CS100
Introduction to Programming

Lecture 5. Functions

Objectives

- At the end of this lecture, you will be able to understand and use the following:
  - Function Definition
  - Function Prototypes
  - Function Flow
  - Scope of Variables
  - Parameter Passing: Call by Value
  - Functional Decomposition
  - Placing Functions in Different Files

Function Definition

- A **function** is a self-contained unit of code to carry out a specific task, e.g. printf(), sqrt().
- A function consists of
  - a header
  - an opening curly brace
  - a body
  - a closing curly brace

Function header

{ Open brace

Function body

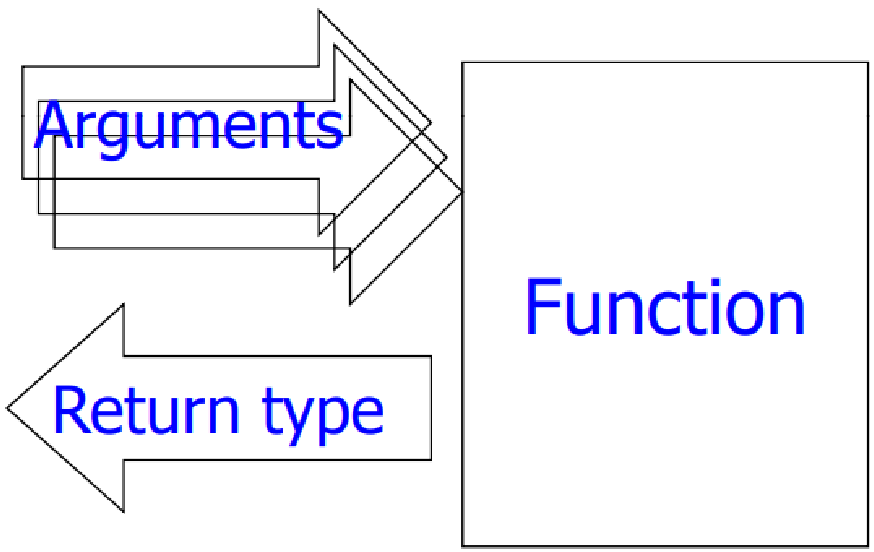
} Close brace

```
float findMaximum(float x, float y)
{
   // variable declaration
   float maxnum;

   // find the max number
   if (x >= y)
      maxnum = x;
   else
      maxnum = y;

   return maxnum;
}
```

Function Header

Format:

Type Function_name(Parameter_list)



Parameter List

- Parameters define the **data** passed into the function.
- Each parameter has a **data type** (e.g. int, char, etc.)
- A function can have no parameter, one parameter or many parameters.

  type parameterName[, type parameterName]

- The data type for each parameter must be declared.
- The function assumes that these input will be supplied to the function when it is being called.

Return Type

Return Type is the data type returned from the function. It can be int, float, char, void, or nothing.

- **int** – the function will return a value of type int

```
int successor(int num)
{
   return num + 1; /* has a return statement */
}
```

- **float** – the function will return a value of type float
- **void** – the function will not return any value.

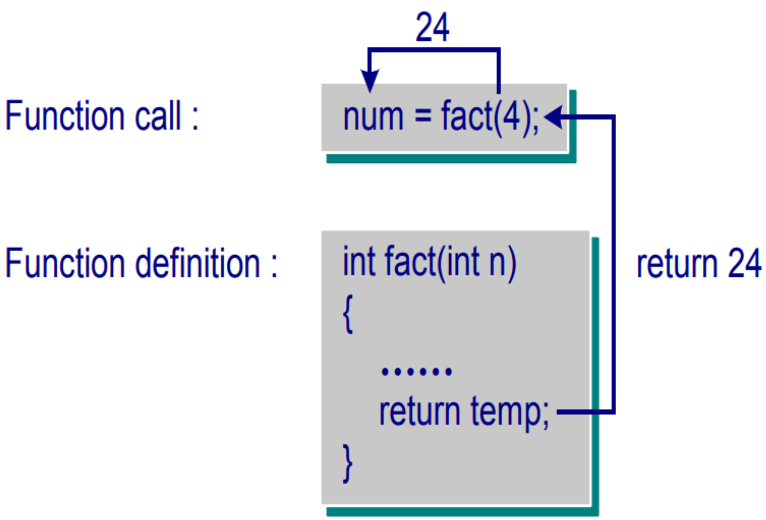
```
void hello_n_times(int n)
{
   int count;
   for (count = 0; count < n; count++)
      printf("hello\n");
   /* no return statement */
}
```

- **nothing** – if defined with no type, the default type is int

The return statement

- It may appear in **any place** and in **more than one place** inside the function body.

```
int fact(int n)
{
   int temp = 1;
   if (n < 0) {
      printf("error: must be positive\n");
      return 0;
   } else if (n == 0) {
      return 1;
   } else {
      for (; n > 0; n--)
         temp *= n;
   }
   return temp;
}
```



Output:

Enter a positive number: ***4***

The factorial of 4 is 24

Function: Examples

```
char findGrade(float marks)
{
   char grade;
   if (marks >= 50)
      grade = 'P';
   else
      grade = 'F';
   return grade;
}
```

```
float areaOfCircle(float radius)
{
   const float pi = 3.14;
   float area = pi*radius*radius;
   return area;
}
```

It's only an example, not a real policy.

Function Prototypes

- This is to declare a function. A function declaration is called a **function prototype**. It provides the information about
  - the **type** of the function
  - the **name** of the function
  - the **number and types of the arguments**
- The declaration may be the same as the function header terminated by a semicolon. For exampe:

  void hello_n_times(int n);

- Or the function is declared without giving the parameter names:

  double distance(double, double);

- The declaration has to be done before the function is called:
  - before the main() header
  - inside the main() body or
  - inside any function which uses it

Function Prototypes: Examples

```
#include <stdio.h>
// before the main()
// function prototype
int factorial(int n);

main()
{
....
}

/* function definition */
int factorial(int n)
{
....
}
```

```
#include <stdio.h>
// inside the main()
main()
{
   // function prototype
   int factorial(int);
   // then use the function factorial()
   ....
}

/* function definition */
int factorial(int n)
{
....
}
```

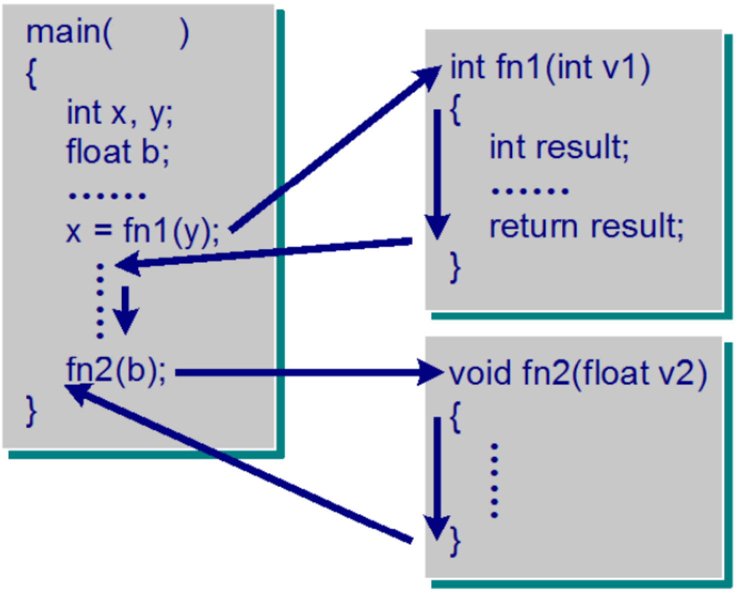
Function Flow

A function call causes the function to be executed.

A function call has the following format:

Function_name(Argument_list);

```
#include <stdio.h>
void hello();  // function prototype

main()
{
    hello();  // function call
    return 0;
}

int hello()  // function definition
{
    printf("hello\n");
}
```



Function Flow: Examples

```c
#include <stdio.h>
char findGrade(float);

main() {
    char answer;
    answer = findGrade(68.5);
    printf("Grade is %c", answer);
    return 0;
}

char findGrade(float marks) {
    ...
}
```

```c
#include <stdio.h>
float areaOfCircle(float);

main() {
    float answer;
    answer = areaOfCircle(2.5);
    printf("Area is %.1f", answer);
    return 0;
}

float areaOfCircle(float radius) {
    ...
}
```

Scope of Variables in a Function

- Variables declared in a function is **ONLY** visible within that function.
- In the example below, variables **radius**, **pi** and **area** are **NOT** visible outside this function.

```
float areaOfCircle(float radius)
{
  const float pi = 3.14;
  float area = pi*radius*radius;
  return area;
}
```

Scope of Variables

```
#include <stdio.h>
int global_var = 5;       // global variable
int fn1(int, int);
float expn(float);
main() {
   char reply;            // local variables - these two variables are
   int num;               // only known inside main()
   ...
}

int fn1(int x, int y) {  // local x, y - formal parameters
   ...                    // only known inside this function

   float fnum;            // local - these two variables are known
   int temp;              // in this function only
   global_var += 10;
   ...
}

float expn(float n) {
   float temp;            // local - this variable is known in expn()
   ...
}
```

Parameter Passing: Call by Value

Communications between a function and the calling body is done through **arguments** and the **return value** of a function.

```
#include <stdio.h>
int add1(int);

main()
{
    int num = 5;
    num = add1(num); // num – argument
    printf("The value of num is: %d", num);
    return 0;
}

int add1(int value) // value – parameter
{
    return ++value;
}
```

Output:

The value of num is: 6

Parameter Passing: Example

```
#include <stdio.h>
#include <math.h>
double distance(double, double);

main(void)
{
    double dist;
    double x=2.0, y=4.5, a=3.0, b=5.5;
    dist = distance(2.0, 4.5); // 2.0, 4.5 – arguments
    printf("The dist is %f\n", dist);
    dist = distance(x*y, a*b); // x*y, a*b – arguments
    printf("The dist is %f\n", dist);
    return 0;
}

double distance(double x, double y) // x, y – parameters
{
    return sqrt(x*x + y*y);
}
```

Output:

The dist is 4.924429

The dist is 18.794946

Function Calling Another Function

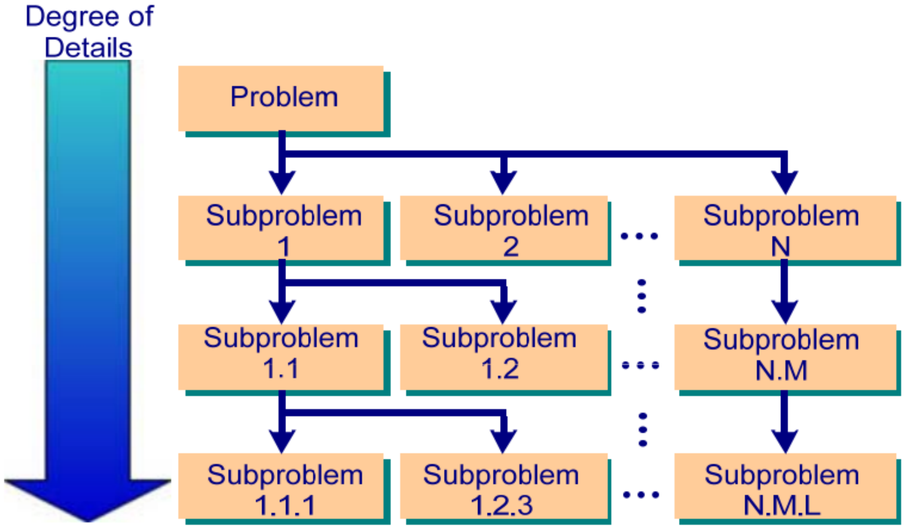
```c
#include <stdio.h>
int max3(int, int, int); // function prototypes
int max2(int, int);

int main(void)
{
    int x, y, z;
    printf("Input 3 integers: ");
    scanf("%d %d %d", &x, &y, &z);
    printf("Maximum of the 3 is %d\n", max3(x, y, z));
    return 0;
}

int max3(int i, int j, int k) {
    printf("Find the max in %d, %d and %d\n", i, j, k);
    return max2(max2(i,j), max2(j, k));
}

int max2(int h, int k) {
    printf("Find the max of %d and %d\n", h, k);
    return h > k ? h : k;
}
```

Output:

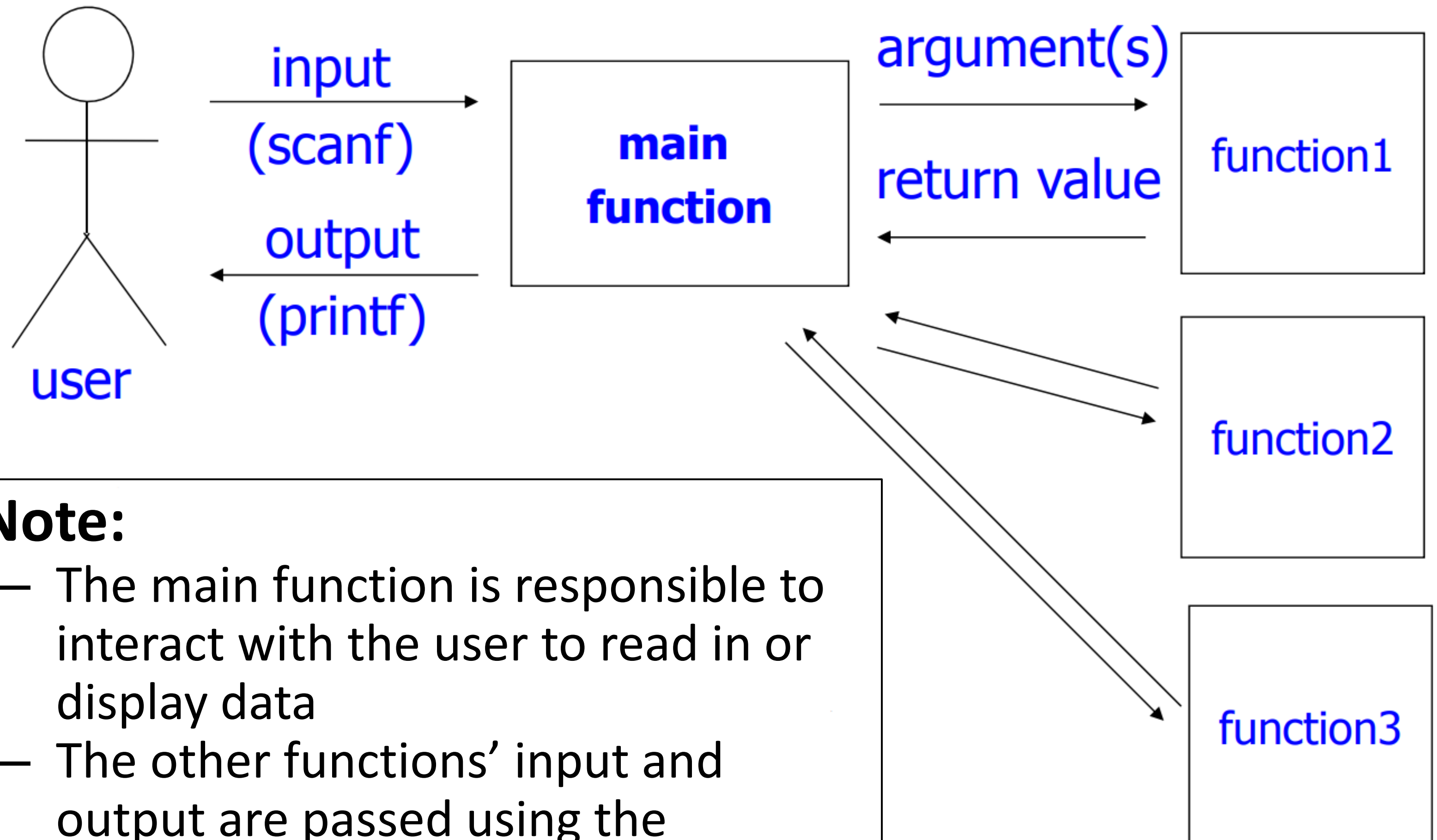
Input 3 integers: ***7 4 9***
Find the max in 7, 4 and 9
Find the max of 7 and 4
Find the max of 4 and 9
Find the max of 7 and 9
Maximum of the 3 is 9

Functional Decomposition

- **Functional decomposition** basically means ***Stepwise Refinement***.
- In C, functional decomposition starts with the high-level description of the program and decomposes the program (the main() function) into successively smaller functions until we arrive at suitably sized **functions**.
- Then design the code for the individual functions using stepwise refinement. At each level, we are only concerned with **what** the lower level functions will do, but not **how**.



Functional Decomposition



- **Note:**
  - The main function is responsible to interact with the user to read in or display data
  - The other functions’ input and output are passed using the arguments and return type via the main function

Functional Decomposition

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

main(void)
{
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
} /* end.  line 2000 */
```

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

main(void)
{
.....
} /* line 20 */

float f1(float h)
{
.....
} /* line 55*/
......
void f18(void)
{
......
} /* line 1560 */
```

Functional Decomposition

- What is a suitably sized function?
  - Smaller functions are **easier to understand** (according to research into human psychology)
  - Smaller functions promote software **reusability**.
  - If functions are very small, we need many of them.
  - Function size should be no longer than a page. Better yet, a function should be no longer than half a page.
- Why do we need functional decomposition?
  - Program better structured
  - Program easier to understand
  - Program easier to modify
  - Shorter program

Placing Functions into Different Files

- Why placing parts of the program in different files:
  - The functions in different files can be used by more than one program (**reusability**).
  - Only the files that are changed need be re-compiled.
- How to place functions in different files?
- For example, the code of a program is placed into two files:
  - One file contains the main(). The main() body calls function1() and function2().
  - These two functions are in another file. There are two constants defined by #define and used by the program, CONST1 and CONST2.
  - The constant definitions and the function declarations are in the header file called def.h.

file1: **mainF.c**

```
#include <stdio.h>
#include “def.h” // double quotes mean the file is in the current directory
int main(void)
{
    ...
    count = function1(h, k);
    function2(&h, &k);
    ...
}
```

file2: **support.c** – contains all the supporting functions

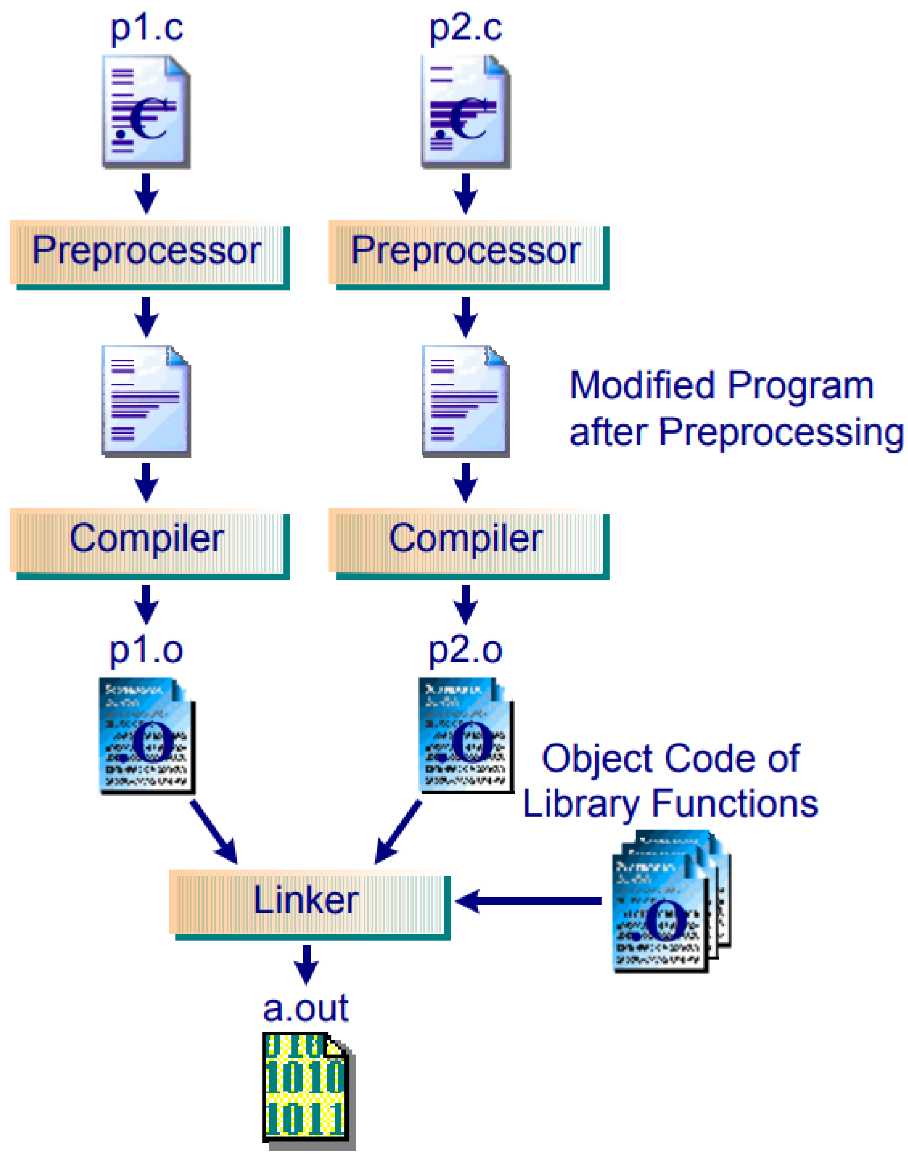
```
#include <stdio.h>
#include “def.h” // double quotes mean the file is in the current directory
int function1(int f, int g)
{
    ...
}

void function2(int *p, float *q)
{
    ...
}
```

file3: **def.h**

```
#define CONST1 80
#define CONST2 100

int function1(int, int);
void function2(int *, float *);
```

- **def.h** contains the **constant definitions** and the **function declarations** for the program.
- If we do not use a header file, these lines have to be in both file1 and file2.

Process of C Program Compilation



Compiling Program with Several Files

- Take the example above:

```
$ gcc -ansi mainF.c support.c -o mainF
```

- The compiler compiles **mainF.c** and produces **mainF.o**, then it compiles **support.c** and produces **support.o**. The linker will produce the executable file **mainF** after linking the two **.o** files and the library functions.
- If, after successful compilation, changes are made to **mainF.c** but not support.c, then

```
$ gcc -ansi mainF.c support.o -o mainF
```

or, changes are made to support.c but not mainF.c, then

```
$ gcc -ansi mainF.o support.c -o mainF
```

- In the last two situations, no re-compilation is done to the file whose **.o** file is given in the command line.

Recap

- The following concepts have been covered in this lecture:
  - Function Definition
  - Function Prototypes
  - Function Flow
  - Scope of Variables
  - Parameter Passing: Call by Value
  - Functional Decomposition
  - Placing Functions in Different Files