***C++***

1. The evolution of c++ from c
2. Support all c syntax
3. Support of reference types
4. Support OOP (classes)
5. access-controllers (encapsulation)
6. Inheritance (substitution/polymorphism)
7. Abstraction (virtual methods = abstract class/interface in java-terms)
8. Support templates (Generics)
9. Support inline methods
10. Support namespaces
11. Support of unique, shared and other types of pointer management
12. The rule of 3

Every class in c++ needs to implement those three members (in addition to the default constructor):

1. Destructor – responsible for freeing the memory of the fields to avoid memory leaks.
2. Copy constructor – responsible to create a deep copy instance out of another instance of the class.
3. Copy assignment operator – responsible to deep copy the right-hand instance into the left-hand instance (and return the curr instance).
4. The rule of 5

On top of the members of the "rule of 3", every class need to implement those two members:

1. Move constructor – responsible for creating a new instance out of the rvalue instance that is received by the ctr.
2. Move assignment operator – responsible for moving the content of the right-hand instance to the left-hand instance (current).

These are an improvement over the copy ctr and assignment operator, since there is no need to deep copy all its members (because we are moving an rvalue instance that will not be referenced).

1. Templates use guidelines
2. There were problems with template use in the past when used with type that might throw exceptions.

In these cases, when we want to copy construct or copy assign from another instance, we risk exceptions and (if the problematic type does not deconstruct itself when exception occurs), in turn, memory leaks.

1. Surrounding the problematic code with try/catch is an ugly solution.
2. The solution is to wrap the field of the generic type in a unique\_ptr in order to not worry about deconstructing when exception occurs.

* It is helpful in situation of copy constructor – even if exception is thrown, the memory is safe, because the unique\_ptr will clean it up for us.
* It is helpful in copy assignment as well – instead of assigning directly from the other instance, we can use the copy constructor to create a new instance. If exception occurs, no harm from a memory management POV. If not, we can call the "swap" method that is swapping all the fields between the temporary instance we've created and our instance (swap is exception safe since it swaps unique\_ptr instances and not the problematic type).
* In the move constructor and move assignment operator we also use the unique\_ptr functionality. Specifically, we use:

1. The release method which sets the other instance's field to nullptr and our instance will point to its data in its stead.
2. The reset method which deletes the unique\_ptr's content and sets new content to it if provided.
3. New topic – on top of including the .h file in the .cpp, there is a need to include the .cpp file in the header file in a generic class.

In order to avoid recursive inclusion we can use "include guards" that will envelope the files' content.

The inclusion of the .cpp file in the header file should appear after the implementation. While the inclusion of the .h in the source file should come before the implementation.

1. Sometimes we would like to optimize/specialize a template for a specific type. C++ supports specialized template class/function.

What required is to declare

"template<> class CLASS\_NAME<SPECILIZED\_TYPE>{}" or

"template<> ret\_type FUNC\_NAME<>(){}"

(with empty trianglular brackets). Replace the generic type with the specialized type.

In principle, the compiler will prefer to call the specialized version.

1. Variadic functions are functions that can accept a varying number of arguments.
2. Containers
3. Sequential – these are containers that store the elements in a linear order (not necessarily in one block), such are – vector, list etc.
4. Associative – these containers implement sorted data (by an associated key) structures that can be quickly searched (*O(log n)* complexity), such are set, map etc.

* Unordered Associative containers implement unsorted (hashed) data structures that can be quickly searched (*O(1)* average, *O(n)* worst-case complexity), such are unordered\_map, unordered\_set etc.

1. Adaptors – these are containers that utilize other containers to support different functionality. Such as stack (uses vector).
2. Functors

Functor is short for function-operator – defined by a struct or a class that is implementing the operator(). By that we can create multiple instances of the class and pass it as an argument for a C++ Algorithm. For example, Sort Algorithm require a criterion by which to sort the data (the default is the operator <), so we can pass some different criteria.

The previous example was a stateless functor since each element is independent of its predecessors. In the for\_each algorithm we can pass an argument that is applied to each of the elements in a range/container. We can use a stateful functor to sum a specific field of each element.

The uses are endless.

Functors are awesome!!

1. Iterator Adaptors

We sometimes want to use an iterator that functions differently than the standard one. For example, we need the iterator to insert the elements at the end of the vector instead of replacing them.

We might want to use back\_inserter that produces back\_inserter\_iterator that in turn implements all the iterator functionality, but instead of the regular assignment operator (overwrite the element at the curr index), it calls the push\_back method of the vector (add the element at the end).

This is not actually an iterator since it does not iterate over the items of the container, but it is still very useful tool for algorithms that expect iterators.

Back\_inserter, front-inserter, inserter…