

FUNCTIONS

CS10003: PROGRAMMING AND DATA STRUCTURES



Introduction

Function

- A program segment that carries out a specific, well-defined task.
- Examples
 - A function to find the gcd of two numbers
 - A function to find the largest of n numbers

A function will carry out its intended task whenever it is called

- Functions may call other functions (or itself)
- A function may be called multiple times (with different arguments)

Every C program consists of one or more functions.

- One of these functions must be called “main”.
- Execution of the program always begins by carrying out the instructions in “main”.

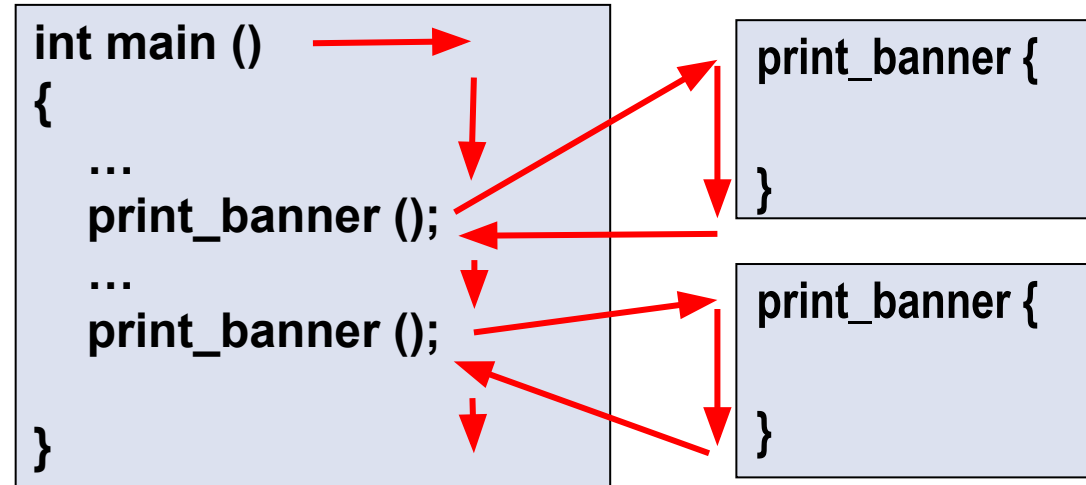
Function Control Flow

Code

```
void print_banner ( )  
{  
    printf("*****\n");  
}
```

```
int main ( )  
{  
    . . .  
    print_banner ( ) ;  
    . . .  
    print_banner ( ) ;  
}
```

Execution



If function A calls function B:

A : calling function / caller function

B : called function

Why Functions?

Functions allow one to develop a program in a modular fashion.

- Codes become readable
- Codes become manageable to debug and maintain

Write your own functions to avoid writing the same code segments multiple times

- If you check several integers for primality in various places of your code, just write a single primality-testing function, and call it on all occasions

Use existing functions as building blocks for new programs

- Use functions without rewriting them yourself every time it is needed
- These functions may be written by you or by others (like `sqrt()`, `printf()`)

Abstraction: Hide internal details (library functions)

Use of functions: *Area of a circle*

```
#include <stdio.h>
```

```
/* Function to compute the area of a circle */  
float myfunc (float r)  
{  
    float a;  
    a = 3.14159 * r * r;  
    return a;      /* return result */  
}
```

Function definition

Function argument

```
main()
```

```
{  
    float radius, area;  
  
    scanf ("%f", &radius);  
    area = myfunc (radius);  
    printf ("\n Area is %f \n", area);  
}
```

Function call

Use of functions: *Area of a circle*

```
#include <stdio.h>
```

```
/* Function to compute the area of a circle */  
float myfunc (float r)  
{  
    float a;  
    a = 3.14159 * r * r;  
    return a; /* return  
}
```

Function definition

- A called function processes information that is passed to it from the calling function, and the called function may return a single value (result) to the calling function.
 - Information passed to the function via special identifiers called *arguments* or *parameters*.
 - The value is returned by the *return* statement.

```
main()  
{  
    float radius, area;  
  
    scanf ("%f", &radius);  
    area = myfunc (radius);  
    printf ("\n Area is %f \n", area);  
}
```

Function call

Defining a Function

A function definition has two parts:

- The first line
- The body of the function

General syntax:

```
return-value-type function-name ( parameter-list )  
{  
    declarations and statements  
}
```

The first line contains the return-value-type, the function name, and optionally a set of comma-separated arguments enclosed in ().

- Each argument has an associated type declaration.
- The arguments are called *formal arguments* or *formal parameters*.

Example:

```
float myfunc (float r)
```

```
int gcd (int A, int B)
```

return value
type



```
#include <stdio.h>  
  
/* Function to compute the area of a  
circle */  
float myfunc (float r)  
{  
    float a;  
    a = 3.14159 * r * r;  
    return a;  
}  
  
main()  
{  
    float radius, area;  
  
    scanf ("%f", &radius);  
    area = myfunc (radius);  
    printf ("\n Area is %f \n", area);  
}
```

Calling a function

- Called by specifying the function name and parameters in an instruction in the calling function.
- When a function is called from some other function, the corresponding arguments in the function call are called **actual arguments** or **actual parameters**.
 - The function call must include a matching actual parameter for each **formal parameter**.
 - Position of an actual parameters in the parameter list in the call must match the position of the corresponding formal parameter in the function definition.
 - The formal and actual arguments would match in their data types. Mismatches are auto-typecasted if possible.
 - The actual parameters can be expressions possibly involving other function calls (like $f(g(x)+y)$).

```
#include <stdio.h>

/* Function to compute the area of a
circle */
float myfunc (float r)
{
    float a;
    a = 3.14159 * r * r;
    return a;
}

main()
{
    float radius, area;

    scanf ("%f", &radius);
    area = myfunc (radius);
    printf ("\n Area is %f \n", area);
}
```


Function Prototypes: declaring a function

Usually, a function is defined before it is called.

- `main()` is usually the last function in the program written.
- Easy for the compiler to identify function definitions in a single scan through the file.

Some prefer to write the functions after `main()`. There may be functions that call each other.

- Must be some way to tell the compiler what is a function when compilation reaches a function call.
- Function prototypes are used for this purpose
 - **Only needed if function definition comes after a call to that function.**

- Function prototypes are usually written at the beginning of a program, ahead of any functions (including `main()`).
- Prototypes must specify the types. Parameter names are optional (ignored by the compiler).
- Examples:

```
int gcd (int , int );  
void div7 (int number);
```

 - Note the semicolon at the end of the line.
 - The parameter name, if specified, can be anything; but it is a good practice to use the same names as in the function definition.

Example:

```
#include <stdio.h>
int sum( int, int );
int main( )
{
    int x, y;
    scanf("%d%d", &x, &y);
    printf("Sum = %d\n", sum(x, y));
}

int sum (int a, int b)
{
    return a + b;
}
```

Function prototype / declaration

This program needs a function prototype or function declaration since the function call comes before the function definition.

Function call

Function definition

Return value

- A function can return a single value

Using return statement

- Like all values in C, a function return value has a type
- The return value can be assigned to a variable in the calling function

```
int main( )
{
    int x, y, s;
    scanf("%d%d", &x, &y);
    s = sum(x, y);
}
int sum (int a, int b)
{
    return a + b;
}
```

- Sometimes a function is not meant for returning anything
- Such functions are of type void

Example: A function which prints if a number is divisible by 7 or not.

```
void div7 (int n)
{
    if ((n % 7) == 0)
        printf ("%d divisible by 7", n);
    else
        printf ("%d not divisible by 7", n);
    return;
}
```

- The return type is void
- The return statement for void functions is optional at the end

The return statement

In a value-returning function, **return** does two distinct things:

- Specify the value returned by the execution of the function.
- Terminate the execution of the called function and transfer control back to the caller function.

A function can only return **one value**.

- The value can be any expression matching the return type.
- It might contain more than one return statement.

In a void function:

- "return" is optional at the end of the function body.
- "return" may also be used to terminate execution of the function explicitly **before reaching the end**.
- No return value should appear following "return".

```
void compute_and_print_itax ()
{
    float income;
    scanf ("%f", &income);
    if (income < 50000) {
        printf ("Income tax = Nil\n");
        return; /* Terminates function execution */
    }
    if (income < 60000) {
        printf ("Income tax = %f\n", 0.1*(income-50000));
        return; /* Terminates function execution */
    }
    if (income < 150000) {
        printf ("Income tax = %f\n",0.2*(income-60000)+1000);
        return ; /* Terminates function execution */
    }
    printf ("Income tax = %f\n",0.3*(income-150000)+19000);
}
```

Another Example: What is happening here?

```
int main()
{
    int numb, flag, j=3;
    scanf("%d",&numb);
    while (j <= numb) {
        flag = prime(j);
        if (flag == 0)
            printf( "%d is prime\n", j );
        j++;
    }
    return 0;
}
```

```
int prime (int x)
{
    int i, test;
    i=2, test =0;
    while ((i <= sqrt(x)) && (test ==0))
    {
        if (x%i==0) test = 1;
        i++;
    }
    return test;
}
```

Tracking the flow of control

```
int main()
{
    int numb, flag, j=3;
    scanf("%d",&numb);
    printf("numb = %d \n",numb);
    while (j <= numb)
    {
        printf("\nMain, j = %d\n",j);
        flag = prime(j);
        printf("Main, flag = %d\n",flag);

        if (flag == 0) printf("%d is prime\n",j);
        j++;
    }
    return 0;
}
```

```
int prime(int x)
{
    int i, test;
    i = 2; test = 0;

    printf("In function, x = %d \n",x);
    while ((i <= sqrt(x)) && (test == 0))
    {
        if (x%i == 0) test = 1;
        i++;
    }
    printf("Returning, test = %d \n",test);

    return test;
}
```

PROGRAM OUTPUT

5
numb = 5

Main, j = 3
In function, x = 3
Returning, test = 0
Main, flag = 0
3 is prime

Main, j = 4
In function, x = 4
Returning, test = 1
Main, flag = 1

Main, j = 5
In function, x = 5
Returning, test = 0
Main, flag = 0
5 is prime

Nested Functions

A function cannot be defined within another function. It can be called within another function.

- All function definitions must be disjoint.

Nested function calls are allowed.

- A calls B, B calls C, C calls D, etc.
- The function called last will be the first to return.

A function can also call itself, either directly or in a cycle.

- A calls B, B calls C, C calls back A.
- Called **recursive call** or **recursion**.

Example: main() calls ncr(), ncr() calls fact()

```
#include <stdio.h>

int ncr (int n, int r);
int fact (int n);

main()
{
    int i, m, n, sum=0;
    scanf ("%d %d", &m, &n);

    for (i=1; i<=m; i+=2)
        sum = sum + ncr(n, i) ;

    printf ("Result: %d \n", sum);
}
```

```
int ncr (int n, int r)
{
    return (fact(n)/fact(r)/fact(n-r));
}

int fact (int n)
{
    int i, temp=1;
    for (i=1; i<=n; i++)
        temp *= i;
    return (temp);
}
```


Local variables

A function can define its own local variables.

The local variables are known (can be accessed) only within the function in which they are declared.

- Local variables cease to exist when the function returns.
- Each execution of the function uses a new set of local variables.

Parameters are also local.

```
/* Find the area of a circle with
diameter d */
double circle_area (double d)
{
    double radius, area;
    radius = d/2.0;
    area = 3.14*radius*radius;
    return (area);
}
```

parameter

local variables

Revisiting nCr

```
int ncr (int n, int r)
{
    return (fact(n) / fact(r) / fact(n-r));
}

int fact (int n)
{
    int i, temp=1;
    for (i=1; i<=n; i++)
        temp *= i;
    return (temp);
}
```

The n in ncr() and
the n in fact() are
different

Scope of a variable

- Part of the program from which the value of the variable can be used (seen).
- Scope of a variable - Within the block in which the variable is defined.
 - **Block = group of statements enclosed within { }**
- Local variable – scope is usually the function in which it is defined.
 - **So two local variables of two functions can have the same name, but they are different variables**
- Global variables – declared outside all functions (even main).
 - **Scope is entire program by default, but can be hidden in a block if local variable of same name defined**
 - **You are encouraged to avoid global variables**

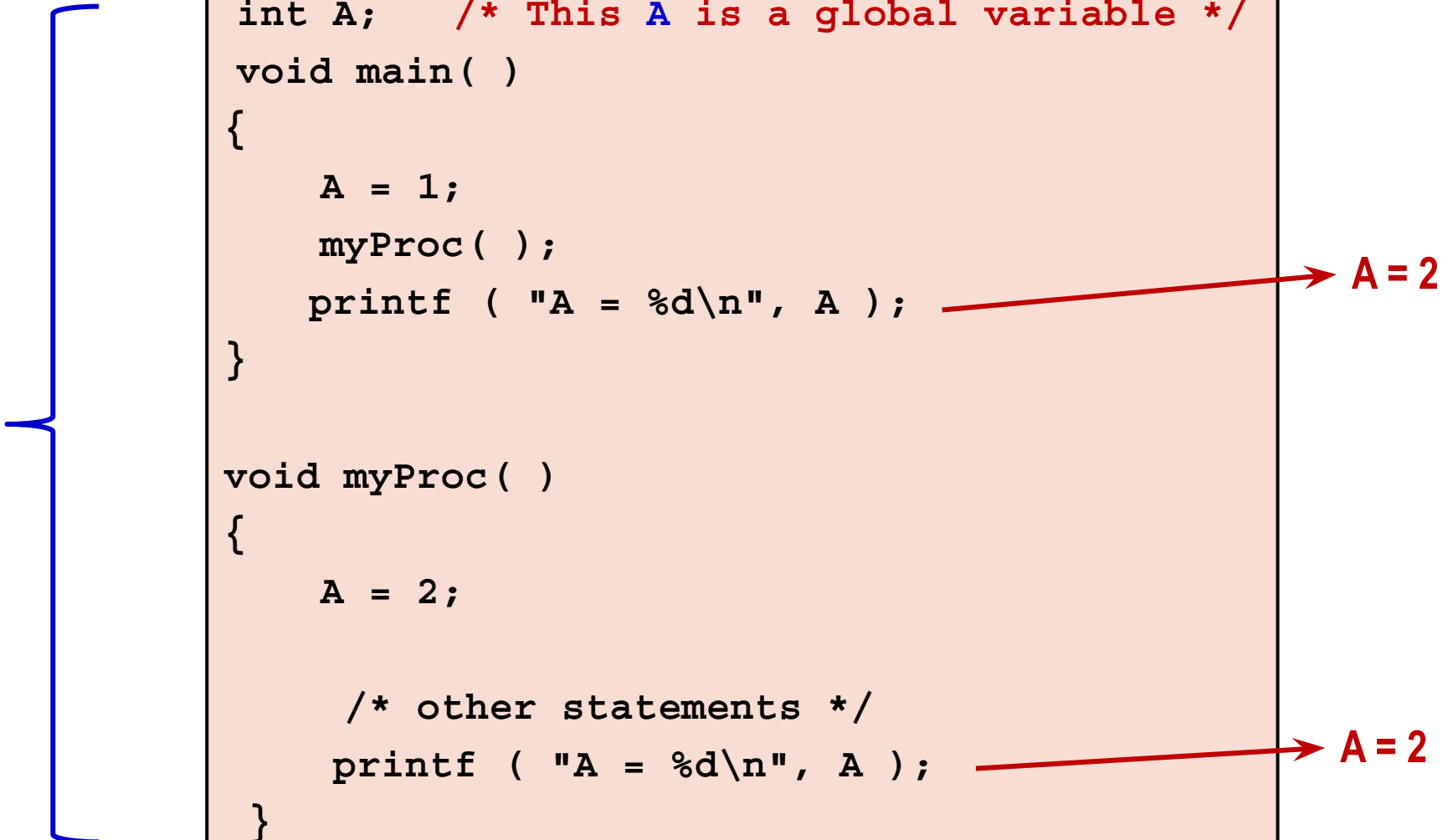
What happens here?

Scope of
global A

```
#include <stdio.h>
int A;  /* This A is a global variable */
void main( )
{
    A = 1;
    myProc( );
    printf ( "A = %d\n", A );
}

void myProc( )
{
    A = 2;

    /* other statements */
    printf ( "A = %d\n", A );
}
```



Local Scope replaces Global Scope

Scope of
global A

Scope of
local A

```
#include <stdio.h>
int A; /* This A is a global variable */
void main( )
{
    A = 1;
    myProc( );
    printf ( "A = %d\n", A ); → A=1
}

void myProc( )
{
    int A = 2; /* This A is a local variable */

    /* other statements */
    /* within this function, A refers to the local A */

    printf ( "A = %d\n", A ); → A=2
}
```

Parameter Passing

When the function is called, the **value** of the actual parameter is **copied** to the formal parameter

parameter passing



The diagram illustrates the process of parameter passing. It features two code blocks in light brown boxes. The left block shows the `main` function with a call to `area(radius)`. A red arrow points from the `radius` argument to the `double r` parameter of the `area` function. The right block shows the `area` function definition. A red arrow points from the text 'parameter passing' to the `double r` parameter. The text 'parameter passing' is centered above the two boxes.

```
int main ()
{
    . . .
    double radius, a;
    . . .
    a = area(radius);
    . . .
}
```

```
double area (double r)
{
    return (3.14*r*r);
}
```

Parameter Passing by Value in C

- Used when invoking functions

Call by value / parameter passing by value

- Called function gets a copy of the value of the actual argument passed to the function.
- **Execution of the function does not change the actual arguments.**
 - All changes to a parameter done inside the function are done on the copy.
 - The copy is removed when the control returns to the caller function.
 - The value of the actual parameter in the caller function is not affected.
- The arguments passed may very well be **expressions** (example: fact(n-r)).

Call by reference

- Passes the **address** of the original argument to a called function.
- Execution of the function may affect the original argument in the calling function.
- Not directly supported in C, but supported in some other languages like C++.
- In C, you can pass *copies* of addresses to get the desired effect.

Parameter passing and return: 1

```
int main()
{
    int a=10, b;
    printf ("Initially a = %d\n", a);
    b = change (a);
    printf ("a = %d, b = %d\n", a, b);
    return 0;
}

int change (int x)
{
    printf ("Before x = %d\n",x);
    x = x / 2;
    printf ("After x = %d\n", x);
    return (x);
}
```

Output

```
Initially a = 10
Before x = 10
After x = 5
a = 10, b = 5
```


Parameter passing and return: 2

```
int main()
{
    int x=10, b;
    printf ("M: Initially x = %d\n", x);
    b = change (x);
    printf ("M: x = %d, b = %d\n", x, b);
    return 0;
}

int change (int x)
{
    printf ("F: Before x = %d\n", x);
    x = x / 2;
    printf ("F: After x = %d\n", x);
    return (x);
}
```

Output

M: Initially x = 10

F: Before x = 10

F: After x = 5

M: x = 10, b = 5

Parameter passing and return: 3

```
int main()
{
    int x=10, y=5;
    printf ("M1: x = %d, y = %d\n", x, y);
    interchange (x, y);
    printf ("M2: x = %d, y = %d\n", x, y);
    return 0;
}

void interchange (int x, int y)
{
    int temp;
    printf ("F1: x = %d, y = %d\n", x, y);
    temp= x; x = y; y = temp;
    printf ("F2: x = %d, y = %d\n", x, y);
}
```

Output

M1: x = 10, y = 5

F1: x = 10, y = 5

F2: x = 5, y = 10

M2: x = 10, y = 5

How do we write an interchange function?
(will see later)

Header files and preprocessor



Header Files

Header files:

- Contain function declarations / prototypes for library functions.
- `<stdlib.h>` , `<math.h>` , etc.
- Load with: `#include <filename>`
- Example: `#include <math.h>`
- The function definitions of library functions are in the actual libraries (e.g., math library).

We can also create custom header files:

- Create file(s) with function prototypes / declarations.
- Save as `filename.h` (say).
- Load in other files with `#include "filename.h"`

C preprocessor

- Statements starting with # are handled by the C preprocessor
- May be done by the compiler or by a separate program
- Preprocessing is done before the actual compilation process begins

- The C preprocessor is basically a text substitution tool
- For instance, #include command is replaced by the contents of the specified header file
- Such commands are called **preprocessor directives**

- We will study another preprocessor directive: #define
- There are more such directives – see book

#define: Macro definition

Preprocessor directive in the following form:

```
#define string1 string2
```

- Replaces string1 by string2 wherever it occurs before compilation. For example,

```
#define PI 3.1415926
```

```
#define PI 3.1415926
main()
{
    float r = 4.0, area;
    area = PI * r * r;
}
```

macro pre-processing



```
main()
{
    float r = 4.0, area;
    area = 3.1415926 * r * r;
}
```

#define with arguments

#define statement may be used with arguments.

- Example: `#define sqr(x) x*x`

- How will macro substitution be carried out?

<code>r = sqr(a) + sqr(30);</code>	<input type="checkbox"/>	<code>r = a*a + 30*30;</code>
<code>r = sqr(a+b);</code>	<input type="checkbox"/>	<code>r = a+b*a+b;</code>

- The macro should better be written as:

<code>#define sqr(x) (x)*(x)</code>	
<code>r = sqr(a+b);</code>	<input type="checkbox"/> <code>r = (a+b)*(a+b);</code>

WRONG?



- Is this still correct?

<code>r = c / sqr(a+b);</code>	<input type="checkbox"/> <code>r = c / (a+b)*(a+b);</code>
--------------------------------	--

Macros are not functions. They are literally substituted without evaluation.

Practice Problems

No separate problems needed.

- Look at everything that you did so far, such as finding sum, finding average, counting something, checking if something is true or false (“ Is there an element in array A such that....) etc. in which the final answer is one thing only (like sum, count, 0 or 1,...).
- Then for each of them, rather than doing it inside main (as you have done so far), write it as a function with appropriate parameters, and call from main() to find and print.

- Normally, read and print everything from main(). Do not read or print anything inside the function. This will give you better practice.
- However, you can write simple functions for printing an array.