Solution 1

a) Output: 2

Explanation: \( p \) points to the element with index 2 and whenever \( a \) is converted to a pointer it points to the element with index 0. What we are outputting is simply the distance between those two pointers, which is 2.

b) Output: 0 1 2 3 4 5 6 7 8 9

Explanation: We go from the front to the back and output each element on the way. Thus this simply corresponds to outputting the array.

c) Output: 5 4 1 6 0

Explanation: In the loop we first output the current element pointed to by \( p \) and then let \( p \) point to another element. The new element is obtained by shifting a pointer to the begin (i.e. \( a[0] \)) \(*p\) times. For example, if \(*p\) currently has value 4, the new element is obtained by shifting a pointer to \( a[0] \) exactly 4 times (which corresponds to \( a[4] \), i.e. the fifth element). We stop this procedure once \( p \) is pointing again to \( a[0] \).

d) Output: Hello

Explanation: In the first loop we shift the value of each element by \(-1\). If the ASCII-code is used (which it is almost everywhere) to represent the characters as ints, the shifted values correspond to the characters in the alphabet placed right before the current ones (e.g. \( b \mapsto a \), \( n \mapsto m \), etc.). Furthermore, the shift respects capitalization. Thus the first loop operates as follows:

\[
\begin{align*}
\text{Hfmp} \\
\text{Hemmp} \\
\text{Helmp} \\
\text{Hellp} \\
\text{Hello}
\end{align*}
\]
Solution 2

```cpp
#include "tests.h"
#include <iostream>

int main ()
{
    // Input
    const int n = 10;
    int inputs[n];
    for (int* i = inputs; i < inputs + n; ++i)
        std::cin >> *i;

    // Output
    std::cout << "* ";
    for (int* i = inputs; i < inputs + n; i += 2)
        std::cout << *i << " ";
    std::cout << "*\n";

    return 0;
}
```

```cpp
#include "tests.h"
#include <iostream>

int main ()
{
    // Input
    const int n = 10;
    int inputs[n];
    for (int* i = inputs; i < inputs + n; ++i)
        std::cin >> *i;

    // Output
    int* i = inputs + n;
    std::cout << "* ";
    while (inputs < i) {
        --i;
    }
```
```cpp
std::cout << *i << " ";
}
std::cout << " \n";

// Remark: The above code makes sure that i is never shifted
// "before the front". The problem with that is that
// unlike the past-the-end pointer this pointer does
// not have to exist. In most cases this still works
// for pointers on arrays, but if a vector-iterator is
// shifted before the first element (i.e. index 0) of
// a vector, this can yield an error depending on the
// compiler flags that are set.

return 0;
```

```cpp
b) // Informatik – Serie 9 – Aufgabe 2c
// Programm: average.cpp
// Autor: ... (Gruppe ...)

#include "tests.h"
#include <iostream>
#include <cassert>

// PRE: [begin, end) is a valid non-empty range.
// POST: returns the average of the elements in [begin, end)
double average (const double* begin, const double* end)
{
    assert(end - begin > 0);
    const unsigned int length = end - begin;
    double sum = 0;
    while (begin < end) {
        sum += *begin;
        ++begin;
    }
    return sum / length;
}

int main ()
{
    // Input numbers
    const unsigned int n = 20;
    double inputs[n];
    for (double* i = inputs; i < inputs + n; ++i)
        std::cin >> *i;
    // Output average
```
std::cout << average(inputs, inputs + n) << "\n";

return 0;
}
Solution 3

```cpp
#include <algorithm>
#include <iostream>
#include <cassert>
#include <vector>

typedef std::vector<int>::iterator Vit;

// PRE: [begin, end) is a valid range
// POST: the elements in [begin, end) are in ascending order
void sort(Vit begin, Vit end)
{
    // There are many possible ways to sort a range. Here we
    // proceed as follows: We shift the iterator begin through the
    // vector. In each round we find the smallest element among
    // those elements with larger (or equal) index and swap it
    // with the one pointed to by begin.
    while (begin < end) {
        // Find minimum
        Vit p_min = begin; // iterator pointing to current minimum
        for (Vit curr = begin; curr < end; ++curr)
            if (*curr < *p_min)
                p_min = curr;
        // Swap with current value
        std::swap(*begin, *p_min);
        // Advance
        ++begin;
    }
}

int main()
{
    // Input
    int n;
    std::cout << " *\n";
    return 0;
}
```cpp
std::cin >> n;
std::vector<int> a(n);
for (int i = 0; i < n; ++i)
    std::cin >> a[i];

// Sort
sort(a.begin(), a.end());

// Output sorted sequence
std::cout << "* ";
for (int i = 0; i < n; ++i)
    std::cout << a[i] << " ";
std::cout << " *\n";
return 0;
}
```

Solution 4

d) // PRE: n >= 0
   // POST: return value is false if n is even and true if n is odd

b) // PRE: n >= 0
   // POST: 2^n stars have been written to standard output

c) // PRE: [begin, end) is a valid range
   // POST: outputs for all subsets in [begin, end) the sum
   //       of its values shifted by t. (The empty set is
   //       defined to have sum 0 here.)

Remark: in many cases the user would call this function for t = 0 as this outputs the
unshifted subset sums. Therefore, we can write a dedicated function to output all possible
subset sums in a given range. This way the user can just pass the range and doesn’t have to
set t manually, which is more fail-safe. All this function has to do is to call our function for
the given range and t = 0.