

OPERATORS, EXPRESSIONS, INSTRUCTIONS

I. THEORETICAL BACKGROUND

1. Operators and expressions

An expression combines variables and constants in order to produce new values.

1.1. Primary expressions:

- identifier
- constant
- string
- (expression)
- primary expression [expression]
- primary expression (expression list)
- lvalue. identifier
- primary expression => identifier

Obs.: Expression list is described as follows:

- expression
- expression list, expression

1.2. Unary operators: *, &, -, !, ~, ++, --, (type name), sizeof.

Unary expressions (are grouped from left to right):

- * expression
- & lvalue
- expression
- ! expression
- ~ expression
- ++ lvalue
- lvalue
- lvalue ++
- lvalue --
- (type name) expression
- sizeof expression
- sizeof (type name)

Explanations:

a) * => the indirection operator. The expression following it is a pointer and the result is a lvalue.

E.g.: `y = *px` /* y represents the content of the address pointed by px */

b) & => the operator that obtains a pointer. The operand is lvalue and the result is a pointer.

E.g.: `px = &x ;`
`y = *px ;` <=> `y = x`

Obs.: & applies for variables.

Formats such as the following are wrong: `&(x+1)`
 `&3`

c) - => negation operator

d) `!` \Rightarrow logical negation operator . The result is either 0 or 1.

e) `~` \Rightarrow operator for complementing to 1. Converts the bytes:

0 \rightarrow 1
1 \rightarrow 0

The operands are either "short int" or "long int".

f) `++` \Rightarrow operator that increments the operand with 1.

Obs.: `++n` \Rightarrow increments n before using it

`n++` \Rightarrow increments n after its values has been used

g) `--` \Rightarrow the decrementing operator.

h) `(type name)` \Rightarrow operator for type conversion.

It produces the conversion of the expression value to the chosen type. This construction is also named "cast".

i) `sizeof` \Rightarrow returns the dimension in bytes associated to its operand.

Obs.: `sizeof (type name)` \Rightarrow returns the dimension in bytes of an object of the desired type

`sizeof (type)` \Rightarrow is taken as an unit

E.g.: `sizeof (type) - 2 <=> (sizeof (type)) - 2`

1.3. Multiplication operators: `*`, `/`, `%`

They are grouped from the left to the right:

`*` \Rightarrow multiplication

`/` \Rightarrow division

`%` \Rightarrow produces the rest when dividing the first expression to the second one.

E.g.: the expression `(a/b)*b+a%b` is equal to a

1.4. Additive operators: `+` (add), `-` (subtract)

1.5. Bitwise shift operators: `<<`, `>>`

The format: `expression << expression`

`expression >> expression`

`<<` \Rightarrow bitwise left shift

`>>` \Rightarrow bitwise right shift

Obs.: The operands situated to the left of the operator have to be of the integer type. The operator in the right is converted to the "int" type. The free bytes turn to 0.

1.6. Relational operators: `<`, `>`, `<=`, `>=`

They produce the 1 value if the specified relation is true and 0 if is false.

Obs.: `i < x - 1 <=> i < (x - 1)`

1.7. Comparison operators: `==` (equality), `!=` (distinct)

E.g.: `a < b == c > d <=> 1, if a<b and c>d`
0, otherwise

1.8. The byte operator AND: &

```
& | 0 1
-----
0 | 0 0
   |
1 | 0 1
```

1.9. The byte operator EXCLUSIVE OR: ^

```
^ | 0 1
-----
0 | 0 1
   |
1 | 1 0
```

1.10. The byte operator INCLUSIVE OR: |

```
"|" | 0 1
-----
0 | 0 1
   |
1 | 1 1
```

Obs.: It could be used for bits setting.

E.g.: $x = x | \text{mask}$, where mask is placed on the bits of interest.

1.11. The operators LOGIC AND (&&) and LOGIC OR (||)

They are grouped from left to right and the result is 0 or 1.

1.12. The conditional operator: "? :"

The conditional expression: **expression1 ? expression2 : expression3**

If the value of expression1 is:

1 – then the result is expression2

0 - else the result is expression3

Obs.: Pay attention to the syntax.

$\text{lval} = \text{ET1} ? (\text{ET2} ? \text{e1} : \text{e2}) : \text{e3}$ is correct

$\text{lval} = \text{ET1} ? \text{e1} : \text{ET2} ? \text{e2} : \text{e3}$ is wrong

1.13. Assignment operators: =, +=, -=, *=, /=, %=, >>=, <<=, &=, ^=, |=.

They are grouped from right to left.

Assignment expression: **lvalue operator expression**

If the operator is "=", the value of the expression replaces the one of the object determined by the lvalue.

Expressions such as:

$\text{E1 op} = \text{E2}$ are equivalent to $\text{E1} = \text{E1 op E2}$

where "op" is one of the operators: +, -, *, /, %, >>, <<, &, ^ or |

E.g.: $x *= y + 1$ \Leftrightarrow $x = x * (y + 1)$ not $x = x * y + 1$

Obs.: Pentru += and -=, the left operator could also be a pointer.

1.14. The comma operator: ,
The comma expression: **expression1, expression2**

They are grouped from left to right.

2. Instructions

2.1. The conditional instruction (**if...else**)

```
if(expression)  
    instruction1;  
else  
    instruction2;
```

Obs.: "else" is connected to the last encountered "if".

E.g.: if (exp1) instr1;
 else if (exp2) instr2;
 else if (exp3) instr3;
 else instr4;

2.2. The "**while**" instruction

```
while (expression) instruction;
```

The instruction is recurrently executed as long as the expression's value is not 0.

The test takes place at the beginning and therefore if from the first check the expression is 0, the instruction from the while body does not take place at all.

2.3. The "**do...while**" instruction

```
do instruction while (expression);
```

The instruction is recurrently executed until the expression becomes 0.

The test takes place after each instruction execution and therefore the instruction will take place at least once.

2.4. The "**for**" loop:

```
for (exp1; exp2; exp3)  
    instruction;  
where: exp1 => initialises the loop (optional)  
       exp2 => the end test of the loop (optional)  
       exp3 => updating the index variable(optional)
```

The for loop is equivalent to:

```
exp1;  
while (exp2)  
{ instruction;  
  exp3;  
}
```

2.5. The "**break**" instruction:

```
break;
```

This instruction forcibly terminates any "while", "do...while", "for" or "switch" instruction that contains it.

The program then jumps to the next instruction.

This instruction is not used within "if...else" instructions or directly inside a function.

2.6. The "**switch**" instruction (for commutation):

```
switch (exp)
{ case exp1 : sir1
  break;
  case exp2 : sir2
  break;
  default : string
}
```

2.7. The "**continue**" instruction:

Format: **continue;**

Forcedly jumps to the next iteration within the "while", "do...while" or "for" instruction it is placed in.

2.8. The "**return**" instruction:

Format: **return;** or **return (expression);**

2.9. The "**goto**" instruction:

```
goto label;
```

Obs.: The term label must start with a letter.

2.10. The instruction **label**:

Any instruction could be preceded by a label, with the following syntax: identifier:

2.11. The null instruction

Format: ;

E.g.: for (nc=0; getchar ()!= EOF; ++nc);

II. Assignment Workflow

Write a C program that receives from the keyboard two operands (two numbers) and an operator, computes the operations between the two operands using the operand and displays the result.

The program is described as follows:

- two variables (operands) of float type are defined and one int variable is defined for the operator
- the operands are read from the keyboard;
- the operator is also read and if it is:
 - '+' - the numbers are added together;
 - '-' - the numbers are subtracted;
 - '*' - the numbers are multiplied;
 - '/' - if the second operand is not 0, the division is computed, otherwise an error message is displayed
 - any other character - an error message is displayed;
- screen visualisation until pressing another key.

In "C" language the program becomes:

```
# include<stdio.h>
main()
{
    int oper;
    float x,y;
    printf("\nOperand 1: ");
    scanf("%f",&x);
    printf("\nOperand 2: ");
    scanf("%f",&y);
    printf("\nOperator : ");
    if((oper=getche ( ))== '+')
        printf("\nAddition result: %f\n", x+y);
    else if(oper=='-')
        printf("\nSubstraction result: %f\n", x-y);
    else if(oper=='*')
        printf("\nMultiplication result: %f\n", x*y);
    else if (oper=='/')
        if (y==0)
            printf("\nError: Division by 0. \n");
        else
            printf("\nDivision result: %f\n", x/y);
    else if (oper=='% '||oper=='&'||oper==':')
        printf("\nThe operation has not taken place \n");
    else
        printf("\nError\n");
    getch ( );
}
```

1. Write the program from above in the "C" editor.
2. Modify the program so that the reading routine to be performed recurrently until a certain key is pressed.
3. Modify the program so that the selection would be made using a "switch" instruction.

III. SOLUTIONS

2. The program modified with a "while":

```
# include<stdio.h>

main()
{
    int sfarsit;
    sfarsit=1;
    while(sfarsit!='0')
    {
        int oper;
        float x,y;
        printf("\nOperand 1: ");
        scanf("%f",&x);
        printf("\nOperand 2: ");
        scanf("%f",&y);
        printf("\nOperator : ");
        if ((oper=getche())=='+')
            printf("\nAddition result: %f\n", x+y);
        else if (oper=='-')
            printf("\nSubstraction result: %f\n", x-y);
        else if (oper=='*')
            printf("\nMultiplication result: %f\n", x*y);
        else if (oper=='/')
            if (y==0)
                printf ("\nWrong operator 2\n");
            else
                printf ("\nDivision result: %f\n", x/y);
        else if (oper == '%' || oper == '&' || oper == ':')
            printf("\nThe operation was not computed\n");
        else
            printf ("\nError\n");
        printf ("\nHit 0 to leave the program\n");
        finish = getch( );
    }
}
```

3. The program rewritten with the "switch" instruction:

```
# include <stdio.h>

main( )

{ int ready;
  ready = 1;
  while (ready != '0')
  {
      int oper;
      float x, y;
      printf("Operand 1: ");
      scanf("%f", &x);
      printf("Operand 2: ");
      scanf("%f", &y);
      printf("Operator : ");
      oper = getche ();
      switch (oper)
      {
          case '+': printf("\nAddition result: %f\n", x+y);
```

```
break;
case '-': printf("\nSubstraction result: %f\n", x-y);
break;
case '*': printf("\nMultiplication result: %f\n",x*y);
break;
case '/': if (y==0)
    printf ("\n Wrong operator 2\n");
    else
        printf("\nDivision result: %f\n", x/y);
        break;
case '%':
case '&':
case ':': printf ("\nThe operation was not computed\n");
        break;
default: printf ("\nError\n");
}
printf("\nPress 0 to exit the program\n");
ready=getch();

}
}
```