# Multi-File Programs in C++

Instructor: Arbish Akram

## 1 Why Bother Splitting a Program into Files?

Every program you have written so far in the programming fundamental course has probably lived in a single file: one `.cpp` file containing everything from `#include` statements down to the closing brace of `main()`. For small programs, that is completely fine. But think about what happens as a program grows, say, once it contains several classes, each with a handful of methods, plus a `main()` that ties everything together. A few things start to hurt:

- **The file becomes hard to navigate.** Scrolling through 2,000 lines to find one function is slow and error-prone.
- **Compilation becomes slow.** If everything lives in one file, changing a single line forces the compiler to reprocess the *entire* file from scratch, every time.
- **Collaboration becomes painful.** If you and a partner are both editing the same file, you are constantly stepping on each other's changes.
- **Reuse becomes impossible.** If you write a handy `Rectangle` class for one assignment, you cannot easily "borrow" it for the next one without dragging along everything else in that file.

The fix is to split a program into multiple files that each hold one coherent piece of it, and to give the compiler a systematic way of stitching those pieces back together. That systematic way is the subject of this reader. It is not a stylistic nicety – it is the standard structure of essentially every real C++ program you will ever encounter, and it is the structure you will use for every class you write for the rest of this course.

**KEY IDEA**

Splitting a program into files is about *separating concerns*: one file says what exists, another file says how it works, and a third file says how it is used. Once you see this three-way split, you will recognize it everywhere, in any language.

## 2 Declarations versus Definitions

Before we can talk about files, we need to be crisp about a distinction that is easy to gloss over: the difference between *declaring* something and *defining* it.

A **declaration** tells the compiler that something, a function, a variable, a class, exists, and describes its type or signature, but does not say how it works or reserve any storage for it.

```
double add(double a, double b);  // a declaration ("prototype")
```

A **definition** is the full story: it provides the actual function body, or, for a variable, it actually allocates storage.

```
double add(double a, double b) {  // a definition
    return a + b;
}
```

Every definition is also a declaration (it tells the compiler the function exists), but not every declaration is a definition. This distinction sounds pedantic right now, but it is the entire reason the file-splitting scheme in the rest of this reader works the way it does.

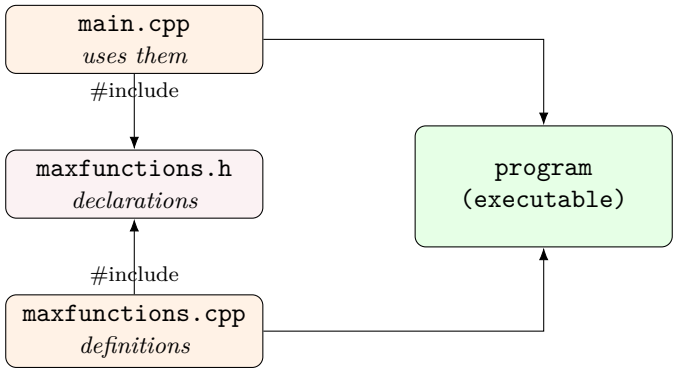
# 3 The Two Kinds of Files: Headers and Sources

C++ projects are organized around two kinds of files, distinguished by convention (not by any hard rule the compiler enforces) using their file extensions:

- **Header files** (`.h` hold *declarations*: function prototypes, class declarations, constants, and type definitions. A header answers the question *"what exists, and what does it look like from the outside?"*
- **Source files** (`.cpp`) hold *definitions*: the actual bodies of functions and methods. A source file answers the question *"how does this actually work?"*

A third kind of file, also a `.cpp` file, but playing a different role, is the file containing `main()`. It typically does not define reusable functions of its own; it just `#include`s the headers it needs and *uses* what they declare.

So a typical small project looks like this:




Both `main.cpp` and `maxfunctions.cpp` `#include` the header, because both of them need to know that `max` exists and what its signature is. Only `maxfunctions.cpp` contains the actual definition of `max`.

**KEY IDEA**

**Rule of thumb, used throughout this course:** declarations go in `.h` files; definitions go in `.cpp` files. If you remember nothing else from this reader, remember this sentence.

## 3.1 What a Header File Contains

A header file is pulled into a source file with the preprocessor directive `#include`. A header typically contains:

- Function **prototypes** (declarations).
- Class declarations: the list of data members and member function prototypes.
- Definitions of true constants (e.g. `const double PI = 3.14159;`).
- `#include` statements for any other headers it depends on, including standard library headers such as `<iostream>` or `<cmath>`.

What normally should *not* go in a header: full function bodies (with a small exception for `inline` functions and templates), and definitely not `using namespace std;`, doing that in a header silently forces that decision on every file that includes it, which is exactly the kind of hidden coupling multi-file design is trying to avoid.

```
#include "maxfunctions.h"    // your own header: quotes
#include <iostream>          // standard-library header: angle brackets
```

Quotes tell the compiler to look for the file in the current project directory first (falling back to system paths); angle brackets tell it to search the compiler's standard system include paths directly. If your own files are spread across subdirectories, you can add extra search locations with the `-I` compiler flag.

## 3.2 Header Guards

Whenever you write `#include "file.h"`, the preprocessor does something almost embarrassingly literal: it copies the *entire contents* of `file.h` and pastes them in at that exact spot, before real compilation even begins. This creates an obvious hazard: what if the same header gets included more than once into the

same file, directly or indirectly? Its contents would be pasted in twice, and the compiler would see the same declarations twice, often triggering a "redefinition" error.

The fix is a **header guard**: a small piece of preprocessor logic that makes a header's contents take effect only the first time they are included.

```
// mathfunctions.h
#ifndef MATHFUNCTIONS_H    // "if MATHFUNCTIONS_H has not been defined yet..."
#define MATHFUNCTIONS_H    // "...define it now, so next time we skip this."

int add(int a, int b);     // the actual declaration(s)

#endif // MATHFUNCTIONS_H
```

The first time this header is included anywhere, `MATHFUNCTIONS_H` is not yet defined, so the preprocessor defines it and lets the declaration through. If the same header is included again later in the same file (perhaps indirectly, through another header), `MATHFUNCTIONS_H` is already defined, so everything between `#ifndef` and `#endif` is skipped.

Most modern compilers, g++ included, also support a one-line shorthand that does the same job:

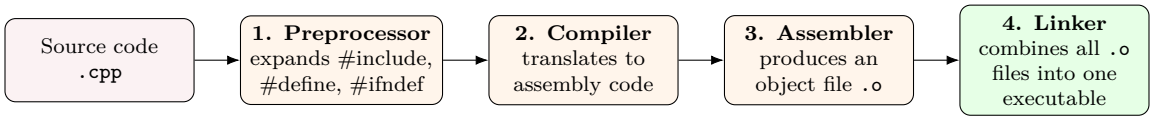
```
#pragma once

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

`#pragma once` is not part of the official C++ standard, but it is supported almost everywhere and is common in industry code. Either style is acceptable in this course – just be consistent within a project, and *never* write a header without some form of guard.

## 4 How a Multi-File Program Actually Gets Built

For *each* `.cpp` file, the compiler pipeline runs through four stages; a final stage then combines the results.




1. **Preprocessing.** Every line starting with `#` is handled here, before the compiler even sees "real" C++. `#include` lines are replaced with the actual contents of the named file; `#define` macros are textually substituted; `#ifndef`/`#endif` guards decide what survives. The output is one large, fully expanded chunk of pure C++ source – no more `#` lines remain.
2. **Compilation.** The compiler translates that expanded C++ code into assembly instructions for your machine's processor, checking syntax and types as it goes. This is where most of the errors you are used to seeing (missing semicolons, type mismatches, undeclared identifiers) get caught.
3. **Assembly.** The assembler converts the assembly code into actual machine code, stored in an **object file** (`.o` on Linux/macOS, `.obj` on Windows). Crucially, this happens *separately for each `.cpp` file* – `main.cpp` and `maxfunctions.cpp` each become their own, independent object file, with no knowledge of each other yet.
4. **Linking.** The **linker** takes all the object files (plus any needed pre-compiled libraries) and resolves every function call to the actual machine code that implements it, producing one executable. This is also where a distinct family of errors appears – "undefined reference" and "multiple definition" – that only show up *after* every individual file has already compiled successfully on its own.

You can watch these stages happen by hand:

```
g++ -c maxfunctions.cpp -o maxfunctions.o   # stop after step 3: produces a .o file
g++ -c main.cpp -o main.o                   # compile main.cpp separately
g++ maxfunctions.o main.o -o program        # step 4: link the two object files
    together
./program
```

The single familiar command `g++ main.cpp maxfunctions.cpp -o program` simply automates all of this for you in one shot.

**KEY IDEA**

Because each `.cpp` file is compiled into its own object file *independently*, changing one `.cpp` file only requires recompiling *that* file, not the whole project – the other object files can be reused as-is and just re-linked. This is called **separate compilation**, and it is the real payoff of splitting a program into files: faster rebuilds as your project grows.

## 5 The One Definition Rule

C++ enforces a rule with a name worth knowing: the **One Definition Rule** (ODR). Informally: any given function, variable, or class may be *defined* exactly once across an entire program, although it may be *declared* many times.

This single rule quietly explains three things you have already seen in this reader:

- Why function *definitions* belong in `.cpp` files rather than headers – a header can be `#include`d into several different `.cpp` files, and if it contained a full definition, that definition would be duplicated into each one, violating ODR.
- Why header guards exist – to stop a header's contents from being pasted twice into the *same* file.
- Why a stray full definition placed in a header tends to compile fine on its own, but then fails at the link stage with a "multiple definition" error once two different `.cpp` files both include it.

If you genuinely need a small piece of code defined in a header (a tiny helper, a constant, a template), C++ gives you specific keywords – `const`, `constexpr`, `inline` – that are explicitly exempted from causing ODR violations. Absent one of those keywords, keep definitions out of headers.

## 6 Controlling Visibility: `extern` and `static`

Two keywords come up as soon as you need to share (or deliberately hide) data across files.

**`extern`** declares that a global variable is *defined somewhere else*, in another file. You put the `extern` declaration in a header so multiple `.cpp` files can all refer to one shared variable, while only one `.cpp` file actually defines (and allocates storage for) it.

```
// globals.h
extern int callCount;   // declaration only -- no storage allocated here

// globals.cpp
int callCount = 0;      // the one and only definition
```

**`static`**, when applied to a function or global variable at file scope, does the opposite: it restricts that name so it is visible *only inside the file where it is defined*. This is useful for small helper functions that a `.cpp` file needs internally but has no business exposing to the rest of the program.

## 7 Common Errors When Working Across Files

Once your project spans more than one file, a handful of errors show up again and again. Recognizing them by shape will save you a great deal of debugging time.

| Error | Typical Cause | Fix |
|---|---|---|
| `undefined reference to `foo()'` | A function was declared (and called) but its `.cpp` definition was never compiled in or linked. | Add the missing `.cpp` file to the build command; double-check for typos in the function name or signature. |
| `multiple definition of `foo()'` | A full function *definition* was placed in a header, and that header got included into more than one `.cpp` file. | Move the definition into a `.cpp` file; leave only the declaration in the header. |
| `redefinition of `X'` | A header without a guard got included more than once into the same file, directly or indirectly. | Add `#ifndef`/`#define`/`#endif` or `#pragma once`. |
| `fatal error: foo.h: No such file or directory` | The header is not in the compiler's search path. | Check the file name and location, or add `-I<directory>` to the compile command. |

Notice that the first two are *linker* errors – they appear only after every file has individually compiled without complaint – while the last two are caught earlier, by the preprocessor or compiler.

## 8 Worked Example: Finding the Maximum of Two Numbers

Let's put all of this together on a genuinely small example, so the mechanics are completely visible.

### Step 1 – Header File: `maxfunctions.h`

```
#ifndef MAXFUNCTIONS_H
#define MAXFUNCTIONS_H

int max(int a, int b);   // declaration only -- no body here

#endif
```

### Step 2 – Source File: `maxfunctions.cpp`

```
#include "maxfunctions.h"

int max(int a, int b) {
    return (a > b) ? a : b;
}
```

### Step 3 – Main Program: `main.cpp`

```
#include <iostream>
#include "maxfunctions.h"
using namespace std;

int main() {
    int num1 = 10, num2 = 15;
    int maximum = max(num1, num2);
    cout << "The maximum is: " << maximum << endl;
    return 0;
}
```

Notice the three roles at work: the header *announces* that `max` exists, the source file *implements* it, and `main.cpp` simply *uses* it without needing to know how it works internally. To build and run it:

```
g++ main.cpp maxfunctions.cpp -o program
./program
```

## 9 Connecting This to Object-Oriented Programming

Everything above was framed around plain functions, but the pattern generalizes directly – and this is the part that matters most for the rest of this course. A **class** splits across files exactly the same way a function does: the class *declaration* (its data members and member function prototypes) goes in the header, and every member function *definition* goes in the matching `.cpp` file, each one tagged with the class name using the **scope resolution operator** `::` to say "this definition belongs to that class."

### Step 1 – Header File: `Rectangle.h`

```
#ifndef RECTANGLE_H
#define RECTANGLE_H

class Rectangle {
private:
    double width;
    double height;

public:
    Rectangle(double w, double h);   // constructor: declared only
    double area() const;             // member functions: declared only
    double perimeter() const;
};

#endif
```

### Step 2 – Source File: `Rectangle.cpp`

```
#include "Rectangle.h"

Rectangle::Rectangle(double w, double h) : width(w), height(h) {}

double Rectangle::area() const {
    return width * height;
}

double Rectangle::perimeter() const {
    return 2 * (width + height);
}
```

### Step 3 – Main Program: `main.cpp`

```
#include <iostream>
#include "Rectangle.h"
using namespace std;

int main() {
    Rectangle r(4.0, 5.0);
    cout << "Area: " << r.area() << endl;
    cout << "Perimeter: " << r.perimeter() << endl;
    return 0;
}
```

Build it the same way as before: `g++ main.cpp Rectangle.cpp -o rectDemo && ./rectDemo`

**KEY IDEA**

A class header is often called its **interface**: it tells any other file everything it needs to know to *use* the class (what data it logically has, what operations are available, what to pass in and what comes back) without revealing *how* those operations are implemented. The `.cpp` file is the **implementation**: it can change completely – a different algorithm, different private helper logic – without anyone who only reads the header ever noticing, as long as the public interface stays the same. This separation of interface from implementation is one of the central ideas of object-oriented design, and multi-file structure is what makes it concrete in C++.

## 10 Automating the Build: A First Look at Makefiles

As a project grows past two or three files, typing the full `g++` command by hand – and re-typing it after every change – becomes tedious, and it throws away the recompilation savings that separate compilation is supposed to give you. A **Makefile** solves this: it records how each file depends on the others, and a tool called `make` uses that information to rebuild only what actually changed.

Every rule in a Makefile has the same three-part shape:

```
target: prerequisite1 prerequisite2 ...
    recipe (the shell command that builds target)
```

Here is a Makefile for the `Rectangle` project from earlier in this reader, which compiles `main.cpp` and `Rectangle.cpp` into a single executable called `rectDemo`:

```
CXX = g++
CXXFLAGS = -Wall -std=c++17

rectDemo: main.o Rectangle.o
    $(CXX) main.o Rectangle.o -o rectDemo

main.o: main.cpp Rectangle.h
    $(CXX) $(CXXFLAGS) -c main.cpp

Rectangle.o: Rectangle.cpp Rectangle.h
    $(CXX) $(CXXFLAGS) -c Rectangle.cpp

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

It helps to read this Makefile from the *bottom rule up*, the way `make` itself resolves it when you type `make rectDemo`:

- `CXX = g++` and `CXXFLAGS = -Wall -std=c++17` are **variables**. They are not required – you could write `g++` directly on every recipe line – but defining them once means that switching compilers, or adding a new warning flag, is a single-line change instead of an edit to every rule. `$(CXX)` and `$(CXXFLAGS)` later in the file simply expand back to these values.
- `Rectangle.o: Rectangle.cpp Rectangle.h` says: *"Rectangle.o depends on Rectangle.cpp and Rectangle.h."* If either source file changes, this rule fires. The recipe uses `-c`, which tells `g++` to compile to an object file *without* linking – the same flag from the separate-compilation section earlier in this reader.
- `main.o: main.cpp Rectangle.h` is the equivalent rule for `main.cpp`. Notice it also depends on `Rectangle.h`, not just `main.cpp` – if the header's class declaration changes, `main.cpp` needs recompiling too, since it includes that header.
- `rectDemo: main.o Rectangle.o` is the **final linking step**. It depends on both object files being present and current; its recipe has no `-c`, so this invocation of `g++` links the two `.o` files into the executable.
- `clean:` is a rule with **no prerequisites**. It does not build anything – it exists purely so you can type `make clean` to delete generated files and force a full rebuild next time. Rules like this, which name an action rather than a real file, are conventionally called *phony* targets.

```
$ make            # builds the first target in the file: rectDemo
$ make rectDemo   # same thing, named explicitly
$ make clean      # removes *.o and rectDemo
```

Try it yourself: run `make`, then edit only `Rectangle.cpp` and run `make` again. You should see `g++` recompile `Rectangle.o` and relink `rectDemo`, but `main.o` is left untouched because none of its prerequisites changed. This is separate compilation happening automatically, driven purely by file timestamps.

You won't hand-write these forever. Real-world C++ projects rarely maintain Makefiles by hand once they grow large – tools like `CMake` generate the Makefile (or an equivalent build script) for you from a shorter, higher-level description of the project, and most IDEs generate and run one behind the scenes every time you click "Build." Learning to read a small, hand-written Makefile like the one above is what makes those generated files – and the build failures they sometimes produce – decipherable instead of mysterious.

## 11 Summary

- A **declaration** says something exists; a **definition** says how it works. Headers hold declarations; source files hold definitions.
- A build has four stages – **preprocessing**, **compilation**, **assembly**, and **linking** – and each `.cpp` file moves through the first three independently before the linker combines everything.
- **Header guards** (`#ifndef`/`#define`/`#endif` or `#pragma once`) stop a header's contents from being pasted into the same file twice.
- The **One Definition Rule** is the underlying reason definitions stay out of headers: it may only be defined once across the whole program.
- `extern` shares one global variable across files; `static` (at file scope) hides a name from every other file.
- **Undefined reference** and **multiple definition** are *linker*-stage errors; **redefinition** and missing-file errors happen earlier, in preprocessing/compilation.
- A **class** splits across files exactly like a function does: the header is its public *interface*, and the `.cpp` file is its *implementation* – the foundation you will build on for every class in this course.