PassQueen
a.k.a dumpsqueen.com

C++ Certified Professional Programmer

C++ Institute CPP

Version Demo

Total Demo Questions: 40

Total Premium Questions: 407

Buy Premium PDF

https://passqueen.com

support@passqueen.com

Topic Break Down

Topic	No. of Questions
Topic 1, Advanced C++ Programming	44
Topic 2, C++ Standard Library	363
Total	407

QUESTION NO: 1

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

#include

#include #include using namespace std; class B { int val; public:

B(int v=0):val(v){} int getV() const {return val;} operator int () const { return val;} }; templatestruct Out { ostream & out; Out(ostream & o): out(o){}

void operator() (const T & val) { out<

struct Add {

B operator()(B & a, B & b) { return a+b; }}; int main() {

int t[]={1,2,3,4,5,6,7,8,9,10}; vector **v1(t, t+10); vector v2(10);**

transform(v1.begin(), v1.end(), v2.begin(), bind1st(1,Add()));

for_each(v2.rbegin(), v2.rend(), Out**(cout));cout<**

Program outputs:

A. 1 2 3 4 5 6 7 8 9 10

B. 2 3 4 5 6 7 8 9 10 11

C. 10 9 8 7 6 5 4 3 2 1

D. 11 10 9 8 7 6 5 4 3 2

E. compilation error

ANSWER: E

Explanation:

`compilation error` is correct. The call to `bind1st(1, Add())` uses the old STL binder adaptor. In the C++ standard-library form of this adaptor, the operation supplied to it must provide the argument and result type information expected by the legacy function-object protocol (traditionally obtained by inheriting from `std::binary_function` or by declaring the corresponding nested type aliases). `Add` declares only an `operator()(B&, B&)`; it does not provide those required type declarations. Consequently, the binder cannot form the type it needs for use as the unary operation passed to `transform`.

There is also an interface mismatch in the call signature: the legacy binder invokes the operation through const argument references, whereas `Add::operator()` requires non-const `B&` parameters. Thus, even when evaluating this as pre-C++17 code where the old adaptor was present, the supplied function object does not meet the adaptor's requirements. In C++17 and later, `std::bind1st` itself was removed from the standard library, which is an additional reason modern conforming builds reject this program. The program therefore does not reach `transform`, reversal, or output. See cppreference: bind1st and bind2nd and cplusplus.com: bind1st.

QUESTION NO: 2

What happens when you attempt to compile and run the following code? Choose all that apply.

#include

#include

#include

#include

#include #include using namespace std; templatestruct Out { ostream & out;

Out(ostream & o): out(o){}

void operator() (const T & val) {out<<><>

int t[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}; fstream f("test.out", ios::trunc|ios::out); list l(t, t+10);

for_each(l.begin(), l.end(), Out(f)); f.close(); f.open("test.out"); for(; f.good() ;) {

int i; f>>i;

cout<

}

f.close();

return 0; }

A. file test.out will be opened writing

B. file test.out will be truncated

C. file test.out will be opened for reading

D. no file will be created nor opened

E. program will display sequence 1 2 3 4 5 6 7 8 9 10

ANSWER: A B C

Explanation:

The statements *file test.out will be opened writing*, *file test.out will be truncated*, and *file test.out will be opened for reading* are correct. The initial `fstream` construction explicitly supplies `ios::out | ios::trunc`. The `ios::out` flag opens the stream for output, allowing the `for_each` call and its output function object to send each list value to `test.out`. The `ios::trunc` flag requests that an existing file's contents be discarded when it is successfully opened, and the output mode permits creation of the file when necessary. After writing, `close()` releases the first file association. The subsequent `f.open("test.out")` uses `basic_fstream::open` with its default mode, `ios::in | ios::out`. Therefore, this second open includes input capability, so formatted extraction using `f >> i` can read from the file. It also retains output capability, although this program uses the reopened stream for input. The relevant stream-mode behavior and default mode are specified in the C++ stream documentation: cppreference: basic_fstream constructors and cppreference: basic_fstream::open.

QUESTION NO: 3

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

#include #include using namespace std; class B { int val; public:

B(int v=0):val(v){} int getV() const {return val;}

operator int () const { return val;} };

templatestruct Out { ostream & out; Out(ostream & o): out(o){}

void operator() (const T & val) { out<

struct Add {

B operator()(B & a, B & b) { return a+b; }}; int main() {

int t[]={1,2,3,4,5,6,7,8,9,10}; vector **v1(t, t+10); vector v2(10);**

transform(v1.begin(), v1.end(), v2.begin(), bind1st(Add(),1));

for_each(v2.rbegin(), v2.rend(), Out**(cout));cout<**

Program outputs:

A. 1 2 3 4 5 6 7 8 9 10

B. 2 3 4 5 6 7 8 9 10 11

C. 10 9 8 7 6 5 4 3 2 1

D. 11 10 9 8 7 6 5 4 3 2

E. compilation error

ANSWER: E

Explanation:

The result is **compilation error**. The intended use of `bind1st(Add(), 1)` relies on the legacy STL binder adapter. In the C++ standard-library design used by `bind1st`, the binary function object must provide adaptable-function typedefs, including `first_argument_type`, `second_argument_type`, and `result_type`. The `Add` class supplies only an `operator()`; it does not provide those required member types. Consequently, the library cannot determine the first argument type to which the integer literal `1` should be converted or construct the binder type, so template instantiation fails.

In addition, the call operator is declared as `B operator()(B&, B&)`. A bound function adapter invokes its stored argument and incoming argument through its adapter interface, which is not compatible with this non-const-reference parameter design. A modern implementation would normally use a lambda, or a function object accepting suitable const references, rather than the obsolete binder facilities. `bind1st` was deprecated in C++11 and removed from the standard library in C++17, so compiling this source under a current language standard can also fail because the facility is unavailable. See cppreference: bind1st and bind2nd and cppreference: std::bind.

QUESTION NO: 4

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

#include

#include #include

using namespace std;

void myfunction(pair i) { cout << " " << i.first;

}

int main() {

int t[] = { 10, 5, 9, 6, 2, 4, 7, 8, 3, 1 }; map m; for(int i=0; i < 10; i++) { m[i]=t[i];

}

for_each(m.begin(), m.end(), myfunction); return 0; }

Program outputs:

A. 10 5 9 6 2 4 7 8 3 1

B. 0 1 2 3 4 5 6 7 8 9

C. 9 8 7 6 5 4 3 2 1 0

D. 1 3 8 7 4 2 6 9 5 10

E. compilation error

ANSWER: B

Explanation:

The program outputs **0 1 2 3 4 5 6 7 8 9**. Each loop iteration stores an element in the map with `i` as its key and `t[i]` as its mapped value. Thus, the map contains key/value associations such as `0 -> 10`, `1 -> 5`, through `9 -> 1`. A `std::map` maintains its elements ordered by key using its comparison object; with the default comparison for integers, this is ascending numeric order. Consequently, iteration from `m.begin()` to `m.end()` visits the elements in key order from zero through nine, rather than in the order represented by the array values.

`std::for_each` invokes `myfunction` once for every iterated map element. The function prints `i.first`, which is the pair's first member: the key. Although map elements use a const key internally, the pair can be converted to the by-value `pair<int, int>` parameter used here. See the std::map reference for its ordered-associative-container behavior and the std::for_each reference for the algorithm's invocation behavior.

QUESTION NO: 5

Which changes introduced independently will allow code to compile and display 0 1 8 9 (choose all that apply)

#include < iostream >

#include < set >

#include < vector >

using namespace std;

class A {

int a;

public:

A(int a):a(a){}

int getA() const { return a;}

/* Insert Code Here 1 */

};

/* Insert Code Here 2 */

int main(){

A t[] = { 3, 4, 2, 1, 6, 5, 7, 9, 8, 0 };

vector < A > v(t, t+10);

set < A > s1(v.begin(),v.end());

s1.insert(v.begin(),v.end());

s1.erase(s1.lower_bound(2),s1.upper_bound(7));

for(set < A > ::iterator i=s1.begin();i!= s1.end(); i++) {

cout < < i? > getA() < < " ";

}

cout < < endl;

return 0;

}

A. operator int() const { return a;} inserted at Place 1

B. bool operator < (const A & b) const { return a < b.a;} inserted at Place 1

C. bool operator < (const A & b) const { return b.a < a;} inserted at Place 1

D. bool operator < (const A & a, const A & b) { return a.getA() < b.getA();} inserted at Place 2

ANSWER: A B D

Explanation:

A `set<A>` uses `std::less<A>` by default, which orders its keys through the less-than operation. The member overload `bool operator < (const A & b) const { return a < b.a;}` directly supplies that natural ascending ordering. The non-member overload `bool operator < (const A & a, const A & b) { return a.getA() < b.getA();}` supplies the identical ordering and is found for expressions comparing two `A` objects. In either case, the set contains each integer value once in ascending order. The conversion operator `operator int() const { return a;}` also permits the built-in integer less-than comparison to be used after conversion, thereby providing ascending ordering for the set. It additionally allows the integer arguments passed to `lower_bound` and `upper_bound` to participate in the required comparisons; the converting constructor permits construction of an `A` key where needed. Thus, after keys in the inclusive range from 2 through 7 are erased, the remaining ordered keys are 0, 1, 8, and 9, which are displayed through the iterator (with the intended `i->getA()` expression). The standard library specifies that ordered associative containers maintain keys according to their comparison object; see cppreference: std::set. Operator-overload and conversion rules are summarized at cppreference: operator overloading.

QUESTION NO: 6

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

#include < vector >

#include < iostream >

#include < algorithm >

#include < functional >

using namespace std;

class B { int val;

public:

B(int v=0):val(v){}

int getV() const {return val;}

B operator +(const B & b)const { return B(val + b.val);} };

ostream & operator < < (ostream & out, const B & v) { out < < v.getV(); return out;}

template < class T > struct Out {

ostream & out;

Out(ostream & o): out(o){}

void operator() (const T & val) { out < < val < < " "; } };

B Add(B a, B b) { return a+b; }

int main() {

int t[]={1,2,3,4,5,6,7,8,9,10};

vector < B > v1(t, t+10);

vector < B > v2(10);

transform(v1.begin(), v1.end(), v2.begin(), bind2nd(ptr_fun(Add),1));

for_each(v2.rbegin(), v2.rend(), Out < B > (cout));cout < < endl;

return 0;

}

Program outputs:

A. 1 2 3 4 5 6 7 8 9 10

B. 2 3 4 5 6 7 8 9 10 11

C. 10 9 8 7 6 5 4 3 2 1

D. 11 10 9 8 7 6 5 4 3 2

E. compilation error

ANSWER: D

Explanation:

`11 10 9 8 7 6 5 4 3 2` is the output in the C++ language version assumed by this legacy-adapter question. The range constructor for `v1` creates ten `B` objects from the integers in `t`; this is permitted because `B(int)` is a converting constructor. `ptr_fun(Add)` adapts the binary `Add` function, and `bind2nd(..., 1)` binds its second argument to `1`, which is likewise converted to a `B`. Consequently, `transform` invokes the effective operation `Add(element, B(1))` for every element of `v1`. The overloaded addition operator returns a new `B` whose stored value is one greater than the input value, so `v2` contains values from 2 through 11 in ascending order. The call to `for_each` uses `rbegin()` and `rend()`, traversing that vector from its last element to its first. `Out<B>` writes each value followed by a space, producing the stated descending sequence before the newline. The algorithm's element-wise transformation behavior is documented at cppreference: transform. These function-adapter facilities are legacy C++ components; bind2nd notes their later removal from C++17, but that does not alter the intended exam result under the earlier standard.

QUESTION NO: 7

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

#include

#include

#include #include

using namespace std;

int main() {

int t[] = {1,2,3,2,3,5,1,2,7,3,2,1,10, 4,4,5}; vector v1(t, t + 15);

set s1(t, t + 15);

pair<>::iterator, vector::iterator > resultSet = equal(s1.begin(), s1.end(), v1.begin

());

cout<<*resultSet.first<<" "<<*resultSet.second<<>

return 0; }

Program outputs:

A. 2 4

B. 4 2

C. 0 5

D. compilation error

ANSWER: D

Explanation:

compilation error is correct. Even when the malformed HTML-like fragments in the displayed source are understood as formatting damage and the intended C++ declarations are reconstructed, the key statement remains ill-formed: `std::equal(s1.begin(), s1.end(), v1.begin())` returns a `bool`. The three-argument overload of `std::equal` compares the range beginning at the first set iterator and ending at the second set iterator against a range beginning at the vector iterator. Its result states only whether the corresponding elements compare equal; it does not return iterators identifying a mismatch or a position in either range. Consequently, that Boolean result cannot initialize a `pair<set<int>::iterator, vector<int>::iterator>`, so the compiler must reject the initialization before execution can occur. The standard library algorithm reference documents the Boolean return value of `equal`: cppreference: std::equal. A pair is a distinct two-member object type, as described by cppreference: std::pair; producing iterator pairs requires an algorithm whose specified return type is such a pair, rather than `std::equal`.

QUESTION NO: 8

Which lambda expressions can be used independently to replace the marked expression and allow the code to compile? Choose all that apply.

```
#include <algorithm>
#include <vector>

using namespace std;

int main() {
    vector<int> v(5, 1);
    int n = 10;

    for_each(v.begin(), v.end(), /* lambda expression */);
}
```

A. `[n](int& x) { x += n; }`

B. `[&n](int& x) { x += n; }`

C. `[](int x) { x *= 2; }`

D. `[n]() { n++; }`

E. `[](const int& x) { return x == 1; }`

ANSWER: A B C E

Explanation:

The callable passed to `for_each` must be invocable with a vector element. A lambda accepting `int&` can modify each element, while a lambda accepting `int` receives a copy and can still be called successfully. Capturing `n` by value or by reference does not prevent invocation when the lambda parameter is suitable.

A lambda with no parameters cannot be called by this use of `for_each`, because the algorithm invokes its function object with each element in the range. See cppreference: std::for_each and cppreference: Lambda expressions.

QUESTION NO: 9

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

#include

#include #include using namespace std;

templateclass B { T val; public:

B(T v):val(v){}

T getV() const {return val;} bool operator < (const B & v) const { return valostream & operator <<(ostream & out, const B & v) { out<

templatestruct Out { ostream & out;

Out(ostream & o): out(o){}

void operator() (const T & val) { out<

bool Less(const B &a, const B &b) { return int(a.getV())

float t[]={2.28, 1.66, 1.32, 3.94, 3.64, 2.3, 2.98, 1.96, 2.62, 1.13}; vector<> > v1; v1.assign(t, t+10); stable_sort(v1.begin(), v1.end(), Less);

for_each(v1.begin(), v1.end(), Out<> >(cout));cout<<>

return 0; }

Program outputs:

A. 1.66 1.32 1.96 1.13 2.28 2.3 2.98 2.62 3.94 3.64

B. 1.13 1.32 1.66 1.96 2.28 2.3 2.62 2.98 3.64 3.94

C. compilation error

D. 3.94 3.64 2.98 2.62 2.3 2.28 1.96 1.66 1.32 1.13

E. the exact output is impossible to determine

ANSWER: A

Explanation:

The output is **1.66 1.32 1.96 1.13 2.28 2.3 2.98 2.62 3.94 3.64**. The comparison function does not compare the complete floating-point values. Instead, it converts each stored value to `int` before comparing it. Consequently, the values are ordered into three integer-valued groups: values whose integer conversion is 1, then those converting to 2, then those converting to 3.

Within each group, the comparator considers the elements equivalent because neither integer value is less than the other. Since `stable_sort` is used, equivalent elements retain their original relative order. The values converting to 1 therefore remain in input order as `1.66, 1.32, 1.96, 1.13`. The next group retains `2.28, 2.3, 2.98, 2.62`, followed by `3.94, 3.64`. The output functor writes each value followed by a space. The standard library guarantee that stable sorting preserves the ordering of equivalent elements is documented at cppreference: std::stable_sort; integer conversion behavior is described at cppreference: integral conversions.

QUESTION NO: 10

Which changes introduced independently will allow the code to compile and display 0 0 1 1 8 8 9 9 (choose all that apply)?

```
#include
#include
#include
using namespace std;
class A {
int a;
public:
A(int a):a(a){}
int getA() const { return a;}
/* Insert Code Here 1 */
};
/* Insert Code Here 2*/
int main(){
A t[] ={ 3, 4, 2, 1, 6, 5, 7, 9, 8, 0 };
sets(t, t+10);/* Replace Code Here 3 */
multiset s1(s.begin(),s.end());/* Replace Code Here 4 */
s1.insert(s.begin(),s.end());
s1.erase(s1.lower_bound(2),s1.upper_bound(7));
multiset::iterator i=s1.begin();/* Replace Code Here 5 */
for( ;i!= s1.end(); i++)
{
cout<getA()<<" ";
}
cout<
return 0;
}
```

A. operator int() const { return a;} inserted at Place 1

B. bool operator < (const A & b) const { return a < b.a;} inserted at Place 1

C. bool operator < (const A & b) const { return b.a < a;} inserted at Place 1

D. replacing line marked 3 with set > s(t, t+10);

ANSWER: A B

Explanation:

Both `operator int() const { return a;}` and `bool operator < (const A & b) const { return a < b.a;}` independently provide the ordering needed by the default comparator used by `set<A>` and `multiset<A>`. The conversion operator permits the built-in integer comparison to be used when two `A` objects are compared. The member less-than operator instead directly defines the natural ascending ordering of the stored integer values. In either case, the containers contain the values in increasing order.

The initial `set` has one copy of every value from 0 through 9. Constructing the `multiset` from that set creates one copy of each value, and inserting the set range again creates a second copy. The range beginning at `lower_bound(2)` and ending at `upper_bound(7)` removes both occurrences of every value from 2 through 7. Iteration therefore visits the remaining sorted elements as `0 0 1 1 8 8 9 9`. The associative-container comparison requirements and the behavior of ordered lookup operations are described at cppreference: std::set and cppreference: std::multiset::erase.

QUESTION NO: 11

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

#include

#include #include using namespace std; class B { int val; public:

B(int v):val(v){}

int getV() const {return val;} bool operator < (const B & v) const { return valstruct Out { ostream & out;

Out(ostream & o): out(o){}

void operator() (const T & val) { out<

B t1[]={3,2,4,1,5}; B t2[]={6,10,8,7,9}; vector **v1(10,0); sort(t1, t1+5); sort(t2, t2+5); copy(t1,t1+5,v1.begin()); copy(t2,t2+5,v1.begin()+5);**

inplace_merge(v1.begin(), v1.begin()+5,v1.end());

for_each(v1.begin(), v1.end(), Out**(cout));cout<**

Program outputs:

A. 1 2 3 4 5 6 10 8 7 9

B. 3 2 4 1 5 6 7 8 9 10

C. 3 2 4 1 5 6 10 8 7 9

D. 1 2 3 4 5 6 7 8 9 10

E. compilation error

ANSWER: D

Explanation:

The program outputs `1 2 3 4 5 6 7 8 9 10`. Objects of type `B` are ordered through the overloaded `operator<`, which compares their stored integer values. First, `sort` rearranges the first array into the sequence from 1 through 5 and the second array into the sequence from 6 through 10. Those two sorted sequences are then copied into adjacent halves of `v1`.

`inplace_merge` is specifically intended for this situation: its input range is split at the middle iterator, and each half is already sorted using the same comparison relation. It merges both halves in place, preserving sorted order across the complete vector. Since every value in the first half is less than every value in the second half, the merged vector contains the consecutive values from 1 through 10. Finally, `for_each` invokes `Out<B>` for every element; the stream insertion operator for `B` writes each stored value followed by a space. The standard specifies the sorted-subrange precondition and merging behavior of `inplace_merge`; see the C++ standard's merge algorithms section and cppreference's inplace_merge reference.

QUESTION NO: 12

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

#include

#include #include using namespace std;

class A { int a; public:

A(int a) : a(a) {}

int getA() const { return a; } void setA(int a) { this?>a = a; } bool operator==(A & b) { return a == b.a; }

};

struct Compare{

bool operator()(const A & a, const A & b) {return a.getA()==b.getA();};

};

int main () {

int t[] = {1,2,3,4,5,1,2,3,4,5}; vector v (t,t+10); vector::iterator it; A m1[] = {A(1), A(2), A(3)};

it = search (v.begin(), v.end(), m1, m1+3, Compare()); cout << "First found at position: " << it?v.begin() << endl; return 0;

}

Program outputs:

A. First found at position: 5

B. First found at position: 0

C. First found at position: 7

D. compilation error

E. First found at position: 10

ANSWER: B

Explanation:

Assuming the formatting artifacts in the displayed source are restored to their intended C++ tokens, the program outputs **First found at position: 0**. The integer array initializes a `vector<A>` containing values 1, 2, 3, 4, 5, 1, 2, 3, 4, 5. This is possible because `A` has a converting constructor that accepts an `int`. The pattern array contains the three values 1, 2, and 3.

`std::search` examines the vector from its beginning and returns an iterator to the first occurrence of the complete pattern. The first three vector elements already match 1, 2, 3, so the returned iterator is `v.begin()`. Subtracting `v.begin()` from that iterator produces an offset of zero, which is then written after the fixed message. The supplied comparison function compares the stored integer values through `getA()`, so it correctly establishes equality for each candidate pattern element. The standard specifies that the overload of `search` with a predicate returns an iterator to the first matching subsequence. See cppreference: std::search and cppreference: std::vector constructors.

QUESTION NO: 13

What happens when you attempt to compile and run the following code? Choose all possible answers.

#include

```
using namespace std;

class C {

public: int _c; C():_c(0){}

C(int c) { _c = c;}

C operator+=(C & b) { C tmp; tmp._c = _c+b._c; return tmp;

} };

ostream & operator<<(ostream & c, const C & v) { c<

template class A { T _v; public:

A() {}

A(T v): _v(v){} T getV() { return _v; }

void add(T & a) { _v+=a; }

};

int main()

{

A b(2); A a (5); a.add(C());

cout << a.getV() <
```

A. program will display:5

B. program will not compile

C. program will compile

D. program will cause runtime exception

E. The call `a.add(C())` is ill-formed because a temporary cannot bind to `C&`.

ANSWER: B E

Explanation:

The program will not compile because `A<C>::add` is instantiated with the parameter type `C&`, which is a non-const lvalue reference. The expression `C()` creates a temporary `C` object (a prvalue), not a named lvalue object. Standard C++ does not allow a temporary to bind to a non-const lvalue reference parameter, so the call `a.add(C())` is ill-formed and requires a compiler diagnostic. Consequently, execution never begins and no value can be displayed by the output statement. A precise diagnostic description is that a temporary `C` cannot bind to the `C&` argument required by `add`. Reference binding rules are described by cppreference: Reference initialization. The fact that `C` can be constructed from the integer used to initialize `a` does not change the argument category of the separate `C()` expression passed to `add`. The template member function is checked using the instantiated type, making its required parameter type exactly `C&`; see cppreference: Templates.

QUESTION NO: 14

Which stack initialization (line numbers) are correct? Choose all that apply.

```
#include

#include

#include
```

#include #include

using namespace std;

int main()

{

deque mydeck; list mylist; vector myvector; stack first;// Line I stack second(mydeck);// Line II stack third(second);// Line III stack > fourth(mylist);// Line IV stack > fifth(myvector);// Line V return 0; }

A. line I

B. line II

C. line III

D. line IV

E. line V

ANSWER: A B C D E

Explanation:

All of the listed stack initializations are valid. `std::stack<int> first` uses the adaptor's default underlying container, which is `std::deque<int>`. A stack may also be constructed from an existing compatible underlying container, so constructing `second` from the `deque<int>` object is valid. Copy construction is supported as well, making construction of `third` from another `stack<int>` valid.

The stack adaptor's second template parameter specifies its underlying container type. Its container must provide the sequence-container operations used by a stack, including `back()`, `push_back()`, and `pop_back()`. Both `std::list<int>` and `std::vector<int>` meet those requirements. Therefore, initializing a stack backed by `list<int>` from `mylist`, and a stack backed by `vector<int>` from `myvector`, are valid. The relevant constructor accepts a const reference to the specified container type and initializes the adaptor with a copy of that container. See the std::stack reference and the stack constructors reference.

QUESTION NO: 15

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

#include #include

int main ()

{

std::vectorv1;

for(int i = 0; i<10; i++) {v1.push_back(i); } std::vector v2(v1.begin()+2, v1.end()?2); std::vector::iterator it = v2.begin();

for(; it != v2.end(); it++) {std::cout<<*it++<<" "; }std::cout<

A. compilation error

B. program outputs 0 1 2 3 4 5 6 7 8 9

C. program outputs 2 3 4 5 6 7

D. program outputs 2 4 6

ANSWER: A

Explanation:

The correct result is **compilation error**. In the displayed source, the second argument to the range constructor of `std::vector<int> v2` is written as `v1.end()?2`. The question-mark token begins C++'s conditional operator, whose required grammar is `condition ? expression : expression`. Here, no colon and no expression for the false branch are present, so this is not a valid expression and the compiler cannot parse the vector-constructor call. Compilation must complete successfully before a program can run; consequently, no output is produced by this source as written. The range constructor itself expects two iterator expressions identifying the beginning and end of a source range, but the malformed conditional expression prevents the second iterator argument from being formed. The code shown also contains HTML-like artifacts in its include directives and return statement, reinforcing that it is not valid C++ translation-unit text in its displayed form. See the conditional-operator syntax at cppreference: Conditional operator and the iterator-range vector constructor at cppreference: std::vector constructors.

QUESTION NO: 16

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

#include

#include

#include

#include <algorithm>

using namespace std;

class B { int val;

public:

B(int v):val(v){}

int getV() const {return val;} bool operator < (const B & v) const { return val

ostream & operator <<(ostream & out, const B & v) { out<

templatestruct Out {

ostream & out;

Out(ostream & o): out(o){}

void operator() (const T & val) { out<

int main() {

int t[]={8, 10, 5, 1, 4, 6, 2, 7, 9, 3};

deque **d1(t, t+10);**

sort(d1.begin(), d1.end());

set s1(t,t+10);

cout<

return 0;

}

Program outputs:

A. 1 0

B. 1 1

C. 0 0

D. 0 1

E. compilation error

ANSWER: B

Explanation:

The program outputs **1 1** (with the separator shown by the stream expression, intended as =). The integer array supplies the values 1 through 10 in an unsorted order. Each integer is used to construct a `B` object through `B(int)`. The comparison operator for `B` compares the stored `val` members, establishing the ordering used by both the sort operation and the set.

After `sort(d1.begin(), d1.end())`, the deque is ordered by increasing stored values. A `set<B>` also maintains its elements in that same strict ordering as it is constructed. Therefore, both ranges supplied to `binary_search` meet the algorithm's sorted-range precondition. Since each range contains an object whose stored value is 4, searching each range for `B(4)` succeeds. `binary_search` returns `true` in both cases; without `boolalpha`, streaming those Boolean values produces `1`.

The standard library specifies that `std::binary_search` tests whether an equivalent element exists in a partitioned (sorted) range, and ordered associative containers maintain their elements according to their comparison relation. See cppreference: std::binary_search and cppreference: std::set.

QUESTION NO: 17

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

#include

#include #include using namespace std;

class A { int a; public:

A(int a) : a(a) {}

int getA() const { return a; } void setA(int a) { this?>a = a; } operator int() const {return a;}

};

int main () {

int t[] = {1,2,3,2,3,5,1,2,7,3,2,1,10, 4,4,5}; set s (t,t+15);

cout<<>

return 0;

}

Program outputs:

A. true

B. false

C. 1

D. 0

E. compilation error

ANSWER: D

Explanation:

The program outputs **0**. The range used to construct the set starts at the first element of `t` and ends at `t + 15`, so it contains the first fifteen integers. Each integer can initialize an object stored by the set, and the conversion operator supplies an integer value when the set's ordering comparison is needed. Since a set stores unique values in ascending order, its sequence is `1, 2, 3, 4, 5, 7, 10`.

The intended call to `equal` compares that ordered set sequence with elements beginning at `t`. The first three corresponding values are equal, but the fourth value from the set is `4`, whereas the fourth array value is `2`. Consequently, `equal` returns `false`. By default, streaming a Boolean value to `cout` prints its numeric representation rather than textual Boolean names: `false` is printed as `0`. This behavior is specified by the stream formatting rules; textual output would require the `boolalpha` manipulator. See std::equal and Boolean stream formatting.

QUESTION NO: 18

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

```
#include < deque >

#include < iostream >

#include < algorithm >

#include < set >

using namespace std;

template < class T > struct Out {

ostream & out;

Out(ostream & o): out(o){}

void operator() (const T & val ) { out < < val < < " "; } };

int main() {

int t[]={8, 10, 5, 1, 4, 6, 2, 7, 9, 3};

int t1[]={1,2,3,4};

deque < int > d1(t, t+10);

set < int > s1(t, t+10);

sort(d1.begin(), d1.end());

cout < < includes(s1.begin(),s1.end(), t1,t1+4) < < " " < < includes(d1.begin(),d1.end(), t1,t1+4)

< < endl;

return 0;

}
```

Program outputs:

A. 1 1

B. 1 0

C. 0 1

D. 0 0

ANSWER: A

Explanation:

`1 1` is the correct output. `std::includes` determines whether every element in the second sorted range occurs in the first sorted range, accounting for multiplicity. Its default comparison is the ordinary less-than ordering, and both input ranges supplied to each call must be sorted using that ordering. The array represented by `t1` is already sorted and contains the values `1, 2, 3, 4`.

A `std::set<int>` maintains its elements in ascending order by default. Constructing `s1` from the ten-element array therefore produces an ordered set containing each integer from `1` through `10`. It consequently includes all values in `t1`. The deque initially receives the same ten values in an unsorted sequence, but `std::sort(d1.begin(), d1.end())` orders the full deque before the algorithm is called. It then also contains every value required by `t1`.

Both calls thus return `true`. Unless the `boolalpha`cout prints `1` for true. The literal space between the insertions and `endl` yield `1 1` followed by a newline. See std::includes and std::set.

QUESTION NO: 19

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

#include

#include

#include <iostream>

#include <algorithm>

using namespace std;

templatestruct Out {

ostream & out;

Out(ostream & o): out(o){}

void operator() (const T & val) { out<<val<<" "; } };

struct Sequence {

int start;

Sequence(int start):start(start){}

int operator()() { return start++; } };

int main() {

vector v1(10);

generate_n(v1.begin(), 10, Sequence(1));

random_shuffle(v1.rbegin(), v1.rend());

sort(v1.begin(), v1.end(), great());

for_each(v1.begin(), v1.end(), Out(cout));cout<<endl;

return 0;

}

Program outputs:

A. 8 10 5 1 4 6 2 7 9 3

B. 1 2 3 4 5 6 7 8 9 10

C. compilation error

D. 10 9 8 7 6 5 4 3 2 1

ANSWER: C

Explanation:

compilation error is correct because the program text is not valid C++ source. In particular, the template declaration introduces a parameter named `t`, while the call operator uses `T`. C++ is case-sensitive, so `T` is not the declared template parameter and is not declared in that scope. Consequently, the compiler cannot form a valid declaration for `Out`'s function-call operator. The source also spells the standard comparison functor as `great<int>()`; the standard library facility is `std::greater<int>`. Therefore the call to `sort` cannot be compiled as written. In addition, the literal closing-tag fragments such as `</vector>` and `</int>`, if present in the actual source file rather than only being accidental formatting artifacts, are not valid C++ tokens in those positions. Compilation must succeed before execution and output are possible, so no output sequence is produced. The language rule that names are case-sensitive is described in the C++ identifier reference; the required ordering functor is documented as std::greater.

QUESTION NO: 20

Which statements about `std::unique_ptr` are correct? Choose all that apply.

A. A unique_ptr automatically deletes its owned object when the unique_ptr is destroyed.

B. A unique_ptr can be move-constructed from another unique_ptr of a compatible type.

C. Two unique_ptr objects can be copy-constructed so that they share ownership of one object.

D. A moved-from unique_ptr is invalid and must not be destroyed.

E. unique_ptr p(new int(5)); is a valid ownership declaration.

ANSWER: A B E

Explanation:

A `std::unique_ptr<T>` represents exclusive ownership of a dynamically allocated object. It automatically destroys the owned object when the `unique_ptr` is destroyed, unless ownership has previously been released or transferred. Copy construction and copy assignment are disabled because two independent `unique_ptr` objects must not own the same resource.

Ownership can be transferred by move construction or move assignment, usually by applying `std::move` to the source pointer. After a move, the source `unique_ptr` is valid but is normally empty. See cppreference: std::unique_ptr.

QUESTION NO: 21

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

#include

#include #include using namespace std; templatestruct Out { ostream & out;

Out(ostream & o): out(o){}

void operator() (const T & val) { out<

int t[]={8, 10, 5, 1, 4, 6, 2, 7, 9, 3}; vector v1(t, t+10); sort(v1.begin(), v1.end(), Greater);

for_each(v1.begin(), v1.end(), Out(cout));cout<

Program outputs:

A. 8 10 5 1 4 6 2 7 9 3

B. 1 2 3 4 5 6 7 8 9 10

C. compilation error

D. 10 9 8 7 6 5 4 3 2 1

ANSWER: D

Explanation:

The program outputs **10 9 8 7 6 5 4 3 2 1**. The intended comparison function, `Greater`, returns true when its first integer argument is greater than its second. When this function is supplied as the third argument to `sort`, it establishes a descending ordering: an element with a larger value is placed before an element with a smaller value. The vector is initialized with each integer from 1 through 10, although they start in a mixed order. Sorting with this comparator therefore rearranges the vector from the greatest value to the least value. Next, `for_each` applies the `Out<int>` function object to every vector element. Its call operator writes each value followed by a space to `cout`, so the values appear in that descending sequence, followed by a newline. The malformed-looking markup around the source is presentation corruption; the intended C++ source uses the template type name and comparator name consistently. The standard behavior of comparator-based sorting is documented by cppreference: std::sort; callable objects used by algorithms are described at cppreference: std::for_each.

QUESTION NO: 22

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

#include

#include

using namespace std;

int main ()

{

float f = 10.126;

cout<<><>

Program outputs:

A. 10.126 10

B. 10.126 10.12

C. compilation error

D. 10.126 10.13

ANSWER: A

Explanation:

The output `10.126 10` is produced because the stream first inserts `f` using the default floating-point formatting. For this value, the default general format displays the significant digits needed here as `10.126`. The subsequent formatting state in

the intended stream expression applies `fixed` and `setprecision(0)` before inserting `f` again. With `fixed`, precision denotes the number of digits printed after the decimal point rather than the total number of significant digits. A precision of zero therefore requests no fractional digits. The value is formatted with normal rounding, so `10.126` becomes `10`. Stream manipulators change the formatting state of `cout`, affecting insertions that follow them in the same expression; they do not retroactively alter the first insertion. Finally, `endl` outputs a newline and flushes the stream, without changing the numeric text. The standard formatting behavior of `fixed` is documented at cppreference: fixed, and the meaning of stream precision is documented at cppreference: setprecision.

QUESTION NO: 23

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

#include

#include

#include

#include

using namespace std;

templatestruct Out {

ostream & out;

Out(ostream & o): out(o){}

void operator() (const T & val) { out<

int main() {

string t[]={"aaa","Aaa", "aAa","aaA","bbb","Bbb", "bBb", "bbB"};

vector v1(t, t+8);

sort(v1.begin(), v1.end());

for_each(v1.begin(), v1.end(), Out(cout));cout<

return 0;

}

Program outputs:

A. Aaa Bbb aAa aaA aaa bBb bbB bbb

B. Aaa aAa Bbb aaA aaa bBb bbB bbb

C. bBb bbB bbb Aaa aAa Bbb aaA aaa

D. Aaa aAa bBb bbB bbb Bbb aaA aaa

E. compilation error

ANSWER: B

Explanation:

`Aaa aAa Bbb aaA aaa bBb bbB bbb` is the output. The formatting in the displayed listing has corrupted several C++ tokens into HTML-like fragments, but the intended program defines the usual output functor template `Out<T>`, constructs a `vector<string>`, sorts it with the default comparator, and passes every sorted element to `for_each`. The default ordering used by `std::sort` is the string type's `operator<`. For the ordinary narrow-character strings used here,

comparison is lexicographical: characters are compared at the first position where they differ. In the usual execution character set, uppercase letters sort before lowercase letters, so `Aaa` precedes `aAa` and `Bbb`. Among strings beginning with lowercase `a`, the second character determines that `aAa` comes before `aaA`, which comes before `aaa`. The same character-by-character rule places the lowercase-`b` strings in the order `bBb`, `bbB`, and `bbb`. The functor writes each value followed by a space, then the program writes a newline. See cppreference: std::sort and cppreference: basic_string comparisons.

QUESTION NO: 24

What happens when you attempt to compile and run the following code? Choose all that apply.

```
#include < iostream >

#include < fstream >

#include < string >

#include < list >

#include < algorithm >

#include < iomanip >

using namespace std;

template < class T > struct Out {

ostream & out;

Out(ostream & o): out(o){}

void operator() (const T & val ) {out < < setw(3) < < hex < < val; } };

int main () {

int t[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

fstream f("test.out", ios::trunc|ios::out);

list < int > l(t, t+10);

for_each(l.begin(), l.end(), Out < int > (f));

f.close(); f.open("test.out");

for( ; f.good() ; ) {

int i; f > > i;

cout < < i < < " ";

}

f.close();

return 0;

}
```

A. file test.out will be opened for writing

B. file test.out will be truncated

C. file test.out will be opened for reading

D. no file will be created nor opened

E. program will display sequence 1 2 3 4 5 6 7 8 9 10

ANSWER: A B C

Explanation:

`fstream f("test.out", ios::trunc|ios::out);` requests output access, so “file test.out will be opened for writing” is correct. The `ios::trunc` flag also causes the file’s existing contents to be discarded when the open succeeds; therefore, “file test.out will be truncated” is correct. This first open creates `test.out` if it does not already exist, subject to ordinary filesystem permissions.

After closing the stream, `f.open("test.out")` uses the default open mode for `std::basic_fstream`, which is `ios::in | ios::out`. Consequently, the stream is opened with input capability, making “file test.out will be opened for reading” correct. The preceding output operation writes each list element in hexadecimal, with a field width of three characters. These stream-opening and formatting behaviors follow the standard library definitions for `basic_fstream` and the `trunc` open mode. See cppreference: basic_fstream::open and cppreference: openmode.

QUESTION NO: 25

Which changes, introduced independently, will allow the code to compile and display “one” “eight” “nine” “ten”? Choose all that apply

```
#include < iostream >

#include < map >

#include < string >

using namespace std;

class A {

int a;

public:

A(int a):a(a){}

int getA() const { return a;}

/* Insert Code Here 1 */

};

/* Insert Code Here 2 */

int main(){

int t[] = { 3, 4, 2, 1, 6, 5, 7, 9, 8, 10 };

string s[] = {"three", "four", "two", "one", "six","five", "seven", "nine","eight","ten"};

map < A, string > m; /* Replace Code Here 3 */

for(int i=0; i < 10; i++) {

m.insert(pair < A,string > (A(t[i]),s[i] ));

}

m.erase(m.lower_bound(2),m.upper_bound(7));

map < A, string > ::iterator i=m.begin(); /* Replace Code Here 4 */
```

for(;i!= m.end(); i++) {

cout < < i? > second < < " ";

}

cout < < endl;

return 0;

}

A. operator int() const { return a;} inserted at Place 1

B. bool operator < (const A & b) const { return a < b.a;} inserted at Place 1

C. bool operator < (const A & b) const { return b.a < a;} inserted at Place 1

D. struct R { bool operator ()(const A & a, const A & b) { return a.getA() < b.getA();} }; inserted at Place 2replacing line marked 3 with map < A, string, R > m;replacing line marked 4 with map < A, string,R > ::iterator i=m.begin();

ANSWER: A B D

Explanation:

A `std::map` stores its keys in the order established by its comparison object. The default comparison is `std::less<A>`, which requires the key values to be orderable. Adding `operator int() const { return a;}` makes an `A` value implicitly convertible to `int`; consequently, the built-in integer less-than comparison can establish the required ascending key order. The non-explicit constructor also permits the integer arguments supplied to `lower_bound(2)` and `upper_bound(7)` to be converted to `A`.

Adding `bool operator < (const A & b) const { return a < b.a;}` directly supplies the natural ascending ordering for keys. After erasing the keys from 2 through 7, inclusive, the remaining ordered keys are 1, 8, 9, and 10, so iteration displays their associated strings as `one eight nine ten`.

The custom comparator solution likewise explicitly orders two `A` objects by `getA()`. Declaring the map and its iterator with `R` ensures the container consistently uses that ascending comparison for insertion, bounds lookup, erasure, and iteration. See cppreference: std::map and cppreference: std::less.

QUESTION NO: 26

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

#include

#include <iostream>

#include <algorithm>

#include <functional>

using namespace std;

templatestruct Out {

ostream & out;

Out(ostream & o): out(o){}

void operator() (const T & val) { out<<val<<" "; } };

struct Add : public binary_function {

int operator() (const int & a, const int & b) const {

return a+b;

}

};

int main() {

int t[]={1,2,3,4,5,6,7,8,9,10};

vector v1(t, t+10);

vector v2(10);

transform(v1.begin(), v1.end(), v2.begin(), bind1st(Add(), 1));

for_each(v2.rbegin(), v2.rend(), Out(cout));cout<<endl;

return 0;

}

Program outputs:

A. 1 2 3 4 5 6 7 8 9 10

B. 2 3 4 5 6 7 8 9 10 11

C. 10 9 8 7 6 5 4 3 2 1

D. 11 10 9 8 7 6 5 4 3 2

E. compilation error

ANSWER: D

Explanation:

`11 10 9 8 7 6 5 4 3 2` is correct. The intended template parameter of `Out` is `T`, so the call `Out<int>(cout)` creates a function object that writes each supplied integer followed by a space. The first vector contains the values from 1 through 10. `transform` applies its unary operation to every element in that input range and writes the corresponding results into `v2`. `bind1st(Add(), 1)` binds the first argument of `Add` to 1; therefore, each input value is increased by 1. Consequently, `v2` holds the ascending sequence 2 through 11.

The call to `for_each` uses `v2.rbegin()` and `v2.rend()`, which designate the same vector range in reverse traversal order. The output function object consequently receives 11 first, then each preceding value down to 2, yielding the stated sequence. The standard algorithm behavior of `transform` is documented at cppreference: transform, and reverse iterator traversal is described at cppreference: reverse_iterator.

QUESTION NO: 27

Which pieces of code inserted independently into places marked 1 and 2 will cause the program to compile and display: 0 1 2 3 4 5 6 7 8 9? Choose all that apply.

#include

#include

using namespace std;

class A { int a; public:

A(int a){ this?>a=a;}

//insert code here 1

};

//insert code here 2

template void print(T start, T end) {

while (start != end) {

std::cout << *start << " "; start++;

}

}

int main() {

A t1[] ={ 1, 7, 8, 4, 5 };list l1(t1, t1 + 5);

A t2[] ={ 3, 2, 6, 9, 0 };list l2(t2, t2 + 5);

l1.sort();l2.sort();l1.merge(l2);

print(l1.begin(), l1.end());

print(l2.begin(), l2.end()); cout<

return 0;

}

A. place 1: operator int() { return a; }

B. bool operator < (const A & b) { return this?>a< b.a;}

C. place 1: bool operator < (const A & b) { return this?>a< b.a;}

D. friend ostream & operator <<(ostream & c, const A & a);

E. place 2: ostream & operator <<(ostream & c, const A & a) { c<

ANSWER: A B C

Explanation:

Both `operator int() { return a; }` and the member `bool operator < (const A & b) { return this->a < b.a;}` supply the ordering needed by `std::list::sort()` and `std::list::merge()`. With the conversion operator, each `A` object can be converted to `int`, allowing the normal integer less-than comparison to order the list elements. That same conversion also lets the stream's integer insertion operation print each element in `print`. Consequently, after both lists are sorted, merging produces the ascending sequence from 0 through 9, while the second list is empty.

Providing the member less-than operator directly gives `A` an ordering relation that the list operations can call. Printing remains valid because an `A` can be implicitly converted to an integer for the appropriate `std::ostream` insertion overload. The option text without an explicit placement is valid when used at the intended member insertion point inside `class A`, as is the explicitly labelled equivalent member declaration. The standard library specifies that `list::sort` orders elements with `operator<` when no comparator is supplied, and `list::merge` merges already sorted lists under that ordering. See cppreference: std::list::sort and cppreference: std::list::merge.

QUESTION NO: 28

Which changes introduced independently will allow the code to compile and display 0 0 1 1 8 8 9 9 (choose all that apply)?

#include

#include #include using namespace std;

class A { int a; public:

A(int a):a(a){}

int getA() const { return a;}

/* Insert Code Here 1 */

};

/* Insert Code Here 2*/

int main(){

A t[] ={ 3, 4, 2, 1, 6, 5, 7, 9, 8, 0 }; sets(t, t+10);/* Replace Code Here 3 */

multiset s1(s.begin(),s.end());/* Replace Code Here 4 */ s1.insert(s.begin(),s.end());

s1.erase(s1.lower_bound(2),s1.upper_bound(7)); multiset::iterator i=s1.begin();/* Replace Code Here 5 */ for(;i!= s1.end(); i++)

{

cout<getA()<<" ";

}

cout<<>

return 0; }

A. operator int() const { return a; } inserted at Place 1

B. bool operator < (const A & b) const { return a < b.a; } inserted at Place 1

C. bool operator < (const A & b) const { return b.a < a; } inserted at Place 1

D. struct R { bool operator()(const A & a, const A & b) const { return a.getA() < b.getA(); } }; inserted at Place 2
Replace line 3 with `set s(t, t + 10);`
Replace line 4 with `multiset s1(s.begin(), s.end());`
Replace line 5 with `multiset::iterator i = s1.begin();`

ANSWER: A B D

Explanation:

Each correct change supplies a valid strict ordering for `A`, which is required by the ordered associative containers. Inserting `operator int() const { return a; }` makes an `A` object implicitly convertible to `int`. Consequently, the default comparison used by `set<A>` and `multiset<A>` can compare the converted integer values. Inserting `bool operator < (const A & b) const { return a < b.a; }` directly defines the natural ascending ordering for `A`, also satisfying the containers' comparison requirement. Defining `R` with a const call operator that compares `getA()` values, and consistently using `R` as the comparator for both containers and the iterator type, provides that same ascending ordering externally.

With any of these approaches, the set contains the unique values from 0 through 9. The multiset is constructed from that set and then receives the same values again, so every value occurs twice. Erasing the range from 2 through 7 removes both copies of those values. The remaining ordered elements are therefore 0, 0, 1, 1, 8, 8, 9, and 9. The standard library's ordered containers use a comparison object to establish this key order, as described in cppreference's set documentation and the Compare named requirement.

QUESTION NO: 29

Which changes introduced independently will allow the code to compile and display 0 0 1 1 8 8 9 9 (choose all that apply)?

#include

```
#include
#include
using namespace std;
class A {
int a;
public:
A(int a):a(a){}
int getA() const { return a;}
/* Insert Code Here 1 */
};
/* Insert Code Here 2*/
int main(){
A t[] ={ 3, 4, 2, 1, 6, 5, 7, 9, 8, 0 };
sets(t, t+10);/* Replace Code Here 3 */
multiset s1(s.begin(),s.end());/* Replace Code Here 4 */
s1.insert(s.begin(),s.end());
s1.erase(s1.lower_bound(2),s1.upper_bound(7));
multiset::iterator i=s1.begin();/* Replace Code Here 5 */
for( ;i!= s1.end(); i++)
{
cout<<i?>getA()<<" ";
}
cout<<endl;
return 0;
}
```

A. operator int() const { return a; } inserted at Place 1

B. bool operator < (const A & b) const { return a < b.a; } inserted at Place 1

C. bool operator < (const A & b) const { return b.a < a; } inserted at Place 1

D. struct R { bool operator()(const A & a, const A & b) const { return a.getA() < b.getA(); } }; inserted at Place 2. Replace the marked declarations with `set s(t, t + 10);`, `multiset s1(s.begin(), s.end());`, and `multiset::iterator i = s1.begin();`.

ANSWER: A B D

Explanation:

The declarations `operator int() const { return a; }` and `bool operator < (const A & b) const { return a < b.a; }` each provide a valid ascending ordering for the stored values. With either declaration, the default ordering used by `set` and `multiset` can arrange the elements by their represented integer values. The initial `set` therefore

contains the unique values from 0 through 9. Constructing the `multiset` from that set and inserting the same range again creates two copies of each value. Erasing the range from `lower_bound(2)` through `upper_bound(7)` removes both copies of every value from 2 through 7, leaving the required ordered sequence 0, 0, 1, 1, 8, 8, 9, 9.

The custom-comparator change is also valid when the comparator calls `getA()` on both operands and is used consistently as the comparator template argument for both containers and their iterator declaration. Its predicate orders values in ascending numeric order, so the associative-container range operations have the same boundaries and yield the same remaining elements. Associative containers require a comparison that establishes their ordering; see cppreference's set documentation and cppreference's multiset documentation.

QUESTION NO: 30

Which changes introduced independently will allow code to compile and display 0 1 8 9 (choose all that apply)

#include

#include #include using namespace std;

class A { int a; public:

A(int a):a(a){}

int getA() const { return a;}

/* Insert Code Here 1 */

};

/* Insert Code Here 2 */

int main(){

A t[] ={ 3, 4, 2, 1, 6, 5, 7, 9, 8, 0 }; vectorv(t, t+10); set s1(v.begin(),v.end()); s1.insert(v.begin(),v.end());

s1.erase(s1.lower_bound(2),s1.upper_bound(7)); for(set::iterator i=s1.begin();i!= s1.end(); i++) { cout<getA()<<" ";

}

cout<

}

A. operator int() const { return a;} inserted at Place 1

B. bool operator < (const A & b) const { return a

C. bool operator < (const A & b) const { return b.a

D. bool operator < (const A & a, const A & b) { return a.getA() < b.getA(); } inserted at Place 2

ANSWER: A B D

Explanation:

The changes that work each provide `std::set<A>` with an ascending ordering compatible with the integer values stored by `A`. Adding `operator int() const { return a; }` allows the built-in less-than comparison to compare two `A` objects after conversion to `int`. Consequently, the default comparator used by `std::set` orders the elements numerically from 0 through 9. Adding the member `operator<` that returns `a < b.a` directly defines that same ascending order. A non-member `operator<` placed after the class definition and returning `a.getA() < b.getA()` also supplies the required ascending comparison.

With any of these independent changes, the set contains the unique values 0 through 9 in ascending order. `lower_bound(2)` identifies 2, while `upper_bound(7)` identifies the first value greater than 7, namely 8. Erasing that half-open range removes 2 through 7 and leaves 0, 1, 8, and 9, which the loop prints in sorted order. The comparison used by an

associative container must establish a strict weak ordering; an ascending value comparison meets this requirement. See cppreference: std::set and cppreference: Compare named requirement.

QUESTION NO: 31

Which statements about the following code are correct? Choose all that apply.

```
#include <vector>

using namespace std;

int main() {
    vector<int> v;
    v.reserve(20);
    v.push_back(7);
    v.push_back(8);
}
```

A. Immediately after reserve(20), v.size() is 0.

B. After both push_back calls, v.size() is 2.

C. reserve(20) creates twenty elements initialized to zero.

D. The final capacity is guaranteed to be exactly 20.

E. The final elements are 7 and 8, in that order.

ANSWER: A B E

Explanation:

Calling `reserve(20)` requests storage sufficient for at least 20 elements, but it does not create any elements. Therefore, immediately after the call to `reserve`, `v.size()` remains zero. The two subsequent `push_back` calls append two elements, so the final size is 2 and the elements are 7 and 8.

The exact capacity is implementation-dependent, although it is guaranteed to be at least 20 after a successful `reserve(20)` call. Reserving capacity can prevent reallocation while appending elements as long as the required size does not exceed the available capacity. See cppreference: std::vector::reserve and cppreference: std::vector::push_back.

QUESTION NO: 32

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

#include

#include

#include #include using namespace std; templatestruct Out { ostream & out;

Out(ostream & o): out(o){}

void operator() (const T & val) { out<

int t[]={8, 10, 5, 1, 4, 6, 2, 7, 9, 3}; deque d1(t, t+10); set s1(t,t+10);

cout<

<<>

return 0; }

Choose all possible outputs (all that apply):

A. 1 0

B. 1 1

C. true true

D. false false

E. compilation error

ANSWER: A B

Explanation:

Assuming the displayed HTML artifacts are formatting damage and the intended source is valid C++, both `1 0` and `1 1` are possible outcomes. The `set<int>` stores its elements in sorted order, so the range supplied by `s1.begin()` and `s1.end()` satisfies the ordering requirement of `std::binary_search`. Since the value `4` is present, that call succeeds. With the default stream formatting for `bool`, a successful result is printed as `1`.

The deque retains the initializer-array order: `8, 10, 5, 1, 4, 6, 2, 7, 9, 3`. This range is not partitioned/sorted as required by `std::binary_search`. Consequently, the standard does not guarantee a meaningful result for that invocation. An implementation's binary-search steps can fail to inspect the position containing `4`, yielding `0`; another implementation or search arrangement can encounter the element and yield `1`. Therefore the first, well-defined search prints `1`, while the second printed value is not reliably determined by the program. The algorithm's ordering precondition is documented by cppreference: std::binary_search and in the C++ standard draft binary-search requirements.

QUESTION NO: 33

What will happen when you attempt to compile and run the following code? Choose all possible answers.

```
#include

using namespace std;

class B {};

template

class A {

T_v;

public:

A() {}

A(T v): _v(v){}

T getV() { return _v; }

void add(T a) { _v+=a; }

};

int main()

{

A a(1);

Ab;
```

a.add(10);

cout << a.getV() <

return 0;

}

A. program will display:11

B. program will not compile

C. program will compile

D. program will cause runtime exception

ANSWER: A C

Explanation:

Interpreting the HTML-corrupted formatting as the intended C++ source—namely `template <typename T>`, a data member `T _v;`, and a normal `cout << a.getV() << endl;` statement—the program compiles and displays `11`. The specialization `A<int>` is constructed with the value `1`. Its `add` member receives `10` and performs `_v += a`, so the stored integer becomes `11`. Calling `getV()` returns that value, and the stream insertion expression writes it to standard output followed by a newline.

The declaration `A<B> b;` is also valid. Class templates are instantiated as needed: merely declaring this object requires a usable default construction of its stored `B` member, which the empty class supplies implicitly. It does not require the `add` member for `A<B>` to be used. Therefore, the valid outcomes are that the program will compile and that it will display `11`. See cppreference: class templates and cppreference: operator expressions.

QUESTION NO: 34

What happens when you attempt to compile and run the following code? Choose all that apply.

#include

#include

#include #include

#include #include using namespace std; class B { int val; public:

B(int v=0):val(v){} int getV() const {return val;} operator int() const { return val; };};

templatestruct Out { ostream & out; Out(ostream & o): out(o){}

void operator() (const T & val) {out<<><>

int main () {

int t[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}; fstream f("test.out", ios::trunc|ios::out);

list **l(t, t+10);**

for_each(l.begin(), l.end(), Out**(f)); f.close();**

f.open("test.out"); for(; f.good() ;) { int i;

f>>i; cout<

}

f.close();

return 0; }

A. file test.out will be opened writing

B. file test.out will be truncated

C. file test.out will be opened for reading

D. no file will be created nor opened

E. program will display sequence 1 2 3 4 5 6 7 8 9 10

ANSWER: A B C

Explanation:

The file stream is first constructed with `ios::trunc | ios::out`. The `ios::out` flag opens `test.out` for output, allowing the `Out<B>` function object to send each list element to the file. The `ios::trunc` flag truncates the file when it is opened: if a file named `test.out` already exists, its previous contents are discarded before output begins. Since output mode is supplied, the file is created if necessary.

After the stream is closed, `f.open("test.out")` uses `basic_fstream::open`'s default mode, `ios::in | ios::out`. Consequently, the same file is opened with input capability, and formatted extraction using `f >> i` can read from it. The conversion operator in `B` permits each object to be inserted as an integer; however, the insertion expression contains no whitespace separator between values. The relevant selected outcomes are therefore the initial output opening, truncation, and subsequent opening with input capability. See the standard stream open-mode descriptions at cppreference: ios_base::openmode and the basic_fstream::open reference.

QUESTION NO: 35

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

#include

#include #include using namespace std;

int main(){

int myints[] ={ 3, 4, 2, 1, 6, 5, 7, 9, 8, 0 }; vectorv(myints, myints+10); set s1(v.begin(),v.end()); set > s2(v.begin(), v.end()); for(set::iterator i=s1.begin();i!= s1.end(); i++) { cout<<*i<<" ";

}

for(set >::iterator i=s2.begin();i!= s2.end(); i++) { cout<<*i<<" ";

}

cout<<>

return 0; }

A. program outputs: 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9

B. program outputs: 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0

C. program outputs: 0 1 2 3 4 5 6 7 8 9 9 8 7 6 5 4 3 2 1 0

D. program outputs: 9 8 7 6 5 4 3 2 1 0 0 1 2 3 4 5 6 7 8 9

ANSWER: C

Explanation:

The program outputs: 0 1 2 3 4 5 6 7 8 9 9 8 7 6 5 4 3 2 1 0. The HTML formatting in the displayed source has obscured the comparator type, but the intended second declaration is a `set<int, greater<int> >`. Both sets are constructed from the vector, whose range contains each integer from 0 through 9 exactly once. A `std::set` stores unique keys and iterates according to its comparison object rather than according to insertion order. The first set uses its default comparison, `std::less<int>`, so traversing it from `begin()` to `end()` prints the integers in ascending order. The second set uses `std::greater<int>`, which reverses the ordering relation; therefore, its forward iteration prints the same keys in descending order. The spaces written after each value do not alter the sequence of values shown in the result. See the cppreference documentation for std::set and the documentation for std::greater.

QUESTION NO: 36

Which statements about the code below are correct? Choose all that apply.

```
#include <vector>

using namespace std;

int main() {
    vector<int> v;
    v.push_back(10);
    v.push_back(20);
    v.push_back(30);
    v.push_back(40);

    vector<int>::iterator it = v.begin() + 1;
    v.erase(it);
}
```

A. The vector contains 10, 30, and 40 after the call to `erase`.

B. The size of the vector is 3 after the call to `erase`.

C. An iterator referring to the element 30 before the call remains valid.

D. The call to `erase` returns an iterator designating 30.

E. The vector capacity is guaranteed to become 3.

ANSWER: A B D

Explanation:

The iterator `it` initially designates the element whose value is 20. Calling `erase(it)` removes that element, reduces the vector size from four to three, and leaves the sequence 10, 30, 40.

For `std::vector`, erasing an element invalidates iterators and references at or after the point of erasure. Therefore, an iterator previously referring to 30 or 40 cannot safely be used after this call. The operation returns an iterator to the element that followed the erased one, which is 30 in this case. See cppreference: vector::erase.

QUESTION NO: 37

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

#include

#include #include using namespace std;

class A { int a; public:

A(int a) : a(a) {}

int getA() const { return a; } void setA(int a) { this?>a = a; }

};

int main () {

int t[] = {1,2,3,2,3,5,1,2,7,3,2,1,10, 4,4,5}; deque d (t,t+15);

int number = count(d.begin(), d.end(), 2); cout<< number<

}

Program outputs:

A. 4

B. 3

C. 2

D. 0

E. compilation error

ANSWER: E

Explanation:

compilation error is correct because the program as shown is not valid C++ source and therefore cannot be translated into an executable program. In particular, the body of `A::setA` contains `this?>a = a;`. To access a non-static data member through the implicit `this` pointer, the valid member-access operator is `->`, so the intended expression would be `this->a = a;`. The displayed token sequence cannot form a valid assignment expression, so a conforming compiler must issue a diagnostic during compilation. This is true even though `setA` is never called: ordinary function bodies must still be syntactically valid when the translation unit is compiled. The displayed source also contains malformed include/markup text, which likewise prevents it from being valid C++ as written. Consequently, execution never begins and there is no numeric output. The C++ language defines `this` as a pointer available in non-static member functions, and the built-in `->` operator is the appropriate way to select a member through that pointer. See cppreference: the this pointer and cppreference: member access operators.

QUESTION NO: 38

Which pieces of code inserted independently into places marked 1 and 2 will cause the program to compile and display: 0 1 2 3 4 5 6 7 8 9? Choose all that apply.

#include #include using namespace std; class A { int a; public: A(int a){ this?>a=a;}

//insert code here 1

};

//insert code here 2

template void print(T start, T end) { while (start != end) {

std::cout << *start << " "; start++;

}

}

int main() {

A t1[] ={ 1, 7, 8, 4, 5 };list l1(t1, t1 + 5); A t2[] ={ 3, 2, 6, 9, 0 };list l2(t2, t2 + 5); l1.sort();l2.sort();l1.merge(l2); print(l1.begin(), l1.end());

print(l2.begin(), l2.end()); cout<

A. place 1: operator int() { return a; }

B. place 1: operator int() { return a; } bool operator < (const A & b) { return this?>a< b.a;}

C. place 1: bool operator < (const A & b) { return this?>a< b.a;}

D. place 1: bool operator < (const A & b) { return this?>a< b.a;} friend ostream & operator <<(ostream & c, const A & a); place 2: ostream & operator <<(ostream & c, const A & a) { c<

E. place 1: bool operator < (const A & b) { return this?>a< b.a;} place 2: ostream & operator <<(ostream & c, const A & a) { c<

ANSWER: A B D

Explanation:

The insertion containing `operator int() { return a; }` is sufficient because it makes each `A` object implicitly convertible to `int`. Consequently, comparisons needed by `std::list::sort()` can use the built-in integer less-than operation after conversion. The same conversion permits `std::cout << *start` in `print` to select the integer stream-insertion overload. Each list is therefore sorted numerically, and `merge` combines the two sorted ranges into the ordered sequence from 0 through 9. The second list is empty after the merge, so its subsequent call to `print` produces no additional values.

The insertion containing both the conversion operator and `bool operator < (const A & b)` also has the required behavior. Its member comparison directly supplies the ordering used for sorting and merging, while the conversion operator supplies stream output as integers. The insertion that declares a friend stream-insertion operator, defines that operator outside the class, and supplies `operator<` likewise meets both requirements: ordering is explicitly defined in the class, and the friend declaration grants the external stream operator access to the private `a` member. This produces the requested numeric output. See cppreference: std::list::sort and cppreference: user-defined conversion function.

QUESTION NO: 39

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

#include

#include

#include <algorithm>

#include

using namespace std;

templatestruct Out {

ostream & out;

Out(ostream & o): out(o){}

void operator() (const T & val) { out<

int main() {

int t[]={8, 10, 5, 1, 4, 6, 2, 7, 9, 3};

deque d1(t, t+10);

set s1(t,t+10);

cout<

return 0;

}

Choose all possible outputs (all that apply):

A. 1 0

B. 1 1

C. true true

D. false false

E. compilation error

ANSWER: A B

Explanation:

The possible outputs are `1 0` and `1 1`. Constructing `s1` as a `set<int>` stores its elements in ascending key order. Consequently, the call to `binary_search(s1.begin(), s1.end(), 4)` searches a correctly ordered range and finds `4`, producing `1`. The stream has not been configured with `boolalpha`, so Boolean results are printed numerically rather than as the words `true` or `false`.

The deque retains the source array's order: `8, 10, 5, 1, 4, ...`, rather than becoming sorted merely because it contains integers. `std::binary_search` requires its input range to be partitioned with respect to the comparison used. Applying it to this unsorted deque violates that algorithm precondition. Therefore, it is not valid to rely on a single deterministic result from the second search; in the intended exam context, the observable Boolean result may be either `0` or `1`, while the set search remains `1`. The standard-library requirements for `binary_search` are documented at cppreference: std::binary_search, and the ordering property of associative containers is described at cppreference: std::set.

QUESTION NO: 40

What will happen when you attempt to compile and run the code below, assuming that file test.in contains the following sequence: 1 2 3?

#include

#include

#include

#include #include using namespace std; templatestruct Out { ostream & out;

Out(ostream & o): out(o){}

void operator() (const T & val) {out<

int main () { ifstream f("test.in"); list l; for(; f.good() ;) { int i;

f>>i;

l.push_back(i);

}

f.close();

for_each(l.begin(), l.end(), Out(cout)); return 0; }

Program will output:

A. 1 2 3

B. 1 2 3 3

C. no output

D. compilation error

E. program runs forever without output

ANSWER: A

Explanation:

`1 2 3` is the expected output. The loop tests the stream state before each extraction. Initially, the input stream is good, so the first three iterations successfully extract the integers `1`, `2`, and `3`, appending each value to the list. When the final integer is extracted from the stipulated input sequence, the extraction reaches end-of-file and sets the stream's end-of-file state. Consequently, `f.good()` is false at the next loop-condition evaluation, so no further value is appended. `for_each` then invokes the output function object once for every element in the list. Its call operator writes each integer followed by a space; answer formatting normally omits that trailing space, leaving the displayed output as `1 2 3`. The stream-state behavior is documented by cppreference's basic_ios::good reference, and formatted integer extraction is described at cppreference's basic_istream extraction reference.