PFI lecture Expression and Operator

- The concept of **expression** in programming
  - The most basic building block of programs is an expression.
  - Literals are expressions.
    - 123, “Hello”, -0.01
  - Variables are expressions.
    - n, amount
  - More complex expressions (infinitely many) may be formed by combining expressions with operators.
    - (123+n)*amount, (123+n)*amount+123, ((123+n)*amount+123)/0.01
  - An expression evaluates to a **value** and has a **type** as well!
  - Compilation errors occur if the expression is ill-formed or it is impossible to evaluate due to type mismatch.

Errors in expressions

- Example of ill-formed and type mismatch expressions
  - As literals: “Hello, A’, 1..2
  - As variables: undeclared
  - As type mismatches:
    - 123/ “Hello”
    - int n; string s; n/s
    - int n; string s; s*s
    - double x; double y; x%y
  - Type mismatch in assignment:
    - int n; string s; n=s;

The type of an expression and type conversion

- For literals and variables: the type is its original type.
- For arithmetic expressions with operands having the same type: the type of operand.
- For arithmetic expressions with int and double operands: the type is double.
- For arithmetic expressions with int and char operands: the type is int (for value of char type, we need to understand ASCII).
- The conversions above are implicit, that is, the compiler converts them for us.
- We may explicitly convert a type: use (type) var_name. We need to make sure such a conversion makes sense for what we try to do.
- Numerical types (int, double, char) may be converted to bool type and zero value converts to false, all other values to true.

Why does an expression need a value and type?

The value is needed in assignment statements and in decision making in the **if** and **loop** statements

The type is needed to make sure the use of the expression makes semantic sense. The **type** concept was introduced to reduce subtle logic errors during run time, which are difficult to figure out. We try to capture them at compile time with type.

Logical operators and expressions

- Logical operators combine bool type operands:
  - Logical OR operator: ||
    - x || y is false if both x and y are false, otherwise it is true.
  - Logical AND operator: &&
    - x && y is true if both x and y are true, otherwise it is false.
  - Logical NOT operator: !
    - ! x is true if x is false and is false if x is true.

Truth table for a logical expression

x	y	x&&y
FALSE	FALSE	FALSE
FALSE	TRUE	FALSE
TRUE	FALSE	FALSE
TRUE	TRUE	TRUE

Relational operators and expressions

- Relational expressions produce **bool** value:
  - Identical or same: == (two equal signs)
    - x == y is true if values of x and y are the same, otherwise it is false.
  - Not identical: !=
    - x != y is false if values of x and y are the same, otherwise it is true.
  - Less than: <
    - x < y is true if value of x is less than that of y, otherwise it is false (x and y are the same, x is greater than y).

Relational operators and expressions

- Relational expressions produce **bool** value:
  - Less than or equal to: <= (< followed by =)
    - x <= y is false if value of x is greater than that of y, otherwise it is true.
  - Greater than or equal to: >=
    - x >= y is false if value of x is less than that of y, otherwise it is true.
  - Greater than: >
    - x > y is true if value of x is greater than that of y, otherwise it is false (x and y are the same, y is greater than x).

Precedence rule and association rule of operators

- !
- * / %
- + -
- < <= > >=
- == !=
- &&
- ||
- The operators that appear first have higher precedence, in the same line the same precedence. All associations are from left to right, except for !, which is from right to left

Examples

- What is the value and type of this expression?
  - x*y > x+y == x && x/y || x-y-x
- To answer this question we need to know the type and value of each variable. Let us assume x and y are int with value 2.
- First fully parenthesize using precedence and association rules:
  - ((((x*y) > (x+y)) == x) && (x/y)) || ((x-y)-x)
- Then evaluate it from inside out

Examples

- Step by step evaluation:
  - ((((x*y) > (x+y)) == x) && (x/y)) || ((x-y)-x)
  - ((((4) > (4)) == x) && (x/y)) || ((x-y)-x)
  - (((4 > 4) == x) && (x/y)) || ((x-y)-x)
  - ((false == x) && (x/y)) || ((x-y)-x)
  - ((0 == 2) && (x/y)) || ((x-y)-x)
  - (false && (x/y)) || ((x-y)-x)
  - (false && (x/y)) || ((x-y)-x)
  - (false && (2/2)) || ((x-y)-x)
  - (false && (1)) || ((x-y)-x)
  - (false && true) || ((x-y)-x)
  - false || ((x-y)-x)
  - false || (0-2)
  - false || (-2)
  - false || true (the final value is true and the type is bool!)

Examples

- Boolean expressions that are true if the value of an integer variable is between 0 and 5 inclusive, that is 0,1,2,3,4,5.
  - x==1 || x==2 || x == 3 || x ==4 || x==5
  - (x==1) || (x==2) || (x == 3) || (x ==4) || (x==5)
  - x>=1 && x<=5 (note it is not or, can you see why?)
  - x>0 && x<6 (note x contains an integer)
  - ! (x <1 || x>5)
  - There are more possible answers…
- If the type of x is double, which of the above may still work if we want the value of x to be between 1 and 5?