---

a).

Variable:
A variable is a location in the computer where the user can store some value which can later be retrieved. Variables have a name as well as a type.

Reference:
A reference variable is an alias, that is, another name for an already existing variable. Once a reference is initialized with a variable, either the variable name or the reference name may be used to refer to the variable.

Pointer:
A pointer is a programming language object, whose value refers to (or "points to") another value stored elsewhere in the computer memory using its address.

The following are the differences between references and pointers:
A pointer can be re-assigned any number of times while a reference can not be re-seated after binding.

1. Pointers can point nowhere (NULL), whereas reference always refer to an object.
2. You can't take the address of a reference like you can with pointers.
3. There's no "reference arithmetics" (but you can take the address of an object pointed by a reference and do pointer arithmetics on it as in &obj + 5).

b).

Output:

```
0x7fffffffe010; 0x7fffffffe010; 0x7fffffffe010
10; 10; 10
15; 15; 15
50; 50; 50
10; 10; 10
```

Explanation:

The first row prints the memory address of the variable where the value is stored. We can see that the address is the same for all the three which means that the reference and the pointer are pointing to the same variable.

In the second cout the value of the variable is changed to 15 using the pointer. As the value at the memory location is changed the changing is reflected in the number, reference and the pointer.

In the third cout the value of the reference is set to 50. This has the same effect as all of the three are representing the same memory address we see all the outputs are 50.

In the fourth cout the value of the number is changed and as the same argument as before the change is reflected in all the three number, pointer and reference as all three are representing the same memory location.

c).

Output:
0
10
20

Explanation:
(Explain call by value, call by reference and call by pointer)

Call by value function takes a value and changes it to 5 locally while not changing the actual value passed to it.

Call by reference function takes the reference to the value passed to it and changes it to 10, change is made to the original value because now the change is made to the same location in the memory.

Call by pointer takes the address of the value and dereferences it and assigns 20 to it. As the value is changed on the memory location so the original value of the variable is modified.

(What exactly is done on calling?)

When callByValue function is called, a copy of the value is passed to the function and changing this value inside of the function has no effect to the original value outside the function.

When callByReference function is called, the refernce of the value is passed to the function so any changing made to this value inside the function changes the original value because now the change is being made to the same location in the memory.

The reason is the same for the callByPointer function.

(Give detailed runtime costs for large Objects?)

Call by value copies the entire object and passes it over to the function, so it tends to me more expensive as compared to call by reference or call by pointer.

(Which objects can be written to?)

In the callByValue function only the value of the copied argument can be changed.
In the callByReference function, the reference can be pointed to some other value but not to NULL and the original referenced value can be altered.
In the callByPointer function the pointer can be pointed to some other value or to NULL and the value of the original pointed value can be changed.

d).

(Understand and explain the following code.)

```
double array[5] = { 10.0, 20.0, 30.0, 40.0, 50.0 }; // array with five values is initialized
double* begin = array; // begin pointer starts pointing to head of the array
std::cout << *(begin + 0) << std::endl; // first item is printed
std::cout << *(begin + 1) << std::endl; // second item is printed
std::cout << *(begin + 2) << std::endl; // third item is printed
double* end = array + 5; // address where 6th item could have been is assigned to end pointer
int diff = end - begin; //  integer diff stores the difference of end and begin variable locations (which is 5)
diff /= 2; // 5/2 comes out to be 2
double value2 = *(begin + 1) + *(end - (diff)); // 20 + 40 = 60 ... 20 is second element, 40 comes if we subtract
// diff number of location from the sixth element's address
std::cout << value2 << std::endl; // 60 is printed
std::cout << *end << std::endl; // address of 6th element (which was initially not the part of the array and hence
// contains garbage value) is printed
```

(What are the results?)

Output:

```
10
20
30
60
4.34385e-28
```

(Is there undefined behavior?)

The last cout tries to print out a value which is not the part of the array and was never initialized, so that value was undefined and any garbage value is printed in its place.

e).

(Understand and explain how the elements of the array are accessed without using operator[]:)

```
double* ptr = begin;    // ptr pointer starts pointing to where begin points which is the start of the array
while (ptr != end) {    // this while loop runs until ptr starts pointing to where end pointer is pointing
    std::cout << *(ptr ++) << " ";
}
std::cout << std::endl; // adding a new line
ptr = end;              // ptr points to the place where end points (which is one after the last element of array)
while (ptr != begin) {  // this while loop breaks when the ptr points to where begin is pointing
    std::cout << *(-- ptr) << " ";  // this cout decreases the pointer value before cout runs
} ;
std::cout << std::endl; // adding a new line
```

f).

(Which versions work?)

All of the versions work, none of them breaks, although the last version gives prints all the elements

(What problems do the others have?)

In the first loop, the ptr starts already at the end of the array position and while printing the '--' is put infront of the ptr which means that the pointer will first move one position behind and then print the value. This is the reason why the loop misses the last value while printing.

In the second loop, the stopping condition of the loop is the problem. The loop exists without printing the last value. If the condition is changed to ptr !- begin-1 then it will work fine.

The third loop is OK and gives the correct output.