Solution 1

a) Types and values:

i) Type: double
   Value: 1.5

ii) Type: bool
    Value: false
    Note that since the arr is of type int, the second argument (with index 1) is not equal to 1.5 but rather to 1 as the conversion from double to int works by rounding towards 0 (as long as it’s in the value range of int).

iii) Type: double
     Value: 0.75

iv) Type: int
    Value: 0
    First notice that str.a has value 1, thus arr[str.a] evaluates to arr[1] which is 1 (notice again that this is 1 and not 1.5) and of type int. We divide this by 2 also of type int resulting in 1/2 which is 0 as integer-division is performed.

v) Type: double
    Value: -1.5
    First *a / 2 is computed which corresponds again to 1/2 evaluating to 0 due to integer-division. From this we subtract str.b which is equal to 1.5 of type double. This results in -1.5 of type double.

Remark: In the line int arr[] = {1, 1.5, 3}; the conversion from double to int mentioned above is performed. We would like to add the remark that conversions in such initializations are considered valid or not valid depending on the C++-version used. Even though the exercise here assumes that this is valid, we would like to emphasize that such things have to be avoided as they can lead to well hidden mistakes.

b) Output: 1 2 3 4 5 3 2
   Explanation: The add member simply outputs its argument to the terminal and increments an internal counter n. Thus the first five outputs correspond to the five calls to add. But notice that both instances hi and hello have separate counters n. Thus the last two
outputs correspond to how often we called add on each of the two instances hi and hello respectively.

c) For add:
   // PRE: c has internal representation between 0 and 127
   // (including both)
   // POST: counted the call for the given char-value

For get:
   // PRE: c has internal representation between 0 and 127
   // (including both)
   // POST: returns how often add was called with the given char-value

Solution 2

Below only the class definitions and operators are printed. These can be inserted into the provided Codeboard-templates.

a) The class Averager:

```cpp
class Averager {
public:
   Averager () : n(0), s(0.0) {}

   // POST: considered v for the current average
   void add_value (const double v) {
      s += v;
      n += 1;
   }

   // PRE: n > 0
   // POST: returns the current average
   double average_value () const {
      assert (n > 0);
      return s/n;
   }

private:
   unsigned int n;       // number of values
   double s;             // sum of values
};
```

b) The class Vector and the operator-overloads:
class Vector {
public:
    Vector (const double x, const double y, const double z) {
        coord[0] = x;
        coord[1] = y;
        coord[2] = z;
    }

    // PRE: char is either 'x', 'y' or 'z'
    // POST: returns the corresponding coordinate of the vector
    double get (const char i) const {
        assert(i == 'x' || i == 'y' || i == 'z');
        return coord[i - 'x'];
    }

private:
    double coord[3];
};

// POST: returns the sum of the two vectors
Vector operator+ (const Vector l, const Vector r) {
    double coord[3];
    for (char c = 'x'; c <= 'z'; ++c)
        coord[c - 'x'] = l.get(c) + r.get(c);
    return Vector(coord[0], coord[1], coord[2]);
}

// POST: returns the product of s and r, where the scalar s is the
// left operand
Vector operator* (const double s, const Vector r) {
    return Vector(s * r.get('x'), s * r.get('y'), s * r.get('z'));}

// POST: returns the product of s and l, where the scalar s is the
// right operand
Vector operator* (const Vector l, const double s) {
    return s * l;
}

// POST: vec has been written to out
std::ostream& operator<< (std::ostream& out, const Vector vec) {
    std::cout << "("
    for (char c = 'x'; c < 'z'; ++c)
        out << vec.get(c) << ", ",
    std::cout << vec.get('z') << ")";
    return out;
}
Solution 3

We implement the second version, the one that returns the normalization of $x$. This one has the advantage that it works for rvalues.

The modification of the function $\text{gcd}$ is as easy as it can be: we only need to replace the type $\text{unsigned int}$ by $\text{int}$ in the parameter and return types. Why does this still work? Let us go back to the proof of Lemma 1 in Section 2.10 (page 251) of the script. Going through it, we realize that we never used nonnegativity of either $a$ or $b$, so the statement extends to all pairs of integers with $b \neq 0$. It remains to prove termination. For this, we show that $|a \text{ mod } b| < |b|$, so we indeed make progress towards termination.

Recall that

$$a \text{ mod } b = a - (a \text{ div } b)b,$$

and that this equation also holds in C++. Furthermore, division may round up or down (we don’t know), but in either case, the rounding makes a mistake of strictly less than 1. This means that

$$\frac{a}{b} - (a \text{ div } b)$$

has absolute value smaller than 1, and this implies (by multiplying with $b$) that

$$a - (a \text{ div } b)b = a \text{ mod } b$$

has absolute value smaller than $|b|$.
struct rational {
    int n;
    int d; // INV: d != 0
};

// POST: a has been written to o
std::ostream& operator<<(std::ostream& o, const rational a) {
    return o << a.n << "/" << a.d;
}

// PRE: i starts with a rational number of the form "n/d"
// POST: a has been read from i
std::istream& operator>>(std::istream& i, rational& a) {
    char c; // separating character, e.g. '/'
    return i >> a.n >> c >> a.d;
}

// POST: return value is the greatest common divisor of a and b
int gcd (const int a, const int b) {
    if (b == 0) return a;
    return gcd(b, a % b); // b != 0
}

// POST: return value is the normalization of r
rational normalize (const rational& r) {
    const int g = gcd (r.n, r.d);
    rational result;
    result.n = r.n / g;
    result.d = r.d / g;
    if (result.d < 0) {
        result.n = -result.n;
        result.d = -result.d;
    }
    return result;
}

int main () {
    rational r;
    std::cin >> r;
    std::cout << normalize(r) << "\n";
Solution 4

Note: there are many ways to implement these. All code pieces in the following can be inserted into extended_stack.h. We can for example implement the following members in the public-area of the class:

```c
// POST: remove the element i wherever it occurs in *this. If
// it doesn't occur, then *this is not modified.
void remove (const int i)
{
    list_node* prev = 0;
    list_node* curr = top_node;
    while (curr != 0)
    {
        if (curr->key == i) {
            // skip current element...
            if (curr == top_node)
                top_node = curr->next;
            else
                prev->next = curr->next;
            // ... and delete it
            list_node* tmp = curr;
            curr = curr->next;
            delete tmp;
        } else {
            prev = curr; // pass without removing
            curr = curr->next;
        }
    }

    // POST: number of elements in *this is returned
    unsigned int size() const
    {
        unsigned int n = 0;
        const list_node* p = top_node;
        while (p != 0) {
            ++n;
            p = p->next;
        }
        return n;
    }

    // POST: returns true if and only if *this and s2 contain the
    // same keys in the same order
    bool equals (const extended_stack& s2) const
```
```c
const list_node* p1 = top_node;
const list_node* p2 = s2.top_node;
while (p1 != 0 && p2 != 0) {
    if (p1->key != p2->key)
        return false;
    p1 = p1->next;
    p2 = p2->next;
}
return (p1 == 0 && p2 == 0);
```

And for the operator== an implementation outside the class which calls equals:

```c
// POST: returns true if and only if s1 and s2 contain the same keys in the same order
bool operator==(const extended_stack& s1,
                const extended_stack& s2)
{
    return s1.equals(s2);
}
```

Pay attention to insert this code piece still inside the namespace ifmp as otherwise the arguments of operator== would have to refer to ifmp::extended_stack instead of just extended_stack.