

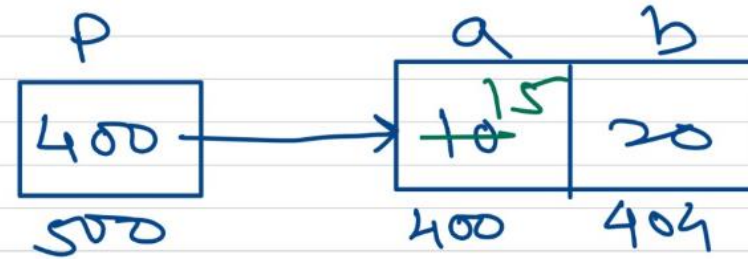


C++ INTERVIEW QUESTIONS

SUNBEAM INFOTECH



C++



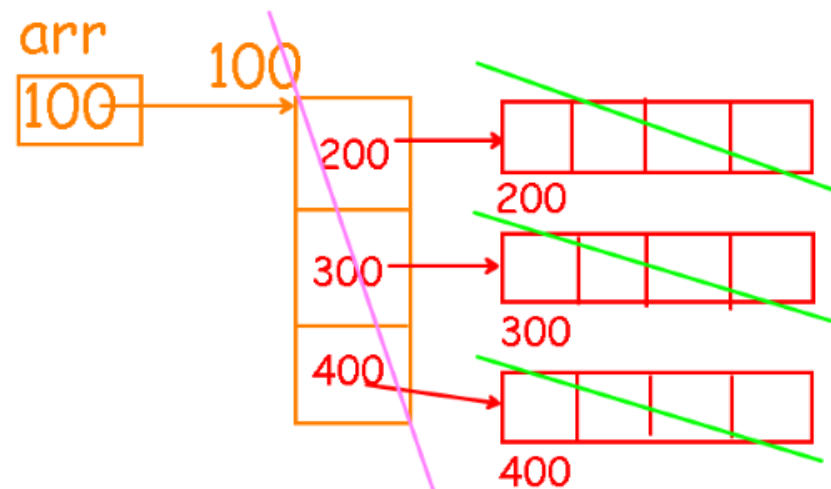
```
class test {
    int a;
    int b;
public:
    test() {
        a=10;
        b=20;
    }
    void display() {
        cout << a;
        cout << b;
    }
};
```

```
main() {
    test obj;
    int *p;
    // p = &obj; // error.
    p = reinterpret_cast<int*>(&obj);
    *p = 15;
    obj.display(); → 15, 20
    return 0;
}
```

```

49 int** allocateArray(int rows, int cols) {
50     int** arr = new int*[rows]; // Allocate memory for array of row pointers
51     for (int i = 0; i < rows; ++i) {
52         arr[i] = new int[cols]; // Allocate memory for each row
53     }
54     return arr;
55 }
56
57 void deallocateArray(int** arr, int rows) {
58     for (int i = 0; i < rows; ++i) {
59         delete[] arr[i]; // Deallocate memory for each row
60     }
61     delete[] arr; // Deallocate memory for array of row pointers
62 }
63
64 int main() {
65     int rows = 3;
66     int cols = 4;
67     int** arr = allocateArray(rows, cols);
68
69     for (int i = 0; i < rows; ++i) {
70         for (int j = 0; j < cols; ++j) {
71             arr[i][j] = i * cols + j; // Example initialization

```





CPP-Interview.md

OOP-Interview.md

Untitled-1

```
16
17 class myclass {
18     int a;
19 public:
20     // ...
21     void func1() {
22         // this -- myclass * const this;
23         this->a = 10; // okay
24     }
25     void func2() const {
26         // this -- const myclass * const this;
27         this->a = 10; // error
28     }
29 };
30
```

FileEditSelectionViewGo...private

CPP-Interview.md xUntitled-1

C: > Niles > sep-23 > Interview > CPP-Interview.md > ### 3. What is "this" pointer? Is it available for static, virtual, const and friend functions?

89

* The "this" pointer in C++ is a pointer that holds the address of the current object. It is a keyword that can be used within the member functions of a class to refer to the object on which the member function is called.

* The "this" pointer is implicitly available within the scope of non-static member functions of a class.

* It is commonly used to access member variables and member functions of the current object. For example:

```
```cpp
class Example {
private:
 int x;
public:
 void setX(int x) {
 this->x = x; // "this" pointer is used to differentiate between member variable and function parameter
 }
};
```
```

```
```cpp
int main() {
 Example ex;
 ex.setX(3);
}
```

if name of local var is same as data member, then using "this" is required to access data member,

200  
this

ex  
x  
200

90

91

92

93

94

95

96

97

98

99

100

101

102

103

104

105

106

107

108

\* \*\*Static Member Functions:\*\* "this" pointer is not available within static member functions because static member functions do not operate on specific class instances/objects

main000

105, Col 24Spaces: 4UTF-8CRLFMarkdown

Search

2:16 PM

CPP-Interview.md x

Untitled-1

C: > Niles > sep-23 > Interview > CPP-Interview.md > ### 3. What is "this" pointer? Is it available for static, virtual, const and friend functions?

83 deallocateArray(arr, rows);

84 return 0;

85 }

86 ...

87

88 ### 3. What is "this" pointer? Is it available for static, virtual, const and friend functions?

89 \* The "this" pointer in C++ is a pointer that holds the address of the current object. It is a keyword that can be used within the member functions of a class to refer to the object on which the member function is called.

90 \* The "this" pointer is implicitly available within the scope of non-static member functions of a class.

91 \* It is commonly used to access member variables and member functions of the current object. For example:

92 ```cpp

93 class Example {

94 private:

95 int x;

96 public:

97 void setX(int x) {

98 this->x = x; // "this" pointer is used to differentiate between member variable and function parameter

99 }

100 }

101 ...

102 ```cpp

103 int main() {

class::method();

ptr->fun();

obj.fun();

if friend fn is global fn

✓ this

✓ this

✗ this

✗ this

✗ this

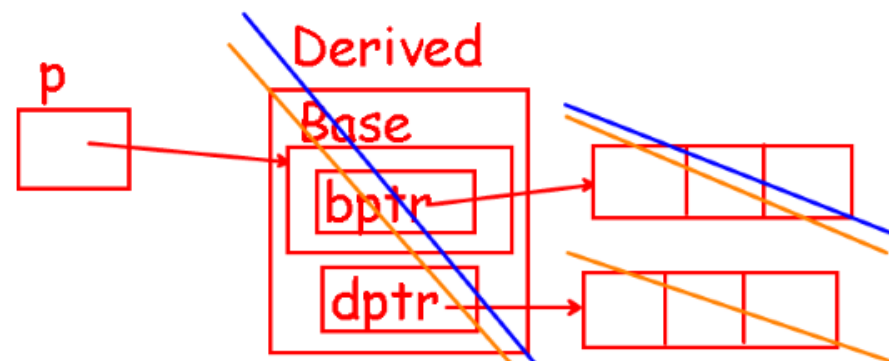


```

130 class Base {
131 int *bptr;
132 public:
133 Base() {
134 bptr = new int[3];
135 std::cout << "Base constructor called" << std::endl;
136 }
137 virtual ~Base() {
138 delete[] bptr;
139 std::cout << "Base destructor called" << std::endl;
140 }
141 };
142 class Derived : public Base {
143 int *dptr;
144 public:
145 Derived() {
146 dptr = new int[3];
147 std::cout << "Derived constructor called" << std::endl;
148 }
149 ~Derived() {
150 delete[] dptr;
151 std::cout << "Derived destructor called" << std::endl;
152 }

```

Base \*p = new Derived; main()+  
// ...  
delete p;



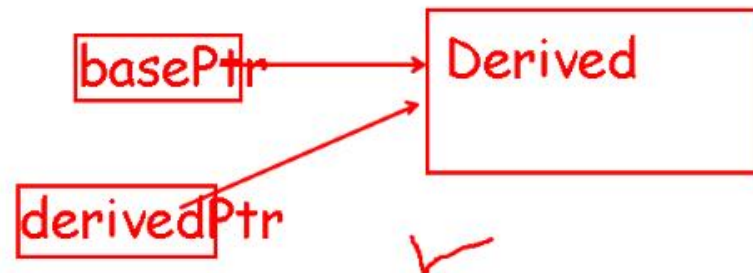
if destructor is not virtual, Derived cls destructor not called. Hence "dptr" pointing memory will leak.

if destructor is virtual, first derived destructor, then base destructor called and then object memory released,

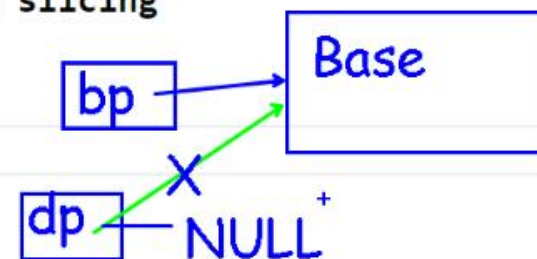
```

177 class Base {
178 public:
179 virtual ~Base() {}
180 };
181 class Derived : public Base {
182 public:
183 void derivedMethod() {
184 std::cout << "Derived method called" << std::endl;
185 }
186 };
187 int main() {
188 Base* basePtr = new Derived(); // Upcasting: Derived object assigned to Base pointer
189 //basePtr->derivedMethod(); // compiler error: due to Object slicing
190
191 // Attempting to downcast using dynamic_cast
192 Derived* derivedPtr = dynamic_cast<Derived*>(basePtr);
193 // Check if downcast was successful
194 if (derivedPtr != NULL) {
195 // Downcast successful, call derived class method
196 derivedPtr->derivedMethod();
197 } else {
198 // Downcast failed, handle accordingly
199 std::cout << "Dynamic cast failed" << std::endl;
200 }

```



Base \*bp = new Base;





File Edit Selection View Go ... private

CPP-Interview.md x Untitled-1

C: > Nileshe > sep-23 > Interview > CPP-Interview.md > ### 7. Explain dynamic\_cast operator. When it is required? Explain with example.

175 #include <iostream>

176 B::type class Base { hidden

177 int b; \*vptr;

178 B::f1() public:

179 virtual void fun1() { }

180 B::f2() virtual void fun2() { }

181 B::dtor virtual ~Base() { }

182 B::vtable};

183 class Derived : public Base {

184 D::type int d;

185 public:

186 D::f1() virtual void fun1() { }

187 B::f2() void derivedMethod() {

188 B::dtor std::cout << "Derived method called" << std::endl;

189 D::vtable};

190 int main() {

191 Base\* basePtr = new Derived; // Upcasting: Derived object assigned to Base pointer

192 //basePtr->derivedMethod(); // compiler error: due to Object slicing

193

194 // Attempting to downcast using dynamic\_cast

195 Derived\* derivedPtr = dynamic\_cast<Derived\*>(basePtr);

196 // Check if downcast was successful

197

Base \*bp = new Base;

Base

vptr \*

b

12 bytes

bp

bp->f1();

0

bp->f2();

1

Base \*bp = new Derived;

Derived

vptr \*

b

d

16 bytes

bp

bp->f1();

0

bp->f2();

1

Late binding: When virtual fn is called on pointer/reference,

it is always late binded. Compiler generate code that will invoke fn at runtime.

1. get index of the fn in the vtable.

2. go to the addr pointed by the pointer and read 8 bytes to jump to vtable.

3. call fn at the index from that vtable.

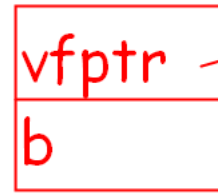
Base bobj;

Base::vftable

```

59 class Base {
60 public: void *vfptr;
61 int b;
62 virtual void f1() { ... }
63 virtual void f2() { ... }
64 void f3() { ... }
65 };

```



by Base()

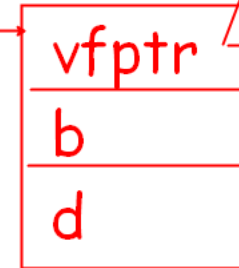
by Derived()

Derived::vftable

```

67 class Derived : public Base {
68 public:
69 int d;
70 virtual void f2() { ... }
71 virtual void f4() { ... }
72 void f5() { ... }
73 };

```



```

74
75 Base *p = new Derived;
76 p->f2();
77 p->f1();
78 p->f3();

```

Base \*ptr = dobj;

ptr-&gt;f2(); // late binding

ptr-&gt;f3(); // early binding

dobj.f2(); // early binding

1. jump to addr & read first member - vptr
2. go to the vtable.
3. call fn at the index of f.

e. vptr of Base is inherited to the Derived.

return 0;

}

...

\* \*\*Multiple Inheritance:\*\*

\* In multiple inheritance, where a class has multiple direct base classes, each base class will have its own virtual function table pointer (`vptr`) and all these vptr will be inherited to the Derived class.

\* Example:

```cpp

class Base1 {

public:

virtual void foo() {}

};

class Base2 {

public:

virtual void bar() {}

};

class Derived : public Base1, public Base2 {

public:

virtual void baz() {}

};

Base1 obj;

Base2 obj;

Derived obj;

vptr

8 bytes

vptr

8 bytes

vptr

16 bytes

B1::foo()

Base1::vtable

B2::bar()

Base2::vtable

B1::foo()

D::baz()

Derived::vtable for B1

B2::bar()

Derived::vtable for Base2

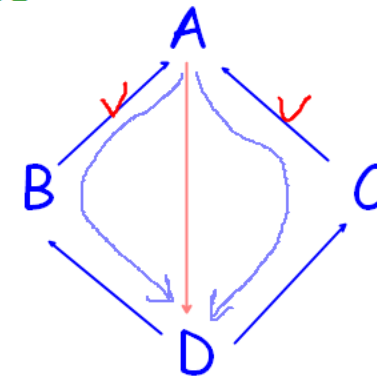
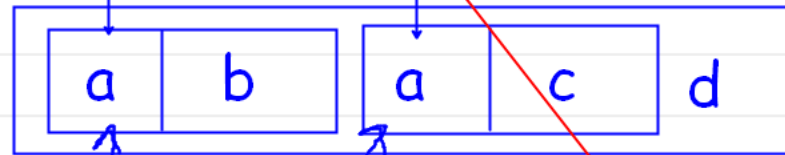
```
80 // -----  
81 class A -- a  
82  
83 class B: A -- b  
84  
85 class C: A -- c  
86  
87 class D: B, C -- d  
88  
89 D obj;  
90
```

obj.a = 10;

ambiguity...

obj::A = 10;

multiple copies ??



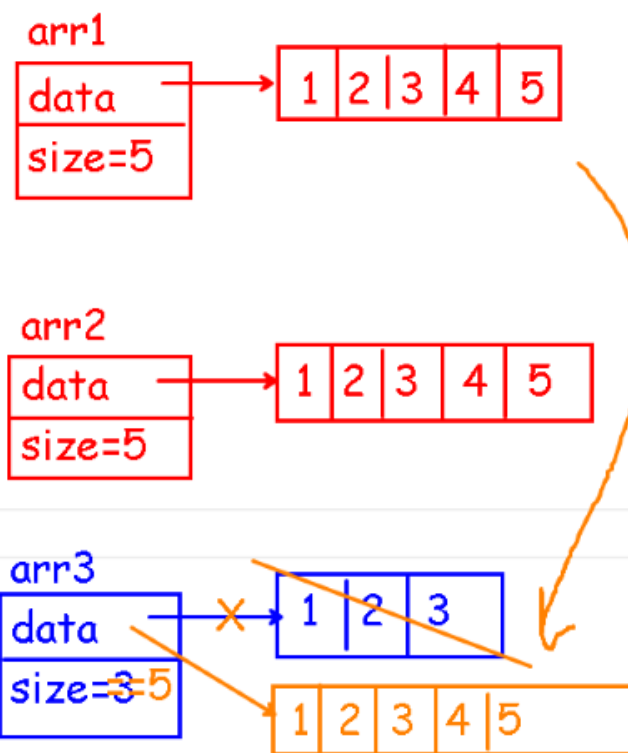
Virtual Inheritance



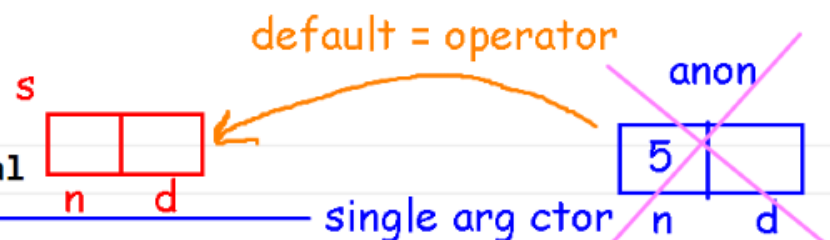
obj.a = 20; // no ambiguity⁺

```
443 class DynamicArray {
444 private:
445     int* data;
446     int size;
447 public:
448     DynamicArray(int sz) {
449         size = sz;
450         data = new int[size];
451         for (int i = 0; i < size; ++i)
452             data[i] = i + 1;
453     }
454     DynamicArray(const DynamicArray& other) {
455         size = other.size;
456         data = new int[size];
457         for (int i = 0; i < size; ++i)
458             data[i] = other.data[i];
459     }
460     DynamicArray& operator=(const DynamicArray& other) {
461         if (this != &other) {
462             delete[] data;
463             size = other.size;
464             data = new int[size];
465             for (int i = 0; i < size; ++i)
466                 data[i] = other.data[i];
467         }
468     }
469 }
```

DynamicArray arr1(5);
DynamicArray arr2(arr1);
DynamicArray arr3(3); ✓
arr3 = arr1; ✓




```
511 }
512 Rational(int num) {
513     numerator = num;
514     denominator = 1;
515 }
516 operator double() const {
517     return (double)numerator / denominator;
518 }
519 private:
520     int numerator;
521     int denominator;
522 };
523
524 int main() {
525     Rational r(3, 2);
526     double d = r; // Implicit conversion to double
527     // d = r.operator double();
528     int x = 5;
529     Rational s;
530     s = x; // Implicit conversion from int to Rational
531     // x --> anonymous Rational object
532     // s = anonymous object (assignment)
533     // anonymous object destroyed
534     return 0;
```





THANK YOU

NILESH GHULE <NILESH@SUNBEAMINFO.COM>

