Solution 1

a) Output: 5

What happens is that the argument \( i \) of \( \text{func}_a \) is a local variable with the same name as the variable in the \text{main}-function. But the two are different variables. And \( ++i \) modifies the local variable. Notice that \( \text{func}_a \) doesn’t do anything!

b) Output: \( x=6 \ y=8 \)

First, notice that the only thing that modifies \( x \) is the \( x++ \) in the function call. This adds 1 to the value of \( x \) (and passes the old value of \( x \) to the argument \( b \)). Thus \( x=6 \) is output.

On the other hand, \( y \) is the return value of the function \( \text{func}_b \) that is called with the arguments \( a \) as 3 and \( b \) as 5. Since the manipulations done to \( a \) and \( b \) in \( \text{func}_b \) are local to the body of \( \text{func}_b \) it evaluates to \( (a + 1) + (b - 1) = (3 + 1) + (5 - 1) = 8 \). Thus the second part of the output is \( y=8 \).

c) Output: Nope!

Before arguing any further, notice that as seen above (since \( (a + 1) + (b - 1) = a + b \)) a valid POST-condition for \( \text{func}_b \) could be for example:

```
// POST: returns a+b
```

Now about the actual subtask: first, the argument \( x \) of \( \text{func}_c \) is initialized with value 0. In the first step \( \text{func}_b \) computes \( 3+0 \) which is 3. This is assigned to \( x \). In the next step the new value for \( x \) is used by \( \text{func}_b \) to compute \( 3+3 \) which is 6. Again, \( x \) gets this as its new value. In the third call, \( \text{func}_b \) computes \( 3+6 \) which is 9. Again, \( x \) gets this as its new value. Finally, we return the result of \( 9 > 10 \), which is obviously false.

d) Output: That’s undefined...

This is a tricky one because \( \text{tmp} \) in the body of \( \text{func}_d \) is not assigned any value. Therefore the behavior is unspecified. In your own programs you should always make sure that variables are never read before they got a value assigned at least once.
Solution 2

Please keep in mind that once again there are multiple ways to solve each subtask. Provided below are code snippets you can insert in each Codeboard template that solve the respective task.

a) Remark: modulus of negative numbers are considered negative. For example: \(-3 \% 2 \rightarrow -1\) but \(3 \% 2 \rightarrow 1\)

```cpp
// POST: returns true if and only if a is not divisible by 2 and
// returns false otherwise.
bool is_odd (int a) {
  return a % 2 != 0;
}
```

b)

```cpp
// POST: returns false if and only if both a and b are true, and
// returns true otherwise.
bool nand (bool a, bool b) {
  return !(a && b);
}
```

c)

```cpp
#include <cassert>

// PRE: lower <= upper
// POST: outputs
//      lower, lower+1, ..., upper-1
//      if lower < upper, and outputs nothing if lower == upper.
void output_range (int lower, int upper) {
  assert(lower <= upper);
  for (int i = lower; i < upper; ++i)
    std::cout << i << " ";
}
```

Solution 3

a) See b)

b)
#include <iostream>
#include <cmath>

// POST: return value is true if and only if n is prime
bool is_prime (unsigned int n)
{
    if (n < 2) return false; // 0 and 1 are not prime

    // Computation: test possible divisors d up to sqrt(n)
    double bound = std::sqrt(n);
    unsigned int d;
    for (d = 2; d <= bound && n % d != 0; ++d);
    return d > bound;
}

int main ()
{
    // obtain n
    int n;
    std::cin >> n;

    // keep primality info for odd i and i+2
    bool curr = false; // i = 1
    bool next = true;  // i = 3
    int twins = 0;     // number of twins
    for (int i = 3; i <= n-2; i += 2) {
        curr = next;     // i
        next = is_prime(i+2); // i+2
        if (curr && next) ++twins;
    }
    std::cout << "Number of twin primes: " << twins << "\n";
    return 0;
}

Solution 4

If your usage of functions differs from the solution below, your solution does not have to be wrong. Especially, it is correct if your reasoning for your usage of functions is good.
// outputs all perfect numbers up to a given input number n

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

// POST: return value is the sum of all divisors of i
// that are smaller than i
unsigned int sum_of_proper_divisors (unsigned int i) {
    unsigned int sum = 0;
    for (unsigned int d = 1; d < i; ++d)
        if (i % d == 0) sum += d;
    return sum;
}

// POST: return value is true if and only if i is a
// perfect number
bool is_perfect (unsigned int i) {
    return sum_of_proper_divisors (i) == i;
}

int main() {
    // input
    unsigned int n;
    std::cin >> n;

    // computation and output
    std::cout << "The following numbers are perfect: ";
    for (unsigned int i = 1; i <= n; ++i)
        if (is_perfect (i))
            std::cout << i << " ";
    std::cout << "\n";
    return 0;
}

We chose to use the functions sum_of_proper_divisors and is_perfect because they both implement clearly specified tasks. Furthermore, we can make the main-function more readable using the function is_perfect: without it one would probably nest a loop similar to the one at line 10 into the loop at line 32. Such nested loops become unreadable very quickly. Another point which we would like to highlight is that it is also important to choose good names for your functions. The names sum_of_proper_divisors and is_perfect immediately tell the reader what the functions do. If we had named our functions f and g instead, the reader would need much more time to understand what these functions do.