

CS1100 – Introduction to Programming

Instructor:

Shweta Agrawal (shweta.a@cse.iitm.ac.in)

Lecture 19

CS1100 – Introduction to Programming

- Programming : From Turtle to C.
- Data Types in C, Operators. Input and the Output.
- Modifying the control flow in Programs if-else, switch, loops : while, do-while, for.
- Implementing numerical methods using a C-program.
- Arrays and Strings in C.



So Far

CS1100 – Introduction to Programming

- Programming : From Turtle to C.
- Data Types in C, Operators. Input and the Output.
- Modifying the control flow in Programs if-else, switch, loops : while, do-while, for.
- Implementing numerical methods using a C-program.
- Arrays and Strings in C.

} So Far

-
- functions in C.
 - Blocks and Scope

} Coming Up...

Character grids

- Given a character grid, and a string `s`, print the indices of the rows and columns of grid that contain `s`.
-

c	a	t	t	y
c	c	s	e	p
e	s	c	e	l
s	e	e	s	e
h	a	p	s	a

- Assume a code `find(x,y)` - that returns the index of `y` in `x`. write a code for `find` now.
- Which rows and columns contain `cse`?

Character grids

- Given a character grid, and a string s , print the indices of the rows and columns of grid that contain s .
-

c	a	t	t	y
c	c	s	e	p
e	s	c	e	l
s	e	e	s	e
h	a	p	s	a

- Which rows and columns contain `cse`?
- Assume a code `find(x,y)` - that returns the index of y in x . write a code for `find` now.
- For each row R , `find(R, s)`.
- For each column C , `find(C, s)`.

Character grids

- Given a character grid, and a string s , print the indices of the rows and columns of grid that contain s .

Pseudo-code:

- For each row R of grid
 - If ($\text{find}(R, s)$) print(index of R).
- $\text{gridT} = \text{transpose}(\text{grid})$.
- For each row R of gridT
 - If ($\text{find}(R, s)$) print(index of R).

Character grids

- Given a character grid, and a string s , print the indices of the rows and columns of grid that contain s .

Pseudo-code:

- For each row R of grid
 - If ($\text{find}(R, s)$) print(index of R).
- $\text{gridT} = \text{transpose}(\text{grid})$.
- For each row R of gridT
 - If ($\text{find}(R, s)$) print(index of R).

functions : concept of writing the programs for `find`, `transpose` etc separately and using them in the main program.

Can we define our own “commands”?

- We already know of commands like :
 - `sqrt(x)` evaluates to the square root of `x`.
 - `pow(x,k)` returns the value of `x` multiplied by itself value of `k` many times.
 - `forward(d)` moves the turtle forward `d` units.

Can we define our own “commands”?

- We already know of commands like :
 - `sqrt(x)` evaluates to the square root of `x`.
 - `pow(x,k)` returns the value of `x` multiplied by itself value of `k` many times.
 - `forward(d)` moves the turtle forward `d` units.
- Can we define new commands? e.g.
 - `gcd(m,n)` should evaluate to the GCD of `m,n`.
 - `dash(d)` should move the turtle forward, but draw dashes as it moves rather than a continuous line.

Can we define our own “commands”?

- We already know of commands like :
 - `sqrt(x)` evaluates to the square root of `x`.
 - `pow(x,k)` returns the value of `x` multiplied by itself value of `k` many times.
 - `forward(d)` moves the turtle forward `d` units.
- Can we define new commands? e.g.
 - `gcd(m,n)` should evaluate to the GCD of `m,n`.
 - `dash(d)` should move the turtle forward, but draw dashes as it moves rather than a continuous line.
- **functions** - official name for such commands, implemented seperately.

functions in C

functions in C-language helps us to :

functions in C

functions in C-language helps us to :

- Define our own subtasks which we want to use in bigger tasks and program them to reuse them whenever needed. This is called **modular approach** to program design. Very effective and less error-prone.

functions in C

functions in C-language helps us to :

- Define our own subtasks which we want to use in bigger tasks and program them to reuse them whenever needed. This is called **modular approach** to program design. Very effective and less error-prone.
- Define our own mathematical (or otherwise) functions, and use them.

functions in C

functions in C-language helps us to :

- Define our own subtasks which we want to use in bigger tasks and program them to reuse them whenever needed. This is called **modular approach** to program design. Very effective and less error-prone.
- Define our own mathematical (or otherwise) functions, and use them.
- Re-use lots of code, tested code.

functions in C

functions in C-language helps us to :

- Define our own subtasks which we want to use in bigger tasks and program them to reuse them whenever needed. This is called **modular approach** to program design. Very effective and less error-prone.
- Define our own mathematical (or otherwise) functions, and use them.
- Re-use lots of code, tested code.
- Giving a job to functions \equiv outsourcing.

functions in C

```
#include <stdio.h>

int main()
{
    int var1 = 10;
    int var2 = 20;
    int var3, var4;
    var3 = FindSum(var1, var2);
    var4 = FindSum(var3, var2);
    printf("%d", var3);
    printf("%d", var4);
    return 0;
}
```


functions in C

```
#include <stdio.h>

int main()
{
    int var1 = 10;
    int var2 = 20;
    int var3, var4;
    var3 = FindSum(var1, var2);
    var4 = FindSum(var3, var2);
    printf("%d", var3);
    printf("%d", var4);
    return 0;
}
```

What happens when you compile this program?

functions in C

```
#include <stdio.h>

int main()
{
    int var1 = 10;
    int var2 = 20;
    int var3, var4;
    var3 = FindSum(var1, var2);
    var4 = FindSum(var3, var2);
    printf("%d", var3);
    printf("%d", var4);
    return 0;
}
```

What happens when you compile this program?

functions in C

```
#include <stdio.h>

int main()
{
    int var1 = 10;
    int var2 = 20;
    int var3, var4;
    var3 = FindSum(var1, var2);
    var4 = FindSum(var3, var2);
    printf("%d", var3);
    printf("%d", var4);
    return 0;
}
```

What happens when you compile this program?

- Before using the function - the compiler needs to be told about the function.
- How to tell them?

functions in C

```
#include <stdio.h>

int main()
{
    int var1 = 10;
    int var2 = 20;
    int var3, var4;
    var3 = FindSum(var1, var2);
    var4 = FindSum(var3, var2);
    printf("%d", var3);
    printf("%d", var4);
    return 0;
}
```

What happens when you compile this program?

- Before using the function - the compiler needs to be told about the function.
- How to tell them?
Declare functions.
- Only name, return type, number of arguments and their type need to be told initially. This is called the **prototype** of a function.

Completing the example

```
#include "stdio.h"
int FindSum(int, int);

int main()
{
    int var1 = 10;
    int var2 = 20;
    int var3, var4;
    var3 = FindSum(var1, var2);
    var4 = FindSum(var3, var2);
    printf("%d\n", var3);
    printf("%d\n", var4);
    return 0;
}

int FindSum(int a, int b)
{
    int c=a+b;
    return c;
}
```

Prototype of a function:

- Name of the function
- Arguments and their types.
- Return type.

Completing the example

```
#include "stdio.h"
int FindSum(int, int);

int main()
{
    int var1 = 10;
    int var2 = 20;
    int var3, var4;
    var3 = FindSum(var1, var2);
    var4 = FindSum(var3, var2);
    printf("%d\n", var3);
    printf("%d\n", var4);
    return 0;
}

int FindSum(int a, int b)
{
    int c=a+b;
    return c;
}
```

Prototype of a function:

- Name of the function
- Arguments and their types.
- Return type.

Defintion of the function:

- Return type
- Function name
- Names of arguments and their types
- Body of the function
- Local variables
- Return statement

Prototype can be replaced by definition

```
#include "stdio.h"
int FindSum(int a, int b)
{
    int c=a+b;
    return c;
}

int main()
{
    int var1 = 10;
    int var2 = 20;
    int var3,var4;
    var3 = FindSum(var1,var2);
    var4 = FindSum(var3,var2);
    printf("%d\n", var3);
    printf("%d\n", var4);
    return 0;
}
```

Prototype : Not provided.

Defintion of the function:

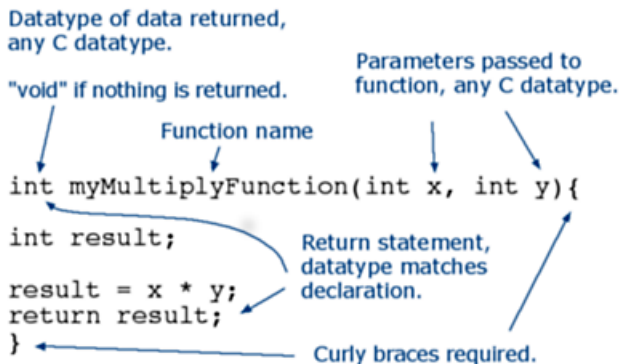
- Return type
- Function name
- Names of arguments and their types
- Body of the function
- Local variables
- Return statement

Extra Example: Anatomy of a function definition

Function `myMultipleFunction` returns the result of multiplication of integers.

Extra Example: Anatomy of a function definition

Function `myMultiplyFunction` returns the result of multiplication of integers.

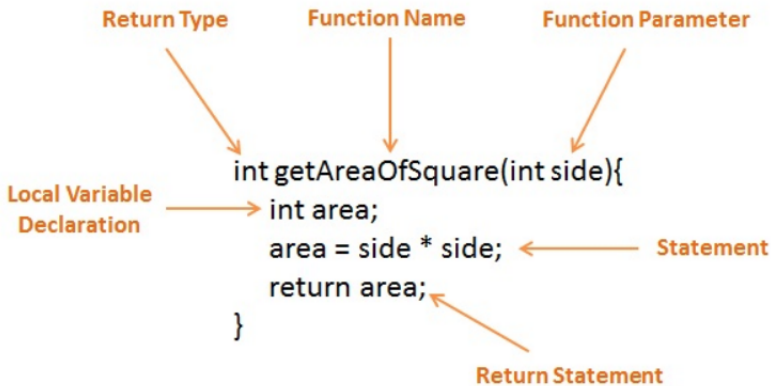


Extra Example : Anatomy of a function definition

Function `getAreaOfSquare` returns the area of a square in cm^2 whose side is of length `side` in cm .

Extra Example : Anatomy of a function definition

Function `getAreaOfSquare` returns the area of a square in cm^2 whose side is of length `side` in cm .



Back to Example 1 : Invocation of the Function

```
#include "stdio.h"
int FindSum(int, int);

int main()
{
    int var1 = 10;
    int var2 = 20;
    int var3, var4;
    var3 = FindSum(var1, var2);
    var4 = FindSum(var3, var2);
    printf("%d\n", var3);
    printf("%d\n", var4);
    return 0;
}

int FindSum(int a, int b)
{
    int c=a+b;
    return c;
}
```

Back to Example 1 : Invocation of the Function

```
#include "stdio.h"
int FindSum(int, int);

int main()
{
    int var1 = 10;
    int var2 = 20;
    int var3, var4;
    var3 = FindSum(var1, var2);
    var4 = FindSum(var3, var2);
    printf("%d\n", var3);
    printf("%d\n", var4);
    return 0;
}

int FindSum(int a, int b)
{
    int c=a+b;
    return c;
}
```

Invocation (function call):

- arguments passed to the function.

Back to Example 1 : Invocation of the Function

```
#include "stdio.h"
int FindSum(int, int);

int main()
{
    int var1 = 10;
    int var2 = 20;
    int var3, var4;
    var3 = FindSum(var1, var2);
    var4 = FindSum(var3, var2);
    printf("%d\n", var3);
    printf("%d\n", var4);
    return 0;
}

int FindSum(int a, int b)
{
    int c=a+b;
    return c;
}
```

Invocation (function call):

- arguments passed to the function.
- receiving the return value from the function.

Back to Example 1 : Invocation of the Function

```
#include "stdio.h"
int FindSum(int, int);

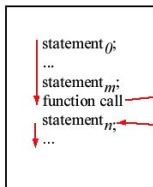
int main()
{
    int var1 = 10;
    int var2 = 20;
    int var3, var4;
    var3 = FindSum(var1, var2);
    var4 = FindSum(var3, var2);
    printf("%d\n", var3);
    printf("%d\n", var4);
    return 0;
}

int FindSum(int a, int b)
{
    int c=a+b;
    return c;
}
```

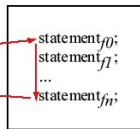
Invocation (function call):

- arguments passed to the function.
- receiving the return value from the function.
- a function can be called multiple times, with different arguments.

Code for caller of a function



Code for function



Another Example : implementing fact

```
#include<stdio.h>
int fact(int);

int main() {
    int x, y;
    printf("Enter a number:");
    scanf("%d", &x);
    y = fact(x);
    printf("%d\n",y);
}

int fact(int n) {
    int i = 1;
    while(n>0) {
        i = i * n;
        n--;
    }
    return i;
}
```


Another Example : implementing fact

```
#include<stdio.h>
int fact(int);

int main() {
    int x, y;
    printf("Enter a number:");
    scanf("%d", &x);
    y = fact(x);
    printf("%d\n",y);
}

int fact(int n) {
    int i = 1;
    while(n>0) {
        i = i * n;
        n--;
    }
    return i;
}
```

Take Aways:

- We can write more complicated code within the body of functions.

Another Example : implementing fact

```
#include<stdio.h>
int fact(int);

int main() {
    int x, y;
    printf("Enter a number:");
    scanf("%d", &x);
    y = fact(x);
    printf("%d\n",y);
}

int fact(int n) {
    int i = 1;
    while(n>0) {
        i = i * n;
        n--;
    }
    return i;
}
```

Take Aways:

- We can write more complicated code within the body of functions.
- We can define our own math functions.

Another Example : implementing fact

```
#include<stdio.h>
int fact(int);

int main() {
    int x, y;
    printf("Enter a number:");
    scanf("%d", &x);
    y = fact(x);
    printf("%d\n",y);
}

int fact(int n) {
    int i = 1;
    while(n>0) {
        i = i * n;
        n--;
    }
    return i;
}
```

Take Aways:

- We can write more complicated code within the body of functions.
- We can define our own math functions.
- In fact, `math.h` has such definitions to compute `sqrt` and `pow` etc.

Another Example : implementing fact

```
#include<stdio.h>
int fact(int);

int main() {
    int x, y;
    printf("Enter a number:");
    scanf("%d", &x);
    y = fact(x);
    printf("%d\n",y);
}

int fact(int n) {
    int i = 1;
    while(n>0) {
        i = i * n;
        n--;
    }
    return i;
}
```

Take Aways:

- We can write more complicated code within the body of functions.
- We can define our own math functions.
- In fact, math.h has such definitions to compute sqrt and pow etc.
- More interestingly, printf and scanf are also functions defined inside stdio.h.

Functions with arguments & no return value

```
#include<stdio.h>

void area(float rad); // Prototype Declaration

int main()
{
    float rad;
    printf("Enter the radius : ");
    scanf("%f",&rad);
    area(rad);
}

void area(float rad)
{
    float ar;
    ar = 3.14 * rad * rad ;
    printf("Area of Circle = %f",ar);
}
```

Functions with no arguments & no return value

```
#include<stdio.h>
void area(); // Prototype Declaration

void main()
{
    area();
}

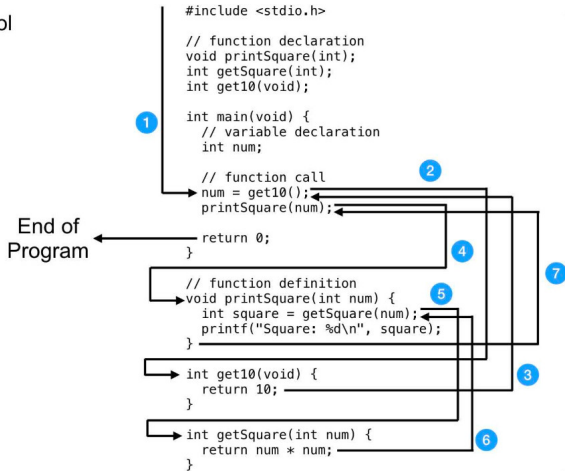
void area()
{
    float area_circle;
    float rad;
    printf("Enter the radius : ");
    scanf("%f",&rad);

    area_circle = 3.14 * rad * rad ;

    printf("Area of Circle = %f",area_circle);
}
```

Control Flow : More complicated example

Flow of control

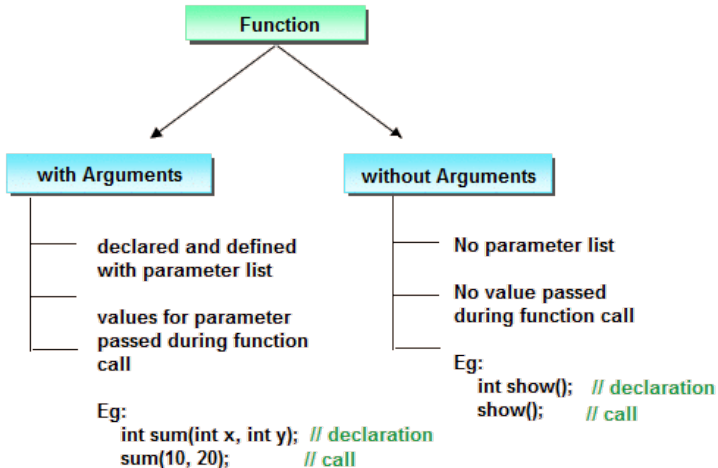


Now this slide may make more sense

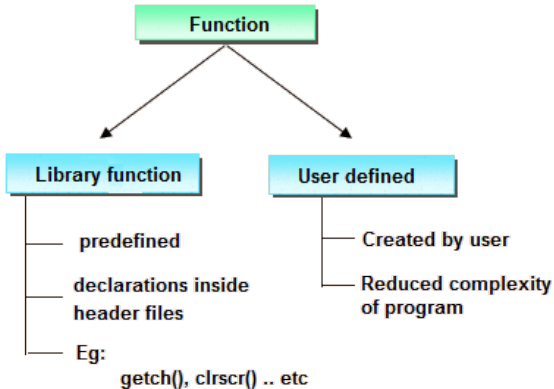
functions in C-language helps us to :

- Define our own subtasks which we want to use in bigger tasks and program them to reuse them whenever needed. This is called **modular approach** to program design. Very effective and less error-prone.
- Define our own mathematical (or otherwise) functions, and use them.
- Re-use lots of code, tested code.
- Giving a job to functions \equiv outsourcing.

Classifying functions in C



Classifying functions in C



New Concept : Blocks and Scope

Block : A program segment written within curly brackets.

Scope : The program segment where a particular declaration of a variable is applicable.

New Concept : Blocks and Scope

Block : A program segment written within curly brackets.

Scope : The program segment where a particular declaration of a variable is applicable.

```
int global;           // a global variable
int main()
{
    int local;       // a local variable

    global = 1;     // global can be used here
    local = 2;     // so can local

    {               // beginning a new block
        int very_local // this is local to the block

        very_local = global+local;
    }

    // We just closed the block
    // very_local can not be used
}
```

The diagram illustrates the scope of variables in the provided code. Three nested curly brackets on the left side indicate the following scopes:

- Scope of global:** Encompasses the entire program, from the first line to the final closing brace.
- Scope of local:** Encompasses the `main()` function, from its opening brace to its closing brace.
- Scope of very_local:** Encompasses the innermost block, from its opening brace to its closing brace.

Practicing the Concept : Blocks and Scope

```
#include <stdio.h>
int FindSum(int, int);
int var1 = 10;

int main()
{
    int var2 = 20;
    {
        int var3;
        var3 = FindSum(var1,var2);
        printf("%d\n", var3);
    }
    float var3=100;
    printf("%f\n", var3);
    return 0;
}

int FindSum(int a, int b)
{
    int c=a+b;
    printf("%d\n",var1);
    return c;
}
```

Practicing the Concept : Blocks and Scope

```
#include <stdio.h>
int FindSum(int, int);
int var1 = 10;

int main()
{
    int var2 = 20;
    {
        int var3;
        var3 = FindSum(var1, var2);
        printf("%d\n", var3);
    }
    float var3=100;
    printf("%f\n", var3);
    return 0;
}
```

```
int FindSum(int a, int b)
{
    int c=a+b;
    printf("%d\n", var1);
    return c;
}
```

- Scope of var2 is the whole of main

Practicing the Concept : Blocks and Scope

```
#include <stdio.h>
int FindSum(int, int);
int var1 = 10;

int main()
{
    int var2 = 20;
    {
        int var3;
        var3 = FindSum(var1, var2);
        printf("%d\n", var3);
    }
    float var3=100;
    printf("%f\n", var3);
    return 0;
}
```

```
int FindSum(int a, int b)
{
    int c=a+b;
    printf("%d\n", var1);
    return c;
}
```

- Scope of var2 is the whole of main
- Scope of int var3 is only the inner block.

Practicing the Concept : Blocks and Scope

```
#include <stdio.h>
int FindSum(int, int);
int var1 = 10;

int main()
{
    int var2 = 20;
    {
        int var3;
        var3 = FindSum(var1, var2);
        printf("%d\n", var3);
    }
    float var3=100;
    printf("%f\n", var3);
    return 0;
}
```

```
int FindSum(int a, int b)
{
    int c=a+b;
    printf("%d\n", var1);
    return c;
}
```

- Scope of var2 is the whole of main
- Scope of int var3 is only the inner block.
- Scope of float var3 is only the outer block.

Practicing the Concept : Blocks and Scope

```
#include <stdio.h>
int FindSum(int, int);
int var1 = 10;

int main()
{
    int var2 = 20;
    {
        int var3;
        var3 = FindSum(var1, var2);
        printf("%d\n", var3);
    }
    float var3=100;
    printf("%f\n", var3);
    return 0;
}
```

```
int FindSum(int a, int b)
{
    int c=a+b;
    printf("%d\n", var1);
    return c;
}
```

- Scope of var2 is the whole of main
- Scope of int var3 is only the inner block.
- Scope of float var3 is only the outer block.
- Scope of int var1 is the whole program.

Practicing the Concept : Blocks and Scope

```
#include <stdio.h>
int FindSum(int, int);
int var1 = 10;

int main()
{
    int var2 = 20;
    {
        int var3;
        var3 = FindSum(var1, var2);
        printf("%d\n", var3);
    }
    float var3=100;
    printf("%f\n", var3);
    return 0;
}
```

```
int FindSum(int a, int b)
{
    int c=a+b;
    printf("%d\n", var1);
    return c;
}
```

- Scope of var2 is the whole of main
- Scope of int var3 is only the inner block.
- Scope of float var3 is only the outer block.
- Scope of int var1 is the whole program.

Local vs **Global** variables : var1 is global but var2 is local for main function.

Take Aways

- Functions : Modular Programming. Build programs brick by brick. Reusing built and tested part.
- Declaration, Definition and Invocation of functions.
- Block and Scope. Local and Global Variables.

Coming up :

- We will do hands-on examples of using functions.
- Is `main` program a function?
Why are we ending with `"return 0;"` Who is it returning to?

Take Aways

- Functions : Modular Programming. Build programs brick by brick. Reusing built and tested part.
- Declaration, Definition and Invocation of functions.
- Block and Scope. Local and Global Variables.

Coming up :

- We will do hands-on examples of using functions.
- Is `main` program a function?
Why are we ending with `"return 0;"` Who is it returning to?
- Can a function invoke other functions?

Take Aways

- Functions : Modular Programming. Build programs brick by brick. Reusing built and tested part.
- Declaration, Definition and Invocation of functions.
- Block and Scope. Local and Global Variables.

Coming up :

- We will do hands-on examples of using functions.
- Is `main` program a function?
Why are we ending with `"return 0;"` Who is it returning to?
- Can a function invoke other functions? **Yes !**

Take Aways

- Functions : Modular Programming. Build programs brick by brick. Reusing built and tested part.
- Declaration, Definition and Invocation of functions.
- Block and Scope. Local and Global Variables.

Coming up :

- We will do hands-on examples of using functions.
- Is `main` program a function?
Why are we ending with `"return 0;"` Who is it returning to?
- Can a function invoke other functions? **Yes !**
- Can a function invoke itself?

Take Aways

- Functions : Modular Programming. Build programs brick by brick. Reusing built and tested part.
- Declaration, Definition and Invocation of functions.
- Block and Scope. Local and Global Variables.

Coming up :

- We will do hands-on examples of using functions.
- Is `main` program a function?
Why are we ending with `"return 0;"` Who is it returning to?
- Can a function invoke other functions? **Yes !**
- Can a function invoke itself? **Yes !** Recursion !.