

ITC213: STRUCTURED PROGRAMMING

Bhaskar Shrestha

National College of Computer Studies
Tribhuvan University

Lecture 08: Control Statements

Readings: Chapter 6

Control Statements and Their Types

- A control statement is a statement that causes a group of C statements to execute in a manner that doesn't relate to the order it appears in the source code
- Types of control statements
 - **Selection:** also known as branching created using `if - else`, `switch`
 - **Repetition:** also known as looping created using `while`, `do - while`, `for`
 - **Jump:** created using `goto`, `break`, `continue`
 - **Label:** created using `case`, `default`, *named label*

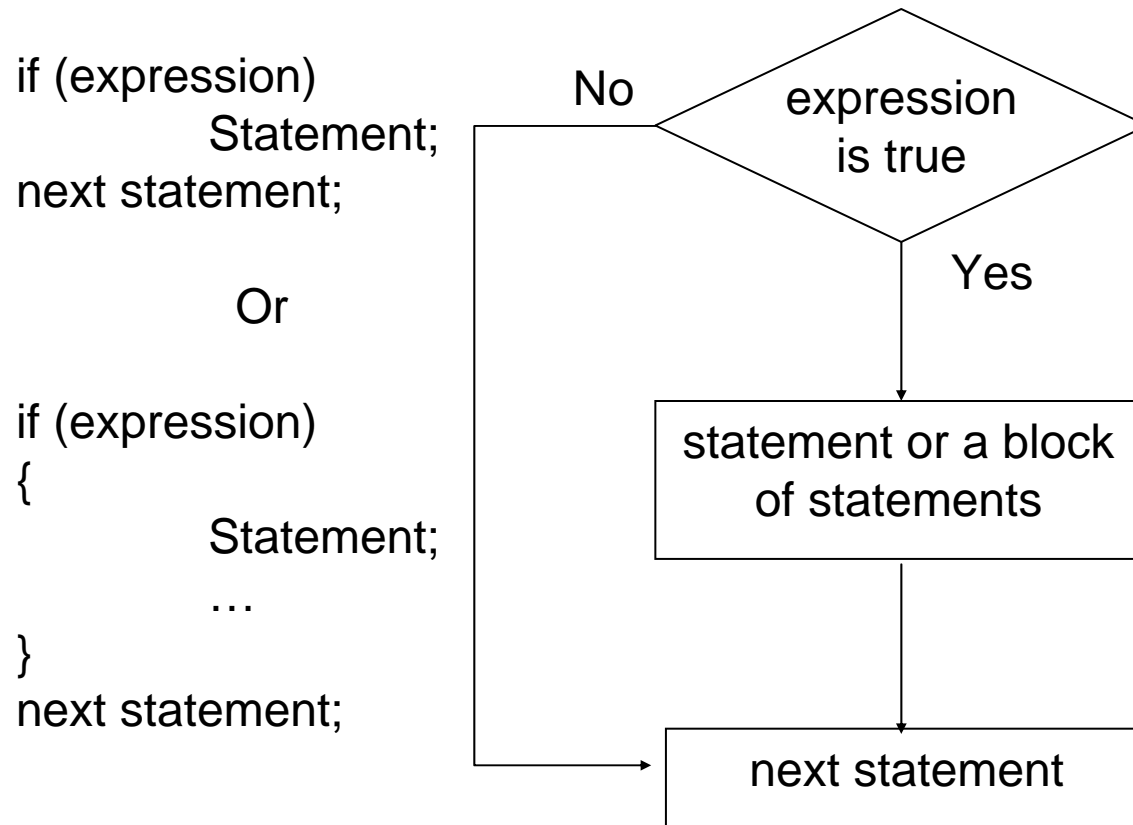
Selection

- A selection statement is a one in which one group of statement is selected from several available groups, depending on the outcome of a logical test
- C supports two selection statements: `if` and `switch`
 - In addition, the `?:` operator is an alternative to `if` in certain circumstances
- 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

The `if` statement

- Allows your program to execute a single statement, or a block of statements enclosed between braces, if a given condition is *true*
- `if (expression)`
statement
- If *expression* evaluates to true (any non-zero value), *statement* is executed. If *expression* evaluates to false (0), *statement* is not executed
 - In either case, execution then passes to whatever code follows the `if` statement
- *expression* can be any valid C expression that produces a scalar value and *statement* can be simple or compound or even another control statement

Flow Chart of if



Examples of i f

```
i f (x > y)
  y = x;
```

assigns the value of **x** to **y** only if **x** is greater than **y**. If **x** is not greater than **y**, no assignment takes place.

```
i f (letter == 'A' )
  printf("The first capital \n");
printf("After i f\n");
```

if the value of **letter** is '**A**', the text **The first capital** is printed otherwise not. The last statement **After if** always gets printed

The `else` clause in an `if` statement

- An `if` statement can optionally include an `else` clause
- `if` (*expression*)
 statement1
`else`
 statement2
- If *expression* evaluates to true, *statement1* is executed. If *expression* evaluates to false, *statement2* is executed. Both *statement1* and *statement2* can be compound statements or blocks
- Using if-else, you can specify an action to be performed both when the condition is *true* and when it is *false*

Flow Chart of i f - e l s e

Example of i f with e l s e clause

```
i f (age >= 18)
    printf("You can vote\n");
e l s e
    printf("You can' t vote\n");
```

```
i f (status == 'S' )
    tax = 0.20 * pay;
e l s e
    tax = 0.14 * pay;
printf("Tax: %f\n", tax);
```

```
i f (pastdue > 0)
{
    printf("account no. %d is overdue", actno);
    credi t = 0;
}
e l s e
    credi t = 1000.0;
```

More i f - e l s e example

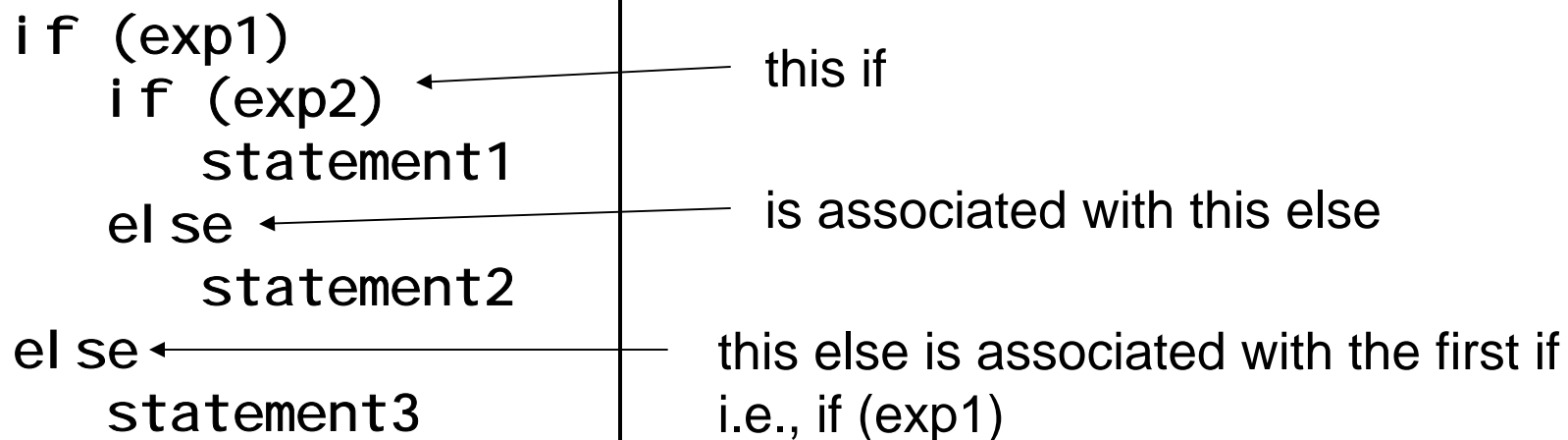
```
i f (c i r c l e)
{
    scanf("%f", &radi us);
    area = 3.14159 * radi us * radi us;
    printf("Area of c i r c l e = %f", area);
}
e l s e
{
    scanf("%f %f", &l e n g t h, &w i d t h);
    area = l e n g t h * w i d t h;
    print("Area of rectangl e = %f", area);
}
```

If **circle** is assigned a nonzero value, the **radius** of circle is read into the computer; the **area** is calculated and then displayed. If the value of **circle** is zero, however, then the **length** and **width** of rectangle are read into the computer, the **area** is calculated and then displayed

Nested-ifs

- A nested if is an if that is target of another if or else
- In a nested if, an else statement always refers to the nearest if statement that is within the same block as the else and that is not already associated with an else

```
if (exp1)
  if (exp2) ← this if
    statement1
  else ← is associated with this else
    statement2
else ← this else is associated with the first if
      i.e., if (exp1)
  statement3
```

The diagram shows a code block with three levels of indentation. The first level is 'if (exp1)'. The second level contains 'if (exp2)' and 'else'. The third level contains 'statement1' and 'statement2'. The fourth level contains 'statement3'. Arrows point from the text on the right to the corresponding parts of the code: 'this if' points to the second-level 'if (exp2)'; 'is associated with this else' points to the second-level 'else'; and 'this else is associated with the first if i.e., if (exp1)' points to the first-level 'else'.

Example

```
if(age >= 21 && (income >= 80000 || balance >= 200000))
    if(2*income < balance/2)
        loan = 2 * income;
    else
        loan = balance/2;
else
    loan = 0.0;
```

```
if (x > y)
    if (x > z)
        printf("x is greater");
    else
        printf("z is greater");
else
    if (y > z)
        printf("y is greater");
    else
        printf("z is greater");
```

if-else-if ladder

- A common programming construct is the `if-else-if` ladder because of its appearance

```
if (expression)
    statement;
else
    if (expression)
        statement;
    else
        if (expression)
            statement;
        .
        else
            statement;
```

The conditions are evaluated from the top downward. As soon as a true condition is found, the statement associated with it is executed and the rest of the ladder is bypassed. If none of the conditions are true, the final else is executed. If the final else is not present, no actions take place if all other conditions are false.

if-else-if example

```
char ch;
scanf("%c", &ch);
if (ch >= 'a' && ch <= 'z' )
    printf("lowercase letter");
else if (ch >= 'A' && ch <= 'Z' )
    printf("uppercase letter");
else if (ch >= '0' && ch <= '9' )
    printf("a digit");
else if (ch==' ' ||ch=='\t' ||ch=='\n' )
    printf("whitespace character");
else
    printf("an unknown character");
```

Repetition

- Repetition is the process of executing a group of statements more than one time as long as some condition remains true
- Repetition in C can be implemented using three control statements: `while`, `do – while`, `for`
- Pseudocode:
 - *While there are more items on my shopping list
Purchase next item and cross it off my list*
- Also known as iteration/looping statements

Counter-controlled repetition

- Definite repetition: know how many times loop will execute
- Control variable used to count repetitions
- 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

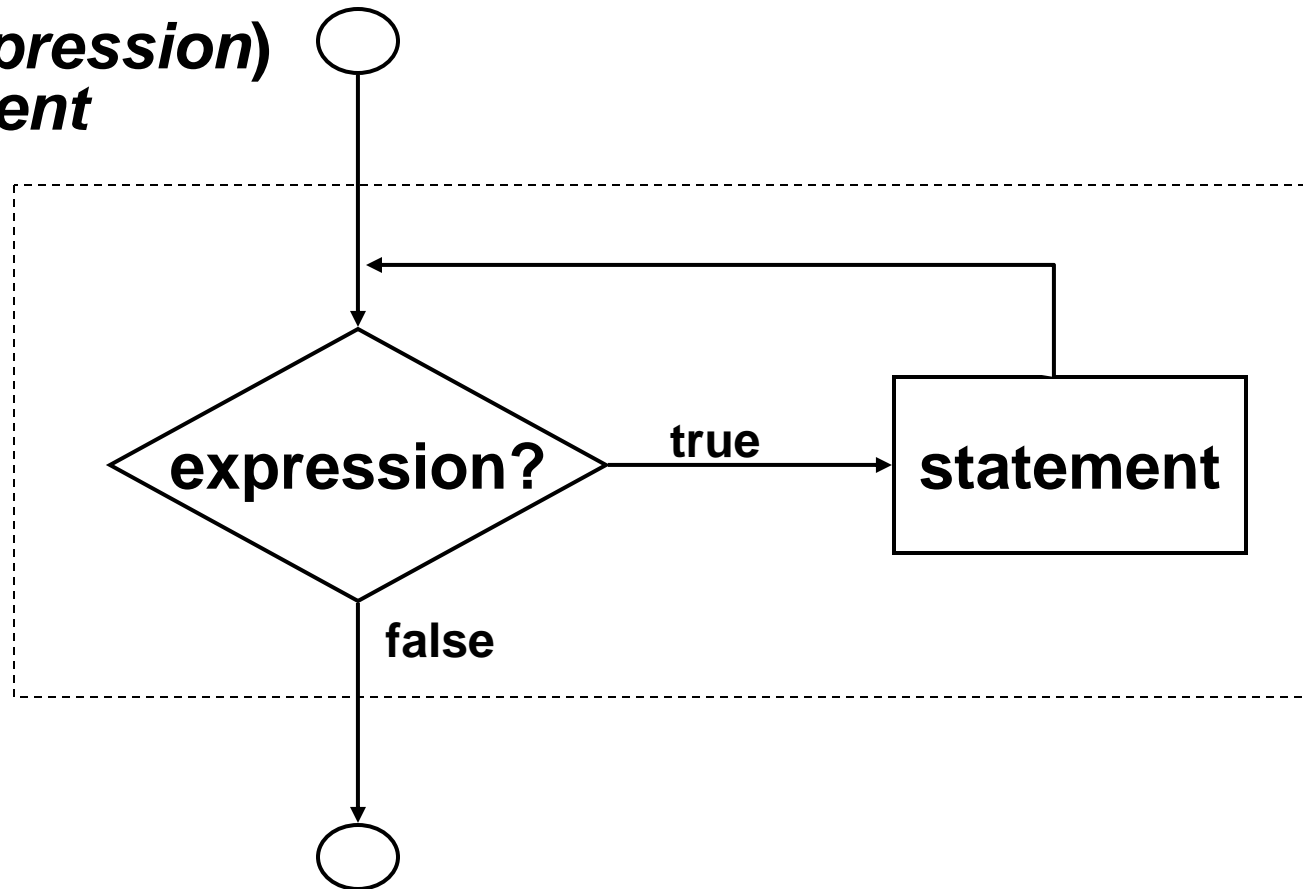
- A class of ten students took a quiz. The grades (integers in the range 0 to 100) for this quiz are available to you. Determine the class average on the quiz
- Pseudocode:
 - Set total to zero
 - Set grade counter to one
 - While grade counter is less than or equal to ten
 - Input the next grade
 - Add the grade into the total
 - Add one to the grade counter
 - Set the class average to the total divided by ten
 - Print the class average

The `while` statement

- Executes a block of statements as long as a specified condition is true
- `while (expression) statement`
- here *statement* is executed as long as the *expression* evaluates to *true*.
 - First *expression* is evaluated
 - If *expression* is nonzero (*true*), then *statement* is executed and control is passed back to the beginning of the `while` statement, otherwise control passes to next statement following the `while` statement

Flowchart of while statement

**while (expression)
statement**



The while statement

- As with `if`, *expression* can be any valid C expression that produces a scalar value and *statement* can be simple or compound or may be a control statement
- The *statement* inside the `while` loop must include some feature that eventually alters the value of the *expression*, thus providing a stopping condition for the loop

```
/* Prints hello world 10 times */  
counter = 1;           /* initialization */  
while (counter <= 10) /* repetition condition */  
{  
    printf("Hello World\n");  
    ++counter;        /* increment */  
}
```

More Examples

```
/* Prints digits 0 through 9 */  
int digit = 0;  
while (digit <= 9)  
{  
    printf("%d\n", digit);  
    ++digit;  
}
```

```
/* Prints digits 9 through 0 */  
int digit = 9;  
while (digit >= 0)  
{  
    printf("%d\n", digit);  
    --digit;  
}
```

Example: Calculation of average grade

```
int grade, counter;
int total;
float average;

total = 0;
counter = 1;
while (counter <= 10)
{
    printf("Enter grade: ");
    scanf("%d", &grade);
    total += grade;
    ++counter;
}
average = (float) total / 10;
printf("The average grade is %.2f\n", average);
```

Sentinel-Controlled Repetition

- Problem :
 - *Develop a class-averaging program that will process an arbitrary number of grades each time the program is run.*
 - Unknown number of students
 - How will the program know to end?
- Use sentinel value
 - Also called signal value, dummy value, or flag value
 - Indicates “end of data entry.”
 - Loop ends when user inputs the sentinel value
 - Sentinel value chosen so it cannot be confused with a regular input (such as -1 in this case)

Example: Calculation of average grade

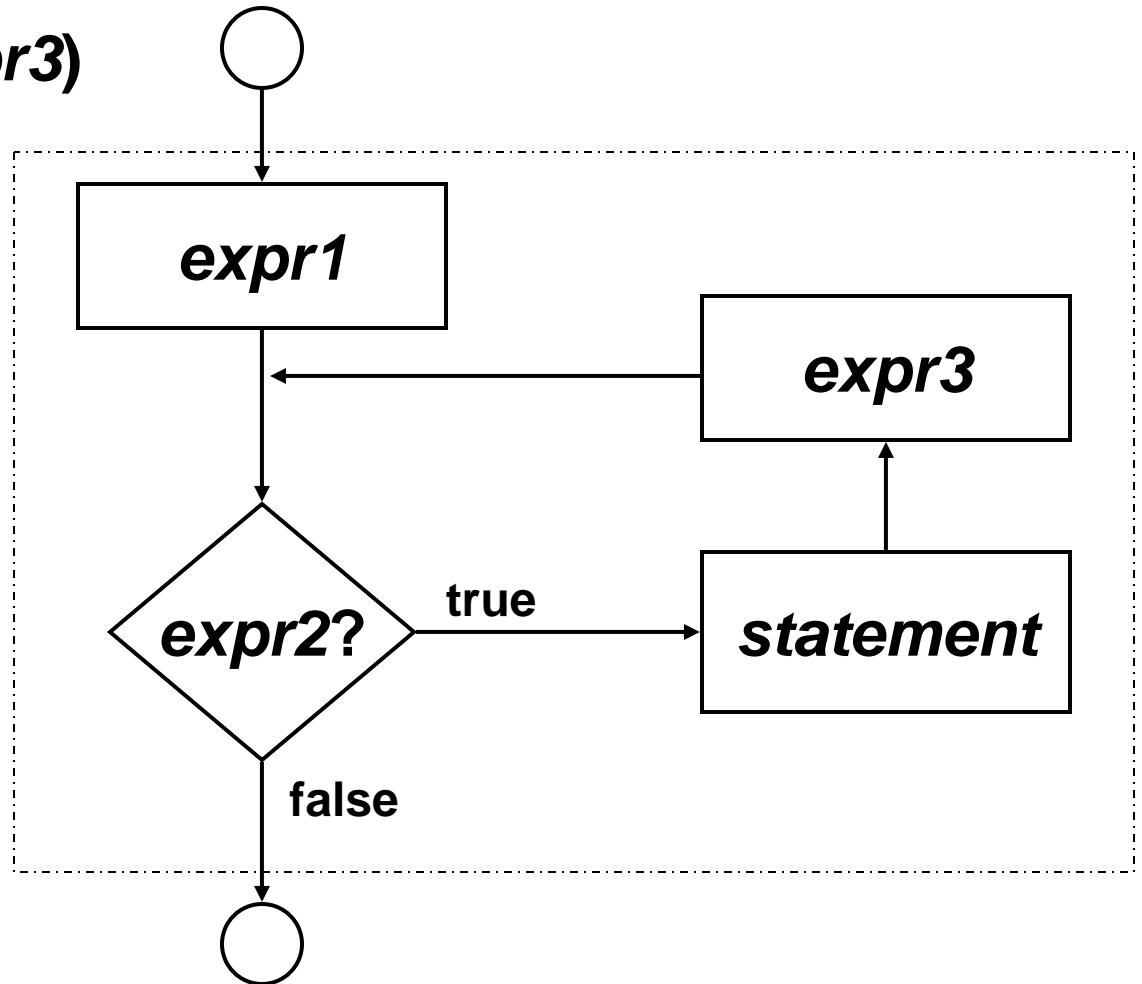
```
total = 0;
counter = 0;
printf("Enter grade (-1 to stop): ");
scanf("%d", &grade);
while (grade != -1) {
    total += grade;
    ++counter;
    printf("Enter grade (-1 to stop): ");
    scanf("%d", &grade);
}
if (counter != 0) {
    average = (float) total / counter;
    printf("The average grade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

The for statement

- `for (expr1; expr2; expr3)
statement`
- First *expr1* is evaluated.
- Then *expr2* is evaluated.
- If *expr2* is nonzero (*true*), then
 - *statement* is executed,
 - *expr3* is evaluated
 - control passes back to the beginning of the **for** loop again, except that evaluation of *expr1* is skipped.
- The process continues until *expr2* is zero (*false*), at which point control passes to next statement following the **for** statement

Flowchart of for statement

**for (*expr1*; *expr2*; *expr3*)
statement**



The for statement

- Typically:
 - *expr1* is used to initialize the loop control variable and is an assignment expression
 - *expr2* is a logical expression and represents a condition that must be true for the loop to continue
 - *expr3* is used to alter the value of control variable and is often an increment/decrement or assignment expression
- ```
for (counter = 1; counter <= 10; counter++)
 printf("Hello World\n");
```

# Examples

```
/* Prints digits 0 through 9 */
for (digit = 0; digit <= 9; digit++)
 printf("%d\n", digit);
```

```
/* Prints digits 9 through 0 */
for (digit = 9; digit >= 0; digit--)
 printf("%d\n", digit);
```

```
/* Sum of first n natural numbers */
scanf("%d", &n);
sum = 0;
for (counter = 1; counter <= n; counter++)
 sum += counter;
printf("The sum of first %d numbers is %d", n, sum);
```

# for and while loop

- Every for statement can be written in terms of while statement and vice versa

**for (*expr1*; *expr2*; *expr3*)  
    *statement***

```
expr1;
while (expr2)
{
 statement
 expr3;
}
```

# The do-while statement

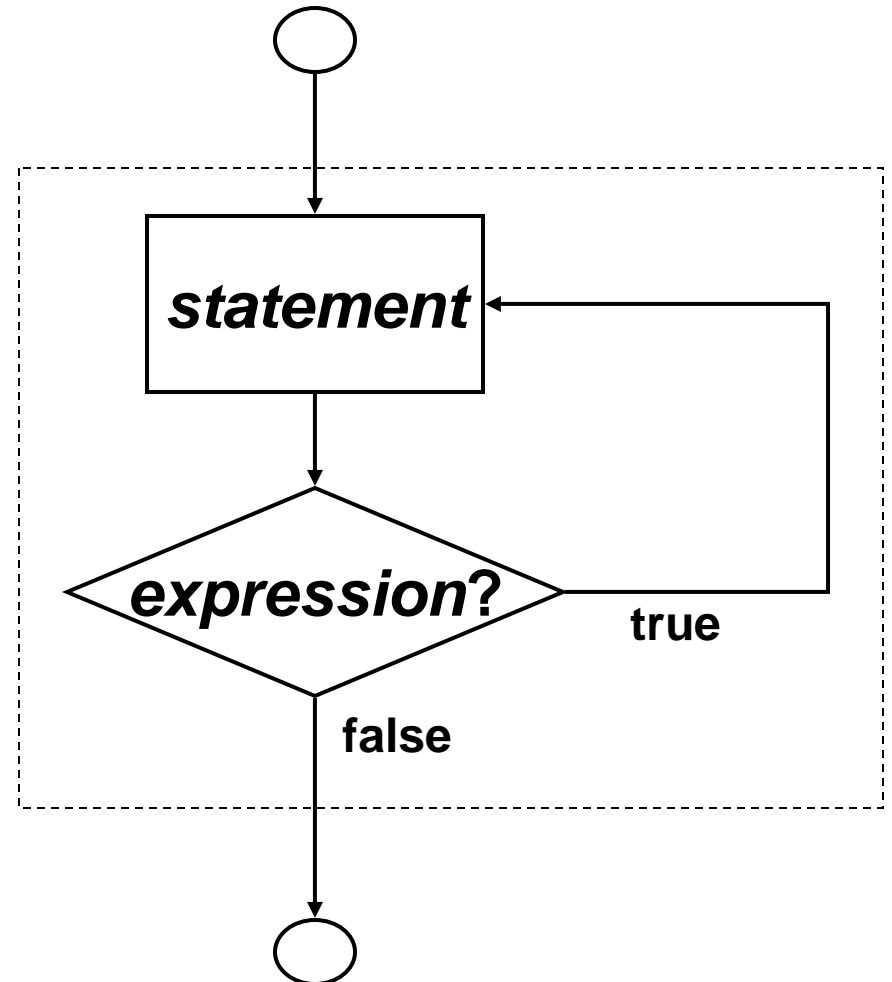
- Similar to the while statement but condition for repetition tested after the body of the loop is performed
- do  
    *statement*  
while (*expr*);
  - First *statement* is executed, and *expr* is evaluated
  - If the value of *expr* is nonzero (*true*), then control passes back to the beginning of the do statement and process repeats itself
  - When *expr* is zero (*false*), control passes to next statement following the do-while statement

# Flowchart of do-while statement

```
do
 statement
while (expr);
```

**Note that a semicolon is present after the closing parenthesis of the expression**

**Note that statement will always be executed at least once, since the test for repetition does not occur until the end of the first pass through the loop**





# The do-while statement

- As with while, *expr* can be any valid C expression that produces a scalar value and *statement* can be simple or compound or may be a control statement

```
counter = 1;
do {
 printf("Hello World\n");
 counter++;
} while (counter <= 10);
```

```
digit = 0;
do {
 printf("%d\n", digit);
 digit++;
} while (digit <= 9);
```

# More Examples

- The do-while statement is most appropriate when the loop body must be executed at least once

```
/* Read a number that is between 1 and 99 */
/* If not, reread the number */
int n;
do
{
 printf("Enter a number between 1 and 99: ");
 scanf("%d", &n);
} while (n < 1 || n > 99);
```

# On Loop Statements

- Because of the features that are built into the `for` statement, it is particularly well suited for loops in which the number of passes is known in advance
- `while` loops should be used when the no of times the statements inside the loop to be executed is not known in advance
- Use `do-while` loop, when you want the loop body to execute at least once for the first time regardless of the outcome of condition

# Comma Operator in for statement

- Comma operator can be used for multiple initialization and multiple processing of loop control variables in a for statement

```
/* sum of numbers from 1 to n */
for (sum = 0, i = 1; i <= n; i++)
 sum += i;
```

```
/*printing numbers from 1 to n, n to 1 */
for (i = 1, j = n; i <= n; i++, j--)
 printf("%2d %2d\n", i, j);
```

# The 3 expressions in for statement

- You can omit, any of the three expression of the for statement
  - However, semicolons must be present
- If you omit first or third expression, nothing happens at the time of their evaluation
- If you omit second expression, it will assume the value 1 (true)

```
int digit = 0;
for (; digit <= 9 ;)
 printf("%d\n", digit++);
```

# The `switch` Multiple-Selection Statement

- `switch` is a multiple branch selection statement, that successively tests the value of an expression against a list of integer or character constants
  - When a match is found, the statements associated with that constant are executed.
- Useful when a variable or expression is tested for all the values it can assume and different actions are taken

```
switch (expression)
{
 case constant1:
 statement sequence
 break;
 case constant2:
 statement sequence
 break;

 ...
 case constantn:
 statement sequence
 break;
 default:
 statement sequence
 break;
}
```

# The `switch` statement

- The *expression* must evaluate to an integer type
- The value of *expression* is tested against the constants present in the case labels
- When a match is found, the *statement sequence*, if present, associated with that case is executed until the `break` statement or the end of the `switch` statement is reached
- The statement sequence following `default` label is executed if no matches are found
  - The `default` label is optional, and if it is not present, no action takes place if all matches fail

# Examples

```
choi ce = getchar();
swi tch (choi ce)
{
case ' r' :
 pri ntf("RED");
 break;

case ' w' :
 pri ntf("WHI TE");
 break;

case ' b' :
 pri ntf("BLUE");
 break;

defaul t:
 pri ntf("Unknown");
}
}
```

```
scanf("%d", &n);
swi tch (n) {
case 1:
case 2:
 pri ntf("1 or 2");
 break;

case 3:
case 4:
 pri ntf("3 or 4");

case 5:
case 6:
 pri ntf("5 or 6?");
 pri ntf("or may be 3 or 4");

defaul t:
 break;
}
```



# Things to remember with `switch`

- A `switch` statement can only be used to test for equality of an expression
  - You cannot use relational or logical expression like in `if`
- `switch` expression must evaluate to an integral value
- No two case constants can be same
- Omission of a `break` statement causes execution to go to next case label
- The statement sequence after the `default` label is executed when no case constants matches the expression value

# Nested Control Statements

- Loops, like `while`, `for` statements, can be nested, one within another
  - The inner and outer loops need not be generated by same type of control structure
  - It is essential, however, that one loop be completely embedded within the other—there can be no overlap
  - Each loop must be controlled by a different index
- Nested control statements can also involve both loops and `if – else` statements
  - Thus, a loop can be nested within an `if – else` statement, and an `if – else` statement can be nested within a loop

# Examples (if nested inside loops)

```
/*Print numbers between 1 to n that are divisible by 3 or 5*/
scanf("%d", &n);
for (i = 1; i <= n; i++)
 if (i%3==0 || i%5 == 0)
 printf("%d\n", i);
```

```
/*count no of characters, excluding spaces in a line of text*/
char ch;
unsigned numchars = 0;
ch = getchar();
while (ch != '\n') {
 if (ch != ' ' || ch != '\t')
 numchars++;
 ch = getchar();
}
printf("Number of characters is %u\n", numchars);
```

# Examples (i f-e l s e nested inside loops)

```
i n t numal pha, numother;
char ch;
numal pha=numother=0;
ch = getchar();
whi l e (ch != '\n') {
 i f (ch>=' A' && ch<=' Z' || ch >= ' a' && ch <= ' z')
 numal pha++;
 e l s e
 numother++;
 ch = getchar();
}
pri n t f("Al phabets: %d, Other: %d", numal pha, numother);
```

# Nested for loops

```
/* Prints a multiplication table */
int i, j;
int size;
scanf("%d", &size);
for (i = 1; i <= size; i++) {
 for (j = 1; j <= size; j++)
 printf("%3d", i*j);
 printf("\n");
}
```

```
/* Print the first n factorials */
int i, j, n;
long int prod;
scanf("%d", &n);
for (i = 0; i <= n; i++) {
 prod = 1;
 for (j = 1; j <= i; j++)
 prod *= j;
 printf("%d ", prod);
}
```

# Nested loops

```
/* average number of characters per line */
char ch;
int numchars, totalchars, numlines;
float avg;
totalchars = numlines = 0;
do {
 numchars = 0;
 ch = getchar();
 while (ch != '\n') {
 numchars++;
 ch = getchar();
 }
 totalchars += numchars;
 if (numchars > 0)
 numlines++;
} while (numchars > 0);
avg = (float) totalchars/numlines;
printf("Total lines: %d, Average: %.2f\n", numlines, avg);
```

# The break statement

- The break statement is used to exit from a while, for, do-while or switch structure
- It can only be used inside the body of a for, while, do – while, or switch statement
- The break statement is written simply as  
break;  
without any embedded expressions or statements.
- Program execution continues with the first statement after the structure

# Use of break

```
scanf("%f", &x);
while (x <= 100) {
 if (x < 0) {
 printf("Error – negative value for x");
 break;
 }
 /* process the nonnegative value of x */

 scanf("%f", &x);
}
```

use break inside a while loop  
if value of x is negative exit from  
loop



# break inside nested loops

- If a break statement is used inside nested while, do – while, for or switch statements, it will cause a transfer of control out of inner enclosing structure

```
for (count = 0; count <= n; ++count)
{

 while (c = getchar() != '\n')
 {
 if (c = '*') break;

 }
}
```

# The continue statement

- Skips the remaining statements in the body of a while, for or do-while structure
  - Proceeds with the next iteration of the loop
- The continue statement can be included within a while, a do – while or a for statement
- It is written simply as  
continue;  
without any embedded statements or expressions
- continue inside while and do-while
  - Loop-continuation test is evaluated immediately after the continue statement is executed

# The continue statement

- continue inside for
  - increment expression is executed, then the loop-continuation test is evaluated

```
do {
 scanf("%f", &x);
 if (x < 0) {
 printf("ERROR – NEGATIVE VALUE FOR X");
 continue;
 };
 /* process the nonnegative value of x */

} while (x <= 100);
```

# The goto statement

- The `goto` statement is used to alter the normal sequence of program execution by transferring control to some other part of the current function
- The `goto` statement is written as  
`goto label;`  
where *label* is an identifier that is used to label the target statement to which control will be transferred
- Control may be transferred to any other statement within the current function

# The goto statement

- The target statement will appear as  
*label: statement*
- Each label statement within the current function must have a unique name
- The use of **goto** should be avoided
- However, **goto** can be helpful when you want to exit from a doubly nested loop
  - This can be done with two **if-break** statements, though this would be awkward

# Example

```
scanf("%d", &x);
while (x <= 100) {
 ...
 for (i = x - 4; i < x + 4; i++) {
 ...
 if ((i + x) % 10 == 0)
 goto errorcheck;
 ...
 }
 ...
}
...
errorcheck:
 printf("ERROR");
...

```