**Operators**

Refer to the program 1\_operators.c.

An operator in a language has

* rank or arity: # of operands required
* precedence : order of evaluation of operators if an expression has more than one operator
* association : order of evaluation when there is cascading of operators at the same level of precedence.

Some operators exist as both unary and binary operators with different levels of precedence and association.

Example:

& : address operator and bitwise operators

\* : multiplicative operator and dereferencing operator

Some unary operators can appear before or after the operand.

Some operators require a lvalue.

We shall discuss these in detail in this document.

We shall classify and discuss various types of operators of ‘C’.

**Arithmetic operators:**

+ - \* / %

There is no exponentiation operator.

All are binary operands – require two operands.

+ and – can also be used as unary operators – in such case, the operator is prefixed – appears to the left of the operand.

If the operands are int for /, the result is int quotient obtained truncation of the result. The operator % is defined on integral types only.

If the operands are mixed types(int and double) in operations + - \* /, the result is of type double.

**Bitwise operators:**

<< >> & | ^ ~

The first 5 are binary operators and the last one is unary operator.

These operators are used on unsigned int only.

Common bitwise operations:

\* multiply variable n by 2

n << 1

\* check whether n is odd

n & 1

\* swap two variables a b

a = a ^ b;

b = a ^ b;

a = a ^ b;

\* check whether ith bit in variable n is 1

n & (1 << I) : 0 implies not set and non 0 implies set

\* set the ith bit in n

n = n | 1 << i;

\* clear the ith bit in n

n = n & ~(1 << I)

**Assignment combined with arithmetic | bitwise operators:**

+= -= \*= /= %= <<= >>= &= |= ^=

a += b is same as a = a + b.

As the same variable occurs on both sides of assignment, the compiler may be able to produce efficient code in the former case.

**Increment and decrement operators:**

These are unary operators ++ and - -. These can be prefixed or postfixed. These are short cuts for assignment. Any assignment operator not only gives a value, but also modifies a variable. A pure expression gives a value without modifying any variable. All assignment operators have side effects – they modify the variable.

The difference between the post and the pre operators manifest only when the value is used.

Both a++ and ++a would increment a by 1. The return value in case of post increment is the old value and pre increment is the new value.

Post operation : use and change.

Pre operation : change and use.

Check the code below from 1\_operators.c.

int e;

// change the variable by 1

// used as stmt; there is no diff

e = 100;

e = e + 1;

printf("%d\n", e);

e += 1;

printf("%d\n", e);

e++; // post increment op

printf("%d\n", e);

++e; // pre increment op

printf("%d\n", e);

// used as expressions

// preincrement

e = 100;

printf("%d\n", ++e); // 101

printf("%d\n", e); // 101

// postincrement

e = 100;

printf("%d\n", e++); // 100

printf("%d\n", e); // 101

Lvalue and Rvalue concept:

Let us look the following code from 2\_operators.c

int a = 10;

int \*p = &a;

printf("pointer to a : %p value \n", p, \*p); // hex number 10

int b = 20;

\*p = 30;

printf("a : %d\n", \*p); // 30

p = &b;

printf("pointer to b : %p value \n", p, \*p); // hex number 20

The variable a is of int type and (int \*p;) p is a pointer to int. We always read a declaration right to left. A pointer variable of some particular type can hold the address of a variable of the same type. & : in unary form – gives the address of the variable.

p = &a; assigns the address of a to p.

We can get back a through p by using \* : in unary form - dereferencing operator.

\*p is same as a.

We can change a by assigning to \*p.

\*p = 30;

We can also change p itself by assigning a pointer to b to p.

p = &b;

We talk about l-value and r-value with respect to assignment operator =.

r-value refers to whatever that could occur to the right of assignment operator.

L-value refers to whatever that could occur to the left of assignment operator.

A constant has only r-value. An initialized variable has both l and r value. An expression of the form a + b is only a r-value. The only operator which gives a value back is the dereferencing operator \*.

**Evaluation of an expression:**

An expression consists of operands and of operators. How is an expression evaluated?

There are two parts in it.

1. Evaluation of operands :

This is fetching the operands from the memory of the computer to the registers of the CPU. All the evaluations happen within the CPU. This order is not defined.

The idea is to allow the compiler to optimize fetching of the operands. The compiler may generate code such a way that if an operand is used more than once, it may get fetched only once.

2. Evaluation of operators:

This follows the rules of precedence and if more than one operator has the same level of precedence, follows the rules of association.

Does the first rule have any effect on our programs? Yes. If affects our programs if the expressions we use are not pure and have side effects.

All assignments have side effects.

Let us consider an example.

a = 10;

b = a \* a++;

The value of a++ is the old value of a and therefore 10. What is the value of the leftmost a? Is the value evaluated before incrementing or after incrementing? It depends on the compiler and the compiler options. So value of leftmost a is undefined. So, value of b is undefined.

// 10 \* 10 => 100

// 11 \* 10 => 110

// undefined; depends on the order of evaluation of operands

// bad code

**Sequence Point:**

How do we write safe and proper code in ‘C’ if operators have side effects and the variable values are not defined in such cases? The language ‘C’ also specifies points in the code beyond which all these effects will definitely be complete. These are called sequence points.

We shall mention sequence points at appropriate places.

**Relational operators:**

The operators are < > <= >= == !=.

The result of relational operation is 1 or 0 – 1 stands for true and 0 for false.

The relational operators should not be cascaded – not logically supported - ‘C’ is not as good as Python in Math!

5 == 5 == 5 is false !!

5 == 5 == 5 becomes 1 == 5 ; this becomes 0 !!

**Logical operators:**

! stands for not

&& stands for and

|| stands for or.

In ‘C’, 0 is false and any non-zero value is true.

As in Python, C follows short circuit evaluation. Evaluation takes place left to right and evaluation stops as soon as the truth or falsehood of the expression is determined.

&& and || are sequence points.

Consider the code from 2\_operators.c.

// a = 10;

// a + a++ undefined

// to support short ckt evaluation, && becomes a sequence point

// expression before &&, || will be completely evaluated.

// a++ == 10 && a == 11 // ok; will be true 1

a + a++ is undefined as the value of left a is not defined.

a++ == 10 && a == 11 will be true.

The value a++ is 10 and a becomes 11 before the evaluation moves across the sequence point &&.

**if expression:**

An expression has a value and a statement does not. If expression is a ternary expression requiring three operands.

E1 ? E2 : E3

where E1, E2 and E3 are expressions.

The expression E1 is first evaluated. If it is true, then the value of the expression is E2 else it is E3.

? of this expression acts like a sequence point.

There are a few more operator not discussed – sizeof (<cast>) ….

We shall stop at this point.