Flow of control: The do-while, break, continue and goto statements.

The do-while statement:

The `do-while` statement in C is an exit-controlled loop that guarantees the body of the loop is executed at least once before the `expr` is checked. The syntax for a `do-while` loop in C is as follows:

```
do {
    // Code to be executed repeatedly
    // Update statement (to make expr eventually false)
} while (expr); // Note the semicolon here
// next statement
```

The `expr` is a logical expression that is evaluated after each iteration. If it is true (non-zero), the loop repeats. If it is false (zero), the loop terminates, and program execution continues with the next statement.

Note the difference between `while` statement and `do-while` statement. The former is an entry-controlled statement, and the latter is exit-controlled. Thus, the body of the `while` loop may not be executed at all. On the other hand, the body of a `do-while` loop executed at least once.

Consider the program given in Listing 27 of Lecture 13 in which we have calculated the value of $\sin(x)$. The same program is rewritten using `do-while` loop as shown below:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
int main()
{
 int i,N;    // N max no. of terms
 double xold,x,eps,t,s;    // eps tolerance
 printf("Enter N & eps:");
 scanf("%d%lf",&N,&eps);
 printf("Enter x:");
 scanf("%lf",&xold);    // xold holds the original x
 x=xold;                // x will hold the value in (-2*pi,2*pi)
 while(x>=2*M_PI)      // if x >= 2*pi
 {
  x -=2*M_PI;
 }
```

```
 while(x<=-2*M_PI)     // if x <= -2*pi
 {
  x +=2*M_PI;
 }
 t=x;      // t first term, now x in (-2*pi,2*pi)
 s=0;     // sum initialized to zero
 i=0;     // i for i-th term
 do
 {
  s +=t;
  i++;
  t *=-(x*x)/((2*i)*(2*i+1.0));
 } while(fabs(t)>eps && i<N);
 if(fabs(t)<=eps)
 {
  printf("Value of sinx at x=%0.4e with %d terms is %.6e\n",xold,i,s);
  printf("Value of sinx with C-math library is %.6e\n",sin(x));
 }
     else
 {
  printf("Desired accuracy not achieved with %d terms\n",N);
 }
 return 0;
}
```

Listing 29: C program "dsine.c"

The outputs corresponding to the input are the same except for the first input. For the first input, `do-while` loop is executed once, which gives rise to a different value. We only show the first three input-output pairs.

```
Enter x:5.e-6
Value of sinx at x=5.0000e-06 with 1 terms is 5.000000e-06
Value of sinx with C-math library is 5.000000e-06
Enter N & eps:100 1.e-6
Enter x:12
Value of sinx at x=1.2000e+01 with 12 terms is -5.365734e-01
Value of sinx with C-math library is -5.365729e-01
Enter N & eps:100 1.e-7
Enter x:12
```

```
Value of sinx at x=1.2000e+01 with 13 terms is -5.365729e-01
Value of sinx with C-math library is -5.365729e-0
```

Similarly, we rewrite the program (Listing 25, Lecture 13) in which a real number is read. If it is nonnegative, its square root is calculated and printed. The program asks for new input in case of invalid input. Thus, it will continue until we enter a valid input. The code below implements this.

```
#include<stdio.h>
#include<math.h>
int main()
{
double x,s;
do
{
 printf("Enter a nonnegative number:");
 scanf("%lf",&x); // double x is read using %lf
}while (x<0); // end of do-while block
s=sqrt(x);
printf("Square root of %.2f is %.2f\n",x,s);
return 0;
}
```

Listing 30: C program "dsine.c"

The following is expected input and output. Note that here we read the input inside `do-while` loop. Thus, we have one less "scanf()" statement compared to `while` loop.

```
Enter a nonnegative number:-5
Enter a nonnegative number:-3
Enter a nonnegative number:5
Square root of 5.00 is 2.24
```

It is important to keep in mind that the body of `do-while` loop is executed at least once. Hence, we must be cautious while implementing `do-while` loop. For this reason, the use of the `do-while` loop is not as widespread as that of the `while` loop and the `for` loop.

Caution about operators in conditional expressions: Consider the following code, where we add `x=2.9` repeatedly until the sum becomes `29.0`. We expect the loop to execute 10 times when the sum will be `29.0`, and the program will stop. We print the number of iterations inside the loop.

```
#include<stdio.h>
int main()
{
 int i=0;
 double x=2.9,s=0;
 while(s!=29.0)
 {
  i++;
  s +=x;
  printf("%d\n",i);
 }
return 0;
}
```

Listing 31: C program "wrng1.c"

The following is a partial output of the code:

```
1
2
3
4
5
6
7
8
9
10
11
12
.
.
.
```

Clearly, it becomes an infinite loop. The reason for this is that `2.9` cannot be expressed exactly in a finite-bit representation. Thus, adding `x=2.9` ten times does not yield `29.0` and this results in an infinite loop. To break the infinite loop, suppose that we use a relational operator as shown in the following code:

```
#include<stdio.h>
int main()
{
 int i=0;
 double x=2.9,s=0;
 while(s<29.0)
 {
  i++;
  s +=x;
  printf("%d\n",i);
 }
return 0;
}
```

Listing 32: C program "wrng2.c"

The following is the output of the code:

```
1
2
3
4
5
6
7
8
9
10
11
```

This time the loop is not infinite, but it added 11 times instead of 10. The reason is that the approximate value of `2.9` stored in `x` is a little less than `2.9`. Summing 10 times results in a value slightly less than `29.0` and thus the loop runs for 11 times. The following code addresses the problems encountered above. We use a tolerance in the code and `fabs(s-29.0)>eps` implies $|s - 29.0| >$ eps.

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
int main()
{
```

```
 int i=0;
 double x=2.9,s=0,eps=1.e-3;
 while(fabs(s-29.0)>eps)
 {
  i++;
  s +=x;
  printf("%d\n",i);
 }
return 0;
}
```

Listing 33: C program "crsum.c"

The following now shows the correct output of the code:

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

The break and continue statements:

Two special statements, "`break;`" and "`continue;`" interrupt the normal flow of control. We have already encountered `break` expression in the context of `switch` statement. Upon encountering `break` statement, the execution transfers to the statement after the `switch` block. A `break` statement can also appear inside a loop, where it causes an exit from the innermost enclosing loop. For example, consider the following code, which prints the prime numbers between 25 and 50. An positive integer `n` is a prime if it has no divisor from 2 to $n/2 - 1$.

```
#include<stdio.h>
int main()
{
 int a=25,b=50,i,j;
 printf("The primes between %d and %d are :",a,b);
 for(i=a;i<=b;i++)
 {
```

```
  for(j=2;j<i/2;j++) // test of prime
  {
   if(i%j==0)
   {
    break;             // non-prime, no need to iterate further
   }
  }
  if(j==i/2)             // break transfers here; also j=i/2 if j-loop runs full
  {
   printf("%d ",i); // i is a prime since j-loops run full
  }
 }
 printf("\n");
 return 0;
}
```

Listing 34: C program "cprime.c"

The output is shown below.

```
The primes between 25 and 50 are :29 31 37 41 43 47
```

In the innermost `for` loop (for `j`), we do not iterate once we find a divisor. The `break` transfers the execution to `if` block [`if(j==i/2)`]. We also arrive at the same `if` block if the `for` loop for `j` finishes normally. However, the value of `j` will be less than `j/2` for non-prime and `j/2` for prime.

The `continue` statement can occur only inside `for`, `while` and `do-while` loops. The `continue` statement causes the current iteration of a loop to stop and causes the next iteration of the loop to start immediately. Consider the following illustration using a `for` loop.

```
for(expr1;expr2;expr3)
{
// 1st block of statements may be here
continue;
// 2nd block of statements may be here
}
```

Once the execution encounters `continue` statement, the execution transfers to `expr3`. In this case, the 2nd block of statements (after `continue`) is not executed. For example, consider the following code, which sums the prime numbers between 25 and 50.

```
#include<stdio.h>
int main()
{
 int a=25,b=50,i,j,sum=0;
 for(i=a;i<=b;i++)
 {
  for(j=2;j<i/2;j++) // test of prime
  {
   if(i%j==0)
   {
    break;              // no need to proceed further
   }
  }
  if(j!=i/2)
  {
   continue;
  }
  sum +=i;
 }
 printf("The sum of primes between %d amd %d is %d\n",a,b,sum);
 return 0;
}
```

Listing 35: C program "sprime.c"

The following is the program output.

```
The sum of primes between 25 and 50 is 228
```

In the above code, the `continue` statement is inside the outermost `for` loop. If `j!=i/2` is true, then it is a non-prime and thus should not be included in the `sum`. The code hits `continue` and the next statement executed is `i++` (`expr3` of `i-for` loop).

In the case `while` loop, upon encountering `continue` statement, the execution transfers to `while(expr)`. Make sure that you are not creating an infinite loop due to `continue` statement. Consider the following code segment, where we sum even integers between 1 and 10 using `continue` statement inside a `while` loop.

```
 int i=1,sum=0;
 while(i<=10)
 {
  if(i%2!=0)
```

```
 {
  continue;
 }
 sum +=i;
 i++;
}
printf("Required sum=%d\n",sum);
```

The above code results in an infinite loop. The reason is that with `i=1`, it enters the `while` loop. But `i%2!=0` is true and thus it transfers the execution to `while(i<=0)` with `i=1` again. The same thing is repeated again. The above code is written in a convoluted way to illustrate the occurrence of an infinite loop. If we want to keep the `continue` statement and still want to get the result, we need to modify it as shown below:

```
int i=1,sum=0;
while(i<=10)
{
 if(i++%2!=0)
 {
  continue;
 }
 sum +=(i-1);
}
printf("Required sum=%d\n",sum);
```

Finally, note that `continue` transfers control to the end of the current iteration, whereas `break` terminates the loop.

The goto statement:

The `goto` statement in C provides an unconditional jump from one point in a function to a specific, labelled statement within the same function [note that "main()" is also a function]. Though it can be used to alter the normal flow of a program, its use is generally discouraged in modern programming practice due to the potential for creating unreadable and unmanageable code.

A label is an identifier used to name a specific point in the code, which can then be jumped to using the `goto` statement. A label is followed by a colon (`:`). It can appear before any statement within the same function. Some examples of valid labelled statements are

```
stop: exit(1);
L1: sum +=a;
err1: err2: err3: printf("Error in the code\n"); // multiple label
```

However, the following are not labelled statements since the labels are not identifiers.

```
200: sum +=a;              // 200 is not an identifier
L#2: printf("Hello world\n"); // L#2 is not an identifier
```

The following program shows the use of `goto` statement. It finds the sum of integers from 1 to 10.

```
#include<stdio.h>
int main()
{
 int i=1,sum=0;

L1: sum +=i;
  i++;
  if(i<=10)
  {
   goto L1;
  }
  printf("Required sum is %d\n",sum);
  return 0;
}
```

Listing 36: C program "cgto.c"

Here, the output is "`Required sum is 55`". Of course, we should write the above code using `for` or `while` loop. In general, the `goto` statement should be avoided. However, in some rare cases, a `goto` statement can make the code significantly more efficient. In some other cases, it can simplify the flow of control. For example, suppose that a special value is tested inside the inner loop of nested loops. When this special value is detected, the execution needs to jump out of the nested loops.

Return to Lecture Topics