A Look Back at Arithmetic Operators: the Increment and Decrement

Increment (++) and Decrement (--)

- Both of these are unary operators; they operate on a single operand.
- The increment operator causes its operand to be increased by 1.
  – Example: a++, ++count
- The decrement operator causes its operand to be decreased by 1.
  – Example: i--, --distance

- **Operator written before the operand (++i, --i))**
  - **Called pre-increment operator.**
  - **Operator will be altered in value *before* it is utilized for its intended purpose in the program.**
- **Operator written after the operand (i++, i--)**
  - **Called post-increment operator.**
  - **Operator will be altered in value *after* it is utilized for its intended purpose in the program.**

Examples

Initial values :: a = 10; b = 20;

x = 50 + ++a;	**a = 11, x = 61**
x = 50 + a++;	**x = 60, a = 11**
x = a++ + --b;	**b = 19, x = 29, a = 11**
x = a++ – ++a;	**Undefined value (implementation dependent)**

Called side effects:: while calculating some values, something else get changed.

Control Structures that Allow Repetition

Types of Repeated Execution

- Loop
  - Group of instructions that are executed 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”.

Counter-controlled Repetition

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

Examples

```
int counter =1;                    // initialization

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

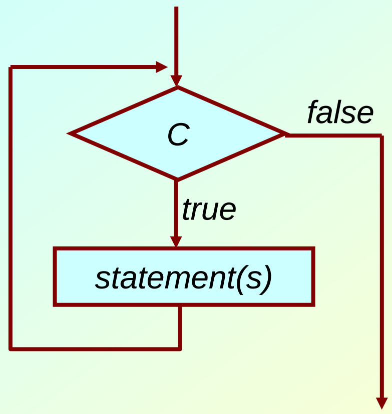
```
int counter;

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

while Statement

- The “while” statement is used to carry out looping operations, in which a group of statements is executed repeatedly, as long as some condition remains satisfied.

```
while (condition)
        statement_to_repeat;
```

```
while (condition) {
        statement_1;
            ...
        statement_N;
}
```



Single-entry / single-exit structure

while :: Examples

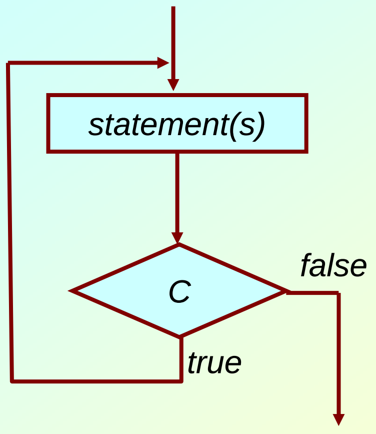
```
int  digit = 0;

while  (digit <= 9)
  printf (“%d \n”, digit++);
```

```
int  weight;

while ( weight > 65 ) {
     printf ("Go, exercise, ");
     printf ("then come back. \n");
     printf ("Enter your weight: ");
     scanf ("%d", &weight);
     }
```

do-while Statement

- Similar to “while”, with the difference that the check for continuation is made at the *end* of each pass.
  - In “while”, the check is made at the *beginning*.
- Loop body is executed at least once.

```
do {
             statement-1
             statement-2

             statement-n
    }   while ( condition );
```



Single-entry / single-exit structure

do-while :: Examples

```
int  digit = 0;

do
   printf (“%d \n”, digit++);
while (digit <= 9);
```

```
int  weight;

do {
    printf ("Go, exercise, ");
    printf ("then come back. \n");
    printf ("Enter your weight: ");
    scanf ("%d", &weight);
    }  while ( weight > 65 ) ;
```

for Statement

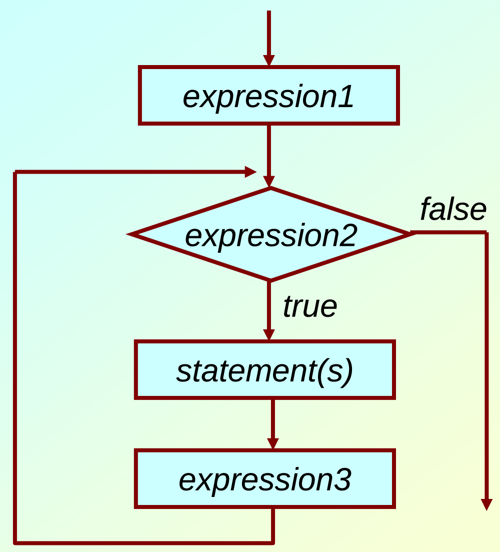
- The "for" statement is the most commonly used looping structure in C.
- General syntax:

```
for (expression1; expression2; expression3)
   statement-to-repeat;
```

```
for (expression1; expression2; expression3)  {
   statement_1;

   statement_N;
}
```

- How it works?
  - "expression1" is used to *initialize* some variable (called *index*) that controls the looping action.
  - "expression2" represents a *condition* that must be true for the loop to continue.
  - "expression3" is used to *alter* the value of the *index* initially assigned by "expression1".

```
int  digit;

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



Single-entry / single-exit structure

for :: Examples

```
int  fact = 1, i;

for  (i=1; i<=10; i++)
     fact = fact * i;
```

```
int  sum = 0, N, count;

scanf (“%d”, &N);

for (i=1; i<=N, i++)
     sum = sum + i * i;

printf (“%d \n”, sum);
```

- The comma operator
  - We can give several statements separated by commas in place of “expression1”, “expression2”, and “expression3”.

```
for  (fact=1, i=1; i<=10; i++)
   fact = fact * i;
```

```
for (sum=0, i=1; i<=N, i++)
   sum = sum + i * i;
```

for :: Some Observations

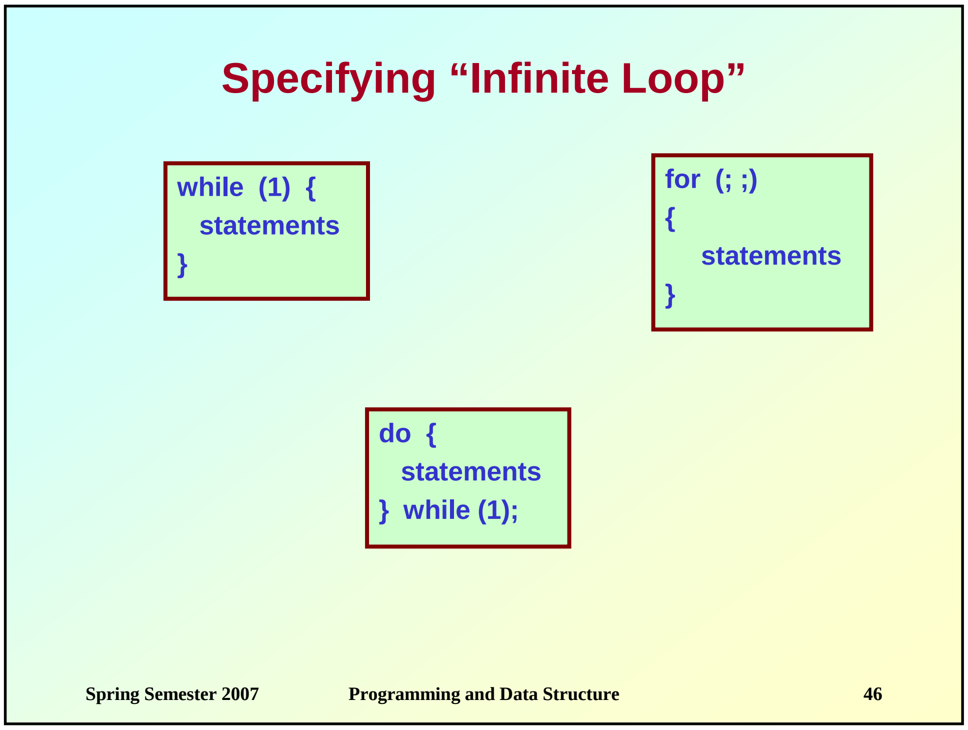
- Arithmetic expressions
  - Initialization, loop-continuation, and increment can contain arithmetic expressions.

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

- "Increment" may be negative (decrement)

```
for  (digit=9; digit>=0; digit--)
```

- If loop continuation condition initially *false:*
  - Body of *for* structure not performed.
  - Control proceeds with statement after *for* structure.

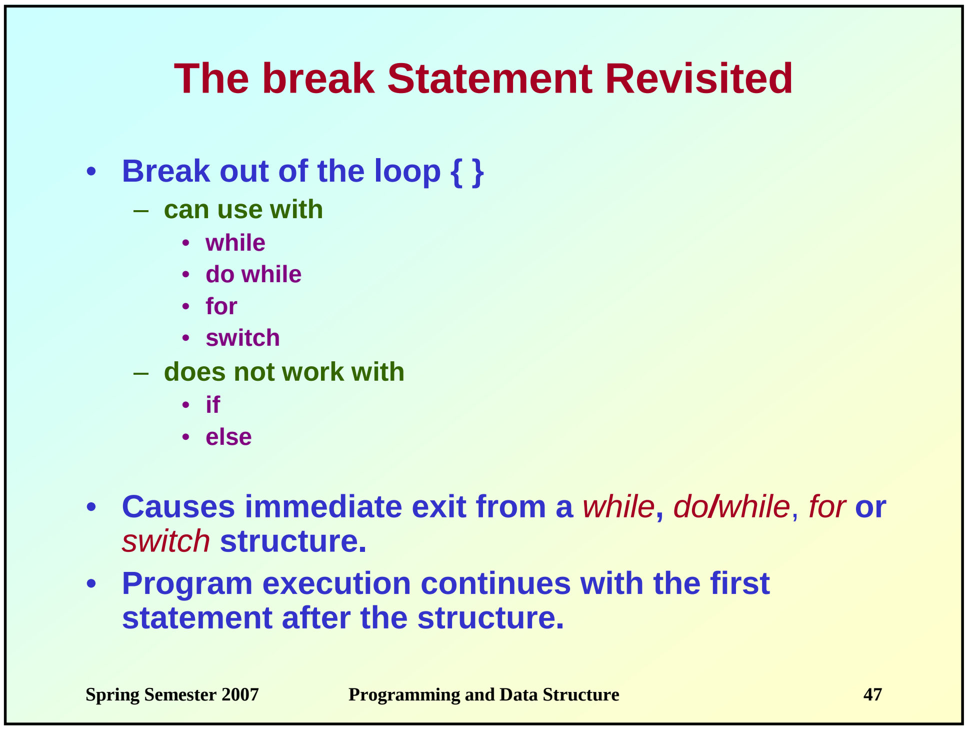


Specifying "Infinite Loop"

```
while  (1)  {
   statements
}
```

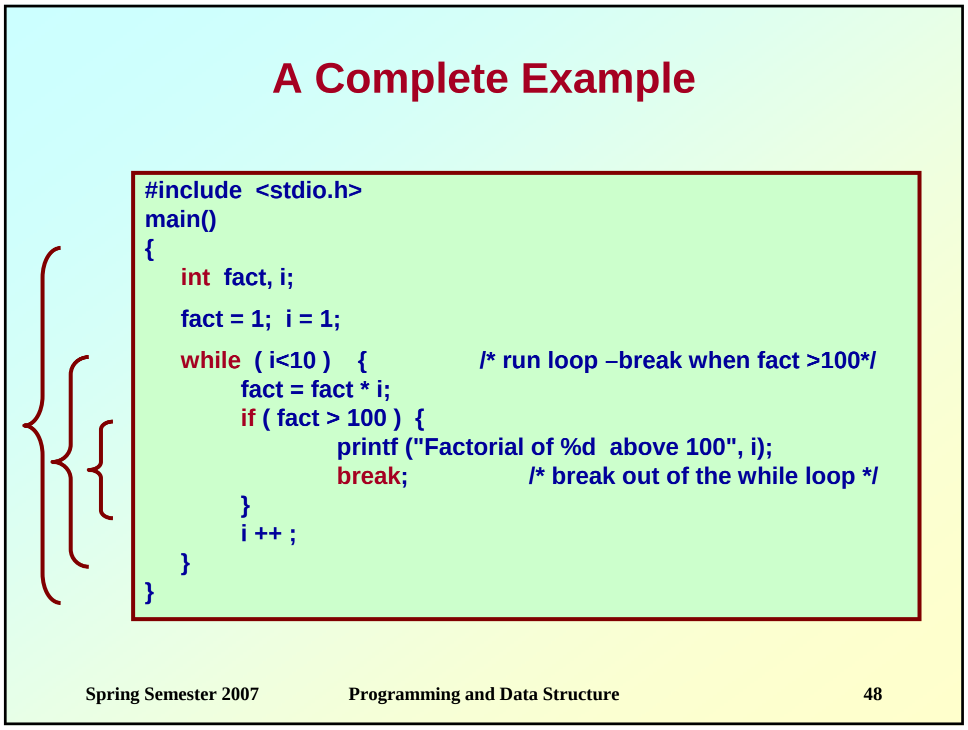
```
for  (; ;)
{
     statements
}
```

```
do  {
   statements
}  while (1);
```



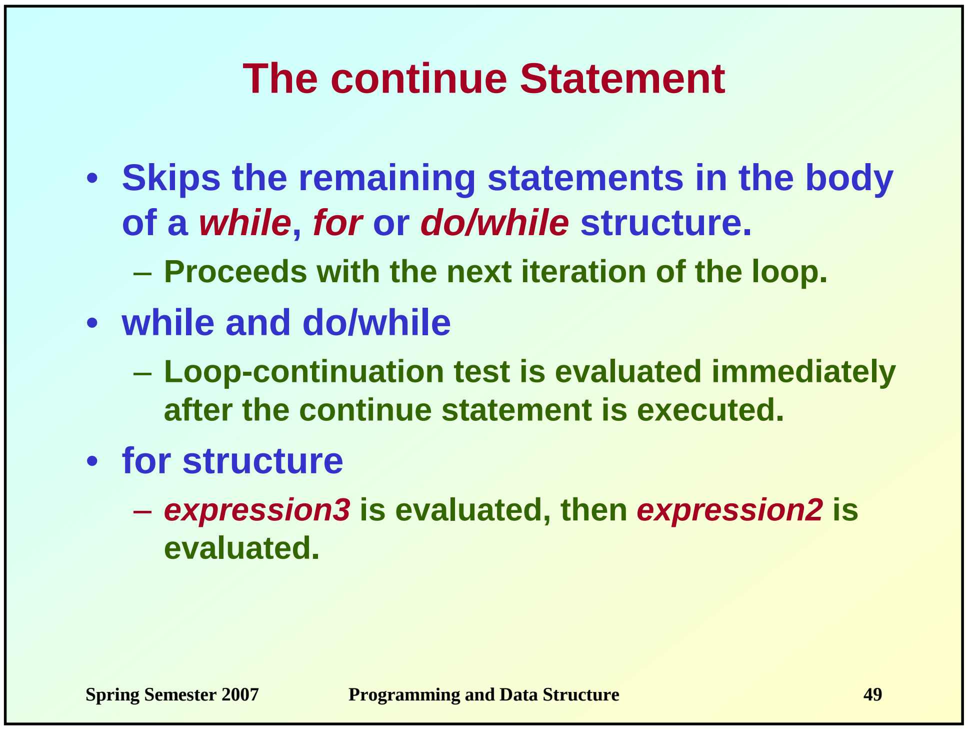
The break Statement Revisited

- Break out of the loop { }
  - can use with
    - while
    - do while
    - for
    - switch
  - does not work with
    - if
    - else
- Causes immediate exit from a *while*, *do/while*, *for* or *switch* structure.
- Program execution continues with the first statement after the structure.



A Complete Example

```
#include  <stdio.h>
main()
{
    int  fact, i;

    fact = 1;  i = 1;

    while  ( i<10 )    {              /* run loop –break when fact >100*/
           fact = fact * i;
           if ( fact > 100 )  {
                      printf ("Factorial of %d  above 100", i);
                      break;              /* break out of the while loop */
           }
           i ++ ;
    }
}
```



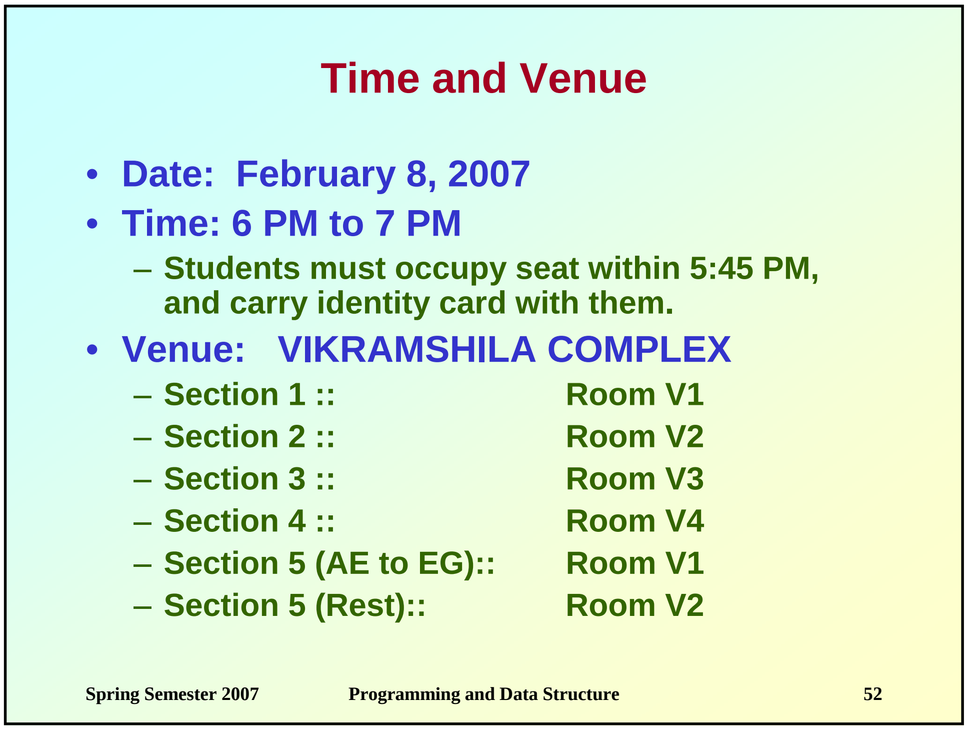
The continue Statement

- Skips the remaining statements in the body of a *while*, *for* or *do/while* structure.
  - Proceeds with the next iteration of the loop.
- while and do/while
  - Loop-continuation test is evaluated immediately after the continue statement is executed.
- for structure
  - *expression3* is evaluated, then *expression2* is evaluated.

An Example with “break” & “continue”

```
fact = 1; i = 1;            /* a program to calculate 10 !
while  (1)  {
    fact = fact * i;
    i ++ ;
    if  ( i<10 )
          continue;         /* not done yet ! Go to loop and
                                      perform next iteration*/
    break;
}
```

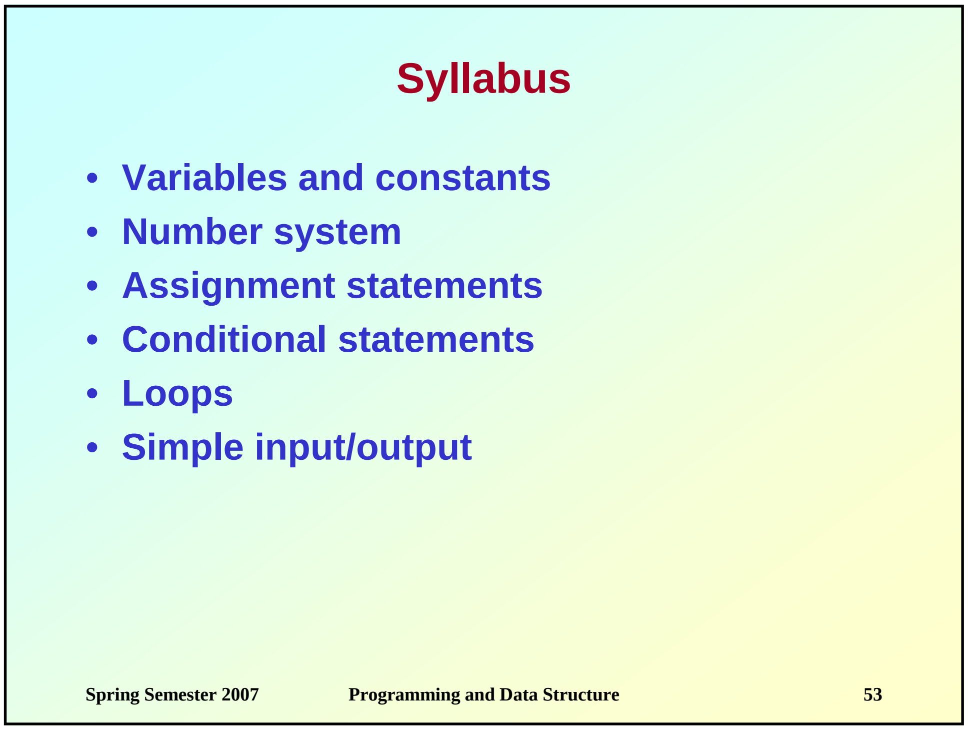
ANNOUNCEMENT REGARDING CLASS TEST 1



Time and Venue

- Date: February 8, 2007
- Time: 6 PM to 7 PM
  - Students must occupy seat within 5:45 PM, and carry identity card with them.
- Venue: VIKRAMSHILA COMPLEX

– Section 1 ::	Room V1
– Section 2 ::	Room V2
– Section 3 ::	Room V3
– Section 4 ::	Room V4
– Section 5 (AE to EG)::	Room V1
– Section 5 (Rest)::	Room V2



Syllabus

- Variables and constants
- Number system
- Assignment statements
- Conditional statements
- Loops
- Simple input/output