Introduction to Scientific and Engineering Computation

(BIL 102E)

LECTURE 8

FUNCTIONS

Instead of repeating the same code again and again we can use functions.

Functions allow

1. generic/parametric solutions to problems
2. division of big problems into smaller ones
3. reuse of program segments in different problems

Before getting in to details let us first examine the data types:

More About Data Types

The two main data types (integer and floating point numbers) in C have variations. Depending on the required precision and range the programmer might want to use one of these variations.

The *short* modifier states that the programmer requires the data type to occupy less space in memory (which results in less accuracy or smaller range of values that can be represented by the type). For example,

short int i, j, k;

The *long* modifier states that the programmer requires the data type to occupy more space in memory (which results in higher accuracy or larger range of values that can be represented by the type). For example,

long int bigvalue;

Long and short modifiers are not allowed to be used with float. To have more accurate floating point numbers we use the type **double**. For example,

double x;

Defines a high accuracy floating point number with name x.

Using **long double** it might be possible to obtain even higher accuracies. The number of bytes used for representing variables in different types depends on the machine, the operating system and the compiler used. For a PC working with a Intel Pentium processor, MS Visual C++ 6.0 compiler uses

2 bytes for **short int**,	4 bytes for **float**
4 bytes for **int**,	8 bytes for **double**
4 bytes for **long int**	8 bytes for **long double**

The number of bytes used for a specific type or for a given variable can be learned by using the **sizeof()** function. For example,

```
long int k;
printf("The size of memory occupied by k : %d\n",sizeof(k));
printf("The size of memory occupied by char type : %d\n",
                                                    sizeof(char));
```

Formatting *long* and *short* variables:

To format an expression in a printf or scanf function call, h and l modifiers are used in front of the usual specifiers, to specify shorter or longer formatting, respectively.

For example,

```
short int sx=1;
long int lx=1000000;
printf("sx = %hd,   lx = %ld \n", sx, lx);
```


When working with derivations of integer type, it is possible to state whether the variables represent only positive values or both positive and negative values, by using unsigned and signed modifiers respectively.

For example for a system where 16 bits are used for integers,

```
char ch;  /* x can hold values between –128 (2^7) and 127 (2^7 –1) */
unsigned char uch;  /* uch can hold values between 0 and 255 (2^8 –1) */

int x;  /* x can hold values between -32768 (2^15) and 32767 (2^15 –1) */
signed int y;  /* the same as int y */
unsigned int z;  /* z can hold values between 0 and 65535 (2^16 –1) */

long int lx;  /* lx can hold values between (2^31) and (2^31 –1) */
signed long int ly;  /* the same as long int ly */
unsigned long int lz;  /* lz can hold values between 0 and (2^32 –1) */
```


Functions

We frequently use functions in mathematics.

Examples include

- Trigonometric functions (sin(x), cos(x)),
- Exponential functions (e^x, 10^x)
- Rational functions such as

$$f(x,y)=\frac{x^2+2xy+5x}{y+1}$$

However, a function in C is not restricted with math functions and any piece of code that can be called again and again can be defined as a function. Every function has a set of *arguments* (as input parameters) such as *x* and *y* in the above examples and an *output* (that is the result of the function).

In order for us not to reinvent the wheel, there are many functions ready for use in C.

Some of these functions, such as the **sizeof**() function, come with the core of the language.

However, a great deal of functions are provided by the standard libraries.

For example, we have already seen that **printf(), getc(), getchar(), putc(), putchar()** and **scanf()** functions are defined in *stdio* library. So, in order to be able to use these functions we have to include the header file stdio.h that holds the declarations in our C program.

There are other standard libraries such as *stdlib*,and *math*, which we will be talking about.

Math Functions

Some of the mathematical functions and constants are defined in *math* library.

- **sin**(x), **cos**(x), **tan**(x) → Trigonometric functions
- **asin**(x), **acos**(x), **atan**(x) → Trigonometric functions
- **pow**(x,y), **sqrt**(x) → x^y, Square Root (the same as *pow(x, 0.5)*)
- **exp**(x), **log**(x), **log10**(x) → Exponential, natural log, log in base 10
- **ceil**(x), **round**(x), **floor**(x) → Converting to integer

In these functions, x can be a real (double) valued expression and the output of the function is a real (double) number.

So these are used anywhere in the program where a real expression can be used. For example,

2***sin**(0.5)/**log**(x+4.0) → $\frac{2\,\mathrm{Sin}(0.5)}{\ln(x+4)}$

asin(**cos**(2*x)) → ASin(Cos(2x))

Example

math.h library is used so that we can make use of ***sin***() function.

```
#include <stdio.h>
#include <math.h>
main()
{
  double x;
  const double PI = 3.14159265358979323846;
  for(x=0.0; x<=90.0; x+=15.0)
    printf("Sin(%2.0lf) = %5.3lf\n",x ,sin(x * PI/180.0));
}
```

```
Sin( 0) = 0.000
Sin(15) = 0.259
Sin(30) = 0.500
Sin(45) = 0.707
Sin(60) = 0.866
Sin(75) = 0.966
Sin(90) = 1.000
```

sin() function call is replaced by the result of the function (that is a **double** value).

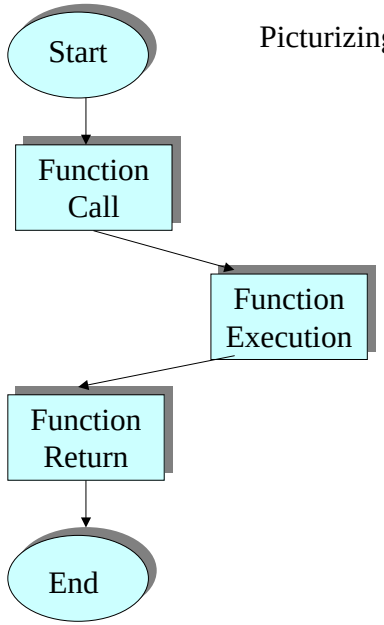
Trigonometric functions use radians. So we have to convert to radians first.

When you *call a function*

When running if a program comes to a function the execution continues from the function. This is named as *calling the function*. Before this, however, all argument expressions of a function is evaluated and the results of these evaluations are passed to the function. The statements in the function are then executed and the result of the function is *returned* to the main program. The main program then acts as if the function statement is replaced by the result of the function. For instance,

```
x=30.0;
y = 1.5 + sin(x * PI/180.0);
```

First, x*PI/180.0 (→0.523...) is evaluated. Then, sin() function is called. The function call statement (ie `sin(x * PI/180.0)`) is replaced by the result of the function (0.5). Last, the statement y=1.5+0.5; is evaluated. Hence, at the end of the second statement the variable y is 2.



Picturizing a function call.

Defining your own functions

Functions defined in standard libraries are generally not enough for writing large programs. In order to solve larger programming problems, you often need to define your own functions and call them from your program. The syntax for defining a function is as follows:

Type_of_result function_name(*type1* arg1, *type2* arg2, ...)
{
[Block of C statements]
}

Here, arg1, arg2, ... are called as the arguments of the function. The parameters of a function are sent via the arguments. We shall see that arguments can also be used to provide additional outputs to the function.

Example

```
#include <stdio.h>
/* Definition of the Times2() function */
double Times2(double x)
{
   double y;
   y=2*x;
   return y;
}
/* Main function */
/* This is where the program starts execution */
main()
{
  /* Call Times2 function */
  printf("2 times 5 is %4.0lf\n", Times2(5));
}
```

Note that you cannot make changes on (make assignments to) arguments.
For example, x = 2*x; is not allowed here.

```
2 times 5 is   10
```


How does it work?

1. As every C program the program starts from the main() statement.
2. The main() function consists of only one statement (printf) so this is executed.
3. For printf to be executed first Times2() function is called.
4. The actual argument 5 is converted to a double value (5.0) before calling the function. The execution of the function is as follows:
    1. A space for a real (double) argument is reserved in memory with label x and initialized to the actual argument value, 5.0.
    2. A space for a double variable is reserved in memory with label y.
    3. y=2*x; is executed. So, y becomes 10.0.
    4. Function returns the value of y. (ie 10.0)
5. The return value of function is used instead of the function so we see 10 in the output.


```
#include <stdio.h>
/* Defining Factorial and Sinus functions */
double Factorial(double n)
{
  double i,fact=1.0;
  for(i=2.0;i<=n;i++)
  {
     fact *= i;
  }
  return fact;
}

double Sinus(double x)
{
    const int n=5;
    int i;
    double x_to_2k1=x, sign=1.0, sin_x=0.0;
    for(i=1;i<=n;i++)
    {
      sin_x += sign * x_to_2k1 / Factorial((double) 2*i-1);
      sign= -sign;
      x_to_2k1 *= x*x;
    }
    return sin_x;
}
```

Note that the variable i here is different from the variable i in **Sinus** function. Variables defined in functions are called local variables. They can be used only inside the function. **When function terminates all local variables are removed from memory.**

The same rule applies for the arguments. Hence the variable x here is different (as a memory location) from the variable x in main() function.

This is called *casting*. The expression on the right is converted to a double. For this particular example has no effect on the program.

```
main()
{
 double x;
 const double PI = 3.14159265358979323846;
 for(x=0.0; x<=90.0; x+=15.0)
   printf("Sin(%2.0lf) = %5.3lf\n",x,Sinus(x * PI/180.0));
}
```

```
Sin( 0) = 0.000
Sin(15) = 0.259
Sin(30) = 0.500
Sin(45) = 0.707
Sin(60) = 0.866
Sin(75) = 0.966
Sin(90) = 1.000
```


Declaring Functions

A function declaration is the interface part (the part without the main body) of the function. Functions must be declared before they are used. However, definition of a function (implementation details) can be given later.

For example,

The function has to be declared before it is used.

```
#include <stdio.h>
double Times2(double x); /* Declaration of Times2 */
main(){
  printf("2 times 5 is %4.0lf\n", Times2(5));
}
/* Definition of the Times2() function */
double Times2(double x){
   double y;
   y=2*x;
   return y;
}
```

The function is defined after the main function.

Functions that do not return a value

Some functions do not need to return a value. These functions are declared as of type **void**. Similarly some functions do not need any arguments. **void** can also be used to state that the function has no arguments. For example,

```
void printWarning(void)
{
   printf("You have been warned!\n");
}
main()
{
  printWarning();
  printWarning();
}
```

```
You have been warned!
You have been warned!
```


Functions with more than one arguments

Some functions require more than one arguments. For example,

```
double MyPow(double x, double y)
{
 return exp(y*log(x));
}
```

If the number of arguments are not known, an ellipsis (...) can be used. For example, the declaration of the printf function looks like

```
int printf(const char *format_str, ...);
```

Write a function that realize the following function:

$$f(x) = \begin{cases} -x^2 & \text{if } (x \le -1.0) \\ \dfrac{1}{x^2} & \text{if } (-1.0 < x < 1.0) \\ x^2 & \text{otherwise} \end{cases}$$

```
double f(double x)
{
  double r;
  if (x <= -1.0)
     r = -x*x;
  else if ((-1.0 < x) && (x < 1.0))
     r = 1/(x*x);
  else
     r = x*x;
 return r;
}
```


Example

Write down a function named Sine(x) to calculate Sin(x) approximately. Use the following equation

$$Sin(x) \cong \sum_{k=1}^{n} \frac{(-1)^{k+1} x^{2k-1}}{(2k-1)!}$$

The function should not take into account any terms with absolute value less than 10^{-6} and should take at most 16 terms of the series. You may assume that math library is being used and a function called Factorial has been defined to find the factorials of numbers.

```
double Sine(double x)   {
  double sin_x=0.0;
  int k;
  double term_k;

  for(k=1;k<=16;k++) {
    term_k = pow(-1, k+1) * pow(x,2*k-1) / Factorial(2*k-1);
    sin_x = sin_x + term_k;
    if (fabs(term_k) < 1.0e-6)
      break;
  }
  return sin_x;
}
```

fabs() function finds the absolute value of floating point numbers.