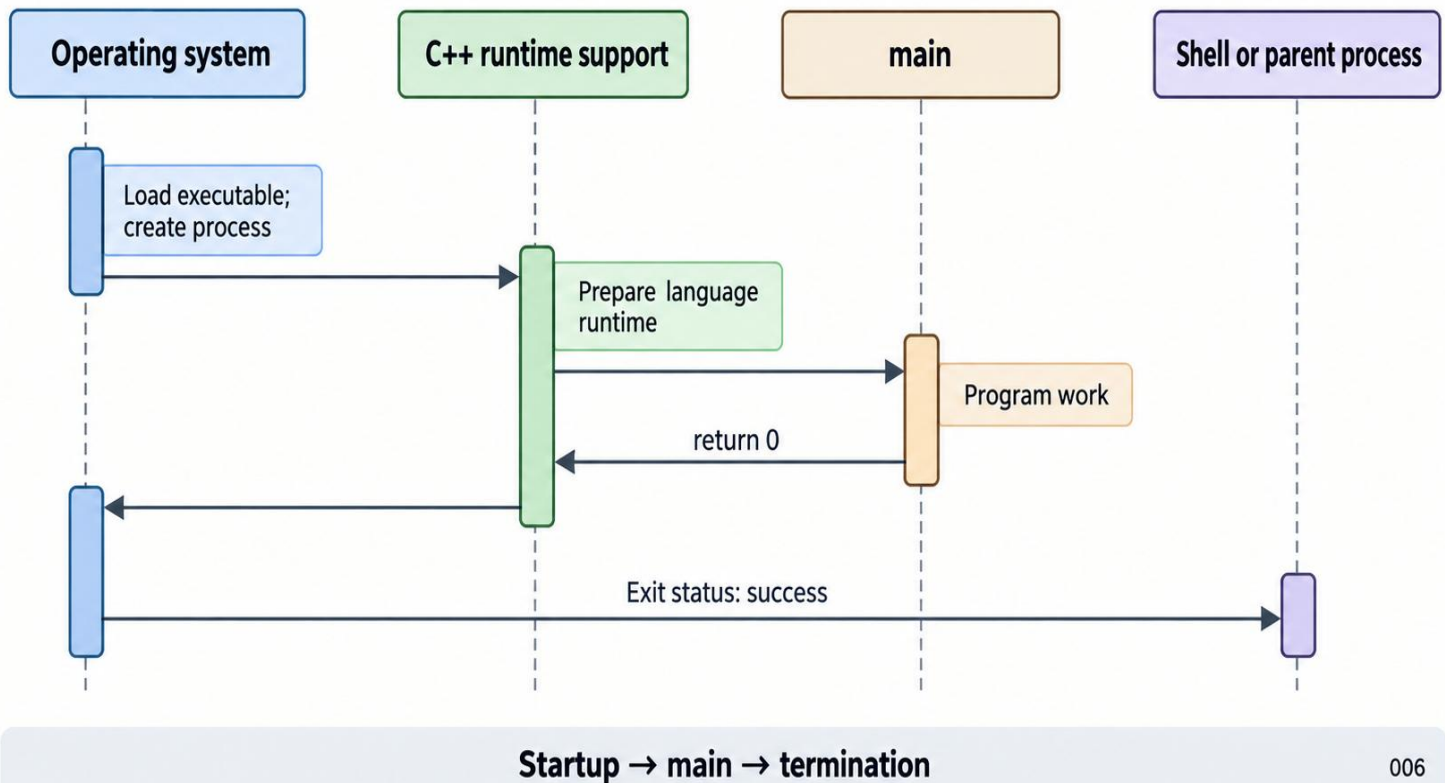


Figure 2-2: Process startup, C++ runtime, main, and exit status

`#include <iostream>` makes standard stream declarations available to this translation unit. The header supplies declarations; the compiler and linker still need the corresponding standard-library implementation.

`std::cout` represents standard output. The insertion operator `<<` sends the following value to that stream.

`return 0;` reports successful completion. Reaching the end of `main` has the same success meaning, but an explicit return is useful while first learning the process exit status.

Build and run with GCC or Clang

Prerequisite: A command shell and a GCC or Clang C++ compiler available as `g++` or `clang++`. Save the program as `hello.cpp`.

Purpose: Compile one C++20 source file with useful warnings and debug information.

GCC example:

```
g++ -std=c++20 -Wall -Wextra -Wpedantic -g hello.cpp -o hello
```

Clang example:

```
clang++ -std=c++20 -Wall -Wextra -Wpedantic -g hello.cpp -o hello
```

The important options are:

- `-std=c++20` selects the C++20 language mode.

- `-Wall -Wextra -Wpedantic` enables useful diagnostic warnings.
- `-g` records debug information for a debugger.
- `-o hello` names the resulting executable `hello`.

Expected result: A successful build normally prints no diagnostics and creates an executable named `hello`.

On a Unix-like shell, run it with:

```
./hello
```

Expected output:

```
C++ is running
```

The shell variable containing the previous process's exit status is environment-specific. In many Unix-like shells, the following displays it:

```
echo $?
```

The expected value is 0.

One MSVC example

Environment-specific example: On Windows, a Visual Studio Developer Command Prompt can provide Microsoft's `cl` compiler.

Prerequisite: A configured Developer Command Prompt and the same source file named `hello.cpp`.

```
cl /std:c++20 /W4 /EHsc /Zi hello.cpp /Fe:hello.exe
```

`/std:c++20` selects C++20, `/W4` enables a high warning level, `/EHsc` selects the usual C++ exception model for this environment, `/Zi` emits debug information, and `/Fe` selects the executable name.

Expected result: The command creates `hello.exe`, which prints the same line when run.

Compiler option names are not part of the C++ language. They vary by toolchain and project policy.

What the compiler does

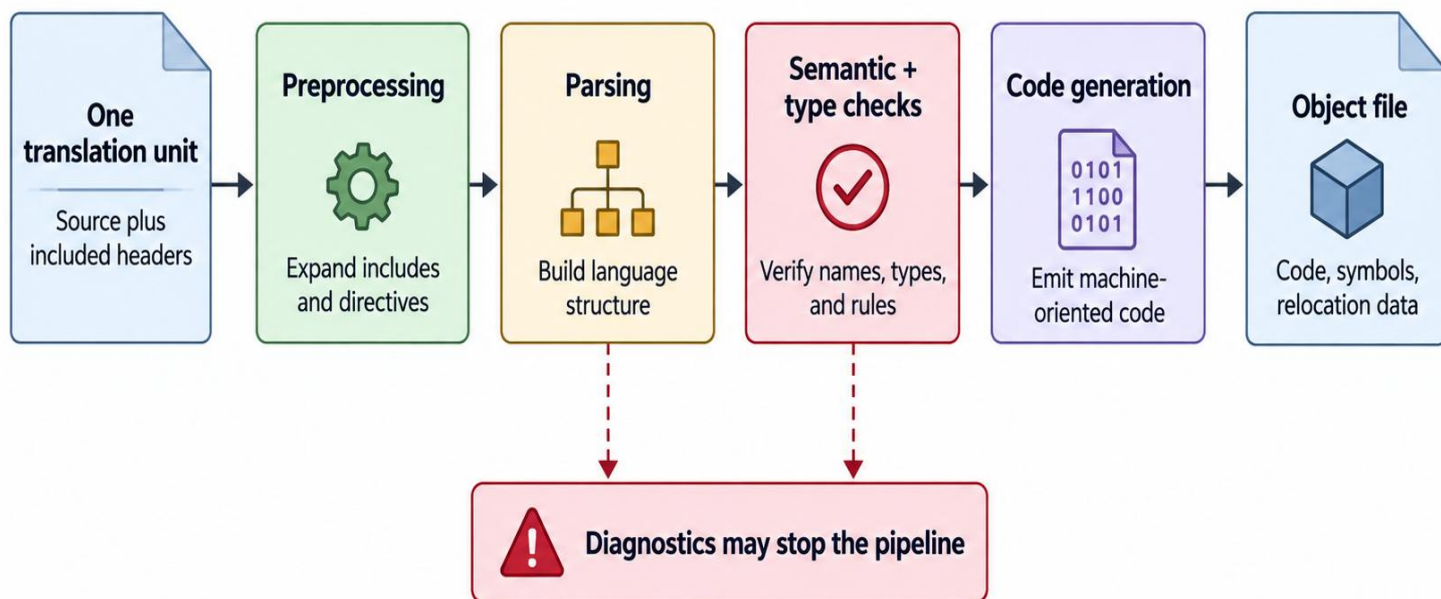
The compiler performs several related jobs:

- It parses tokens according to C++ grammar.
- It resolves names and checks types.
- It verifies many language rules.
- It may optimize the program while preserving required observable behavior.

- It emits machine-oriented object code and metadata.

Chapter 2: From Source File to Running Program

2-5 What the compiler does internally



007

Figure 2-3: What the compiler does internally

An **optimization** is a transformation intended to improve properties such as execution speed or code size without changing the behavior required by the language. Optimization does not repair incorrect code. In fact, undefined behavior may become more visible in optimized builds because the compiler is allowed to assume that valid programs do not perform undefined operations.

What the linker does

The compiler can accept a call to a function after seeing its declaration, even when the function body is in another translation unit. The linker later resolves the call to the function's definition.

Chapter 2: From Source File to Running Program

2-6 How the linker resolves declarations to definitions

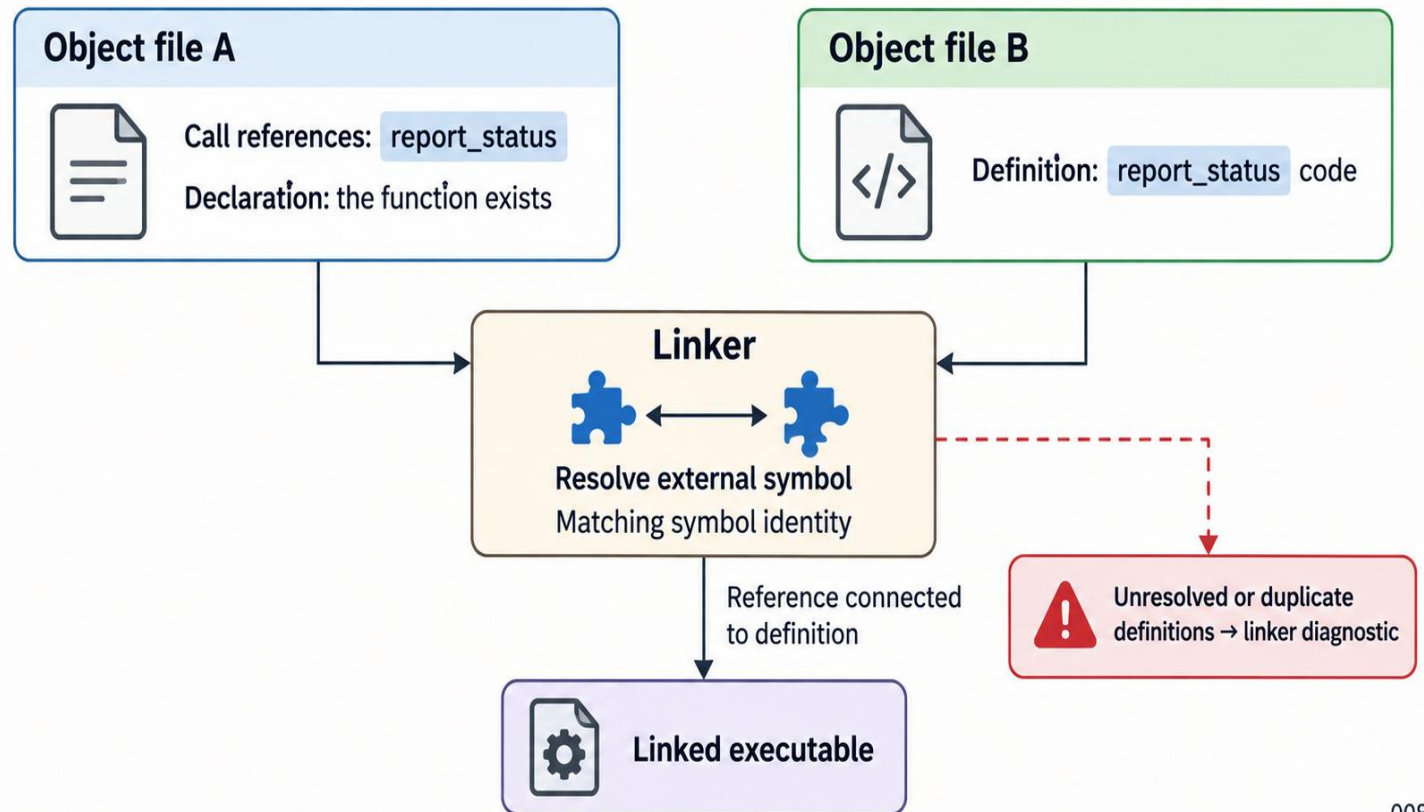


Figure 2-4: How the linker resolves declarations to definitions

Consider three files.

status.hpp:

```
#ifndef STATUS_HPP
#define STATUS_HPP

int active_worker_count();

#endif
```

status.cpp:

```
#include "status.hpp"

int active_worker_count() {
    return 3;
}
```

main.cpp:

```
#include "status.hpp"
#include <iostream>

int main() {
    std::cout << active_worker_count() << '\n';
}
```

```
}
```

Prerequisite: GCC or Clang and the three files in the current directory.

Purpose: Compile the two source files separately, then link their object files.

```
g++ -std=c++20 -Wall -Wextra -Wpedantic -g -c status.cpp -o status.o
g++ -std=c++20 -Wall -Wextra -Wpedantic -g -c main.cpp -o main.o
g++ status.o main.o -o status_app
```

The `-c` option compiles without linking. The final command links the two object files.

Expected result: Running `./status_app` prints:

```
3
```

The include guard formed by `#ifndef`, `#define`, and `#endif` prevents the header's contents from being processed more than once in a translation unit. Header organization is covered in depth later; for now, notice that both source files see the same function declaration.

Four distinct failure classes

Failure stage	Typical symptom	First place to investigate
Compile time	Syntax error, unknown name, or type mismatch	The reported source line and related declarations
Link time	Unresolved symbol or duplicate definition	Missing object/library, inconsistent declaration, or repeated definition
Runtime	Crash, failed operation, or reported error after launch	Inputs, resource state, lifetime, and runtime diagnostics
Logical result	Program runs but produces the wrong answer	Requirements, algorithm, assumptions, and tests

A compiler error

This program attempts to initialize an integer with text:

```
int main() {
    int worker_count{"three"};
}
```

The compiler rejects it because the initializer is not convertible to `int`. The program never reaches linking.

A linker error

Suppose `main.cpp` calls `active_worker_count`, but the build omits `status.o`:

```
g++ main.o -o status_app
```

The compiler has already accepted the declaration. The linker now reports an unresolved reference because it cannot find the definition.

A **symbol** is a linker-visible name representing code or data. Toolchains describe missing symbols with different wording, such as “undefined reference” or “unresolved external symbol.”

A runtime failure

This example builds successfully but performs invalid division when the denominator is zero:

```
#include <iostream>

int main() {
    int denominator{0};
    std::cout << 10 / denominator << '\n';
}
```

Integer division by zero is undefined behavior, not a guaranteed, cleanly reported error. The program may terminate, but the language does not require one specific outcome. This illustrates why successful compilation is not enough.

A wrong result without an error message

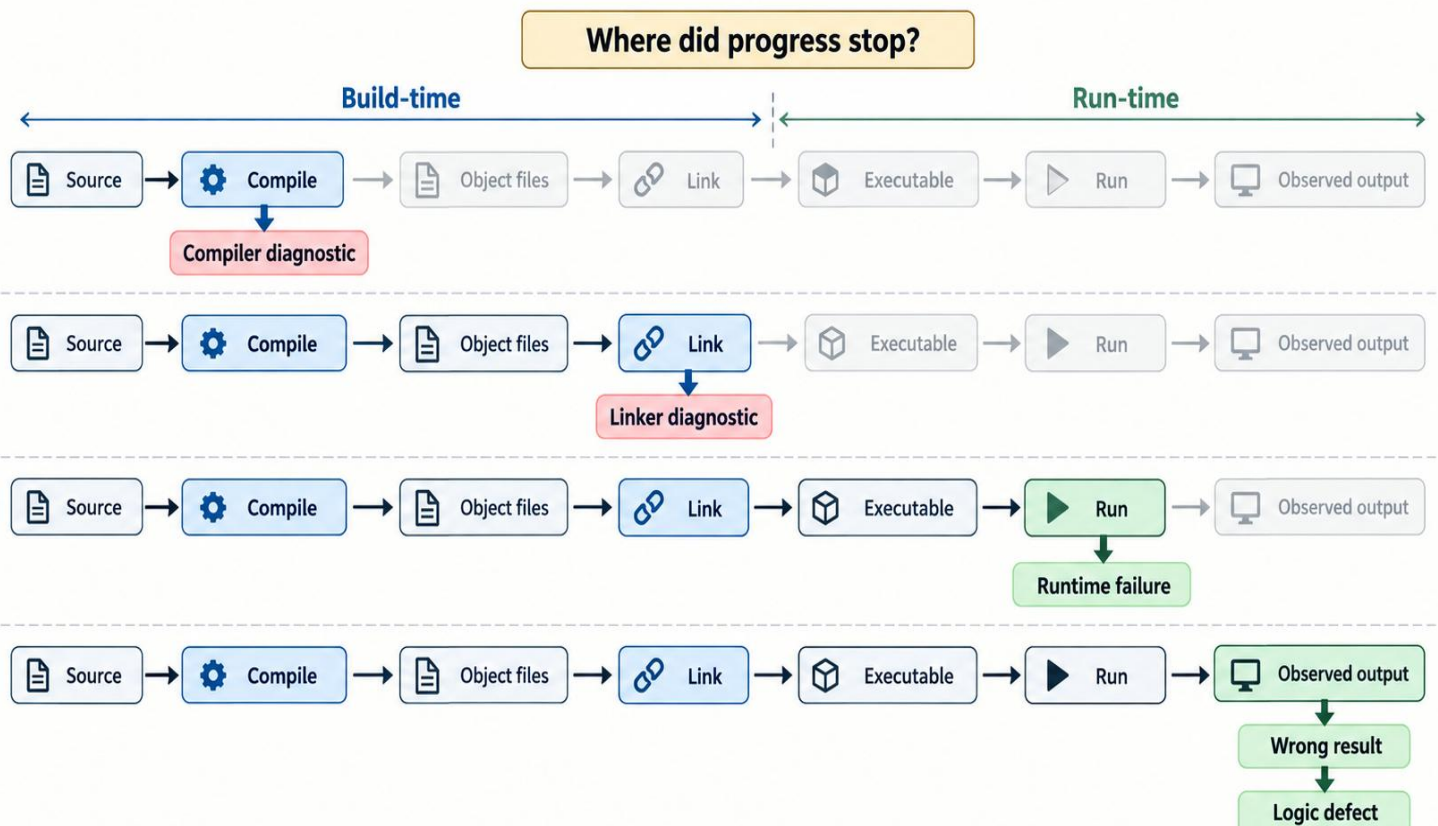
```
#include <iostream>

int main() {
    int completed{8};
    int total{10};
    int percent{completed / total * 100};
    std::cout << percent << "%\n";
}
```

The output is 0%, because integer division evaluates $8 / 10$ as 0 before multiplication. Every build stage succeeds. The defect is in the calculation.

Chapter 2: From Source File to Running Program

2-7 Compiler, linker, runtime, and logic failure classes



009

Figure 2-5: Compiler, linker, runtime, and logic failure classes

main, arguments, and exit status

A command-line program can receive arguments:

```
#include <iostream>

int main(int argc, char* argv[]) {
    std::cout << "argument count: " << argc << '\n';

    if (argc > 1) {
        std::cout << "first argument: " << argv[1] << '\n';
    }
}
```

```

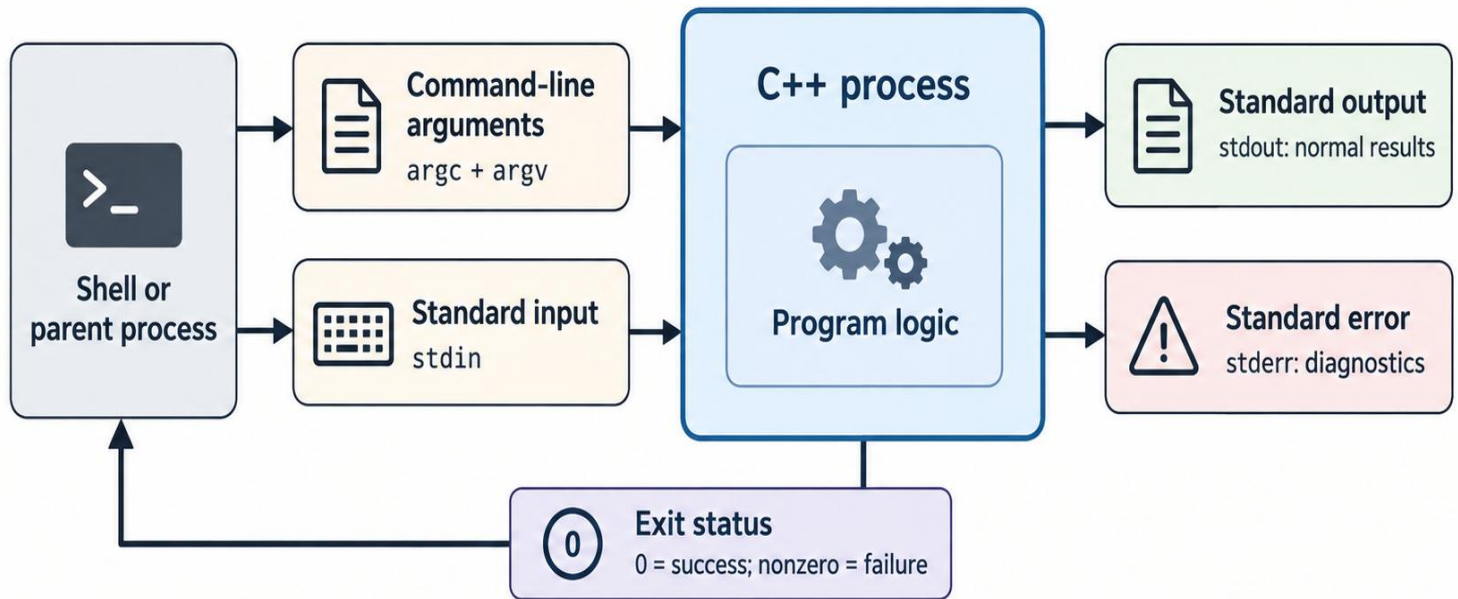
}

return 0;
}

```

Chapter 2: From Source File to Running Program

2-8 Command-line arguments, standard streams, and exit status



Inputs cross in; results and status cross out

010

Figure 2-6: Command-line arguments, standard streams, and exit status

`argc` is the number of argument strings, including the program name. `argv` points to an array of null-terminated character strings. These strings come from the process environment and must be treated as untrusted input.

For now, the key rules are:

- Check `argc` before indexing `argv`.
- Parse and validate text before using it as a number, path, or size.
- Use a nonzero exit status to indicate failure to the calling environment.

Detailed input parsing belongs later in the learning sequence.

Warnings are engineering information

A **warning** is a diagnostic for code that the compiler can translate but that may indicate a defect or nonportable assumption. Useful warnings can reveal:

- A local value that might be uninitialized.
- A signed-to-unsigned conversion.
- An unused result or variable.
- A suspicious condition.
- Unreachable code.

Warnings should be reviewed, not suppressed reflexively. Many teams treat selected warnings as build failures once their codebase is clean. The exact warning policy is a project decision because compilers and versions differ.

Debug information and optimization levels

Debug information maps machine code back to source-level names and lines so that a debugger can show variables and call frames. It usually increases artifact size but need not change the language behavior.

Compilers also provide optimization levels. A low-optimization debug build is often easier to inspect because statements and variables more closely resemble the source. A production build may be optimized and therefore harder to step through. Both should be tested because some defects appear only with particular timing, layout, or optimization assumptions.

A practical diagnosis sequence

When a build or run fails, identify the stage before changing code:

1. Did preprocessing or compilation fail?
2. Were all expected object files produced?
3. Did linking fail?
4. Was the executable rebuilt after the last source change?
5. Did the correct executable actually run?
6. Did the process fail, or did it merely produce an incorrect result?

Chapter 2: From Source File to Running Program

2-11 Stage-first diagnosis sequence

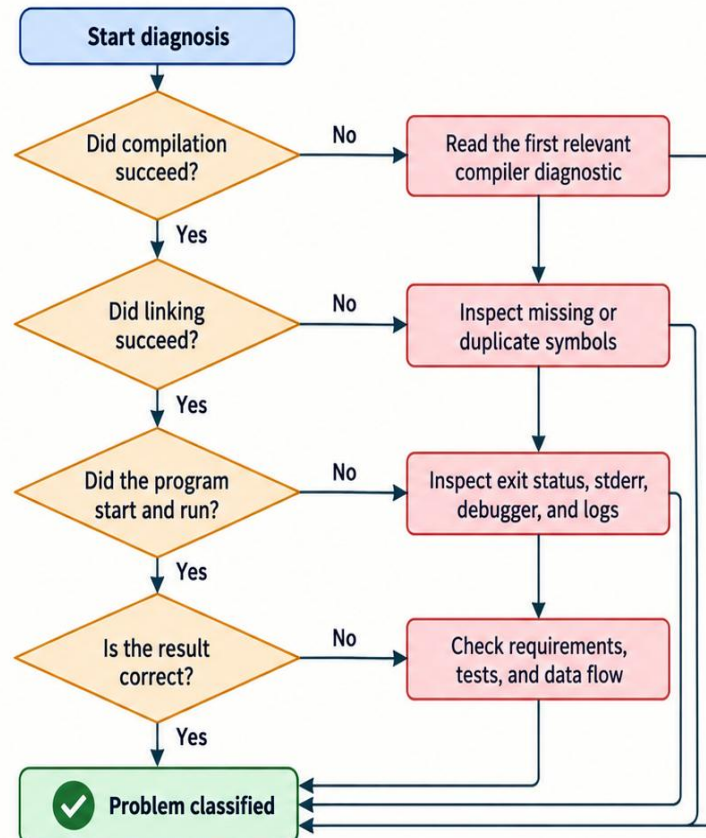


Figure 2-7: Stage-first diagnosis sequence

This sequence prevents a common waste of time: debugging source code while accidentally running an older executable left behind by a failed build.

Checkpoint

Compilation checks and translates one translation unit; linking connects compiled definitions; running exercises the result in a process. Keep these stages separate in your mental model.

A useful exercise is to build the two-file example, then deliberately perform three changes: misspell a type name, omit one object file from the link command, and change the function's returned value without rebuilding. Observe how the failure signal differs in each case.