Program Control 2

Essentials of Counter-Controlled Repetition

- 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**

Example 3A

```
/* 3A
Counter-controlled repetition */
#include <stdio.h>
#include <conio.h>
int main()
{
  int counter = 1;            /* initialization */

  while ( counter <= 10 ) {   /* repetition condition */
    printf  ( "%d\n", counter );
    ++counter;               /* increment */
  }
  getch();
  return 0;
}
```

Output Result

The for Repetition Structure

- Format when using **`for`** loops

  `for` (*initialization*; *loopContinuationTest*; *increment*)
  statement
- Example:

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

  - Prints the integers from one to ten

No semicolon (**;**) after last expression

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

Rewrite in While-loop

The `for` Repetition Structure

- For loops can usually be rewritten as while loops:

  initialization;
  `while` (*loopContinuationTest*) **{**
    statement;
    increment;
  }
- Initialization and increment
  - Can be comma-separated lists
  - Example:

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

The `for` Structure: Notes and Observations

- 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

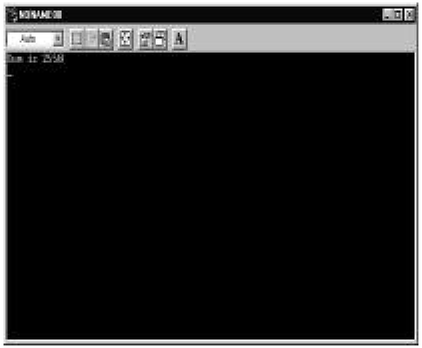
```
for ( j = 2; j <= 80; j += 5 )
```

- 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
    - Control proceeds with the next statement after the **`for`** structure
  - Control variable
    - Often printed or used inside for body, but not necessary

Example 3B

```
■ /* 3B
■ Summation with for */
■ #include <stdio.h>
■ #include <conio.h>
■ int main()
■ {
■   int sum = 0, number;

■   for ( number = 2; number <= 100; number += 2 )
■     sum += number;

■   printf( "Sum is %d\n", sum );
■   getch();
■   return 0;
■ }
```

Output Result



```
Sum is 2550
_
```

The `switch` Multiple-Selection Structure

- **`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

Example 3C

```
■  /* 3C
■  Counting letter grades */
■  #include<stdio.h>
■  #include<conio.h>
■  int main()
■  {
■     int grade;
■     int aCount = 0, bCount = 0, cCount = 0,
■        dCount = 0, fCount = 0;

■     printf(  "Enter the letter grades.\n"  );
■     printf(  "Enter the EOF character (Ctrl-Z) 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;
```

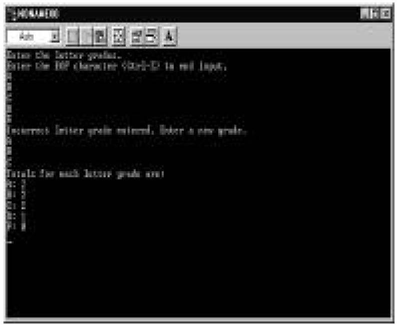
Example 3C (cont')

```
■  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;

■           case '\n': case' ':  /* ignore these in input */
■        break;
```

Example 3C (cont')

```
default:     /* catch all other characters */
   printf( "Incorrect letter grade entered." );
   printf( " Enter a new grade. \n" );
   break;
  }
 }

printf( "\nTotals for each letter grade are:\n" );
printf( "A: %d\n", aCount );
printf( "B: %d\n", bCount );
printf( "C: %d\n", cCount );
printf( "D: %d\n", dCount );
printf( "F: %d\n", fCount );
getch ();
 return 0;
}
```

Output Result



```
Enter the letter grades.
Enter the EOF character (Ctrl-Z) to end input.
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
Incorrect letter grade entered. Enter a new grade.
[illegible]
[illegible]
[illegible]

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

The `do`/`while` Repetition Structure

- The **`do`**/**`while`** repetition structure
  - Similar to the **`while`** structure
  - Condition for repetition tested after the body of the loop is performed
    - All actions are performed at least once
  - Format:

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

- Example (letting counter = 1):

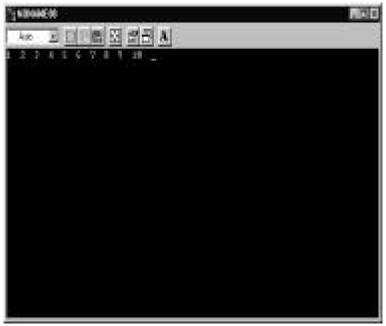
```
do {
   printf( "%d  ", counter );
} while (++counter <= 10);
```

  - Prints the integers from **1** to **10**

Example 3D

```
/* 3D
Using the do/while repetition structure */
#include <stdio.h>
#include <conio.h>
int main()
{
  int counter = 1;

  do {
    printf( "%d  ", counter );
  } while ( ++counter <= 10 );
  getch();
  return 0;
}
```

Output Result



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

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

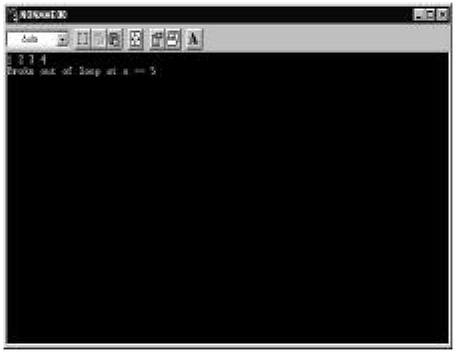
The break and continue Statements

- **`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

Example 3E

```
■   /* 3E
■   Using the break statement in a for structure */
■   #include <stdio.h>
■   #include <conio.h>
■   int main()
■   {
■     int x;

■     for ( x = 1; x <= 10; x++ ) {

■       if ( x == 5 )
■         break;    /* break loop only if x == 5 */

■       printf( "%d ", x );
■     }

■     printf( "\nBroke out of loop at x == %d\n", x );
■     getch();
■     return 0;
■   }
```

Output Result

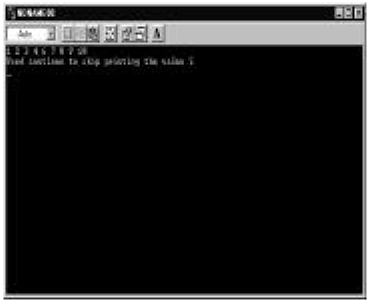


```
1 2 3 4
Broke out of loop at x == 5
```

Example 3F

```
/* 3F
Using the continue statement in a for structure */
#include <stdio.h>
#include <conio.h>
int main()
{
   int x;

   for ( x = 1; x <= 10; x++ ) {

      if ( x == 5 )
         continue;  /* skip remaining code in loop only
                        if x == 5 */

      printf( "%d ", x );
   }

   printf( "\nUsedcontinue to skip printing the value 5\n" );
   getch();
   return 0;
}
```

Output Result



```
1 2 3 4 6 7 8 9 10
Used continue to skip printing the value 5
_
```

Logical Operators

- Precedence:
  - logical operators <relational operators < arithmetic operators

Negation: *! Expr*	■Reverses the truth/falsity of its condition ■Unary operator, has one operand ◆! false = true
And: *Expr1 && Expr2*	■Returns true if both conditions are true ◆true && false = false ◆true && true = true
Or: *Expr1 || Expr2*	■Returns true if both conditions are true ◆true || false = true ◆false || false = false

Equality (==) vs. Assignment (=)

- 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

Equality (==) vs. Assignment (=)

- 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

Equality (==) vs. Assignment (=)

- lvalues
  - Expressions that can appear on the left side of an equation
  - Their values can be changed, such as variable names
    - **`x = 4;`**
- rvalues
  - Expressions that can only appear on the right side of an equation
  - Constants, such as numbers
    - Cannot write **`4 = x;`**
    - Must write **`x = 4;`**
  - lvalues can be used as rvalues, but not vice versa
    - **`y = x;`**