KU LEUVEN

C: an introduction

Expressions

Program: building blocks

- Variables
  - Store data (input, intermediate values, results)
- Expressions
  - Manipulate variables
- Control structures
  - Make decisions (if) or repeat (for, while) statements
- Functions
  - Combine expressions and structures for parameterization and re-use

Operators: overview

- arithmetic operators
- relational operators
- logical operators
- bitwise operators
- assignment operators
- incremental operators
- conditional operator

https://github.com/gjbex/training-material

Arithmetic Operators

- Arithmetic operators are
  - `+` plus
  - `-` minus
  - `*` multiply
  - `/` divide
  - `=` assignment
  - `%` modulus (remainder after division)
- The first 5 are valid for integer and floating-point types.
- The % is valid only for integer types (including **`char`**).

Arithmetic Expressions

```
3.0 / 5.0 - equals 0.6
  3 / 5   - integer division truncates, equals 0
 17 / 6   - equals 2
 18 % 7   - equals 4

2*7 + 5*9 - equals 14 + 45: 59
```

File: arithmetic_1.c

Hands-on: change type `int` into double


```c
#include <stdio.h>

/*
   arthmetic_1.c
*/

int main ()
{
  int var1 = 10;
  int var2 = 2;
  int var3 = 35;
  int var4 = 8;
  int result;

  result = var1 + var2;

  printf ("Sum of var1 and var2 is %d\n", result);
  result = var3 * var3;
  printf ("Square of var3 is %d\n", result);
  result = var2 + var3 * var4;  /* precedence */
  printf ("var2 + var3 * var4 =%d\n", result);
  result = var3 % var1;
  printf ("var3 %% var1 is %d\n", result);

  return 0;

}
```

```
frankvp@CRD-L-08004:.../Expressions$ gcc arithmetic_1.c -o arithmetic_1
frankvp@CRD-L-08004:.../Expressions$ ./arithmetic_1
Sum of var1 and var2 is 12
Square of var3 is 1225
var2 + var3 * var4 =282
var3 % var1 is 5
frankvp@CRD-L-08004:.../Expressions$
```

```c
#include <stdio.h>

/*
   arthmetic_1_double.c
*/

int main ()
{
  double var1 = 10;
  double var2 = 2;
  double var3 = 35;
  double var4 = 8;
  double result;

  result = var1 + var2;

  printf ("Sum of var1 and var2 is %f\n", result);
  result = var3 * var3;
  printf ("Square of var3 is %f\n", result);
  result = var2 + var3 * var4;  /* precedence */
  printf ("var2 + var3 * var4 =%f\n", result);
/*  result = var3 % var1;
  printf ("var3 %% var1 is %d\n", result);
*/
  return 0;

}
```

```
frankvp@CRD-L-08004:.../Expressions$ gcc arithmetic_1_double.c -o arithmetic_1_double
frankvp@CRD-L-08004:.../Expressions$ ./arithmetic_1_double
Sum of var1 and var2 is 12.000000
Square of var3 is 1225.000000
var2 + var3 * var4 =282.000000
frankvp@CRD-L-08004:.../Expressions$
```

KU LEUVEN

Arithmetic Evaluation

- Precedence and order of evaluation.
  eg, `a + b * c`
- Order of evaluation from left to right.
- **`*`**, / and **`%`** take precedence over **`+`** and **`-`**, so that

  `a + b * c` is the same as

  `a + (b * c)`

- Precedence table exists, but use brackets **`()`** instead for safety!!

KU LEUVEN

Incremental Operators

- Valid operators on integer or floating-point numbers.
- Prefix
  - `++x` is a shortcut for `x=x+1`
  - `--x` is a shortcut for `x=x-1`
  - `y=++x` is a short cut for `x=x+1;y=x;`
    x is evaluated **after** it is incremented.
  - `y=--x` is a short cut for `x=x-1;y=x;`
    x is evaluated **after** it is decremented.

Incremental Operators

- Postfix
  - `x++` is a short cut for `x=x+1`
  - `x--` is a short cut for `x=x-1`
  - `y=x++` is a short cut for `y=x;x=x+1`
    x is evaluated **before** it is incremented.
  - `y=x--` is a short cut for `y=x;x=x-1`
    x is evaluated **before** it is decremented.

Incremental Operators

- ++ as postfix-operator n++
  - x is incremented by 1, after using the (old) value in the expression

```
x = 5;
y = ++x;    /* y is 6, x is 6 */

x = 5;
y = x++;    /* y is 5, x is 6 */
```

- x++; are identical ++x; (as single statement)
- only applicable to variables
  - (i+j)++ is not allowed

- *increment.c*

```c
/*
increment.c

Usage of pre-fixing and post-fixing the increment operator
*/

#include <stdio.h>

void main()

{

    int x = 10;

    printf("Value of x after pre-fixing ++ is %d\n", ++x);

    printf("Value of x after post-fixing ++ is %d\n",x++);

    printf("be careful \n");
    printf("What is this - x: %d  - x++ : %d  - ++x: %d \n",x, x++, ++x);


}

```

```
frankvp@CRD-L-08004:.../Expressions$ gcc increment.c -o increment
frankvp@CRD-L-08004:.../Expressions$ ./increment
Value of x after pre-fixing ++ is 11
Value of x after post-fixing ++ is 11
be careful
What is this - x: 14  - x++ : 13  - ++x: 14
frankvp@CRD-L-08004:.../Expressions$
```

Relational Operators

- Relational operators are
  - `>` greater-than
  - `<` less-than
  - `>=` greater-than-or-equal-to
  - `<=` less-than-or-equal-to
  - `==` equal-to
  - `!=` not-equal-to
- These operators are valid for integer and floating-point types.
- Evaluate to 1 if TRUE, and 0 if FALSE

```
3.2 < 7 equals 1, and x != x equals 0
```


Logical Operators

- Logical operators are
  - `&&` AND
  - `||` OR
  - `!` NOT

- **`&&`** and **`||`** connect multiple conditional expressions.
- **`!`** negates a conditional expression (non-zero becomes 0, zero becomes 1).

Relational and Logical Expressions

```
int a=1, b=2, c=3, d=3;
a < b && b < c && c < d   /* FALSE */
a < b && b < c && c <= d  /* TRUE */
(a < b && b < c) || c < d /* TRUE */
a && !b /* FALSE */
```

- **&&** and **||** are evaluated left-to-right and, once the result of TRUE or FALSE is known, evaluation stops – leaving the remaining expressions unevaluated.

logical.c

```c
#include <stdio.h>

/*
logical.c
Demonstrate logical and relational operators
*/

int main(void)
{
    int a=1, b=2, c=3, d=3;
    int e;

    printf("a = %d, b = %d, c = %d, d = %d\n", a, b, c, d);
    printf("(a < b && b < c && c < d) = %d\n", a < b && b < c && c < d);
    printf("(a < b && b < c && c <= d) = %d\n", a < b && b < c && c <= d);
    printf("((a < b && b < c) || c < d) = %d\n", (a < b && b < c) || c < d);


   e = (a == b);
   printf("e = (a == b): %d \n", e);
   e = (c == d);
   printf("e = (c == d): %d \n", e);

   return 0;
}
```

```
frankvp@CRD-L-08004:.../Expressions$ gcc logical.c -o logical
frankvp@CRD-L-08004:.../Expressions$ ./logical
a = 1, b = 2, c = 3, d = 3
(a < b && b < c && c < d) = 0
(a < b && b < c && c <= d) = 1
((a < b && b < c) || c < d) = 1
e = (a == b): 0
e = (c == d): 1
frankvp@CRD-L-08004:.../Expressions$
```

leapyear.c

```c
#include <stdio.h>

/*
leapyear.c
Determine whether year is a leap-year
*/
int main(void)
{
    const int MIN_YEAR = 1828;
    const int MAX_YEAR = 3003;
    int year;

    printf("Enter a year between %d and %d: ", MIN_YEAR, MAX_YEAR);
    scanf("%d", &year);
    if (year < MIN_YEAR || year > MAX_YEAR) {
        printf("Error: invalid year!!\n");
        return -1;
    }

    /* A leap year must be divisible by 4 but not by 100, except
     * that years divisible by 400 are leap years */
    if ((year % 4 == 0 && year % 100 != 0) || year % 400 == 0)
        printf("The year %d is a leapyear.\n", year);
    else
        printf("The year %d is not a leapyear.\n", year);
}
```

```
frankvp@CRD-L-08004:.../Expressions$ gcc leapyear.c -o leapyear
frankvp@CRD-L-08004:.../Expressions$ ./leapyear
Enter a year between 1828 and 3003: 2000
The year 2000 is a leapyear.
frankvp@CRD-L-08004:.../Expressions$ ./leapyear
Enter a year between 1828 and 3003: 2001
The year 2001 is not a leapyear.
frankvp@CRD-L-08004:.../Expressions$ ./leapyear
Enter a year between 1828 and 3003: 2100
The year 2100 is not a leapyear.
frankvp@CRD-L-08004:.../Expressions$
```


Bitwise Operators

- Used to manipulate individual bits inside an integer.
- Bitwise operators are
  - `&` bitwise AND
  - `|` bitwise OR
  - `^` bitwise XOR
  - `<<` left shift
  - `>>` right shift
  - `~` one's complement (bitwise NOT)
- Beware:
  - `&` is not `&&`
  - `|` is not `||`

bitoper_2.c

```c
/*
bitoper_2.c
taken from https://www.programiz.com/c-programming/bitwise-operators

12 = 00001100 (In Binary)
25 = 00011001 (In Binary)

*/

#include <stdio.h>
int main(void)
{
    int a = 12, b = 25;

    printf("Output = %d \n", a&b);
    printf("Output = %d \n", a|b);
    printf("Output = %d \n", a^b);
    printf("\n");

    int num=210, i;
    for (i=0; i<=2; ++i)
        printf("Right shift by %d: %d\n", i, num>>i);

    printf("\n");

    for (i=0; i<=2; ++i)
        printf("Left shift by %d: %d\n", i, num<<i);

    return 0;
}
```

```
frankvp@CRD-L-08004:.../Expressions$ gcc bitoper_2.c -o bitoper_2
frankvp@CRD-L-08004:.../Expressions$ ./bitoper_2
Output = 8
Output = 29
Output = 21

Right shift by 0: 210
Right shift by 1: 105
Right shift by 2: 52

Left shift by 0: 210
Left shift by 1: 420
Left shift by 2: 840
frankvp@CRD-L-08004:.../Expressions$
```

KU LEUVEN

Assignment Operators

- Assignment operators - for example,

  `a += b;` is equivalent to

  `a = a + b;`

  `x *= y+1;` is equivalent to

  `x = x * (y+1);`

- Assignment also with other arithmetic operators: `+,-,*,/,%`

KU LEUVEN

factorial.c

```c
#include <stdio.h>
/*
factorial.c
Calculate the factorial of a non-negative integer.
Result = n*(n-1)*(n-2)* ... *2*1
Note, the factorial of 0 is 1.
*/

int main(void)
{
    int result=1, n;

    /* Get user input */
    printf("Enter a non-negative integer value: ");
    scanf("%d", &n);
    if (n < 0) {
        printf("Error: number must be 0 or greater!!\n");
        return -1;
    }

    /* Calculate factorial */
    while (n)
        result *= n--;

    printf("The factorial is %d\n", result);
}
```

```
frankvp@CRD-L-08004:.../Expressions$ gcc factorial.c -o factorial
frankvp@CRD-L-08004:.../Expressions$ ./factorial
Enter a non-negative integer value: 6
The factorial is 720
frankvp@CRD-L-08004:.../Expressions$
```