Structured Programming

Dr. Mohamed Khedr

Lecture 5

http://webmail.aast.edu/~khedr

Arithmetic Operators

Shortcut assignment

"Short cut" assignment operators combine an operation with an assignment.

`a += b`	`a = a + b`
`a -= b`	`a = a - b`
`a *= b`	`a = a * b`
`a /= b`	`a = a / b`
`a %= b`	`a = a % b`

For instance, instead of writing:

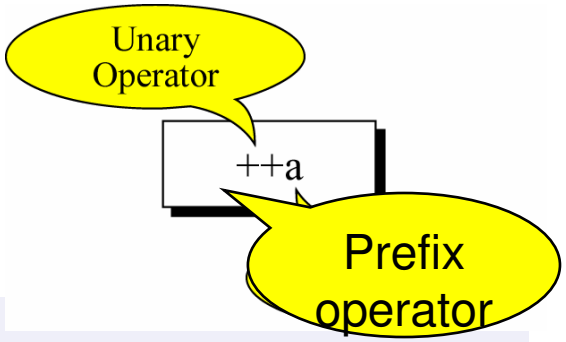
```
a = a + 1;
```

you could write

```
a += 1;
```

Arithmetic Operators

Prefix form



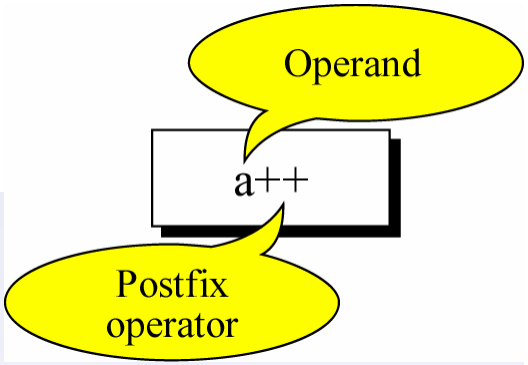
- Prefix increment and decrement operators increment or decrement the variable, then return its resulting value.

```
int a, b;
a = b = 10;
printf("%d\n", ++a);      /* Prints 11 */
printf("%d\n", a);        /* Prints 11 */
printf("%d\n", --b);      /* Prints 9 */
printf("%d\n", b);        /* Prints 9 */
```

- Remember: If the ++ comes *before* the variable, it increments *before* determining the result.

Arithmetic Operators

Postfix form



Postfix Increment and Decrement

- Postfix increment and decrement operators return the original value of the variable, then increment or decrement the variable.

```
int a, b;
a = b = 10;
printf("%d\n", a++);     /* Prints 10 */
printf("%d\n", a);       /* Prints 11 */
printf("%d\n", b--);     /* Prints 10 */
printf("%d\n", b);       /* Prints 9 */
```

Assignment Operators

- Syntax:

  `var = expression;`

  – Assign the value of expression to variable (**`var`**)

  Example:

```
int x, y, z;
 x = 5;
 y = 7;                ⇒   Z = (x = 5) + (y = 7)   much faster
 z = x + y;
```

```
int x, y, z;
 x = y = z = 0;        ⇒ same as   x = (y = (z = 0));
```

```
int x = y = z = 0;     ⇒ wrong !   int x = 0, y = 0, z = 0;
```

```
int i, j;
float f, g;
 i = f = 2.5;          ⇒ i = 2;      f = 2.5;
 g = j = 3.5;          ⇒ g = 3.0;    j = 3;
```

Short Hand Assignment

- Syntax

  `f = f op g` can be rewritten to be `f op= g`

  such as: `a = a + 2` ⇒ `a += 2`, `a = a - 2` ⇒ `a -= 2`, `a = a * 2` ⇒ `a *= 2`,
  `a = a / 2` ⇒ `a /= 2`, `a = a % 2` ⇒ `a %= 2`, `a = a << 2` ⇒ `a <<= 2`,
  `a = a & 2` ⇒ `a &= 2`, `a = a | 2` ⇒ `a |= 2`, `a = a ^ 2` ⇒ `a ^= 2`

  - No blanks between **op** and **=**
  - `x *= y + 1` is actually `x = x * (y+1)` rather than `x = x * y + 1`

    Example:

    ```
    q = q / (q+2)  ⇒  q /= q+2
    j = j << 2     ⇒  j <<= 2
    ```

  - Advantage: help compiler to produce more efficient code

    More complicated examples:

    ```
    int a=1, b=2, c=3, x=4, y=5;
    a += b += c *= x + y - 6;
    printf("%d %d %d %d\n",a,b,c,x,y); /* result is 12 11 9 4 5 */
    a += 5 + b += c += 2 + x + y;      /* wrong */
    a += 5 + (b+= c += 2 + x + y);     /* result is 22 16 14 4 5 */
    ```

Increment / Decrement Operators

++ (increment) -- (decrement)

- Prefix Operator
  - Before the variable, such as **++n** or **--n**
  - Increments or decrements the variable before using the variable
- Postfix Operator
  - After the variable, such as **n++** or **n--**
  - Increments or decrements the variable after using the variable

- `++n`
  1. Increment **n**
  2. Get value of **n** in expression
- `--n`
  1. Decrement **n**
  2. Get value of **n** in expression
- `n++`
  1. Get value of **n** in expression
  2. Increment **n**
- `n--`
  1. Get value of **n** in expression
  2. Decrement **n**

Increment / Decrement Operators (cont.)

– Simple cases

```
++i;
i++;      (i = i + 1; or i += 1;)
--i;
i--;      (i = i - 1; or i -= 1;)
```

Example:

```
i = 5;
i++; (or ++i;) ⇒ 6

i = 5;
i--; (or --i;)
printf("%d", i) ⇒ 4
```

– Complicated cases

```
i = 5;
j = 5 + ++i;
```

```
i = 5;
j = 5 + i++;
```

```
i = 5;
j = 5 + --i;
```

```
i = 5;
j = 5 + i--;
```

```
   i    j
   6   11
```

```
   6   10
```

```
   4    9
```

```
   4   10
```

Increment / Decrement Operators (cont.)

- Invalid cases

```
++3          3++           --3          3--
++(x+y+z)    (x+y+z)++     --(x+y+z)    (x+y+z)--
++x++        --x--         ++x--        --x++
```

Note: Can not increment or decrement constant and expression

```
i ++j        or      i --j (WRONG)
i + ++j      i + --j       i - --j   i - ++j      (OK)
```

Other Input / Output

`puts(line)` Print a string to standard output and append a newline

Example:
```
puts("12345");
```

`putchar(c)` Print a character to standard output

Example:
```
putchar('A');
```

`gets(line)` Read a string from standard input (until a newline is entered)

Example:
```
char buf[128];
gets(buf);  /* space is OK, and the '\n' won't be read in */
```

- Newline will be replaced by '\0'

`getchar()` Get a character from standard input

Example:
```
int c;
c = getchar();    /* c must be int */
```

- In-memory Format Conversion

```
sprintf(string, control, variables);
```

Program Control

- Standard C Statements

Outline

- This Topic Introduces
  - selection structure
    - **`if`**
    - **`if/else`**
  - repetition control structures
    - **`While`**
  - additional repetition control structures
    - **`for`**
    - **`do/while`**
  - **`switch`** additional multiple selection structure
  - **`break`** statement
    - Used for exiting immediately and rapidly from certain control structures
  - **`continue`** statement
    - Used for skipping the remainder of the body of a repetition structure and proceeding with the next iteration of the loop

Selection Structure: `if`

- Selection structure:
  - Used to choose among alternative courses of action
  - Pseudocode:

```
If (student's grade is greater than or equal to 60)
        Print "Passed"
```

- If condition **`true`**
  - Print statement executed and program goes on to next statement
  - If **`false`**, print statement is ignored and the program goes onto the next statement
  - Indenting makes programs easier to read
    - C ignores whitespace characters
- Pseudocode statement in C:

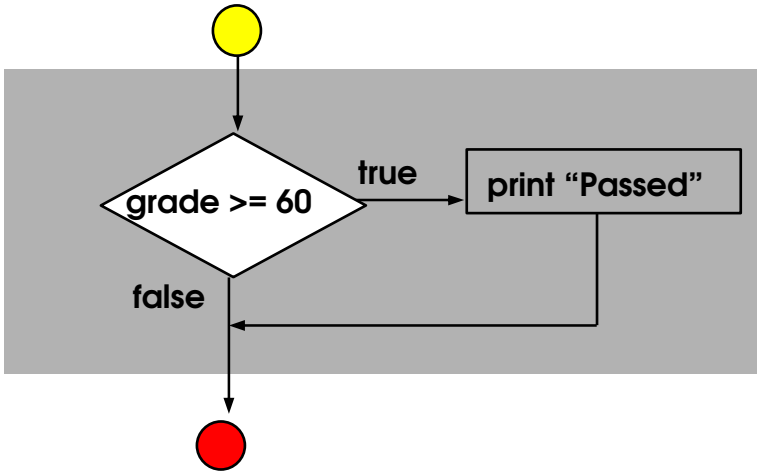
```
if ( grade >= 60 )
   printf( "Passed\n" );
```

  - C code corresponds closely to the pseudocode

The `if` Selection Structure (cont.)

- A decision can be made on any expression.
  - zero - **false**
  - nonzero - **true**
  - Example:

    `(3 - 4)` is **true**



Selection Structure: `if/else`

- **`if/else`**
  - **`if:`** only performs an action if the condition is **`true`**
  - **`if/else:`** Specifies an action to be performed both when the condition is **`true`** and when it is **`false`**
- Pseudocode:

```
If (student's grade is greater than or equal to 60)
   Print "Passed"
else
    Print "Failed"
```

  - Note spacing/indentation conventions
- C code:

```
if ( grade >= 60 )
   printf( "Passed\n");
else
   printf( "Failed\n");
```

The `if/else` Selection Structure

- Compound statement:
  - Set of statements within a pair of braces
  - Example:

```
if ( grade >= 60 )
   printf( "Passed.\n" );
else {
   printf( "Failed.\n" );
   printf( "You must take this course again.\n" );
  }
```

  - Without the braces,

```
if ( grade >= 60 )
   printf( "Passed.\n" );
else
   printf( "Failed.\n" );
printf( "You must take this course again.\n" );
```

    the statement

```
printf( "You must take this course again.\n" );
```

    would be executed under every condition.

3.6 The `if...else` Selection Statement

– Pseudocode for a nested `if...else` statement

```
If student's grade is greater than or equal to 90
  Print "A"
else
  If student's grade is greater than or equal to 80
     Print "B"
  else
     If student's grade is greater than or equal to 70
        Print "C"
     else
        If student's grade is greater than or equal to 60
           Print "D"
        else
           Print "F"
```

The Essentials of Repetition

- Loop
  - Group of instructions computer executes repeatedly while some condition remains **`true`**
- Counter-controlled repetition
  - Definite repetition: know how many times loop will execute
  - Control variable used to count repetitions
- Sentinel-controlled repetition
  - Indefinite repetition
  - Used when number of repetitions not known
  - Sentinel value indicates "end of data“

Essentials of Counter-Controlled Repetition

- Counter-controlled repetition requires
  - The name of a control variable (or loop counter)
  - The initial value of the control variable
  - A condition that tests for the final value of the control variable (i.e., whether looping should continue)
  - An increment (or decrement) by which the control variable is modified each time through the loop

Example:

```
int counter = 1;                /* initialization */
while ( counter <= 10 ) {       /* repetition condition */
   printf( "%d\n", counter );
   ++counter;                   /* increment */
}
```

  - The statement

```
int counter = 1;
```

    - Names **counter**
    - Declares it to be an integer
    - Reserves space for it in memory
    - Sets it to an initial value of **1**
    - This is ***not*** an executable statement, it is a declaration.

Repetition Structure: `while`

```
1    /* Fig. 3.6: fig03_06.c
2    Class average program with
3    counter-controlled repetition */
4   #include <stdio.h>
5
6   int main()
7   {
8    int counter, grade, total, average;
9
10     /* initialization phase */
11     total = 0;
12     counter = 1;
13
14     /* processing phase */
15     while ( counter <= 10 ) {
16        printf( "Enter grade: " );
17        scanf( "%d", &grade );
18        total = total + grade;
19        counter = counter + 1;
20     }
21
22     /* termination phase */
24     printf( "Class average is %d\n", average );
25
26     return 0;   /* indicate program ended successfully */
27  }
```

```
printf( "Enter grade, -1 to end: " );
scanf( "%d", &grade );
while ( grade != -1 ) {
   total = total + grade;
   counter = counter + 1;
   printf( "Enter grade, -1 to end: " );
   scanf( "%d", &grade );
}    /* termination phase */
if ( counter != 0 ) {
   average = ( float ) total / counter;
   printf( "Class average is %.2f", average );
}
else
   printf( "No grades were entered\n" );
```

Program Output:

```
Enter grade: 98
Enter grade: 76
Enter grade: 71
Enter grade: 87
Enter grade: 83
Enter grade: 90
Enter grade: 57
Enter grade: 79
Enter grade: 82
Enter grade: 94
Class average is 81
```

Repetition Structure: `for`

- **`for`** loops syntax

  for (initialization ; loopContinuationTest ; increment)
  statement

  Example: Prints the integers from one to ten

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

- For loops can usually be rewritten as **`while`** loops:

```
initialization;
while ( loopContinuationTest ) {
   statement;
   increment;
}
```

No semicolon `(;)` after last expression

- Initialization and increment
  - Can be comma-separated list of statements

    Example:

```
for ( i = 0, j = 0;  j + i <= 10; j++, i++)
   printf( "%d\n", j + i );
```

The `for` Structure (cont.)

- Arithmetic expressions
  - Initialization, loop-continuation, and increment can contain arithmetic expressions. If **`x`** equals **`2`** and **`y`** equals **`10`**

```
for ( j = x; j <= 4 * x * y; j += y / x )
```

  is equivalent to

```
for ( j = 2; j <= 80; j += 5 )
```

- Notes about the **`for`** structure:
  - "Increment" may be negative (decrement)
  - If the loop continuation condition is initially **`false`**
    - The body of the **`for`** structure is not performed (i.e. pre-test)
    - Control proceeds with the next statement after the **`for`** structure
  - Control variable
    - Often printed or used inside for body, but not necessarily

The `for` Structure (cont.)

```
1  /* Fig. 4.5: fig04_05.c
2     Summation with for */
3  #include <stdio.h>
4
5  int main()
6  {
7     int sum = 0, number;
8
9     for ( number = 2; number <= 100; number += 2 )
10       sum += number;
11
12    printf( "Sum is %d\n", sum );
13
14    return 0;
15 }
```

1. **Initialize variables**
2. **`for` repetition structure**

Program Output:

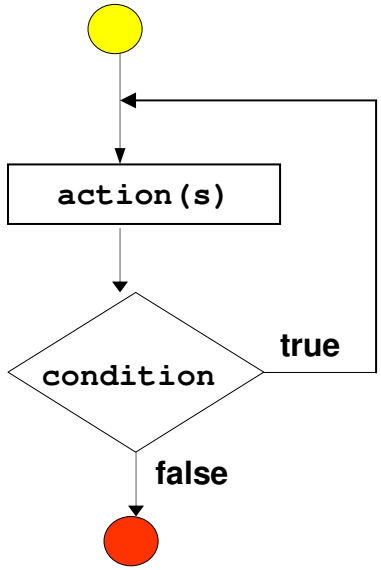
```
Sum is 2550
```

2 + 4 + 8 + … +100 = 2550

Repetition Structure: do/while

- The **do/while** repetition structure
  - Similar to the **while** structure
  - do/while is a "post-test" condition. The body of the loop is performed at least once.
    - All actions are performed at least once
  - Format:

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



Example: Prints the integers from **1** to **10**.

(letting counter = 1):

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

- Flowchart of the **do/while** repetition structure

Repetition Structure: do/while

```
1  /* Fig. 4.9: fig04_09.c
2     Using the do/while repetition structure */
3  #include <stdio.h>
4
5  int main()
6  {
7     int counter = 1;
8
9     do {
10       printf( "%d  ", counter );
11    } while ( ++counter <= 10 );
12
13    return 0;
14 }
```

1. Initialize variable
2. Loop
3. Print

Program Output:

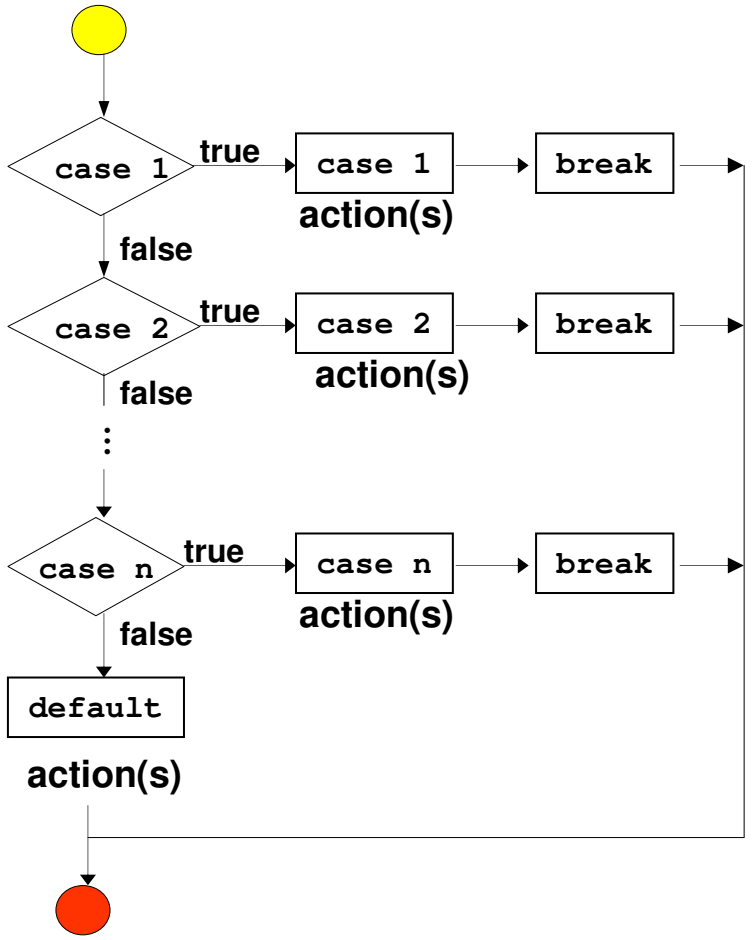
```
1  2  3  4  5  6  7  8  9  10
```

Multiple-Selection Structure: `switch`

- **`switch`**
  - Useful when a variable or expression is tested for all the values it can assume and different actions are taken
- Format
  - Series of **`case`** labels and an optional **`default`** case

```
switch ( value ){
   case '1':
      actions
   case '2':
      actions
   default:
      actions
   }
```

  - **`break;`** exits from structure
- Flowchart of the **`switch`** structure



```c
/* Fig. 4.7: fig04_07.c
 Counting letter grades */
#include <stdio.h>

int main()
{
 int grade;
 int aCount = 0, bCount = 0, cCount = 0, dCount = 0, 9
                fCount = 0;

   printf(  "Enter the letter grades.\n"  );
   printf(  "Enter the EOF character to end input.\n"  );

   while ( ( grade = getchar() ) != EOF ) {

      switch ( grade ) {    /* switch nested in while */

         case 'A': case 'a':  /* grade was uppercase A */
            ++aCount;         /* or lowercase a */
            break;

         case 'B': case 'b':  /* grade was uppercase B */
            ++bCount;         /* or lowercase b */
            break;

         case 'C': case 'c':  /* grade was uppercase C */
            ++cCount;         /* or lowercase c */
            break;

         case 'D': case 'd':  /* grade was uppercase D */
            ++dCount;         /* or lowercase d */
            break;

         case 'F': case 'f':  /* grade was uppercase F */
            ++fCount;         /* or lowercase f */
            break;

```

1. **Initialize variables**
2. **Input data**
3. **Use switch loop to update count**

```
38          case '\n': case' ':  /* ignore these in input */
39             break;
40
41          default:        /* catch all other characters */
42             printf( "Incorrect letter grade entered." );
43             printf( " Enter a new grade.\n" );
44             break;
45       }
46    }
47
48    printf( "\nTotals for each letter grade are:\n" );
49    printf( "A: %d\n", aCount );
50    printf( "B: %d\n", bCount );
51    printf( "C: %d\n", cCount );
52    printf( "D: %d\n", dCount );
53    printf( "F: %d\n", fCount );
54
55    return 0;
56 }
```

4. Print results

Program Output:

```
Enter the letter grades.
Enter the EOF character to end input.
A
B
C
C
A
D
F
C
E
Incorrect letter grade entered. Enter a new grade.
D
A
B

Totals for each letter grade are:
A: 3
B: 2
C: 3
D: 2
F: 1
```

The `break` and `continue` Statements

- **`break`**
  - Causes immediate exit from a **`while`**, **`for`**, **`do/while`** or **`switch`** structure
  - Program execution continues with the first statement after the structure
  - Common uses of the **`break`** statement
    - Escape early from a loop
    - Skip the remainder of a **`switch`** structure
- **`continue`**
  - Skips the remaining statements in the body of a **`while`**, **`for`** or **`do/while`** structure
    - Proceeds with the next iteration of the loop
  - **`while`** and **`do/while`**
    - Loop-continuation test is evaluated immediately after the **`continue`** statement is executed
  - **`for`**
    - Increment expression is executed, then the loop-continuation test is evaluated

`continue` Statement

```
while (expr) {
  statement
  …
  continue;
skip statement
  …
}
```

```
do {
statement
   …
   continue;
skip statement
   …
} while(expr)
```

```
for (expr1; expr2; expr3) {
   statement
   …
   continue;
   statement skip
   …
}
```

break Statement

```
    while (expr) {
      statement;
      …
      if (expr)
        break;
      statements;
    }
    statement;
    …
for (expr1; expr2; expr3)
{   statement
    …
    if (expr)
          break;
    statements;
}
statements;
```

```
switch (i) {
    case 1:
          statement_1;
    case 2:
          statement_2;
    case 3:
          statement_3;
          break;
    case 4:
          statement_4;
    }
statements;
```

Equality (==) vs. 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: using **`==`**:

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

    - Checks **`paycode`**, if it is **`4`** then a bonus is awarded

    Example: replacing **`==`** with **`=`**:

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

    - This sets **`paycode`** to **`4`**
    - **`4`** is nonzero, so expression is **`true`**, and bonus awarded no matter what the **`paycode`** was
  - Logic error, not a syntax error

Examples

Ex_1:

```
  if (i=1) y = 3;
```

⇒ y = 3 is always executed
this is not the same as

```
  if (i==1) y = 3;
```

Ex_2:

```
  if (i!=0) y=3;
```

⇒ `if (i) y=3;`

Ex_3:

```
if (i==0) y=3;
```

⇒ `if (!i) y=3;`

Examples:

```
Ex_1:
if (i>2)
     if (j==3)
          y=4;
     else
       y=5;
```

≠

```
if (i>2) {
    if (j==3)
          y=4;
}
else
     y=5;
```

=

```
if (i>2)
     if (j==3)
          y=4;
     else
          ;
else
          y=5;
```

Ex_2:

```
if (a>b)
      c = a;
else
      c = b;
```

⇒ `c=(a>b)?a:b`

```
if (x==5)
     y = 1;
else
     y = 0;
```

⇒ `y = (x==5);`

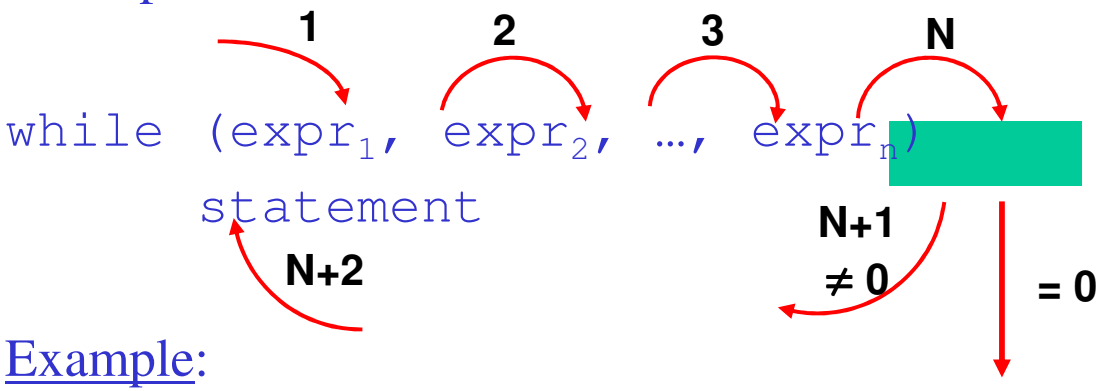
```
if (x<6)
    y = 1;
else
    y = 2;
```

⇒ `y = 2-(x<6);`

⇒ `or y = 1+(x>=6);`

Examples:

- **`while`** loop:



Example:

```
while (scanf("%d", &i), i--)
      printf("%d",i);
```

- **Switch**

```
switch (i) {
 case 1: j+=5;
 case 2;
 case 3: j+=4;
 case 4: j+=3;
}
```