extern - entity declared in another scope

static – entity limited to this scope / global value for all objects of a class

**int main(int argv, char\* argc[])** – number of args – array of pointers to char\* strings of arg values

constexpr – value to be evaluated at compile time

**static\_assert(bool condition, const char\* message);** - generates custom compile message if condition is false;

Unlike pointers, references always point to the same object, cannot be null, and are accessed just like the name of the object would be.

Lvalue - & - std::ref(); - something that has a identity, cannot be moved

Rvalue - && - std::move(); - something with no identity, we can move

**Typename (const Typename&);** - copy constructor

**Typename& =operator(const Typename&);** - copy operator

**Typename (Typename&&);** - move constructor

**Typename& =operator(Typename&&);** - move operator

**for(int& e : a)** – a is a array, range based for

**enum [class] Typename { 1, 2, 3 }** class enums are scoped, and preferable to unscoped

**template<Param List>  
returntype name () {}** - function template **Class name() {} –** class template

**template<>  
returntype name<type>(){} -** specialization to sets specific template to be used for a specific type   
eg. Different code for char\* than ints, floats, doubles ect.

**Composition**  
- “has a” relationship, complete ownership, require subtype to be complete  
-updates to component don’t change composer class  
-manages objects it creates

**Aggregation  
-** “uses”  
- doesn’t require subtype to be complete  
- doesn’t manager sub object

**Associations  
-** types are independent

A union is a struct where all members are allocated at the same memory address

enum ProductId {sku, upc};  
Struct Item{  
 ProductId id;  
 union { // A item can have either have a sku or a upc  
 int sku;  
 char upc[13];

Item i;

i.id = sku;  
i.code = 2349;

i.id = upc;  
i.upc = “2178392173981921378”

}code;  
}

try {  
 // code that might generate exceptions  
 } catch (Type identifier) {  
// handler code for a specific type of exception  
 } catch (Type identifier) {  
// handler code for a specific type of exception  
 } catch (...) {  
 // handler code for all other types of exception  
}

The standard C++ libraries include a library of exception classes. The base class for the exception hierarchy is called exception and is defined in the <exception> header file. Classes derived from this base class include:

logic\_error - handles problems in a program's internal logic, which in theory are preventable. The following classes are derived from logic\_error:  
length\_error  
domain\_error  
out\_of\_range  
invalid\_argument

runtime\_error - handles problems that can only be caught during execution. The following classes are derived from runtime\_error:  
range\_error  
overflow\_error  
underflow\_error  
bad\_alloc - handles the allocation exception thrown by new. This class needs the <new> header file  
bad\_cast - handles the exception thrown by dynamic\_cast. This class needs the <typeinfo> header file

T (\*ptrToFunction)(T&);   
T\* function(T&);  
**Change a function pointer:**  
T (\*ptrToFunction)(T&) = fn;   
// ...  
ptrToFunction = gn;

**The definition of an array of pointers to functions takes the form:**  
 return-type (\*identifier[n])(parameter-type-list) = { initialization-list };

A function object that is called only once can be replaced by a lambda expression. A lambda expression is an anonymous function nested within the body of another function. It represents an unnamed function object that can capture variables within the scope of its caller.

[capture-list](parameter-declaration-clause)-> optional-return-type {  
// function body  
}  
capture-list is an optional comma separated list of the capture specifications for the non-local variables accessed by the function body.

[=] denotes capture by value.   
[&] denotes capture by reference.