Introduction to Problem Solving, and Programming

Prof. Deepak Gupta

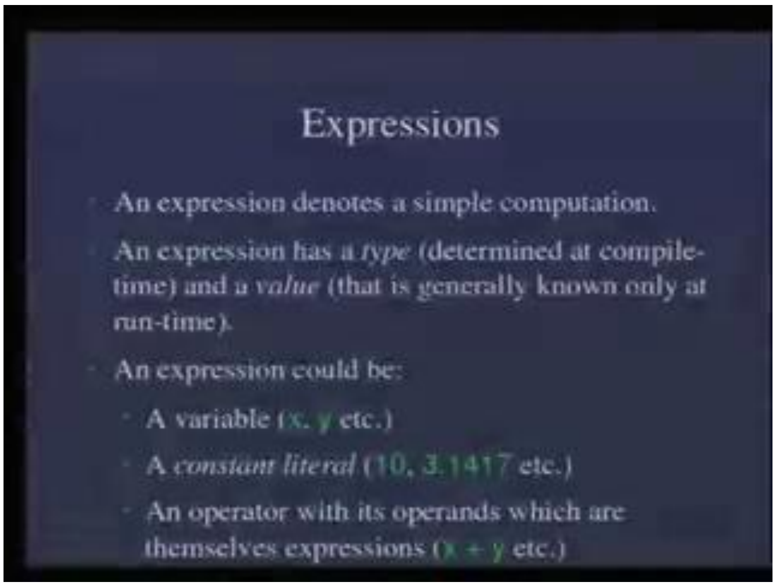
Department of Computer Science Engineering

Indian Institute of Technology, Kanpur

Lecture – 7

In the last lecture, we have talked about the Boolean types, and character types. And in this lecture start talking little more firmly about the C syntax in what expression in C are and how they are evaluated.

(Refer Slide Time: 00:36)



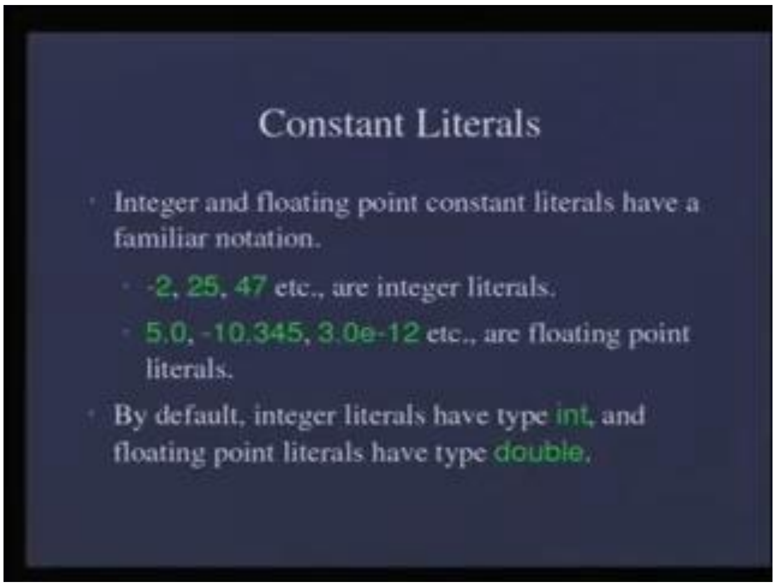
Expressions

- An expression denotes a simple computation.
- An expression has a *type* (determined at compile-time) and a *value* (that is generally known only at run-time).
- An expression could be:
  - A variable (x, y etc.)
  - A *constant literal* (10, 3.1417 etc.)
  - An operator with its operands which are themselves expressions (x + y etc.)

So, an expression is essentially denotation of a simple computation we are all familiar with expression from out high school arithmetic here, you see every expression has a type the types are timed off type. We have already seen integers long integer short char and so on, the type is always determined at compile time or a compiler for any given expression, and the expression to evaluate to evaluate the value that is the value of the expression, and that is the entering on only at run time when the expression gets evaluated.

For example, if you look at the expression x plus y where x has the value 3 y has the value 5, then the value of the expression x plus y will be 8, I think you are very familiar about this, then expression in general could be simply a variable for example, x or y or whatever variable a simple variable also an expression an expression could also be a constant literal such as the number can or the number 3.1417 etcetera etcetera, and the more complicated expression performed out of these kinds of simple expression by using operator. So, we are already familiar with some arithmetic operator like plus minus division multiplication and so on every operator has some operand most of the operators that we have seen. So, far are binary operator which means that there are two operands from this example x plus y is an expression plus is the operator, and x, and y are the operand for the expression. let us talk little bit more about the constant literal of various kinds.

(Refer Slide Time: 02:25)



Constant Literals

- Integer and floating point constant literals have a familiar notation.
  - -2, 25, 47 etc., are integer literals.
  - 5.0, -10.345, 3.0e-12 etc., are floating point literals.
- By default, integer literals have type int, and floating point literals have type double.

The integer and floating point constants literals have familiar notation will not deliver the syntax of these integers, and floating point constants there very familiar to you already there re some examples minus 2 is an integer literals or 25, and 47 etcetera. So, the integer literal essentially a sequence of (( )) optionally followed optionally preceded plus or minus sign, and here are some examples of floating point literals 5.0 that can be sign plus or minus sign 10.345 for example, and we could also use the familiar exponent notation. So, for example, 3.0 e minus 12 is the same as the 3.0 into 10 to the power minus 12 these are all the floating point literals by default integer literals of this kind have the type int, and floating point literals of these kind have the type double, but we can say that the particular integer literal or the floating point literal has its different type as follows.

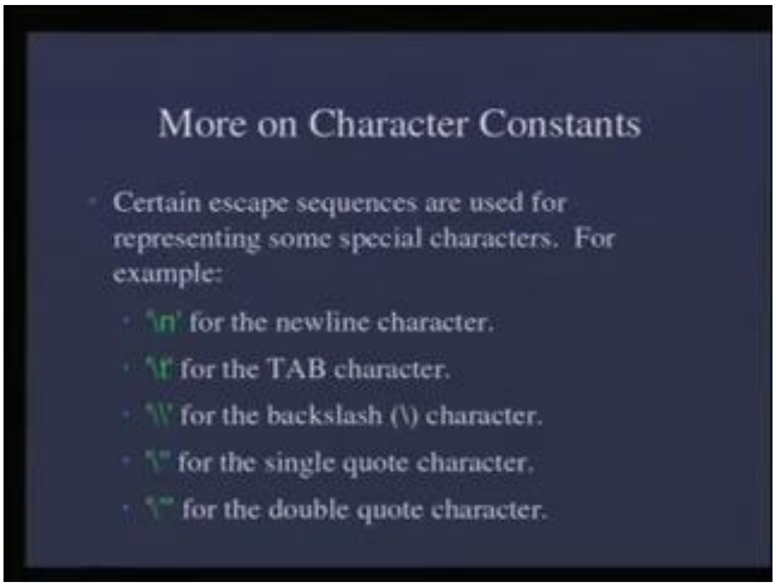
So, for an integer literal constant use the suffix the letter l, and the type is declared to be long rather than the default type int similarly if a suffix go after the literal, then the type becomes unsigned int, and is the suffix used the type becomes unsigned long. So, for example, 5has type int 45 l has type long, and 45l has would have the type unsigned long, similarly for the floating point literal if we use the f or l has the suffix after the constant, then we are forcing it to be of type float or long double respectively remember that before type of floating point constant, and literal is double.

So, for example, just 5.4 has a type double 5.4 f has a type float, and 5.4 has type long double, we will be wondering why the type of these constant literals are important as we will see soon the type of the various components of the expression determines the type of the overall expression, and in some case depending on the type from the value of the expression. And so therefore it is important to talk about the types of these constant literals as well as any other kinds of component, that is expression might have we had talked about the character type or the char type. In the last lecture we can also have constant of type char remember that character are stored by the c cides inside the machine.

So, a character constant is denoted by the character which we use within the single code character. So, for example, the a within the quotes or C capital C within the quote etceteras these are all different constant literal of type character note that the such the constant literal denotes an integer of type char, and the value of the integer is the ascii code of the character we call that the type char is actually just an integer type of a byte size one byte or eight bit.

So, these are also actually integers, but the integer that this particular constant denotes has the value which is same as the ascii code of the corresponding character a or cpital C or 5 etcetera etcetera. So, therefore, it is important to know that the value of the constant literal 5 within quotes is the integer 53 which is the ascii code for the character 5, and not 5 itself. So, here put no quotes arounthe character 5 than it could have the value 5, and of course, all such character literals have the type char now there are apart from the usual character there are certain characters which are not printable or some other reason cannot be easily denoted in this notation we have seen. So, these are denoted certain (( )).

(Refer Slide Time: 06:54)



More on Character Constants

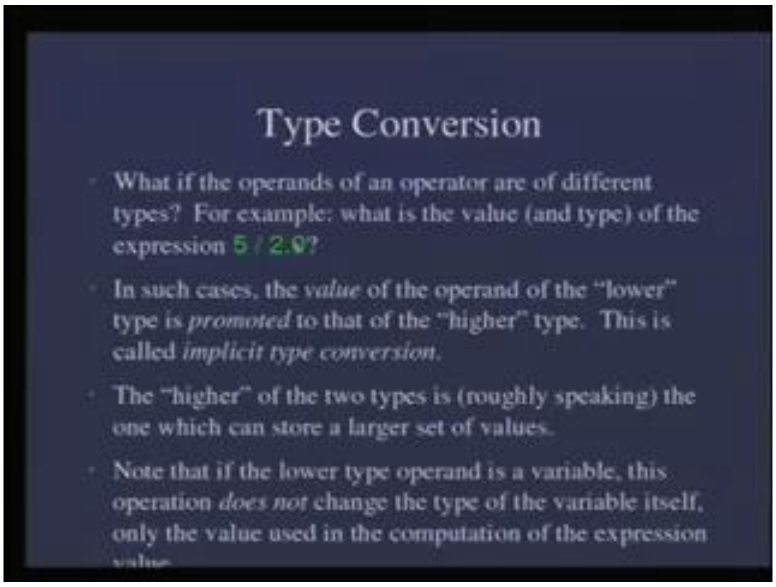
- Certain escape sequences are used for representing some special characters. For example:
  - '\n' for the newline character.
  - '\t' for the TAB character.
  - '\\' for the backslash (\) character.
  - '\'' for the single quote character.
  - '\"' for the double quote character.

For example, within quotes within single quotes letters backslash followed by n denotes the new line character, which is the character corresponding to the enter key on the key board similarly backslash t will denotes the tab character, and so on. If you want to see the backslash character itself you have to use backslash back slash that is two backslashes within the quotes, and the single quote character is denoted by a single quote followed by backslash, and then two single quotes. So, in other words within single quotes backslash quote denotes the single quotes character.

And similarly the double quotes character is denoted by backslash double quote within the quote character let us move on to the operators now, and let begin with arithmetic operators we are already familiar with some of the basic arithmetic operators, like plus minus star which stands for multiplication, and the slash is stands for the division etcetera these are all binary operators that is they have the just they have two operands the operators plus, and minus can also be used in the unary form that is just one operand.

So, for example, plus x is in the succession plus x we are using plus as the unary operator, and similarly the minus x is the expression we are unary minus operator being used of these operators all of them can be used for integers as well as the floating point operand, but the behavior is slightly difference especially in the case of the division operation that is the lash operator in the slash operator the result differ depending upon the type of operand. So, the operands are integer type, then the result of doing the division is the quotient of the saying by dividing the two numbers where as the two number are of floating point type, then the exact value is one that is the result of the expression. So, for example, 5 slash 2 would be 2 for 5, and 2 are integer, and therefore, 5 by two is evaluate the quotient of the division which is 2, and 5 slash 2.0 5 point for example, 5.0 slash 2.0 both result in the value 2.5, because in this case the two operands are floating point operands there is another operator called percent operator which is applicable only to the integer types, and this essentially returns the remainder of the division operation. So, for example, the value of the expression 5 percent 2 would be one, because it divides by 2 the quotient is 2, and the remainder is 1. So, 5 percent 2 is evaluate the remainder which in this case is 1.

(Refer Slide Time: 09:46)



Type Conversion

- What if the operands of an operator are of different types? For example: what is the value (and type) of the expression 5 / 2.0?
- In such cases, the *value* of the operand of the "lower" type is *promoted* to that of the "higher" type. This is called *implicit type conversion.*
- The "higher" of the two types is (roughly speaking) the one which can store a larger set of values.
- Note that if the lower type operand is a variable, this operation *does not* change the type of the variable itself, only the value used in the computation of the expression

So far we had been assuming that the 2 operand involved in a particular expression are of the same type, but in general it is possible that the two operand are of different type. So, for example, what is the value indeed what is the type of the expression 5 slash 2.0 is the result of type floating point with the value 2.5 or is the result an integer with the value two that is the question that we need to answer, and the answer is that in such cases where the two operands for an operator are of different types, then the value of the operand of the low type is promoted to that of the higher type, this is known as implicit type conversion. We will discuss what the higher, and lower type means is a minute essentially the higher of the two type is the roughly speaking the one which can store the largest set of values.

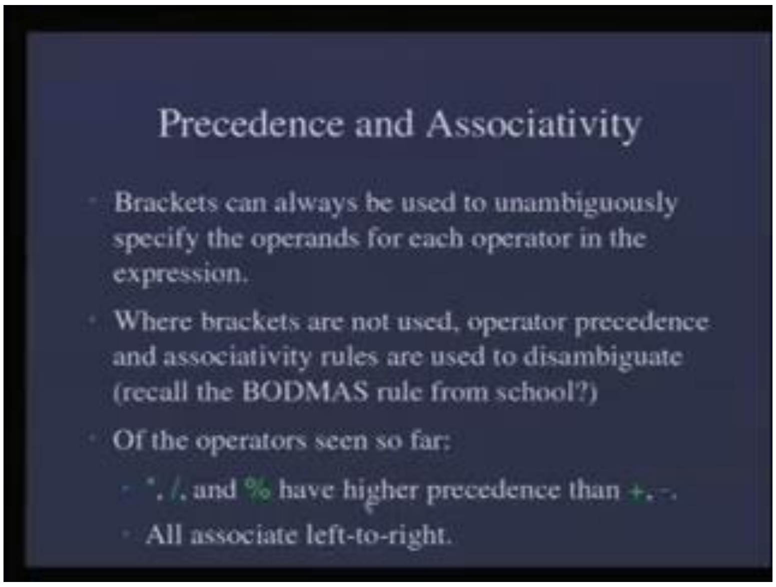
For example, the type float would be higher than the type int. So, in this particular example you see that the 2 operand 5, and 2.0 as we know by now by default the type of the integer literal 5 is int where as that of the floating point literal to 2.0 is double. So, the double is higher type than int. So, therefore, what is going to happen is the value 5 is going to get promoted to a double, and that will result in the value 5.0, and the result of the operation therefore, it also be a double bit the operands are now of the type double. And of course, the floating point division is going to get use since both the operands are of the type double, and the answer would be 2.5 of type double note that in this implicit type conversion if one of the operands happens to be, for example to be variable, then this is not the type of the variable that is itself getting change.

So, if you had double in this case the 5 divided by 2.0 we had 5 divided by x where x is the type of float value is 2.0 for example, in this expression 5 by 2.0. Suppose we had slightly different expression let us say we had an integer variable called x of that with value five, and then we had the expression x divide by 2.0, then the type of x would not change as the result of the implicit type conversion that is x would continue to have the value five, and the type int, but the value of x as used in this expression that is what is going to get change. So, the value will become 5.0, and type will become floating point, but let me get it once again it is not the type of x is changing it is only the value of x as used in the expression whose type is being changed.

So, let us look at this type error at higher end lower type. So, as we have said roughly speaking the higher type is one which can hold a larger range of value. So, you can imagine that the type error c is going to look something like this long double is the highest type followed by double, then float, then unsigned long, then long, then unsigned int, and int more than unsigned long does not really hold the largest sets of the values than long for example, because the only difference is (( )) unsigned long type does not allow for the negative integers, where as the long does allow for the negative integers, but the largest positive value can be stored in the type unsigned long is is larger than the largest integer that can be stored in type long, but still the language defined type unsigned long to be higher than the type long now the type which has smaller than int which are these types types like short unsigned short char, and unsigned char these are always promoted to the type int before being used in an arithmetic operation.

So, for example, if you had 2 short x, and y of with the values 5, and two respectively, then x divide by y in the expression x divide by y both the values both actually get promoted to the type int and. So, the expression would be equivalent to integer 5 divided by integer 2 which would result in a integer value of integer value two. So, the expression 5 slash 2.5 applied the rules 5 has type int 2.5 has type double. So, the 5 get promoted to double and. So, it becomes double 5.0 divides double 2.5, and therefore, the result is the value 2.0 with type double lets now talk about another important notion associated with expressions, and operations that is the notion of the precedence, and associativity.

(Refer Slide Time: 15:00)

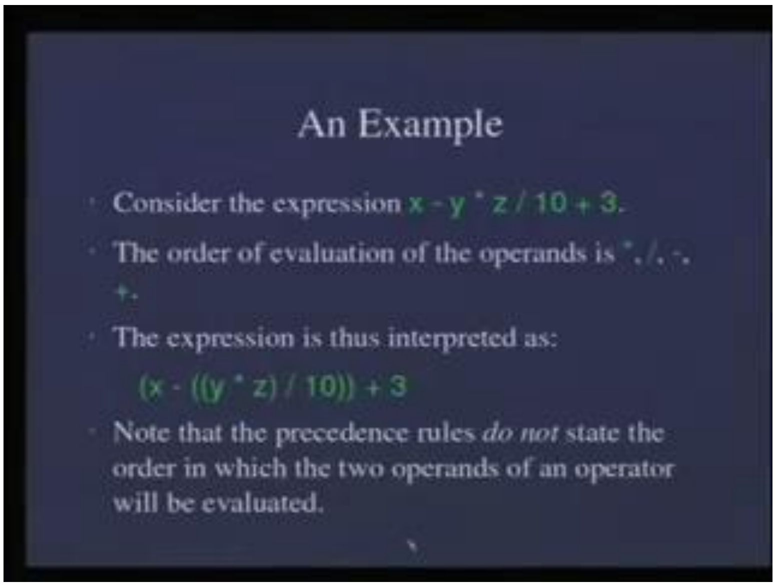


Precedence and Associativity

- Brackets can always be used to unambiguously specify the operands for each operator in the expression.
- Where brackets are not used, operator precedence and associativity rules are used to disambiguate (recall the BODMAS rule from school?)
- Of the operators seen so far:
  - *, /, and % have higher precedence than +, -.
  - All associate left-to-right.

Now, again this is something that should be familiar to from your high school arithmetic as you know in some expression. We need to use brackets to unambiguously specify the operands for each operator in the expression here brackets are not used as you know some convention is used to determine, which operator will be evaluate it first, if you recall you had the BODMAS rules from school which says the precedence brackets are evaluated first followed by division multiplication, and addition, and then subtraction, and within the two operators of the same kind the left to right rule applies that is the operators to be left evaluated first. So, the C languages if I find similar rule for determining how the expression can be evaluated when the operands for the each operator are not here. So far the three operators slash percent, and ampersand have higher precedence than the other two operators namely plus, and minus here we are talking only about the binary operator by unary operator as we see little later all have higher precedence, then all of you are binary operator, and all has to all of the 5 operator associated left to right what is that mean?

(Refer Slide Time: 16:19)



An Example

- Consider the expression x - y * z / 10 + 3.
- The order of evaluation of the operands is *, /, -, +.
- The expression is thus interpreted as:
  (x - ((y * z) / 10)) + 3
- Note that the precedence rules *do not* state the order in which the two operands of an operator will be evaluated.

Let us take an example to clarify this issue. So, let us consider the expression x minus y in to z divide by 10 plus 3. So, the question is what are the operands for these operators which are being used there are four operators is this expression. So, minus star slash, and plus. So, recall the precedence of star, and slash higher than that of minus, and plus which means that star, and plus are going to be evaluated before the minus, and plus, but out of these star, and slash since the associativity for both of them is left to right therefore, the one which access from the left will be evaluated first which means that operator star the one which will get evaluated first followed by slash, and after that minus and, then plus, because again the associativity of minus, and plus is left to right which essentially means that at the same level at the same (( )) level the operator on the left is going to be evaluated first therefore, the evaluation of these operands is star followed by slash followed by minus followed by plus and…

So, therefore, this expression is going to be treated as equivalent to (( )) or that the whole thing divide by ten x minus the result of this, and three is added to the result of the subtraction note that the precedence, and the associativity rules tell us in what order the different operators within the excursion going to be evaluated, but it does not says in what order the two operand of the particular operator are going to be evaluated. For example, the precedence rule do not tell us whether the expression y is going to be evaluated before the expression before the expression z now first of the kinds of expression that we have seen, so far this does not really make it any difference, but when we look at operators with (( )). We will see we adjust some time may be...