

Object Oriented Programming – C++

Chapter 5: Pointers in C++

Exam-Oriented Questions and Answers

Section A: Short Questions and Answers

The following short-answer questions are frequently asked in examinations on this chapter. Each answer is written concisely for quick revision.

Q1. What is a pointer in C++?

Ans. A pointer is a variable that stores the memory address of another variable instead of storing a data value directly. It is declared using the data type it points to, followed by an asterisk, e.g., `int *p;`

Q2. What is the difference between the & and * operators when used with pointers?

Ans. The & operator (address-of operator) returns the memory address of a variable, and is used to initialise a pointer, e.g., `p = &x;`. The * operator (dereference operator) accesses the value stored at the address held by a pointer, e.g., `cout << *p;`

Q3. What is a null pointer and why is it used?

Ans. A null pointer is a pointer that has been assigned the value `nullptr` (or `NULL`), meaning it does not point to any valid memory location. It is used to indicate that a pointer is not currently pointing to a valid object, and should always be checked before dereferencing to avoid undefined behaviour.

Q4. How does pointer arithmetic differ from ordinary integer arithmetic?

Ans. When a pointer is incremented or decremented, it moves by `sizeof(type)` bytes rather than by one byte, so that it correctly steps to the next or previous element of the type it points to. Multiplying or dividing two pointers is not allowed; only addition/subtraction with an integer, and subtraction between two pointers of the same type, are meaningful.

Q5. Differentiate between call by value and call by reference.

Ans. In call by value, a copy of the argument is passed, so changes inside the function do not affect the original variable. In call by reference (implemented using pointers or reference variables), the address of the argument is passed, so the function can directly modify the original variable.

Q6. What is the size of a pointer variable in C++?

Ans. The size of a pointer variable does not depend on the data type it points to; it depends on the system architecture, and is typically 4 bytes on a 32-bit system and 8 bytes on a 64-bit system, since a pointer only stores an address.

Q7. What is the 'this' pointer?

Ans. 'this' is an implicit pointer available inside every non-static member function of a class, holding the address of the object through which that member function was called. It is used to distinguish member variables from parameters of the same name and to return the current object from a member function.

Q8. Why can the 'this' pointer not be used inside a static member function?

Ans. A static member function belongs to the class as a whole and is not invoked on any specific object, so there is no calling object whose address 'this' could hold. Hence 'this' is meaningful only inside non-static member functions.

Q9. Can a base-class pointer point to a derived-class object? Explain briefly.

Ans. Yes. Since a derived-class object contains a complete base-class sub-object within it, a base-class pointer can point to a derived-class object without any explicit cast. This is the foundation of runtime polymorphism, especially when the base class declares virtual functions.

Q10. What is the main difference between new/delete and malloc()/free()?

Ans. new and delete are C++ operators that automatically call the constructor and destructor respectively, return the correct pointer type without a cast, and throw bad_alloc on failure. malloc() and free() are C library functions that return void*, require an explicit cast, do not call constructors/destructors, and return NULL on failure.

Q11. What is the difference between malloc() and calloc()?

Ans. malloc(size) allocates a single block of size bytes without initialising it, so the memory contains garbage values. calloc(n, size) allocates memory for n elements of size bytes each and initialises every byte to zero.

Q12. What is a dangling pointer?

Ans. A dangling pointer is a pointer that continues to point to a memory location after that memory has been freed (using delete or free()) or after the variable it pointed to has gone out of scope. Dereferencing a dangling pointer leads to undefined behaviour, so such pointers should be set to nullptr immediately after the memory is released.

Q13. What happens if delete is used on memory allocated with new[]?

Ans. Using delete instead of delete[] on an array allocated with new[] leads to undefined behaviour, because delete only calls the destructor for a single object and does not correctly walk through all the elements of the array. The array form of allocation must always be released with the matching array form delete[].

Q14. What is the arrow operator (->) used for?

Ans. The arrow operator is used to access a member of a class through a pointer to an object, and is equivalent to first dereferencing the pointer and then applying the dot operator, i.e., ptr->member is the same as (*ptr).member.

Q15. What are pointer-to-member operators?

Ans. The operators `.*` and `->*` allow a member (data or function) of a class to be accessed dynamically through a separate pointer-to-member variable, rather than a fixed member name. `.*` is used with an object, and `->*` is used with a pointer to an object.

Section B: Broad / Descriptive Questions and Answers

The following long-answer questions require detailed explanation, supported by programs, and are typically asked for 5–10 marks.

Q1. Explain the concept of pointers in C++ with pointer declaration, the address operator, and the dereference operator. Support your answer with a program.

Ans. A pointer is a variable that stores the memory address of another variable rather than a data value. It is declared by writing the data type of the variable it will point to, followed by an asterisk and the pointer's name, for example `int *p`. This tells the compiler that `p` will hold the address of an integer variable, and that whenever `p` is dereferenced, the value at that address should be interpreted as an `int`.

The address operator (`&`), when placed before a variable name, returns that variable's memory address, and is used to initialise a pointer. The dereference operator (`*`), when placed before a pointer variable, accesses the value stored at the address the pointer holds; this operation is called dereferencing. These two operators are inverses of each other: `&x` gives an address, and `*(&x)` gives back the value of `x`.

```
#include <iostream>
using namespace std;
int main() {
    int x = 25;
    int *p = &x;    // p stores address of x
    cout << "Value of x: " << x << endl;
    cout << "Address of x: " << &x << endl;
    cout << "Value via pointer *p: " << *p << endl;
    *p = 50;        // modify x through pointer
    cout << "New value of x: " << x << endl;
    return 0;
}
```

Output: Value of x: 25
Address of x: 0x61ff08 (varies)
Value via pointer *p: 25
New value of x: 50

Q2. Differentiate between call by value and call by reference. Explain call by reference using pointers with a suitable program.

Ans. Call by value passes a copy of the actual argument to the function's parameter; any changes made inside the function affect only this local copy, leaving the original variable in the caller unchanged. Call by reference, on the other hand, passes the address of the actual argument, so the function receives a pointer to the original variable and can read or modify it directly.

Call by reference is essential whenever a function needs to change more than one value, return multiple results, or avoid the overhead of copying a large structure or array. The following program swaps two numbers using call by reference with pointers:

```
#include <iostream>
using namespace std;
void swap(int *a, int *b) {
```

```

    int temp = *a;
    *a = *b;
    *b = temp;
}
int main() {
    int x = 10, y = 20;
    cout << "Before swap: x=" << x << " y=" << y << endl;
    swap(&x, &y); // pass addresses
    cout << "After swap: x=" << x << " y=" << y << endl;
    return 0;
}

```

Output: Before swap: x=10 y=20
After swap: x=20 y=10

Q3. Explain pointer arithmetic in detail. How is a pointer related to an array? Illustrate with a program that traverses an array using a pointer.

Ans. Pointer arithmetic in C++ is scaled automatically according to the size of the data type the pointer points to, rather than operating in raw bytes. When a pointer *p* of type *T* is incremented (*p++*), it advances by `sizeof(T)` bytes so that it points to the next element of type *T* in memory. Similarly, *p--* moves it back by `sizeof(T)` bytes. Only addition/subtraction of an integer to a pointer, and subtraction of two pointers of the same type (which yields the number of elements between them), are permitted; multiplying or dividing pointers is meaningless and not allowed.

This behaviour connects directly to arrays: the name of an array, when used in an expression, decays into a pointer to its first element. As a result, array elements can be accessed either using subscript notation `arr[i]` or using pointer notation `*(arr + i)`; both forms are equivalent and the compiler treats them identically.

```

#include <iostream>
using namespace std;
int main() {
    int arr[5] = {10, 20, 30, 40, 50};
    int *p = arr; // p points to arr[0]
    for (int i = 0; i < 5; i++) {
        cout << *(p + i) << " "; // equivalent to arr[i]
    }
    cout << endl;
    return 0;
}

```

Output: 10 20 30 40 50

Q4. What is an object pointer? Explain the 'this' pointer with an example showing how it can be used for method chaining.

Ans. An object pointer is a pointer that stores the address of an object of a class, declared using the class name followed by an asterisk, e.g., `ClassName *p = &obj;`. Its members are accessed using the arrow operator (`->`), which is equivalent to first dereferencing the pointer and then applying the dot operator.

Every non-static member function receives an implicit pointer called 'this', which holds the address of the object that invoked the function. Besides distinguishing a member from a parameter of the same name, 'this' can be

dereferenced with `*this` to return the current object by reference, allowing several member function calls to be chained together on the same line, as shown below:

```
#include <iostream>
using namespace std;
class Box {
    int length, width;
public:
    Box& setLength(int l) { this->length = l; return *this; }
    Box& setWidth(int w) { this->width = w; return *this; }
    void display() { cout << length << " x " << width << endl; }
};
int main() {
    Box b;
    b.setLength(10).setWidth(5).display(); // method chaining
    return 0;
}
```

Output: 10 x 5

Q5. Explain how a base-class pointer pointing to a derived-class object achieves runtime polymorphism. Support your answer with a program.

Ans. Because a derived-class object always contains a complete base-class sub-object, a pointer of the base-class type can point to an object of the derived class without any explicit conversion. If the function being called is declared virtual in the base class, C++ resolves the call at run time based on the actual type of the object being pointed to, rather than the declared type of the pointer. This mechanism is called dynamic binding, and it is the basis of runtime polymorphism.

```
#include <iostream>
using namespace std;
class Animal {
public:
    virtual void sound() { cout << "Some generic sound" << endl; }
};
class Dog : public Animal {
public:
    void sound() override { cout << "Dog barks" << endl; }
};
class Cat : public Animal {
public:
    void sound() override { cout << "Cat meows" << endl; }
};
int main() {
    Animal *a;
    Dog d; Cat c;
    a = &d; a->sound(); // Dog barks
    a = &c; a->sound(); // Cat meows
    return 0;
}
```

Output: Dog barks
Cat meows

Without the keyword `virtual` in the base class, the compiler would instead bind the call at compile time based on the pointer's declared type (`Animal`), and both calls would incorrectly print "Some generic sound".

Q6. Compare `new/delete` with `malloc()/calloc()/free()` in detail. Why are `new` and `delete` preferred in C++ programs?

Ans. `new` and `delete` are operators built into C++, while `malloc()`, `calloc()`, and `free()` are library functions inherited from C, declared in `<cstdlib>`. `new` returns a pointer of the exact type requested, so no explicit cast is needed, whereas `malloc()/calloc()` return `void*`, which must be cast to the required type. `new` automatically invokes the constructor of an object, and `delete` automatically invokes its destructor; `malloc()/calloc()/free()` know nothing about constructors or destructors and simply manage raw memory.

On failure, `new` throws a `bad_alloc` exception, while `malloc()/calloc()` return `NULL`, which the programmer must check manually. `calloc()` additionally zero-initialises the allocated memory, while `malloc()` leaves it uninitialised, and `new` does not zero-initialise memory for built-in types unless value-initialisation syntax is used. Because of automatic constructor/destructor handling and type safety, `new` and `delete` are strongly preferred over `malloc()/calloc()/free()` for allocating class objects in C++, and the two families should never be mixed on the same block of memory (i.e., never `free()` memory obtained from `new`, and never `delete` memory obtained from `malloc()`).

```
int *p1 = new int(5);           // C++ style, constructor-aware
delete p1;

int *p2 = (int*) malloc(sizeof(int)); // C style, needs cast
*p2 = 5;
free(p2);
```

Q7. What are dangling pointers and memory leaks? Explain their causes and how they can be avoided.

Ans. A dangling pointer is a pointer that still holds the address of a memory location after that memory has already been released — for example, after `delete` or `free()` has been called on it, or after a local variable it pointed to has gone out of scope. Dereferencing a dangling pointer produces undefined behaviour, since the memory may have been reused for something else. This is avoided by setting a pointer to `nullptr` immediately after freeing its memory, and by never returning the address of a local variable from a function.

A memory leak occurs when memory allocated dynamically with `new` or `malloc()` is never released with the matching `delete` or `free()`, so the memory remains reserved for the entire lifetime of the program even though it is no longer needed. Repeated leaks gradually exhaust available memory. Leaks are avoided by ensuring that every successful `new` has a corresponding `delete` (and every `new[]` a corresponding `delete[]`), and by using care when a pointer is overwritten or a function returns early, since either can cause the original address to be lost before it is freed.

Q8. Explain the four member-access/dereferencing operators in C++ (`.`, `->`, `.*`, `->*`) with examples.

Ans. C++ provides four operators for accessing members of a class, depending on whether an object or a pointer is used, and whether the member itself is fixed or selected dynamically through a pointer-to-member variable. The dot operator (`.`) accesses a member directly through an object, as in `obj.member`. The arrow operator (`->`) accesses a member through a pointer to an object, as in `ptr->member`, and is equivalent to `(*ptr).member`.

The pointer-to-member operators are used less frequently: `.*` accesses a member through an object when the member itself is stored in a separate pointer-to-member variable (`obj.*ptrToMember`), and `->*` does the same when a pointer to the object is available instead of the object itself (`objPtr->*ptrToMember`). These are useful when the specific member to be accessed needs to be decided at run time rather than being hard-coded.

```
#include <iostream>
using namespace std;
class Test {
public:
    int val = 42;
};
int main() {
    Test t;
    Test *pt = &t;
    int Test::*pm = &Test::val;    // pointer to member

    cout << t.val << endl;        // dot operator
    cout << pt->val << endl;       // arrow operator
    cout << t.*pm << endl;        // pointer-to-member via object
    cout << pt->*pm << endl;       // pointer-to-member via pointer
    return 0;
}
```

Output: 42

42

42

42

Q9. What is a pointer to pointer? Explain with a program how multiple levels of indirection work.

Ans. A pointer to pointer is a pointer that stores the address of another pointer, rather than the address of an ordinary variable. It is declared using two asterisks, e.g., `int **pp`. Dereferencing it once (`*pp`) gives the address stored in the first pointer, and dereferencing it twice (`**pp`) gives the actual value of the original variable. This concept, called multiple levels of indirection, is useful in dynamic two-dimensional arrays, arrays of strings, and certain linked data structures.

```
#include <iostream>
using namespace std;
int main() {
    int x = 100;
    int *p = &x;
    int **pp = &p;
    cout << "x = " << x << endl;
    cout << "*p = " << *p << endl;
    cout << "**pp = " << **pp << endl;
    return 0;
}
```

Output: x = 100

*p = 100

**pp = 100

Q10. Explain dynamic memory allocation for arrays using new[] and delete[], with a program that reads values into a dynamically allocated array and prints their sum.

Ans. When the size of an array is not known until run time, memory for it can be allocated dynamically on the heap using new[], which returns a pointer to the first element of the newly allocated block. This memory must later be released using delete[] (not plain delete), since the array form of new must always be matched with the array form of delete to ensure that memory for every element is correctly released.

```
#include <iostream>
using namespace std;
int main() {
    int n;
    cout << "Enter number of elements: ";
    cin >> n;
    int *arr = new int[n];    // dynamic array
    int sum = 0;
    for (int i = 0; i < n; i++) {
        cin >> arr[i];
        sum += arr[i];
    }
    cout << "Sum = " << sum << endl;
    delete[] arr;            // release array memory
    return 0;
}
```

Output: Enter number of elements: 3
10 20 30
Sum = 60

Section C: Coding Questions and Answers

The following programming questions are commonly asked to test the practical application of pointer concepts. Complete programs with expected output are given for each.

Q1. Write a C++ program to find the sum and difference of two numbers using pointers.

```
#include <iostream>
using namespace std;
int main() {
    int a = 30, b = 10;
    int *p1 = &a, *p2 = &b;
    cout << "Sum = " << (*p1 + *p2) << endl;
    cout << "Difference = " << (*p1 - *p2) << endl;
    return 0;
}
```

Output: Sum = 40

Difference = 20

Q2. Write a program to find the largest of three numbers using pointers and functions (call by reference).

```
#include <iostream>
using namespace std;
void findLargest(int *a, int *b, int *c, int *largest) {
    *largest = *a;
    if (*b > *largest) *largest = *b;
    if (*c > *largest) *largest = *c;
}
int main() {
    int a = 15, b = 42, c = 27, largest;
    findLargest(&a, &b, &c, &largest);
    cout << "Largest = " << largest << endl;
    return 0;
}
```

Output: Largest = 42

Q3. Write a program that dynamically allocates memory for a Student object using new, sets its data through a pointer, and releases the memory using delete.

```
#include <iostream>
using namespace std;
class Student {
public:
    int roll;
    Student(int r) : roll(r) { cout << "Constructor called" << endl; }
    ~Student() { cout << "Destructor called" << endl; }
    void display() { cout << "Roll No: " << roll << endl; }
};
int main() {
    Student *s = new Student(101); // dynamic object
    s->display();
    delete s;                      // destructor runs, memory freed
}
```

```
    return 0;
}
```

Output: Constructor called

Roll No: 101

Destructor called

Q4. Write a program to reverse the elements of an array using pointer arithmetic (without using array subscripts).

```
#include <iostream>
using namespace std;
int main() {
    int arr[5] = {1, 2, 3, 4, 5};
    int *start = arr;
    int *end = arr + 4;          // points to last element
    while (start < end) {
        int temp = *start;
        *start = *end;
        *end = temp;
        start++;
        end--;
    }
    for (int i = 0; i < 5; i++) cout << arr[i] << " ";
    cout << endl;
    return 0;
}
```

Output: 5 4 3 2 1

Q5. Write a program using malloc() and calloc() to allocate memory for an integer array, and compare their initial contents.

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main() {
    int *a = (int*) malloc(3 * sizeof(int)); // uninitialised
    int *b = (int*) calloc(3, sizeof(int));  // zero-initialised
    cout << "calloc values: ";
    for (int i = 0; i < 3; i++) cout << b[i] << " ";
    cout << endl;
    free(a);
    free(b);
    return 0;
}
```

Output: calloc values: 0 0 0

(malloc's array 'a' would contain unpredictable garbage values)

Q6. Predict the output of the following program and justify your answer.

```
#include <iostream>
using namespace std;
class Base {
public:
    void show() { cout << "Base show" << endl; }
```

```
        virtual void greet() { cout << "Base greet" << endl; }
};
class Derived : public Base {
public:
    void show() { cout << "Derived show" << endl; }
    void greet() override { cout << "Derived greet" << endl; }
};
int main() {
    Base *b = new Derived();
    b->show();    // ?
    b->greet();   // ?
    delete b;
    return 0;
}
```

Output: *Base show*
Derived greet

Explanation: show() is not virtual, so the call is resolved at compile time based on the pointer's declared type (Base), giving "Base show". greet() is virtual, so it is resolved at run time based on the actual object type (Derived), giving "Derived greet". This demonstrates the difference between static binding and dynamic binding through a base-class pointer.