WEEK 1

Getting started with C++ and OOP

AGENDA

- Week 1-1
  - Type Safety, Input / Output, Namespaces
- Week 1-2
  - Object Oriented Programming Concepts and Terms
- Week 1-3
  - Modular Programming, Unit Testing

WEEK 1-1

Type Safety, Namespaces, Input/Output

| Getting started with C++

TYPE SAFETY

- Type safety is an important feature of the C++ language and was a central concept to its design philosophy.
- A programming language that is type-safe allows:
  - Containing errors at compilation (errors messages are your friends).
  - Lessening errors occurring during run time (exploding during run time is bad)

TYPE SAFETY EX 1-1

- Consider this snippet that compiles fine (albeit with a warning) in C but results in a segmentation fault.

```c
#include <stdio.h>

void foo();

void foo(char x[]){
  printf("%s", x);
}

int main(){

  foo(-25);
  return 0;
}
```

TYPE SAFETY EX 1-2

- The seg fault occurs due to trying to print out a char array with an invalid address

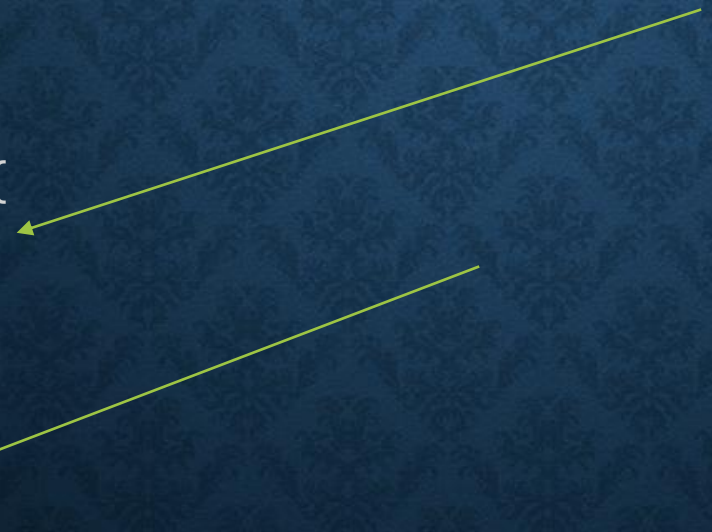
```c
#include <stdio.h>

void foo();

void foo(char x[]){
  printf("%s", x);
}

int main(){

  foo(-25);
  return 0;
}
```



TYPE SAFETY EX 1-3

- Compiling this same example as a C++ program results in an error message and no executable:

```
unsafe.cpp: In function 'int main()':
unsafe.cpp:12:15: error: invalid conversion from 'int' to 'char*' [-
fpermissive]
        foo(25);
              ^
unsafe.cpp:5:6: error:   initializing argument 1 of 'void foo(char*)' [-
fpermissive]
 void foo(char x[]){
       ^
```

TYPE SAFETY EX 1-4

- Having errors occurring before run time is good
- Having errors occurring during run time is bad
  - When the application is a critical piece of software, reducing errors during run time is paramount

- We'll see more ways C++ promotes type safety in the future

NAMESPACES

- Naming conflicts can become common when multiple people work on the same application or code base.
- C++ offers namespaces to avoid these issues
- Namespaces is a scope or enclosing space that separates code from other segments of code. These different scopes can be referenced to access that code (of things that may have similar names).
- Namespaces are also useful just to organize code that fall under similar domains

NAMESPACE EX 1-1

- If we have two functions named max, we will get a compilation error

```
int max() {

  return 100;
}

int max() {

  return 101;
}
```

```
names.cpp: In function 'int max()':
names.cpp:6:5: error: redefinition of 'int max()'
 int max() {
     ^
names.cpp:1:5: error: 'int max()' previously
defined here
 int max() {
     ^
```

NAMESPACES EX 1-2

- Scott's namespace world

```
namespace scott {

  int x = 2;
  int max() {
    return 888;
  }
}
```

- Nick's namespace world

```
namespace nick {

  int x = 3;
  int max() {
    return 999;
  }
}
```

- Now we can have two function named max and two x variables as long as keep them in their own namespaces.

NAMESPACE EX 1-3

- Making use of those namespace'd functions or variables is done by explicitly referring to the namespace that the functions or variable belong to:

```
nick::x++ // Increasing nick's x
scott::x-- // Decreasing scott's x
```

NAMESPACE USING KEYWORD

- An alternative to specifying the namespace every time you can specify which scope you are making use of in your code from the get go:

```
using namespace nick;
x++ // Increasing nick's x
```

- If you only want a single identifier from a namespace you can also specify that with the using keyword:

```
using nick::x;
x++ // Increasing nick's x
```

INPUT/OUTPUT

- Printing to the standard output device or accepting data from the standard input device in C++ takes a different shape than in C
- C++ makes use of objects from its <iostream> library (as compared to the <stdio.h> library in C) to do input and output. These objects cout and cin are used differently from the C languages printf() and scanf() but they are quite convenient.

OUTPUT C VS C++

C

```c
#include <stdio.h>

int main(){

int x = 2;
char y [] = "Hello";
printf("%s %d", y, x);
}
```

C++

```cpp
#include <iostream>
using namespace std;

int main(){

int x =2;
char y [] = "Hello";
cout << y << x << endl;
}
```

OUTPUT C VS C++

C

- Notice the format strings

```c
#include <stdio.h>

int main(){

int x = 2;
char y [] = "Hello";
printf("%s %d", y, x);
}
```

Format string

C++

- Notice the lack of format strings

```cpp
#include <iostream>
using namespace std;

int main(){

int x =2;
char y [] = "Hello";
cout << y << x << endl;
}
```

INPUT C VS C++

C

- Notice the format strings

```c
#include <stdio.h>

int main(){

int x;
printf("Enter a number: ");
scanf("%d", &x); // Input
printf("You entered a number: %d\n", x);
}
```

C++

- Notice the lack of format strings

```cpp
#include <iostream>
using namespace std;

int main(){

int x;
cout << "Enter a number: ";
cin >> x; //Input
cout << "You entered a number: " << x << endl;
}
```

INPUT/OUTPUT IN C++

- Notice that the syntax for input and output appear be like a stream which data flows from one side to another via the use of operators like the >> and <<
- It is fairly similar to the redirection of data you have seen and used in Linux
- This syntax is more natural and cleaner compared to printf and scanf
  - The cin and cout objects appear to understand the kind of data you're intending on without the use of format strings specifying it explicitly

WEEK 1-2

Abstraction, Encapsulation, Inheritance, Polymorphism

| Object Oriented Terminology

OBJECT ORIENTED

- An object in this class will refer to an organization of data and logic contained in a structure of sorts.
- This object reflects and attempts to describe an idea or physical being in a clean and concise manner
- Examples of such objects so far are the rather clean and perhaps magical cin and cout that perform input and output

OBJECT ORIENTED

- In the OOP style of programming there are 3 main qualities / ideas that comprise the style:
  - Encapsulation
  - Inheritance
  - Polymorphism

ABSTRACTION

- The idea of abstraction is fairly key to OOP.
- It involves the process of generalization and reduction so that commonalities of ideas are grouped together to create a crisper representation. This also leads to reduction of code.
- Consider the cout object that represents the standard output device
  - In its usage so far it appears to be abstraction of how we would interact with printf. Details such as format codes are now hidden with in the object. Out interaction with cout is crisp in that we only need to supply the data to be outputted which is what we really find important.

ENCAPSULATION

- Encapsulation relates closely to the idea of objects and classes in that is the primary concept of object-oriented programming. The main idea is the integration of logic (or behavior) and data within an object (or a class of objects).
- This closely coupling of logic and data allows for the crisp interfaces as seen from cout or cin but the details about how those objects work are hidden internally.

ENCAPSULATION

- A well encapsulated object/class is like a Blackbox. We can interact with it without knowing how it works in and out. This means any code interacting with the object doesn't need to change if the object's internal design changes

CLASSES

- Objects that share a common structure with each other are said to be of the same class (of objects)
- A class in C++ is quite similar to a struct, in that they both describe a sort of blueprint of an entity that contains some data and logic
- Consider the idea of a table, a chair and a bed. They're all furniture and as fellow furniture they likely share some qualities with one another

INHERITANCE

- Inheritance refers to classes that inherit the properties of another class
- This is another pillar of the OOP style
- It allows for code reuse

POLYMORPHISM

- The poly in polymorphism refers to "many" and the morphism refers to to "forms"
- In OOP this refers to the relation or calling of an implementation based on its type (of object)
- It is yet another pillar of OOP

WEEK 1-3

Modules, Compiling

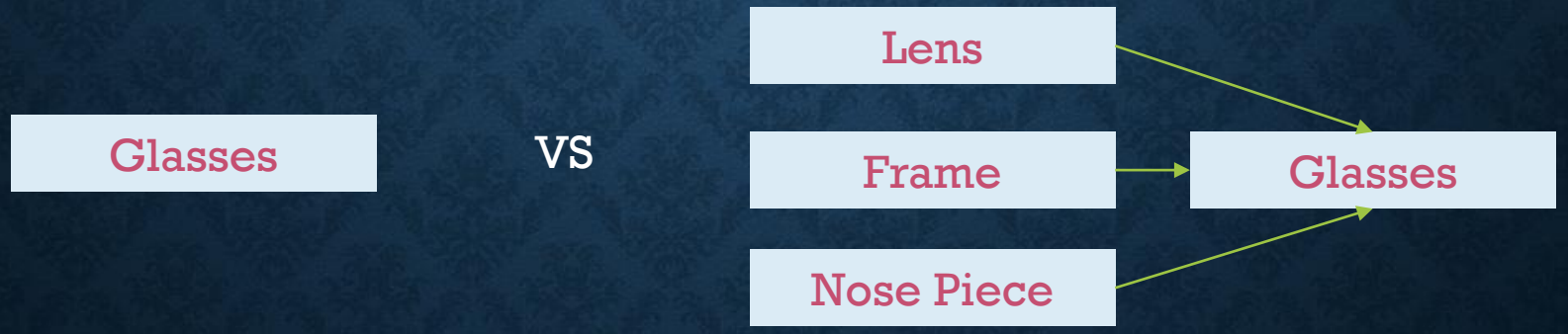
| Modular Programming

MODULES

- Splitting off portions of code into their own separate modules is generally a good idea.
- The main reasoning being each portion can work on its own and be incorporated as needed.
- This is the idea of being ‘loosely coupled’

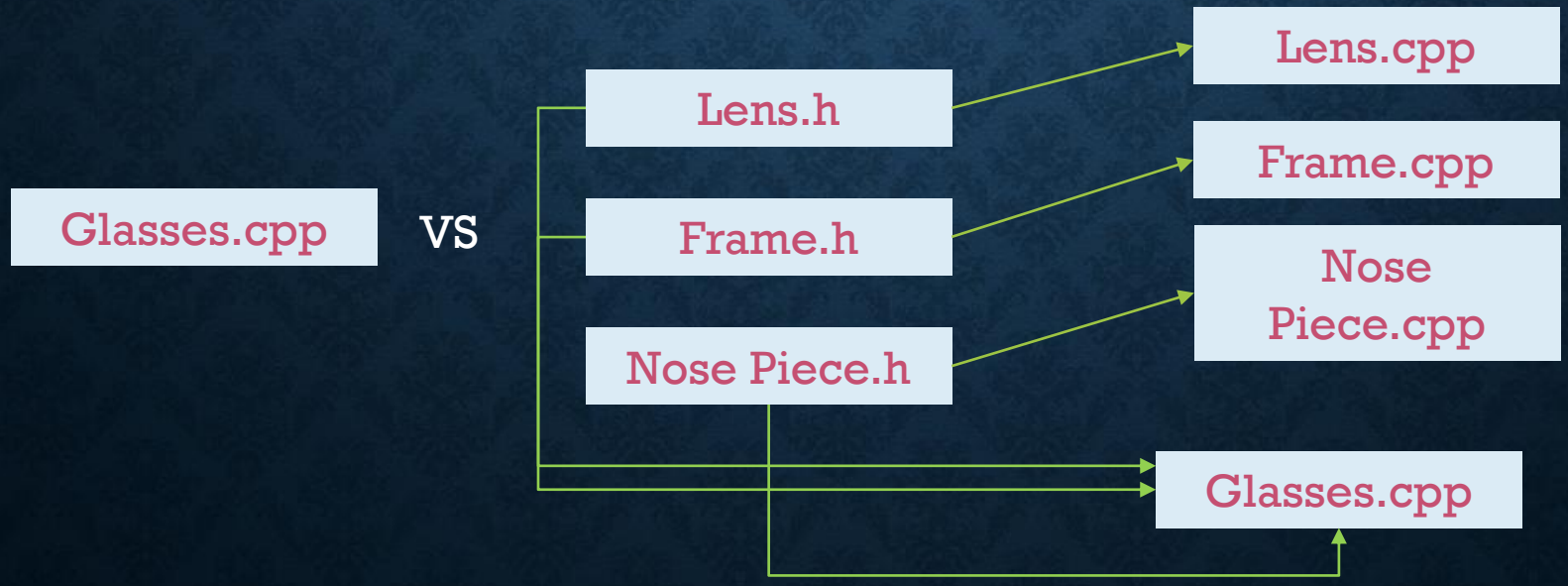
MODULES

- Consider the following example with Glasses.
- Glasses aren't a single item but rather an assembly of parts
- Should a piece of it need maintenance typically it's isolated to that piece.



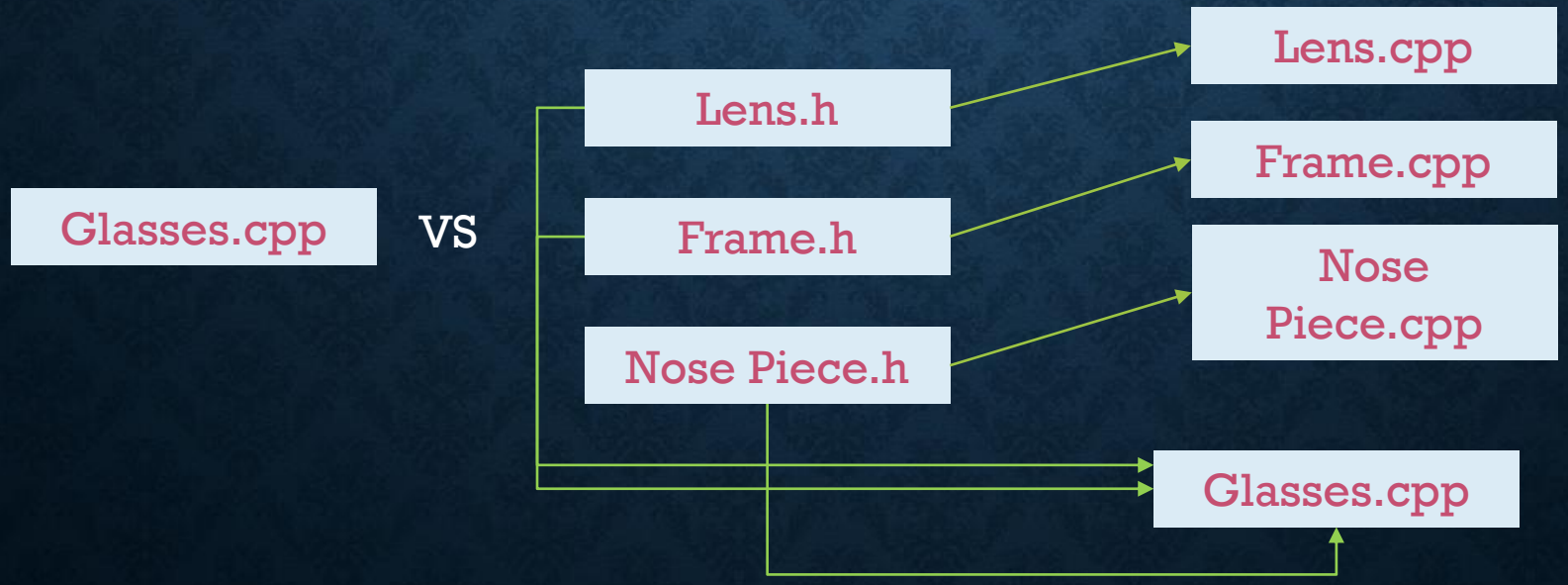
MODULES

- If instead of a real pair of glasses it was a C++ program representation of glasses, the idea of modules remains
- The more apt comparison here would be instead of storing all of our code in inside a single Glasses.cpp file we would keep the separation of pieces, each with their owner header/implementation and #include them in our Glasses.



MODULES

- This notion of modularity allows for the idea of separate compilation. If we need to change the Lens of our Glasses, the respective Lens.h and Lens.cpp files are the only things we would need to affect. Everything else could be compiled as is.

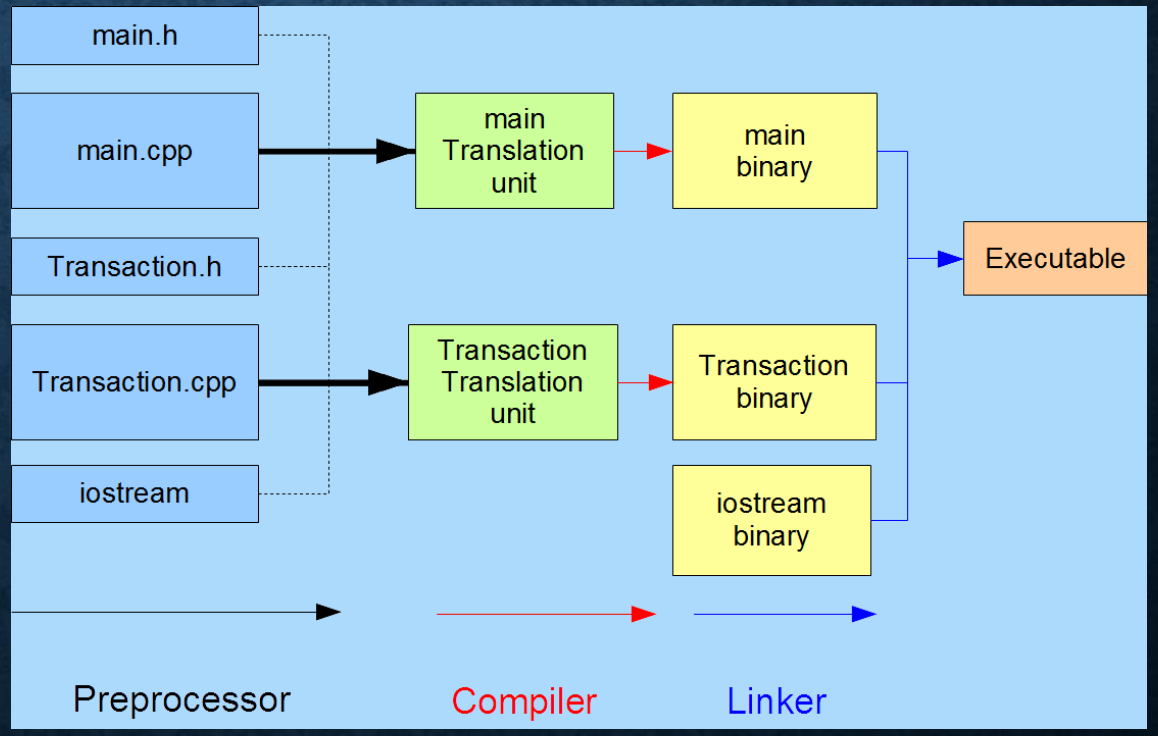


COMPILATION

- Much like C, C++ is a compiled language
- We create our implementation files (.cpp) with any headers files (.h) they may include and are then compiled through a compiler like g++
- The end results of this process is usually an executable binary

COMPILATION PROCESS

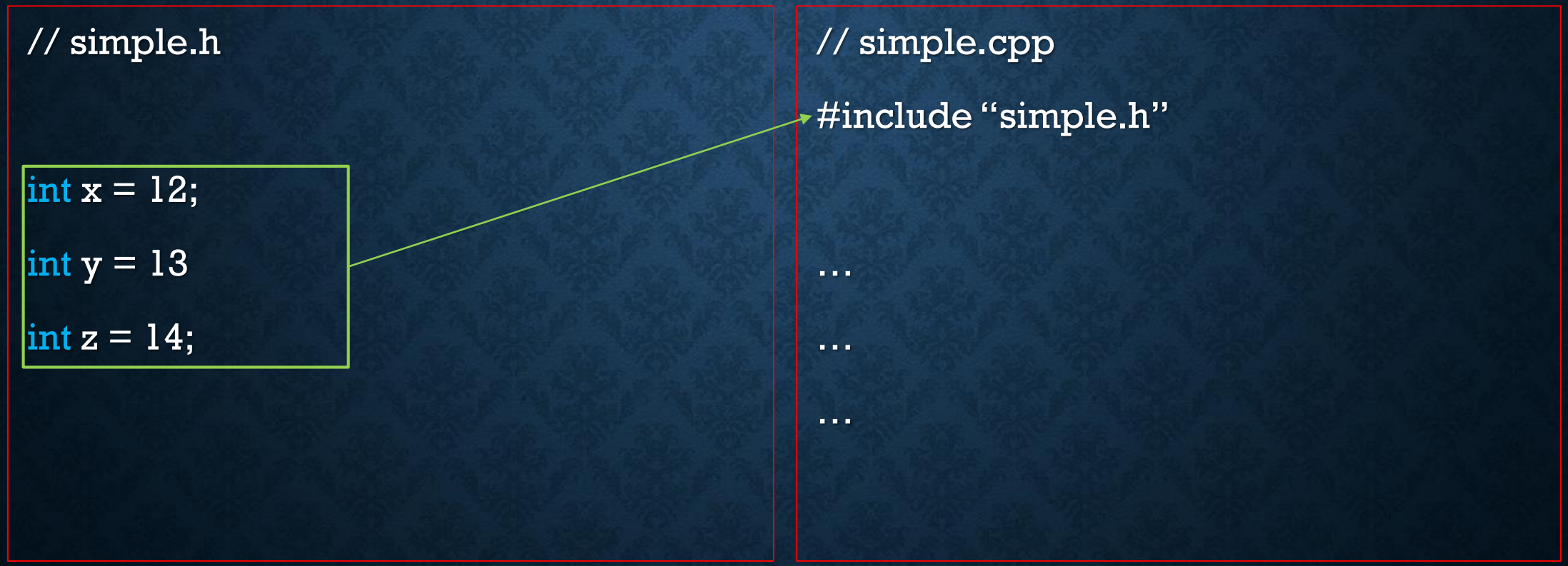
- The compilation process is actually broken down into 3 major steps:
  1. Preprocessor
  2. Compiler
  3. Linker



PREPROCESSOR

- This first step in the compiling process refers to the insertion / substitution of code where directives occur.
- Directives are the lines of code that generally start with the # symbol.
- #include and #define lines are examples of directives
- The preprocessor step replaces these lines with what they refer to
  - Eg. in the case of an #include the header file you specified to be included will have its contents inserted at that line
- Once this process is done the end result is something called a translation unit.
- Separate implementation files produce separate translation units

PREPROCESSOR



COMPILER

- The compiler step of the process then takes each of those translation units and does the actual compiling
- This generates binary code (machine code) from those units
- The resulting files of this process are referred to as object files

LINKER

- At the last step of the process the produced object files from the compiler are linked together to form an executable binary

UNIT TESTING

- As you continue to learn programming you will encounter and will continue to encounter situations where some code will rely on previously written stuff.
- If that previous stuff doesn't have consistent behavior then YOU're building on top of some assumptions that it does what you expect.
- What if it was YOU yourself that wrote that old code?
- In comes the need for testing.

UNIT TESTING

- A Unit Test is a code snippet that tests a single assumption in a module or work unit of a larger application.
- In our context an example of a work unit is a single function or a struct/class.
- If we can have this unit pass our test then we can be assured it works properly. If we change our unit and it still passes the test then great, we didn't break anything.
- The workflow that has it such that the tests are written before the actual functions is considered to be "Test Driven Development" or TDD.
- Writing tests is fairly connected to modular programming, such that we can confirm the working state of our individual modules.

UNIT TESTING

- As your assignments/projects get more complicated, consider writing test cases.