# C++ Interview Questions

Get ready for your C++ programming interview

C++ coding interviews can be tricky. Don't get caught off guard. These are some of the most common C++ interview questions. Learn these and you'll be more prepared for your C++ programming interview.

And once you're done with these, we have a bunch of general data structures and algorithms questions in C++ (/all-questions/cpp).

Interview Cake is not just another question database—we walk you through the question step-by-step, giving hints and explanations as you need them, just like a real interviewer.

## What is the difference between a shallow and a deep copy of a class?

### Answer

**Shallow copying** is faster, but it's "lazy" in how it handles *pointers* and *references*. Instead of making a fresh copy of the actual data the pointer points to, it just copies over the pointer value. So, both the original and the copy will have pointers that reference the same underlying data. (Ditto for references.)

**Deep copying** actually clones the underlying data; it's not shared between the original and the copy.

Let's get specific. Here's a class representing a bus:

```C++
class Bus
{
private:
    int yearBuilt_;
    string name_;
    vector<string>* routeList_;

public:
    Bus(int yearBuilt, const string& name, vector<string>& routeList) :
        yearBuilt_(yearBuilt),
        name_(name),
        routeList_(&routeList)
    {
    }

    void setName(const string& name);
    void addStop(const string& stopName);
    void removeStop(const string& stopName);
    void busDetails() const;
};
```

Pay close attention to our `routeList_` variable. We've made it a pointer so that it can be updated remotely (in case there's a detour). We've also included methods to manipulate the list of stops and print out the bus information.

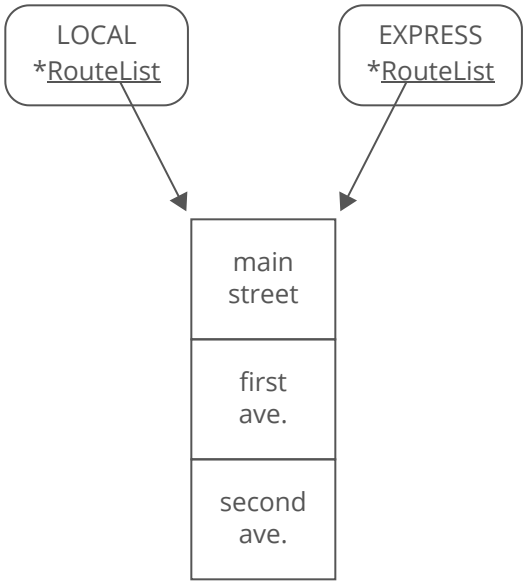
What do you think happens when we run this code?

```C++
void test()
{
   vector<string>* route = new vector<string>();

   // Add one bus to our fleet
   Bus local(2015, "LOCAL", *route);
   local.addStop("Main St.");
   local.addStop("First Ave.");
   local.addStop("Second Ave.");

   // Add another bus that skips First Ave.
   Bus express = local;
   express.setName("EXPRESS");
   express.removeStop("First Ave.");

   // Let's see what we've got
   local.busDetails();
   express.busDetails();
}
```

Here's the output:

```C++
The LOCAL bus was built in 2015 and its current
route is: Main St., Second Ave.

The EXPRESS bus was built in 2015 and its current
route is: Main St., Second Ave.
```

Oops. Looks like the local and express bus were both sharing the same route list. Like this:




Okay, so when we created the express bus (`Bus express = local`), we made a *shallow copy*, where both objects ended up with a pointer to the same route list.

We want the express to be a *deep copy* of the local bus, so that they each have their own route list.

How can we get deep copies?

Well, what happens when we run this line of code:

```
Bus express = local;
```
C++

In C++, there are *two* functions that copy the values from one object to another: **copy constructors** and **copy assignment operators**.

The copy constructor gets called whenever we are *constructing* a new object from an existing one. So, this declaration uses copy constructors:

```
Bus express(local);
```
C++

And, this code block *also* uses the copy constructor (even though there's an equals sign):

```
Bus express = local;
```
C++

The copy assignment operator gets called when we're updating an existing object:

```
Bus local;
Bus express;
express = local;
```

C++

Let's look back at the line where we created our express bus. Would it call the copy constructor or the copy assignment operator?

```
Bus express = local;
```

C++

Since we're *creating* the express bus, we'll use the copy constructor here.

Wait a second though. We didn't ever define a copy constructor! Where's the function that's getting called?

When writing a class, defining a copy constructor and a copy assignment operator is optional. If you don't specify one, then the C++ compiler will generate it for you, filling in a method that makes shallow copies.

If we want deep copies, we can get them be defining a copy constructor and assignment operator for our Bus class. Here's what our copy constructor could look like:

```
Bus::Bus(const Bus& src) :
    yearBuilt_(src.yearBuilt_),
    name_(src.name_),
    routeList_(new vector<string>(*src.routeList_))
{
}
```

C++

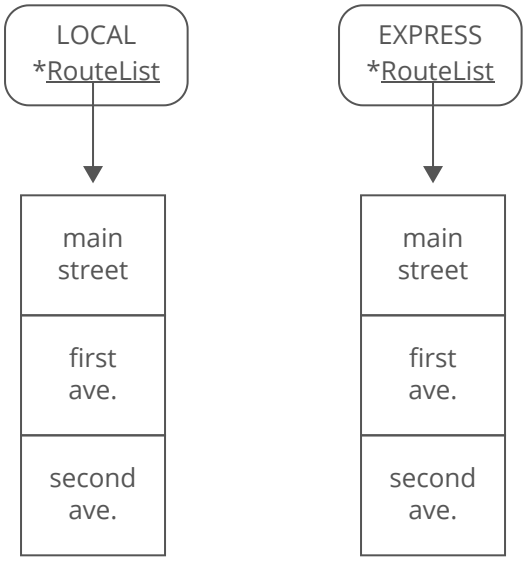
The line of code: `routeList_(new vector<string>(*src.routeList_))` creates new space so that the new Bus object can have its own route list.

Now, when we create our express bus, it'll call the copy constructor we just wrote:

```
Bus express = local;
```

C++

Under the hood, our buses will look like this:




That covers the copy constructor. What about the copy assignment operator?

We probably want to define that too. Otherwise, we'll get deep copies whenever we use the copy constructor but shallow copies whenever we use the copy assignment operator. That sounds like a surefire way to get some bugs.

So, let's write out a copy assignment operator that will make deep copies. It'll look a lot like the copy constructor, except we'll want to clean up the existing object before we start assigning new values. Like this:

C++

```
Bus& Bus::operator=(const Bus& src)
{
    if (this != &src) {
        // Clean up the existing routeList_
        delete routeList_;

        // Create a deep copy of the values from src
        yearBuilt_ = src.yearBuilt_;
        name_ = src.name_;
        routeList_ = new vector<string>(*src.routeList_);
    }
    return *this;
}
```

One last thing to note: copy constructors don't *have to* create deep copies. Sometimes, we want a separate copy of *some* of the class members and a shared reference for others.

For instance, if we add information about the bus supervisor, we'll probably want the same supervisor for all buses. So, we can let them share a reference to the supervisor name, while creating a separate copy of the route list:

```C++
Bus::Bus(int yearBuilt, const string& name, vector<string>& routeList, const string& superv
    yearBuilt_(yearBuilt),
    name_(name),
    routeList_(&routeList),
    supervisor_(&supervisor)
{
}

Bus::Bus(const Bus& src) :
    yearBuilt_(src.yearBuilt_),
    name_(src.name_),
    routeList_(new vector<string>(*src.routeList_)), // New bus gets its own copy of the rou
    supervisor_(src.supervisor_) // But shares the same supervisor
{
}
```

# What is a template function?

## Answer

**C++ makes us specify the types for function arguments and return values. Templates let us write functions that take in any type. This helps us keep our code DRY.**

For example, let's look at this swap function. Sometimes, we need to swap two integers:

```C++
void mySwap(int& x, int& y)
{
    int tmp = x;
    x = y;
    y = tmp;
}
```

But what if we wanted to swap two chars?

```C++
void mySwap(char& x, char& y)
{
    char tmp = x;
    x = y;
    y = tmp;
}
```

Or two strings?

```C++
void mySwap(string& x, string& y)
{
    string tmp = x;
    x = y;
    y = tmp;
}
```

Aside from the types, these functions are exactly the same! That's annoying—it means that our code is harder to read and there are more places for us to introduce bugs. It would be nice if we could write one function to swap two things of the same type that we could use instead of these three versions.

Template functions to the rescue!

Using templates, we can combine all three swap functions above, like this:

```C++
template <typename T>
void mySwap(T& x, T& y)
{
    T tmp = x;
    x = y;
    y = tmp;
};
```

So now we can write code like this:

```C++
int a = 1, b = 2;
mySwap(a, b);
cout << "a=" << a << ", b=" << b << endl;  // "a=2, b=1"

string c = "one", d = "two";
swap(c, d);
cout << "c=" << c << ", d=" << d << endl;  // "c=two, d=one"
```

**Careful:** C++ templates can make programs "bloated" by increasing the amount of code that needs to be compiled. Remember the three different swap functions we created earlier? Behind the scenes, the C++ compiler is generating all three of those functions: one for integers, one for strings, and one for characters. Using templates saves us time and makes our code shorter, but we're definitely not saving any space.

# What is the Diamond problem? How can we get around it?

The diamond problem is a common question used to test your understanding of multiple inheritance.

## Answer

The diamond problem refers to an issue when a base class is inherited by two subclasses, and both subclasses are inherited by a *fourth* class. When this happens, we need to give the compiler a bit of guidance about the exact structure of inheritance we want.

Let's use the example of lions, tigers, and ... *ligers*:

We'll start with a base class, Mammal, with the method `eat()`:

C++

```
class Mammal
{
protected:
    const string name_;

public:
    Mammal(const string& name);

    void eat()
    {
        cout << "This Mammal " << name_ << " is eating" << endl;
    }
};
```

Lion and Tiger both inherit from Mammal:

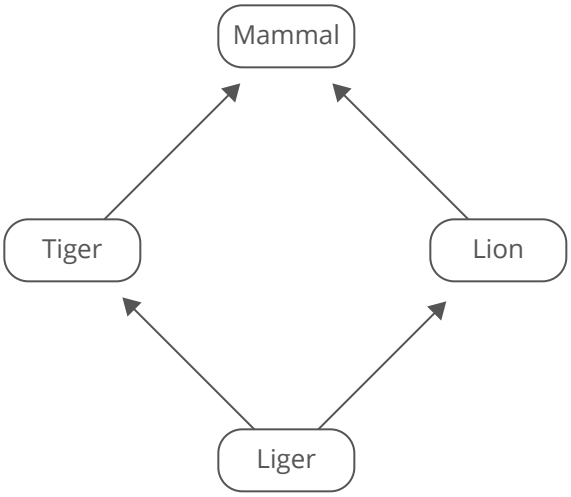
C++

```
class Tiger : public Mammal
{
public:
   void groom();
};

class Lion : public Mammal
{
public:
   void groom();
};
```

Finally, Liger inherits from *both* Lion and Tiger:

```
class Liger : public Tiger, public Lion
{
public:
   void nap();
};
```
C++

This is the inheritance structure we're going for. (See how it looks like a diamond!)




Satisfied with our structure so far, we decide to write some code using the Liger class:

```
Liger Lyle("Lyle");
Lyle.groom();
Lyle.eat();
```
C++

But, when we compile it, we get this error message:

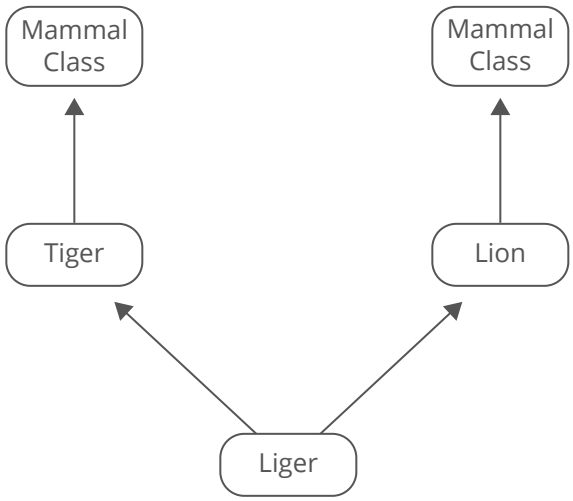
```
$ g++ ligers.cpp
error: non-static member 'eat' found in multiple
base-class sub-objects of type 'Mammal':
class Liger -> class Tiger -> class Mammal
class Liger -> class Lion -> class Mammal
Lyle.eat();
```
C++

What's happening here?

The compiler is complaining that the Liger class has two versions of the eat method—one from the Tiger class (which inherited it from the Mammal class) and one from the Lion class (which also inherited it from the Mammal class). So, the compiler sees two different eat methods, and it

doesn't know which version it should use for Ligers.

This happens because C++ doesn't recognize that the Lion and Tiger classes are inheriting from the same Mammal class. What it sees instead is something like this:




We can ensure that the compiler inherits the same Mammal class into the Lion and Tiger classes with the virtual keyword, like this:

C++

```
class Tiger : virtual public Mammal
{
public:
    void groom();
};

class Lion : virtual public Mammal
{
public:
    void groom();
};
```

When a base class is **virtually inherited**, the C++ compiler makes sure that this class is created only once. That means that all subclasses, like Lion and Tiger, will lead back to the same Mammal base class, giving us the diamond structure we've wanted all along.

Note that we have to put the virtual keyword in the definition of *both* Tiger and Lion. If we don't then we'll get the incorrect structure we had before.

The virtual keyword can also be used with class methods. When a class method is declared virtual, it can be overridden by inheriting classes.

Going back to our example with animals, we might want to let different mammals define their own `eat()` method. Since we'll be overriding the eat method in inheriting classes, we need to declare it as virtual, like this:

C++

```
class Mammal
{
protected:
    const string name_;

public:
    Mammal(const string& name);

    virtual void eat()
    {
        cout << "This Mammal " << name_ << " is eating" << endl;
    }
};
```

> When developing classes, most public or protected methods will be declared virtual so that inheriting classes can override them as needed. In fact, the only method that *can't* be declared virtual is the class constructor. (Clearly, we don't want others to override that!)

# Why are destructors important?

## Answer

Destructors are important because they give us a chance to free up an object's allocated resources before the object goes out of scope. Since C++ doesn't have a garbage collector, resources that we don't free ourselves will never be released back to the system, eventually making things grind to a halt.

> A **resource leak** is when a program finishes using some resource (allocated memory, open files, network sockets) but doesn't properly release it, hoarding resources it's not using any more. Mozilla was famous for having memory leaks with Firefox: the browser would cache recently visited sites so that users could quickly go back to them, but it wouldn't delete saved sites. Over time, the amount of data saved would get pretty large, slowing down the browser.

> We're focusing on C++98. We should note, though, that newer versions of C++ have introduced "smart pointers," which *are* automatically freed when they go out of scope. If you can use smart pointers, they're generally preferable since you can avoid some of the headaches of manual memory management. That said, many projects still use manual memory management, so it's important to understand destructors.

To make this concrete, let's look at an example: shopping carts. Suppose our shopping cart class keeps track of its items in a dynamically allocated vector, like this:

C++

```
class ShoppingCart
{
private:
    vector<string>* items_;

public:
    ShoppingCart() :
        items_(new vector<string>())
    {
    }

    ~ShoppingCart()
    {
         // Need to free up items here
    }

    void addItem(const string& item)
    {
        items_->push_back(item);
    }
};
```

A destructor method is declared with the same name as the constructor, with a ~ in front. For our `ShoppingCart` class, that's `~ShoppingCart`. The C++ compiler exclusively reserves this syntax for destructors, so it's important to not use this name for anything else.

The `items` vector is dynamically allocated and could take up a lot of space, so we need to free it when we're done. How do we do that?

We can just call the vector's destructor!

Careful, though! We don't call destructors directly.

- If we allocated an object dynamically (using `new`), then we call the object's destructor using the `delete` keyword.
- For objects on the stack (not allocated using `new`), the destructor will be called automatically once it is out of scope.

Let's see how this works if we have a few shopping carts.

```C++
{
    ShoppingCart cart;
    cart.addItem("eggs");
    cart.addItem("cheese");
}

// When cart goes out of scope here,
// destructor ShoppingCart::~ShoppingCart() is called automatically
```

This shopping cart is allocated on the stack, so its destructor gets called automatically once it goes out of scope.

```C++
ShoppingCart* cart = new ShoppingCart();
cart->addItem("milk");
cart->addItem("bread");
delete cart;  // calls destructor ShoppingCart::~ShoppingCart() for cart
```

This shopping cart is dynamically allocated, so we have to manually call its destructor when we're done with it.

Okay, but what should actually go in the destructor for `ShoppingCart`? Since we dynamically allocated the `items` vector, we need to use the `delete` keyword to call its destructor. We'd do that in `~ShoppingCart`, like this:

C++

```
ShoppingCart::~ShoppingCart()
{
    delete items_;
}
```

The nice thing about destructors is they centralize object cleanup in one place. As we make the `ShoppingCart` class more complex, we can always make sure we're not leaking resources by updating our destructor to do any necessary cleanup. That's *much* easier to maintain than having cleanup code scattered around our code base.

## Inflight Entertainment »

Writing a simple recommendation algorithm that helps people choose which movies to watch during flights keep reading »

## (/question/cpp/inflight-entertainment)

## Permutation Palindrome »

Check if any permutation of an input string is a palindrome. keep reading »

## (/question/cpp/permutation-palindrome)

## Reverse A Linked List »

Write a function to reverse a linked list in place. keep reading »

## (/question/cpp/reverse-linked-list)

## Kth to Last Node in a Singly-Linked List »

Find the kth to last node in a singly-linked list. We'll start with a simple solution and move on to some clever tricks. keep reading »

## (/question/cpp/kth-to-last-node-in-singly-linked-list)

## Recursive String Permutations »

Write a recursive function of generating all permutations of an input string. keep reading »

## (/question/cpp/recursive-string-permutations)

## In-Place Shuffle »

Do an in-place shuffle on an array of numbers. It's trickier than you might think! keep reading »

## (/question/cpp/shuffle)

## Cafe Order Checker »

Write a function to tell us if cafe customer orders are served in the same order they're paid for. keep reading »

## (/question/cpp/cafe-order-checker)

## Simulate 5-sided die »

Given a 7-sided die, make a 5-sided die. keep reading »

## (/question/cpp/simulate-5-sided-die)

## Simulate 7-sided die »

Given a 5-sided die, make a 7-sided die. keep reading »

## (/question/cpp/simulate-7-sided-die)

## Word Cloud Data »

You're building a word cloud. Write a function to figure out how many times each word appears so we know how big to make each word in the cloud. keep reading »

## (/question/cpp/word-cloud)

## Two Egg Problem »

A building has 100 floors. Figure out the highest floor an egg can be dropped from without breaking. keep reading »

## (/question/cpp/two-egg-problem)

## Find Repeat, Space Edition »

Figure out which number is repeated. But here's the catch: optimize for space. keep reading »

## (/question/cpp/find-duplicate-optimize-for-space)

## Find Repeat, Space Edition BEAST MODE »

Figure out which number is repeated. But here's the catch: do it in linear time and constant space! keep reading »

## (/question/cpp/find-duplicate-optimize-for-space-beast-mode)

## Find Duplicate Files »

Your friend copied a bunch of your files and put them in random places around your hard drive. Write a function to undo the damage. keep reading »

## (/question/cpp/find-duplicate-files)

## Apple Stocks »

Figure out the optimal buy and sell time for a given stock, given its prices yesterday. keep reading »

## (/question/cpp/stock-price)

## Binary Search Tree Checker »

Write a function to check that a binary tree is a valid binary search tree. keep reading »

## (/question/cpp/bst-checker)

## Parenthesis Matching »

Write a function that finds the corresponding closing parenthesis given the position of an opening parenthesis in a string. keep reading »

## (/question/cpp/matching-parens)

## MillionGazillion »

I'm making a new search engine called MillionGazillion(tm), and I need help figuring out what data structures to use. keep reading »

## (/question/cpp/compress-url-list)

## The Cake Thief »

You've hit the mother lode: the cake vault of the Queen of England. Figure out how much of each cake to carry out to maximize profit. keep reading »

## (/question/cpp/cake-thief)

## Making Change »

Write a function that will replace your role as a cashier and make everyone rich or something. keep reading »

## (/question/cpp/coin)

## Implement A Queue With Two Stacks »

Implement a queue with two stacks. Assume you already have a stack implementation. keep reading »

## (/question/cpp/queue-two-stacks)

## Compute nth Fibonacci Number »

Computer the nth Fibonacci number. Careful--the recursion can quickly spin out of control! keep reading »

## (/question/cpp/nth-fibonacci)

## Rectangular Love »

Find the area of overlap between two rectangles. In the name of love. keep reading »

## (/question/cpp/rectangular-love)

## Balanced Binary Tree »

Write a function to see if a binary tree is 'superbalanced'--a new tree property we just made up. keep reading »

## (/question/cpp/balanced-binary-tree)

## Bracket Validator »

Write a super-simple JavaScript parser that can find bugs in your intern's code. keep reading »

## (/question/cpp/bracket-validator)

## 2nd Largest Item in a Binary Search Tree »

Find the second largest element in a binary search tree. keep reading »

## (/question/cpp/second-largest-item-in-bst)

## Largest Stack »

You've implemented a Stack class, but you want to access the largest element in your stack from time to time. Write an augmented LargestStack class. keep reading »

## (/question/cpp/largest-stack)

## The Stolen Breakfast Drone »

In a beautiful Amazon utopia where breakfast is delivered by drones, one drone has gone missing. Write a function to figure out which one is missing. keep reading »

## (/question/cpp/find-unique-int-among-duplicates)

## Delete Node »

Write a function to delete a node from a linked list. Turns out you can do it in constant time! keep reading »

## (/question/cpp/delete-node)

## Top Scores »

Efficiently sort numbers in an array, where each number is below a certain maximum. keep reading »

## (/question/cpp/top-scores)

## Which Appears Twice »

Find the repeat number in an array of numbers. Optimize for runtime. keep reading »

## (/question/cpp/which-appears-twice)

## Find in Ordered Set »

Given an array of numbers in sorted order, how quickly could we check if a given number is present in the array? keep reading »

## (/question/cpp/find-in-ordered-set)

## Find Rotation Point »

I wanted to learn some big words to make people think I'm smart, but I messed up. Write a function to help untangle the mess I made. keep reading »

## (/question/cpp/find-rotation-point)

## Product of All Other Numbers »

For each number in an array, find the product of all the other numbers. You can do it faster than you'd think! keep reading »

## (/question/cpp/product-of-other-numbers)

## Highest Product of 3 »

Find the highest possible product that you can get by multiplying any 3 numbers from an input array. keep reading »

## (/question/cpp/highest-product-of-3)

## Merging Meeting Times »

Write a function for merging meeting times given everyone's schedules. It's an enterprise end-to-end scheduling solution, dog. keep reading »

## (/question/cpp/merging-ranges)

## Temperature Tracker »

Write code to continually track the max, min, mean, and mode as new numbers are inserted into a tracker class. keep reading »

## (/question/cpp/temperature-tracker)

## Ticket Sales Site »

Design a ticket sales site, like Ticketmaster keep reading »

## (/question/cpp/ticket-sales-site)

# What's next?

Check out our mock coding interview questions (/next).

They mimic a real interview by offering hints when you're stuck or you're missing an optimization.

**Try some questions now ➜**

Want more coding interview help?

Check out **interviewcake.com** for more advice, guides, and practice questions.