ESc 101: Fundamentals of Computing

Lecture 16-17

Feb 8 and 10, 2010

Outline

Functions in Mathematics

- Functions in C represent a algorithm for carrying out a specific task.
- Functions in other areas also do the same!
- For example, the functions sin, cos, . . ., in mathematics.

Functions in Mathematics

- Functions in C represent a algorithm for carrying out a specific task.
- Functions in other areas also do the same!
- For example, the functions sin, cos, . . ., in mathematics.

Functions in Mathematics

- Functions in C represent a algorithm for carrying out a specific task.
- Functions in other areas also do the same!
- For example, the functions sin, cos, . . ., in mathematics.

The sin Function

The sin function represents the following algorithm:

```
real sin(real x)

1. Draw a right-angle triangle with the diagonal of length
    one (in your favorite unit) and the angle from base
    equal to x radians.
2. Measure the perpendicular length (in the same unit).
3. Return the measured value.
```

Advantages of Using sin Function

Saves Time: We do not have to write the whole algorithm every time we refer to $\sin(x)$.

Saves Space: Writing algorithm every time consumes more space.

More Understandable: Anyone can understand more easily on seeing to $\sin(x)$ than the the algorithm.

No Loss of Information: Since everyone knows that sin corresponds to the above algorithm, its use is unambiguous.

Advantages of Using sin Function

Saves Time: We do not have to write the whole algorithm every time we refer to $\sin(x)$.

Saves Space: Writing algorithm every time consumes more space.

More Understandable: Anyone can understand more easily on seeing to $\sin(x)$ than the the algorithm.

No Loss of Information: Since everyone knows that sin corresponds to the above algorithm, its use is unambiguous.

Advantages of Using sin Function

Saves Time: We do not have to write the whole algorithm every time we refer to $\sin(x)$.

Saves Space: Writing algorithm every time consumes more space.

More Understandable: Anyone can understand more easily on seeing to $\sin(x)$ than the the algorithm.

No Loss of Information: Since everyone knows that sin corresponds to the above algorithm, its use is unambiguous.

Advantages of Using sin Function

Saves Time: We do not have to write the whole algorithm every time we refer to $\sin(x)$.

Saves Space: Writing algorithm every time consumes more space.

More Understandable: Anyone can understand more easily on seeing to $\sin(x)$ than the the algorithm.

No Loss of Information: Since everyone knows that sin corresponds to the above algorithm, its use is unambiguous.

Functions on Real Life

- We implicitly use functions in real life everywhere.
- Examples abound: Booting PC, Cooking rice, Reaching Rave Moti, ...
- Let us expand some of these.

Functions on Real Life

- We implicitly use functions in real life everywhere.
- Examples abound: Booting PC, Cooking rice, Reaching Rave Moti, ...
- Let us expand some of these.

Functions on Real Life

- We implicitly use functions in real life everywhere.
- Examples abound: Booting PC, Cooking rice, Reaching Rave Moti, ...
- Let us expand some of these.

Reaching Rave Moti

The algorithm represented by this function is something similar to the following:

```
reach_Rave_Moti()

1. If it is close to IITK bus time go to bus stop
    and catch the bus.
2. Else, go to IITK gate and catch a tempo.
3. Get off at Rawatpur.
4. Walk until railways crossing.
5. Turn right, walk a bit more.
6. Find Rave Moti on the right.
```

Advantages of Using this Function

- In our conversation, we almost never mention the algorithm.
- Instead, we just say Go to Rave Moti, essentially referring to the algorithm.
- The only time we mention the algorithm is when someone is new to Kanpur and does not know where Rave Moti is.
- To such a person, we describe it once, and then onwards refer to it by just the name.
- The description first time corresponds to defining the function in C (both are done once).
- Subsequence usage corresponds to calling the function in C.

Advantages of Using this Function

- In our conversation, we almost never mention the algorithm.
- Instead, we just say Go to Rave Moti, essentially referring to the algorithm.
- The only time we mention the algorithm is when someone is new to Kanpur and does not know where Rave Moti is.
- To such a person, we describe it once, and then onwards refer to it by just the name.
- The description first time corresponds to defining the function in C (both are done once).
- Subsequence usage corresponds to calling the function in C.

Advantages of Using this Function

- In our conversation, we almost never mention the algorithm.
- Instead, we just say Go to Rave Moti, essentially referring to the algorithm.
- The only time we mention the algorithm is when someone is new to Kanpur and does not know where Rave Moti is.
- To such a person, we describe it once, and then onwards refer to it by just the name.
- The description first time corresponds to defining the function in C (both are done once).
- Subsequence usage corresponds to calling the function in C.

Advantages of Using this Function

- In our conversation, we almost never mention the algorithm.
- Instead, we just say Go to Rave Moti, essentially referring to the algorithm.
- The only time we mention the algorithm is when someone is new to Kanpur and does not know where Rave Moti is.
- To such a person, we describe it once, and then onwards refer to it by just the name.
- The description first time corresponds to defining the function in C (both are done once).
- Subsequence usage corresponds to calling the function in C.

Advantages of Using this Function

- In our conversation, we almost never mention the algorithm.
- Instead, we just say Go to Rave Moti, essentially referring to the algorithm.
- The only time we mention the algorithm is when someone is new to Kanpur and does not know where Rave Moti is.
- To such a person, we describe it once, and then onwards refer to it by just the name.
- The description first time corresponds to defining the function in C (both are done once).
- Subsequence usage corresponds to calling the function in C.

Functions with Parameters in Real Life

- The Cook Rice function takes raw rice as parameter (the amount and type of the rice).
- Once it is over, the parameter changes to cooked rice (unless there is an error somewhere).
- Again, this corresponds to a C function with parameters whose values change after execution.

Functions with Parameters in Real Life

- The Cook Rice function takes raw rice as parameter (the amount and type of the rice).
- Once it is over, the parameter changes to cooked rice (unless there is an error somewhere).
- Again, this corresponds to a C function with parameters whose values change after execution.

Functions with Parameters in Real Life

- The Cook Rice function takes raw rice as parameter (the amount and type of the rice).
- Once it is over, the parameter changes to cooked rice (unless there is an error somewhere).
- Again, this corresponds to a C function with parameters whose values change after execution.

Functions in C

- Parameters to functions in C do not change value except for arrays after their execution of over.
- This is due to the fact that C follows call-by-value policy.

Functions in C

- Parameters to functions in C do not change value except for arrays after their execution of over.
- This is due to the fact that C follows call-by-value policy.

Call by Value Policy

Consider following segment of code:

```
main()
{
    int x;
    int y;

    x = 10;
    y = 20;
    swap(x, y);
    printf("x = %d, y = %d\n", x, y);
}
```

Call by Value Policy

```
void swap(int a, int b)
{
    int c;

    c = a;
    a = b;
    b = c;
}
```

Call by Value Policy

- The value of `x` and `y` before function `f` is called is 10 and 20.
- These will remain so even after the function is called.
- The reason is that when `f` is called, two new memory locations are reserved and given names `a` and `b`.
- In these two locations, the values of `x` and `y` are copied.
- Once the execution of the function is over, these two memory locations are discarded.
- Their values are not copied back to the memory locations named `x` and `y`!

Call by Value Policy

- The value of `x` and `y` before function `f` is called is 10 and 20.
- These will remain so even after the function is called.
- The reason is that when `f` is called, two new memory locations are reserved and given names `a` and `b`.
- In these two locations, the values of `x` and `y` are copied.
- Once the execution of the function is over, these two memory locations are discarded.
- Their values are not copied back to the memory locations named `x` and `y`!

Call by Value Policy

- The value of `x` and `y` before function `f` is called is 10 and 20.
- These will remain so even after the function is called.
- The reason is that when `f` is called, two new memory locations are reserved and given names `a` and `b`.
- In these two locations, the values of `x` and `y` are copied.
- Once the execution of the function is over, these two memory locations are discarded.
- Their values are not copied back to the memory locations named `x` and `y`!

Call by Value Policy

- The value of `x` and `y` before function `f` is called is 10 and 20.
- These will remain so even after the function is called.
- The reason is that when `f` is called, two new memory locations are reserved and given names `a` and `b`.
- In these two locations, the values of `x` and `y` are copied.
- Once the execution of the function is over, these two memory locations are discarded.
- Their values are not copied back to the memory locations named `x` and `y`!

Call by Value Policy

- The value of `x` and `y` before function `f` is called is 10 and 20.
- These will remain so even after the function is called.
- The reason is that when `f` is called, two new memory locations are reserved and given names `a` and `b`.
- In these two locations, the values of `x` and `y` are copied.
- Once the execution of the function is over, these two memory locations are discarded.
- Their values are not copied back to the memory locations named `x` and `y`!

Call by Value Policy

- The value of `x` and `y` before function `f` is called is 10 and 20.
- These will remain so even after the function is called.
- The reason is that when `f` is called, two new memory locations are reserved and given names `a` and `b`.
- In these two locations, the values of `x` and `y` are copied.
- Once the execution of the function is over, these two memory locations are discarded.
- Their values are not copied back to the memory locations named `x` and `y`!

Why are Arrays Treated Different?

- They are not treated differently!
- But is it better for now to pretend that they are treated differently.
- We will see the explanation slightly later.

Why are Arrays Treated Different?

- They are not treated differently!
- But is it better for now to pretend that they are treated differently.
- We will see the explanation slightly later.

Outline

Scope of a Variable

- In C, each variable has a scope associated with it.
- This scope is a block of statements.
- The variable is visible only within this block of statements.
- Any attempt to access its value outside this block results in an error.

Scope of a Variable

- In C, each variable has a scope associated with it.
- This scope is a block of statements.
- The variable is visible only within this block of statements.
- Any attempt to access its value outside this block results in an error.

Scope of a Variable

- In C, each variable has a scope associated with it.
- This scope is a block of statements.
- The variable is visible only within this block of statements.
- Any attempt to access its value outside this block results in an error.

Scope of a Variable

- In C, each variable has a scope associated with it.
- This scope is a block of statements.
- The variable is visible only within this block of statements.
- Any attempt to access its value outside this block results in an error.

Scope of a Variable

- The first statement in the scope is, of course, the statement declaring the variable.
- The last statement in the scope is the last statement of the statement block in which the variable is defined.
- So a variable defined inside a function definition, or declared as parameter in the definition, is available only until the last statement of the function.

Scope of a Variable

- The first statement in the scope is, of course, the statement declaring the variable.
- The last statement in the scope is the last statement of the statement block in which the variable is defined.
- So a variable defined inside a function definition, or declared as parameter in the definition, is available only until the last statement of the function.

Scope of a Variable

- The first statement in the scope is, of course, the statement declaring the variable.
- The last statement in the scope is the last statement of the statement block in which the variable is defined.
- So a variable defined inside a function definition, or declared as parameter in the definition, is available only until the last statement of the function.

Variable Declaration and Scope

```
int global = 0;
void foo();

int main()
{
    printf("in main global = %d\n", global);
    foo(0);
    global = 42;
    foo(1);

    int global = 100;
    printf("in main after dec global = %d\n",global);
    foo(2);
    global=10;
    foo(3);
    printf("in main after update global = %d\n",global);
}
```

Variable Declaration and Scope

```
void foo(int t)
{
    int local = 120;
    printf("in foo(%d) global = %d, local = %d\n",
                t, global, local);
}
```

Variable scope

- A variable comes to life when it is declared.
- A variable lives as long as the smallest block that contains its declartion is active
- A variable outside every functions is global and lives forever.
- Local variables have precedence over global ones.

Variables in for loop

```
for (int i = 0; i < 100; i++)
{
     /* do something */
}
```

The variable `i` is valid only within the for loop.

Variables in for loop

```
for (int i = 0; i < 100; i++)
{
     /* do something */
}
```

The variable `i` is valid only within the for loop.

Variables inside function

```
int foo(int x)
{
      /* some stuff */
      float local;

      foo(bar);
  }
```

- The variable is local to the function.
- For a new call of `foo` there is a new variable named local valid for that call.

Variables inside function

```
int foo(int x)
{
      /* some stuff */
      float local;

      foo(bar);
  }
```

- The variable is local to the function.
- For a new call of `foo` there is a new variable named local valid for that call.

Variables inside function

```
int foo(int x)
{
      /* some stuff */
      float local;

      foo(bar);
  }
```

- The variable is local to the function.
- For a new call of `foo` there is a new variable named local valid for that call.

Outline

Example of Bad Programming

```
main()
{
int i,j,k;
int a[100];
for (i=0;i<100;i++) {
scanf("%d",&a[i]);
if (a[i]<0)
break;}
for (j=i-1;j>=0;j--)
printf("%d ",a[j]);
}
```

So What is Wrong?

```
main()
{
int i,j,k;
int a[100];
for (i=0;i<100;i++) {
scanf("%d",&a[i]);
if (a[i]<0)
break;}
for (j=i-1;j>=0;j--)
printf("%d ",a[j]);
}
```

Bad declaration: variable `k` is never used. Also, variables should be declared in different lines.

So What is Wrong?

```
main()
{
int i;
int j;
int a[100];
for (i=0;i<100;i++) {
scanf("%d",&a[i]);
if (a[i]<0)
break;}
for (j=i-1;j>=0;j--)
printf("%d ",a[j]);
}
```

No indentation!

So What is Wrong?

```
main()
{
    int i;
    int j;
    int a[100];
    for (i=0;i<100;i++) {
        scanf("%d",&a[i]);
        if (a[i]<0)
        break;}
    for (j=i-1;j>=0;j--)
        printf("%d ",a[j]);
}
```

Braces should be aligned.

So What is Wrong?

```
main()
{
    int i;
    int j;
    int a[100];
    for (i=0;i<100;i++) {
        scanf("%d",&a[i]);
        if (a[i]<0)
        break;
    }
    for (j=i-1;j>=0;j--)
        printf("%d ",a[j]);
}
```

Use blanks to separate parts of code.

So What is Wrong?

```
main()
{
    int i;
    int j;
    int a[100];
    for (i = 0; i < 100; i++) {
        scanf("%d", &a[i]);
        if (a[i] < 0)
        break;
    }
    for (j = i-1; j >= 0; j--)
        printf("%d ", a[j]);
}
```

Insert a blank line between variable declarations and statements, and add comments!

So What is Wrong?

```
/* Reads a sequence of positive numbers terminated by a
 * negative number, and outputs the sequence in reverse order.
 */
main()
{
    int i;
    int j;
    int a[100]; /* stores the sequence */

    for (i = 0; i < 100; i++) { /* read the sequence */
        scanf("%d", &a[i]);
        if (a[i] < 0) /* end of input */
        break;
    }
    for (j = i-1; j >= 0; j--) /* output in reverse order */
        printf("%d ", a[j]);
}
```