C Programming

Lesson 2: Operators and Type Conversion

Lesson Objectives

- **To understand the following topics:**
  - Operators used in C
  - Precedence and Order of Evaluation
  - Type Conversion

Relational Operators

- **Value of the relational expression is of integer type, and is ‘1’ if the result of comparison is true and ‘0’ if it is false**
  - **For example:**

14 > 8	Has the value1, as it is true
34 <= 19	Has the value 0, as it is false
X+y == p+q	Has the value 1(true), only of the sum of ‘x’ and ‘y’ equals the sum of ‘p’ and ‘q’.

Logical Operators

- **Logical Operators are:**
  - Used to combine two or more expressions to form a single expression
  - Evaluated left to right, and evaluation stops as soon as the truth or the falsehood of the result is known

Operator	Name	Meaning
&&	Logical AND	Co njunction
||	Logical OR	Disjunction
!	Logical NOT	Negation

Logical Operators

- **Following table shows operation of the logical && operator:**

Expr1	Expr2	expr && expr2
0	0	0
0	non zero	0
non zero	zero	0
non zero	non zero	1

- **Following table shows operation of the logical || operator:**

Expr1	Expr2	expr1 || expr2
non zero	non zero	1
non zero	zero	1
zero	non zero	1
zero	zero	0

Logical Operators

- **Following table shows operation of the logical ! operator:**

Expr	!expr
nonzero(true)	0
zero (false)	1

Unary Increment/Decrement Operators

- **Unary Operators are:**
  - ++ and --
  - Used as postfix or prefix operators
  - Operators whose value of expression depends on placement of the operator

Assume count = 10 and the expression result=count++;

Now, ‘result’ has value 10, since it is postfix operator.

‘result’ has value 11, if it is result=++count;

Though after both expressions the value of count will be 11.

2.1: Operators used in C

Bitwise Operators

- **Bitwise Operators are**
  - Applied to operands of type char, short, int and long, whether signed or unsigned

&	bitwise AND
|	bitwise inclusive (OR)
^	bitwise exclusive (XOR)

- **For example:**

if x = 45 and y = 10, z = x | y   gives 47

x = 0 0 1 0 1 1 0 1 = 45
y = 0 0 0 0 1 0 1 0 = 10
z = x | y = 0 0 1 0 1 1 1 1 = 47

Ternary/Conditional Operator

- **Ternary Operators:**
  - Provide an alternate way to write the if conditional construct
  - Take three arguments (Ternary operator)
- **Syntax:**

```
expression1 ? expression2 : expression3
```

- **If expression1 is true (i.e. Value is non-zero), then the value returned would be expression2 otherwise the value returned would be expression3**

```
int x, y ;
scanf("%d",&x); /*scanf() accepts the value from the user*/
y = (x>5 ? 3 : 4);
```

This statement will store 3 in 'y' if 'x' is greater than 5, otherwise it will store 4 in 'y'.

Precedence and Associativity of Operators

Operators	**Associativity**
! ++ -- + - (unary)	right to left
* / %	left to right
+ - (binary)	left to right
< <= > >=	left to right
= = !=	left to right
&	left to right
^	left to right
|	left to right
&&	left to right
||	left to right
? :	right to left
= += -= *= /= %= &= ^= |=	right to left

Type Conversion

- **When an operator has operands of different types, they are converted to a common type according to a small number of rules shown below:**

Oper1	Oper2	Result
char	char	char
char	int	int
char	long int	long int
char	float	float
char	double	double

Oper1	Oper2	Result
int	char	int
int	int	Int
int	long int	long int
int	float	float
int	double	double

Oper1	Oper2	Result
float	char	float
float	int	float
float	long int	float
float	float	float
float	double	double

Oper1	Oper2	Result
double	char	double
double	int	double
double	long int	double
double	float	double
double	double	double

Oper1	Oper2	Result
long int	char	long int
long int	int	long int
long int	long int	long int
long int	float	float
long int	double	double

Type Casting

- Explicit type conversions can be forced in any expression, with a unary operator called a cast
- In the construction (type-name) expression the expression is converted to the named type by the conversion rules
- The precise meaning of a cast is as if the expression were assigned to a variable of the specified type, which is then used in place of the whole construction

Type Casting

```
/* Example :Type casting */
# include <stdio.h>
void main(void)
{
  float balance = 6.35 ;
  printf(“Value of balance on typecasting = %d\n”, (int) balance);
  printf(“Value of balance = %f\n”, balance);
}
```

Output :

```
                Value of balance on typecasting = 6
                Value of balance = 6.350000
```

Lab

- **Lab 3**
- **Lab 4**

Operators

- **What does a+++++b mean?**
  - The only meaningful way to parse this is:
    - a ++ + ++ b
  - However, broken down as:
    - a ++ ++ + b
  - This is syntactically invalid: it is equivalent to:
    - ((a++)++) + b

Operators

- **Avoid to make use of compound assignment operator**
  - count++ /* count value evaluated in one cycle */
  - count = count + 1 /* count value evaluated in two cycle */

- **So from performance point of view use count++ instead of count=count+1**

- **Don’t change the value of variable in conditional expression**

Operators

Instead of doing this	**Do this**

void main()
{int marks[]={78,12,63};
int index=0;
if(marks[index++]==2)
printf(“Marks=%d",marks);
else
printf("bad"); /*will go to else part*/}
	void main(){
	int marks[]={78,12,63};
	int index=0,discriminent;
	discriminent=index+1;
	If(marks[discriminent]==2)
	printf(“Marks=%d",marks); /*will go to if part*/
	else
	printf("bad");}

Operators

- **++ and --**
  - When the increment or decrement operator is used on a variable in a statement, that variable should not appear more than once in the statement because order of evaluation is compiler-dependent
  - Do not write code that assumes an order, or that functions as desired on one machine but does not have a clearly defined behavior: int index = 0, marks[5];
  - marks[index] = index++;   /* assign to marks[0]? or marks[1]? */

Operators

- **Never rely on sizeof() to determine the size of an array passed as an argument**

```
finding(char grade[10])
{
			int capacity = sizeof(grade)
			printf(“%d\n”,capacity);
}
Output:4
```

Type Casting

- **Data truncation & Data widening caused due to type conversion & type casting**

- **Truncation**
  - Casting floating-point values to integers may produce useless values
  - The conversion is made simply by truncating the fractional part of the number
  - For example, the floating-point value 3.712 is converted to the integer 3, and the floating-point value -504.2 is converted to -504

Type Casting

- **Here are some more examples:**

```
 void main(void)
{
                    float  rate =3.789;
      printf("%d %d %d",(int)rate,(unsigned int)rate, (char)rate);


}
Outpu:3 3 3
```

Type Casting

- **Widening**
  - Casting an integer to a larger size is fairly straightforward. The value remains the same, but the storage area is widened
  - The compiler preserves the sign of the original value by filling the new leftmost bits with ones if the value is negative, or with zeros if the value is positive

Type Casting

- **When it converts to an unsigned integer, the value is always positive, so the new bits are always filled with zeros**

  char i = 37

  (short) i => 0037

  (int) i => 00000037

```
void main(void)
{
char letter = 37;
printf("%d %d",(short)letter,(int)letter);
}
Output: 37 37
```

Common Best Practices in C Programming

Decision Control Statements-If

- **Test for true or false:**
  - Do not default the test for non-zero, that is:

    if (sample() != FAIL)

    is better than

    if (sample())

    even though FAIL may have the value 0 which C considers to be false. An explicit test will help you out later when somebody decides that a failure return should be -1 instead of 0.

Decision Control Statements-If

- **Do not check a boolean value for equality with 1 (TRUE, YES, etc).**
  - Instead test for inequality with 0 (FALSE, NO, etc). Most functions are guaranteed to return 0 if false, but only non-zero if true. Thus,

- **Bad Example:**

```
if (func() == TRUE) {...
```

- **Good Example:**

```
if (func() != FALSE)
```

Decision Control Statements-If

- **Good practice to write if(0==count) instead of (count==0).**
  - It is a trick to guard against the common error of writing if (count=0). If you are in the habit of writing the constant before the ==, the compiler will complain if you accidentally type if (0 = count).

Decision Control Statements-Else

- **Dangling else:**
  - Stay away from “dangling else” problem unless you know what you are doing:

```
if (level == 1)
if (level == 2)
    printf("***\n");
      else
   printf("###\n");
```

  - The rule is that an else attaches to the nearest if. When in doubt, or if there is a potential for ambiguity, add curly braces to illuminate the block structure of the code

Decision Control Statements-Switch

- **Fall-through in switch:**
  - The break statement causes an immediate exit from the switch. Cases serve just as labels. Hence, after the code for one case is done, execution falls through to the next unless you take explicit action to escape. Break and return are the most common ways to leave a switch.

```
switch (expr) {
    case ABC:
    case DEF:
        statement;
        break;
    case UVW:
```

Decision Control Statements-Switch

```
          statement;            /*FALLTHROUGH*/
      case XYZ:
          statement;
          break;
      }
```

- **While the last break is technically unnecessary, the consistency of its use prevents a fall-through error if another case is later added after the last one**
- **Implies that normally each case must end with a break to prevent falling through to the next**
- **The default case, if used, should always be last and does not require a final break statement if it is last**

Control Loops

- **Null statement:**
  - The null body of a for or while loop should be alone on a line   and commented so that it is clear that the null body is intentional and not missing code.

```
while (*dest++ = *src++);
/* VOID */
```

Control Loops-while

- **Do not let yourself believe you see what is not there.**
  - Look at the following example:

    ```
    while (choice == '\t' || choice = ' ' || choice == '\n')
    choice = getc(file);
    ```

  - The statement in the while clause appears at first glance to be valid C
  - The use of the assignment operator, rather than the comparison operator, results in syntactically incorrect code

Control Loops-while (Contd..)

- The precedence of = is lowest of any operator so it would have to be interpreted this way (parentheses added for clarity):

```
while ((choice == '\t' || choice) = (' ' || choice == '\n'))
choice = getc(file);
```

- The clause on the left side of the assignment operator is:

```
(choice == '\t' || choice)
```

which does not result in an lvalue. If c contains the tab character, the result is “true” and no further evaluation is performed, and “true” cannot stand on the left-hand side of an assignment

Lab

- Lab 2

Summary

- **An operator is a symbol, which represents a particular operation that can be performed on some data**
- **The logical operators && (AND), || (OR) allow two or more expressions to be combined to form a single expression**
- **The sizeof operator returns the number of bytes the operand occupies in memory**
- **Explicit type conversions can be forced in any expression, with a unary operator called a cast**

Review Question

- **Complete the following:**
  1. By default real number is treated as ____________________.
  2. The sizeof operator returns ________________.
  3. The operator used for type casting is ________________.

Review Question: Match the Following

1. !	1. Operand
2. A	1. Relational Operator
3. !=	1. Logical Operator
4. ||	1. Logical Operator