Q 8.1)

x is a lvalue

static_cast<std::vector<int>&&>(x) is a rvalue (unnamed rvalue reference to an object is rvalue)

x.begin() is rvalue

++i is lvalue (reference to rvalue is lvalue?)

i is rvalue

*i and *i += 5 are lvalues

x[0] is a lvalue

++a is a lvalue (return *this)

a++ is a rvalue (return copy)

func1(x) is lvalue

y = func1(x) is a rvalue (unnamed rvalue reference is rvalue)

Q 8.9)

First /* ??? */

z = x + y;

x + y calls operator+(x, y). If we look at the implementation of operator+ we see it creates another new temporary object (by calling counter(x)).

The counter result of x + y is stored as
counter _temp(x+y)

z = _temp;

so the code creates two temporary objects: one at z = x + y (since operator= expects another counter) and the other inside operator+

Second /* ??? */

z = z + z;

same reasoning as first. Two temporary objects created.

Third /* ??? */

y = ++z;

No temporary objects created. prefix ++ does not create temporary objects\

Fourth /* ??? */

z = y++;

one temporary object created. postfix ++ returns value y had before the increment. Hence must create a new temp object to store that data. (the temp object here is old, created inside operator++(int))

Fifth /* ??? */

x = z;

No temporary objects.

Q 8.12)

49 Widget b(a); // Copy ctor

50 Widget c = a; // Copy assignment

51 Widget d(std::move(c)); // move ctor

52 Widget e = std::move(d); // move assignment

53 Widget f(make_widget_1()); // copy ctor

54 Widget g(make_widget_2(true)); // move ctor

55 c = a; // copy assignment

56 b = std::move(c); // move assignment

57 a = make_widget_1(); // move assignment

58 a = make_widget_2(true); // move assignment

59 func_1(a); // copy ctor

60 func_1(std::move(a)); // move ctor

61 func_1(make_widget_1()); // move ctor

62 func_2(std::move(b)); // move ctor

Q 8.28)

If capacity exceeded in array based stack, must reallocate and copy, which is time consuming.

Node based stack does not have this problem since it can create nodes on the go. Hence worst case constant time push for node based stack vs amortized constant time for array based stack.

Since node based stack also store pointer to next member (i.e., per element storage overhead) they generally take up more space than array based stacks.

Since array based stack stores elements contiguously in memory, other elements in the stack can be accessed using an offset in constant time in array based stack. But to access other elements in node based stack it is O(n), where n is number of elements.

But the offset needs to be modified every time a push or pop is performed on an array based stack. This is not needed in node based stack since the element references are stable.