A Fast Review of C Essentials Part II

Structural Programming

by

Z. Cihan TAYSI

Outline

- Operators
  - expressions, precedence, associativity
- Control flow
  - if, nested if, switch
  - Looping

Expressions

- **Constant expressions**
  - 5
  - 5 + 6 * 13 / 3.0
- **Integral expressions (int j,k)**
  - j
  - j / k * 3
  - k –'a'
  - 3 + (int) 5.0
- **Float expressions (double x,y)**
  - x / y *5
  - 3 + (float) 4
- **Pointer expressions (int * p)**
  - p
  - p+1
  - "abc"

Precedence & Associativity

- All operators have two important properties called ***precedence*** and ***associativity***.
  - Both properties affect how operands are attached to operators
- Operators with higher precedence have their operands bound, or grouped, to them before operators of lower precedence, regardless of the order in which they appear.
- In cases where operators have the same precedence, associativity (sometimes called binding) is used to determine the order in which operands grouped with operators.

- 2 + 3 * 4
- 3 * 4 + 2

- a + b – c;
- a = b = c;

Precedence & Associativity

Class of operator	Operators in that class	Associativity	Precedence
primary	() [] -> .	Left-to-Right	HIGHEST
unary	cast operator sizeof & (address of) * (dereference) - + ~ ++ -- !	Right-to-Left	
multiplicative	* / %	Left-to-Right	
additive	+ -	Left-to-Right	
shift	<< >>	Left-to-Right	
relational	< <= > >=	Left-to-Right	
equality	== !=	Left-to-Right	

Precedence & Associativity

Class of operator	Operators in that class	Associativity	Precedence
bitwise AND	&	Left-to-Right	
bitwise exclusive OR	^	Left-to-Right	
bitwise inclusive OR	|	Left-to-Right	
logical AND	&&	Left-to-Right	
logical OR	||	Left-to-Right	
conditional	? :	Right-to-Left	
assignment	= += -= *= /= %= >>= <<= &= ^=	Right-to-Left	
comma	,	Left-to-Right	LOWEST

Parenthesis

- The compiler groups operands and operators that appear within the parentheses first, so you can use parentheses to specify a particular grouping order.
  - (2 – 3) * 4
  - 2 – (3 * 4)
- The inner most parentheses are evaluated first. The expression (3+1) and (8-4) are at the same depth, so they can be evaluated in either order.

  1 + ((3+1) / (8 – 4) – 5)
  1 + (4 / (8 – 4) – 5)
  1 + (4 / 4 – 5)
  1 + (1 – 5)
  1 + -4
  -3

Binary Arithmetic Operators

Operator	Symbol	Form	Operation
multiplication	*	**x * y**	x times y
division	/	**x / y**	x divided by y
remainder	%	**x % y**	remainder of x divided by y
addition	+	**x + y**	x plus y
subtraction	-	**x - y**	x minus y

The Remainder Operator

- Unlike other arithmetic operators, which accept both integer and floating point operands, the remainder operator accepts only integer operands!
- If either operand is negative, the remainder can be negative or positive, depending on the implementation
- The ANSI standard requires the following relationship to exist between the remainder and division operators
  - ***a*** equals ***a*%*b*** + (***a*/*b***) * ***b*** for any integral values of ***a*** and ***b***

Arithmetic Assignment Operators

Operator	Symbol	Form	Operation
assign	=	**a = b**	put the value of ***b*** into ***a***
add-assign	+=	**a += b**	put the value of ***a*+*b*** into ***a***
substract-assign	-=	**a -= b**	put the value of ***a*-*b*** into ***a***
multiply-assign	*=	**a *= b**	put the value of ***a*******b*** into ***a***
divide-assign	/=	**a /= b**	put the value of ***a*/*b*** into ***a***
remainder-assign	%=	**a %= b**	put the value of ***a*%*b*** into ***a***

Arithmetic Assignment Operators

int m = 3, n = 4;
float x = 2.5, y = 1.0;

m += n + x – y
m /= x * n + y
n %= y + m
x += y -= m

m = (m + ((n+x) –y))
m = (m / ((x*n) + y))
n = (n % (y + m))
x = (x + (y = (y – m)))

Increment & Decrement Operators

Operator	Symbol	Form	Operation
postfix increment	++	a++	get value of a, then increment a
postfix decrement	--	a--	get value of a, then decrement a
prefix increment	++	++a	increment a, then get value of a
prefix decrement	--	--b	decrement a, then get value of a

Increment & Decrement Operators

```
main ()  {
        int j=5, k=5;
        printf("j: %d\t k : %d\n", j++, k--);
        printf("j: %d\t k : %d\n", j, k);
        return 0;
}
```

Postfix

```
main ()  {
        int j=5, k=5;
        printf("j: %d\t k : %d\n", ++j, --k);
        printf("j: %d\t k : %d\n", j, k);
        return 0;
}
```

Prefix

Increment & Decrement Operators

int j = 0, m = 1, n = -1;

m++ - --j	(m++) – (--j)	(2)
m += ++j * 2	m = (m + ((++j) * 2)	(3)
m++ * m++	(m++) * (m++)	(implementation-dependent)

Comma Operator

- Allows you to evaluate two or more distinct expressions wherever a single expression allowed!
- Ex : for (j = 0, k = 100; k – j > 0; j++, k--)
- Result is the value of the rightmost operand

Relational Operators

Operator	Symbol	Form	Result
greater than	>	**a > b**	1 if a is greater than b; else 0
less than	<	**a < b**	1 if a is less than b; else 0
greater than or equal to	>=	**a >= b**	1 if a is greater than or equal to b; else 0
less than or equal to	<=	**a < = b**	1 if a is less than or equal to b; else 0
equal to	==	**a == b**	1 if a is equal to b; else 0
not equal to	!=	**a != b**	1 if a is NOT equal to b; else 0

Relational Operators

int j=0, m=1, n=-1;

float x=2.5, y=0.0;

j > m	j > m	(0)
m/n < x	(m / n) < x	(1)
j <= m >= n	((j <= m) >= n)	(1)
++j == m != y * 2	((++j) == m) != (y * 2)	(1)

Logical Operators

Operator	Symbol	Form	Result
logical AND	**&&**	**a && b**	1 if a and b are non zero; else 0
logical OR	**||**	**a || b**	1 if a or b is non zero; else 0
logical negation	**!**	**!a**	1 if a is zero; else 0

Logical Operators

int j=0, m=1, n=-1;

float x=2.5, y=0.0;

j && m	(j) && (m)	(0)
j < m && n < m	(j < m) && (n < m)	(1)
x * 5 && 5 || m / n	((x * 5) && 5) || (m / n)	(1)
!x || !n || m + n	((!x) || !n) || (m + n)	(0)

Bit Manipulation Operators

Operator	Symbol	Form	Result
right shift	>>	**x >> y**	x shifted right by y bits
left shift	<<	**x << y**	x shifted left by y bits
bitwise AND	&	**x & y**	x bitwise ANDed with y
bitwise inclusive OR	|	**x | y**	x bitwise ORed with y
bitwise exclusive OR (XOR)	^	**x ^ y**	x bitwise XORed with y
bitwise complement	~	**~x**	bitwise complement of x

Bit Manipulation Operators cont'd

Expression	Binary model of Left Operand	Binary model of the result	Result value
5 << 1	00000000 00000101	00000000 00001010	10
255 >> 3	00000000 11111111	00000000 00011111	31
8 << 10	00000000 00001000	00100000 00000000	2^{13}
1 << 15	00000000 00000001	10000000 00000000	-2^{15}

Expression	Binary model of Left Operand	Binary model of the result	Result value
-5 >> 2	11111111 11111011	00111111 11111110	$2^{13} - 1$
-5 >> 2	11111111 11111111	11111111 11111110	-2

Bit Manipulation Operators cont'd

Expression	Hexadecimal Value	Binary representation
9430	0x24D6	00100100 11010110
5722	0x165A	00010110 01011010
9430 & 5722	0x0452	00000100 01010010

Expression	Hexadecimal Value	Binary representation
9430	0x24D6	00100100 11010110
5722	0x165A	00010110 01011010
9430 | 5722	0x36DE	00110110 11011110

Bit Manipulation Operators cont'd

Expression	Hexadecimal Value	Binary representation
9430	0x24D6	00100100 11010110
5722	0x165A	00010110 01011010
9430 ^ 5722	0x328C	00110010 10001100

Expression	Hexadecimal Value	Binary representation
9430	0x24D6	00100100 11010110
~9430	0xDB29	11011011 00101001

Bitwise Assignment Operators

Operator	Symbol	Form	Result
right-shift-assign	>>=	**a >>= b**	Assign a>>b to a.
left-shift-assign	<<=	**a <<= b**	Assign a<<b to a.
AND-assign	&=	**a &= b**	Assign a&b to a.
OR-assign	|=	**a |= b**	Assign a|b to a.
XOR-assign	^=	**a ^= b**	Assign a^b to a.

cast & sizeof Operators

- Cast operator enables you to convert a value to a different type
- One of the use cases of cast is to promote an integer to a floating point number of ensure that the result of a division operation is not truncated.
  - 3 / 2
  - (float) 3 / 2
- The ***sizeof*** operator accepts two types of operands: an expression or a data type
  - **the expression may not have type function or void or be a bit field !**
- ***sizeof*** returns the number of bytes that operand occupies in memory
  - sizeof (3+5) returns the size of int
  - sizeof(short)

Conditional Operator (? :)

Operator	Symbol	Form	Operation
conditional	**? :**	**a ? b : c**	if a is nonzero result is b; otherwise result is c

- The conditional operator is the only ternary operator.
- It is really just a shorthand for a common type of ***if...else*** branch

```
z = ( (x<y) ? x : y );
```

```
if (x<y)
        z = x;
else
        z = y;
```

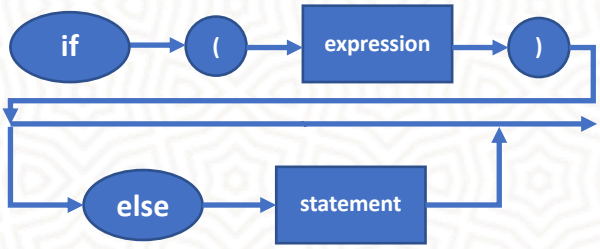
Memory Operators

Operator	Symbol	Form	Operation
address of	**&**	**&x**	Get the address of x.
dereference	*	***a**	Get the value of the object stored at address a.
array elements	**[]**	**x[5]**	Get the value of array element 5.
dot	**.**	**x.y**	Get the value of member y in structure x.
right-arrow	**->**	**p -> y**	Get the value of member y in the structure pointed to by p

Control Flow

- **Conditional branching**
  - if, nested IF
  - switch
- **Looping**
  - for
  - while
  - do...while

The if...else statement



Ex1 :

```
if (x)
   statement1;     // executed only if x is nonzero
statement2;        //always executed
```

Ex2:

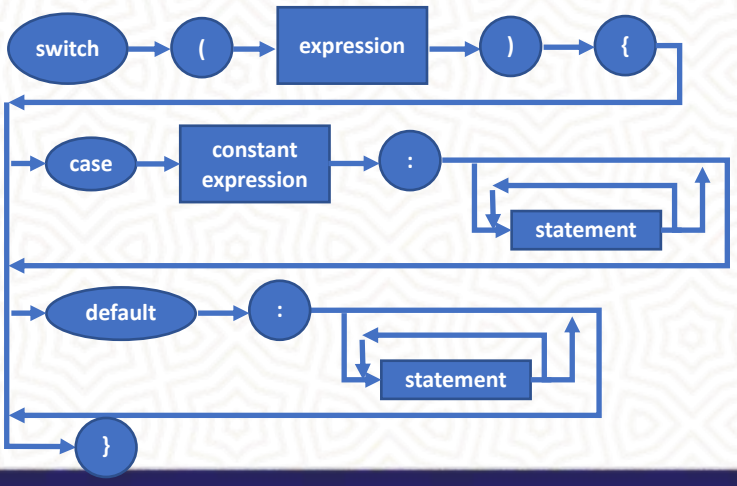
```
if (x)
   statement1;     // executed only if x is nonzero
else
   statement2;     // executed only if x is zero
statement3;        //always executed
```

Nested if statements

- Note that when an ***else*** is immediately followed by an ***if***,
  - they are usually placed on the same line.
  - this is commonly called an ***else if*** statement.
- Nested if statements create the problem of matching each else phrase to the right if statement.
  - This is often called the ***dangling else*** problem !
  - An else is always associated with the nearest previous if.

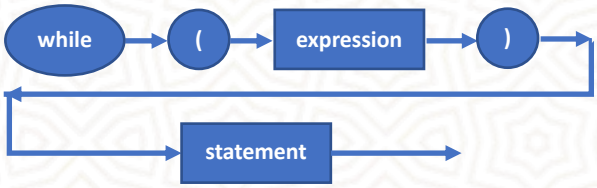
```
if(a<b)
  if(a<c)
        return a;
  else
        return c;
else if (b<c)
        return b;
else
  return c;
```

The switch Statement



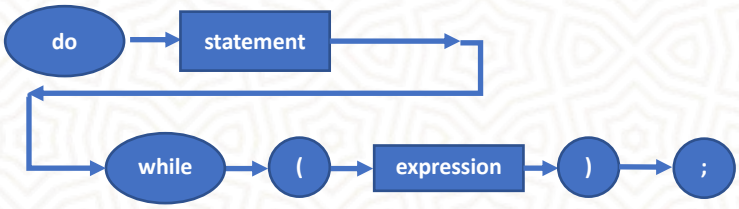
- The ***switch*** expression is evaluated,
  - if it matches one of the case labels, program flow continues with the statement that follows the matching case label.
  - If none of the case labels match the switch expression, program flow continues at the default label, **if exists!**
- No two case labels may have the same value!
- The default label need not be the last label, though it is good style to put it last

The while Statement



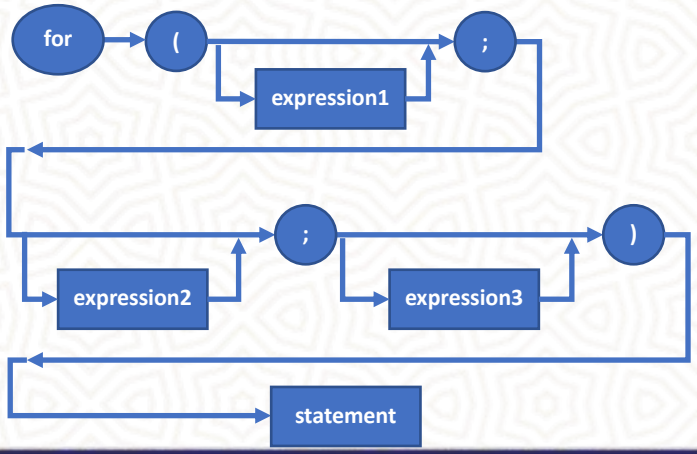
- First the expression is evaluated. If it is a ***nonzero*** value, statement is executed.
- After statement is executed, program control returns to the top of the while statement, and the process is repeated.
- This continues indefinitely until the expression evaluated to zero.

The do...while Statement



- The only difference between a do..while and a regular while loop is that the test condition is at the bottom of the loop.
  - This means that the program always executes statement ***at least one.***

The for Statement



- First, ***expression1*** is evaluated.
- Then ***expression2*** is evaluated.
  - This is the conditional part of the statement.
  - If ***expression2*** is **false**, program control exists the for statement.
  - If ***expression2*** is **true**, the ***statement*** is executed.
- After ***statement*** is executed, ***expression3*** is evaluated.
- Then the statement loops back to test ***expression2*** again.

NULL Statements

- It is possible to omit one of the expressions in a for loop, it is also possible to omit the body of the for loop.

```
for(c = getchar(); isspace(c); c = getchar());
```

- **ATTENTION**
- Placing a semicolon after the test condition causes compiler to execute a null statement whenever the if expression is **true**

```
if ( j == 1);
        j = 0;
```

Nested Loops

- It is possible to nest looping statements to any depth
- However, keep that in mind inner loops must finish before the outer loops can resume iterating
- It is also possible to nest control and loop statements together.

```
for( j = 1; j <= 10; j++)  {
        // outer loop
        printf("%5d|", j);
        for( k=1; k <=10; k++)  {
                printf("%5d", j*k);
        // inner loop
        }
        printf("\n");
}
```

break & continue & goto

- ***break***
  - We have already talked about it in ***switch statement***
  - When used in a loop, it causes program control jump to the statement following the loop
- ***continue***
  - continue statement provides a means for returning to the top of a loop earlier than normal.
  - it is useful, when you want to bypass the reminder of the loop for some reason.
  - Please do NOT use it in any of your C programs.
- ***goto***
  - goto statement is necessary in more rudimentary languages!
  - Please do NOT use it in any of your C programs.