SOME COMPUTATIONAL PROBLEMS

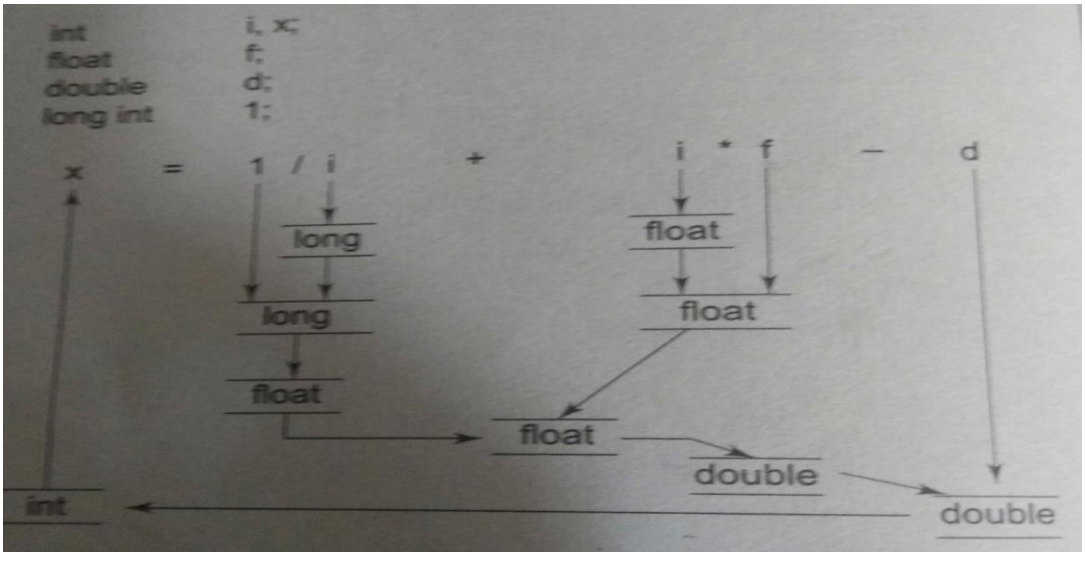
- When expressions include real values, then it is important to take necessary precautions to guard against certain computational errors due to approximations.
- Another problem is division by zero. On most computers, any attempt to divide a number by zero will result in abnormal termination of the program.
- The third problem is to avoid overflow or underflow errors. It is our responsibility to guarantee that the operands are of the correct type and range, and the result may not produce any overflow or underflow.

TYPE CONVERSIONS IN EXPRESSIONS

Implicit Type Conversion

C permits mixing of constants and variables of different types in an expression. C automatically converts any intermediate values to the proper type so that the expression can be evaluated without losing any significance. This automatic conversion is known as *implicit type conversion*.

During evaluation it adheres to very strict rules of type conversion. If the operands are of different types, the 'lower' type is automatically converted to the 'higher' type before the operation proceeds. The result is of the higher type. A typical type conversion process is illustrated in the below Fig.



Given below is the sequence of promotion rules that are applied while evaluating expressions.

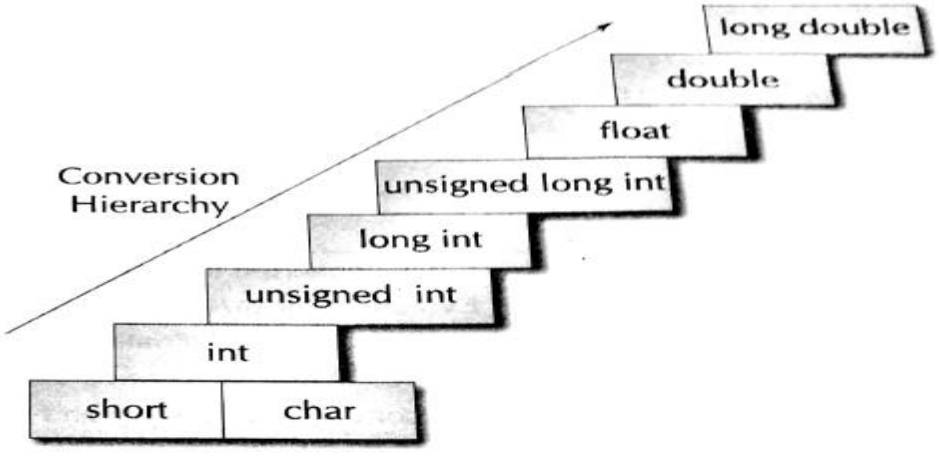
All **short** and **char** are automatically converted to **int**, then

1. If one of the operands is **long double**, the other will be converted to **long double** and the result will be **long double.**
2. Else, if one of the operands is **double**, the other will be converted to **double** and the result will be **double.**
3. Else, if one of the operands is **float**, the other will be converted to **float** and the result will be **float.**

4. Else, if one of the operands is **unsigned long int**, the other will be converted to **unsigned long int** and the result will be **unsigned long int;**
5. Else, if one of the operands is **long int** and the other is **unsigned int**, then
   (a) If **unsigned int** can be converted to **long int**, the **unsigned int** operand will be converted to **long int.**
   (b) Else, both operands will be converted to **unsigned long int** and the result will be **unsigned long int.**
6. Else, if one of the operands is **long int**, the other will be converted to **long int** and the result will be **long int.**
7. Else, if one of the operands is **unsigned int**, the other will be converted to **unsigned int** and the result will be **unsigned int.**

Conversion Hierarchy

Note that, C uses the rule that, in all expressions except assignments, any implicit type conversions are made from a lower size type to a higher size type as shown below:



The final result of an expression is converted to the type of the variable on the left of the assignment sign before assigning the value to it.

However, the following changes are introduced during the final assignment.

1. **float, double** to **int** causes truncation of the fractional part.
2. **double** to **float** causes rounding of digits.
3. **long int** to **int** causes dropping of the excess higher order bits.

```
int main()
{
int a=1;
```

```
float b=4,c;
c=a/b;
printf("%f",c);
return 0;
}
```

output : 0.25

Explicit Conversion

There are instances when we want to force a type conversion in a way that is different from the automatic conversion. Consider, for example, the calculation of ratio of females to males in a town.

ratio = female_number / male_number

Since **female_number** and **male_number** are declared as integers in the program, the decimal part of the result of the division would be lost and **ratio** would represent a wrong figure. This problem can be solved by converting locally one of the variables to the floating point as shown below:

ratio = **(float)** female_number / male_number

The operator **(float)** converts the **female_number** to floating point for the purpose of evaluation of the expression. Then using the rule of automatic conversion, the division is performed in floating point mode, thus retaining the fractional part of the result.

The process of such a local conversion is known as *explicit conversion* or *casting a value*. The general form of a cast is:

(type-name) expression

Where *type-name* is one of the standard C data types. The expression may be a constant, variable or an expression. Example

x= (int) 7.5 ; here 7.5 is converted to integer by truncation, (i.e) 7.

```
#include<stdio.h>
int main()
{
        int a=1,b=4;
        float c;
        c=(float)a/b;
        printf("%f",c);
        return 0;
}
```

output : **0.25**

#include<stdio.h> int main()	#include<stdio.h> int main()

{
int a=1,b=4,c;
c=a/b;
printf("%d",c);
}
 	
output: 0
	{
	int a=1;
	float b=4,c;
	c=a/b;
	printf("%f",c);
	}
	output: 0.25

#include<stdio.h>
int main()
{
int a=1,c;
float b=4;
c=a/b;
printf("%d",c);
return 0;
}
output: 0
	#include<stdio.h>
	int main()
	{
	float a=1,b=4,c;
	c=a/b;
	printf("%f",c);
	return 0;
	}
	output: 0.25

#include<stdio.h>
int main()
{
int a=1,b=4;
float c;
c=(float)a/b;
printf("%f",c);
return 0;
}
output: 0.25
	#include<stdio.h>
	int main()
	{
	int a=1,b=4;
	float c;
	c=a/(float)b;
	printf("%f",c);
	return 0;
	}
	output: 0.25

#include<stdio.h>
int main()
{
int a=1,b=4,c;
c=(float)a/b;
printf("%d",c);
return 0;
	#include<stdio.h>
	int main()
	{
	float a=1,b=4;
	int c;
	c=a/b;
	printf("%d",c);

}	return 0;
	}
output: 0	**output: 0**

EVALUATION OF EXPRESSIONS

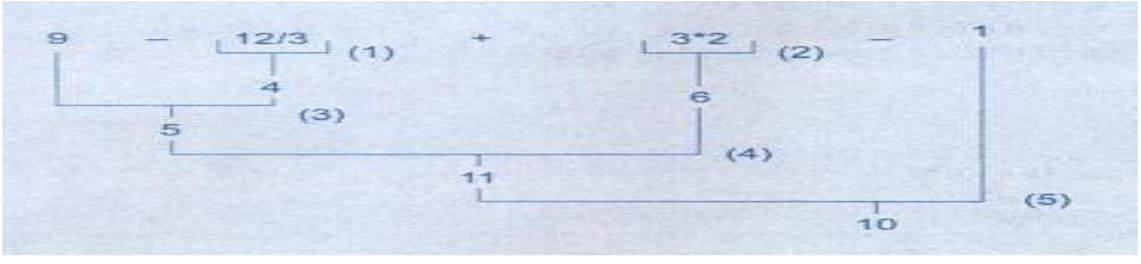
Expressions are evaluated using an assignment statement of the form:

Variable = expression;

Examples of evaluation statements are:

x = a * b – c;

The numbers inside parenthesis refer to step numbers.



Consider the same expression with parenthesis as shown below:

9 – 12 / (3 + 3) * (2 – 1)

Whenever parentheses are used, the expressions within parentheses assume highest priority.

Step 1: 9 – 12 / 6 * (2 – 1)

Step 2: 9 – 12 / 6 * 1

Step 3: 9 – 2 * 1

Step 4: 9 – 2

Step 5: 7

While parentheses allow us to change the order of priority, we may also use them to improve the understand-ability of the program.

- First, parenthesized sub expressions from left to right are evaluated.
- If parentheses are nested, the evaluation begins with the innermost sub-expression.
- The precedence rule is applied in determining the order of application of operators in evaluating sub-expressions.
- The associativity rule is applied when two or more operators of the same precedence level appear in a sub-expression.

Evaluate the following expressions:

(a) If a=3, d=7, e=2, c=5, b=4, x= --a*d/e-c++ * b, find a, b, c, d, e, x.

x= --a*d/e-c++ * b

x=--a*d/e-5*b (c++=5, c=c+1=5+1=6)

x=2*d/e-5*b (--a=2,a=2)

x=2*7/2-5*4

x=14/2-5*4 (2*7=14)

x=7-5*4 (14/2=7)

x=7-20 (5*4=20)

x=-13 (7-20=-13)

a=2

b=4

c=6

d=7

e=2

x=-13

(b) If x=4, y=3, z=2, m=++x + --y + z++ + --z, find m, x, y, z.

m=++x + --y + z++ + --z

m=++x + --y + z++ + 1(--z=1, z=1)

m=++x + --y + 1 + 1 (z++=1, z=z+1=2)

m=++x + 2 + 1 + 1 (--y=2, y=2)

m=5 + 2 + 1 + 1 (++x=5, x=5)

m=7+1+1 (5+2=7)

m=8+1 (7+1=8)

m=9 (8+1=9)

x=5

y=2

z=2

(c) if x=20, y=5, find the value of the expression x= =10+15&&y<10

x= =10+15&&y<10

20==25&&5<10 (10+15=25)

20==25&&1 (5<10 is true, which is 1)

0&&1(20==25 is false, which is 0)

0 (0&&1 is 0)

(d) if a=9,b=12,c=3, find the value of the expression a-b/3+c*2-1

a-b/3+c*2-1

9-12/3+3*2-1

9-4+3*2-1 (12/3=4)

9-4+6-1 (3*2=6)

5+6-1 (9-4=5)

11-1 (5+6=11)

10 (11-1=10)

(e) if a=9,b=12,c=3, find the value of the expression a-b/(3+c)*(2-1)

a-b/(3+c)*(2-1)

9-12/(3+3)*(2-1)

9-12/6*(2-1) [3+3=6]

9-12/6*1 [2-1=1]

9-2*1 [12/6=2]

9-2 [2*1=2]

7 [9-2=7]

(f) if a=9,b=12,c=3, find the value of the expression a-(b/(3+c)*2)-1

a-(b/(3+c)*2)-1

9-(12/(3+3)*2)-1

9-(12/6*2)-1 [3+3=6]

9-(2*2)-1 [12/6=2]

9-4-1 [2*2=4]

5-1 [9-4=5]

4 [5-1=4]

(g) if a=9,b=12,c=3, find the value of the expression a-((b/3)+c*2)-1

a-((b/3)+c*2)-1

9-((12/3)+3*2)-1

9-(4+3*2)-1 [12/3=4]

9-(4+6)-1 [3*2=6]

9-10-1 [4+6=10]

-1-1 [9-10=-1]

-2 [-1-1=-2]

(h) 10!=10 || 5<4 && 8
10!=10 || 5<4 && 8
10!=10 || 0 && 8 (5<4 =0, false)
0||0 && 8 (10!=10 = 0, false)
0||0 (0 && 8 = 0, false)
0 (0 || 0 = 0, false)