C++11

What are *closures*? What are *lambdas*? Lambda = anonymous local function

Closure = lambda + reference to the environment (a table storing a reference to each of the non-local captured variables)

What is a *move assignment* operator? When to use it? Tied to the concept of rvalue-reference (or xvalue)

The move assignment operator is called whenever it is selected by overload resolution, e.g. when an object appears on the left side of an assignment expression, where the right-hand side is an rvalue of the same or implicitly convertible type.

Move assignment operators typically "steal" the resources held by the argument (e.g. pointers to dynamically-allocated objects, file descriptors, TCP sockets, I/O streams, running threads, etc), rather than make copies of them, and leave the argument in some valid but otherwise indeterminate state. For example, move-assigning from a std::string or from a std::vector leaves the right-hand side argument empty.

More http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2027.html#Move_Semantics

std::move() Obtains an rvalue reference to its argument and converts it to an xvalue.

Code that receives such an xvalue has the opportunity to optimize away unnecessary overhead by moving data out of the argument, leaving it in a valid but unspecified state.

Basically it's just a cast:

```
template<typename _Tp>
constexpr typename std::remove_reference<_Tp>::type&& move(_Tp&& __t) noexcept
{
    return static_cast<typename std::remove_reference<_Tp>::type&&>(__t);
}
```

What are the *atomic types* and how would you use them? It allows lockless concurrent programming.

Each atomic operation is indivisible with regards to any other atomic operation that involves the same object. Atomic objects are the only C++ objects free of data races; that is, if one thread writes to an atomic while another thread reads from it, the behavior is well-defined.

STL

Container	Implementation	Insert	Remove	Index	Find
vector	dynamic array	O(n)	O(n)	O(1)	O(log n)
list	double link list	O(1)	O(1)	-	O(n)
map	red-black b tree	O(log n)	O(log n)	O(1)	O(log n)
hashmap	hash table	O(1)	O(1)	O(1)	O(1)

containers speed

C++ syntax notes

***static* - all the different meanings?**

1. at *file scope*: signifies "internal-linkage" i.e. not shared between translation units
2. at *function scope*: variable retains value between function calls
3. at *class scope*: signifies independence of class instance

const* and *mutable Const member function doesn't alter the data it operates on; except the one marked as *mutable*

default *copy constructor* and *assignment operator* - when to override? Rule of 3 in C++03 / Rule of 5 in C++11 When the class needs to be copy/move assignable - that is when it needs to be *cloneable* i.e. has non-shareable data

***volatile* keyword** Depends on language and compiler. Usually marks *atomicity* for data (but not guarantees it): reads from threads are guaranteed to have latest; marks that variable can be modified "externally"

***restrict* keyword** http://stackoverflow.com/questions/776283/what-does-the-restrict-keyword-mean-in-c

Optimisation hint to limit pointer aliasing and aid caching - it means a particular data is accessed only thru that pointer thus making optimisations like storing the ptr value in a registry for subsequent access

***in place* new** Allows to explicitly specify the memory management of individual objects, i.e. their “placement” in memory.

new (expression) [(arguments)]; for example:

```
char buffer[] = new char[256];
string *str = new (buffer) string("Hello world");
```

there is no placement delete syntax (but both *new* and *delete* functions can be overrided to specify the in-place)

***typename* keyword**

1. alias for the *class* keyword when declaring template parameters
2. a method to indicate that a dependent name is a type

If the compiler can’t tell if a dependent name (one that contains a template parameter) is a value or a type, then it will assume that it is a value.

```
template <typename T>
void foo() {
   T::bar * p; // won't compile, because without the typename prefix it will be interpreted
}

struct Gotcha {
   typedef int bar;
};

foo<Gotcha>();
```

C++ virtual

What is a *virtual* function? A virtual function allows derived classes to replace the implementation provided by the base class.

When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class’s version of the function. Virtual functions ensure that the correct function is called for an object, regardless of the expression used to make the function call

***pure* virtual function** A virtual function that is required to be implemented by a derived class.

Classes containing pure virtual methods are termed “abstract”; they cannot be instantiated directly.

virtual *destructor* - when/why? At the root of a class hierarchy to insure proper cleanup

virtual call in *assembly*

```
mov eax, dword ptr [this]
mov edx, dword ptr [eax]
mov eax, dword ptr [edx+4]
mov ecx, dword ptr [this]
call eax
```