Flow of control: the while and the for statements

The while statement:

The `while` statement in C is an entry-controlled loop that repeatedly executes a block of code as long as a specified condition remains true. A basic construct of `while` statement is

```
 while (expr) {
    // code to be executed repeatedly
    // update statement (to eventually make the expr false)
}
```

Here, `expr` is evaluated before each iteration. The loop continues as long as `expr` is true (nonzero value) and terminates when `expr` becomes false (zero). An update to variables used in the `expr` is crucial within the loop body to avoid an infinite loop.

Consider the program (Listing 16, Lecture 11) in which a real number is read. If it is nonnegative, its square root is calculated and printed. If it is negative, the program stops with an appropriate message. We now modify the program by insisting that 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;
printf("Enter a nonnegative number:");
scanf("%lf",&x); // double x is read using %lf
while(x<0.0){ // start of while block
printf("You have enterd a negative number\n");
printf("Enter a nonnegative number again:");
scanf("%lf",&x);
} // end of while block
s=sqrt(x);
printf("Square root of %.2f is %.2f\n",x,s);
return 0;
}
```

Listing 25: C program "csqrtm.c"

The following is expected input and output.

```
Enter a nonnegative number:-5
You have entered a negative number
Enter a nonnegative number again:-3
You have entered a negative number
Enter a nonnegative number again:-4
You have entered a negative number
Enter a nonnegative number again:5
Square root of 5.00 is 2.24
```

Note that the first `scanf("lf",&x)` is before the `while` block. Thus, `x<0.0` is evaluated with the first input value of `x`. If `x`≥ 0, then the `while` loop is not executed. On the other hand, if `x`< 0.0, then the execution enters the `while` loop.

It is very important that the `expr` in `while(expr)` becomes false (zero) during execution of the loop. Otherwise, we get an infinite loop. Consider the program below, where we read a negative integer `n` and then print all the even negative integers between `n` and `0`.

```
#include<stdio.h>
int main()
{
 int n;
 printf("Enter n:");
 scanf("%d",&n);
 printf("Negative even integers between %d and 0 are\n",n);
 while(n!=0)
 {
  if(n%2==0)
   {
   printf("%d\n",n);
   n +=2;
   }
 }
 return 0;
}
```

The expected input and output for $n = -6$ are as follows:

```
Enter n:-6
Negative even integers between -6 and 0 are
-6
-4
-2
```

If we input a negative odd integer, then `n!=0` is always true, but the value of `n` remains unchanged. Below is the output for $n = -7$, which results in an infinite loop, and the program keeps running.

```
Enter n:-7
Negative even integers between -7 and 0 are
```

To remedy this, we could increase `n` by one, like the following. Then we don't need the `if` statement inside the `while` block.

```
#include<stdio.h>
int main()
{
 int n;
 printf("Enter n:");
 scanf("%d",&n);
 printf("Negative even integers between %d and 0 are\n",n);
 if(n%2!=0)
 {
  n++;
 }
 while(n!=0)
 {
   printf("%d\n",n);
   n +=2;
 }
 return 0;
}
```

Below is the output for $n = -7$

```
Enter n:-7
Negative even integers between -7 and 0 are
-6
-4
-2
```

The only problem that remains is an infinite loop for positive integers. This can be removed in many ways. One is replacing `while(n!=0)` by `while(n<0)` like the following, which will work for all integer inputs.

```
#include<stdio.h>
int main()
{
 int n;
 printf("Enter n:");
 scanf("%d",&n);
 printf("Negative even integers between %d and 0 are\n",n);
 if(n%2!=0)
 {
  n++;
 }
 while(n<0)
 {
   printf("%d\n",n);
   n +=2;
 }
 return 0;
}
```

Listing 26: C program "cnegn.c"

Finally, let us write a program that calculates the value of $\sin(x)$ using Maclaurin series

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots$$

Since it is an infinite series, we can only sum a finite number of terms. Our strategy is as follows: We continue to add terms as long as their magnitude is greater than some $\epsilon > 0$. However, we add at most N terms. If the magnitude of the N-th term is greater than ϵ, we say that the desired accuracy is not achieved. In the case when the desired accuracy is achieved, we also compare our value with that provided by the C-math library. Before implementing the code, we observe a few things. First, note that each term has the power of x and hence for larger powers, the terms will have a large magnitude. Though the factorial in the denominator ultimately dominates, we need to take many terms to get the desired accuracy. To get a moderate value of x, we use the periodic property of $\sin(x)$, which states that $\sin(x) = \sin(x+2k\pi) = \sin(x-2k\pi)$ for a natural number k. Thus, if the value of x is greater than or equal to 2π, we repeatedly subtract 2π from it until the value lies in $(-2\pi, 2\pi)$. Similarly, if x is less than or equal to -2π, we repeatedly add 2π to it until the value lies in $(-2\pi, 2\pi)$. Second, the factorial of an integer can be very large, and the power function `pow` is time-consuming. To avoid both, we derive a recurrence relation among terms. We write

$$\sin(x) = \sum_{n=1}^{\infty} (-1)^{n-1} \frac{x^{2n-1}}{(2n-1)!} = \sum_{n=1}^{\infty} t_n$$

Hence,

$$\frac{t_{n+1}}{t_n} = -\frac{x^2}{(2n+1)(2n)} \Rightarrow t_{n+1} = t_n\left(-\frac{x^2}{(2n+1)(2n)}\right), \qquad n = 1, 2, 3, \cdots$$

Thus, we assign x to t, which is the first term. To get the next term, we multiplied t by $-x^2/6$, and we may denote it by the same t. This process can be continued.

The following code implements the above.

```
#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
 while(fabs(t)>eps && i<N)         // fabs(x) absolute value of real x
 {
  s +=t;
  i++;
  t *=-(x*x)/((2*i)*(2*i+1.0));       //next terms
 }
 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 27: C program "wsine.c"

Below are some input and output:

```
Enter N & eps:100 1.e-5
Enter x:5.e-6
Value of sinx at x=5.0000e-06 with 0 terms is 0.000000e+00
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-01
Enter N & eps:100 1.e-7
Enter x:-10
Value of sinx at x=-1.0000e+01 with 10 terms is 5.440211e-01
Value of sinx with C-math library is 5.440211e-01
Enter N & eps:10 1.e-7
Enter x:-12
Desired accuracy not achieved with 10 terms
```

The first input-output corresponds to $x = 5 \times 10^{-6}$ and $\epsilon = 10^{-5}$. Since the magnitude of the first term is less than ϵ, the body of the `while` loop is not executed. The next input-output corresponds to $x = 12$ and $\epsilon = 10^{-6}$. Here, the accurate value provided by C library is a little different from the calculated value. However, if we decrease ϵ, then both values agree with each other. Finally, if we take a small value of N, then the desired accuracy is not achieved.

The for statement:

The `for` statement in the C programming language is a control flow statement used to repeat-

edly execute a block of code a specific number of times or until a given condition is met. This is very similar to the `while` statement. The basic syntax of a `for` loop in C is:

```
for (expr1; expr2; expr3) {
    // Code block to be executed
}
// Next statement after the loop here
```

Here, `expr1` is the initialization expression. This expression is executed *only once* at the very beginning of the loop. It is usually used to declare and initialize variables. The second `expr2` is the condition expression, which is evaluated *before* each iteration of the loop. If `expr2` is true (nonzero), the loop body executes. If `expr2` is false (zero), the loop terminates, and control passes to the next statement after the loop. Finally, `expr3` is a loop expression that is executed *after* each iteration of the loop body. It is typically used to update the loop control variables. The block of code is executed in each iteration as long as `expr2` remains true (nonzero). *Note that* `expr1`, `expr2`, *and* `expr3` *are optional.* One or more of them can be omitted. As in the case of `while` loop, it is important that `expr2` in `for` loop becomes false (zero) during the execution. Otherwise, we will have an infinite loop. To break an infinite loop, we may use a statement within the body of the `for` loop.

We have mentioned that `for` statement is similar to `while` statement. In fact, anything we do using `for` can be done using `while` and vice versa. For example, the syntax of `for` can be written using `while` as follows:

```
expr1;
while(expr2) {
    // Code block to be executed
expr3;
}
// Next statement after the loop here
```

Let us rewrite the code given in Listing 27 using `for` instead of `while`.

```
#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);
 x=xold;

 for( ;x>=2*M_PI; )      //when x>=2*pi
 {
    x -=2*M_PI;
 }

 for( ;x<=-2*M_PI; x+=2*M_PI)    // when x<=-2*pi
 {
 ;                               // empty loop body
 }

 for(s=0,i=0,t=x;fabs(t)>eps && i<N; )
 {
  s +=t;
  i++;
  t *=-(x*x)/((2*i)*(2*i+1.0));
 }
 if(fabs(t)<=eps)
 {
  printf("Value of sinx at x=%0.2lf with %d terms is %e\n",xold,i,s);
  printf("Value of sinx with C-math library is %e\n",sin(x));
 }
     else
 {
  printf("Desired accuracy not achieved with %d terms\n",N);
 }
 return 0;
}
```

Listing 28: C program "fsine.c"

We get the same output as in the previous program with `while` loop (see Listing 27). Note that both `expr1` and `expr3` are blank in the first `for` loop. In the second `for` loop, `expr1` is absent and the body of the `for` loop is blank, which is indicated by a semicolon `;`. In the third `for` loop, `expr3` is absent. The third `for` loop can be written in the following equivalent way.

```
for(s=0,i=1,t=x; fabs(t)>eps && i<=N; i++)
 {
  s +=t;
  t *=-(x*x)/((2*i)*(2*i+1.0));
 }
```

This is the typical form of a `for` loop, where `expr3` is used for incrementing or decrementing the control variable.

The comma operator:

Note that the expression `s=0,i=1,t=x` in `expr1` of the last `for` loop shows the working of `comma` operator. The comma operator has the lowest precedence of all the operators in C. It is a binary operator with expressions as operands, and it associates from left to right. In a `comma` expression of the form

`expr1,expr2`

`expr1` is evaluated first, and then `expr2`. The comma expression as a whole has the value and type of its right operand. Consider the following code segment:

```
int a,b;
a=1,b=5;
```

The `comma` expression has value `5` and type `int`. Note that the `comma` appearing in expressions such as `int a,b` or `int a=1,b=5` is not `comma` operator, it is used as a separator. The expression `s=0,i=1,t=x` (of the `for` loop) is equivalent to (due to precedence and associativity) `(((s=0),(i=1)),(t=x))`, that has the value of `x` and type `double`. To see this, consider the following code segment:

```
 int i;
 double s,x=12,t;
 printf("%0.2f\n",(((i=2),(s=0)),t=x)); // 12.00 is printed
```

Finally, we extend the operator precedence and associativity table discussed in Lecture 12.

Category	Operator	Associativity
Postfix operators	`( )` (parenthesis), `++` (postfix increment), `--` (postfix decrement),	left to right
Unary operators	`+` (unary plus), `-` (unary minus), `++` (prefix increment), `--` (prefix decrement), sizeof, (type) (type casting), `!` (logical not)	right to left
Multiplicative operators	`*` (multiplication), `/` (division), `%` (modulus)	left to right
Additive operators	`+` (addition), `-` (subtraction)	left to right
Relational operators	`<` (less than), `>` (greater than), `<=` (less than or equal to), `>=` (greater than or equal to)	left to right
Equality operators	`==` (equal to), `!=` (not equal to)	left to right
Logical and	`&&`	left to right
Logical or	`||`	left to right
Conditional (ternary) operator	`?:`	right to left
Assignment operators	`=`, `+=`, `-=`, `*=`, `/=`, `%=`	right to left
Comma operator	`,`	left to right

Return to Lecture Topics