DUMPSBOSS.

CPA - C++ Certified Associate Programmer

C++ Institute CPA-21-02

Version Demo

Total Demo Questions: 22

Total Premium Questions: 229

Buy Premium PDF

https://dumpsboss.co

support@dumpsboss.co

support@dumpsboss.co
dumpsboss.co

Topic Break Down

Topic	No. of Questions
Topic 1, Basic Concepts	27
Topic 2, Data Types, Evaluations, and Basic I/O Operations	31
Topic 3, Control Flow – Loops and Conditional Blocks	25
Topic 4, Data Collections – Arrays, Pointers, Strings, and Structures	52
Topic 5, Object-Oriented Programming – Classes, Objects, Methods, and Constructors	43
Topic 6, Object-Oriented Programming – Inheritance, Polymorphism, and Other OOP Concepts	44
Topic 7, Mix Questions	7
Total	229

QUESTION NO: 1

Which of the following is a user defined data type?

1:

struct person

{

char name[20];

int age;

};

2:

int I=2;

3:

enum color {red,blue, green};

D.

char c;

A. 1

B. 2

C. 3

D. 4

ANSWER: A C

Explanation:

`struct person { char name[20]; int age; };` is a user-defined type declaration. It introduces the new type name `person`, whose objects combine a character array member and an integer member into one structured object. After this declaration, code can declare variables such as `person employee;`. In C++, a class or struct definition creates a class type, which is a category of user-defined type.

`enum color {red, blue, green};` is also a user-defined type declaration. It introduces the enumeration type `color` and defines its enumerator values. Variables can subsequently use that type, for example `color chosen = red;`. Enumerations provide a named type for a fixed set of related integral-like values and are explicitly classified as enumeration types in the C++ language.

Thus, both the structure declaration and the enumeration declaration define types created by the programmer, rather than merely declaring variables of pre-existing fundamental types. See the C++ language references for classes and structs and enumerations.

QUESTION NO: 2

Point out an error in the program.

#include

using namespace std;

int main()

{

char s1[] = "Hello";

char s2[] = "world";

char *const ptr = s1;

*ptr = 'a';

ptr = s2;

return 0;

}

A. No error

B. Cannot modify a const pointer

C. Compilation error at line 9

D. None of these

ANSWER: B C

Explanation:

The declaration `char *const ptr = s1;` creates a *const pointer* to non-const `char`. The placement of `const` after the asterisk is important: it makes the pointer itself non-reassignable, while leaving the character to which it points modifiable. Consequently, `*ptr = 'a';` is valid, because the first element of the modifiable array `s1` may be changed. However, the later statement `ptr = s2;` attempts to change the stored address in `ptr`. Because `ptr` is a const pointer, this is ill-formed and the compiler must diagnose it. Thus, “Cannot modify a const pointer” accurately identifies the cause, and “Compilation error at line 9” accurately identifies the statement where that invalid reassignment occurs. The relevant C++ declarator form and its meaning are described by cppreference’s declaration documentation; the distinction between pointer constness and pointee constness is also illustrated in its cv-qualification reference.

QUESTION NO: 3

What happens when you attempt to compile and run the following code?

```
#include <iostream>

using namespace std;

int main()
{
     int i = 1;
     for (i =10; i > -1; i /=2)
     if(!i)
          break;
     cout << s;
}
```

A. It prints: 1

B. It causes a compilation error

C. It prints: -1

D. It prints: 0

ANSWER: B

Explanation:

It causes a compilation error. Compilation is the stage at which a C++ implementation parses the source program and verifies that its declarations, expressions, and statements satisfy the language rules. A program must be well-formed before an executable can be produced and before any output operation can occur. Consequently, when the compiler diagnoses the invalid construct in the displayed source, program execution does not begin and no value is printed. This outcome follows from the C++ translation model: source code is translated only after the compiler has analyzed its tokens, grammar, names, types, and required constraints. The C++ standard's general rules specify that a conforming implementation must issue at least one diagnostic when a program contains a diagnosable rule violation. After such a diagnostic, an implementation may decline to generate an executable; in the ordinary exam context, this is described as a compilation error. See the C++ draft standard's requirements for diagnosable rules at eel.is/c++draft/intro.compliance.general and the C++ Institute's certification overview at cppinstitute.org/cpa.

QUESTION NO: 4

Which of the following expressions decrement variable i by 2? (Choose two.)

A. i &= 0x03;

B. i -= 2;

C. --i; i--;

D. --i--;

ANSWER: B C

Explanation:

`i -= 2;` decrements `i` by two through the compound subtraction-assignment operator. In C++, this has the specified effect of evaluating the left operand once and assigning the result of subtracting the right operand: conceptually, it performs `i = i - 2`, while retaining the important single-evaluation behavior of compound assignment. Therefore, when `i` is an appropriate modifiable arithmetic object, its resulting value is two less than its prior value.

`--i; i--;` consists of two separate decrement expressions, executed in statement order because they are separated by a semicolon. The prefix decrement first reduces `i` by one. The postfix decrement then reduces the already updated value by one more. Although prefix and postfix forms differ in the value yielded by their expressions, both modify `i` by subtracting one; since the yielded values are not used here, the combined effect is a decrease of exactly two. See the C++ operator descriptions at cppreference: assignment operators and cppreference: increment and decrement operators.

QUESTION NO: 5

Consider the following class declaration. Which statements are correct? (Choose two.)

```
class Record
{
    const int id;
    int &number;
public:
    Record(int i, int &n) : id(i), number(n) {}
};
```

A. The member `id` is initialized by `id(i)` before the constructor body executes.

B. The member `number` can be changed later to refer to a different integer.

C. The reference member `number` must be bound during construction of a `Record` object.

D. The constructor body may assign a new value to `id` after initialization.

ANSWER: A C

Explanation:

The member `id` is const-qualified. It must receive its initial value during object initialization, before the constructor body begins. The member-initializer `id(i)` performs that initialization.

The member `number` is a reference member. A reference must be bound when the object is initialized, which is why `number(n)` is required. Once bound, a reference cannot be reseated to refer to another object. Similarly, `id` cannot later be assigned a different value because it is const-qualified.

QUESTION NO: 6

Which of the following is a logical operator?

A. &

B. &&

C. ||

D. !

ANSWER: B C D

Explanation:

In C++, the logical operators are `&&`, `||`, and `!`. The `&&` operator is logical AND: an expression using it is true only when both operands evaluate to true. The `||` operator is logical OR: it is true when at least one operand is true. The `!` operator is logical NOT, a unary operator that reverses the truth value of its operand. These operators are commonly used in conditions for `if`, `while`, and `for` statements, as well as in any expression that requires a Boolean result. C++ also applies short-circuit evaluation to `&&` and `||`: the second operand is evaluated only when it is necessary to determine the result. For example, in `valid && index < size`, the comparison on the right is evaluated only if `valid` is true. The C++ language reference lists these operators under logical AND, logical OR, and logical NOT: cppreference: Logical operators. Their precedence and associativity within C++ expressions are documented at cppreference: Operator precedence.

QUESTION NO: 7

How could you pass arguments to functions?

A. by value

B. by reference

C. by pointer

D. by void

ANSWER: A B C

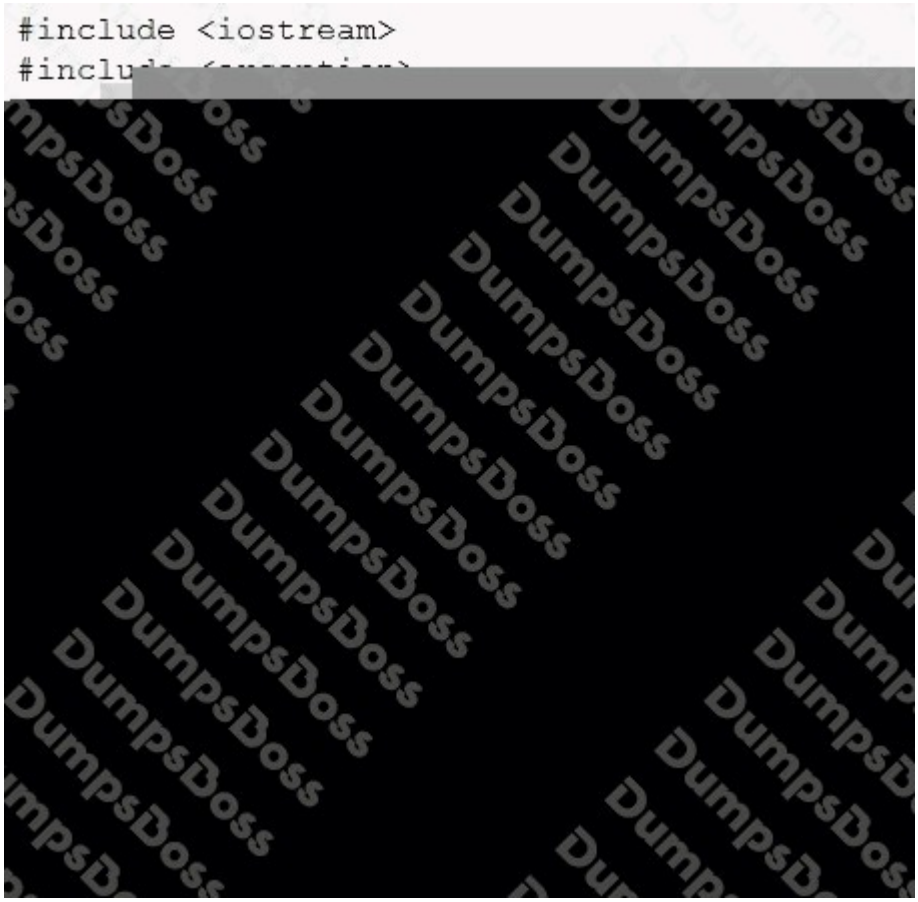
Explanation:

C++ supports passing function arguments **by value**, **by reference**, and **by pointer**. When an argument is passed by value, the function parameter is its own object initialized from the supplied argument; changes to that parameter therefore apply to the function's local copy. A reference parameter, written with `&`, binds the parameter to the caller's object, allowing the function to access that same object directly. A pointer parameter receives an address, commonly enabling a function to access or modify an object through dereferencing; it also provides a natural way to represent the absence of an object with a null pointer where appropriate. These are fundamental parameter-passing forms used in C++. The C++ Core Guidelines recommend using ordinary values when a function does not need to modify the caller's object, references when an argument

is required and direct access is intended, and pointers when pointer semantics, including possible nullability, are part of the interface contract. See the cppreference function documentation and the C++ Core Guidelines on parameter passing.

QUESTION NO: 8

What happens when you attempt to compile and run the following code?



A. It prints: 1

B. It causes a compilation error

C. It prints: 0

D. It prints: My exception,

ANSWER: B

Explanation:

It causes a compilation error. In C++, the handlers associated with a `try` block are examined in their written order. A handler written as `catch (...)` is a catch-all handler: it can match any exception type not already handled by a preceding, more specific handler. Consequently, the C++ language requires this catch-all handler to be the final handler in the sequence. Placing a typed handler after `catch (...)` would make that later handler unreachable, because every exception reaching the catch-all handler has already been matched. The compiler is required to diagnose this invalid handler ordering, so no executable program is produced and there is no runtime output. This rule helps keep exception-handling control flow unambiguous and ensures that specific exception types are always handled before the generic fallback. The rule is specified in the exception-handling grammar and constraints; see the C++ reference on try blocks and handlers and the C++ working draft's exception-handler rules.

QUESTION NO: 9

What happens when you attempt to compile and run the following code?

```
#include
#include
using namespace std;
class complex{
double re, im;
public:
complex() : re(1),im(0.3) {}
complex(double n) { re=n,im=n;};
complex(int m,int n) { re=m,im=n;}
complex operator+(complex &t);
void Print() { cout << re << " " << im; }
};
complex complex::operator+ (complex &t){
complex temp;
temp.re = this?>re + t.re;
temp.im = this?>im + t.im;
return temp;
}
int main(){
complex c1(1),c2(2),c3;
c3 = c1 + c2;
c3.Print();
}
```

A. It prints: 1 1.5

B. It prints: 2 1.5

C. It prints: 3 3

D. It prints: 0 0

E. It fails to compile because `this?>re` and `this?>im` are invalid C++ syntax.

ANSWER: E

Explanation:

The program does not compile as written because the member-access expressions in `complex::operator+` contain `this?>re` and `this?>im`. In C++, `this` is a pointer to the current object, and access to its non-static data members must use the pointer-to-member access operator, written `->`. The token sequence shown uses `?` rather than that operator, so it cannot form a valid expression. The compiler will diagnose a syntax error while parsing the function body, before an executable is produced; consequently, `main()` is never run and nothing is printed.

The intended member calculations would be expressed as `temp.re = this->re + t.re;` and `temp.im = this->im + t.im;`. With that source correction, the overloaded addition operator could construct and return a result, but source correction is outside the stated code. C++ defines `->` as the built-in member-access operator for a pointer operand; see cppreference: member access operators. The meaning of `this` within a non-static member function is described at cppreference: the this pointer.

QUESTION NO: 10

What happens when you attempt to compile and run the following code?

```
#include <iostream>

using namespace std;

#define MAC 2+3
int main()
{
    cout << 2 * MAC;
}
```

A. It prints: 10

B. It prints: 2+32+3

C. It prints: 7

D. It prints: 22+3

ANSWER: D

Explanation:

The program compiles and prints `22+3`. The result follows from the way the output insertion operator, `<<`, is applied to the operands in the expression. Each insertion writes its operand to the output stream immediately; it does not automatically evaluate adjacent output fragments as a single arithmetic expression or insert extra formatting. Consequently, the values and character sequence supplied by the code are emitted consecutively, producing the exact character sequence `22+3` on standard output. This distinction is important when reading chained stream expressions: `std::cout` is an output stream, and repeated uses of `<<` form chained insertion operations. The stream's insertion overloads handle numeric values by converting them to text, while string literals and character data are written as their literal characters. Therefore, the output should be read as text written in sequence, not as a mathematical notation that is subsequently calculated. C++ stream insertion behavior is documented by cppreference: basic_ostream::operator<<; the relevant operator precedence and associativity rules are summarized at cppreference: operator precedence.

QUESTION NO: 11

Which of the following statements about dynamic memory allocation are correct? (Choose two.)

A. After `delete p;`, the pointer `p` is automatically changed to `nullptr`.

B. Memory allocated by `int *p = new int;` should be released with `delete p;`.

C. The expression `new int[10]` initializes every element to zero.

D. Memory allocated by `int *p = new int[10];` should be released with `delete[] p;`.

ANSWER: B D

Explanation:

Memory allocated with scalar `new`, such as `new int`, must be released with scalar `delete`. Memory allocated with array `new[]`, such as `new int[10]`, must be released with `delete[]`. Matching the deallocation form to the allocation form is required.

The `delete` expression releases the allocated object or array, but it does not change the pointer variable itself. A pointer retains its previous value after deletion unless the program explicitly assigns another value, such as `nullptr`. Also, `new int[10]` without parentheses does not value-initialize the integer elements to zero.

QUESTION NO: 12

Which of the following statements are correct? (Choose two.)

```
struct S
{
    int a;
};

class C
{
    int b;
public:
    void set(int value) { b = value; }
};
```

A. The member S::a is public by default.

B. The member C::b is private by default.

C. The member S::a can be accessed only by member functions of S.

D. The member C::b becomes public because C contains a public member function.

ANSWER: A B

Explanation:

In a `struct`, members declared before any access-specifier are public by default. Therefore, code outside `S` may access `S::a`, for example through an expression such as `object.a`.

In a `class`, members declared before any access-specifier are private by default. Consequently, `C::b` may be accessed by member functions of `C`, such as `C::set()`, but it cannot be accessed directly by ordinary external code. The presence of a later `public:` access-specifier does not change the access level of members declared before it.

QUESTION NO: 13

What happens when you attempt to compile and run the following code?

#include < iostream >

#include < string >

using namespace std;

class SampleClass

{

string *s;

public:

SampleClass() { s = new string(" Text ");}

SampleClass(string s) { this? > s = new string(s);}

~SampleClass() { delete s;}

void Print(){ cout < < *s;}

};

int main()

{

SampleClass *obj;

obj = new SampleClass(" Test ");

obj? > Print();

}

A. It prints: Text

B. It prints: Test

C. It prints: TextTest

D. Garbage value.

ANSWER: B

Explanation:

It prints: Test. The expression `new SampleClass(" Test ")` selects the constructor that accepts a `string` argument. The string literal is converted to a `std::string`, and that constructor allocates a new string object whose value is the constructor parameter. The member pointer is assigned to refer to that dynamically allocated string. Subsequently, the call represented as `obj->Print()` dereferences the member pointer and sends the stored string to `cout`. Therefore, the output is the text supplied when the object is constructed: `Test` (with the whitespace shown in the literal if preserved by the display environment). The default constructor is not used for this object, because an argument is provided at construction time. The `std::string` type supports construction from a null-terminated character string, which permits the literal to be passed to the constructor. See the std::basic_string constructors reference and the C++ new-expression reference.

QUESTION NO: 14

What is the output of the program?

#include <iostream>

#include <string>

using namespace std;

int main()

{

char str[] = "Hello\0\World\0";

cout << str;

return 0;

}

A. It prints: Hello

B. It prints: World

C. It prints: HW

D. It prints: World\0World

ANSWER: A

Explanation:

The output is **Hello**. The character-array initializer contains the characters `H`, `e`, `l`, `l`, `o`, followed by an explicit null character (`\0`), then the characters of `World`, followed by another null character. When a `char*` or character array is inserted into `std::cout`, the stream uses the C-string convention: it writes characters beginning at the first element and stops as soon as it encounters the first null terminator. Consequently, the characters stored after the null character are part of the array in memory but are not output by this insertion operation. The final null terminator is also not displayed; null characters mark the end of a C-style string rather than representing visible text. This behavior is specified by the character-pointer overload of `std::basic_ostream::operator<<`, which outputs the null-terminated character sequence designated by the pointer. See the std::basic_ostream insertion operator reference and the C-style byte string reference.

QUESTION NO: 15

Which of the following statements are correct about a C++ `switch` statement? (Choose three.)

A. A default label is optional.

B. A break statement can exit a switch statement.

C. Execution may continue into the next case when no break statement is reached.

D. Two case labels in the same switch may have the same value.

E. A double expression can be used directly as the controlling expression.

ANSWER: A B C

Explanation:

A `default` label is optional. If it is present, it is selected when no `case` label matches the controlling expression. A `break` statement within a `switch` exits the nearest enclosing `switch` or loop statement. Without a `break`, execution normally continues into the following labelled statement, which is commonly called fall-through.

Each `case` label in a single `switch` must have a distinct constant value after conversion to the controlling expression type. Duplicate case values make the program ill-formed. A `double` expression cannot be used directly as the controlling expression of a built-in `switch`.

QUESTION NO: 16

A condition expression used by if(), while(), and do-while() must evaluate to and only to:

A. an int type value

B. any value that can be treated as truth/falsehood

C. a float type value

D. a bool type value

ANSWER: B

Explanation:

“any value that can be treated as truth/falsehood” is correct because the controlling expression of an `if`, `while`, or `do-while` statement is evaluated in a Boolean context. C++ applies a contextual conversion to `bool`: a resulting value of `true` causes the controlled statement or loop body to execute, while `false` prevents execution or ends further loop iterations. This does not require the expression’s original type to be `bool`. For example, an integer expression can serve as a condition, with zero treated as false and a nonzero value treated as true; a pointer expression can similarly be tested for nullness. User-defined types may also participate when they provide an appropriate conversion usable in a Boolean context. Thus, the requirement concerns whether the expression is validly convertible to a truth value in that context, rather than requiring one particular arithmetic or Boolean type as its direct result. The C++ reference describes conditions as expressions that are contextually converted to `bool`; see cppreference: if statement and cppreference: contextual conversions.

QUESTION NO: 17

For an automatic object of class `Derived` that publicly inherits from class `Base`, which statements about construction and destruction are correct? (Choose two.)

A. The `Derived` constructor executes before the `Base` constructor.

B. The `Base` constructor executes before the `Derived` constructor.

C. The `Derived` destructor executes before the `Base` destructor.

D. The `Base` destructor must be virtual for it to run when an automatic `Derived` object leaves scope.

ANSWER: B C

Explanation:

When a `Derived` object is constructed, its base-class subobject must be initialized before the derived-class constructor body runs. Therefore, the `Base` constructor executes before the `Derived` constructor.

Destruction occurs in the reverse order. When an automatic `Derived` object leaves its scope, its `Derived` destructor executes first, followed by the destructor for its `Base` subobject. A virtual base destructor is important when deleting a derived object through a base-class pointer, but it is not required merely for normal destruction of an automatic derived object.

QUESTION NO: 18

Which code, inserted at line 19, generates the output " 23 " ?

#include < iostream >

#include < string >

using namespace std;

class A {

int x;

protected:

int y;

public:

int z;

```
A() { x=1; y=2; z=3; }
};
class B : public A {
string z;
public:
int y;
void set() { y = 4; z = " John " ; }
void Print() {
//insert code here
}
};
int main () {
B b;
b.set();
b.Print();
return 0;
}
```

A. cout < < y < < z;

B. cout < < y < < A::z;

C. cout < < A::y < < A::z;

D. cout < < B::y < < B::z;

ANSWER: C

Explanation:

`cout << A::y << A::z;` produces `23` because construction of a `B` object first constructs its public base-class `A` subobject. The `A` constructor initializes the base member `y` to `2` and the base member `z` to `3`. Within the member function `B::Print()`, the qualified names `A::y` and `A::z` explicitly select those members belonging to the inherited `A` subobject. The base `y` is protected, so it is accessible from a member function of the derived class; the base `z` is public and is also accessible. The qualification is important because `B` declares members with the same names, which hide the corresponding inherited names for ordinary unqualified lookup. Explicitly naming the base class accesses the initialized integer values from `A`. Since no separator is inserted between the two streamed integers, the output is the consecutive characters `23`. See the C++ reference material on derived classes and member access control.

QUESTION NO: 19

What happens when you attempt to compile and run the following code?

A. It prints: 5.2110.0

B. It prints: 5.210.0

C. It prints: 52.10

D. It prints: 5210

E. The output cannot be determined because the code is missing.

ANSWER: E

Explanation:

The code required to determine the program's behavior is absent from the question, so the output cannot be established from the information provided. In C++, output depends on the actual expressions sent to an output stream, their types, the stream manipulators in effect, overload resolution, and any surrounding program statements. For example, adjacent output operations normally produce concatenated text unless the source explicitly inserts whitespace or a separator; floating-point formatting can also vary according to precision and formatting flags. None of those details can be inferred from an empty code block. A valid compile-and-run question must include at least the relevant source statements, typically including the required headers, the entry-point function, and the output expression. Without that source, selecting any specific displayed sequence such as a combination of numeric values would be speculation rather than a conclusion derived from C++ rules. The C++ standard library specifies formatted output through stream insertion operations and their formatting state; see the basic_ostream insertion operator reference. General language and library references are also maintained by the ISO C++ organization. Therefore, the only supportable result is that the question cannot be answered until the missing code is supplied.

QUESTION NO: 20

What is the output of the program?

```
#include

#include

using namespace std;

class First

{

string name;

public:

First() {

name = "Alan";

}

void Print(){

cout << name;

}

};

int main()

{

First ob1,*ob2;

ob2 = new First();

ob1.Print();

ob2?>Print();
```

}

A. Garbage value

B. It prints: AlanAlan

C. It prints: Alan

D. It prints: Al

ANSWER: B

Explanation:

It prints: AlanAlan is correct. Creating `ob1` as an automatic object invokes `First::First()`, which assigns `"Alan"` to that object's `name` member. Creating the second `First` object with `new First()` also invokes the same default constructor, so the dynamically allocated object likewise has its `name` member set to `"Alan"`. The call `ob1.Print()` writes the first object's string to `std::cout`. The pointer-member invocation shown with HTML-affected syntax is intended to be `ob2->Print()`; it calls `Print()` on the dynamically allocated object and writes its string immediately afterward. Neither call writes whitespace or a newline, so the two strings appear consecutively as `AlanAlan`. C++ object initialization by constructors is described in the cppreference constructor documentation, and the arrow operator used to access a member through a pointer is covered in the member-access operator reference.

QUESTION NO: 21

What is the output of the program?

#include < iostream >

using namespace std;

#define SQR(x)(x*x)

int main(int argc, char *argv[]) {

int x, y=2;

x = SQR(y);

cout < < x < < " , " < < y;

return 0;

}

A. It prints: 3, 2

B. It prints: 4, 2

C. It prints: 3, 3

D. It prints: 9, 2

ANSWER: B

Explanation:

The output is **4, 2**. The preprocessor macro `SQR(x)` is defined as `(x*x)`. Before the C++ compiler processes the program, the preprocessor replaces the invocation `SQR(y)` with `(y*y)`. Since `y` has been initialized to `2`, the assignment is effectively `x = (2 * 2)`, so `x` receives the value `4`. This expression only reads `y`; it does not assign to it or otherwise modify it. Consequently, `y` remains `2`. The output statement sends `x`, then the literal string `" , "`, and then `y` to the standard output stream. Therefore, the displayed values are 4 followed by 2, with the spaces and comma supplied by the string literal. Function-like macros perform token substitution, which is why the macro call expands directly into the multiplication

expression before compilation. The C++ reference describes macro replacement during preprocessing at cppreference: macro replacement. For background on standard output via `std::cout` and insertion operators, see cppreference: std::cout.

QUESTION NO: 22

Which of the following statements about public inheritance and virtual member functions are correct? (Choose two.)

A. A virtual member function cannot be called through a base-class pointer.

B. A base-class pointer can point to an object of a publicly derived class.

C. A non-virtual member-function call through a base-class pointer always invokes the derived-class function.

D. A virtual member-function call through a base-class reference can invoke an overriding derived-class function.

ANSWER: B D

Explanation:

A pointer to a public base class can point to an object of a publicly derived class. For example, if `Derived` publicly inherits from `Base`, then `Base *p = new Derived;` is permitted.

When a member function is virtual in the base class and overridden in the derived class, a call made through a base-class pointer or reference uses the dynamic type of the object. Thus, a call through a `Base&` referring to a `Derived` object invokes the derived override. In contrast, a non-virtual member-function call through a base pointer is resolved using the pointer's static type.