CS 247

Software Engineering Principles

Chapter 4

Modules

Victoria Sakhnini

Table of Contents

Introduction

Real software is never a single file. A C++ program of any meaningful size is built from many source files that the developer writes, which the compiler processes one by one, and the linker eventually fuses them into a single executable. The mechanics of that fusion, how a function is defined in rational.cpp becomes callable from main.cpp, and what goes wrong when the rules are violated is the subject of this chapter.

In Chapter 2, we drew a clear line between an ADT's interface and its implementation. We argued that clients should depend only on the interface. C++ enforces that separation through a build model that splits each module into a header file (the interface) and a source file (the implementation), compiling each source file independently before linking them together. The same machinery that supports separate compilation also introduces new failure modes, including duplicate inclusions, circular dependencies, and link errors from forgotten externs, which you will encounter for the first time when you split a working program into two files. Understanding the build pipeline turns those failures from mysterious into mechanical.

By the end of this chapter, you should be able to draw the module-dependency diagram for a small program, write a correct header guard, decide when a forward declaration suffices instead of an #include, and write a Makefile that rebuilds only the minimum set of files after an edit.

Program Decomposition

Decomposing a program into multiple source files is not a stylistic choice; it is what makes a non-trivial program manageable. Four practical wins come from doing it:

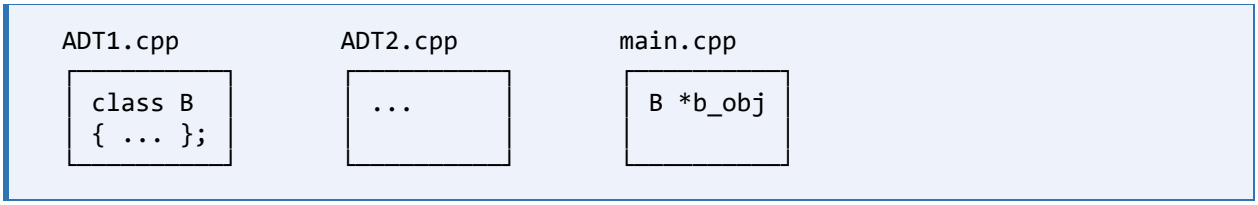
- **Independent development.** Different team members can work on different modules in parallel without stepping on each other's edits.
- **Separate compilation.** Each source file is compiled independently to an `.o` (object) file, then the linker stitches the objects together. After an edit, you only recompile what changed, not the whole program.
- **Reuse.** A module that defines a useful data type or function can be linked into many different programs.
- **Incremental development.** You can build, test, and ship a working subset of the program, then grow it module by module.

The goal of decomposition is the **separation of concerns**: each source file should own one cohesive piece of functionality. In practice, though, source files are never completely independent. A function in main.cpp calls a constructor defined in rational.cpp; a class in graph.cpp stores a std::vector<Node> defined in node.cpp. As soon as one file mentions a name introduced in another, the compiler has work to do.

💡 **Tip:** The C++ compiler will refuse to compile a source file that references names it has not seen declared. Every name a file uses must be either defined within that file, or declared in the file before its first use. The rest of this chapter is, essentially, the answer to the question "how do we get those declarations in front of the compiler without manually copy-pasting them into every file?"

Names Across Modules

Suppose we have three source files: ADT1.cpp defines a class B; ADT2.cpp defines another class; and main.cpp wants to use a pointer to B. The picture looks like this:



When the compiler processes main.cpp, it encounters the line B *b_obj; without having been told what B is. It will issue an error and stop. The same problem arises whenever one file refers to classes, variables, types, functions, or constants defined in another file. The cure is to declare the name before it is used. The wrinkle is finding a way to do that that scales beyond toy programs.

Global Declarations

C++ provides syntax for declaring (as opposed to defining) every kind of named entity. Here is the full menu:

```
// Class declaration (forward declaration, no body)

class C;

// Class definition, has the body

class C {
    public:  ...
    private: ...
};

// Function prototype

void f(int, int);
```

```
// Global variable declaration, extern says "defined elsewhere."

extern int i;

// Constant declaration, extern needed because constants
// are normally local to a file

extern const int j;
```

Each of these tells the compiler *a name with this signature exists somewhere*. The compiler can now type-check expressions that use the name, even though it has not yet seen the body or the storage. The linker takes over at the end to ensure that each declared name is actually defined in some file.

Declarations vs. Definitions

This is the single most important distinction in the C++ build model. Get it right, and most of what follows is bookkeeping. Get it wrong, and you will spend afternoons chasing "undefined reference" and "multiple definition" errors.

	Declaration	Definition
What it does	Announces the existence of a name and its type to the compiler.	Allocates storage (for variables) or generates code (for functions).
How often	May appear in many places, in many files.	Exactly once in the entire program.
Compiler/linker	Multiple identical declarations are fine.	A second definition is a linker error.
Example (var)	`extern int x;`	`int x = 42;`
Example (fn)	`int sum(int, int);`	`int sum(int a, int b) { return a+b; }`
Example (class)	`class Rational;`	`class Rational { ... };`

Two rules govern the interplay:

1. **All declarations of the same name must agree.** If one file says `extern int x;` and another says `extern double x;`, the compiler will not necessarily catch the mismatch (they are in different translation units), but the linker may, and even if it does not, the program is ill-formed and the behaviour is undefined.
2. **Each definition appears once.** If you write `int x = 0;` in two different source files, both files compile, but when the linker tries to merge them it finds two definitions for the same symbol and reports a multiple definition of `x` error.

❓ **Question:** Is a class body in a header file a declaration or a definition?

It is a **definition** of the class type, but it does not allocate storage; there is no storage to allocate for a type. That is why the one-definition rule for class types is relaxed: the *same* class definition may appear in multiple translation units (because every `#include` duplicates it), as long as every copy is textually identical. The definitions of the class's member functions, on the other hand, follow the usual rule: exactly one per program.

Constants and extern

Constants get special treatment that often surprises newcomers. By default, a const defined at file scope is *local to that file*, it has internal linkage. Two different .cpp files can each contain const double PI = 3.14159; with no conflict, because each PI is a private file-scope constant. The same name from one file is simply invisible to the other.

If you want a single shared constant, you must promote it to external linkage with extern:

```
// constants.h
extern const double PI;     // declaration: "defined elsewhere"

// constants.cpp
extern const double PI = 3.14159;   // the one definition

// any other .cpp file
#include "constants.h"
double area = PI * r * r;   // uses the shared definition
```

💡 **Tip:** Normally, the compiler optimizes `const`s by folding their value directly into expressions, with no storage allocated. The moment you write `extern`, you force the compiler to allocate real storage so the linker has a symbol to resolve across translation units. That is a small price for a single source of truth.

⚠️ **Warning:** If you forget `extern` on the declaration in the header and put the definition in a `.cpp` file, every `.cpp` that includes the header gets its own private copy of `PI`. The program compiles, links, and is wrong: changes to one copy do not propagate, addresses do not match, and tests that compare pointers fail mysteriously.

Two Approaches to Sharing Names

We have established the problem: any file that uses a name must see a declaration of that name first. The remaining question is how to deliver those declarations. There are two strategies; only one of them is used in practice, but the broken one is worth understanding because it explains why we adopted the other.

Approach #1: Declare-What-You-Need

The naive approach is to ask every programmer to manually declare every external name their module uses. If main.cpp uses class B from ADT1.cpp, the programmer of main.cpp writes the declaration itself:

```
// main.cpp
class B {                       // declaration copy-pasted
    public:
        B();
        void render();
    private:
        int data_;
};

int main() {
    B b;
    b.render();
}
```

This works in principle: the compiler sees a complete class declaration, type-checks the uses of b, and produces an object file with unresolved references to `B::B()` and `B::render()` that the linker will resolve against ADT1.o. In practice, it is unusable for two reasons:

- **It is error-prone.** Every client of `B` duplicates the declaration. If B's author changes the signature of render(), every duplicated declaration must be updated. Miss one, and the linker either silently picks the wrong overload or produces a confusing error far from the actual mistake.
- **It is time-consuming.** Real classes have many members. Copying a fifty-line class declaration into every client file is busywork that no one will do correctly.

The lesson, which we will see again in this course, is that any rule the language requires programmers to follow by hand will eventually be violated. The right move is to mechanize the work.

Approach #2: Header Files

The standard solution is to put every declaration that *other* modules need into a **header file**, and ask each client to #include that file. A header is just a text file (conventionally with the extension .h or .hpp) that contains declarations, class definitions without member-function bodies, function prototypes, extern variable declarations, type aliases, and so on.

The **preprocessor**, invoked automatically before the compiler proper, replaces each #include "foo.h" directive with the textual contents of foo.h. By the time the compiler sees the source file, every name it uses has been declared.

```
// adt1.h , interface, shared with all clients

#ifndef ADT1_H
#define ADT1_H

class B {
    public:
        B();
        void render();
    private:
        int data_;
};

#endif

// adt1.cpp, implementation, owns the definitions
#include "adt1.h"

B::B() : data_(0) {}
void B::render() { /* ... */ }

// main.cpp, client, only needs the interface
#include "adt1.h"

int main() {
    B b;
    b.render();
}
```

Two files now know about B via one shared declaration. If B's author changes the signature of render(), they update adt1.h once, every client gets the new declaration on the next build, and the compiler catches mismatches at compile time. The duplication has been pushed into the preprocessor, where machines do not make mistakes.

Benefits of Header Files

The benefits of header files break down into four:

- **Abstract view of the module.** The header is what clients read to learn how to use the module. It is the module's API document. Member-function bodies, helper functions, and `static` data live in the `.cpp` file, are invisible to clients, and can be changed freely.
- **Saves work.** One declaration serves every client. Maintaining N copies of a class declaration requires O(N) effort; maintaining a single header requires O(1) effort.
- **Ensures consistency.** Every client sees the same declaration, so there is no chance of one client compiling against one signature and another compiling against an incompatible one.
- **Eases maintenance.** A signature change is a single edit. The compiler will then point out every place that still relies on the old signature.

The header also reinforces the chapter's broader theme: it separates the interface (what the module promises) from the *implementation* (how it delivers). A well-written header contains only what the client needs: class definition, public member declarations, type aliases, free function prototypes, and

nothing about how any of it works. It specifies the size and layout of objects (so the compiler can allocate them), the signatures of operations (so the compiler can type-check calls), and the names of free functions; it does not specify their bodies or default values.

> 💡 **Tip:** A useful test: if you change the body of a member function without changing its signature, only the `.cpp` that contains it should need to be recompiled. If the change forces a recompile of every client, your header is exposing too much.

Duplicate Header Inclusions

Header files solve the consistency problem, but immediately create a new one. Consider this small dependency picture:

```
// adt1.h           // adt2.h             // adt2.cpp
class B { ... };    #include "adt1.h"     #include "adt1.h"
int bar();          class A {             #include "adt2.h"
extern int k;           B b_;             // implementation
                    };
```

When the compiler processes adt2.cpp, the preprocessor expands the two `#include` directives. The first pulls in the contents of adt1.h directly. The second pulls in adt2.h, which itself contains `#include "adt1.h"`, so the contents of adt1.h are pasted into the translation unit *a second time*. The result, after preprocessing, looks like this:

```
// adt2.cpp after preprocessing

class B { ... };            // from adt1.h, first time
int bar();
extern int k;
class B { ... };            // from adt1.h, second time (via adt2.h)
int bar();
extern int k;
class A {
    B b_;
};

// implementation
```

Three problems follow:

- **Class redefinition.** C++ does not allow a class type to be redefined in the same translation unit, even with the identical body. The compiler reports a redefinition error.
- **Variable redefinition.** If any header contains a **definition** rather than just a declaration (an `inline` variable, a class with a defined static member, an enumeration), the second copy is also a redefinition.
- **Compile-time cost.** Even when nothing redefines, parsing the same declarations repeatedly bloats compile times. A real header network can expand into thousands of redundant lines per translation unit if no protection is in place.

Solving all three requires teaching the preprocessor to *include each header at most once per translation unit*. That is what header guards do.

The Preprocessor

Before the C++ compiler proper ever sees your code, the **preprocessor** runs a textual pass over it. Its job is small but essential: it strips out comments, expands `#include` directives, substitutes `#defined` names, and evaluates `#if`/`#ifdef` conditionals to decide which lines of code to keep and which to skip. The preprocessor is not aware of C++ syntax; it does not know what a class is, nor does it parse expressions. It is a glorified copy-and-paste engine.

Preprocessor directives all start with # in the first column (whitespace before # is permitted but discouraged). Names introduced by `#define` are called **preprocessor variables**, or more commonly, **macros**.

> 💡 **Tip:** *A preprocessor variable, more commonly called a macro, is a name defined with* `#define` that the preprocessor textually substitutes throughout your source code before the compiler ever sees it. It is not a real variable: it has no type, no address, no storage, and does not exist at runtime.

Preprocessor Directives

The directives that matter for header management are:

```
#define FLAG              // define a macro with no value
#define PI 3.14159        // define a macro with a value
#undef FLAG               // remove a previously-defined macro

#ifdef FLAG               // is FLAG currently defined?
    // ... compiled only if FLAG is defined
#endif

#ifndef FLAG              // is FLAG currently undefined?
    // ... compiled only if FLAG is not defined
#endif
```

```
#if defined(FLAG) && !defined(OTHER)
    // ... compiled if the boolean expression is true
#endif
```

These directives let us conditionally include text. The skipped branches are not just hidden from the compiler; they are deleted before the compiler sees the file, which means they do not need to be syntactically valid C++. You can use this to switch debug code on and off, to compile platform-specific bodies, or, most importantly, to guard headers.

Header Guards

The standard idiom is to wrap the entire contents of every header in an `#ifndef` / `#define` / `#endif` triple that uses a unique macro name:

```
// rational.h
#ifndef RATIONAL_H
#define RATIONAL_H

class Rational {
    // ...
};

Rational operator+(const Rational&, const Rational&);

#endif   // RATIONAL_H
```

The mechanism is simple. The first time a translation unit sees this header, the macro `RATIONAL_H` is not defined, so the `#ifndef` branch runs: the macro gets defined, and the class declaration is included. Any later `#include "rational.h"` within the same translation unit finds the macro already defined, so the `#ifndef` branch is skipped and the body is excluded.

That is the trick. The header is included logically once per translation unit, regardless of how many other headers request it.

> **Tricky:** The guard macro name RATIONAL_H is *global to the entire build*. If two different headers in the project happen to choose the same guard name, the second one will be silently skipped wherever both are included, and the resulting linker errors will be baffling. The convention is to derive the macro name from the most distinctive entity in the header (the class name) rather than from the file name, e.g. RATIONAL_H not RAT_H. For larger projects, prefix the name with the project name: CS247_RATIONAL_H.

⚠ **Warning:** The #define inside the guard must immediately follow the #ifndef, typos like #ifndef RATIONAL_H / #define RATIONAL_HH leave the guard non-functional and re-inclusion silently breaks again. Pair the names with care.

Circular Dependencies

The Problem

A **circular dependency** arises when two headers each need to know about the type defined in the other. Imagine modelling a small graph: a Node has a list of Edges leaving it; an Edge has pointers to its source Node and target Node. If we naively put each class in its own header and each header #includes the other, we get:

```
// node.h
#ifndef NODE_H
#define NODE_H
#include "edge.h"
class Node {
    std::vector<Edge> edges_;
};
#endif

// edge.h
#ifndef EDGE_H
#define EDGE_H
#include "node.h"
class Edge {
    Node *src_;
    Node *dst_;
};
#endif
```

Suppose we compile node.cpp, which #includes node.h. The preprocessor starts expanding node.h: macro NODE_H is not yet defined, so it enters the branch, defines NODE_H, and hits #include "edge.h". The preprocessor starts expanding edge.h: macro EDGE_H is not yet defined, so it enters, defines EDGE_H, and hits #include "node.h". Now NODE_H *is* defined, so the include of node.h expands to nothing.

The compiler then continues parsing edge.h and reaches class Edge { Node *src_; ... }. But the class Node has not been declared at this point in the file; its declaration is later in node.h and gives up.

Header guards prevent infinite expansion, but they do not solve the underlying problem: each header needs information that the other header has not yet provided.

Forward Declarations

The fix is to break the cycle by introducing a **forward declaration**, a one-line statement that announces the existence of a class without describing its members:

```
// node.h
#ifndef NODE_H
#define NODE_H

class Edge;           // forward declaration

class Node {
    Edge **edges_;    // pointers only, no Edge body needed
    int    n_edges_;
};
#endif

// edge.h
#ifndef EDGE_H
#define EDGE_H

class Node;           // forward declaration

class Edge {
    Node *src_;
    Node *dst_;
};
#endif
```

Neither header now #includes the other. Each tells the compiler "there exists a class with this name; I will not tell you anything about its members yet." That is enough information to declare members that are *pointers* to that class; the compiler knows the size of a pointer without knowing the size of what it points to.

The implementation files (node.cpp and edge.cpp) then include both complete headers because their member functions call methods on Node and Edge objects and need to know the full layout.

When Forward Declarations Work

A forward declaration provides an **incomplete type**. The compiler knows the name and that it is a class type. It does not know the size, the members, or the base classes. From that limited knowledge, you can do exactly four things:

1. Declare a **pointer** to it: `Edge *e;` or `Edge **e;`.
2. Declare a **reference** to it: `Edge &e` (typically as a function parameter).
3. Declare a function whose parameters or return type involve it: `Edge make_edge(Node *src, Node *dst);`.
4. Form a typedef or using alias: using `EdgePtr = Edge*;`.

You cannot do any of the following with only a forward declaration; these all require the full class definition:

- Declare a **value member**: `Edge e_;` requires knowing the size of Edge.
- Access **any member** through a pointer or reference: `e->weight()` requires knowing what members `Edge` has.
- Allocate one on the stack or heap: `Edge e;` or `new Edge` both need the constructor signature and size.
- Inherit from it: `class Cable : public Edge` requires the full base-class layout.
- Delete one through a pointer to it: `delete e_;` needs the destructor signature.

💡 **Tip:** The rule of thumb: forward declarations work in headers, where most uses are through pointers or references; they *rarely* work in .cpp files, which usually need to call methods or construct objects and therefore need the full type. Hence the common pattern: forward-declare in the header, include the full header in the .cpp.

❓ **Question:** Why does this even matter? Why not just always include the full header?

Two reasons. First, it breaks circular dependencies, as we just saw. Second, it dramatically reduces **compile-time coupling**: a header that forward-declares Edge rather than including edge.h does not need to be recompiled when edge.h changes. In large projects this is the difference between a 5-second incremental build and a 5-minute one.

The Build Process

So far we have treated "the compiler" as a black box that turns .cpp files into an executable. Inside that box is a pipeline of four programs that the driver g++ invokes in sequence:

The Four Stages

Stage	Program	Input	Output
Preprocessing	cpp	.cpp + .h files	preprocessed source
Compilation	cc1plus	preprocessed source	assembly code (.s)
Assembly	as	assembly code	object code (.o)
Linking	ld	object code + libraries	executable program

Each stage transforms its input into a lower-level form until the final stage produces a runnable program. The steps are:

- **Preprocessor (cpp).** Takes a C++ source file. Removes comments. Expands every #include directive by inserting the contents of the included file. Substitutes #defined macros. Evaluates #if/#ifdef conditionals. Outputs a single large source file with no preprocessor directives remaining.
- **Compiler (cc1plus).** Takes the preprocessed source. Parses it as C++. Performs type-checking, template instantiation, name resolution, and optimization. Outputs assembly language for the target architecture.
- **Assembler (as).** Takes assembly language. Translates it line by line into machine code. Outputs an **object file**, a binary file containing machine instructions for each function, along with a table of names this file *defines* (exports) and names it *uses* but does not define (unresolved references).
- **Linker (ld).** Takes one or more object files plus any libraries the program uses. Matches each unresolved reference in one object file against a definition in another. Stitches the machine code together, computes final addresses, and produces a single executable file.

The driver g++ runs all four programs in order. The command

```
g++ main.cpp adt1.cpp adt2.cpp -o program
```

preprocesses, compiles, and assembles each .cpp file into an object, then links the three objects into the executable program. If you omit -o program the executable is named a.out by default.

Translation Units

A useful term from the standard. A **translation unit** is one .cpp file *plus* everything its #include directives pull in, after the preprocessor has finished. In other words, a translation unit is the input to the compiler stage, after the preprocessor has done its work.

This is the unit at which most C++ rules are stated:

- The **one-definition rule** for variables and non-inline functions applies *across all translation units*, exactly one definition in the whole program.
- The **relaxed one-definition rule** for class types and inline functions allows the same definition in many translation units, as long as every copy is textually identical.
- A static name at file scope has **internal linkage**; it is visible only within *its translation unit*. Two different .cpp files can both define static int counter = 0; without conflict; each gets its own private counter.

It pays to keep the term in your head: many compiler error messages talk about "this translation unit" rather than "this file", and the distinction occasionally matters (a header that #includes another header is part of every translation unit that includes the outer header).

Separate Compilation

The four-stage pipeline gives us an enormous practical benefit: because linking is a separate stage from compilation, we can recompile only the files that changed and then relink.

```
g++ -c main.cpp        # produces main.o
g++ -c adt2.cpp        # produces adt2.o
g++ -c adt1.cpp        # produces adt1.o
g++ adt1.o adt2.o main.o -o exec    # link step
```

The -c flag tells g++ to stop after producing an object file. The final command takes the three object files and produces the executable exec. After editing only adt2.cpp, we recompile only it and re-link:

```
g++ -c adt2.cpp
g++ adt1.o adt2.o main.o -o exec
```

For a small program, the savings are modest, but for a project with hundreds of source files, compilation is the dominant cost of a build, and recompiling only what changed turns a fifteen-minute full build into a five-second incremental one.

> 💡 **Tip:** The price of separate compilation is that you, the developer, must keep track of which `.o` files depend on which `.cpp` and `.h` files, so you know what to rebuild when something changes. Doing that bookkeeping by hand is tedious and error-prone, exactly the kind of job we want to mechanize. That is what Make is for.

Compilation Dependencies

Before mechanizing, we need a clear model of what depends on what. When you change a file, what must be rebuilt?

The dependency graph for a typical program has three layers:

- The **executable** depends on all of the .o files.
- Each .o file depends on its corresponding .cpp file.
- Each .cpp file depends on every .h it includes (directly or transitively).

Drawing this for a small project clarifies what changes when:

Reading this:

- If adt2.cpp changes, only adt2.o needs to be rebuilt, then the executable re-linked. main.o and adt1.o are unaffected.
- If adt1.h changes, every .cpp that includes it (here: main.cpp, adt1.cpp, and adt2.cpp) must be recompiled. Their .o files are stale because the header's declarations may have changed. Then the executable is re-linked.
- If adt2.h changes, only files that include it (main.cpp and adt2.cpp) need recompilation. adt1.cpp does not include adt2.h, so adt1.o is unaffected.

⚠ **Warning:** This is the practical reason headers should be minimal: every *client* of a header pays a recompilation cost whenever the header changes. A header that exposes implementation details (e.g., declaring private member functions that no client calls, or `#include`ing other headers it does not need) ripples compile-time work across the project. This is the same recompilation problem we saw with the PImpl idiom in Chapter 3.

Automated Builds with Make

Goals of an Automated Build

We want a build system that:

- Reliably produces an executable that incorporates every change to every source file.
- Is **incremental**: rebuilds only files that need rebuilding.
- Tracks the full dependency graph so it knows which file is stale when any other file changes.

The classic tool that meets these goals is **Make**. A Makefile is a recipe book: each entry says "this target file depends on these prerequisite files; if any prerequisite is newer than the target, run this command to rebuild the target."

Anatomy of a Makefile

A minimal Makefile for the three-file program above:

```
# Makefile

exec: main.o adt1.o adt2.o
      g++ main.o adt1.o adt2.o -o exec

main.o: main.cpp adt1.h adt2.h
      g++ -c main.cpp

adt1.o: adt1.cpp adt1.h
      g++ -c adt1.cpp

adt2.o: adt2.cpp adt2.h adt1.h
      g++ -c adt2.cpp

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

Each entry is a **rule** with the form:

```
target: prerequisite1 prerequisite2 ...
<TAB>command to build target
```

⚠️ **Warning:** The character before the command *must* be a literal tab, not spaces. This is the single most common source of confused-looking errors for new Make users ("missing separator. Stop."). Configure your editor to insert tabs in Makefiles.

Running make exec (or just make, which builds the first target) causes Make to:

1. Look at the target exec and its prerequisites main.o adt1.o adt2.o.
2. Recursively build each prerequisite first, checking each .o's rule.
3. For each .o, compare its modification time with the modification times of its prerequisites. If any prerequisite is newer, run the command; otherwise, the target is up to date and is skipped.
4. Finally, if any .o was rebuilt, run the link command.

The clean target is a convention. It has no prerequisites, so the command runs whenever you type make clean, removing all build artifacts so the next make rebuild runs from scratch.

Variables and Pattern Rules

Repeating `g++ -c` in every rule is the sort of duplication we taught ourselves to avoid. Make supports variables (*macros*, in Make's terminology) and **pattern rules** that match families of files:

```
# Variables
CXX      = g++
CXXFLAGS = -std=c++17 -Wall -Wextra -g
```

```
OBJECTS  = main.o adt1.o adt2.o
EXEC     = program

# Explicit link rule
$(EXEC): $(OBJECTS)
      $(CXX) $(CXXFLAGS) $(OBJECTS) -o $(EXEC)

# Pattern rule: how to build any .o from its .cpp
%.o: %.cpp
      $(CXX) $(CXXFLAGS) -c $< -o $@

.PHONY: clean
clean:
      rm -f $(OBJECTS) $(EXEC)
```

Three things to notice:

- **$(VAR)** expands a variable. Variables can be defined on the command line as well (e.g. make CXXFLAGS=-O3) to override the Makefile's value.
- **%.o: %.cpp** is a pattern rule. The % is a wildcard that matches any stem; $< expands to the first prerequisite (the matching .cpp), and $@ expands to the target (the .o). This single rule replaces an entry for every .o in the project.
- **.PHONY: clean** declares that clean is not a real file. Without this, if a file named clean happened to exist in the directory, make clean would see the target was "up to date" and refuse to run the command.

💡 **Tip:** The most useful *automatic variables* inside a recipe are: $@ (the target), $< (the first prerequisite), and $^ (all prerequisites). Memorize these three, and most Makefiles become readable.

Auto-Generating Header Dependencies

Our Makefile so far lists each .o's header dependencies by hand: main.o: main.cpp adt1.h adt2.h. This is back to the manual bookkeeping we wanted to eliminate. Worse, if main.cpp gains a new #include and we forget to update the Makefile, Make will not rebuild main.o when that new header changes, and we will have stale object code linking into the executable. The bug surfaces as a runtime error that disappears after make clean, the worst kind.

The compiler can compute the dependency information for us. GCC's -MMD flag tells it to emit, alongside each object file, a .d file containing a Make-format dependency rule listing every header transitively included by that .cpp:

```
CXX      = g++
CXXFLAGS = -std=c++17 -Wall -Wextra -g -MMD -MP
OBJECTS  = main.o adt1.o adt2.o
DEPS     = $(OBJECTS:.o=.d)
EXEC     = program

$(EXEC): $(OBJECTS)
      $(CXX) $(CXXFLAGS) $(OBJECTS) -o $(EXEC)

%.o: %.cpp
      $(CXX) $(CXXFLAGS) -c $< -o $@

# Include the auto-generated dependency files (if they exist)
-include $(DEPS)

.PHONY: clean
clean:
      rm -f $(OBJECTS) $(DEPS) $(EXEC)
```

Three additions matter:

- **-MMD**: after compiling, also emit a .d file with this object's header dependencies.
- **-MP**: add a phony target for each header listed, so that deleting a header does not produce a confusing error.
- **-include $(DEPS)**: at the end of the Makefile, include every .d file we have. The leading - tells Make to ignore the error if a .d file does not yet exist (which is the case on the first build, before any .o has been compiled).

After the first build, Make has accurate header dependencies for every .o in the project, derived automatically by the compiler. Add a new #include to any .cpp, and the next build's .d file picks it up. This is the practical setup we will use for the rest of the course.

Sharing Variables Across Translation Units

Recall that a non-`const` variable defined at file scope has external linkage by default, and a `const` has internal linkage by default. What if you want exactly one shared variable visible to every `.cpp` in the project? There are three ways, with very different consequences. This is one of those topics where the C++17 answer is genuinely simpler than the historical answer, and worth knowing both.

Option A: **static in header** (caution, usually wrong)

```
// header.h
static int counter = 0;
```

Adding static at file scope forces **internal linkage**. Every .cpp that includes header.h gets its *own private copy* of counter. Mutations stay local; there are no linker errors; the program compiles cleanly. This is

fine for a file-local constant accidentally placed in a header, but it is *a mistake* if you wanted a single shared global. Two different .cpps will see two different values of counter, and the bug is invisible until you try to coordinate state between them.

Option B: **extern in header, definition in one .cpp** (works in every standard)

```
// header.h
extern int counter;

// header.cpp
int counter = 0;
```

The header declares the variable; exactly one .cpp defines it. Every other .cpp that #includes the header sees the declaration, and the linker resolves all uses to the one definition. This is the traditional way to share a global, and it is the only way that works before C++17. The cost is that you need both a header and a .cpp, and the declaration and definition must agree exactly.

Option C: **inline variable in header** (C++17 and later, preferred)

```
// header.h
inline int counter = 0;
```

As of C++17, `inline` applied to a variable means the same thing it has long meant for functions: "this definition may appear in multiple translation units; the linker, please merge them." Every .cpp that includes the header sees a definition, but all those definitions resolve to one shared variable. No separate .cpp is needed. The same mechanism also lets you define `static` class members directly in the header:

```
// widget.h
class Widget {
    static inline int instance_count = 0;    // C++17
    // ...
};
```

Without `inline`, `instance_count` would still need a separate definition in widget.cpp (the pre-C++17 burden). The summary table:

Approach	Linkage	How many copies	When to use
static in header	Internal	One per translation unit (PRIVATE)	File-local constants only, almost never what you want for a global

Approach	Linkage	How many copies	When to use
extern + .cpp	External	One shared (declared everywhere)	Pre-C++17 codebases, or when you want forced separation of decl and defn
inline in header (C++17+)	External	One shared (merged by linker)	Modern preferred form; header-only; works for static class members too

💡 **Tip:** For shared *constants*, `inline constexpr` combines the best of both worlds: header-only, no allocation, no runtime cost, available in every translation unit. `inline constexpr double PI = 3.14159;` is the modern replacement for the `extern const` idiom from earlier in this chapter.

Summary

Recognition

- The benefits of modular design: independent development, separate compilation, reuse, and incremental work.
- The benefits of separate compilation: rebuild only what changed, fast incremental builds.

Comprehension

- The principle of **separation of concerns**: each module owns one cohesive piece of functionality.
- The principle of **information hiding**: the header is the public interface; the `.cpp` hides everything else.
- The difference between a *declaration* (any number, all identical) and a *definition* (exactly one).
- The four stages of the build: preprocessor → compiler → assembler → linker.
- What a translation unit is and why the one-definition rule is stated in terms of it.

Application

- Drawing a program's module-dependency diagram and using it to predict what a change forces to recompile.
- Writing a correct C++ header file with header guard, minimal includes, and forward declarations where they suffice.
- Breaking circular dependencies with forward declarations.
- Writing a Makefile that uses variables, pattern rules, automatic variables, and `-MMD` to auto-derive header dependencies.
- Choosing between `static`, `extern`, and `inline` to share variables across translation units.

Practice Problems

1.

Below is a single-file C++ program. Split it into a properly-structured multi-file project.

```
// all_in_one.cpp
#include <iostream>
#include <string>

class Account {
    public:
        Account(std::string owner, double balance)
            : owner_{std::move(owner)}, balance_{balance} {}
        void deposit(double amt)  { balance_ += amt; }
        bool withdraw(double amt) {
            if (amt > balance_) return false;
            balance_ -= amt;
            return true;
        }
        double balance() const { return balance_; }
        const std::string& owner() const { return owner_; }
    private:
        std::string owner_;
        double      balance_;
};

void print_account(const Account& a) {
    std::cout << a.owner() << ": $" << a.balance() << '\n';
}

int main() {
    Account a{"Alice", 100.00};
    a.deposit(50);
    print_account(a);
}
```

Produce four files: account.h, account.cpp, main.cpp, and a Makefile that builds the executable bank with auto-generated header dependencies. Decide where print_account belongs (a free function in account.h? a member function? a function in main.cpp?) and justify your choice in one sentence.

2.

The following header compiles and the program links, but it is wrong in three different ways, two of which will eventually cause real problems. Find all three and explain how to fix each.

```
// shape.h

const double PI = 3.14159265358979;

#include <vector>

class Shape {
    public:
        Shape();
        virtual double area() const;
        virtual ~Shape();
    private:
        std::vector<double> sides_;
        int    cached_centroid_x_;
        int    cached_centroid_y_;
        bool   cache_valid_;
        void   recompute_cache_();
};
```

3.

Consider a project with the following dependency structure (arrows mean *file on left includes file on right*, and every .cpp includes its own .h):

```
main.cpp –> engine.h –> entity.h
                \
                 –> texture.h
engine.cpp –> engine.h
          –> renderer.h –> texture.h
entity.cpp –> entity.h
renderer.cpp –> renderer.h –> texture.h
texture.cpp –> texture.h
```

Answer each of the following:

(a) After editing texture.h, which .o files must be rebuilt?

(b) After editing engine.cpp, which .o files must be rebuilt?

(c) renderer.h only uses Texture through pointers in its class members. Propose a change that reduces the recompilation cost when texture.h changes, and redraw the affected part of the dependency graph after your change.

(d) Could you do the same trick for entity.h, given that engine.h stores entities by value? Explain why or why not.

4.

Design two classes Customer and Order that have a circular relationship: a Customer owns a list of pointers to its Orders, and an Order has a pointer back to its Customer. The constraint: every member function of Customer must be defined in customer.cpp, and every member function of Order in order.cpp.

Produce customer.h, order.h, customer.cpp, and order.cpp. Use forward declarations to make the headers free of cycles. Include at least one member function on each class that actually calls a method on the other (e.g. Order::customer_name() calls Customer::name()). Verify that the .cpp files do *not* need forward declarations, they include the full headers.

5.

You are auditing a small game engine. The maintainer added the following at the top of `game.h`:

```
// game.h
#ifndef GAME_H
#define GAME_H

int  score = 0;
bool game_over = false;

// ... class Game definition follows ...

#endif
```

Three .cpp files include game.h: game.cpp, main.cpp, and ui.cpp. The program **fails to link** with a "multiple definition of `score`" error.

(a) Explain precisely what went wrong, in terms of the one-definition rule and translation units.

(b) Give *three* different fixes that make the program link: one using extern, one using inline (C++17), and one using static. For each fix, describe the resulting semantics, does the variable end up shared across files, or private to each, and why?

(c) Which fix would you actually choose for a real game engine, and what is the reason?

6.

Write a Makefile for a project with the following files and dependencies:

- main.cpp includes matrix.h and vector3.h.
- matrix.cpp includes matrix.h and vector3.h.
- vector3.cpp includes vector3.h.
- matrix.h includes vector3.h.

Requirements:

1. Variables for CXX, CXXFLAGS, OBJECTS, and EXEC.
2. Use a pattern rule rather than one rule per .o.
3. Use -MMD and -include so that header dependencies are derived automatically.
4. A clean target declared .PHONY that removes object files, dependency files, and the executable.
5. A default target all that builds the executable.

Then trace what make does on each of the following scenarios, starting from a clean state:

(a) First make, list every command Make runs and in what order.

(b) Edit vector3.cpp, then make, which commands run?

(c) Edit vector3.h, then make, which commands run, and why is the answer different from (b)?

7.

⭐ **Stretch, interface design under recompilation pressure.** You maintain a widely-used Renderer class in a large codebase. The renderer.h header is included, transitively, by approximately 200 .cpp files. A full rebuild after touching renderer.h takes 90 seconds.

Currently, the header looks like:

```
// renderer.h
#ifndef RENDERER_H
#define RENDERER_H

#include "texture.h"      // for Texture
#include "shader.h"       // for Shader
#include "mesh.h"         // for Mesh
#include "camera.h"       // for Camera
#include <vector>
#include <string>

class Renderer {
    public:
        void draw(const Mesh& m, const Texture& t, const Shader& s, const
Camera& c);
        void set_clear_color(float r, float g, float b);
        void register_texture(const std::string& name, Texture t);
        Texture get_texture(const std::string& name) const;
    private:
        std::vector<Texture>  texture_cache_;
        std::vector<Shader*>  active_shaders_;
        Camera*               current_camera_;
        // ... 30 more private members ...
};

#endif
```

(a) Identify every include in renderer.h that could be replaced by a forward declaration, and explain for each include why a forward declaration is or is not sufficient given how the corresponding type is used.

(b) The private section is forcing 200 client files to recompile every time any of those 30 private members changes. Propose and write the header for a fix using the **PImpl idiom** (Chapter 3) so that changes to the implementation no longer trigger a project-wide recompile. Show both the new renderer.h and a sketch of renderer.cpp.

(c) Estimate, qualitatively, how much faster recompilation becomes after each of (a) and (b). Where does the cost come from in the original setup, and which fix attacks which cost?

8.

⭐ **Stretch, Debug a build failure.** A student has the following files. The program fails to link with an error of the form

```
undefined reference to `Logger::instance()'
undefined reference to `Logger::log(std::string const&)'
```

```
// logger.h
#ifndef LOGGER_H
#define LOGGER_H
#include <string>

class Logger {
    public:
        static Logger& instance();
        void log(const std::string& msg);
    private:
        Logger() = default;
};
#endif

// main.cpp
#include "logger.h"
int main() {
    Logger::instance().log("hello");
}

// Compile command:
// g++ -std=c++17 main.cpp -o app
```

(a) Explain in build-pipeline terms (*which stage* failed, *why*) what is missing.

(b) Show the contents of the `logger.cpp` the student needs to add, and the new compile/link command (or Makefile) that produces a working executable.

(c) Could *any* of the same problem be solved by changing the header alone (no new `.cpp`) using a C++17 feature from this chapter? Show how, and explain when this approach is and is not appropriate.