Topic 2. C Basics (1). Theory

Data Types, Operators, Decision and Repetition Statements (1)

COMP ENG 2SH4

Principles of Programming

McMaster University, 2015

Instructor: Sorina Dumitrescu

Textbook Reading

- Chapters 2, 3, 4 covers the material for Topics 2 and 3 (C Data Types, Operators, Decision and Repetition Statements)

Variables

- May change their value during program execution.
- Any variable has a name – any valid identifier:
  - a sequence of letters, digits or underscores that does not begin with a digit;
  - maximum length is 31
  - case sensitive;
  - not a keyword
- Any variable must be declared with a type.
- The type indicates:
  - the amount of memory needed to store the variable and how the values are represented
  - the set of possible values
  - what operations can be performed with the variable

Data Types

- Basic data types
  - **char** (for characters) – size: **1 byte** (8bits)
  - **int** (integers) – either 16 or 32 bits – implementation defined.
  - **short int (short), long int (long)**
  - **float** (single-precision floating points)
  - **double** (double-precision floating points)
  - **long double** (extended-precision floating point)
  - **signed/unsigned** -- apply to **char, int, short, long**
  - **unsigned** types - only positive values
  - **signed** types – positive and negative values

Arithmetic Operators

- Unary operators:
  - +, -
- Binary operators:
  - +, -, *, /, %
  - If both operands of “/” are integers, integer division is performed: 5/2 is 2, not 2.5
  - % is the **remainder** or **modulus** operator
    - it applies **only to integer** types.
    - a%b is the remainder of the division of a by b
    - Ex: value of 18%4 is 2
  - result of % with negative operands is implementation-dependent.

Evaluating Expressions

- *6 + 7 /9 * 3*
- *In what order to evaluate the operations?*
- ***Precedence*** and ***Associativity*** rules specify the order of evaluation of operations.
- Parentheses are evaluated first.
- Unary operators have higher precedence than binary operators.
- +,- have lower precedence than *,/, %
- +, - *, / % **associate from left to right**

Precedence and Associativity Table

Operators						Associativity	Type
()						left to right	parentheses
++ preincrement	-- predecrement	+	-	!	(type)	right to left	unary
*	/	%				left to right	multiplicative
+	-					left to right	additive
<	<=	>	>=			left to right	relational
==	!=					left to right	equality
&&						left to right	logical AND
||						left to right	logical OR
=	+=	-=	*=	/=	%=	right to left	assignment
,						left to right	comma operator

Assignment Operator “=“

- Assignment expression*:* **var** = **expr$_1$**
- Reads: **var gets** the value of **expr1**
- Evaluate **expr$_1$** and assign its value to **var**.

  y = 7 * 3 - 8

Compound Assignment Operators

- +=, -=, *=, /= , %=
- x += 3 ←→ x= x+3
- x -= 2.5 ←→ x= x-2.5
- y /= 5.0 ←→ y = y/5.0
- a *= 7 ←→ a = a*7
- b%=2 ←→ b=b%2

Increment and Decrement Operators

- Unary operator
  - pre-increment: *++***var** (increments **var** by 1)
  - post-increment: **var***++*
  - pre-decrement: --**var** (decrements **var** by 1)
  - post-decrement: **var**--
  - *Can be applied to a variable, but not to a constant or an expression which is not an l-value*
  - *"l-value" means that it can be on the left side of an assignment*

☑ x++;

☒ ++5; (error C2105: '++' needs l-value)

☒ (x+3)++;

Increment and Decrement Operators

- The pre-increment or pre-decrement is applied to the variable before it is used in the expression.
- The post-increment or post-decrement is applied to the variable after it is used in the expression.
- Example:

int i=1, a=0, b=0;

a=++i; /*first ++ is executed, then the assignment */

i=1;

b=i++; /* first the assignment is executed, then ++ */

Thus, the value of a is 2, the value of b is 1.

Relational and Equality Operators

- Relational operators: >, <, >=, <=.
- Equality operators: == (equal), != (nonequal).
  - a >= 0 (a relational expression)
  - n%2 == 0 (an equality expression)

Logical Operators

- Logical negation operator: !
- !**var** or !(**expr**)
- !(7<=6)

expr	!(expr)
T	F
F	T

Logical Operators &&, ||

- && (logical **AND**)
- expr1 && expr2
- TRUE if and only if both expressions are true
- || (logical **OR**)
- TRUE if and only if at least one is true
- Condition that **int** a is between 2 and 6 inclusive
  - 2<=a<=6 is not syntactically correct
  - a>=2 && a<=6 is correct
- When is the following true?
- 2<=a || a<=6

Truth Tables for &&, ||

X	Y	X&&Y	X||Y
T	T	T	T
T	F	F	T
F	T	F	T
F	F	F	F

Precedence and Associativity Table

Operators						Associativity	Type
()						left to right	parentheses
++ preincrement	**--** predecrement	**+**	**-**	**!**	**(type)**	right to left	unary
*****	**/**	**%**				left to right	multiplicative
+	**-**					left to right	additive
<	**<=**	**>**	**>=**			left to right	relational
==	**!=**					left to right	equality
&&						left to right	logical AND
||						left to right	logical OR
=	**+=**	**-=**	***=**	**/=**	**%=**	right to left	assignment

Simple Program

```
// example of program containing only
//function main
#include <stdio.h>

int main(void)
{
  optional-declaration-list
  optional-statement-list
   return 0;  /* indicate that program
                ended successfully */
}
```

Variables Declarations

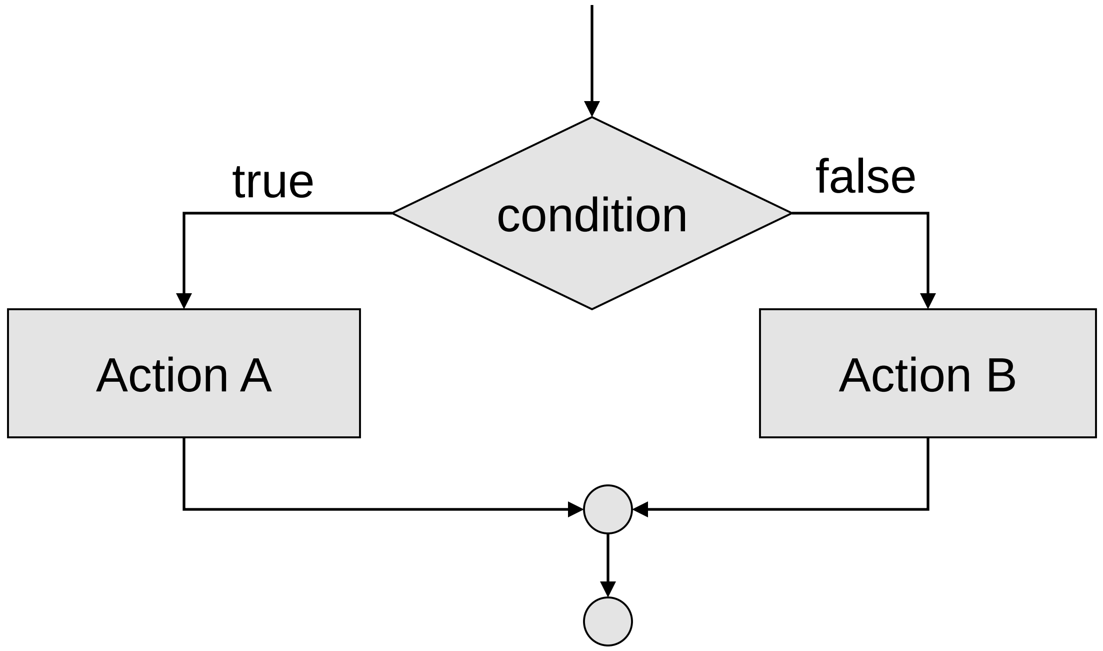
- ✓ **unsigned int** day, month, year;
- ✗ **int** day, month, year,
- ✓ **float** operand1, operand2;
- ✗ **float** sum_1; sum_2;
- ✗ **float** result%, result~;
- ✓ **int** i=0, m, j=1; // i and j are also assigned initial values

Simple Statement

- **A simple statement: an expression followed by “;”**
  - ✓ a++;
  - ✓ a = b+c;
  - ✓ b+c;
  - ✓ ;   /* does nothing */
  - ✓ printf("sum = %d\n", sum);
  - ✗ a = a + 1
  - ✗ a = 1,

if…else Selection Statement

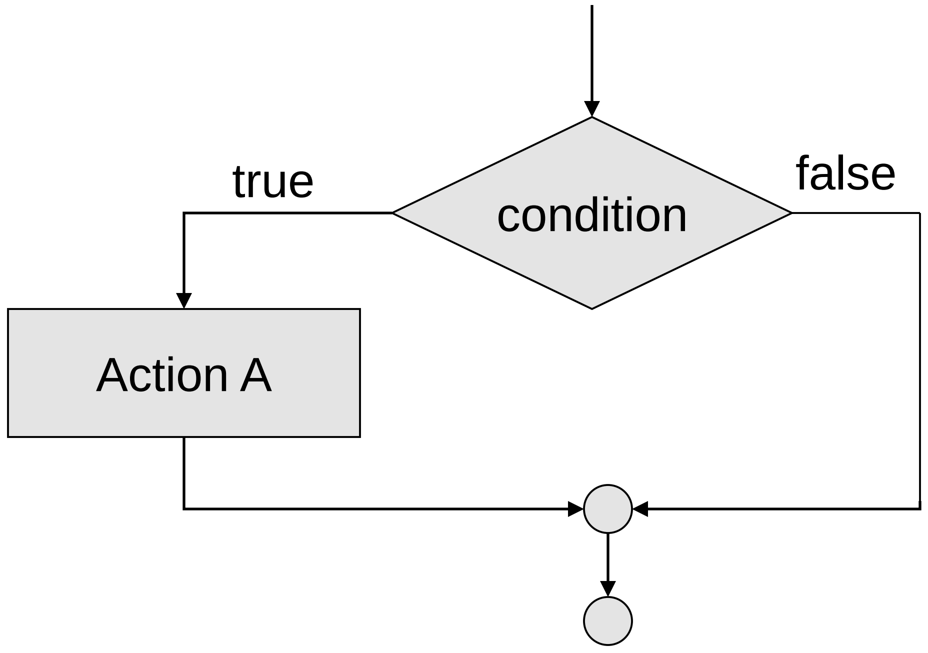
```
if( condition )
{
      statements1
     /* action A */
}
else
{
      statements2
      /* action B */
}
```



if Selection Statement

```
if( condition )
{
     statements1
    /* action A */
}
```

If **statements1** consists of only one statement, the enclosing braces may be omitted.



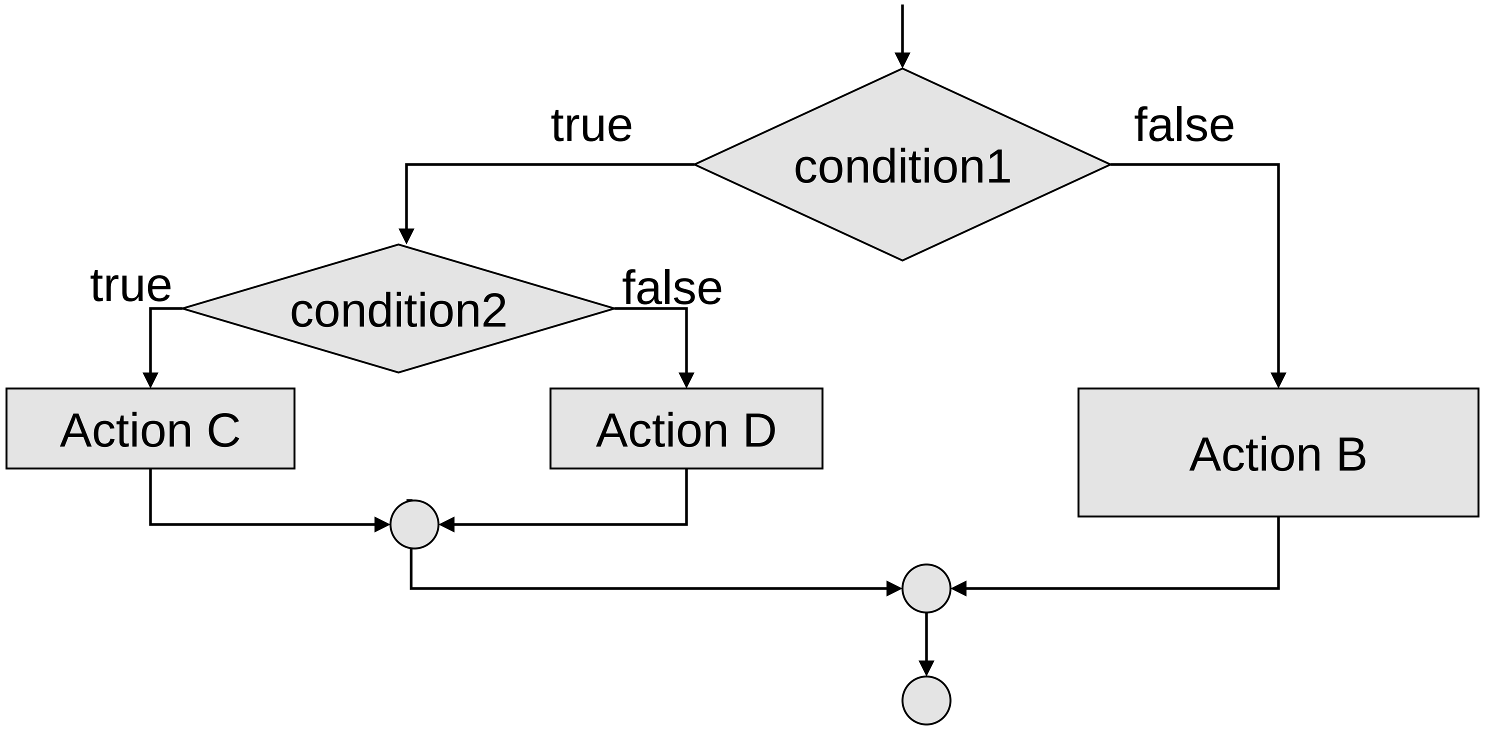
Example

Assume that x,y and z are of type **int**.
What does each of the following pieces of code do?

```
/* variant 1 */
if( x<y )
{
     z=x;
     printf("%d",z);
}
else
{
      z=y;
     printf("%d",z);
}
```

```
/* variant  2*/
z=y;
if( x<y )
     z=x;
printf("%d",z);
```

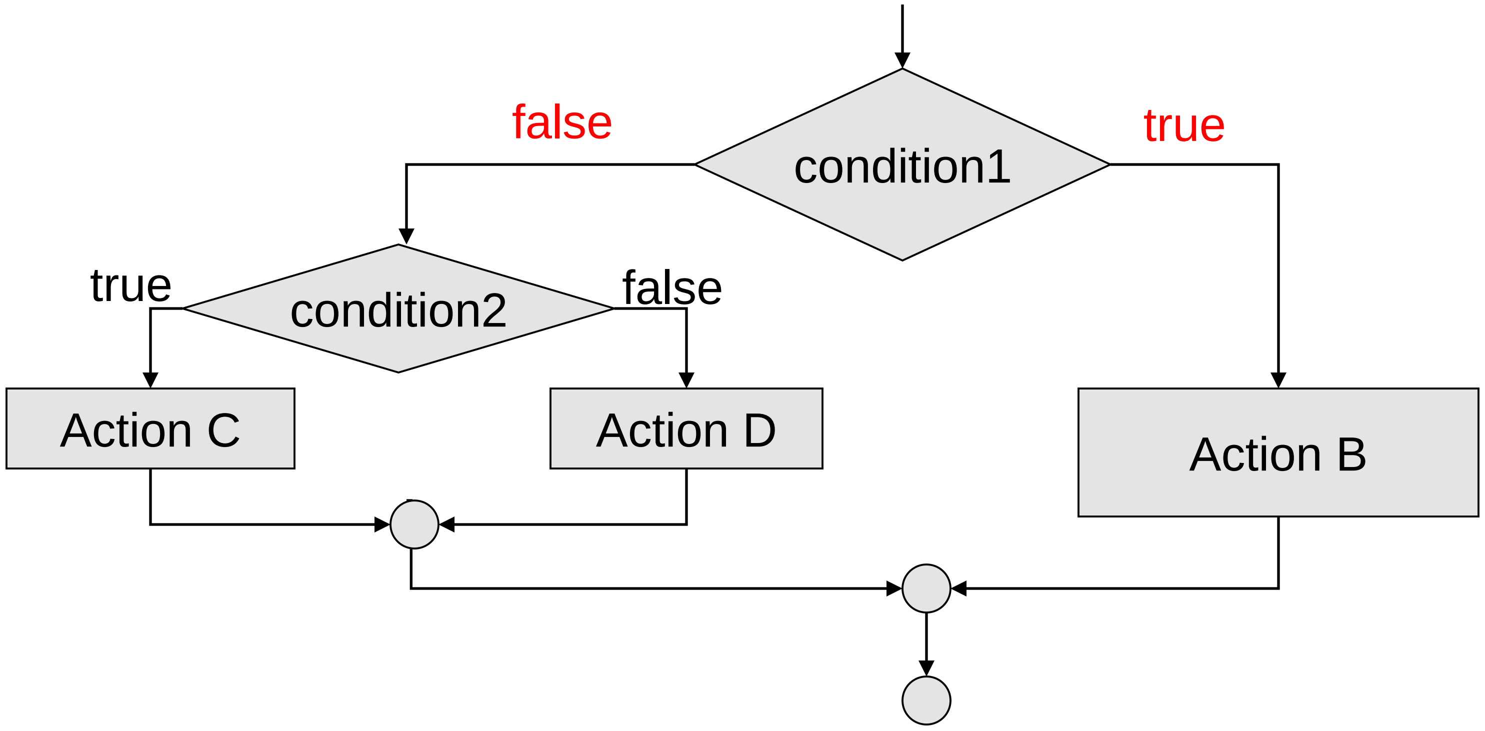
Nested **if…else**



```
if( condition1 )
{
      if( condition2 )
      {
      statements3  /* action C */
      }
      else
      {
       statements4 /* action D */
      }
}
else
{
       statements2  /* action B */
}
```

```
/* braces enclosing a single
      statement may be omitted */
if( condition1 )
      if( condition2 )
      {
      statements3  /* action C */
      }
      else
      {
       statements4 /* action D */
      }
else
{
       statements2  /* action B */
}
```

Nested **if…else** (2)



```
if( condition1 )
{
     statements2  /* action B */
}
else
{
    if( condition2 )
    {
    statements3  /* action C */
    }
    else
    {
     statements4 /* action D */
    } //end inner else
} //end outer else
```

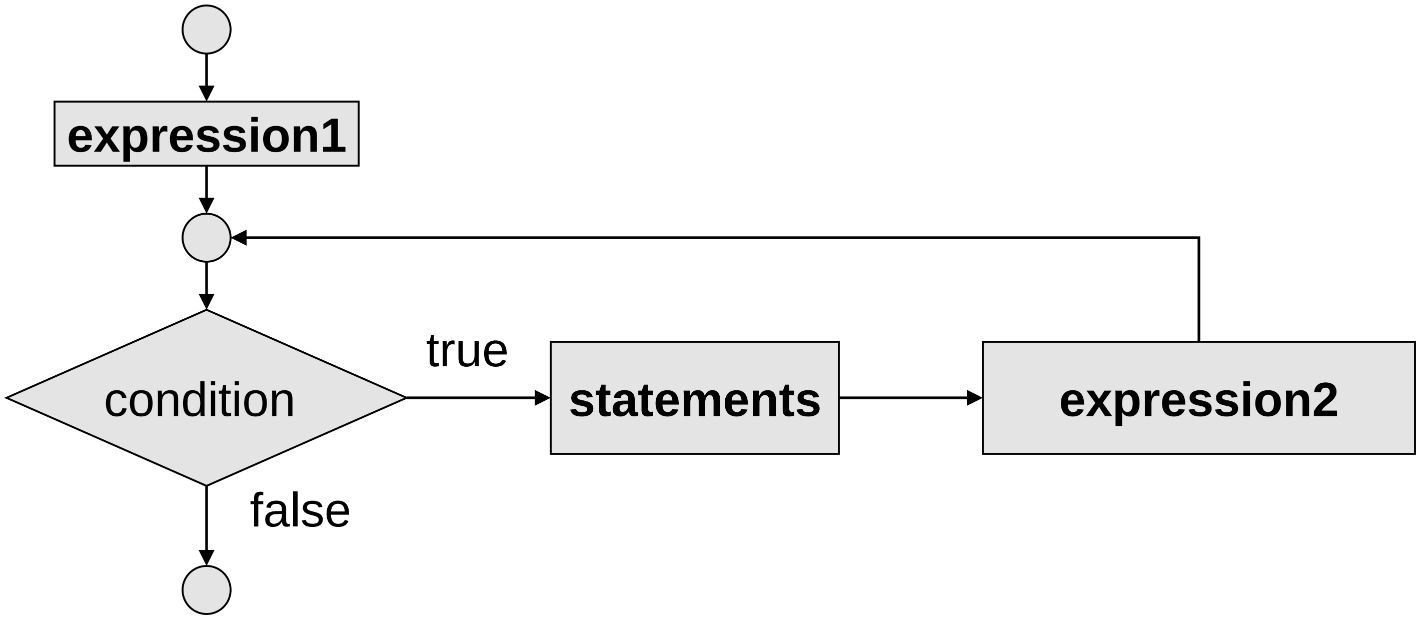
```
if( condition1 )
{
     statements2  /* action B */
}
else if( condition2 )
    {
    statements3  /* action C */
    }
else
    {
     statements4 /* action D */
    }
```

Repetition Structure

- A group of instructions that are repeatedly executed while a ***condition*** remains TRUE.
- ***while***
- ***do   while***
- ***for***

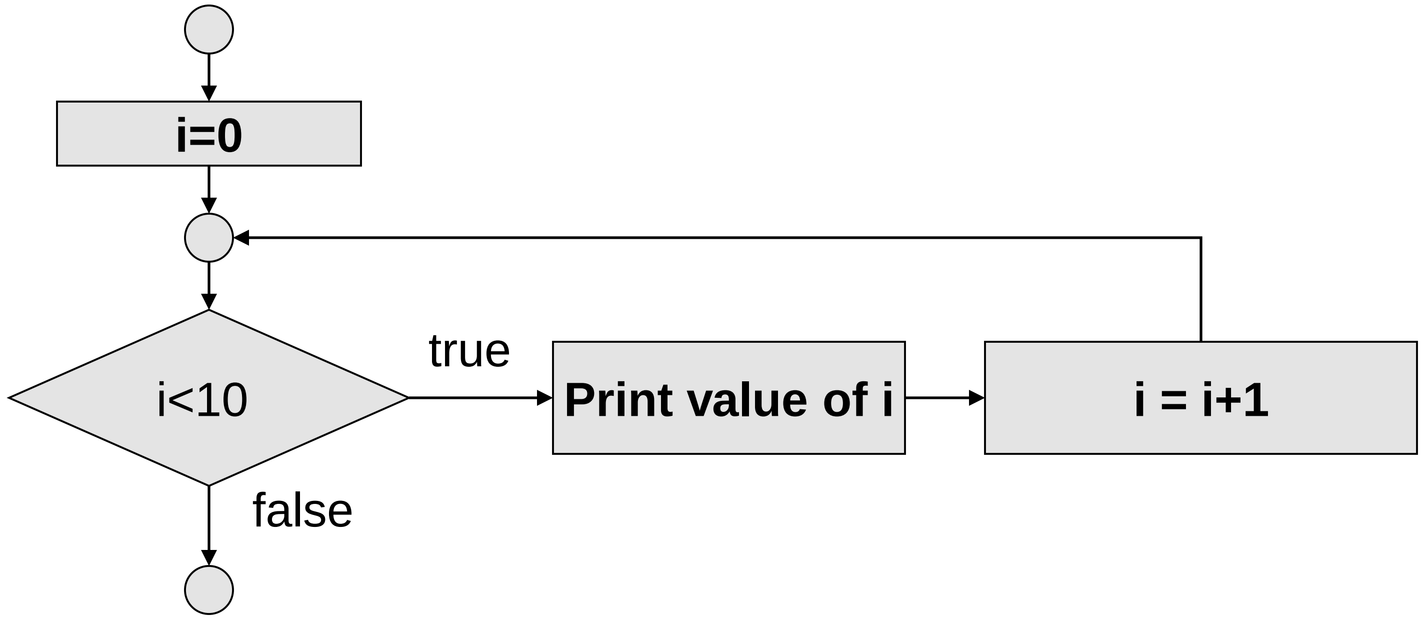
for statement

```
for( expression1; condition; expression2 )
{
        statements
}
```



for statement Example

```
for( i=0; i<10; i++ )
{
        printf("%d", i );
}
```



for Statement Example

Print 20 stars

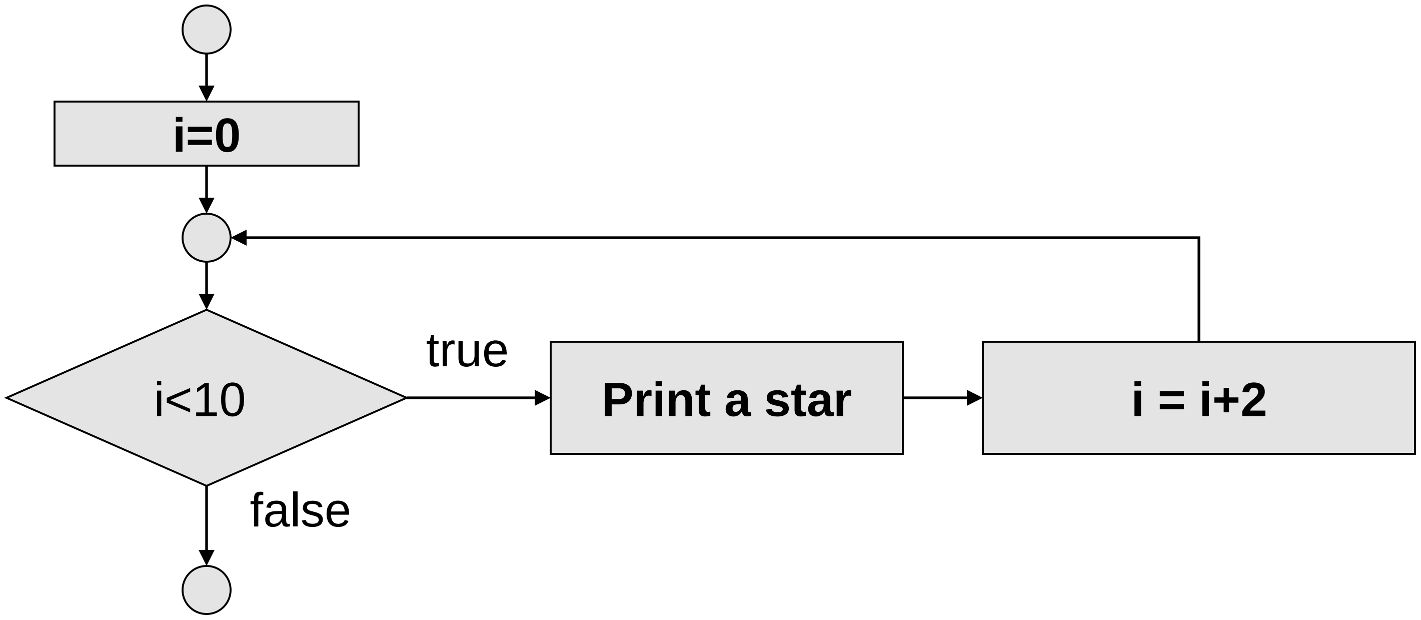
```
for  (i=0; i <20; i++)
     printf( “*”);
```

```
for  (i=1; i <= 20; i++)
     printf( “*”);
```

```
for  (i=20; i >=1; i--)
     printf( “*”);
```

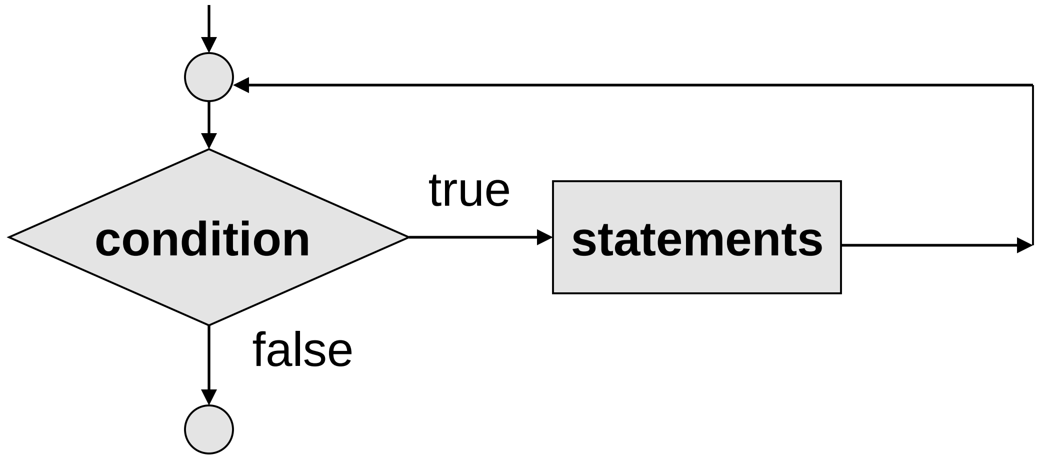
for statement Example

```
// how many stars are printed?
for( i=0; i<10; i +=2 )
{
        printf("*");
}
```



while Statement

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



for, while Statements Example

Assume that i is of type **int**.

```
for  (i=1; i <= 4; i++)
{
    printf( “%d\n”, i);
}
```

```
Output:
1
2
3
4
```

```
/* equivalent while loop */
i=1;
while  (i <= 4)
{
    printf( “%d\n”, i);
    i++;
}
```

for and Equivalent **while** Statement

```
for( expression1; condition; expression2 )
{
        statements
}
```

while equivalent

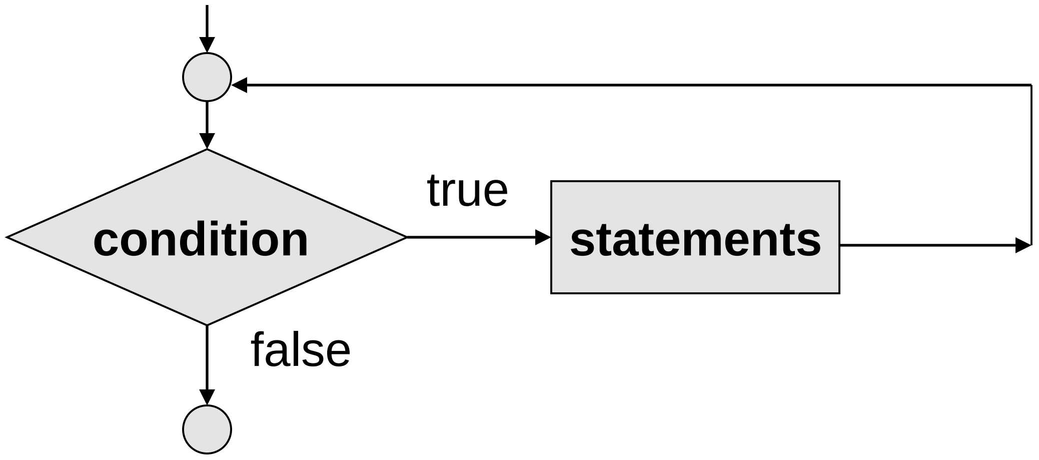
```
expression1;
while(condition)
{
        statements
        expression2;
}
```

while Statement

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

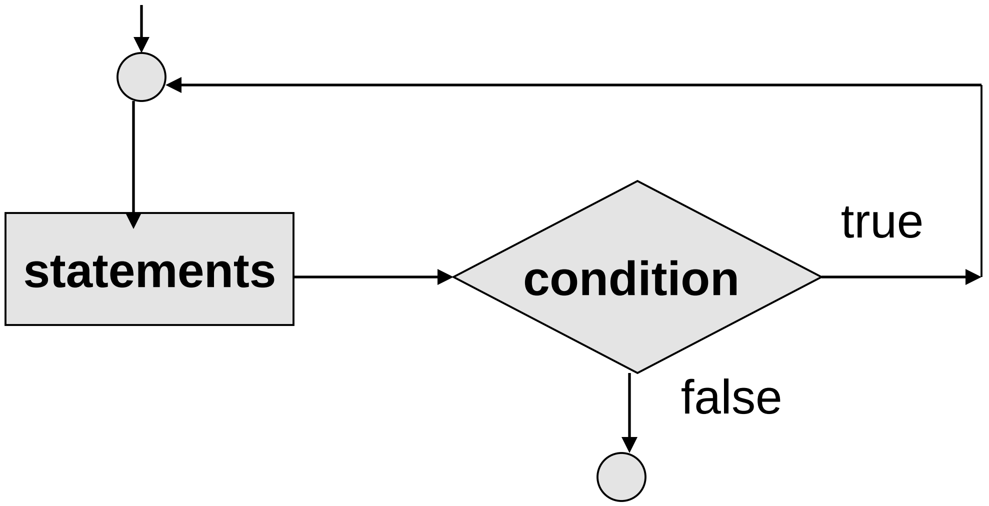
Equivalent **for** statement

```
for(  ; condition;  )
   {
         statements
   }
```



do…while Statement

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



Example

Assume that i and n are of type **int**.
Are the following pieces of code equivalent
(do they have the same effect all the time)?

```
i=1;
while  (i <= n)
{
    printf( “%d\n”, i);
    i++;
}
```

```
i=1;
do
{
    printf( “%d\n”, i);
    i++;
}
while (i<=n);
```