COP3503

Building C++ Programs

Welcome!

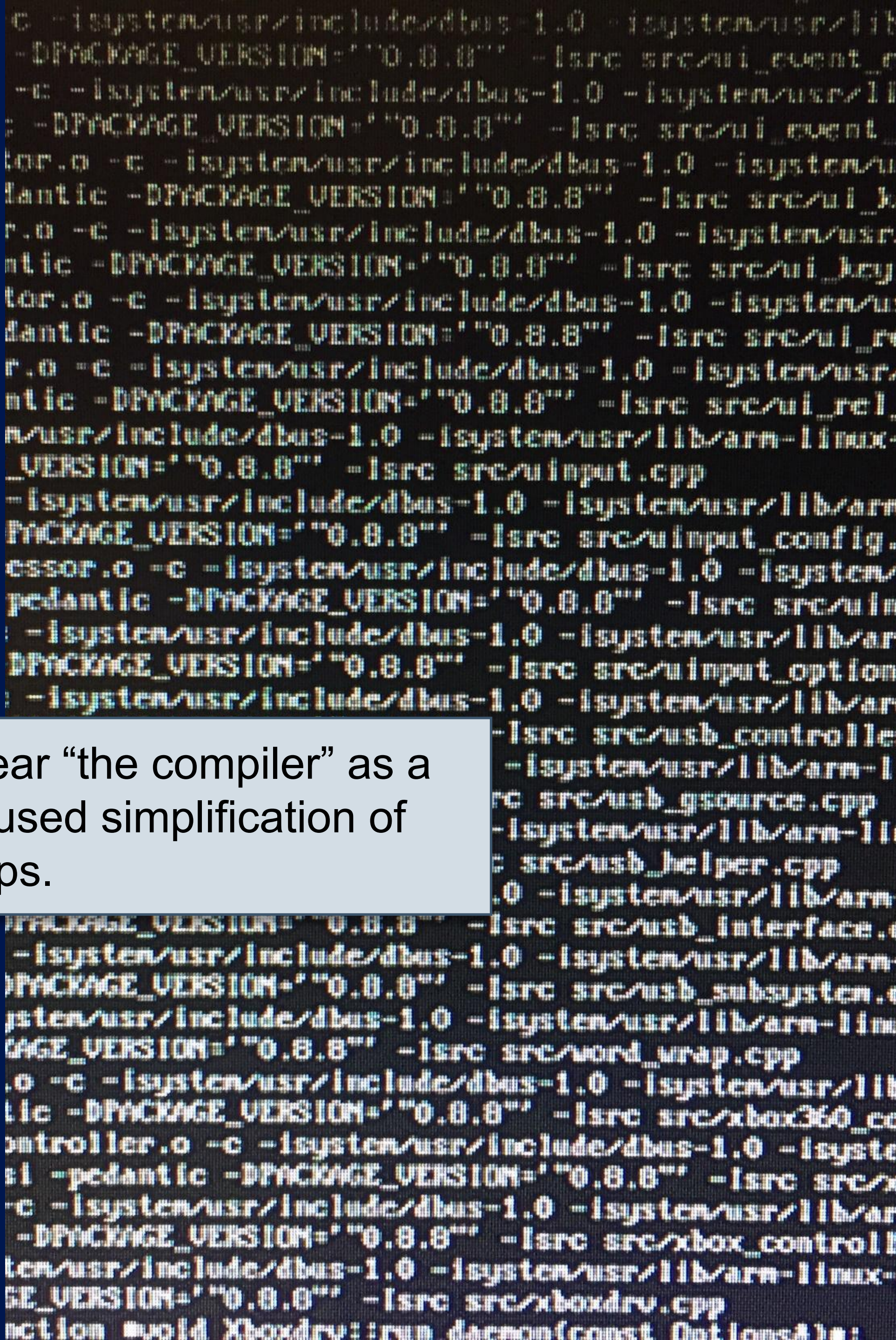
Placeholder for the instructor's welcome message. Video team, please insert the instructor's video here.

From Code to Program

When you “build a C++ program” a 3-step sequence happens:

- **Preprocessing**
  Create “units” of code for compiling
- **Compiling**
  Ensure the units are valid C++ code
- **Linking**
  Combine valid C++ code into an executable

You may hear “the compiler” as a commonly used simplification of these 3 steps.



```
c -isystem/usr/include/dbus-1.0 -isystem/usr/lib
-DPACKAGE_VERSION='"0.8.8"' -Isrc src/ui_event_c
-c -isystem/usr/include/dbus-1.0 -isystem/usr/li
c -DPACKAGE_VERSION='"0.8.8"' -Isrc src/ui_event_
tor.o -c -isystem/usr/include/dbus-1.0 -isystem/u
dantic -DPACKAGE_VERSION='"0.8.8"' -Isrc src/ui_k
r.o -c -isystem/usr/include/dbus-1.0 -isystem/usr
ntic -DPACKAGE_VERSION='"0.8.8"' -Isrc src/ui_key
tor.o -c -isystem/usr/include/dbus-1.0 -isystem/u
dantic -DPACKAGE_VERSION='"0.8.8"' -Isrc src/ui_r
r.o -c -isystem/usr/include/dbus-1.0 -isystem/usr/
ntic -DPACKAGE_VERSION='"0.8.8"' -Isrc src/ui_rel_
m/usr/include/dbus-1.0 -isystem/usr/lib/arm-linux-
_VERSION='"0.8.8"' -Isrc src/uinput.cpp
-isystem/usr/include/dbus-1.0 -isystem/usr/lib/arm
PACKAGE_VERSION='"0.8.8"' -Isrc src/uinput_config.
essor.o -c -isystem/usr/include/dbus-1.0 -isystem/
pedantic -DPACKAGE_VERSION='"0.8.8"' -Isrc src/uin
: -isystem/usr/include/dbus-1.0 -isystem/usr/lib/ar
DPACKAGE_VERSION='"0.8.8"' -Isrc src/uinput_option
: -isystem/usr/include/dbus-1.0 -isystem/usr/lib/ar
                            -Isrc src/usb_controlle
                             -isystem/usr/lib/arm-l
                           rc src/usb_gsource.cpp
                            -isystem/usr/lib/arm-li
                            c src/usb_helper.cpp
                           .0 -isystem/usr/lib/arm
PACKAGE_VERSION='"0.8.8"' -Isrc src/usb_interface.
-isystem/usr/include/dbus-1.0 -isystem/usr/lib/arm-
DPACKAGE_VERSION='"0.8.8"' -Isrc src/usb_subsystem.c
stem/usr/include/dbus-1.0 -isystem/usr/lib/arm-lin
KAGE_VERSION='"0.8.8"' -Isrc src/word_wrap.cpp
.o -c -isystem/usr/include/dbus-1.0 -isystem/usr/li
tic -DPACKAGE_VERSION='"0.8.8"' -Isrc src/xbox360_co
ontroller.o -c -isystem/usr/include/dbus-1.0 -isyste
al -pedantic -DPACKAGE_VERSION='"0.8.8"' -Isrc src/x
-c -isystem/usr/include/dbus-1.0 -isystem/usr/lib/ar
-DPACKAGE_VERSION='"0.8.8"' -Isrc src/xbox_controll
tem/usr/include/dbus-1.0 -isystem/usr/lib/arm-linux-
GE_VERSION='"0.8.8"' -Isrc src/xboxdrv.cpp
nction void Xboxdrv::run_daemon(const Options&)':
```

Preprocessor

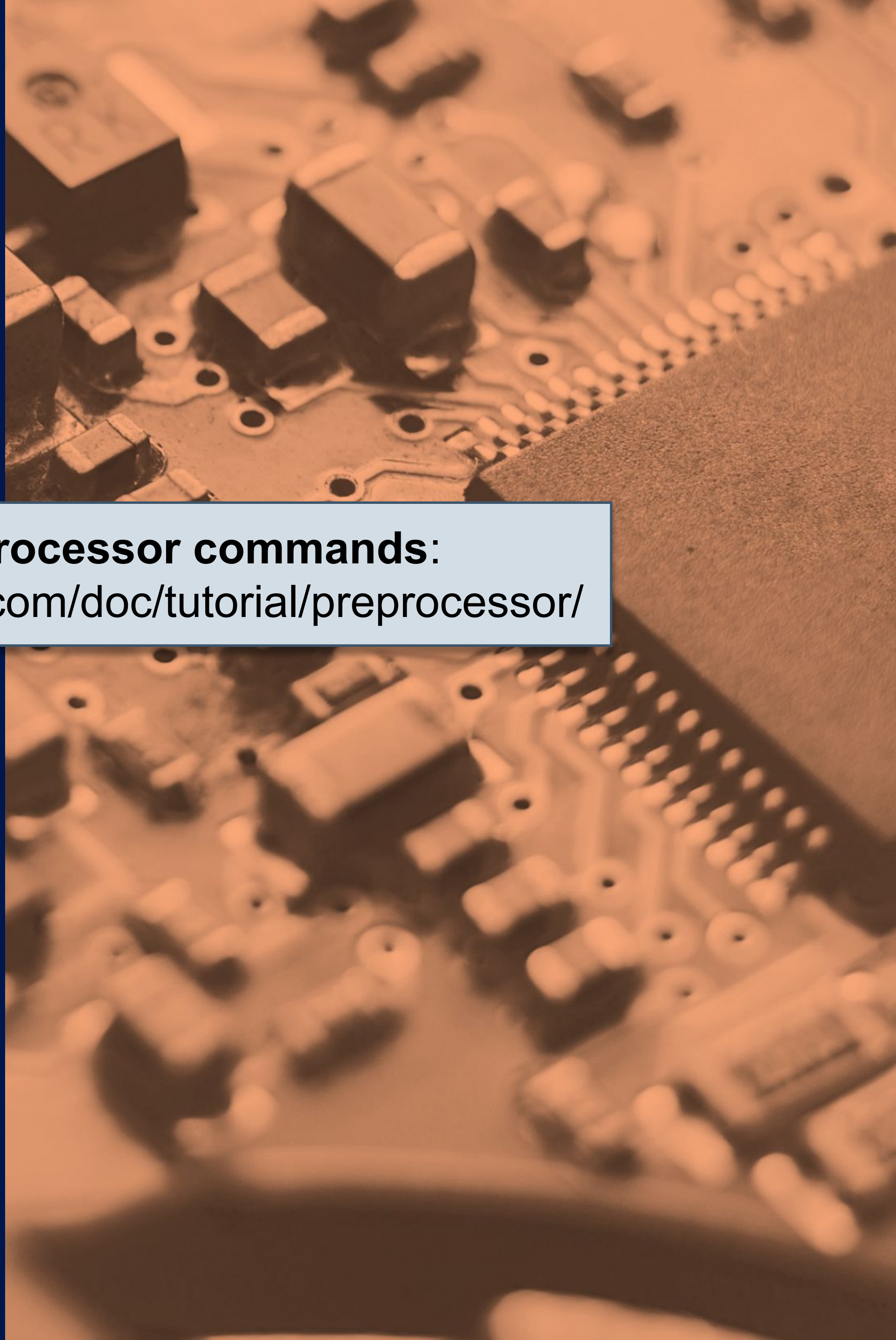
- Searches project files from top to bottom for **preprocessor directives**

- Keywords beginning with #
  - `#include`, `#pragma`, `#define` and more

More on preprocessor commands:
http://www.cplusplus.com/doc/tutorial/preprocessor/

- Performs the necessary steps for each of these

- Creates **translation units** from the results



Translation Unit

From the C++ Standard

- The C++ Standard is an enormous document (2000+ pages!) that describes the language in explicit detail.
- A paraphrased excerpt about translation units:

> **A source file together with all the headers and source files included via the preprocessing directive #include**, less any source lines skipped by any of the conditional inclusion preprocessing directives, is called a **translation unit**.

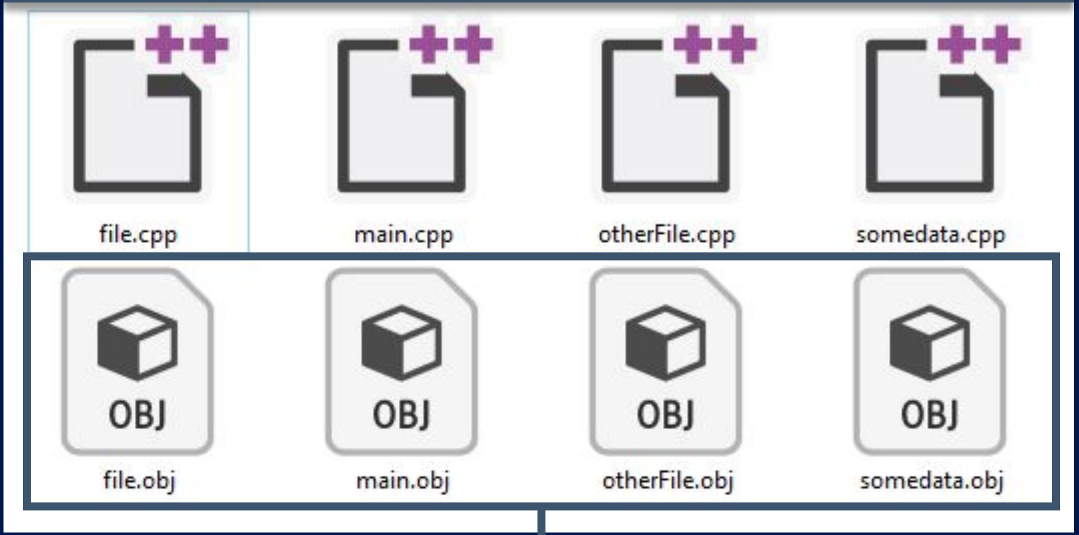
Simplified: A source file (a .cpp file) becomes a translation unit (some may call it a compilation unit).

Translation units get sent to the next step: compiling!

Compiling

- Checks all of your translation units for proper syntax
- Creates **object files** from your translation units
- Object files contain **machine code** that our computer will understand.
- We write “high level” instructions, and the compiler converts that to machine instructions.

Object files created from a compiler



You don’t use these .obj

Compiler Errors

- The compiler says “Hey, this is not correct, syntactically. I don’t understand what you want.”
- If there are any compiler errors in a translation unit, no object file is created.
- If there are any errors in this stage, the linking stage won’t start, and the build process ends.

Compiler error examples:

```
int x = 5;
x = x + "dinosaur"; // Error: Cannot convert from ‘const char *’ to ‘int’
```

```
void SomeFunction(int x, int y) { }
SomeFunction(10); // Error: function does not take 1 argument
```

Linking

- The final stage, combines **object files** into an **executable** (or a library, more on that later!)

- Errors at this stage are not always easy to interpret (especially for new programmers)

The same error, reported in two different ways…

Programming languages don’t always report errors the same way.

Even different C++ compilers might “speak” differently!

```
unresolved external symbol "public: __cdecl Foo::~Foo(void)" (??1Foo@@QEAA@XZ)
referenced in function main
```

```
undefined reference to Foo::~Foo()
```

Compiler Errors vs. Linker Errors

Compiler error
Something is wrong syntactically

- Missing semi-colon
- Typo with a function
- Function expects a string, you passed an integer
- Etc… This is invalid C++ code.

Linker error
Some definition is missing

- Some function definition doesn’t exist or can’t be found.
- Some library (and its functions) doesn’t exist or can’t be found.

How Does This Affect You?

(i.e., Why am I talking about it?)

- The way you write C++ code revolves around these steps.
- ...and there are also “rules” and suggested guidelines.
  - The language enforces very little.
  - C++ is not a language that holds your hand.
- There are rules we must follow...
- Syntax vs. Style
  - Bad syntax? Your program won’t build!
  - Bad style? Your code is just messy.

Coworkers might dislike working with you and your code!

Definition and Declaration

Building code properly revolves around these.

Declaration

- Creates an **identifier** that can be referenced elsewhere in our code
- Identifiers are the **variables**, **functions**, and **user-defined types (classes)** we need to make our program work.
- Referencing an identifier that doesn’t exist is a compiler error.

```
int main()
{
    score = 5; // Error, undeclared identifier "score"
    Foo(10);   // Error, undeclared identifier "Foo"
    return 0;
}
```

Declaration Examples

```
int Raise(int value, int exponent); // Declare the function

int main()
{
    int score;        // Declare the identifier "score"
    score = 5;     // Assign a value to the variable
    int numPlayers = 3;   // Declare AND initialize a variable

    int result = Raise(10, 2);   // Call the function
    Foo();     // Error, undeclared identifier

    return 0;
}

void Foo(); // Declare the Foo function
```

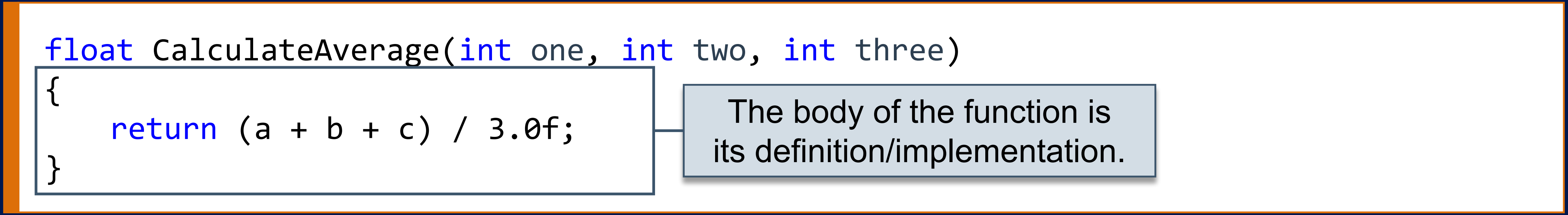
C++ compiles from **top** to **bottom**; identifiers must be declared before (above) lines that use them.

Prototypes

- Function declarations are called **prototypes**.
  - You may also hear **forward declaration** used in some places.
- They create an identifier for the compiler.
- They serve as a description of how to **call** the function, not what the function actually does.

```
float CalculateAverage(int one, int two, int three);
int GetNumberFromUser();
int RandomInt(int min, int max);
```

Prototypes



```
float CalculateAverage(int one, int two, int three)
{
    return (a + b + c) / 3.0f;
}
```

The body of the function is its definition/implementation.

- The **implementation** of a function, what it actually **does**.
- If we were to call this function, what would happen?
- Without definitions, nothing would happen (and the build process would fail).
- Needed by the **linker**

Missing Definition? That’s a Linker Error.

```
float CalculateAverage(int one, int two, int three);

int main()
{
    float result = CalculateAverage(4, 5, 1);
    return 0;
}
```

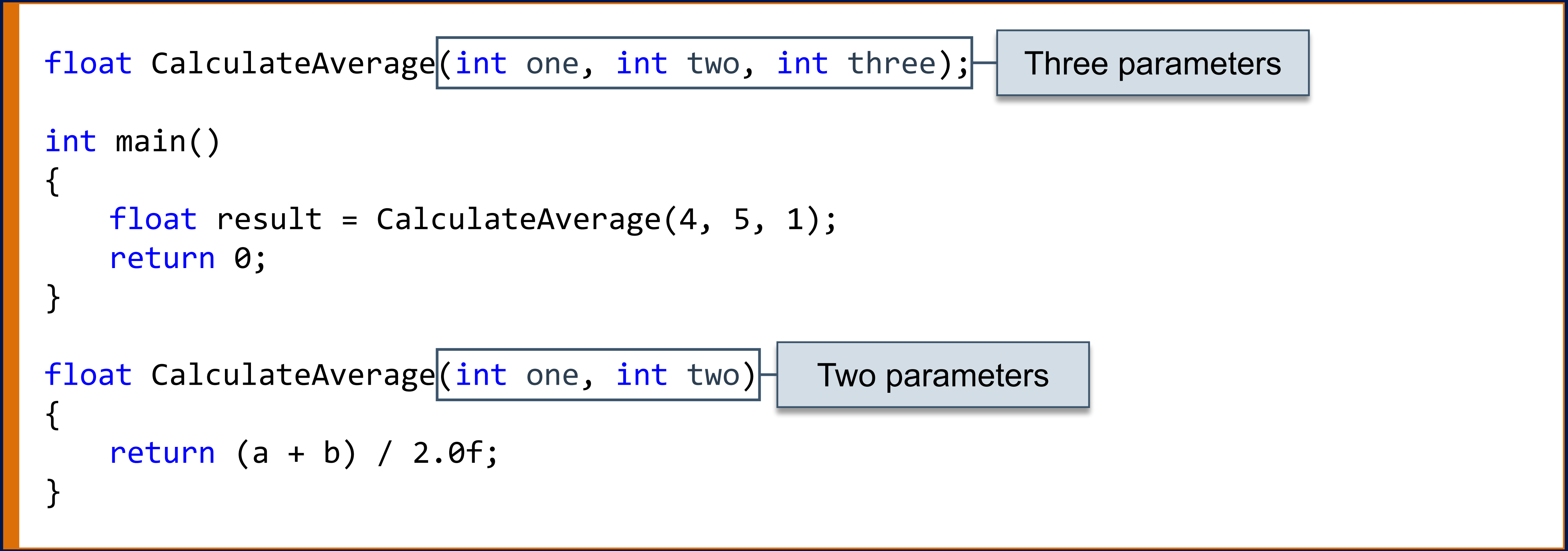
The **LNK** means linker error.

Unresolved external symbol essentially means “I can’t find a definition for a function”.

```
Error LNK2001 unresolved external symbol "float __cdecl
CalculateAverage(int,int,int)" (? CalculateAverage@@YAMHHH@Z)
```

In the middle of all this, we can see the name of our missing function.

Prototypes and Definitions Must Match!



```
float CalculateAverage(int one, int two, int three);

int main()
{
    float result = CalculateAverage(4, 5, 1);
    return 0;
}

float CalculateAverage(int one, int two)
{
    return (a + b) / 2.0f;
}
```

```
void Foo();
void Bar();
void Baz();

int main()
{
    Foo();
    Bar();
    Baz();
    return 0;
}

void Foo()
{}

void Bar()
{}

void Baz()
{}
```

Foo? Bar? Generic placeholder names, to indicate “the name is irrelevant”.

This feels like a lot of repetition.

Why not this?

```
void Foo()
{}

void Bar()
{}

void Baz()
{}

int main()
{
    Foo();
    Bar();
    Baz();
    return 0;
}
```

Why not just declare and define everything above main()?

It’s Not All About main()

```
void Foo()
{}

void Bar()
{
    Foo(); // OK, the compiler "sees" Foo()
    Baz(); // Compiler error, Baz() not declared yet
}

void Baz()
{}

int main()
{
    return 0;
}
```

Solution:
Change the order of the functions!

#problemsolved

```
void Foo()
{}

void Baz()
{}

void Bar()
{
    Foo(); // OK
    Baz(); // OK
}

int main()
{
    return 0;
}
```

What if we make another change?

```
void Foo()
{
    Baz(); // Sigh, error…
}
void Baz()
{}

void Bar()
{
    Foo(); // OK
    Baz(); // OK
}

int main()
{
    return 0;
}
```

Prototypes

Prototypes prevent you from having to reorder your definitions…

```
void Foo();
void Bar();
void Baz();
```

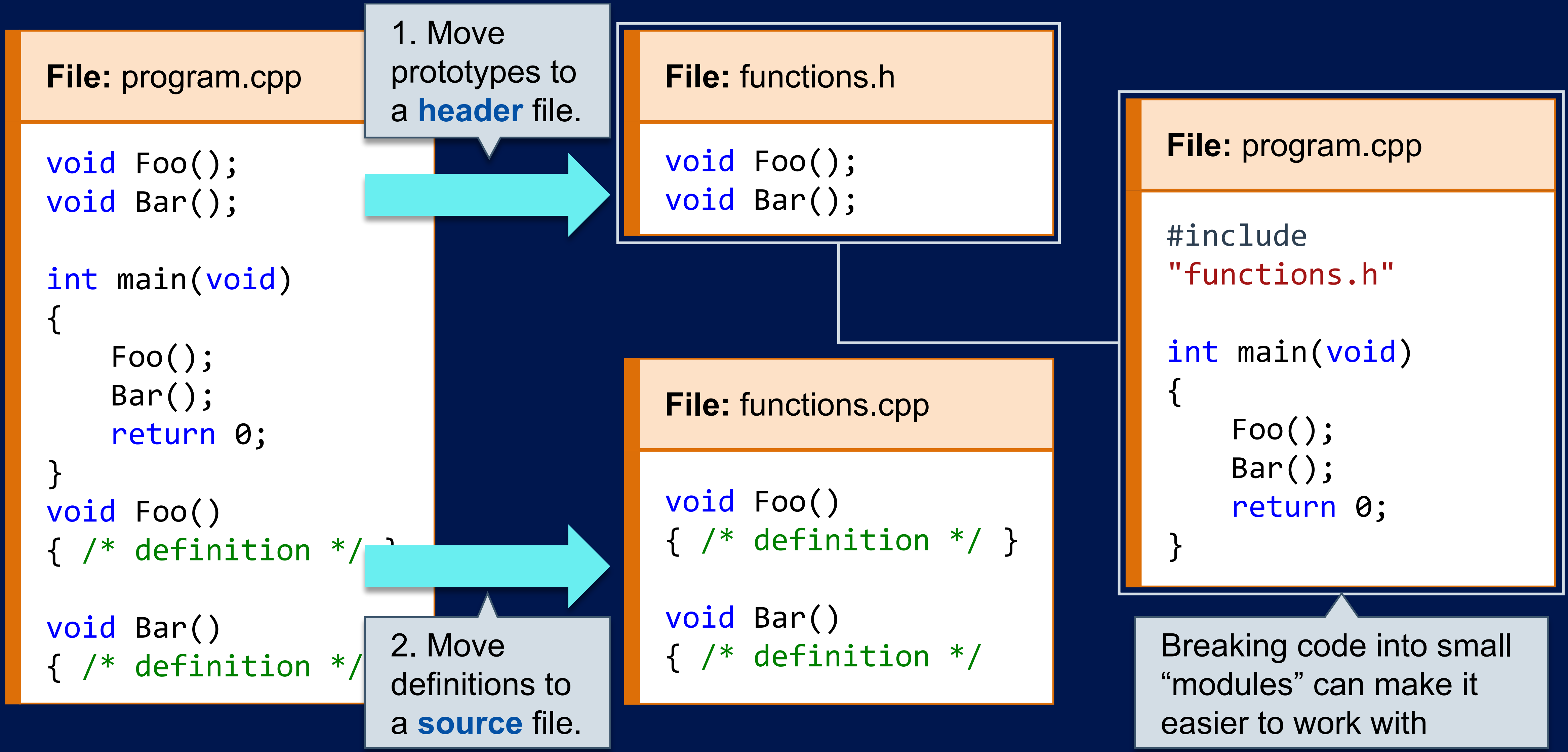
All the code below this knows about all the functions.

```
int main()
{
    return 0;
}

void Foo()
{
    Baz(); // No problem
}

void Bar()
{
    Baz(); // OK
    Foo(); // OK
}

void Baz()
{}
```

Reordering your functions every time you make some changes is not a good way to write code!

This may seem like a lot of minutiae right now, but it will be very important later!

Working With Multiple Files



File: program.cpp

```
void Foo();
void Bar();

int main(void)
{
    Foo();
    Bar();
    return 0;
}
void Foo()
{ /* definition */

void Bar()
{ /* definition */
```

1. Move prototypes to a **header** file.

File: functions.h

```
void Foo();
void Bar();
```

2. Move definitions to a **source** file.

File: functions.cpp

```
void Foo()
{ /* definition */ }

void Bar()
{ /* definition */
```

File: program.cpp

```
#include
"functions.h"

int main(void)
{
    Foo();
    Bar();
    return 0;
}
```

Breaking code into small “modules” can make it easier to work with

Header Files? Source Files?

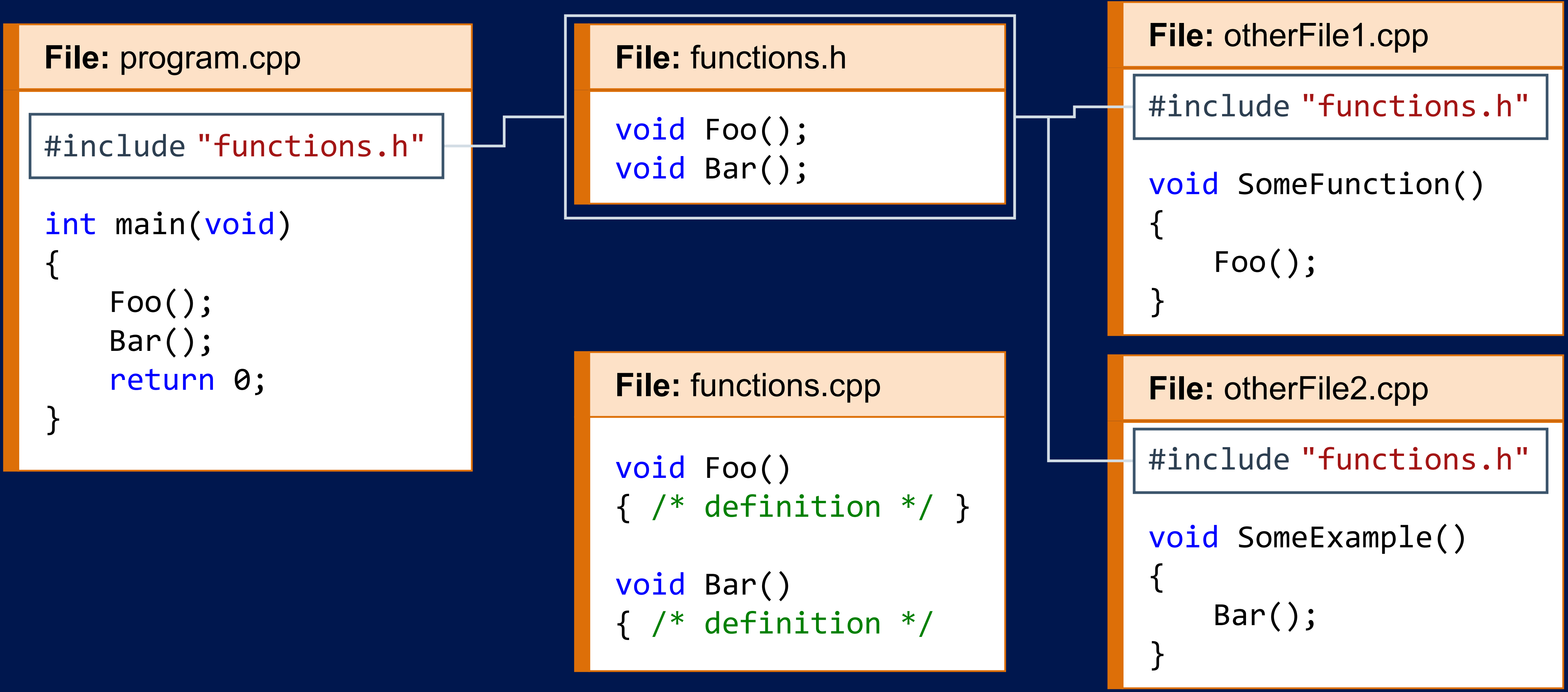
Header files

- Where the **declaration** goes
- Serve as a “table of contents” for other parts of your code
- Typically .h files, but could also be .hpp, .hxx, .h++

Source files

- Where the **definition** / **implementation** goes—i.e., the “real” code
- Typically .cpp files, but could be .c, .cc, .cxx or even .c++

Using and Reusing Functions Across Files



File: program.cpp

```cpp
#include "functions.h"

int main(void)
{
    Foo();
    Bar();
    return 0;
}
```

File: functions.h

```cpp
void Foo();
void Bar();
```

File: functions.cpp

```cpp
void Foo()
{ /* definition */ }

void Bar()
{ /* definition */
```

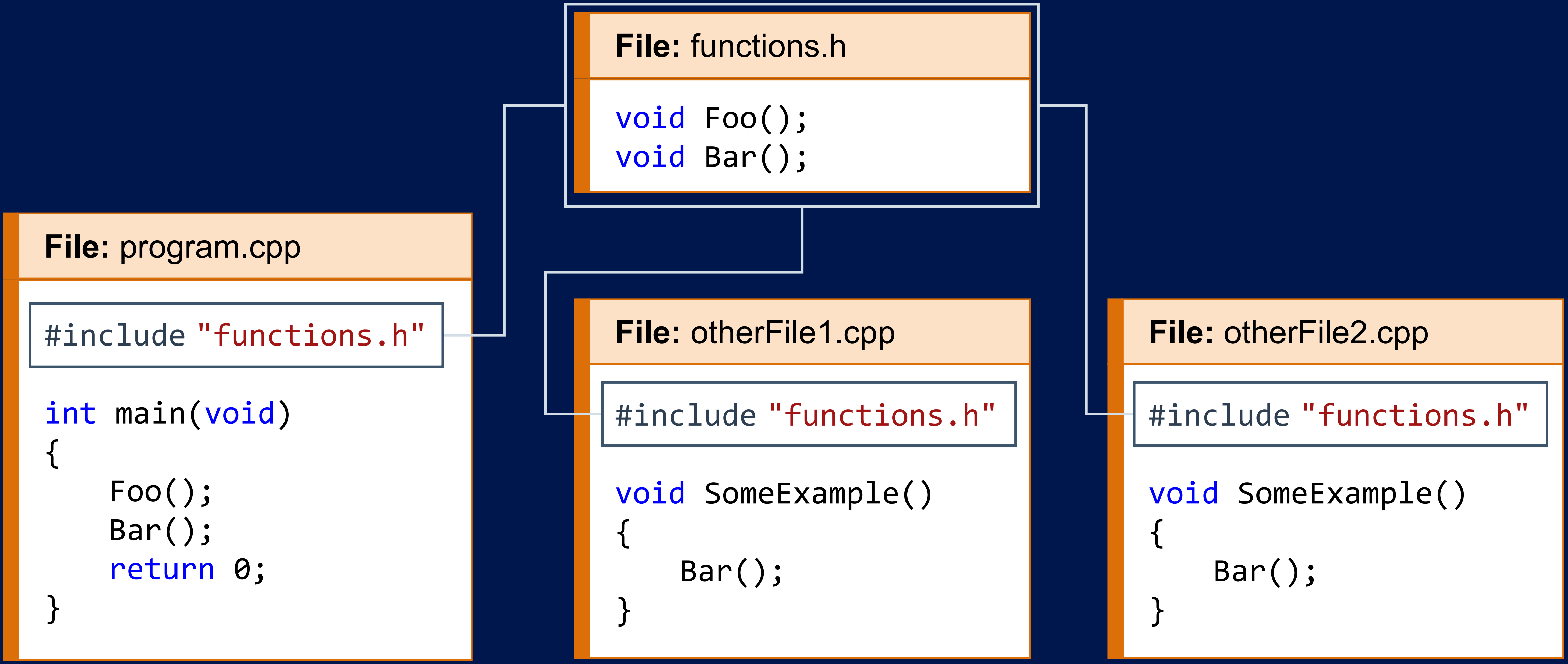
File: otherFile1.cpp

```cpp
#include "functions.h"

void SomeFunction()
{
    Foo();
}
```

File: otherFile2.cpp

```cpp
#include "functions.h"

void SomeExample()
{
    Bar();
}
```

Getting Code With #include



File: functions.h

```
void Foo();
void Bar();
```

File: program.cpp

```
#include "functions.h"

int main(void)
{
    Foo();
    Bar();
    return 0;
}
```

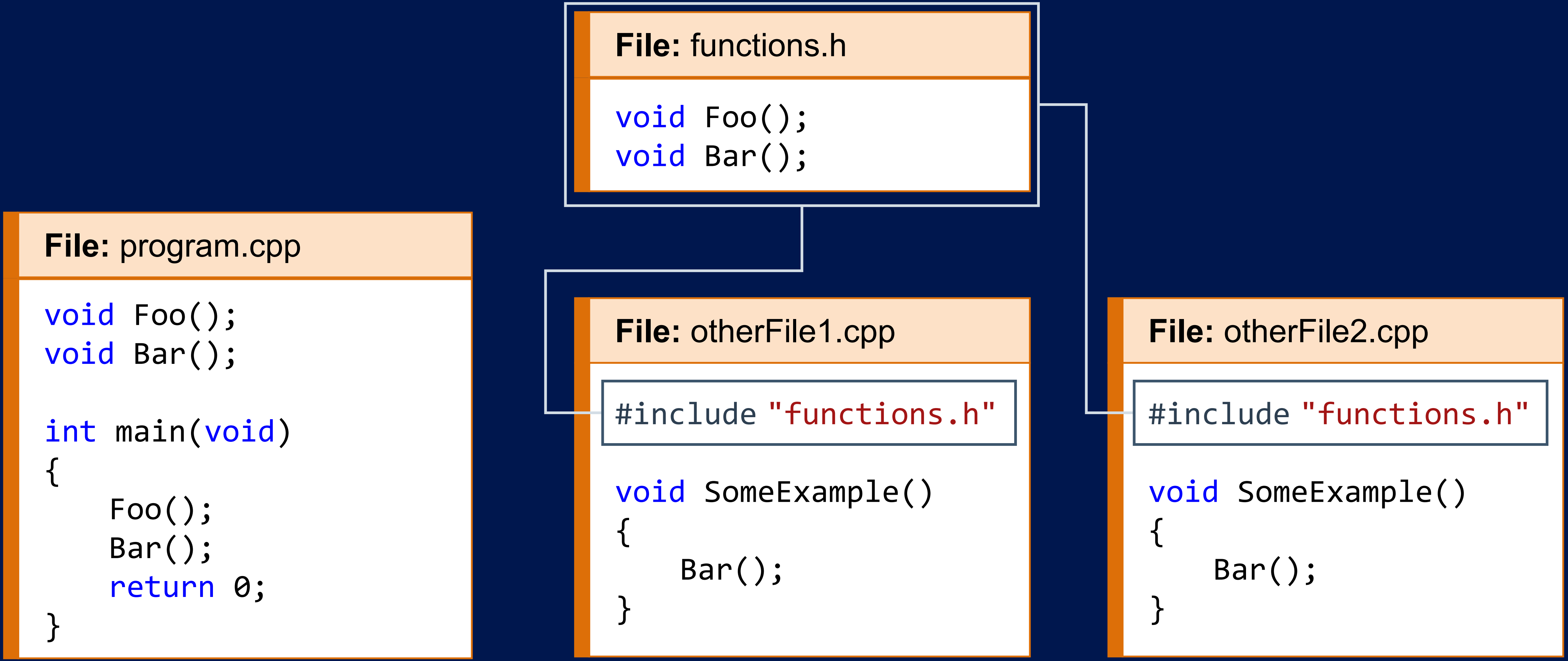
File: otherFile1.cpp

```
#include "functions.h"

void SomeExample()
{
    Bar();
}
```

File: otherFile2.cpp

```
#include "functions.h"

void SomeExample()
{
    Bar();
}
```

Getting Code With #include



File: functions.h

```cpp
void Foo();
void Bar();
```

File: program.cpp

```cpp
void Foo();
void Bar();

int main(void)
{
    Foo();
    Bar();
    return 0;
}
```

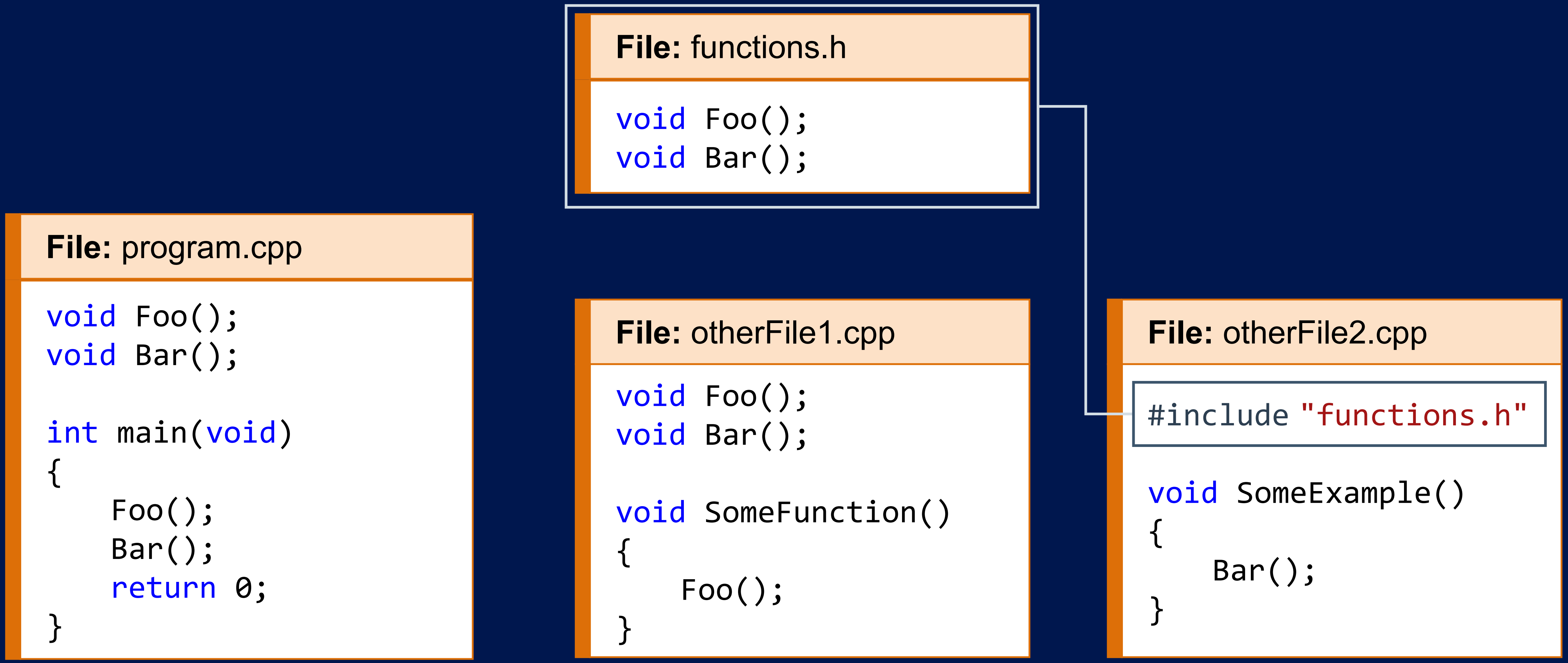
File: otherFile1.cpp

```cpp
#include "functions.h"

void SomeExample()
{
    Bar();
}
```

File: otherFile2.cpp

```cpp
#include "functions.h"

void SomeExample()
{
    Bar();
}
```

Getting Code With #include



File: functions.h

```cpp
void Foo();
void Bar();
```

File: program.cpp

```cpp
void Foo();
void Bar();

int main(void)
{
    Foo();
    Bar();
    return 0;
}
```

File: otherFile1.cpp

```cpp
void Foo();
void Bar();

void SomeFunction()
{
    Foo();
}
```

File: otherFile2.cpp

```cpp
#include "functions.h"

void SomeExample()
{
    Bar();
}
```

Getting Code With #include

File: functions.h

```
void Foo();
void Bar();
```

File: program.cpp

```
void Foo();
void Bar();

int main(void)
{
    Foo();
    Bar();
    return 0;
}
```

File: otherFile1.cpp

```
void Foo();
void Bar();

void SomeFunction()
{
    Foo();
}
```

File: otherFile2.cpp

```
void Foo();
void Bar();

void SomeExample()
{
    Bar();
}
```

Getting Code With #include

File: functions.h

```
void Foo();
void Bar();
void Baz();
void Test();
```

File: program.cpp

```
void Foo();
void Bar();

int main(void)
{
    Foo();
    Bar();
    return 0;
}
```

File: otherFile1.cpp

```
void Foo();
void Bar();

void SomeFunction()
{
    Foo();
}
```

File: otherFile2.cpp

```
void Foo();
void Bar();

void SomeExample()
{
    Bar();
}
```

Getting Code With #include

File: functions.h

```
void Foo();
void Bar();
void Baz();
void Test();
```

File: program.cpp

```
void Foo();
void Bar();
void Baz();
void Test();

int main(void)
{
    Foo();
    Bar();
    return 0;
}
```

File: otherFile1.cpp

```
void Foo();
void Bar();
void Baz();
void Test();

void SomeFunction()
{
    Foo();
}
```

File: otherFile2.cpp

```
void Foo();
void Bar();
void Baz();
void Test();

void SomeExample()
{
    Bar();
}
```

What About the Function Definitions?

File: program.cpp

```
#include "functions.h"

int main(void)
{
    Foo();
    Bar();
    return 0;
}
```

File: functions.h

```
void Foo();
void Bar();
```

File: otherFile1.cpp

```
#include "functions.h"

void SomeExample()
{
    Bar();
}
```

File: functions.cpp

```
void Foo()
{ /* definition */ }

void Bar()
{ /* definition */
```

File: otherFile2.cpp

```
#include "functions.h"

void SomeExample()
{
    Bar();
}
```

Can I `#include` a .cpp File?

- **Can you**? Yes. The language doesn't prevent you from doing it
- **Should you**? No. It goes against common style and is likely to cause errors.
- .CPP files are for definitions, and definitions must be unique.
- If you copy/paste definitions, you are redefining things.

To avoid issues, only include header files, **never** source (.cpp) files.

Lots of #include Directives

```cpp
#include <iostream>
#include <string>
#include <vector>
#include <map>
#include "MyFile.h"

int main()
{
    /* Your awesome code here */
    return 0;
}
```

`#include <iostream>`

- Search “official directories” for this file
- Use this for built-in C++ files, or library files

`#include "MyOwnHeader.h"`

- Search locally, in the directory of the file containing the directive
- File not found? Then check the official locations
- Use quotes for **programmer-generated files** (i.e. files **you** create)

You can also add custom directories for your compiler to search; we’ll look at that later.

#include Guards

- If you #include a header file in multiple places, it may cause issues.
- We can use include guards to protect against this.
- Including guards prevents something in a file from being defined more than once.

```
#ifndef _UNIQUE_ID_      // If NOT defined
#define _UNIQUE_ID_      // Define it

// Your code goes here

#endif /* _UNIQUE_ID_ */ // All done defining stuff
```

#include Guards

The old way

```
// File "HelperFunctions.h"
#ifndef _HELPER_FUNCTIONS_H_
#define _HELPER_FUNCTIONS_H_

#include <string>
using std::string;

float CalculateAverage(int a, int b, int c);
string RemoveFileExtension(string str);
void Foo();
void Bar(int foo);

#endif
```

The first part is actually two parts:

#ifndef means “if the identifier that comes after this is not already defined”.

#define means “define the identifier that comes after this”.

The header file code goes in between.

We end with **#endif**.

Writing these 3 lines for every header file can get tedious; newer C++ has a shortcut!

#include Guards Shortcut

#pragma once

```
// File "HelperFunctions.h"
#pragma once


#include <string>
using std::string;

float CalculateAverage(int a, int b, int c);
string RemoveFileExtension(string str);
void Foo();
void Bar(int foo);
```

This replaces #ifndef, #endif, etc…

Most major modern compilers support this.

Much simpler, and the preferred way to do things today.

A Lot to Unpack

```
#include <iostream>
#include <string>
#include "MyFile.h"

int main()
{
    string text = "Hello, world!";
    std::cout << text << std::endl;
    FunctionFromMyFile_H();
    return 0;
}
```

- C++ has its own rules, like any language.
- Even small amounts of code can be dense!
- Learning how to write code in the language is one thing
- Learning how the language works is another thing.
- The two are often closely related—we have to learn about them both

Recap

- Building C++ programs is a complicated process.
- **Preprocessor, Compiler, Linker**
  3-step build affects how you write your code
- C++ compiles code from top to bottom.

Recap

- We need **declarations** and **definitions** to make things work.
- Code can be split into multiple files (**header** and **source** files) for organization.
- `#include` statements can share code across files.

Conclusion

Placeholder for the instructor’s welcome message. Video team, please insert the instructor’s video here.

Thank you for watching.