Interview Preparation

Top 50 C++ Interview Questions and Answers

Preparing for the interview C++

by Ihor Gudzyk
C++ Developer

Apr, 2024 •
40 min read

Beginner Questions

1. What is C++ and why is it used?

C++ is a statically typed, compiled, general-purpose programming language that supports procedural, object-oriented, and generic programming features. It's widely used for developing complex systems where hardware level manipulation is needed, such as

operating systems, game engines, and enterprise software.

2. What are the differences between C and C++?

C is primarily a procedural language, focusing on function-based programming. In contrast, C++ supports both procedural and object-oriented programming, allowing for the use of classes and objects. Additionally, C++ includes features which are not present in C.

3. What are the primitive data types in C++?

Primitive data types are built-in or predefined data types and can be used directly by the user to declare variables.

Data Type	Description	Examples
Integer Types	Used to store whole numbers.	int, short, long, long long
Character Type	Used to store single characters.	char
Floating Point Types	Used for storing numbers with fractional parts.	float, double, long double
Boolean Type	Represents truth values.	bool

4. What is the ASCII table?

The ASCII (American Standard Code for Information Interchange) table is a standard character encoding scheme that defines the correspondence between characters and numerical values in the range from 0 to 127. Each character in the ASCII table has a corresponding numerical value representing it as a byte. This table standardizes the representation of text in computers and enables data exchange between different systems.

For example, the letter "A" has the numerical value 65, "B" has 66, and so on.

5. What is the preprocessor?

The preprocessor in C++ is a component of the compiler that performs preliminary processing of the source code before its actual compilation. It executes various tasks such as defining symbolic constants using `#define` directives, excluding parts of code using `#ifdef`, `#endif` directives, and more. The preprocessor helps to make the code more structured, extends its functionality, and provides greater flexibility in managing the software project.

6. What are header files in C++?

Header files in C++ contain declarations of functions, classes, variables, etc., intended for reuse across multiple source files. They're included at the beginning of source files using `#include`. Header files promote modularity, code reuse, and organization in large projects by separating interface declarations from implementation details.

7. How does the #include directive work in C++?

The `#include` directive in C++ is used to include the contents of another file in the current source file. When encountered, the preprocessor replaces the `#include` line with the contents of the specified file before compilation begins. There are two syntaxes: `<headerfile>` for standard library headers and `"headerfile"` for user-defined headers. For example, `#include "headerfile.h"` brings declarations from `headerfile.h` into the current file.

8. How to protect a header from being included multiple times?

To protect a header from being included multiple times, you can use include guards or `#pragma once`. Include guards involve defining a unique identifier within the header file and using conditional preprocessor directives to include the contents only if the identifier hasn't been defined.

```
#pragma once

// Contents of the header file go here
```

9. What is a namespace in C++ and how is it used?

In C++, a namespace is a way to organize and encapsulate code to prevent naming conflicts. It's declared using the namespace keyword, and entities within it are accessed using the scope resolution operator `::`.

```
namespace myNamespace {
    // Code goes here
}

myNamespace::myFunction();
```

Namespaces improve code modularity and readability, especially in larger projects or when integrating multiple libraries.

10. What is a compiler?

A compiler is a software tool that translates source code into machine code or executable code that a computer can understand and execute. It performs various tasks such as lexical analysis, syntax analysis, optimization, and code generation. The compiler ensures that the source code adheres to the syntax and rules of the programming language and produces

efficient and executable output.

11. What is a linker in C++?

Linker is a software utility responsible for combining multiple object files generated by the compiler into a single executable program or library. It resolves references between different modules or object files, ensuring that functions and variables defined in one file can be properly accessed and utilized by other parts of the program. The linker also handles the inclusion of libraries and external dependencies, allowing for the creation of standalone executables or shared libraries that can be executed or used by other programs.

12. What is an array in C++?

In C++, an array is a fixed-size collection of elements of the same data type stored in contiguous memory locations. It allows for the storage and manipulation of multiple values under a single variable name. Elements in an array are accessed using an index, which represents their position within the array.

```
int arr[5]; // Static array of size 5
```

Arrays in C++ have a fixed size determined at compile time, and their elements can be of any valid data type, such as integers, characters, or user-defined types.

13. What is a string in C++?

In C++, a string is a sequence of characters stored as a contiguous block of memory. It represents text data and is a fundamental data type in the C++ Standard Library. Strings can be manipulated using various built-in functions and operators, making them versatile for text processing tasks. C++ provides two primary string types: `std::string`, which is a part of the C++ Standard Library, and C-style strings, which are arrays of characters terminated by a null character (`'\0'`). `std::string` provides additional functionality, such as dynamic resizing, compared to C-style strings.

14. What are loops in C++?

Loops in C++ are control structures that allow the repeated execution of a block of code while a specified condition is true. They enable automating repetitive tasks and iterating over collections of data. C++ provides several types of loops:

1. `for` **loop:** Executes a block of code repeatedly based on a specified initialization, condition, and update.

```
for (initialization; condition; update) {
    // Code to be executed
}
```

2. `while` **loop:** Executes a block of code repeatedly as long as a specified condition is true.

```
while (condition) {
    // Code to be executed
}
```

15. What is the difference between while and do-while loops in C++?

The main difference between while and do-while loops is when their condition is evaluated. `while` loops are entry-controlled loops, where the condition is checked before entering the loop body. In contrast, `do-while` loops are exit-controlled loops, where the condition is

16. What is a function in C++?

In C++, a function is a named block of code that performs a specific task. Functions are reusable and modular, allowing you to break down complex tasks into smaller, more manageable parts. They typically accept input parameters, perform some operations, and optionally return a result.

```
1 void myFunction() {
2     // code goes here
3 }
```

Functions help improve code organization, readability, and maintainability by promoting code reuse and abstraction.

17. How does const affect a variable?

`const` affects a variable by making it immutable after assigning a value. When a variable is declared as `const`, it cannot be changed later in the program. Attempting to modify the value of a `const` variable will result in a compilation error. Using `const` also helps optimize code and ensures safety by guaranteeing that the variable's value remains unchanged during program execution.

18. How does static affect global/local variables?

In C++, `static` affects global variables by limiting their visibility to the current source file. For local variables, `static` changes their storage duration.

19. What is a pointer variable in C++?

A pointer variable in C++ is a special type of variable that stores the memory address of another variable. Instead of storing the actual value, a pointer variable holds the location (address) in memory where the value is stored. This allows indirect access to the value

stored in memory. Pointer variables are declared using an asterisk (*) before the variable name, followed by the data type of the value it points to.

```
int* ptr; // Declaration of a pointer variable that points to an integer
```

To access the value stored at the memory address pointed to by a pointer variable, you dereference the pointer using the asterisk (*) operator. For example:

```
1 int x = 10;
2 int* ptr = &x; // Assigning the address of x to ptr
3 int value = *ptr; // Dereferencing ptr to get the value stored at the addre
```

20. What is the difference between passing parameters by reference and by value in C++?

Passing parameters by value involves making a copy of the actual parameter's value and passing it to the function. This means any modifications made to the parameter within the function do not affect the original variable outside the function.

```
1  void increment_val(int x) {
2      x++;
3  }
4
5  void increment_ptr(int* x) {
6      (*x)++;
7  }
8
9  int main() {
10     int num = 5;
11     // num remains 5 because the function operates on a copy of num
12     increment_val(num);
13     // num becomes 6 because the function operates on a reference of num
14     increment_ptr(&num);
```

Intermediate Questions

1. What is OOP (Object-Oriented Programming)?

OOP (Object-Oriented Programming) is a programming paradigm based on objects and classes. It bundles data and functionality into objects, which interact. Key principles include encapsulation, inheritance, polymorphism, and abstraction, providing a structured approach to software development.

Principle	Description
Encapsulation	Objects store their data (attributes) and methods (functions) in a single unit known as a class.
Inheritance	Allows a class to inherit properties and behaviors from another class, promoting code reusability.
Polymorphism	Objects can be treated as instances of their parent class, enabling code to be written in a more generic way.
Abstraction	Represents essential features without including background details, managing complexity effectively.

2. What is a constructor?

A constructor in C++ is a special member function of a class that is automatically called when an object of that class is created. Its purpose is to initialize the object's data members and set up the object's state. Constructors have the same name as the class and do not have a return type, not even void. They can be overloaded, allowing multiple constructors with different parameter lists to initialize objects in different ways.

```cpp
#include <iostream>

class MyClass {
public:
    // Constructor
    MyClass() {
        std::cout << "Constructor called" << std::endl;
    }
};

int main() {
    // Creating an object of MyClass
```

```
13      MyClass obj; // Constructor called
14  }
```

Constructors are commonly used to perform tasks such as memory allocation, initialization of member variables, and setting default values.

3. What are the types of constructors in C++?

Default constructor is used to create objects when no specific initial values are provided. Parameterized onstructor allows you to create objects with given values right from the start. Copy constructor helps in making a new object that's an exact copy of another object from the same class. Copy assignment operator It's used to set one object's values to another through assignment. Move constructor added in C++11, this constructor optimizes resource transfer, making things run faster and more efficiently.

4. What does the inline keyword mean?

The `inline` keyword in C++ is a specifier used to suggest that a function should be expanded inline by the compiler, rather than being called as a separate function. This can potentially improve performance by avoiding the overhead of a function call. However, it's just a suggestion to the compiler, and the compiler may choose not to inline the function. Typically, small and frequently called functions are good candidates for inlining.

5. What is a delegating constructor?

A delegating constructor in C++ is a constructor within a class that calls another constructor of the same class to perform part of its initialization. Essentially, it delegates some or all of its initialization responsibilities to another constructor within the same class. This allows for code reuse and simplifies the initialization process, especially when multiple constructors share common initialization logic. Delegating constructors were introduced in C++ 11.

6. What is an initializer list?

An initializer list in C++ is a way to initialize the data members of an object or invoke the constructors of its base classes in the constructor initialization list.

```
// Constructor without initializer list
MyClass(int _value, const std::vector<int>& _data)
{
    value = _value;
    data = _data;
}
```

It consists of a comma-separated list of expressions enclosed in braces `{ }` and is placed after the colon `:` following the constructor's parameter list. Using an initializer list allows for efficient initialization of data members and base classes, especially for non-default constructible types or const members.

```
// Constructor with initializer list
MyClass(int value, const std::vector<int>& data)
    : value(value), data(data) {}
```

It's a recommended practice for initializing class members in constructors to ensure proper initialization and avoid unnecessary overhead.

7. What is the this keyword?

The this keyword in C++ is a pointer that points to the current object. It is automatically available within member functions of a class and allows access to the object's own data members and member functions. When a member function is called on an object, this points to that object's memory location, enabling the function to operate on the object's data. The this pointer is useful in situations where member function parameters have the same names as data members, allowing for disambiguation. Additionally, it's used to enable method chaining and to pass the object itself as an argument to other functions.

8. What is STL?

The Standard Template Library (STL) is a collection of generic algorithms and data structures provided as a part of the C++ Standard Library. It includes containers (such as vectors, lists, and maps), algorithms (such as sorting and searching), and iterators (for iterating over elements in containers). STL components are highly efficient, flexible, and reusable, making them a fundamental part of C++ programming. The Standard Template Library (STL) components are typically found in several header files provided by the C++ Standard Library:

```
#include <vector>
#include <list>
#include <deque>
#include <set>
#include <map>
#include <unordered_set>
#include <unordered_map>
#include <algorithm>
#include <iterator>
#include <utility>
```

9. What are the types of polymorphism in C++?

Compile-time Polymorphism resolved during compile-time, includes function overloading and operator overloading. And run-time polymorphism which resolved during run-time, includes virtual functions and function overriding.

10. What is the difference between overload and override?

Overload refers to defining multiple functions or operators with the same name but different parameters within the same scope (e.g., function overloading, operator overloading) while override refers to providing a specific implementation of a function in a derived class that is already defined in its base class, using the override keyword. This is

essential for achieving run-time polymorphism through virtual functions.

11. What is an abstract class?

An abstract class in C++ is a class that cannot be instantiated directly and is designed to serve as a base for other classes. It may contain one or more pure virtual functions, which are declared with the virtual keyword and assigned a 0 or = 0 as their body.

```
1 // Abstract class
2 class Shape {
3 public:
4     // Pure virtual function
5     virtual void draw() const = 0;
6 };
```

Abstract classes are intended to define an interface or a common set of functionalities that derived classes must implement. They cannot be instantiated on their own but can be used as a base class for creating concrete classes. This concept helps achieve abstraction and polymorphism in object-oriented programming.

12. What is virtual function in C++?

A virtual function in C++ is a member function of a class that is declared with the virtual keyword and can be overridden in derived classes. When a virtual function is called through a base class pointer or reference, the actual implementation of the function is determined at runtime based on the type of the object pointed to or referenced.

Virtual function can be a pure virtual function in C++ if it declared in a base class without providing an implementation, using the virtual keyword followed by = 0 in its declaration. It serves as a placeholder for functionality that must be implemented by derived classes, ensuring polymorphic behavior.

13. What types of conversions are present in C++?

C++ supports several types of conversions that allow for the manipulation of data types during program execution. These conversions are crucial for facilitating operations between different types.

Type of Conversion	Description	Examples/Usage
Implicit (Automatic) Type Conversions	Occurs automatically without explicit programmer indication. Happens during operations requiring different types.	Conversion of expressions with different types, passing arguments to functions.
Explicit (Qualitative) Type Conversions	Occurs through explicit programmer indication using type cast operator (data_type) expression.	Used for conversions between compatible types or to suppress compiler warnings.
Static Type Conversion	Performed using the type cast operator, occurs at compile time.	Cast conversions in expressions to enforce type compatibility.
Dynamic Type Conversion	Performed using dynamic_cast, occurs at runtime. Used for safe type conversion of pointers in class hierarchies.	Converting pointers to objects of a base class to pointers to derived classes.
Variants of Implicit Type Conversions	Includes conversions among integers, floating-points, and pointers without explicit casting.	Integer and floating-point promotions or conversions, pointer conversions.

14. What is an exception? How do you throw and catch one?

An exception is an event in a program that disrupts the normal flow of execution. It typically arises from situations like division by zero, file not found, or invalid input. In C++, exceptions are used to handle such anomalies gracefully, allowing the program to continue running or terminate cleanly.

To throw an exception in C++, you use the throw keyword followed by an exception object. This object could be of any data type, including built-in data types, pointers, or objects of a user-defined class.

```
throw "Division by zero error";  // Throwing a string literal
throw -1;                        // Throwing an integer
```

15. What are try-throw-catch blocks?

Try-throw-catch blocks are constructs in C++ used for exception handling. They allow developers to handle runtime errors gracefully by separating error detection (throwing) from error handling (catching).

- **try block**: This block encloses the code where exceptions may occur. It's followed by one or more catch blocks.

- **throw statement**: This statement is used to raise an exception when an error condition is encountered within the try block.

- **catch block**: These blocks follow the try block and are used to catch and handle exceptions thrown within it. Each catch block specifies the type of exception it can catch.

16. What happens if an exception is not caught?

If an exception is thrown in C++ but not caught, it will propagate up the call stack, seeking a matching catch block that can handle it. If no such catch block is found throughout the entire call stack, the program will terminate abnormally. This termination process also includes calling the destructors of all objects that have been fully constructed in the scope between the throw point and the point where the exception exits the program. Additionally, before the program terminates, the C++ runtime system will typically call a special function named `std::terminate()`. This function, by default, stops the program by calling `std::abort()`. Therefore, uncaught exceptions lead to the abrupt termination of the program, which can result in a loss of data or other undesirable outcomes. It is generally a good practice to catch and properly handle exceptions to maintain robustness and prevent unexpected behaviors in software.

17. What is the mutable keyword and when should it be used?

The `mutable` keyword in C++ allows a member of a `const` object to be modified. It is useful in scenarios where a member variable needs to be changed without affecting the external state of the object, such as when implementing caching mechanisms. A `mutable` member

can be modified even inside a `const` member function, helping maintain const-correctness of the method while allowing changes to the internal state of the object. For example, it allows updating a cache or a logging counter in an otherwise constant context.

18. What is the friend keyword and when should it be used in C++?

The `friend` keyword in C++ is used to grant access to private and protected members of a class to other classes or functions. When a class or function is declared as a friend of a class, it has full access to the private and protected members of that class.

However, the `friend` keyword should be used cautiously as it breaks encapsulation and can make the code less understandable and harder to maintain. Its usage should be limited only to cases where it's truly necessary to achieve the objective.

19. What is a lambda and an anonymous function in C++?

In C++, a lambda expression, also known as an anonymous function, is a convenient way of defining an inline function that can be used for short snippets of code that are not going to be reused elsewhere and therefore do not need a named function. Introduced in C++11, lambdas are widely used for functional programming styles, especially when using standard library functions that accept function objects, such as those in `<algorithm>` and `<functional>` modules.

```
1 // Lambda to print each element multiplied by a captured value
2 std::for_each(vec.begin(), vec.end(), [multiplier](int n) {
3     std::cout << n * multiplier << std::endl;
4 });
```

20. What is a functor in C++?

In C++, a functor, also known as a function object, is any object that can be used as if it were a function. This is achieved by overloading the `operator()` of a class.

```
class MultiplyBy {
private:
    int factor; // Member to store the multiplication factor
public:
    MultiplyBy(int x) : factor(x) {} // Constructor to initialize the fact

    // Overloaded operator() allows the object to act like a function
    int operator()(int other) const {
        return factor * other;
    }
};
```

Functors can maintain state and have properties, which distinguishes them from ordinary functions and allows them to perform tasks that require maintaining internal state across invocations or holding configuration settings.

Advanced Questions

1. How templates work in C++?

In C++, templates provide a way to create generic classes and functions that can work with any data type. They allow you to write code once and use it with different data types without having to rewrite the code for each type.

1. **Template Declaration**: To create a template, you use the `template` keyword followed by a list of template parameters enclosed in angle brackets (`<>`). These parameters can represent types (`typename` or `class`) or non-type parameters (like integers).

2. **Template Definition**: You define your class or function as you normally would, but instead of specifying a concrete type, you use the template parameter.

3. **Template Instantiation**: When you use the template with a specific data type, the compiler generates the necessary code by replacing the template parameters with the actual types or values.

4. **Code Generation**: The compiler generates separate code for each instantiation of the template, tailored to the specific data types or values used.

```cpp
#include <iostream>

// Template function declaration
template <typename T>
T add(T a, T b) {
    return a + b;
}

int main() {
    // Template function instantiation with int
    std::cout << add(5, 3) << std::endl;  // Output: 8

    // Template function instantiation with double
    std::cout << add(3.5, 2.1) << std::endl;  // Output: 5.6

    return 0;
}
```

2. What is lvalue and rvalue?

In C++, an lvalue refers to an expression that represents an object that occupies some identifiable location in memory (i.e., it has a name). On the other hand, an rvalue refers to an expression that represents a value rather than a memory location, typically appearing on the right-hand side of an assignment expression.

1. **Lvalue (Locator value)**: Represents an object that has a memory location and can be assigned a value. Examples include variables, array elements, and dereferenced pointers.

2. **Rvalue (Right-hand side value)**: Represents a value rather than a memory location. It's typically temporary and cannot be assigned to. Examples include numeric literals, function return values, and the result of arithmetic operations.

Understanding the distinction between lvalues and rvalues is important in contexts like assignment, function calls, and overload resolution. It helps determine whether an expression can be assigned to or modified.

3. When can std::vector use std::move?

`std::vector` can use `std::move` when you want to efficiently transfer ownership of its elements to another vector or to another part of your program.

1. **Move Semantics**: When you want to transfer the contents of one vector to another efficiently without deep copying. For example:

```
1 std::vector<int> source = {1, 2, 3};
2 std::vector<int> destination = std::move(source);
```

2. **Returning from Functions**: When a function returns a vector, you can use `std::move` to transfer ownership of the vector contents efficiently. For example:

```
1 std::vector<int> createVector() {
2     std::vector<int> vec = {4, 5, 6};
3     return std::move(vec);
4 }
```

4. What is metaprogramming?

Metaprogramming is a programming technique where a program writes or manipulates other programs (or itself) as its data. It involves writing code that generates code dynamically during compilation or runtime. Metaprogramming is commonly used in languages like C++ and Lisp, where it enables tasks such as template instantiation, code generation, and domain-specific language creation.

5. What is SFINAE in simple words?

SFINAE stands for "Substitution Failure Is Not An Error." In simple words, SFINAE is a principle in C++ template metaprogramming where if a substitution during template instantiation fails (often due to a type deduction failure), it's not considered a compilation error. Instead, the compiler tries alternative templates or overloads, allowing the program to continue compilation without raising an error.

6. How do I work with build systems like Make and CMake?

To work with build systems like Make and CMake, you first need to understand your project's structure. Then, you write scripts (Makefile for Make, CMakeLists.txt for CMake) to describe how to compile and link your project. After configuring the build environment, you run the build process using commands like `make` (for Make) or `cmake` (for CMake). Pay attention to any errors or warnings during compilation, and make sure your built executable behaves as expected. Additionally, manage project dependencies properly to ensure a smooth build process.

7. What are the characteristics of the std::set, std::map, std::unordered_map, and std::hash containers?

These containers provide different trade-offs in terms of performance and ordering guarantees, allowing developers to choose the most appropriate one based on their specific requirements.

Container/Function	Description	Properties	Complexity
std::set	Represents a sorted collection of unique elements.	Elements are sorted according to a specified comparison function.	Insertion, removal, and search operations have a logarithmic complexity (O(log n)).
std::map	Represents an associative container that stores key-value pairs.	Keys are unique and sorted based on a comparison function.	Insertion, removal, and search operations have a logarithmic complexity (O(log n)).
std::unordered_map	Represents an associative container that stores key-value pairs.	Keys are unique and unordered, based on a hash function.	Insertion, removal, and search operations have an average constant complexity (O(1)).
std::hash	Is a function object that generates hash values for its arguments.	Used by hash-based containers like std::unordered_map and std::unordered_set.	Used to compute the bucket index where elements should be stored.

8. Describe the purpose and operating principle of std::shared_ptr, std::unique_ptr, and std::weak_ptr.

`std::shared_ptr`, `std::unique_ptr`, and `std::weak_ptr` are smart pointers provided by the C++ Standard Library (`<memory>` header) to manage dynamic memory allocation and avoid memory leaks. Here's an overview of their purpose and working principles:

std::shared_ptr

- **Purpose**: Manages shared ownership of a dynamically allocated object. Multiple `shared_ptr` instances can share ownership of the same object.
- **Working Principle**: Each `shared_ptr` maintains a reference count (control block) that keeps track of how many `shared_ptr` instances point to the same object. When the last `shared_ptr` pointing to the object is destroyed or reset, the object is deleted.

std::unique_ptr

- **Purpose**: Manages unique ownership of a dynamically allocated object. Only one `unique_ptr` instance can own the object, and it is responsible for deleting it when it goes out of scope.
- **Working Principle**: `unique_ptr` ensures exclusive ownership of the object it points to. When a `unique_ptr` is moved or destroyed, it automatically releases the ownership and

deletes the object.

std::weak_ptr

- **Purpose**: Provides non-owning ("weak") references to an object managed by `std::shared_ptr`. It allows you to observe an object without affecting its lifetime.
- **Working Principle**: `weak_ptr` does not contribute to the reference count of the shared object. It can be converted to a `shared_ptr` to access the object, but if the object has been deleted, the conversion will result in an empty `shared_ptr`.

9. What is Return Value Optimization (RVO)?

Return Value Optimization (RVO) is a compiler optimization technique in C++ that eliminates unnecessary copying of objects when returning them from a function. Instead of creating a temporary copy of the object and returning it, the compiler constructs the object directly in the memory location of the caller's destination object. This optimization reduces the overhead associated with copying large objects, improving performance and reducing memory usage.

10. What is template specialization?

Template specialization in C++ allows providing custom implementations for templates for specific types or sets of types. It enables tailoring the behavior of a template for particular cases where the default implementation may not be suitable. There are two types: explicit specialization, where you override the default behavior for specific types, and partial specialization, where you specialize the template for a subset of possible template arguments.

```
template<typename T>
struct MyTemplate {
    // Default implementation
};

template<>
struct MyTemplate<int> {
```

```
8     // Specialized implementation for int
9 };
```


FAQs

Q: What is the most common type of question asked in C++ interviews?
A: Most C++ interviews start with basic questions about syntax, data types, and control structures. They often move into more complex topics like memory management, object-oriented programming principles, and algorithmic challenges.

Q: How should I prepare for C++ interview questions?
A: Preparation should include a solid understanding of C++ fundamentals, practice with

coding problems, and familiarity with common libraries like STL. Reviewing key concepts such as pointers, class inheritance, and polymorphism is also crucial.

Q: Are there any specific C++ features I should focus on understanding deeply?
A: Yes, mastering topics such as templates, exceptions, and the Standard Template Library (STL) can be particularly beneficial. Understanding modern C++ features introduced in C++11 and later, like smart pointers, lambda expressions, and concurrency features, is also highly recommended.

Q: What kind of programming tasks might I expect during a C++ interview?
A: You might be asked to solve algorithmic problems using C++, write classes or functions to implement specific behaviors, or refactor existing C++ code to improve its performance or readability. Understanding design patterns and being able to apply them in C++ is also a plus.

Q: How important are data structures and algorithms in a C++ interview?
A: A strong grasp of data structures (like arrays, lists, stacks, queues, trees, and graphs) and algorithms (such as sorting, searching, recursion, and dynamic programming) is essential, as many technical interview questions revolve around efficiently solving problems using these concepts.

C++ OOP

Welcome to the exciting world of Object-Oriented Programming (OOP) with C++! This course will...

C++ 4.8

COURSE Beginner

C++ Introduction Video Course

Welcome to C++ Introduction Video Course – your gateway to mastering the core concepts of...

C++ 4.5

Interview Preparation BackEnd Development Java

The 80 Top Java Interview Questions and Answers

Key Points to Consider When Preparing for an Interview

by Daniil Lypenets
Full Stack Developer

Apr, 2024 • 30 min read

Interview Preparation BackEnd Development SQL

The 50 Top SQL Interview Questions and Answers

For Junior and Middle Developers

by Oleh Lohvyn
Backend Developer

Apr, 2024 • 31 min read

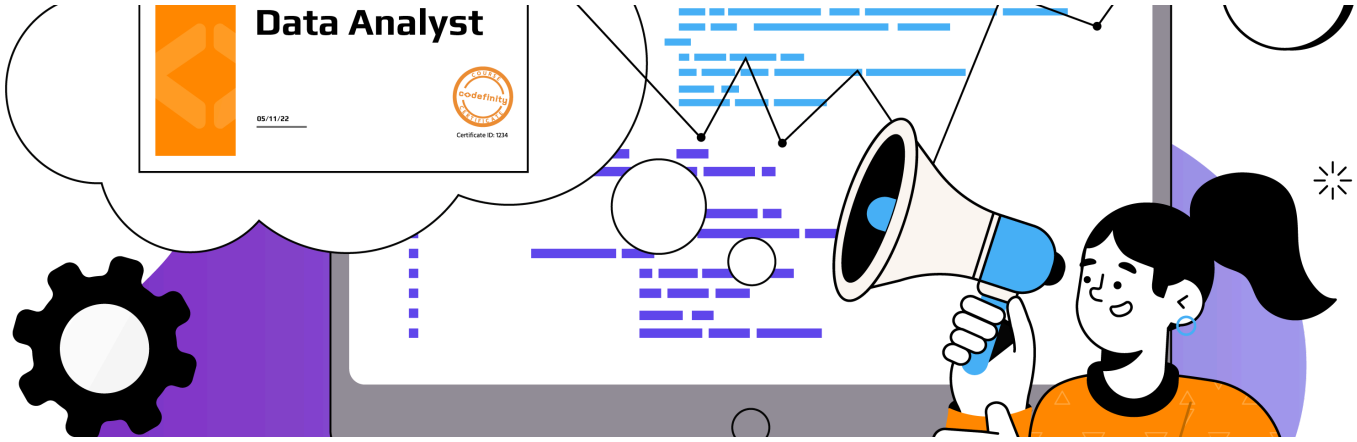
Python Data Analytics Interview Preparation

Top 50 Python Interview Questions for Data Analyst

Common Python questions for DA interview

by Ruslan Shudra
Data Scientist

Apr, 2024 • 27 min read



Topics

Artificial Intelligence
Data Visualization
Data Science
Computer Science
Machine Learning
Web Development
Development Tools
Mathematics
Frontend Development
Backend Development
Data Analytics
Game Development

Technologies

Python
SQL
HTML/CSS
C++
R
JavaScript
Java
React
C
Golang
C#
Dart
TypeScript
ChatGPT
Excel
Git

Learning Tracks

SQL from Zero to Hero
Python Data Analysis and Visualization
Frontend Development Foundation
C++ for Beginners