Repetitive Structure

Chapter 04

CMPE-112 *Programming Fundamentals*

Lecture Plan

General Idea (I)

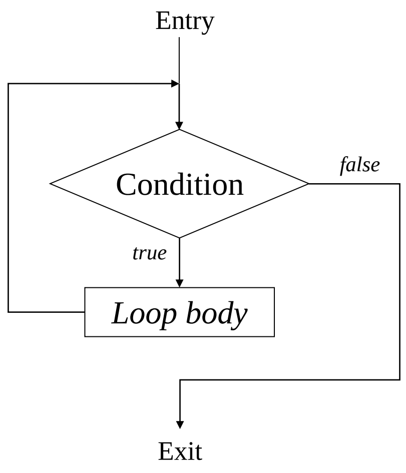
- The *repetitive structure* allows a sequence of program statements to be executed several times even though those statements appear only once in the program
- It consists of
  - an entry point that may include initialization of certain variables
  - a loop continuation condition, which tests once for each execution of the loop body
  - a loop body may consist of statements that are normally executed several times
  - an exit point is the first statement following the loop body
- With *pre-test loops* the continuation condition is tested before the loop body
- With *post-test loops* - after the loop body

3

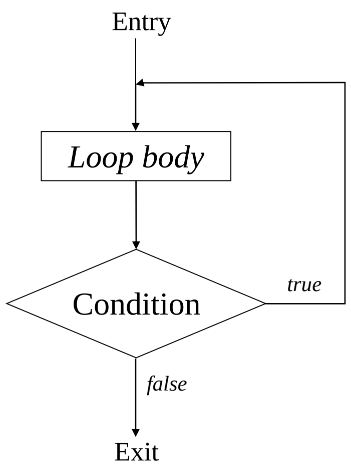
3

General Idea (II)

- Pre-test loop



- Post-test loop



4

4

while Loop

- The *while* loop is a pre-test loop, whose general form is

```
loop initialization

while ( expression )
   statement_1

statement_2
```

Here the *expression* is evaluated. If it is a nonzero value (*true*), then the *statement_1* is executed. Then the *expression* is evaluated again, and if its result is *true*, the *statement_1* is executed again. The process continues until the result of the *expression* becomes zero (*false*). After that the program continues with executing the *statement_2* and so on.

NOTE: In the *while* loop the *statement_1* may not be executed at all

5

5

Sample Program (I)

```
/* The program enters, displays    */
/*     and counts characters       */
#include  <stdio.h>

int main()
{
  int ch, count;

  count = 0;
  ch = getchar();

  while (ch != EOF) {
    putchar(ch);
    count++;
    ch=getchar();
  }

  printf("\nEntered %d characters\n", count);

  return 0;
}
```

ctrl-d for UNIX
ctrl-z for DOS

```
/* The program enters, displays    */
/*     and counts characters       */
#include  <stdio.h>

int main()
{
  int ch, count;

  count = 0;

  while ((ch = getchar()) != EOF) {
    putchar(ch);
    count++;
  }

  printf("\nEntered %d characters\n", count);

  return 0;
}
```

6

6

Infinite Loop

- If the continuation condition is **always** *true*, the loop can never terminate - it is an *infinite loop*
- For instance, a programmer forgets to change the value of a variable within continuation condition

```
int i = 0, n, sum = 0;

while (i < 25) {
   scanf("%d", &n);
   sum += n;
}

printf("This statement is never executed ! ");
```

```
while ( 1 )
   statement_1
```

7

do-while Loop

- The *do-while* loop is a post-test loop, whose general form is

```
loop initialization

do
   statement_1
while ( expression );

statement_2
```

Here the *statement_1* is executed and then the *expression* is evaluated. If it is a nonzero value (*true*), then the *statement_1* is executed again. The process continues until the result of the *expression* becomes zero (*false*). After that the program continues with executing the *statement_2* and so on.

NOTE: In the *do-while* loop the *statement_1* is executed at least once

8

Sample Program (II)

```
/* The program enters a number and computes the sum of its digits */
#include  <stdio.h>

int main()
{
  int number, sum = 0;

  printf(“\nEnter a number = ”);
  scanf(“%d”, &number);

  do {
    sum += number % 10;
    number /= 10;
  }
  while (number > 0);

  printf(“\nThe sum of the digits is %d\n”, sum);

  return 0;
}
```

for Loop (I)

- The *for* loop is a pre-test loop, whose general form is

  for (*expression_1 ; expression_2 ; expression_3*)
      statement_1

  statement_2

Here the *expression_1* is used to perform loop initialization. *Expression_2* is the loop continuation condition. First, *statement_1* is executed and then *expression_3* known as re-initialization expression, is evaluated before the next iteration begins. *Statement_2* is the exit point for the loop

for Loop (II)

A *for* loop is equivalent to the following *while* loop:

```
expression_1;

while ( expression_2 ) {
   statement_1;
   expression_3;
}
```

```
int i, sum = 0;

for (i = 1; i <= n; i++)
   sum += i;

printf("\n%d", sum);
```

```
int i, sum = 0;

i = 1;
while (i <= n) {
   sum += i;
   i++;
}

printf("\n%d", sum);
```

for Loop (III)

- All three expressions within parentheses is the *for* loop are optional and may be omitted. However, the semicolons separating them are to be provided. For example,

```
int i, sum = 0;

i = 1;
for ( ; i <= n ; ) {
   sum += i;
   i++;
}

printf("\n%d", sum);
```

- The *for* loop can be used to set up an infinite loop:

```
for ( ; ; )
   statement
```

Nested Loops

- Loops may be nested, i.e. one loop may contain other loops within its body
- When nesting one loop within another, the inner loop must be entirely contained within the body of the outer loop
- Each loop must have its own unique loop continuation expression
- Nested loops may have the same exit point
- The inner loop is indented with respect to the outer loop for better readability

Sample Program (III)

```
/* The values a, b and c, such that a<b<c, form a Pythagorean triplet */
/*    if a^2 + b^2 = c^2. This program finds all triplets for a, b <=25  */

#include <stdio.h>
#include <math.h>

#define  LIMIT  25

int main()
{
   int a, b, c, c_sqr;

   for (a = 1; a <= LIMIT; a++)
      for (b = a+1; b <= LIMIT; b++) {
         c_sqr = a * a + b * b;
         c = sqrt(c_sqr);          /* This is to truncate fraction */
         if (c * c = = c_sqr)
             printf("\nAnother triplet is %4d %4d %4d", a, b, c);
      }

   return 0;
}
```

Loop Interruption (I)

- On a special occasion loop can be interrupted within the loop body. This causes all statements following the interruption point, to be skipped and the control given either to the exit point or back to the beginning of the loop body
- Accordingly, C provides two statements for implementing loop interruptions
  - *break* Statement
  - *continue* Statement
- When a *break* statement is encountered within a loop body, the execution of the loop body is interrupted, and the program control transfers to the exit point of the loop
- Within a nested loop, *break* statement results in interruption of the innermost loop whose body contains the *break* statement

Sample Program (IV)

```
/* This program computes the sum of square roots for             */
/*     non-negative values. Zero value is to finish calculations  */

#include <stdio.h>
#include <math.h>

int main()
{
  float a, sum = 0;

  printf(“\nEnter numbers: ”);
  do {
    scanf(“%f”, &a);
    if (a < 0)  break;
    sum += sqrt(a);
  }
  while (a != 0);

  if (a >= 0)   printf(“\nThe sum is %.3f\n”, sum);
  else          printf(“\nError: a negative number is not allowed.\n”);

  return 0;
}
```

Loop Interruption (II)

- The *continue* statement does not terminate the loop; it only interrupts a particular iteration
- When a *continue* statement is encountered within a loop body of a *while* or *do-while* loop, all the remaining statements in the loop body are skipped and the loop continuation condition is evaluated next
- Within the *for* loop, any statements in the loop body are skipped, and the re-initialization expression (the third one) is evaluated next
- Then the execution of the repetition continues as normal

17

17

Sample Program (V)

```
/* This program computes the sum of all integer values */
/*    from 1 to n excluded those divisible by 5          */

#include <stdio.h>

int main()
{
  int value, number, sum = 0;

  printf(“\nEnter the limit: ”);
  scanf(“%d”, &number);

  for (value = 1; value <= number; value++) {
    if (value % 5 = = 0)  continue;   /* Skip a value divisible by 5 */
    sum += value;
  }

  printf(“\nThe sum is %1d\n”, sum);

  return 0;
}
```

18

18

Loop Interruption (III)

- The *continue* statements are used mainly to avoid excessive nesting within a loop
- However, it is possible to replace a *continue* statement by an appropriate *if* statement, and thus make the program easy to understand

```
for (i = 1; i <= n; i++) {          for (i = 1; i <= n; i++)
  if (i % 5 == 0)  continue;          if (i % 5 != 0)
  sum += i;                              sum += i;
}
```

Null Statement

- C permits a statement consisting of a semicolon only. It is known as the **null** statement and its general form is

  ;
- Execution of the null statement has no affect, but it is necessary in case the syntax requires a statement, e.g.
- The following statement just counts the number of characters in the input, and the null statement is the only one in the loop body

```
for (count = 0; getchar() != EOF; count++)
      ;
```

Comma Operator

- The comma operator (**,**) is used to combine two related expressions into one, making programs more compact
- The compound expression so formed is evaluated from left to right, and the type and the value of the result are those of the right operand
- The comma operator has the lowest precedence of any other operator
- Example: the following code interchanges the values of *x* and *y*

```
t = x;
x = y;      ⇒      t = x,  x = y,  y = t;
y = t;
```

21

21

Sample Program (VI)

```
/* This program enters two integer values and prints      */
/*    all values between them in the descending order      */

#include <stdio.h>

int main()
{
   int v1, v2, tmp, i;

   puts("\nEnter two integer : ");
   scanf("%d,%d", &v1, &v2);

   if (v1 < v2)     /* Interchange the values of v1 and v2 */
      tmp = v1, v1 = v2, v2 = tmp;

   for (i = v1; i >= v2; i--)  /* Print out the values from the range */
      printf("%4d", i);

   return 0;
}
```

22

22