C++ Interview Preparation Questions Bible

Template (also auto deduction rules)

Passing type	template <typename T> void func(T value) Type of T, type of value	template <typename T> void func(T & value) Type of T, type of value	template <typename T> void func(T && value) Type of T, type of value
int	Int , int	Int , int &	Int &, int &
int &	Int , int	Int , int &	Int &, int &
int &&	Int , int	Int, int &	Int &, int &
‘6’	Int , int	<not valid>	Int, int &&

Questions

1. **What is the purpose of the `cin` and `cout` objects in C++?**

cin	**Standard input stream**
cout	**Std::ostream . Prints to standard output stream. buffered**
cerr	**Prints to std output stream. Not buffered. Generally for error messages**
clog	**Prints to standard output stream. Buffered. Generally for logging (what ever you think it means)**

2. **How do you redirect cout (or cerr or clog) to out.txt**

```
int main(){

    std::ofstream out("C:\\temp\\out.txt");
    std::ifstream in("C:\\temp\\in.txt");
    auto cin_orig_buffer = std::cin.rdbuf();
    std::cin.rdbuf(in.rdbuf());
    auto cout_orig_buffer = std::cout.rdbuf();
    std::cout.rdbuf(out.rdbuf());
    std::string line;
    while (std::getline(std::cin,line))
    {
        std::cout<<"OUPUT:"<<line<<std::endl;
    }
    std::cin.rdbuf(cin_orig_buffer);
    std::cout.rdbuf(cout_orig_buffer);
    std::cout<<"Bye"<<std::endl;
}
```

3. **Explain the difference between declaration and definition and initialization in C++.**
    a. **Declaration - this is a thing and it exists somewhere**
    b. **Definition - this is the thing and make memory for it**
    c. **Initialization - this is the initial value of it**

4. **Differentiate between auto and decltype in C++.**
    a. auto' lets you declare a variable with a particular type whereas decltype **lets you extract the type from the variable so decltype is sort of an operator that evaluates the type of passed expression**

5. **What is the difference between using auto and decltype(auto) as the return type value?**
    a. **auto returns what value-type would be deduced if you assigned the `return` clause to an `auto` variable. `decltype(auto)` returns what type you would get if you wrapped the return clause in `decltype`.**
    b. **auto returns by value, `decltype` maybe not.**

```
auto foo()
{
    int x = 0 ;
    return (x) ; // fine
}

decltype(auto) bar()
{
    int x = 0 ;
    return (x) ; // trouble
}
expression auto   decltype(auto)
-----------------------------------------
10          int   int
x           int   int
(x)         int   int&
f()         int   int&&
```

6. What are the differences between break and continue statements?
   a. Break - go out of the loop
   b. Continue - skip the current iteration

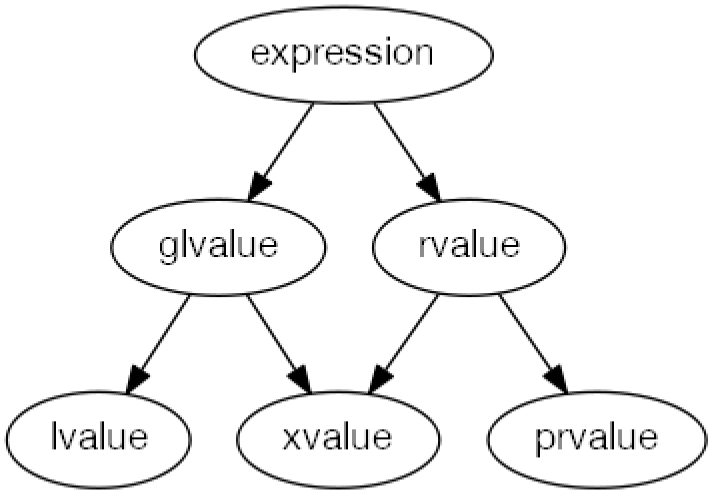
7. Explain the C++ name mangling and its significance.
   a. To solve case of same name being given to multiple identifies (e.g function overloading)
   b. Done by compiler
   c. No standardization across compilers. Arguably this is good because incompatibility is found at linker stage
   d. Tool to demangle - C++ filt

```
#include <stdio.h>
#include <stdlib.h>
#include <cxxabi.h>

int main() {
      const char *mangled_name =
"_ZNK3MapI10StringName3RefI8GDScriptE10ComparatorIS0_E16D
efaultAllocatorE3hasERKS0_";
      int status = -1;
      char *demangled_name =
abi::__cxa_demangle(mangled_name, NULL, NULL, &status);
      printf("Demangled: %s\n", demangled_name);
      free(demangled_name);
      return 0;
}
```

8. What is the role of the volatile keyword in C++?
   a. It is for variables that have access to memory mapped i/o
   b. So compiler shouldnt do any optimization on them
   c. They are not thread safe
   d. In C++11 , they are only meant for hardware access

9. **Difference between references and pointers?**
    a. **References cant reassigned. Pointers can (reseating)**
    b. **Pointers can be assigned to nullptr.**
    c. **You cant take address of a reference (it gives you the address of the referent)**
    d. **Reference is an alias for an object**

10. **What kinds of smart points do exist?**
    a. Unique, shared and weak
11. **what are rvalue and lvalue and glvalue & prvalue & xvalue?**

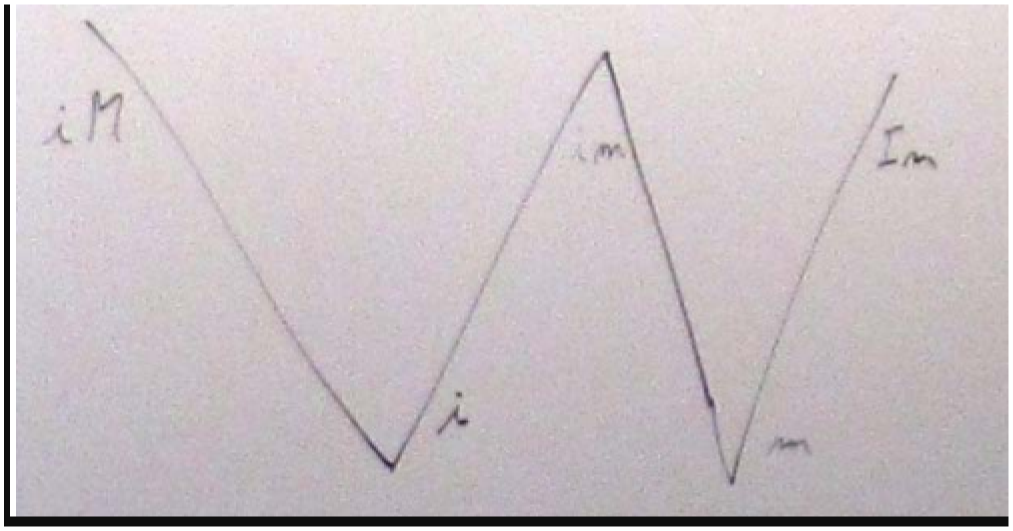


The above classification is based on two properties
- **Identity**
- **Movable**

Lvalues - have identity and cannot be moved from
Glvalue - has identity
Xvalue - has identity and can be moved from
Rvalue - can be moved from
Prvalue - does not have identity , can be moved from

it is safe to move something:

1. **When it's a temporary or subobject thereof. (prvalue)**
2. **When the user has explicitly said to move it (using std::move or std::forward)**



iM	**lvalue**
i	**glvalue**
im	**Xvalue** **a function call or an overloaded operator expression, whose return type is rvalue reference to object, such as std::move(x);**
m	**rvalue**
Im	**Prvalue (literals etc)**

- **named rvalue references are lvalues:**
- **std::move(x) is a xvalue**
- **Const lvalue ref can be assigned to a rvalue re**

12. What are std::move and std::forward()

a. **Both `std::forward` and `std::move` are nothing but casts.**
b. **Std::move - cast its operator to rvalue references**
c. **Std::forward - casts its operator to universal references**
d. **Std::forward preservers the cv and ref properties of the**

```
// std::move
template <typename T>
```

```
typename remove_reference<T>::type&& move(T&& arg) noexcept {
    return static_cast<typename
remove_reference<T>::type&&>(arg);
}

// std::forward
template <typename T>
T&& forward(typename remove_reference<T>::type& arg) noexcept
{
    return static_cast<T&&>(arg);
}

template <typename T>
T&& forward(typename remove_reference<T>::type&& arg) noexcept
{
    return static_cast<T&&>(arg);
}
```

Explanation from Howard Hinnant

`X x; std::move(x);`

The above casts the lvalue expression x of type X to an rvalue expression of type X (an xvalue to be exact). `move` can also accept an rvalue:

`std::move(make_X());`

and in this case it is an identity function: takes an rvalue of type X and returns an rvalue of type X.

With `std::forward` you can select the destination to some extent:

`X x; std::forward<Y>(x);`

Casts the lvalue expression x of type X to an expression of type Y. There are constraints on what Y can be.

Y can be an accessible Base of X, or a reference to a Base of X. Y can be X, or a reference to X. One can not cast away cv-qualifiers with `forward`, but one can add cv-qualifiers. Y can not be a type that is merely convertible from X, except via an accessible Base conversion.

If Y is an lvalue reference, the result will be an lvalue expression. If Y is not an lvalue reference, the result will be an rvalue (xvalue to be precise) expression.

`forward` can take an rvalue argument only if Y is not an lvalue reference. That is, you can not cast an rvalue to lvalue. This is for safety

reasons as doing so commonly leads to dangling references. But casting an rvalue to rvalue is ok and allowed.

If you attempt to specify Y to something that is not allowed, the error will be caught at compile time, not run time.

13. Ways to access private fields of some class?

a. **Not in any way other than from a friend class or function**
b. **Hack**

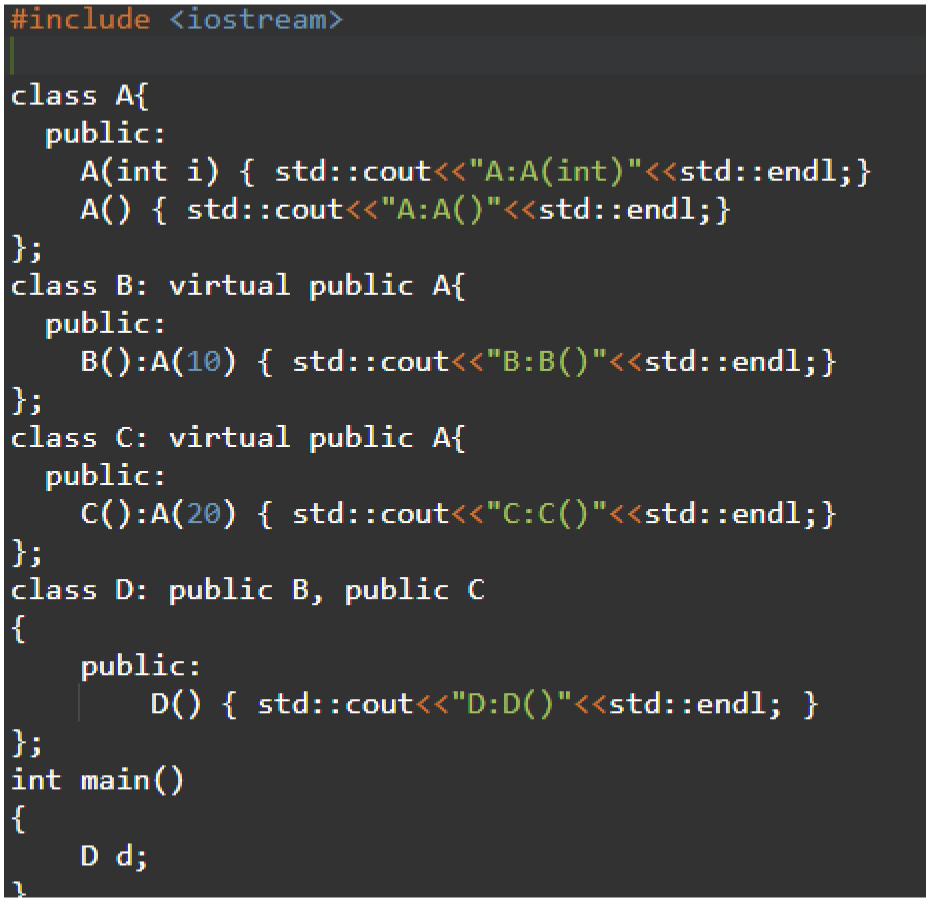
```
class A {
     int iData;
     public:
     A(int x):iData(x){}
     void print() { std::cout<<iData<<std::endl;}
};

int main ()
{
    A a(10);
    a.print();
    struct ATwin { int pubData; }; // define a twin class with public
members
    reinterpret_cast<ATwin*>( &a )->pubData = 42; // set or get value
    a.print();
    return 0;
}
```

14. What is ABI?

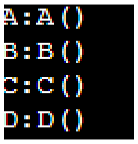
- **ABI stands for Application binary interfaces. Whether applications are binary compatible**
- **ABI covers topics like**
    - **The calling conventions: how functions are translate to assembly code**
    - **Name mangling**
    - **Datatypes, alignment, size etc**

15. Can a class inherit multiple classes?

a. **Yes. Major issue is the diamond problem**
b. **How to solve diamond problem? By using virtual inheritances from both B & C**
c. **virtual keyword causes to call the only default constructor of the base class. Notice A:A() is called, not A:A(). If there is no default constructor for A, the program wont compile**

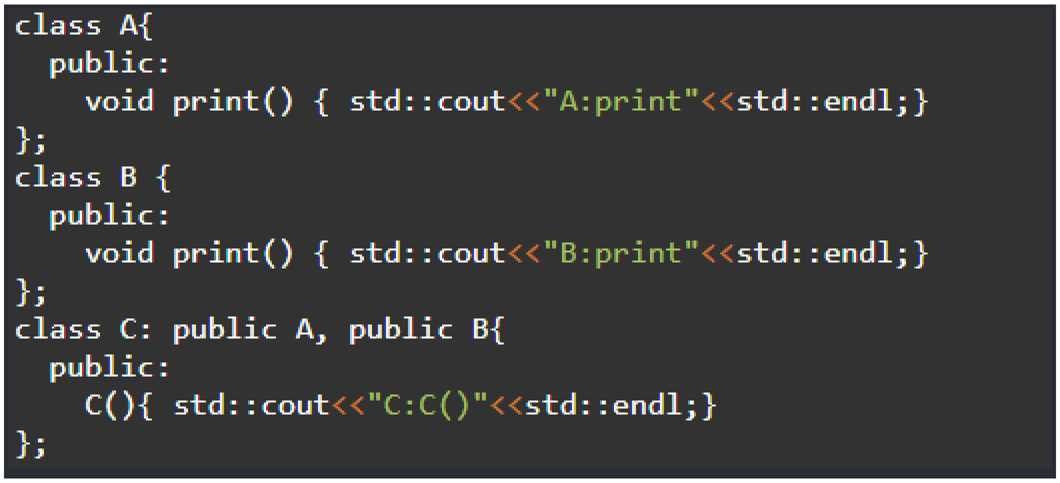


```
#include <iostream>

class A{
  public:
    A(int i) { std::cout<<"A:A(int)"<<std::endl;}
    A() { std::cout<<"A:A()"<<std::endl;}
};
class B: virtual public A{
  public:
    B():A(10) { std::cout<<"B:B()"<<std::endl;}
};
class C: virtual public A{
  public:
    C():A(20) { std::cout<<"C:C()"<<std::endl;}
};
class D: public B, public C
{
    public:
        D() { std::cout<<"D:D()"<<std::endl; }
};
int main()
{
    D d;
}
```



```
A:A()
B:B()
C:C()
D:D()
```

d. **Another problem: ambiguity . Solve by specifying class explicitly**



```
class A{
  public:
    void print() { std::cout<<"A:print"<<std::endl;}
};
class B {
  public:
    void print() { std::cout<<"B:print"<<std::endl;}
};
class C: public A, public B{
  public:
    C(){ std::cout<<"C:C()"<<std::endl;}
};
```

16. Is static field initialized in class constructor?
17. Can an exception be thrown in constructor / destructor? How to prevent that?
18. What are virtual methods?
19. Why do we need virtual destructor?
20. Difference between abstract class and interface?
21. Can be constructor virtual?
    a. no
22. How keyword const is used for class methods?

23. How to protect object from copying?
    a. Delete copy ctor, operator=

24. What variables are better to copy by value and by reference?
25. Define the four pillars of OOP and explain them.
    a. Abstraction
    b. Encapsulation
    c. Inheritance
    d. polymorphism
26. Explain the concept of inheritance in C++.
27. What is encapsulation in C++ and why is it important?
28. Describe the concept of polymorphism and provide an example.
29. What is an abstract class, and how is it different from an interface?
30. What is operator overloading in C++?
31. Explain the use of constructors and destructors in C++.
32. Define inheritance in C++.
33. Explain single inheritance and multiple inheritance.
34. What is the base class and derived class in inheritance?
35. How is function overriding used in polymorphism?
36. Describe virtual functions and their purpose.
37. What is a pure virtual function, and how is it implemented?
38. What are the different types of inheritances

39. What is the C++ Standard Library, and what does it include?
40. What is the difference between `#include <iostream>` and `#include "iostream"`?
41. What is a C++ header file, and how is it different from a source file?
42. Explain the use of the `vector` container in the Standard Library.
43. What is an iterator in C++ and why is it useful?
44. Describe the purpose of the `string` class in the C++ Standard Library.
45. How can you read and write files in C++ using the Standard Library?
46. What is the purpose of the C++ Standard Template Library (STL)?
47. What are smart pointers in C++ and why are they used?
    a. They are pointers that clean up memory
    b. Avoid memory leak
48. Explain the concept of RAII (Resource Acquisition Is Initialization).
    a. Idiom where lifetime of resource acquire== life time of object
    b. The resource is release when the objects life time ends
    c. The resources are released in the reverse order of acquiral
49. What is the purpose of the C++ STL algorithms library?
50. Describe the differences between `std::for_each`, `std::transform`, and `std::accumulate`.
51. How can you sort elements in a container using the `std::sort` algorithm?
52. What is the difference between `std::set` and `std::unordered_set`?
53. Explain the use of the `std::map` and `std::unordered_map` containers.
54. How does the `std::pair` class work in C++ STL?
55. What is a binary search tree, and how can you implement it in C++?
56. How do you use the `std::queue` and `std::stack` containers?
57. What is a heap data structure, and how is it used in C++?
58. Discuss the purpose of the `std::priority_queue` container.
59. What are templates in C++?
60. How is function template specialization used?
61. Discuss the concept of class templates.
62. Explain the Standard Template Library (STL) in C++.
63. What are some common STL containers, and when would you use them?
64. How can you use iterators in the STL?
65. How do you use `std::vector` and `std::array` in C++?
66. Explain the purpose of `std::map` and `std::set`.
67. Describe some common algorithms provided by the STL.
68. What is the difference between `std::sort` and `std::stable_sort`?
69. How to rotate using list::splice
70. Time complexities and underlying containers of different containers
71. Difference between vector and list?

72. Difference between map and unordered map?
- Red black tree vs hash (O(logn) vs O(1)

- Need to provide a hash for unordered map for UDT

73. When calling push_back() make iterator in vector invalid?
   a. Yes - think reallocation.

74. How to modify your class to use with map and unordered_map?
   a. You need a hash function for the UDT
   b. The UDT needs to implement operator==
   c. For map, if the key is a UDT, you should implement < (or less)

75. Are stl containers thread safe?
76. what are predicates and functors?
77. How do you use splice to rotate a list?
78. What is the difference between emplace_back & push_back
79. What is shrink_to fit and what is the swap trick
80. What is dynamic memory allocation in C++?
81. What is the difference between `new` and `malloc` for memory allocation?
82. Explain the role of the `delete` operator in C++.
83. What are memory leaks, and how can they be avoided in C++?
84. What is a pointer in C++? How is it different from a reference?
85. Discuss the concept of null pointers and the use of `nullptr`.
86. What is the difference between stack and heap memory in C++?
87. How does C++ handle array index boundaries and boundary checking?
88. Explain the concept of pointer arithmetic and provide an example.
89. What are function pointers, and how are they used in C++?
90. What is dynamic memory allocation in C++?
91. Describe new and delete operators.
92. Explain the difference between the stack and heap memory.
93. How do you prevent memory leaks in C++?
94. What are the different casting operations ? When to cast and not to cast
    a. https://stackoverflow.com/questions/332030/when-should-static-cast-dynamic-cast-const-cast-and-reinterpret-cast-be-used
95. Are smart pointers thread safe?
96. What are C++ templates, and why are they useful?
97. Explain the difference between function templates and class templates.
98. What is template specialization in C++?
99. Discuss the concept of template metaprogramming in C++.
100. How do you define and use a generic function in C++?
101. What is the Standard Template Library (STL) in C++?
102. Explain the use of the `STL containers` and provide examples.
103. What are iterators in the context of the C++ STL?
104. How are algorithms implemented in the C++ STL?
105. Discuss the importance of concepts and constraints in C++20.

106. How unique_ptr is implemented? How do we force that only one owner of the object exist in - unique_ptr ?
107. How does shared_ptr work? How reference counter is synchronized between objects?
108. Can we copy unique_ptr or pass it from one object to another?
109. Are smart pointers thread safe?
110. What are the different template specializations? For class and functions.
111. How many kinds of template parameters exist and what are they?
112. What is partial specialization? What about full specialization?
113. What are the main disadvantages of using templates?
114. What category of types can be used for non-type template parameters?
115. Where are default template arguments not allowed?
116. What is explicit instantiation declaration and how does it differ syntactically from explicit instantiation definition?
117. What is an alias template?
118. What are template lambdas?
119. What is multithreading in C++?
120. Explain the difference between processes and threads.
121. How can you create and manage threads in C++?
122. What is a mutex, and why is it used in multithreading?
123. Describe the concept of thread safety in C++.
124. What is the purpose of atomic operations in C++?
125. How can you handle thread synchronization using condition variables?
126. Discuss the C++ Standard Library's support for multithreading.
127. What is a race condition, and how can it be prevented in C++?
128. Explain the concept of thread-local storage (TLS).
129. Why is std_ref needed while calling thread functions?
130. What is deadlock and livelock?
131. What is a spinlock
132. Write pseudocode for compare_exchange_weak()
133. What is a hazard pointer?

134. What is Amdahls law
   a. Efficiency improvement from multithread
   b. fs - fraction done serially, N - number of concurrent threads that could be run
   c. (1/(fs+(1-fs)/N)

135. What is thread::yield
136. What is multithreading in C++?
137. Describe the `std::thread` class.
138. How do you prevent data races in multithreaded programs?
139. What is a mutex, and how is it used in C++?
140. What are the different memory ordering types?

141. What are fences (or memory barriers?
142. Why do we need hierarchical mutexes?
143. What is wrong with double checked locking pattern?
144. What are the different memory ordering types?
145. What are fences (or memory barriers?
146. Difference between processes and threads?
147. Can the same thread be ran twice?
148. Ways to synchronize threads?
149. What is deadlock?
150. What are lock free algorithms?
151. What are std::ref and std::mem_fn for?

152. What are the different policies for std::task
153. What is exception handling in C++?
154. Describe the difference between `try`, `catch`, and `throw`.
155. How can you define custom exceptions in C++?
156. Explain the purpose of the `std::exception` class.
157. What is stack unwinding in the context of exception handling?
158. Discuss the use of RAII with exception safety in C++.
159. How are exceptions and error codes different in C++ error handling?
160. What are the best practices for handling exceptions in C++?
161. How do you re-throw an exception in C++?
162. Explain the `noexcept` specifier in C++.
    a. Performance low if noexcept not specified
    b. Push_back exception guarantee means push_back(rvalue) if no except will end up in copy
163. Explain the difference between standard exceptions and user-defined exceptions.
164. How do you handle multiple exceptions in C++?
165. Why is it important to specify no exceptions
166. Exceptions from constructors and destructors
167. What is the purpose of lambda expressions in C++?
168. What are closure classes and objects
169. Explain the use of C++ move semantics and rvalue references.
170. What are variadic templates in C++ and how do they work?
171. Describe the purpose of the `constexpr` keyword in C++.
172. What is perfect forwarding in C++ and when is it useful?
173. How can you implement a custom allocator in C++?
174. Discuss the concept of memory-mapped files in C++.
175. What are user-defined literals in C++?
176. How can you use C++17's structured bindings?
177. Explain the use of the `std::optional` and `std::variant` types.
178. Explain some new features introduced in C++11.
179. What is the purpose of lambda expressions in C++11?
180. Describe the `auto` keyword in C++11.
181. What are smart pointers, and how do they differ from raw pointers?
182. Discuss features introduced in C++14, C++17, and C++20.
183. What is RAII (Resource Acquisition Is Initialization)?
184. How does the C++ type system handle inheritance?
185. Explain the Pimpl (Pointer to Implementation) idiom.
186. Discuss the Rule of Three and the Rule of Five in C++.
187. What is SFINAE (Substitution Failure Is Not An Error)?
188. How do you debug C++ code? Name some debugging tools.
189. Describe the concept of profiling and optimization in C++.

190. What are some common optimization techniques in C++?
191. What are some best practices for writing efficient C++ code?
192. Explain the importance of code readability and maintainability.
193. How do you handle memory management in modern C++?
194. What are C++ functors?
195. Explain the purpose of C++ type traits.
196. Discuss the differences between deep copy and shallow copy.
197. How do you implement multithreaded parallelism in C++?
198. What is Software transaction memory?
199. What is dining philosophers problem? How to solve it
200. What is Producer consumer problem? How to solve it
201. Why do you say std::move & std::forward are casting operations?
202. What are string literals in C++11
203. How do you use make_shared to make array of objects
204. What is temporary move idiom?
205. What is a xml parser to use?
206. Why prefer make_unique over uniquePtr(new)
207. Why do we say move & forward are casting operations?
208. What is an allocator in a container
209. When should std::move & std::forward be used
210. What is reference collapsing?
211. What is call once, once_flag
212. What is the Singleton design pattern, and when is it used?
213. Describe the Observer pattern and provide an example in C++.
214. What is the Factory Method pattern, and how is it implemented?
215. Discuss the Adapter design pattern in C++.
216. How can the Strategy pattern be applied in C++?
217. Explain the concept of the Decorator pattern.
218. What are the SOLID principles, and why are they important?
219. Explain the Singleton design pattern.
220. Discuss the Observer design pattern
221. What is the Factory Method design pattern?
222. Describe the Model-View-Controller (MVC) design pattern.
223. How do you open and close a file in C++?
224. Describe reading and writing to a file in C++.
225. What are binary and text files? How do you work with them?
226. Explain the concept of file streams in C++.
227. Explain pointers in C++. What are null pointers?
228. How do you use references in C++?
229. Discuss the concept of pointer arithmetic.
230. What is the difference between `const int*` and `int const*`?

231. What are the different types of iterators? (5)
232. How do you convert a const iterator tor a non const iterator
233. What is C++'s most vexing problem?
234. Explain move semantics in C++11 and how they differ from copy semantics.
235. Provide an example of a situation where move semantics are beneficial.
236. How is std::array different from other containers (clue: storage)
237. What are the threading issues with singleton? What is double locking problem? What is a monostate pattern?
238. What are the differences between C++ template specialization and overloading? When would you use one over the other?
239. Discuss the RAII (Resource Acquisition Is Initialization) idiom. Provide an example of how it can be used in C++ for resource management.
240. Explain the concept of the "Rule of Three" in C++. How does it relate to classes that manage dynamic resources, such as memory allocation?
241. What are lambda expressions in C++11 and later standards? How are they useful, and can you provide an example of their usage?
242. Describe the differences between std::shared_ptr, std::unique_ptr, and std::weak_ptr. When would you use each of these smart pointers, and what are the advantages of smart pointers over raw pointers?
243. What is SFINAE (Substitution Failure Is Not An Error) in C++ template metaprogramming? Provide an example of how it can be used to enable or disable template overloads.
244. What is template overloading and template specialization?
   a. Template overloading - template function with same name but different parameters (templated or otherwise)
   b. Template specialization - template function specific for a specific type
245. Explain C++17's std::variant and std::visit. How do they improve handling of variant types, and what are the advantages over traditional approaches using unions or inheritance hierarchies?
246. Discuss the use of C++20's coroutines. What problems do they solve, and can you provide an example of an application that benefits from coroutines?
247. Explain how the C++ Standard Library's move semantics and the Rvalue Reference (&&) concept work together to improve performance and resource management.
248. What is the "as-if" rule in C++? How does it impact compiler optimizations and the behavior of C++ code?
   a. The language specifies that the compiler can optimize the code as long as it doesnt change the observable behaviour of the code. (as-if rule)
   b. Exception - RVO/NRVO
249. Discuss the concept of perfect forwarding in C++. When is it useful, and how would you implement it in a function template?
   a. Std::forward takes arguments and pass it exactly as is to the calling function
   b. Used in calls like make_shared and make_unique
   c.
   d. {

```
                    Template <typename T, typename… Args>
                    unique_ptr<T> make_unique(Args&&... params)
                    {
                            Return unqiue_ptr<T>(std::forward<Args>(params)...))
                    }
            }
    .
```

250. What are expression templates in C++ and how are they used to optimize mathematical operations, such as matrix multiplication? Provide an example of implementing expression templates
251. Explain the purpose of type traits and how you would use them in generic programming to make code more flexible and efficient.
252. Discuss the differences between C++'s std::thread, std::async, and std::packaged_task in the context of multithreading and parallel programming.
253. What are the advantages and disadvantages of using C++ for systems programming compared to other languages like C or Rust?
254. Explain the concept of tag dispatch in C++. When is it useful, and can you provide an example of its application in a library or framework?
255. How does C++ handle memory management for objects with cyclic dependencies (e.g., objects linked through smart pointers)?
256. Discuss the role of the C++ Standard Template Library (STL) and its various components in modern C++ development. Give examples of situations where you would use containers, algorithms, or other STL components.
257. What are the new features and improvements introduced in the most recent C++ standard (as of your knowledge cutoff date), and how do they affect C++ development?
258. Explain the "Rule of Three" in C++ and why it's important. How has it evolved in modern C++ standards?
259. What are rvalue references and move semantics in C++11? How do they improve the efficiency of C++ programs?
260. What is the RAII (Resource Acquisition Is Initialization) principle in C++? Provide examples of how you've used it in your code.
261. Describe the "Big Three" and "Big Five" in C++. What are their roles, and how do you implement them in your classes?
262. Explain the concept of template metaprogramming. Provide an example of a situation where you would use metaprogramming techniques.
263. What is SFINAE (Substitution Failure Is Not An Error) in C++? How is it used in template specialization and overloading?
264. Discuss the benefits and drawbacks of multiple inheritance in C++. When is it appropriate to use it, and how do you handle the Diamond Problem?
265. What is type erasure, and how can it be implemented in C++ to achieve polymorphism without templates?
266. Explain C++ smart pointers (e.g., std::shared_ptr, std::unique_ptr). When and why would you use each type, and what are the potential pitfalls?

267. Discuss the differences between the C++ Standard Library and the Boost C++ Libraries. Provide examples of Boost libraries that you've found particularly useful.
268. What are C++ functors, and how do they differ from regular function objects and lambdas? Can you provide an example of a functor in action?
269. Describe C++11/14/17/20 features that improve concurrency and parallelism. How do they facilitate multi-threaded programming?
270. How does C++ handle exception handling, and what are the best practices for using exceptions in your code?
271. What is the Pimpl (Pointer to Implementation) idiom in C++? When and why would you use it, and what are the advantages and disadvantages?
272. Explain the C++ Standard Library's memory model and how it relates to atomic operations and multi-threaded programming.
273. Discuss C++17's std::filesystem library. How does it simplify file system manipulation, and what are some of its key features?
274. What is the C++ Standard Library's <algorithm> header, and can you provide examples of how you've used STL algorithms in your code?
275. What is CRTP (Curiously Recurring Template Pattern), and how can it be applied in C++? Provide a practical example of where CRTP is useful.
276. Explain the C++20 concepts feature. How does it improve template programming and type safety?
277. Discuss the use of custom allocators in C++. When and why might you want to use a custom allocator, and how would you implement one?
278. Explain the purpose and usage of variadic templates in C++. Provide an example of a variadic template function.
279. How does the C++ memory model work, and what are the differences between atomic and non-atomic operations?
280. Discuss C++17's std::optional and how it can be used to represent possibly missing values.
281. Explain what the Pimpl (Pointer to Implementation) idiom is and why it's used in C++. Provide an example of its implementation.
282. Discuss C++20 features such as concepts and modules. How do these features improve C++ programming and code organization?
283. What are the benefits and drawbacks of using C++ in low-level and embedded programming compared to languages like C or assembly?
284. Explain how the C++ Standard Library deals with multithreading and concurrency, including features like std::thread and std::mutex.
285. Explain the concept of perfect forwarding and universal references in C++. When should you use std::forward and how does it help in maintaining type information during forwarding?
286. What are the C++ memory model and memory barriers? How do they relate to multithreading and concurrent programming?
287. C++ Standard Library Containers: Compare and contrast the use of std::vector, std::list, and std::deque in C++. In what situations would you choose one container over another?

288. C++ Best Practices: Discuss some best practices for writing efficient and maintainable C++ code. Topics can include resource management, exception safety, code organization, and performance considerations.
289. What are the benefits and drawbacks of using smart pointers like std::shared_ptr in a multi-threaded application? How can you ensure thread safety with shared pointers?
290. Discuss the concept of type erasure in C++ and how it's achieved using techniques like std::any, std::function, or custom implementations.

291. What is slicing ? How can you avoid it?

a. This happens when you pass a derived class to a function that takes base class

b. Or assign a derived class object to an base class reference

292. How is std::array different from other containers?

a. Created in stack

b. Size specified in declaration as template argument std::array<int,5>

293. What is special about vector<bool>
294. What is row major and column major ?
295. What is iota for?
296. What is std::search() for?
297. What are the difference memory ordering ? Illustrate the differences? Why is it important
298. What is memory fence?
299. What are the different synchronization mechanism?
300. Write pseudo code for compare_exchange_weak

301. Are atomic calls truly lock free? How can you know?

a. No they are not.

b. Atomic types have a method is_lock_free to check this

c. All atomic types except for std::atomic_flag may be implemented using mutexes or other locking operations, rather than using the lock-free atomic CPU instructions. Atomic types are also allowed to be sometimes lock-free, e.g. if only aligned memory accesses are naturally atomic on a given architecture, misaligned objects of the same type have to use locks.

302. What is copy elison?

a. Copy elision is a compiler optimization technique that eliminates unnecessary copying/moving of objects.

b. RVO & NRVO are techniques of copy-elison

c. NRVO - if a function returns a class type by value and the return statement's expression is the name of a non-volatile object with automatic storage duration (which isn't a function parameter), then the copy/move that would be performed by a non-optimising compiler can

be omitted. If so, the returned value is constructed directly in the storage to which the function's return value would otherwise be moved or copied.

d. **RVO - If the function returns a nameless temporary object that would be moved or copied into the destination by a naive compiler, the copy or move can be omitted.**

303. What are the exceptions to RVO & NRVO?

- **returning std::move(x)**
- **NRVO wont be invoked if you have multiple return points with named vars**
- **If the return argument is a function variable**

304. What are the 5 new major C++20 features?

- **Coroutines**
- **Concepts**
- **range**
- **fmt**
- **Import**
- **3 way operator ⇔**

305. What is osyncstream?

a. **In C++ 20**
b. **osyncstream(std::cout)<<"message "**
c. **To avoid interleaving messages between threads**
d. **Way to synchronize output messages**

306. What is reference collapsing?
307. What is needed when you create a structure to be stored in unordered set/map
308. Write code for make_unique (t31A, A231p,Ap3)

309. What is std::exchange()

```
T exchange( T& obj, U&& new_value )//returns old value
 - Can be used to implement move ctros and operators
 - struct S
 - {
 -     int n;
 -
 -     S(S&& other) noexcept : n{std::exchange(other.n,
   0)} {}
 -
 -     S& operator=(S&& other) noexcept
 -       {
 -           if (this != &other)
 -               n = std::exchange(other.n, 0); // Move n,
 while leaving zero in other.n
 -           return *this;
 -       }
 -   };
 -
```

310. **What does this method f() & and g() && do**
   a. **This ensures they are only called by lvalue (&) or rvalue (&&)**

```
class X{

  public:
    void f() &{std::cout<<"f()"<<std::endl;}
    void g() &&{ std::cout<<"g()"<<std::endl;}
};

int main()
{
    X x;
    x.f();
    x.g()//compile error
    X().f()//compile error
    X().g();
    return 0;
}
```

More Questions

1. Advanced Class Inheritance: Implement a complex inheritance hierarchy involving multiple base classes, virtual inheritance, and multiple levels of derived classes. Use polymorphism to demonstrate the dynamic behavior of these classes.
2. Thread Synchronization: Write a multi-threaded C++ program that simulates a scenario where multiple threads are accessing shared resources. Use

synchronization mechanisms such as mutexes or semaphores to prevent race conditions and demonstrate proper thread management.

3. Custom Data Structures: Implement a custom data structure, such as a red-black tree, AVL tree, or hash table, from scratch in C++. Provide methods for insertion, deletion, and searching within the data structure. Analyze and compare the time and space complexities of your implementation.
4. Lambda Expressions and STL: Create a program that uses lambda expressions with standard template library (STL) algorithms. For example, write a program that sorts a vector of custom objects using a custom sorting criterion defined in a lambda expression.
5. Template Metaprogramming: Write a C++ template metaprogram that performs a complex compile-time task, such as generating a specific type of data structure or performing compile-time computations. Explain how and where this technique can be useful.
6. Exception Handling: Create a program that demonstrates exception handling in C++. Design a custom exception class hierarchy and use it in conjunction with standard exceptions. Show how to gracefully handle exceptions and perform cleanup operations.
7. Memory Management: Develop a program that efficiently manages memory using techniques like custom memory allocators or smart pointers. Compare the performance and memory usage of your solution with standard memory management practices.
8. C++17/20 Features: Write code that utilizes the latest C++ features introduced in C++17 or C++20. This may include concepts, coroutines, ranges, or any other new language features. Explain the advantages of using these features.
9. Concurrency and Parallelism: Create a program that takes advantage of multi-core processors using C++ concurrency and parallelism features like `std::async`, `std::thread`, and parallel algorithms. Show how it improves performance.
10. Low-Level Programming: Write a program that involves low-level system programming, such as interacting with system calls, memory-mapped files, or assembly language inline within C++.

Programming Tasks

1. Implement a shared pointer (including make_shared identical function)
2. Implement a generic log function with SFINAE & enable_if
3. Impleate a factorial with TMP
4. Implement a circular Q

5. Implement a BFS and DFS graph
6. Implement Quick sort
7. Singleton using (a) CRTP (b) avoiding double locking problem
8. Implement producer consumer with (a) cv (b) thread safe q © using packaged task
9. Implement a Thread RAII class
10. Implement parallel version of std::accumulate with (a) threads (b) tasks
11. Implement a threadpool class
12. Write a variadic function to print using templates
13. Implement a thread safe stack
14. Implement hierarchical mutex
15. Read the contents of file using 3 methods
16. Implement spinlock
17. Implement a threadsafe lookup dictionary
18. Implement a Widget with a resource that is exception safe (temp move idiom)

Cppcon presentations

Template Metaprogramming

https://www.youtube.com/watch?v=Am2is2QCvxY

https://www.youtube.com/watch?v=_doRiQS4GS8&t=2097s

https://www.youtube.com/watch?v=tiAVWcjIF6o

C++ 20
https://www.youtube.com/watch?v=FRkJCvHWdwQ&list=RDCMUCMlGfpWw-RUdWX_JbLCukXg&index=3

STL
https://www.youtube.com/watch?v=2olsGf6JIkU&list=RDCMUCMlGfpWw-RUdWX_JbLCukXg&index=4

C++ 20 MT

https://www.youtube.com/watch?v=A7sVFJLJM-A&list=RDCMUCMlGfpWw-RUdWX_JbLCukXg&index=5

Concurrency

https://www.youtube.com/watch?v=F6Ipn7gCOsY&list=RDCMUCMlGfpWw-RUdWX_JbLCukXg&index=7

C++ bugs

https://www.youtube.com/watch?v=lkgszkPnV8g&list=RDCMUCMlGfpWw-RUdWX_JbLCukXg&index=8

Atomics

https://www.youtube.com/watch?v=ZQFzMfHIxng&list=RDCMUCMlGfpWw-RUdWX_JbLCukXg&index=9

Virtual Funcions

https://www.youtube.com/watch?v=n6PvvE_tEPk&list=RDCMUCMlGfpWw-RUdWX_JbLCukXg&index=11

Type deduction

https://www.youtube.com/watch?v=wQxj20X-tIU&list=RDCMUCMlGfpWw-RUdWX_JbLCukXg&index=12

Cute C++

https://www.youtube.com/watch?v=gOdcNko2xc8&list=RDCMUCMlGfpWw-RUdWX_JbLCukXg&index=13

Best parts of C++

https://www.youtube.com/watch?v=iz5Qx18H6lg&list=RDCMUCMlGfpWw-RUdWX_JbLCukXg&index=15

Lock free programming

https://www.youtube.com/watch?v=c1gO9aB9nbs&list=RDCMUCMlGfpWw-RUdWX_JbLCukXg&index=32

Smart pointers

https://www.youtube.com/watch?v=YokY6HzLkXs&list=RDCMUCMlGfpWw-RUdWX_JbLCukXg&index=29

bs - C++ 23

https://www.youtube.com/watch?v=u_ij0YNkFUs&list=RDCMUCMIGfpWw-RUdWX_JbLCukXg&index=37

RAII

https://www.youtube.com/watch?v=Rfu06XAhx90&list=RDCMUCMIGfpWw-RUdWX_JbLCukXg&index=39

Performance

https://www.youtube.com/watch?v=0iXRRCnurvo&list=RDCMUCMIGfpWw-RUdWX_JbLCukXg&index=35

C++ 23++

https://www.youtube.com/watch?v=fJvPBHErF2U

Performance

https://www.youtube.com/watch?v=NH1Tta7purM

Templates

https://www.youtube.com/watch?v=HqsEHG0QJXU

Multithreading

https://www.youtube.com/watch?v=F6Ipn7gCOsY&t=1s

Performance

https://www.youtube.com/watch?v=uzF4u9KgUWI&list=RDCMUCMIGfpWw-RUdWX_JbLCukXg&index=21

Move semantics

https://www.youtube.com/watch?v=St0MNEU5b0o

Cache Line

https://www.youtube.com/watch?v=WDIkqP4JbkE&t=3769s