Week 1

Introduction & Review & Overview

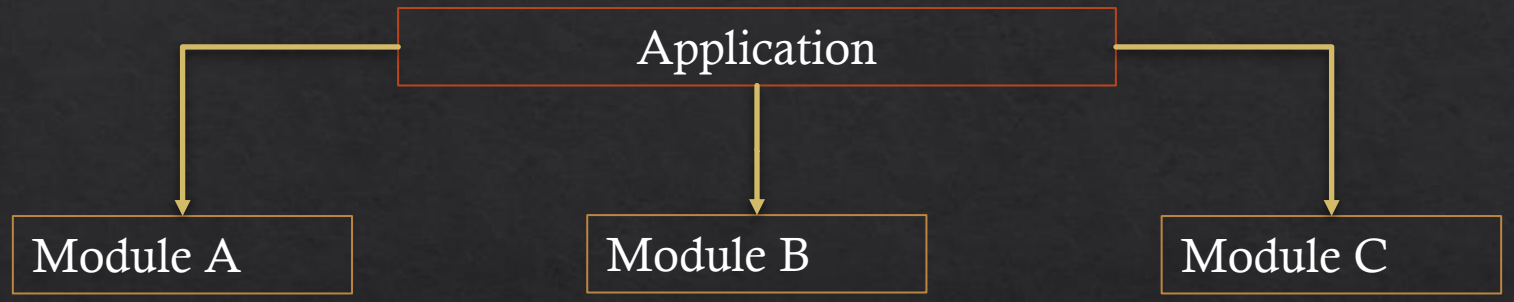
Agenda

- Week 1-1 – Old & New Ideas
  - Objects / Types
  - Declarations / Definitions
  - Linkage
  - Anonymous Namespaces
  - Compiling / Main Program
  - Compile time evaluations

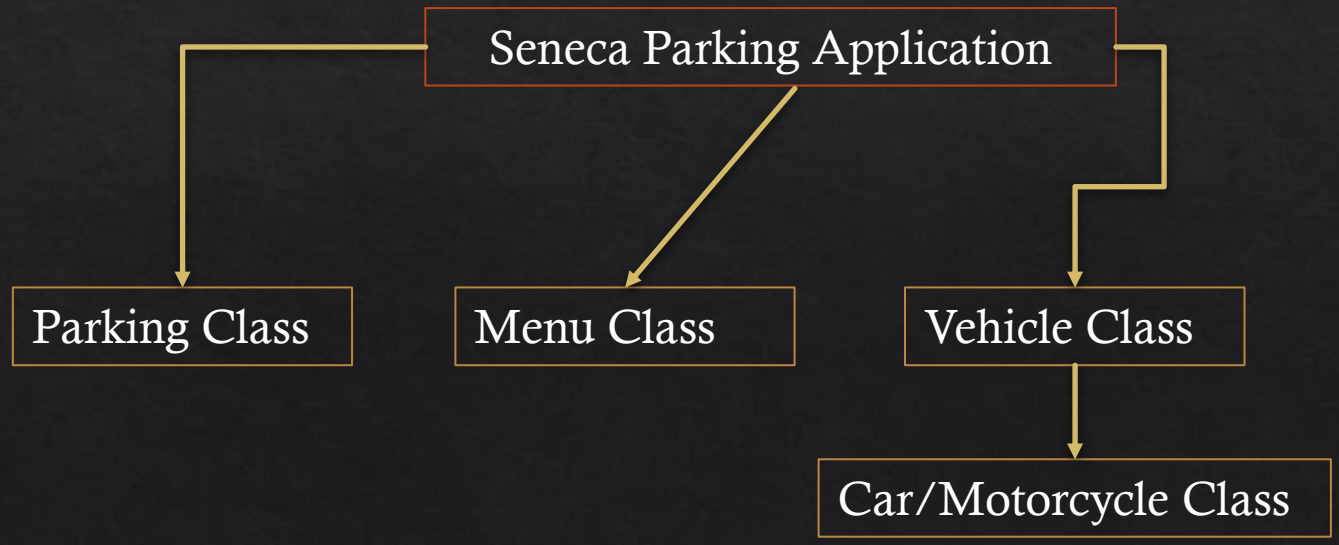
Object Oriented Programming

- Encapsulation
  - Objects
  - Classes/Structs
  - Member Functions
  - …
- Inheritance
  - Derived Classes
  - Shadowing
  - Protected access
  - …
- Polymorphism
  - Virtual functions
  - Templates
  - Abstract base classes

Modularity



Each module being separately compiled and a loosely coupled component of the whole application



Objects & Types

- The general building blocks of an OOP language are **objects**
- Sometimes these objects have names, and for these ones we call them **variables**
- Other times they don't and just occupy a region of memory (temporary objects)
- Regardless objects hold values that designate their current state

Objects & Types

- With the use of OOP, we also have the notion of using **objects** of a **type**
- Some of the types we've come to know are:
  - **Fundamental** types (int, char…)
  - **Built-in** types (eg. std::string)
  - **User-defined** types
    - **Concrete** (Regular classes)
    - **Abstract** (ABCs)
- Depending on the type, our objects act and behave differently (to represent different things)

Types – Type Definition

- In cases where we might have long type names it can be useful to shorten them through the use of typedefs
- A typedef is an alias for a type:

Consider a user defined type called:

```
class mylonglonglongtype{
…
}
```

We could declare an instance via:
mylonglonglongtype object;

Use a typedef to shorten it first:

```
typedef mylonglonglongtype mylt;
```

Declare an instance via:
mylt object;

Types – cv Qualifiers

- Along with the types themselves we can also apply some qualifiers that change how that type is stored in our program. These are known as cv qualifers
- The ‘c’ portion is a well known as the const keyword which causes a type to be unmodifiable as well as not be subject to any side-effects
- The ‘v’ portion represents the volatile keyword which indicates a type to be modifiable but subject to some side-effects
- When const and volatile qualifiers are used together both of their qualities are granted (unmodifiable but subject to some side-effects)
- When a type doesn’t apply either qualifier it is then considered cv unqualified

Types – cv Qualifiers

- In practice what does volatile mean in our programs?
- Effectively it means that:
  - Our volatile type has no guarantee its value won't be changed outside of program or be affected by any action that carries side effects
  - Any compiler applied optimizations will be avoided for this type that assumes no side effects
  - In the case of hardware related operations (consider working in embedded systems with their registers), when accessing the value of this type, a cached value won't be used and instead the actual memory location will be read instead

Declarations

- Declarations are used to introduce an identity into our programs (int x – declares a integer named x…) and when that occurs the identity or name becomes visible to that part of the program
- That 'part' of the program is then known as the scope of which there are a variety:
  - **Local, Class, Namespace, Global**
- **At the end of the scope is also the end of the lifetime of our variables**

Declarations – Block Scope

- The notion of a code block in C++ is any set of instructions enclosed with curly braces. And the block scope is then defined as by those same braces.

```
{ // Start of Code Block
int x;



} // End of Code Block
```

Beginning of the life time of any declarations

End of that life time

Declarations – Block Scope

- A example of a common block scope in use is an if statement:

```
int x = 15;
if (x > 10){

    int x = 5;
    cout << x << endl;
}
cout << x << endl;
```

What is the value of x in each printout line?

Declarations – Shadowing

- Shadowing being the notion of having similarly named identifiers present in different scopes (from local to less local) causing the effect of the most 'local' version taking precedence in that scope

```
int x = 15;
if (x > 10){

    int x = 5;
    cout << x << endl;
}
cout << x << endl;
```

For the sake of readability/maintainability, shadowing like this is something to be avoided

Declarations – Shadowing

- With shadowed or similarly named identifiers there is sometimes the option to resolve them with scope resolution operators or the this keyword

```
int x = 15;
if (x > 10){

    int namespace::x = 5;
    std::cout << x << std::endl;
}
std::cout << x << std::endl;
```

Declarations – Initialization in Selection

- Something new to C++17 is the ability to declare and initialize variables inside the condition of a selection statement (like an if statement or a switch case)

```
int y;
cin >> y;
int z = y / 2;
if (z > 10){
    cout << "greater" << endl;
}
else{
    cout << "lesser" << endl;
}
```



```
int y;
cin >> y;
if (int z = y / 2; z > 10){
    cout << "greater" << endl;
}
else{
    cout << "lesser" << endl;
}
```

Declarations – Initialization in Selection

- Using this kind of initialization requires the use of the C++17 build flag (--std=c++17)
- It's quite similar to the transition from C to C++ with regards to looping variables

```
int x = 0;
for (x < 10; x++){

…
}
```

```
for (int x = 0; x < 10; x++){

…
}
```

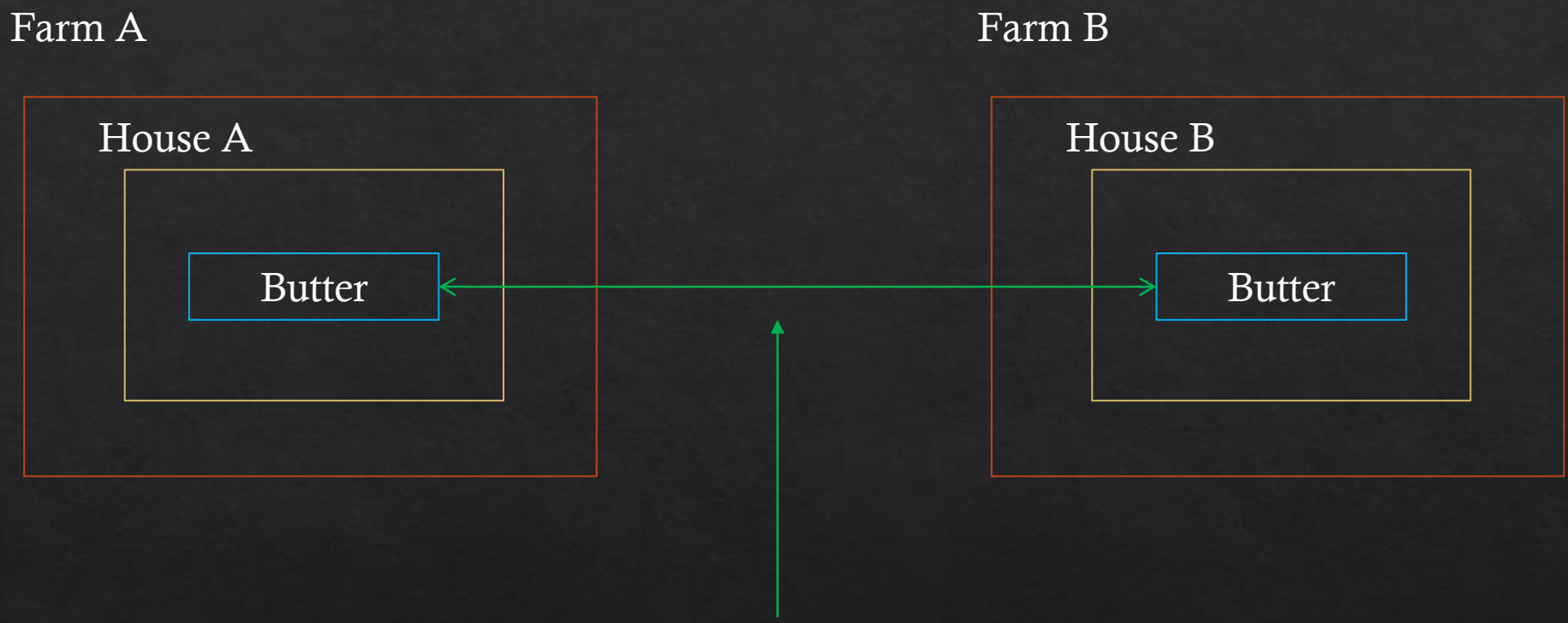
Definitions

- With each Declaration in our programs they're likely paired with a definition
- The definition gives some meaning to the declared name in our program scopes
- There are a variety of definitions from variables to functions to classes
- One rule we abide by with each declaration is there should only be one definition for a given name/identifier – The one definition rule

Linkage

- The notion of Linkage is somewhat as it sounds, it is the idea of whether or not a identifier / name refers to (or is 'linked' to) another identical name in another scope
- There are two kinds of linkage that could be in play:
  - External – Where two connected names are in different scopes in different modules
  - Internal – Where two connected names are in different scopes but in the same module
- Linkage is optional so an identifier can be just itself standalone

External Linkage



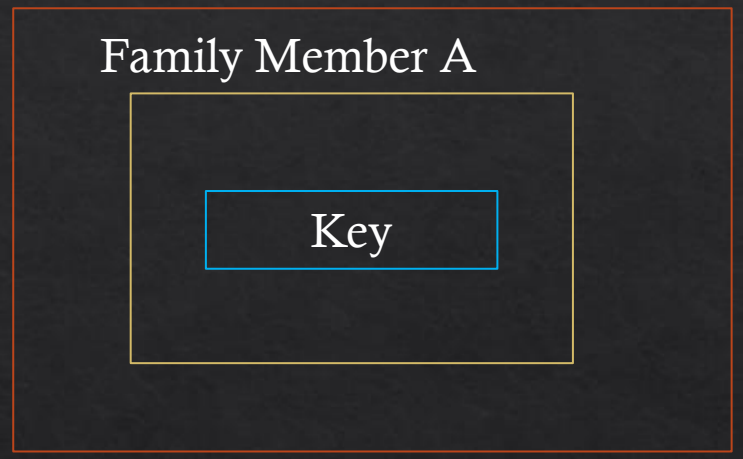
Imagine two neighbor farms that share butter with each other, while they're not on the same plot of land, the butter is linked

External Linkage

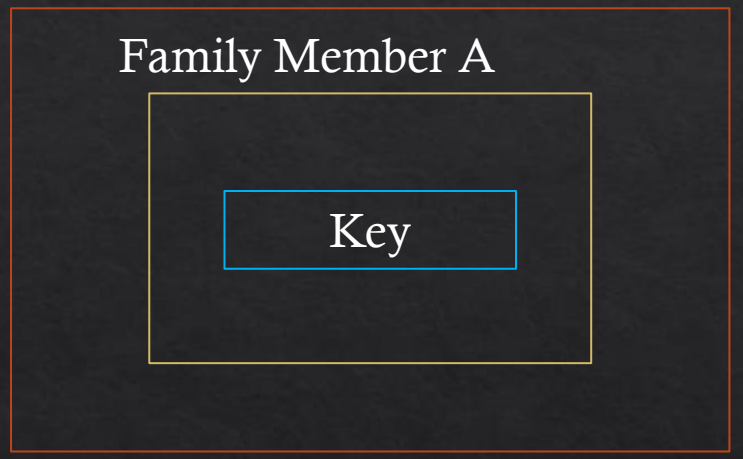
- To enable external linkage we can make use of the extern keyword
- Appending extern prior to a declaration will indicate to the compiler that this identifier should be resolved externally (in another translation unit/module)

Internal Linkage

House A



House B



Imagine two households where the family members share a house key. This is only for those in their family.

Internal Linkage

- Enabling internal linkage means that a given identifier will become invisible outside of its own translation unit but will be visible to only those within (like the family house key)
- To enable something to be internally linked we append the keyword static before an identifier:

Module A

```
static int shared_int = 0;
```

Module B

```
static int shared_int;
```

Each of these ints are unique to each module

Anonymous Namespaces

- Similar to internal linkage described earlier, anonymous namespaces offer another method of localizing an entity to only one translation unit / module
- An entity enclosed by an anonymous namespace is only accessible from within that translation unit but anywhere in that unit can access it

```
namespace { // anon namespace

  void potato() {
    std::cout << "potato" << std::endl;
  }
}
namespace dog{
  void display() {
    std::cout << "dog ";
    potato();
  }
}
namespace cat{
  void display() {
    std::cout << "cat ";
    potato();
  }
}
```

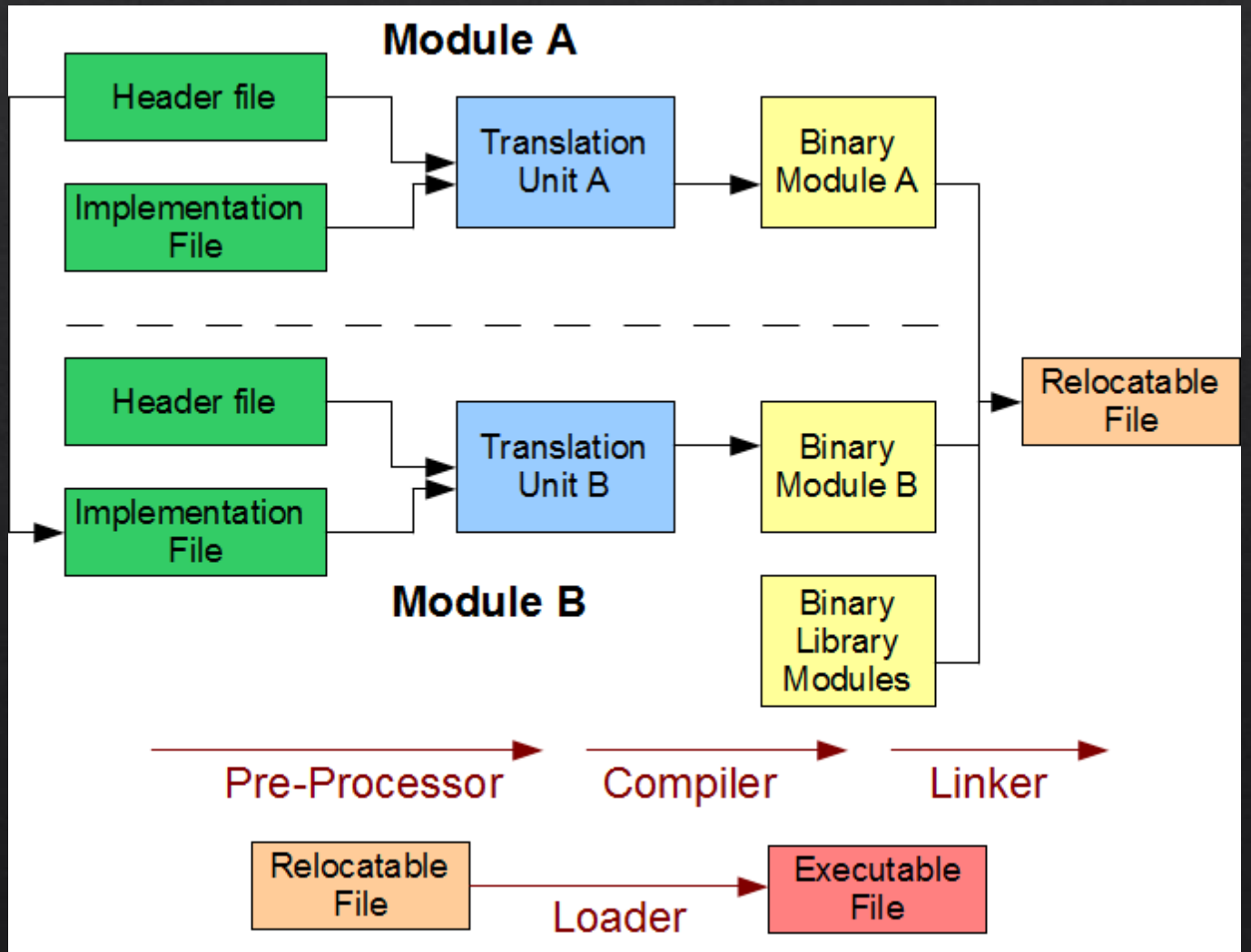
Anonymous Namespaces

- The use of an anonymous namespace can also help to address duplication within a set of other existing namespaces which may have similar functionality or needs

```
namespace { // anon namespace

  void potato() {
    std::cout << "potato" << std::endl;
  }
}
namespace dog{
  void display() {
    std::cout << "dog ";
    potato();
  }
}
namespace cat{
  void display() {
    std::cout << "cat ";
    potato();
  }
}
```

Compiling


Module A
Header file
Implementation File
Translation Unit A
Binary Module A
Header file
Implementation File
Translation Unit B
Binary Module B
Relocatable File
Module B
Binary Library Modules
Pre-Processor
Compiler
Linker
Relocatable File
Loader
Executable File

Main Program

- The entry point to our C++ programs is the main() function and the form we've mostly used up to this point in the courses here is:

```
int main(){ // default main with no arguments, returns an integer
        return 0;
}
```

- However there's a secondary form (overload) that takes in two parameters and with these parameters allows us to make use of command-line arguments with our program:

```
int main(int argc, char* argv[]){
        return 0;
}
```

Main Program

The first param argc here is an integer that represents the number of arguments passed to our program

The second param argv is an array of character pointers that represent the actual arguments themselves

```
int main(int argc, char* argv[]){
    return 0;
}
```

Main Program

```
int main(int argc, char* argv[]){
    for (int i = 0; i < argc; i++){
        cout << argv[i] << endl;
    }
    return 0;
}
```

Through using those parameters we can access our command line arguments and this case we can simply print them out with a for loop

Main Program

```
int main(int argc, char* argv[]){
    for (int i = 0; i < argc; i++){
        cout << argv[i] << endl;
    }
    return 0;
}
```

One note about the return 0 at the end of main functions. If this return statement is missing from our program the compiler will insert a return 0 for us

This is actually optional to put

Compile Time Evaluations – Constant Expressions

- Constant expressions are much like what they sound like in that they're expressions that the compiler can evaluate at compile-time (vs doing it at run time)
- We make use of the constexpr keyword to indicate that this entity should be evaluated at compile-time
- Constexpr can be applied to both variables and functions to establish constant variables and constant functions

Compile Time Evaluations – Constant Expressions

Constexpr variables

```
constexpr int x = 10;

constexpr long int sumUp(long int x){
  if (x > 0)
    return x + sumUp(x - 1);
  return 0;
}
```

Constexpr functions

Compile Time Evaluations – Static Assertions

- In addition to constexpr are static assertions that allow for compile time custom error messages if an assertion fails
- It can be used in creating test cases for your code
- The syntax for static assertions look like this:

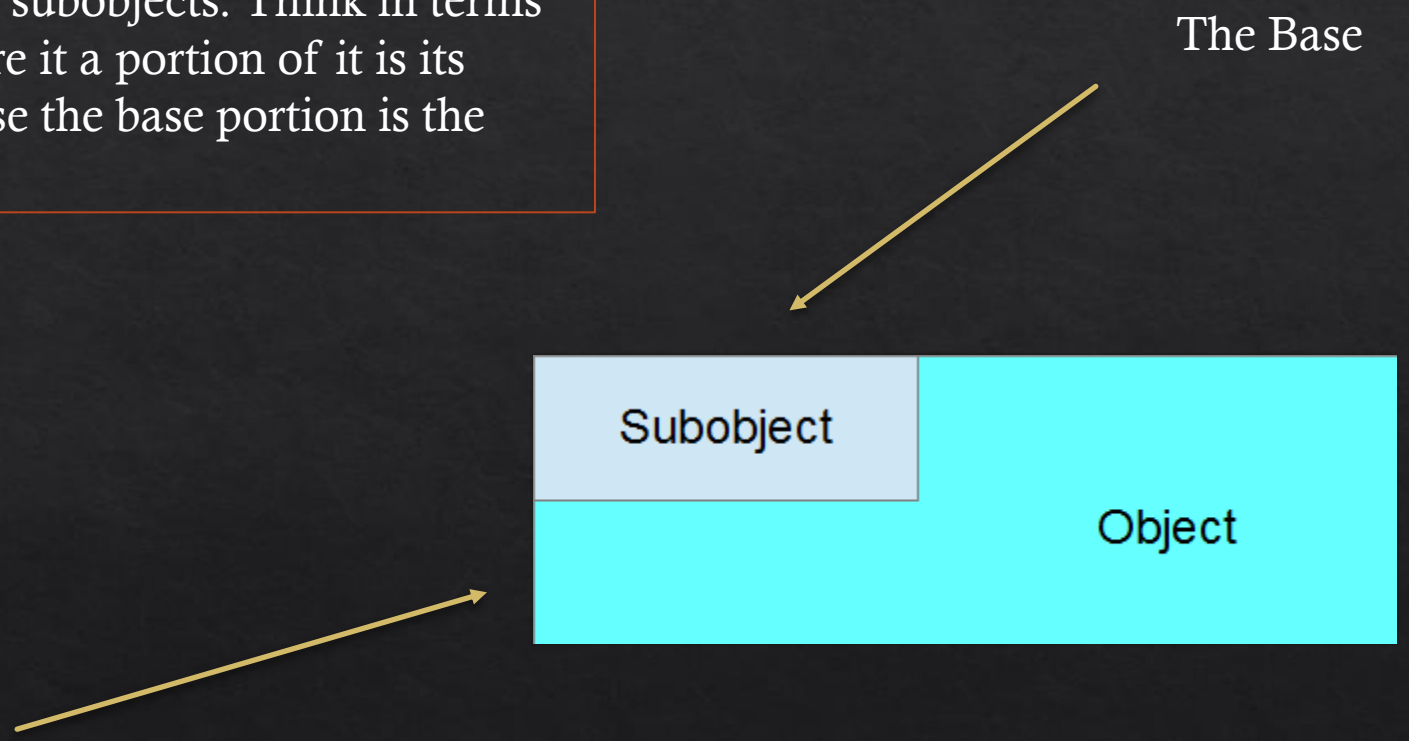
```
static_assert(bool condition, const char* message)
```

If this fails then…

This message gets printed out

Subobjects

C++ has the notion of subobjects. Think in terms of a derived class where it a portion of it is its parent class. In this case the base portion is the subobject



Lifetime in Memory

- There are a variety of life times that can be applied to objects in our programs depending on how they were defined / allocated
- Some of these you're likely familiar with:
  - Automatic – Lasts from its declaration till the end of the scope – no keywords used (default)
  - Dynamic – Created using the keyword new – lasts until deallocated using the delete keyword
  - Thread – lasts the life time of thread – uses keyword thread_local, we'll visit threads later in the semester
  - Static – lasts the entire lifetime of the program – use keyword static
- When speaking of life time with respect to subobjects they will share the same lifetime as their complete object

Storage Duration (Static)

```
#include <iostream>

void foo() {
   static int a = 1;
   std::cout << a++ << std::endl;
} // a doesn't go out of scope here
int main(){

  foo();
  foo();
  foo();
} // a will persist in memory until the end of the program
```