


master

AVR-Sandbox / HelloOperations / README.md

Go to file

...



Scrappers-glitch

HelloOperations/README.md: added an example to the basic mod function

Latest commit fc1c976 4 minutes ago

History

1 contributor

425 lines (328 sloc)

19 KB

<>

File icon

Raw

Blame

Edit icon

Dropdown arrow

Copy icon

Delete icon

Hello Operations in C

Table of contents:

- Expressions
- The Standard Assignment Operator
- The Compound Assignment Operators
- Incrementing and Decrementing
- Arithmetic Operators
- Comparison Operators
- Logical Operators
- Bitwise Logical Operators
- Bit Shifting
- Pointer Operators
- DIY

Expressions:

Syntax: l-value = first-operand [OPERATOR] second-operand ...

An operand expression consists of at least one or more operands at the r-value.

Operands are typed objects to be operated on, such as: constants, variables, and return-values.

In case of operations, the `r-value` is evaluated to the `l-value`.

The Standard Assignment Operator:

Syntax: `l-value = r-value`

The standard assignment operator `=` assigns the r-value (right operands) to the l-value (left operand).

For example:

```
const char* debugFileName = "jme3-alloc-debug.log"; /* declare and assign the file */
```

The Compound Assignment Operator:

Syntax: `l-value [OPERATOR]= r-value`

The compound assignment operators perform an operation involving both the `l-value` and `r-value`, and then assign the resulting expression to the `l-value`.

Here is the list of compound assignment operators and examples:

```
#include<stdio.h>

int main(void* args) {
    int x = 0;
    int y = 0;

    x += 100; /* is the same as [x = x + 1] */
    printf("%i\n", x);

    x -= 2; /* is the same as [x = x - 1] */
    printf("%i\n", x);

    x *= y; /* is the same as [x = x * y] */
    printf("%i\n", x);

    x += 100;
    x = (x + 100) / 2;
    printf("%i\n", x);

    char binary = 0b00000001;

    binary <=< 1; /* Left bitwise shift compound assignment operator => 0b00000001 << 1 = 0b00000010 = 2 */

    printf("%b\n", binary);

    binary >>= 1; /* Right bitwise shift compound assignment operator => 0b00000010 >> 1 = 0b00000001 = 1 */
```

```
printf("%b\n", binary);

binary = binary << 2; /* shifts the binary integer to the left 2 times, it's a simple assignment equivalent to the com

printf("%b\n", binary);

return 0;
}
```

Output:

```
100
98
0
100
10
1
```

Incrementing and Decrementing:

Syntax: ++[l-value]; , --[l-value]; , [l-value]++; , [l-value]--;

Both ++[l-value] and --[l-value] are known as pre-fix increment and decrement respectively and they are equivalent to l-value += 1 and l-value -= 1.

Remember: l-value += 1 is the same as l-value = l-value + 1.

Both [l-value]++ and [l-value]-- are known as post-fix increment and decrement respectively and they are equivalent to adding 1 and subtracting 1 respectively, but the value is evaluated to the l-value on the next-line.

For example:

```
#include<stdio.h>

int main(void* args) {
    int x = 5;
    printf("%d \n", x++); /* print x then increments it, x stills [5] here */
    /* x is now incremented by to be [6] */
    printf("%d \n", ++x); /* increments x then print it, x is now [7], notice this is equivalent to `x += 1` and `x = x +
    return 0;
}
```

Arithmetic Operators:

Syntax: `l-value = 1st-Operand [OPERATOR] 2nd-Operand [OPERATOR]`

- ADDITION `l-value = 1st-Operand [+] 2nd-Operand` : adds the 2 operands and the result is evaluated to the `l-value` .
- SUBTRACTION `l-value = 1st-Operand [-] 2nd-Operand` : subtracts the second operand from the first and the result is evaluated to the `l-value` .
- MULTIPLICATION `l-value = 1st-Operand [*] 2nd-Operand` : multiplies both operands and the result is evaluated to the `l-value` .
- DIVISION `l-value = 1st-Operand [/] 2nd-Operand` : divides the first operand by the second operand and the result is evaluated to the `l-value` .
- MODULUS `l-value = 1st-Operand [%] 2nd-Operand` : returns the remainder of dividing the first operand by the second operand and the result is evaluated to the `l-value` .

Here is an example demonstrating the arithmetic operations in C:

```
#include<stdio.h>

int main(void* args) {
    int n0 = 2;
    int n1 = 4;

    printf("Tests additon: %d\n", n0 + n1); /* returns 6 */
    printf("Tests subtraction: %d\n", n0 - n1); /* returns -2 */
    printf("Tests multiplication: %d\n", n0 * n1); /* returns 8 */
    printf("Tests division: %d\n", n0 / n1); /* Notice: this returns 0, because we are evaluating the result to an integer */
    printf("Tests division: %f\n", ((float) n0) / n1); /* Notice: this returns 0.50000, because we are evaluating the result to a float */
    printf("Tests division: %f\n", n0 / ((float) n1)); /* Notice: this returns 0.50000, because we are evaluating the result to a float */
    printf("Tests modulus: %d\n", n0 % n1); /* returns 2, because n0 is less than n1 */

    /** How the modulus function work under-the-hood ? */
    /**
     * The modulus operation operates on integers only.
     * It finds the remainder of a division operation without going over the denominator.
     *
     * For example: (remember, we are performing an integer division, forget about fractions here...)
     * 1 / 4 = 0 R 1, as (1) cannot be divided equally into (4) pieces, so it is retained in the remainder.
     * 2 / 4 = 0 R 2, as (2) cannot be divided equally into (4) pieces, so it is retained in the remainder.
     * 3 / 4 = 0 R 3, as (3) cannot be divided equally into (4) pieces, so it is retained in the remainder.
     * 4 / 4 = 1 R 0, as (4) can be divided equally into (4) pieces with no remainder, so (zero) is retained in the remainder.
     * 5 / 4 = 1 R 1, as (5) can be divided equally into (4) pieces with a remainder of 1 which is retained in the remainder.
     * 6 / 4 = 1 R 2, as (6) can be divided equally into (4) pieces with a remainder = numerator - denominator = 6 - 4 = 2
     * 7 / 4 = 1 R 3, as (7) can be divided equally into (4) pieces with a remainder of (7-4) which is 3.
     * 8 / 4 = 2 R 0, as (8) can be divided equally into (4) pieces with no remainder, so zero is retained in the remainder.
     *
     * AND SO ON....
     *
     * TASK: Create a loop that displays the above example on the console.
     */
}
```

```
return 0;
}
```

Remember: you can always utilize the arithmetic operations in compound assignment operations, but be careful as this changes the original value of the l-value.

Comparison Operators:

Syntax: `l-value = 1st-Operand [OPERATOR] 2nd-Operand`

The comparison operators used to compare 2 operands and returns 1 if the condition is met and 0 if the condition isn't true.

- IS-NOT-EQUAL `!=`: Tests whether both operands are not equal, if this condition is met, the expression returns 1 (true), 0 otherwise.
- IS-EQUAL `==`: Tests whether both operands are equal, if this condition is met, the expression returns 1 (true), 0 otherwise.
- IS-GREATER-THAN-OR-EQUAL `>=`: Tests whether the first operand is greater than or equal to the second operand, if this condition is met, the expression returns 1, 0 otherwise.
- IS-LESS-THAN-OR-EQUAL `<=`: Tests whether the first operand is less than or equal to the second operand, if this condition is met, the expression returns 1, 0 otherwise.
- IS-GREATER-THAN `>`: Tests whether the first operand is greater than the second operand, if this condition is met, the expression returns 1, 0 otherwise.
- IS-LESS-THAN `<`: Tests whether the first operand is less than the second operand, if this condition is met, the expression returns 1, 0 otherwise.

Here is an example demonstrating the comparison operators:

```
#include<stdio.h>

int main(void* args) {
    int n0 = 5;
    int n1 = 5;

    printf("Tests whether 2 numbers are equal, %d\n", n0 == n1); /* returns 1 */
    printf("Tests whether 2 numbers are not equal, %d\n", n0 != n1); /* returns 0 */
    printf("Tests whether first is greater than second, %d\n", n0 > n1); /* returns 0 */
    printf("Tests whether first is greater than or equal second, %d\n", n0 >= n1); /* returns 1 */
    printf("Tests whether first is less than or equal second, %d\n", n0 <= n1); /* returns 1 */

    printf("Tests whether 3 numbers are equal, %d\n", n0 == n1 == 1); /* tests whether the return from this comparison (n0
    return 0;
}
```

Using the comparison operators in conditional statements:

```
#include<stdio.h>

int main(void* args) {
    int n0 = 5;
    int n1 = 5;

    if (n0 == n1) { /* This executes if the conditional expression (n0 == n1) evaluates to 1 */
        printf("The 2 numbers are equal \n");
    }

    if (n0 != n1) { /* This executes if the conditional expression (n0 != n1) evaluates to 1 */
        printf("The 2 numbers aren't equal \n");
    }

    if (n0 > n1) { /* This executes if the conditional expression (n0 > n1) evaluates to 1 */
        printf("The first number is greater than the second \n");
    }

    /* and so on */

    return 0;
}
```

Logical Operators:

Syntax: `l-value` = `Expression-0` `[OPERATOR]` `Expression-1`

The logical operators test the truth value of a pair of operands, any non-zero expression (-1, -2, -3,, 1, 2, 3, 4,.....) is considered true in C, while zero expressions are considered false in C.

- Logical AND `&&`: Tests whether both operands are true, if first operand is false, the condition returns with 0 (false) and the second operand isn't evaluated.
- Logical OR `||`: Tests whether one of the operands is true, if the first operand is true, the condition returns with 1 (true) and the second operand isn't evaluated.
- Logical NOT `!`: Flips the conditional expression, if !(1) then the expression is evaluated to 0 (means false), if !(0) then the expression is evaluated to 1 (means true).
- Logical Combinatorial operators: a compound of mutiple expressions, for example: `(n0 == n1) && ((n2 < n3) && (n2 == 2))`, the most inner parenthesis is evaluated first.

Here is an example demonstrating:

```
#include<stdio.h>

int main(void* args) {
    int n0 = 5;
    int n1 = 5;
```

```
int n2 = 6;
int n3 = 8;

/** The logical AND operator */
printf("Tests the truth value of a pair of operands: %d\n", (n0 == n1) && (n2 < n3)); /* returns 1 as both conditions
/* notice: in this case the first expression is false, so the second expression is not evaluated and the l-value is ev
printf("Tests the logical AND when the first operand is [false]: %d\n", (n0 != n1) && ((n2 / 0) == 0)); /* returns 0,
/** Logical AND ends */

/** The logical OR operator */
printf("Tests the truth value of a pair of operands: %d\n", (n0 != n1) || (n2 < n3)); /* returns 1 as the second opera
/* Notice: in this case the first expression is true, so the second expression is not evaluated and the l-value is eva
printf("Tests the logical OR when the first operand is [true]: %d\n", (n0 == n1) || ((n2 / 0) == 0)); /* returns 1 as
/** Logical OR ends */

/** The logical NOT operator */
printf("Tests the NOT operator: %d\n", !((n0 != n1) || (n2 < n3))); /* returns 0, as the evaluation is flipped ! */
/* Notice the different, in this case, the internal n0 != n1 is evaluated first and then the NOT is applied to this co
printf("Tests the NOT operator on the first operand: %d\n", !(n0 != n1) || (n2 < n3)); /* returns 1, as the l-value is
/** Logical NOT operator ends */

/** TASK: Create 2 integers (n0, n1) such that n0 equals n1 and test this code:
*
* printf("Test Logical AND: %d\n", (n0 != n1) && n0++); /* Remember: the AND logic gates need both operands to evalua
* printf("Print n = %d\n", n0);
* printf("Test Logical OR: %d\n", (n0 == n1) || ++n1);
* printf("Print n = %d\n", n1);
*
* Notice the increment and the decrement expressions are not evaluated in this example; because the logical expressio
*/

return 0;
}
```

Bitwise Logical Operators:

Syntax: l-value = Expression-0 [OPERATOR] Expression-1

4) Bitwise operations:

- Logical operations:

Gate	Notation	Maths Notation	Usage	Boolean Expression
AND	(...&...)	Q = A . B	Multplying 2 binary digits	Q = A & B

Gate	Notation	Maths Notation	Usage	Boolean Expression
OR	(...OR...)	$Q = A + B$	Adding up 2 binary digits	$Q = A \text{ OR } B$
NOT	$\sim(\dots)$	$Q = !A$	Flipping the binary bits (finding the 1s complement of a binary number)	$Q = \sim A$
XOR (Ex-OR)	(...^...)	$Q = (!A) \cdot B$ OR $(A \cdot \sim B)$	Checking if 2 binary digits' bits aren't the same	$(\sim(A) \& B) \text{ OR } (A \& \sim B)$
N-XOR (NOT-Ex-OR)	$\sim(\dots^{\sim}\dots)$	$Q = !((!A) \cdot B) \text{ OR } (A \cdot \sim B))$	Checking if 2 binary digits' bits are the same	$\sim((\sim(A) \& B) \text{ OR } (A \& \sim B))$
N-AND	$\sim(\dots\&\dots)$	$Q = !(A \cdot B)$	State and circuit control when 2 states are true, the result is false and when the two states are false or one of them is false, the result is true	$Q = \sim(A \& B)$
N-OR	$\sim(\dots\text{OR}\dots)$	$Q = !(A + B)$	-	$Q = \sim(A \& B)$

Here is an example in C:

```
#include<stdio.h>
#include<inttypes.h>

/** uint8_t is defined as unsigned char */
/**
 * Prints a [substrate] in a new-line with a base radix.
 *
 * @param substrate an unsigned 8-bit integer to print
 * @param radix the base to use, 2 for binary, 16 for hexadecimal, otherwise the value is evaluated to 10 for decimal
 */
void println(const uint8_t substrate, const uint8_t radix) {
    char* base;
    if (radix == 2) {
        base = (char*) "%b";
    } else if (radix == 16) {
        base = (char*) "%x";
    } else {
        base = (char*) "%d";
    }
    printf(base, substrate);
    printf("\n");
}

int main(void* args) {
    // 6) Bitwise logical operations:
```



```
/* -- Principal logic gates -- */
// (|): Bitwise logical OR, designated by the (+) sign in boolean algebra.
// (&): Bitwise logical AND, designated by the (.) sign in boolean algebra.
// (~): Bitwise logical NOT, designated by the dash or the bar (A`) sign in boolean algebra.

/* -- Combinatorial logic gates -- */
// (^): Bitwise exclusive OR (XOR), produces Q = 1, if only and only one of the inputs is 1, otherwise produces Q = 0
// Note: Q = A ^ B is the same as Q = ~(A) & B) | (A & ~B)
// ~(&): Bitwise logical NAND
// ~(|): Bitwise logical NOR
// ~(^): Bitwise logical NXOR, produces Q = 1, if only and only the two inputs are the same (whether they are zero or
// Note: Q = ~(A ^ B) is the same as Q = ~((~(A) & B) | (A & ~B))
uint8_t inputA = 0b00000011;
uint8_t inputB = 0b11000000;
const uint8_t OR_AB = inputA | inputB; /* 0b11000011, usage: concatenates the binary commands into a single command */
println(OR_AB, 2);
const uint8_t AND_AB = inputA & inputB; /* 0b00000000, usage: compares the binary commands and find if they both are e
println(AND_AB, 2);
const uint8_t XOR_AB = inputA ^ inputB; /* 0b11000011, usage: compares the binary commands and find if they are both e
println(XOR_AB, 2);
const uint8_t NAND_AB = ~AND_AB; /* 0b11111111, usage: */
println(NAND_AB, 2);
const uint8_t NOR_AB = ~OR_AB; /* 0b00111100, usage: finds all the non-involved bits in the OR */
println(NOR_AB, 2);
const uint8_t NXOR_AB = ~XOR_AB; /* 0b00111100, usage: tests whether the 2 inputs are the same (Q = 1) or not (Q = 0).
println(NXOR_AB, 2);

/* and so on */

return 0;
}
```

Bitwise Shifting:

- Shifting operations

Operator	Notation	Name	Usage	Equivalent maths equation	Example
<<	BINARY_NUMBER << NUM_OF_SHIFTS	Left shift	Left shifting bits on (BINARY_NUMBER) by a number of times (NUM_OF_SHIFTS)	BINARY_NUMBER * pow(2, NUM_OF_SHIFTS)	(0b00001110 << 3) is the same as (((0b00001110 << 1) << 1) << 1)
>>	BINARY_NUMBER >> NUM_OF_SHIFTS	Right shift	Right shifting bits on (BINARY_NUMBER) by a	BINARY_NUMBER * pow(2, - NUM_OF_SHIFTS)	(0b00111000 >> 3) is the same as

Operator	Notation	Name	Usage	Equivalent maths equation	Example
			number of times (NUM_OF_SHIFTS)		<code>((0b00111000 >> 1) >> 1) >> 1)</code>

- Addition

Operation	Example 1	Example 2
0 + 0 = 0	0b00000000 + 0b00000000 = 0b00000000	-
0 + 1 = 1	0b00000000 + 0b00000001 = 0b00000001	0b00001100 + 0b00000011 = 0b00001111
1 + 1 = 0 and carry 1	0b00000001 + 0b00000001 = 0b00000010	0b00000100 + 0b00000100 = 0b00000100 * 2 = 0b00000100 * pow(2, 1) = 0b00000100 << 1 = 0b00001000

- Subtraction

General layout	Steps	Example 1	Example 2
Minuend - Subtrahend = Minuend + (-Subtrahend) = Minuend + (1s Complement of Subtrahend + 0b01) = Minuend + (2s Complement of the Subtrahend)	First, find the 2s' complement of the subtrahend, which is the 1s' complement plus one, then add the 2s' complement of the subtrahend to the minuend	0b00001111 (15) - 0b00000011 (3) = 0b00001111 + (0b11111100 + 0b00000001) = 0b00001111 + 0b11111101 = 0b00001100 (12)	0b00000010 (2) - 0b00000001 (1) = 0b00000010 + (0b11111111) = 0b00000001 (1)

Notes:

- The 1s complement of a binary number is taken by applying a NOT gate to the number, ie by flipping bits.
- The 2s complement of a binary number is taken by adding 1 to the 1s complement of that binary number.

Here is an example in C:

```
#include<stdio.h>
#include<inttypes.h> /* includes the [uint8_t] a short-hand for the unsigned char, an 8-bit unsigned integer */

int main(void* args) {

    // 5) Bitwise operators: Shifting bits and logical operators.
    // left shifting: BINARY << SHIFT_TIMES = BINARY * Math.pow(2, SHIFT_TIMES)
    uint8_t x = (1 << 0b00000001) /*0b000000010*/ | (1/*0b0001*/ << 0b000000011 /*3*/) /*0b00001000*/; /*0b01010*/ /* 10 */
    uint8_t y = (0b00000001 << 1) /*0b000000010*/ | (0b000000011 << 1) /*0b000000110*/; /*0b000000110*/ /* 6 */

    // right shifting: BINARY >> SHIFT_TIMES  = BINARY * Math.pow(2, -SHIFT_TIMES)
    uint8_t testRightShifting = (0b00001000 >> 3); /* 0b0001 = 1*/
```

```
uint8_t testLeftShifting = (0b00001000 << 3); /* 0b0001000000 = 64*/

// TIPS to minimize errors: the BINARY value should be your lvalue.

return 0;
}
```

Pointer Operators: WIP

DIY (Do-it-your-own):

- Task-A: build a simple calculator that can add, subtract, divide, multiply and mod only 2 operands.
- Task-B: build a simple scientific calculator that utilizes the logic gates using the logical operators on 2 operands only.
- Task-C: expand your calculator and add bitwise shifting (left and right).

Resources:

- ☒ [The GNU C Language Reference.](#)
- ☒ [The AVR-Sandbox Project.](#)

[Give feedback](#)