Programming in C

# Control Structures

# Control Structures

- Control Structures are also known as control statements.
- They control the flow of program execution.
- Generally program executes from top to bottom and left to right.
- Control structures allows us to change the normal flow of program with or without condition.

# Conditional Operator [ `?:` ]

A conditional expression is of the form

```
expr1 ? expr2 : expr3
```

The expressions can recursively be conditional expressions.

A substitute for `if-else`

Example :

```
(a<b)?((a<c)?a:c):((b<c)?b:c)
```

What does this expression evaluate to?

# Ternary Operator

```
#include<stdio.h>

void main(){

    int a=5;

    int b=6;

    a>b? printf(“Hi”):printf(“Bye”);

}
```

# Flow of Control

- Control structures

  combination of individual statements into a logical unit that regulates the flow of execution in a program or function

  - Sequence
  - Selection (Making Decisions)
  - Repetition (Looping)

# Selection Statement (Branching)

- Allows to jump from one statement to another with or without condition.
- It means execution does not occurs normally.
- Branching is process of jumping.
- if statement, goto statement, and select case statement

# GOTO Branching

- Unconditional branching statement
- Jumps from one line to another.
- It helps to repeat the execution of some statements.
- It also helps to exit the nested loop.
- We need to use line label for goto.
- Line label is the name of line followed by colon.
- Since goto is unconditioned, it makes program less readable and difficult to debug.
- GOTO is avoided

# GOTO Branching

```
#include<stdio.h>
void main(){
    first:
        printf(“first”);
        goto third;
    second:
        prinf(“second);
    third:
        printf(“third”);
}
```

# Branching: *The if Statement*

```
if (expression)
        statement;

if (expression) {
        Block of statements;
}
```

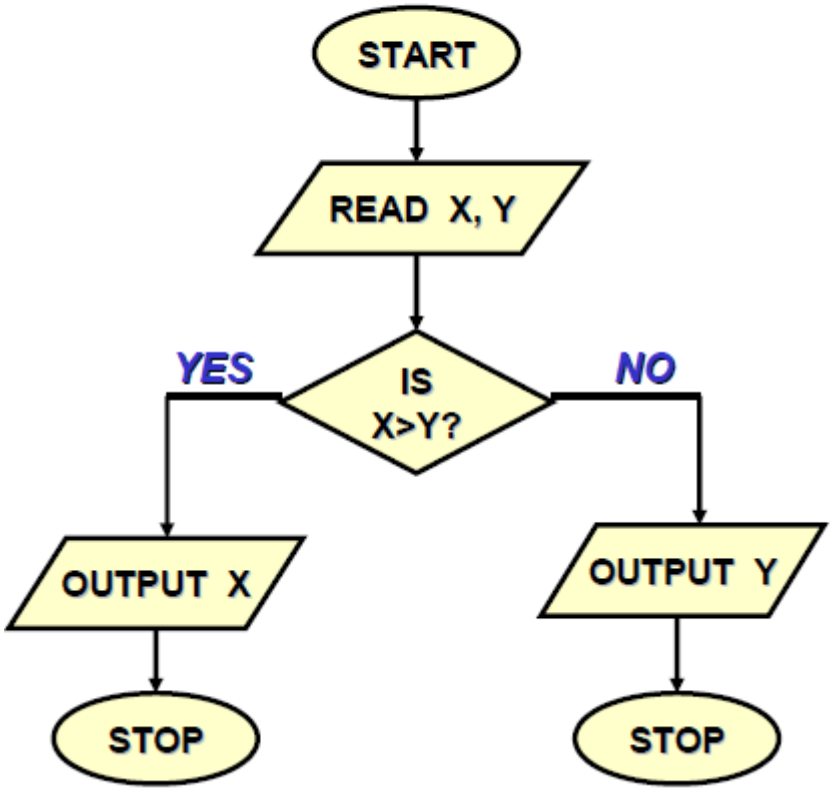
The condition to be tested is any expression enclosed in parentheses. The expression is evaluated, and if its value is non-zero, the statement is executed.

# Branching: *if-else Statement*

```
if (expression) {
    Block of statements;
}
else {
    Block of statements;
}
```

```
if (expression) {
    Block of statements;
}
else if  (expression) {
    Block of statements;
}
else {
    Block of statements;
}
```

# Find the larger of two numbers

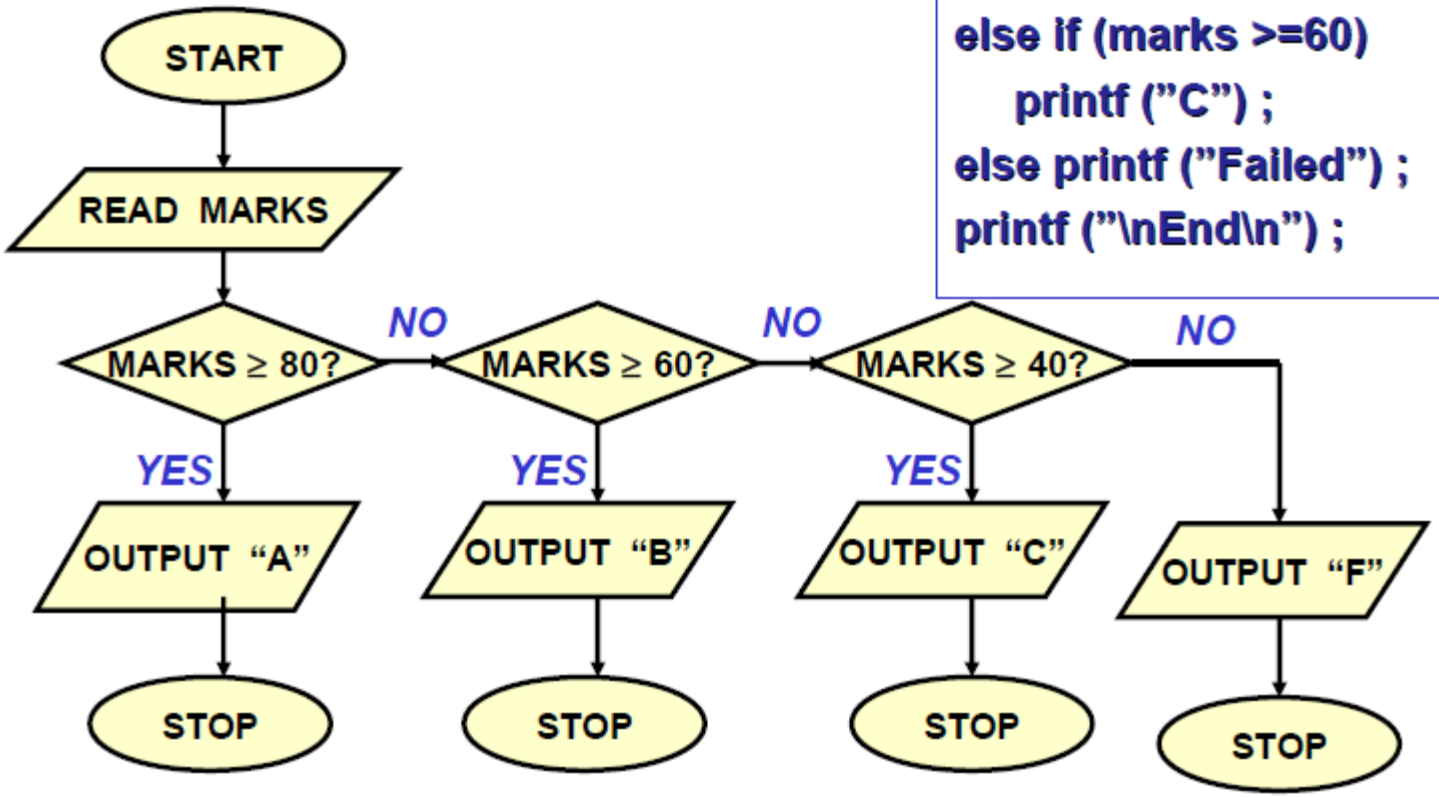



```
int main () {
    int x, y;

    scanf ("%d%d", &x, &y) ;
    if (x>y)
        printf ("%d\n", x);
    else
        printf ("%d\n", x);
}
```

# Grade Computation

```
if (marks >= 80)
    printf ("A") ;
else if (marks >= 60)
    printf ("B") ;
else if (marks >=60)
    printf ("C") ;
else printf ("Failed") ;
printf ("\nEnd\n") ;
```



# If Statement

```
#include<stdio.h>
void main(){
    float cgpa;
    scanf("%f",&cgpa);
    if(cgpa>4.0)
        printf("Wrong CGPA");
    else if (cgpa>=3.6)
        printf("A+");
    else
        printf("A");
}
```

# Confusing Equality (==) and Assignment (=) Operators

- **Dangerous error**
  - Does not ordinarily cause syntax errors.
  - Any expression that produces a value can be used in control structures.
  - Nonzero values are true, zero values are false.

- **Example:**

```
if ( payCode == 4 )
   printf( "You get a bonus!\n" );
```

```
if ( payCode = 4 )
   printf( "You get a bonus!\n" );
```

# Nesting of if-else Structures

- It is possible to nest if-else statements, one within another.
- All “if” statements may not be having the “else” part.
  - Confusion??
- Rule to be remembered:
  - An “else” clause is associated with the closest preceding unmatched “if”.
  - Some examples shown next.

# Dangling else problem

if (exp1) if (exp2) stmta else stmtb

```
if (exp1) {
   if (exp2)
      stmta
   else
      stmtb
}
```

OR

```
if (exp1) {
   if (exp2)
      stmta
}
else
   stmtb
```

?

Which one is the correct interpretation?

# The *switch* Statement

- **This causes a particular group of statements to be chosen from several available groups.**
  - Uses “switch” statement and “case” labels.
  - Syntax of the “switch” statement:

```
switch (expression)  {
   case expression-1: { …….. }
   case expression-2: { …….. }

   case expression-m: { …….. }
   default: { …….. }
}
```

where “expression” evaluates to int or char

# Examples

```
switch ( letter ) {
    case 'A':
        printf ("First letter \n");
        break;
    case 'Z':
        printf ("Last letter \n");
        break;
    default :
        printf ("Middle letter \n");
        break;
}
```

*Will print this statement for all letters other than A or Z*

# Examples

```
switch (choice = getchar())
{
    case 'r' :
    case 'R': printf("Red");
              break;
    case 'b' :
    case 'B' : printf("Blue");
               break;
    case 'g' :
    case 'G':
printf("Green");
               break;
```

*Since there isnt a* **break** *statement here, the control passes to the next statement* (printf) ***without*** *checking the next condition.*

# Another way

```
switch  (choice = toupper(getchar()))  {

    case 'R':       printf ("RED \n");
                    break;
    case 'G':       printf ("GREEN \n");
                    break;
    case 'B':       printf ("BLUE \n");
                    break;
    default:        printf ("Invalid choice \n");

}
```

# The *break* Statement

- Used to exit from a switch or terminate from a loop.
- With respect to “switch”, the “break” statement causes a transfer of control out of the entire “switch” statement, to the first statement following the “switch” statement.
- Can be used with other statements also …

# Loop Statement

- Loop Statement executes set of statements repeatedly till the condition is true.

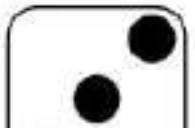
# Why Looping?

# Example 1

```
// Read two integers and print sum
int num1, num2, sum;

scanf("%d %d", &num1, &num2);
sum = num1 + num2;
printf("%d + %d = %d\n", num1, num2, sum);
```

- What if we want to process three different pairs of integers?



# Example 2

- One solution is to copy and paste the necessary lines of code. Consider the following modification:

```
scanf("%d %d", &num1, &num2);
sum = num1 + num2;
printf("%d + %d = %d\n", num1, num2, sum);

scanf("%d %d", &num1, &num2);
sum = num1 + num2;
printf("%d + %d = %d\n", num1, num2, sum);

scanf("%d %d", &num1, &num2);
sum = num1 + num2;
printf("%d + %d = %d\n", num1, num2, sum);
```

- What if you wanted to process four sets? Five? Six? ....

# Processing an arbitrary number of pairs

- We might be willing to copy and paste to process a small number of pairs of integers but
- How about 1,000,000 pairs of integers?
- The solution lies in mechanisms used to control the flow of execution
- In particular, the solution lies in the constructs that allow us to instruct the computer to perform a task repetitively

# Repetition (Looping)

- Use looping when you want to execute a block of code several times
  - Block of code = Body of loop
- C provides three types of loops

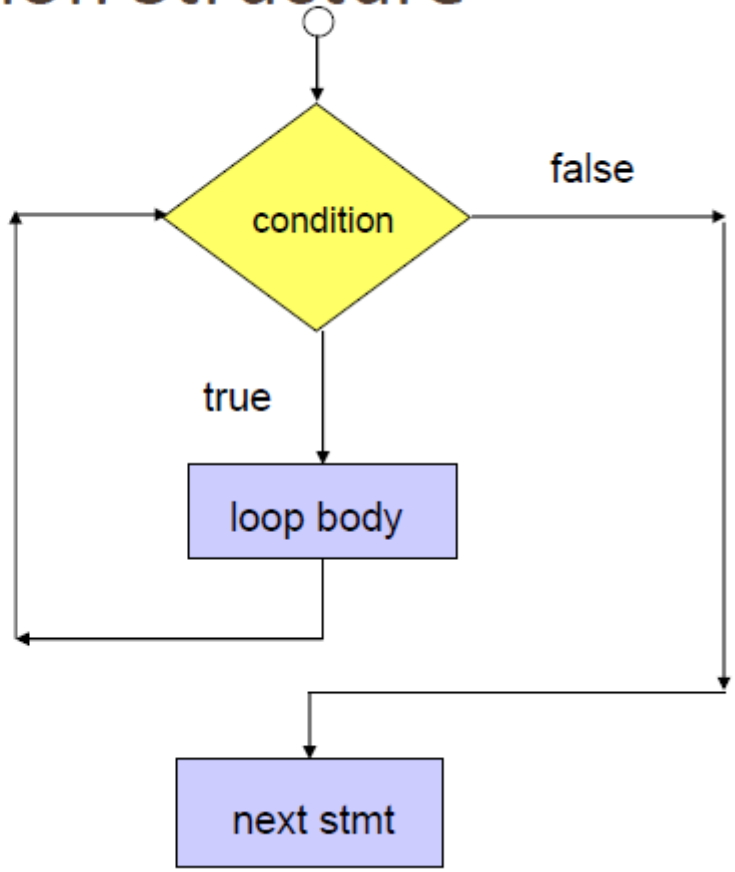

*while* statement
- *Most flexible*
- *No 'restrictions'*


*for* statement
- *Natural* 'counting' loop


*do-while* statement
- *Always executes body at least once*

# The while Repetition Structure

- The condition is tested
- If the condition is true, the loop body is executed and the condition is retested.
- When the condition is false, the loop is exited.



# The while Repetition Structure

- Syntax:

```
while (expression)
      basic block
```

- Expression = Condition to be tested
  - Resolves to true or false
- Basic Block = Loop Body
  - Reminder – Basic Block:
    - Single statement or
    - Multiple statements enclosed in braces

# While LOOP

```
#include<stdio.h>
void main(){
    int count=1;
    int num=10;
    while (count<=10){
        printf("%d\n",count);
        count++;
    }
}
```

# While LOOP: sum of digits

```
#include<stdio.h>
void main(){
    int N,sum=0,R;
    scanf("%d",&N);
    while (N>0){
        R= N % 10;
        sum=sum+R;
        N=N/10;
    }
    printf("%d",sum);
}
```

# While LOOP: reverse of number

```
#include<stdio.h>
void main(){
    int N,rev=0,R;
    scanf("%d",&N);
    while (N>0){
        R= N % 10;
        rev=rev*10+R;
        N=N/10;
    }
    printf("%d",rev);
}
```

# While LOOP: prime or not

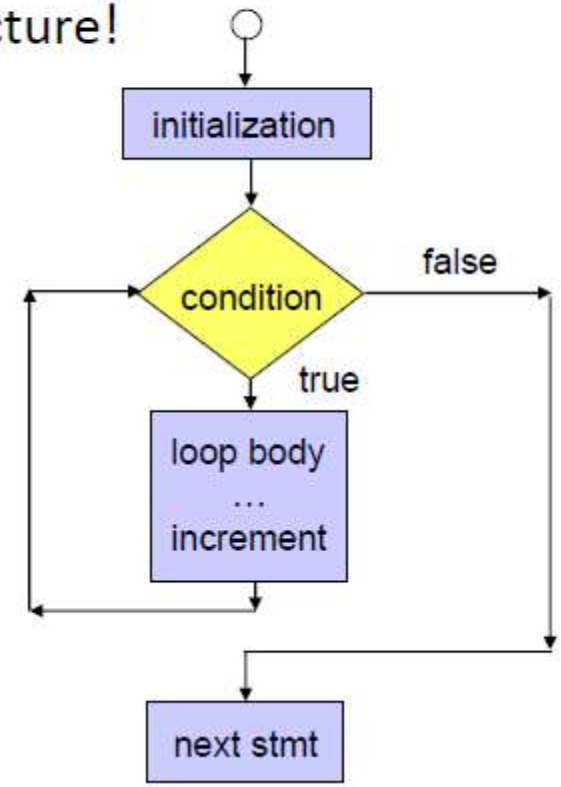
```
#include<stdio.h>
#include<math.h>
#include<stdlib.h>
int main(){
    int N,count,i=2;
    scanf("%d",&N);
    while (i<sqrt(N))
{
        if(N % i==0)
    {
    count++;
    }
        i++;
    }
    if(count==0)
    printf("prime");
else
printf("Composite");
}
```

# While LOOP: $a^b$

```
#include<stdio.h>
void main(){
    int a,b,i=0,res;
    scanf("%d %d",&a,&b);
    res=a;
    while (i<b-1){
        res=res*a;
        i++;
    }
    printf("Result is %d",res);
}
```

# The for Repetition Structure

- A natural 'counting' loop
- Steps are built into for structure!
  1. Initialization
  2. Loop condition test
  3. Increment or decrement



# The *for* Repetition Structure

- Syntax:

```
for (initialization; test; increment)
     basic block
```

# *for* loop example

- Prints the integers from one to ten

```
int counter;
for (counter = 1; counter <= 10; counter++)
{
    printf("%d\n", counter);
}
```

```
int counter;
counter = 1;
while (counter <= 10)
{
    printf("%d\n", counter);
    counter++;
}
```

# for Loop Example

How many times does loop body execute?

```
int count;
for (count = 0; count < 3; count++) {
   printf("Bite %d  --  ", count+1);
   printf("Yum!\n");
}
```

```
Bite 1  --  Yum!
Bite 2  --  Yum!
Bite 3  --  Yum!
```

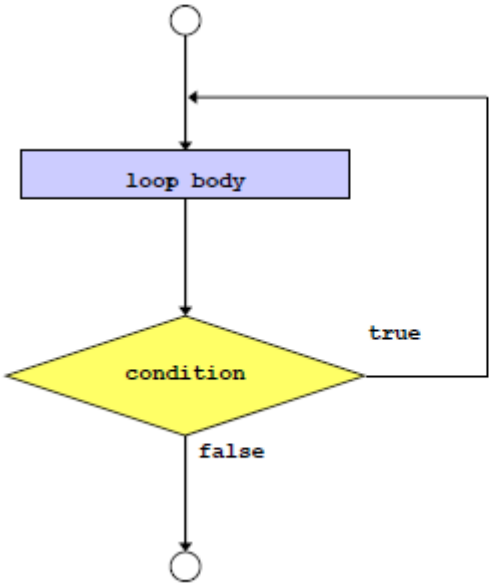
# For LOOP:

```
#include<stdio.h>
void main(){
    int i;
    for(i=0;i<5;i++){
        printf("%d",i);
    }
    printf("%d",i);
}
```

Output 012345

# The do-while Repetition Structure

- The **`do-while`** repetition structure is similar to the **`while`** structure
  - Condition for repetition tested after the body of the loop is executed




All actions are performed at least once!

# The do-while Repetition Structure

- Syntax:

```
do {
    statements
} while ( condition );
```

# The do-while Repetition Structure

- Example

```
int counter = 1;
do {
   printf("%d\n", counter);
   counter ++;
} while (counter <= 10);
```

  - Prints the integers from 1 to 10

## The break Statement

- **`break`**
  - Causes immediate exit from a **`while`**, **`for`**, **`do/while`** or **`switch`** structure
  - **We will use the break statement only to exit the switch structure!**

## The continue Statement

- `continue`
  - Control passes to the next iteration
  - **We will not use the continue statement!**

# Real Arithmetic

- Arithmetic operations involving only real or floating-point operands.
- Since floating-point values are rounded to the number of significant digits permissible, the final value is an approximation of the final result.

  1.0 / 3.0 * 3.0 will have the value 0.99999 and not 1.0
- The modulus operator cannot be used with real operands.

# Mixed-mode Arithmetic

- When one of the operands is integer and the other is real, the expression is called a *mixed-mode* arithmetic expression.

- If either operand is of the real type, then only real arithmetic is performed, and the result is a real number.

  25 / 10 ➔ 2

  25 / 10.0 ➔ 2.5

- Some more issues will be considered later.

# Type Casting

```
int a=10, b=4, c;
float x, y;

c = a / b;
x = a / b;
y = (float) a / b;
```

*The value of c will be 2*

*The value of x will be 2.0*

*The value of y will be 2.5*

# Arithmetic Operator

```
#include<stdio.h>

void main(){

    printf("%f\n",1.0/3.0*3.0);

    printf("%d\n",8/4/2);

    printf("%d\n",8%6/2*5);

}
```

# Relational Operators

- Used to compare two quantities.

| | |
|---|---|
| < | is less than |
| > | is greater than |
| <= | is less than or equal to |
| >= | is greater than or equal to |
| == | is equal to |
| != | is not equal to |

# Examples

10 > 20 is false

25 < 35.5 is true

12 > (7 + 5) is false

- When arithmetic expressions are used on either side of a relational operator, the arithmetic expressions will be evaluated first and then the results compared.

  a + b > c – d is the same as (a+b) > (c+d)

# Relational Operator

```
#include<stdio.h>
void main(){
    int a=5;
    int b=6;
    printf("%d\n",a>b);
    printf("%d\n",a<b);
    printf("%d\n",a>=b);
    printf("%d\n",a<=b);
    printf("%d\n",a==b);
    printf("%d\n",a!=b);
    printf("%d\n",4>5+6-a-- <9);}
```

# Logical Operators

- There are two logical operators in C (also called logical connectives).
  - && ➔ Logical AND
  - || ➔ Logical OR

- What they do?
  - They act upon operands that are themselves logical expressions.
  - The individual logical expressions get combined into more complex conditions that are true or false.

# Logical Operators

- **Logical AND**
  - Result is true if both the operands are true.
- **Logical OR**
  - Result is true if at least one of the operands are true.

| X | Y | X && Y | X \|\| Y |
|---|---|---|---|
| FALSE | FALSE | FALSE | FALSE |
| FALSE | TRUE | FALSE | TRUE |
| TRUE | FALSE | FALSE | TRUE |
| TRUE | TRUE | TRUE | TRUE |

# Logical Operator

```
#include<stdio.h>

void main(){

    int a=5;

    int b=6;

    printf("%d\n",a>b&&a<b);

    printf("%d\n",!a);

    printf("%d\n",!a-a);

    printf("%d\n",!++a-a);

    printf("%d\n",4>5+6||!8+9 ==9);

}
```

| Symbol | Operation | Usage | Precedence | Assoc |
|---|---|---|---|---|
| ! | logical NOT | `!x` | 4 | r-to-l |
| && | logical AND | `x && y` | 14 | l-to-r |
| \|\| | logical OR | `x \|\| y` | 15 | l-to-r |

Treats entire variable (or value)
as TRUE (non-zero) or FALSE (zero).

Result is 1 (TRUE) or 0 (FALSE).

| Symbol | Operation | Usage | Precedence | Assoc |
|---|---|---|---|---|
| ~ | bitwise NOT | `~x` | 4 | r-to-l |
| << | left shift | `x << y` | 8 | l-to-r |
| >> | right shift | `x >> y` | 8 | l-to-r |
| & | bitwise AND | `x & y` | 11 | l-to-r |
| ^ | bitwise XOR | `x ^ y` | 12 | l-to-r |
| \| | bitwise OR | `x \| y` | 13 | l-to-r |

Operate on variables bit-by-bit.

| a | b | a & b | a \| b | a ^ b |
|---|---|---|---|---|
| **0** | 0 | 0 | 0 | 0 |
| **0** | 1 | 0 | 1 | 1 |
| **1** | 0 | 0 | 1 | 1 |
| **1** | 1 | 1 | 1 | 0 |

# Conditional operator (Ternary operator)

It is actually the if condition that we use in C language decision making, but using conditional operator, we turn the if condition statement into a short and simple operator.

**The syntax of a conditional operator is :**

```
expression 1 ? expression 2: expression 3
```

# Special operator

| Operator | Description | Example |
|---|---|---|
| **sizeof** | Returns the size of an variable | **sizeof(x)** return size of the variable **x** |
| **&** | Returns the address of an variable | **&x** ; return address of the variable **x** |
| * | Pointer to a variable | ***x** ; will be pointer to a variable **x** |

# Comma operator

- Comma operator is generally used for separating variables or arguments.
- Comma operator can be used to separate expressions also.
- x=(a++,b++)
- Here, x is assigned with value of b.
- Then a is increased and then be is increased.
- Bracket is important here since comma operator has low precedence than assignment operator.

The Bitwise operators supported by C language are listed in the following table. Assume variable A holds 60 and variable B holds 13, then:

| Operator | Description | Example |
|---|---|---|
| & | Binary AND Operator copies a bit to the result if it exists in both operands. | (A & B) will give 12, which is 0000 1100 |
| \| | Binary OR Operator copies a bit if it exists in either operand. | (A \| B) will give 61, which is 0011 1101 |
| ^ | Binary XOR Operator copies the bit if it is set in one operand but not both. | (A ^ B) will give 49, which is 0011 0001 |
| ~ | Binary Ones Complement Operator is unary and has the effect of 'flipping' bits. | (~A ) will give -60, which is 1100 0011 |
| << | Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand. | A << 2 will give 240, which is 1111 0000 |
| >> | Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand. | A >> 2 will give 15, which is 0000 1111 |

# Assignment Operators

= += -= *= /= %=

| Statement | Equivalent Statement |
| --- | --- |
| a = a+2 ; | a += 2 ; |
| a = a-3 ; | a -= 3 ; |
| a = a*2 ; | a *= 2 ; |
| a = a/4 ; | a /= 4 ; |
| a = a%2 ; | a %= 2 ; |
| b = b+(c+2); | b += c + 2 ; |
| d =d*(e-5); | d *= e - 5 ; |

# More Practice

Given

```
int a =1, b =2, c =3, d =4 ;
```

What is the value of this expression?

```
++b / c + a * d++
```

= 1+4 =5

What are the new values of a, b, c, and d?

a=1, b=3,c=3,d=5

# Practice with Assignment Operators

```
int i = 1, j = 2, k = 3, m = 4 ;
```

| Expression | Value |
| --- | --- |
| `i += j + k` | **i=6** |
| `j *= k = m + 5` | **k=9, j=18** |
| `k -= m /= j * 2` | **m=1, k=2** |

# Practice with Relational Expressions

```
int a = 1, b = 2, c = 3 ;
```

| Expression | Value | Expression | Value |
|---|---|---|---|
| `a < c` | T | `(a + b)>= c` | T |
| `b <= c` | T | `(a + b)== c` | T |
| `c <= a` | F | `a != b` | T |
| `a > b` | F | `(a + b)!= c` | F |
| `b >= c` | F | | |

# Practice with Arithmetic Expressions

```
int       a = 1, b = 2, c = 3 ;
float   x = 3.33, y = 6.66 ;
```

| Expression | Numeric Value | True/False |
|---|---|---|
| a + b | 3 | T |
| b-2*a | 0 | F |
| c- b-a | 0 | F |
| c-a | 2 | T |
| y-x | 3.33 | T |
| y-2*x | 0.0 | F |

# Type conversion

- Automatic type conversion may occur when two operands to a binary operator are of a different type
- Generally, conversion "widens" a variable (e.g. `short` → `int`)
- However "narrowing" is possible and may not generate a compiler warning; for example:

```
1 int i = 1234;
2 char c;
3 c = i+1; /* i overflows c */
```

- Type conversion can be forced by using a cast, which is written as: (type) exp; for example: `c = (char) 1234L;`

# Type Casting

`1 / 2.0` gives a result of `0.5`

Given the following:

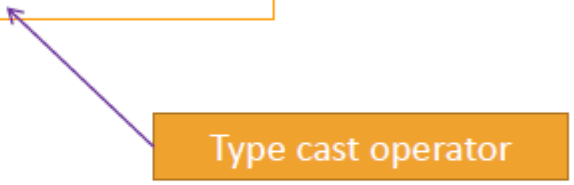
```
int m = 1;
int n = 2;
int result = m / n;
```

result is 0, because of integer division

# Type Casting

- To get floating point-division, you must do a type cast from int to double (or another floating-point type), such as the following:

```
int m = 1;
int n = 2;
double doubleAnswer = (double) m / n;
```




  - This is different from `(double) (m/n)`

# Type Casting

- Two types of casting
  - Implicit – also called 'Automatic'
    - Done for you, automatically

      `17 / 5.5`

      This expression causes an 'implicit type cast' to take place, casting the 17 → 17.0
  - Explicit type conversion
    - Programmer specifies conversion with cast operator

```
(double)17 / 5.5
(double) myInt / myDouble
```

# Type Casting

```
int a=10, b=4, c;
float x, y;

c = a / b;
x = a / b;
y = (float) a / b;
```

*The value of c will be 2*
*The value of x will be 2.0*
*The value of y will be 2.5*