3
Functions

Why Learning Functions?

- With Sequential, Branching and Looping, you will be able to build programs for simple applications. However, for more complex applications, your programs may be long and certain code may be repeated in the program.
- Functions aim to group specific tasks, so that code will not be repeated. It also helps to improve your program readability and efficiency.
- In this lecture, we discuss the concepts

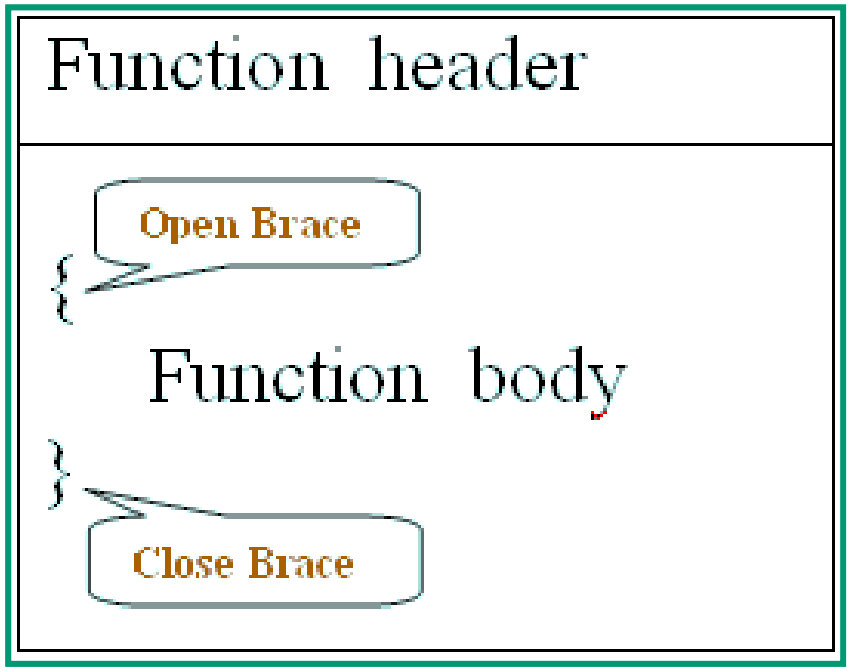
  on functions.

Functions

- **Function Definition**
- Function Prototypes
- Function Flow
- Parameter Passing: Call by Value
- Storage Scope of Variables
- Functional Decomposition

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 function body
  - a closing curly brace



Example:

```
float findMax(float x, float y)  // header
{
    // function body
    float maxnum;

    if (x >= y)
       maxnum = x;
    else
        maxnum = y;

   return maxnum;
 }
```

Function Header

Return_type Function_name (Parameter_list)

- **Function_name**
  - – specifies the name given to the function. Try to give a meaningful name to the function.
- **Parameter_list**
  - – specifies a list of parameters which contains the data that are passed in by the calling function.
- **Return_type**
  - – specifies the **type** of the data to be returned to the calling function.

Function Header: Parameter List

- Parameters define the **data** **passed into** the function.
- A function can have **no** parameter, **one** parameter or **many** parameters.

type parameterName[, type parameterName]

Example: float **findMax**(float x, float y)

- Each parameter has:
  - **parameter name**
  - **data type** (such as int, char, etc.) of the parameter
- The function assumes that these parameter inputs will be supplied to the function when they are being called.

Function Header: Return_type

- **Return Type** is the data type **returned from** the function, it can be int, float, char, void, or nothing.
- The syntax for the **return** statement is **return (expression);**
- **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**.

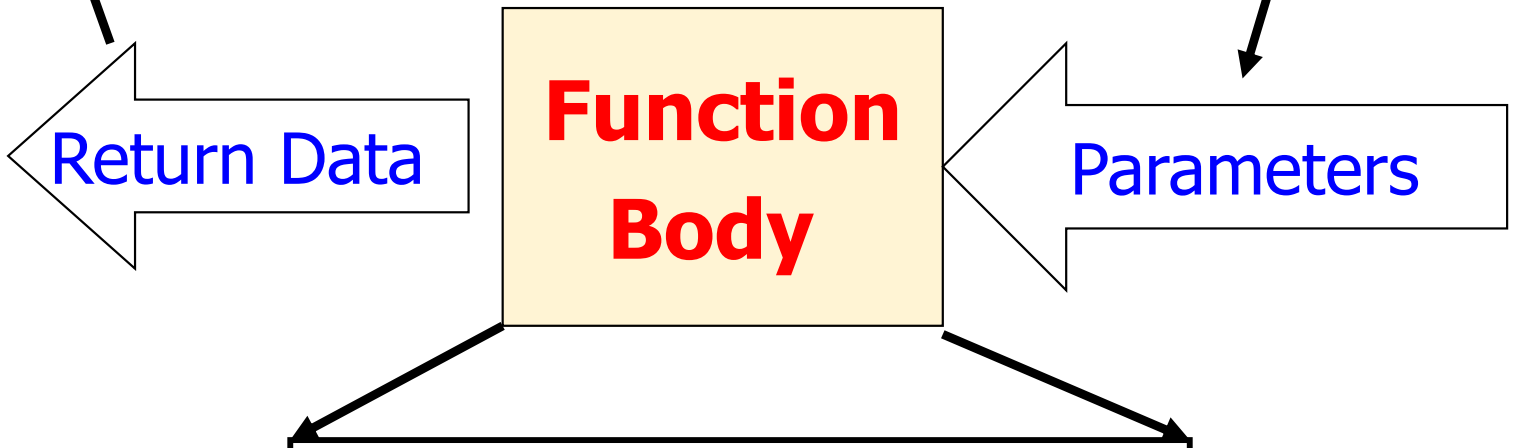
```
successor(int num)   /* i.e. int successor(int num) */
{
     return num + 1; /* has a return statement */
}
```

Function Body

Function Header:

Return_type Function_name (Parameter_list)

Example: float **findMax**(float x, float y)



consists of a group of statements which perform the logic of the function

Multiple Return Statements

- The **return** statement terminates the execution of a function and passes the control to the calling function.
- The return statement may appear in **any place** or in **more than one place** inside the function body.

```
int factorial(int n)
{
    int temp = 1;     // local variable

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

return a value

```
void hello_n_times(int n)
{
    int count;
    if (n < 0)
        return;
    else
        for (count = 0; count < n; count++)
            printf("Hello!\n");
}
```

no return value

Function: Examples

Compute Grade:

```
char findGrade(float marks) {
    char grade;   // variable

    /* function body */
    if (marks >= 50)
        grade = 'P';
    else
        grade = 'F';
    return grade;
}
```

Function: Examples

Compute Grade:

```
char findGrade(float marks) {
    char grade;   // variable

    /* function body */
    if (marks >= 50)
        grade = 'P';
    else
        grade = 'F';
    return grade;
}
```

Compute Circle Area:

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

   /* function body */
   area = pi*radius*radius;
   return area;
}
```

Functions

- Function Definition
- **Function Prototypes**
- Function Flow
- Parameter Passing: Call by Value
- Storage Scope of Variables
- Functional Decomposition

Function Prototypes

- We need to declare a function before using it in other functions.
- **Function prototype** is used to declare a function. It provides the information about
  1. the **return type** of the function
  2. the **name** of the function
  3. the **number** and **types** of the arguments
- The declaration may be the same as the function header but terminated by a **semicolon**.

Example:

```
float findMax(float x, float y) ;
```

- Two ways to declare parameters in the parameter list:

  (1) With parameter name:

  ```
  void hello_n_times(int n);
  ```

  (2) Without parameter names:

  ```
  double distance(double, double);
  ```

Function Prototypes: Where to declare it?

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

Function Prototypes: Before the main()

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

Before the main():

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

int main( )
{   int x;
    x = factorial(5);     // use factorial() here
}

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

Function Prototype: Inside the main()

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

Inside the main():

```
#include <stdio.h>

int main()
{   int x;
    int factorial(int);          // function prototype
     x = factorial(5);           // use factorial() here
    ....
}

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

Functions

- Function Definition
- Function Prototypes
- **Function Flow**
- Parameter Passing: Call by Value
- Storage Scope of Variables
- Functional Decomposition

Calling Function

- A function is executed when it is called.
- A function call has the following format:

  Function_name (Argument_list);

Example:
float num1=5, num2=10, max;
max = findMax(num1, num2);

Pass in

ARGUMENTS

num1 , num2

Function Header:

Return_type Function_name (Parameter_list)

Example: float **findMax**(float x, float y)

x=5 , y=10

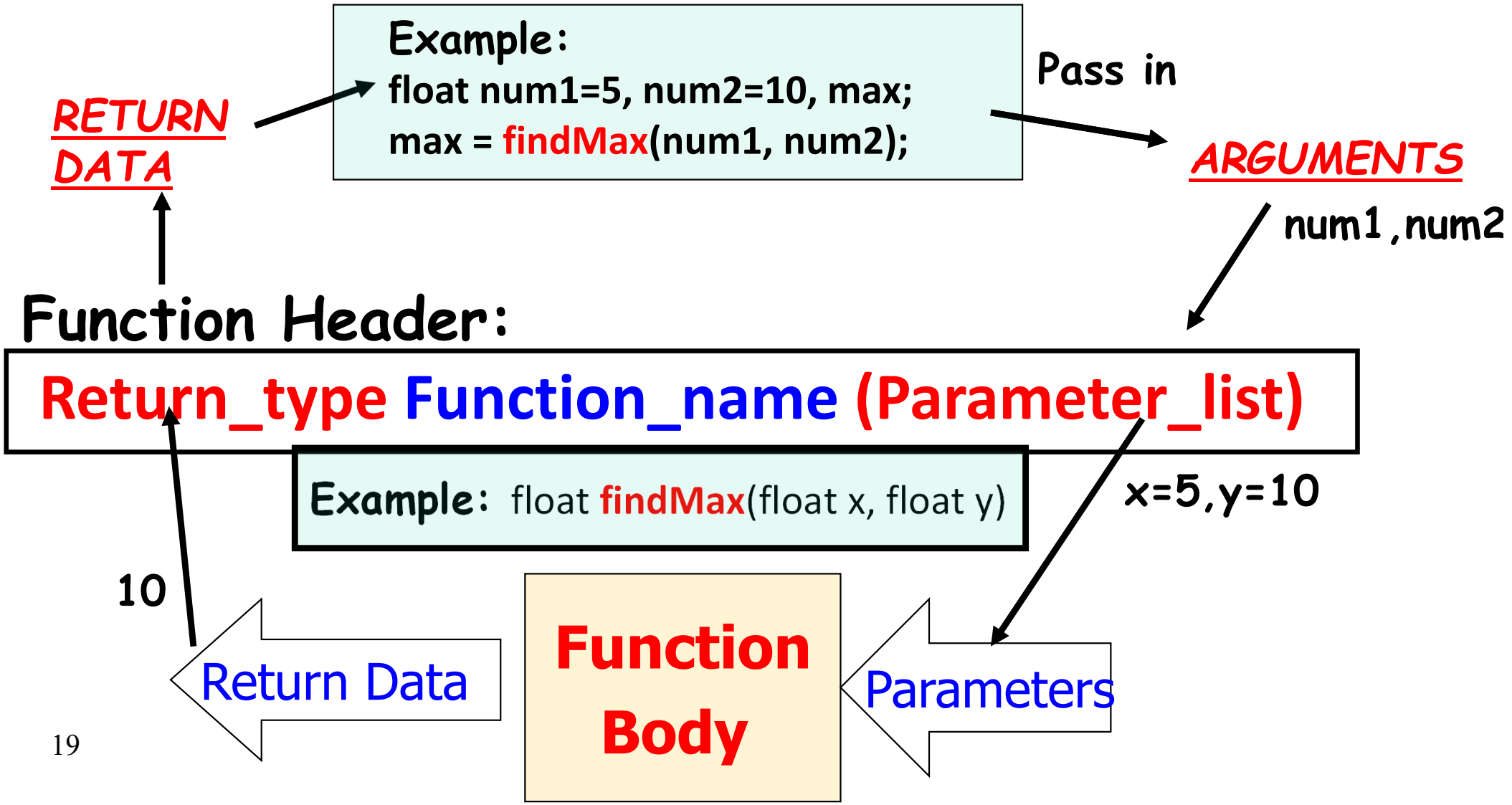
Function Body

Parameters

Returning Value

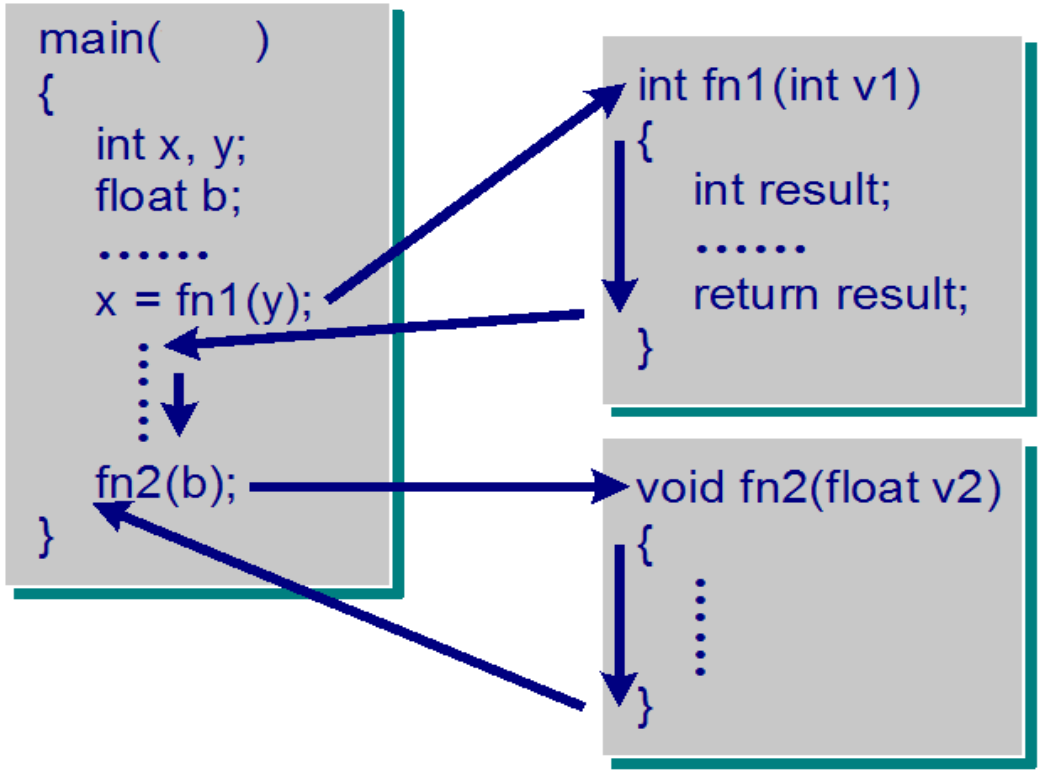
- A function is executed when it is called.
- A function call has the following format:

  Function_name (Argument_list);



Function Flow

```
#include  <stdio.h>
int fn1(int v1);        // fn prototype
void fn2(float v2);  // fn prototype
int main( )
{   …
    x=fn1(y);  // fn call - with return value
    ….
    fn2(b);     // fn call - no return value
    return 0;
}

int fn1 (int v1)     // fn definition
{      int result;   ….  return result; }

void fn2 (float v2)   // function definition
{      ….   }
```



Function Flow: Example

Compute Grade:

```
#include <stdio.h>
char findGrade(float marks);
int main( )
{
     char answer;
     answer = findGrade(68.5);
     printf("Grade is %c", answer);
     return 0;
}
char findGrade(float marks) {
    char grade;   // variable
    if (marks >= 50)
      grade = 'P';
    else
      grade = 'F';
    return grade;
}
```

Output

Grade is P

21

Function Flow: Example

Compute Circle Area:

```
#include <stdio.h>
float areaOfCircle(float);
int main( )
{
    float answer;
    answer = areaOfCircle(2.5);
    printf("Area is %.1f", answer);
    return 0;
}
float areaOfCircle(float radius) {
   const float pi = 3.14;
   float area;

   /* function body */
   area = pi*radius*radius;
   return area;
}
```

Output

Area is 19.6

Functions

- Function Definition
- Function Prototypes
- Function Flow
- **Parameter Passing: Call by Value**
- Storage Scope of Variables
- Functional Decomposition

Parameter Passing: Call by Value

- **Call by Value - Communication** between a function and the calling body is done through **arguments** and the **return value** of a function.

```
#include <stdio.h>
int add1(int);
int main( )
{
    int num = 5;
    num = add1(num);      // num – called argument
    printf("The value of num is: %d", num);
    return 0;
}
int add1(int value)       // value – called parameter
{
     value++;
     return value;
}
```

num 5 -> 6

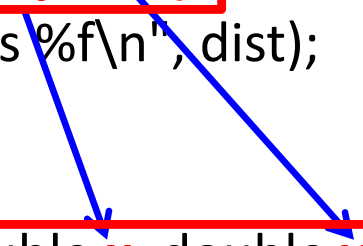
value 5 -> 6

Output
The value of num is: 6

- **Call by Reference – passing more than one value**

Parameter Passing: Example

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

int main()
{
    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

```
#include  <stdio.h>
 int max3(int, int, int);          /* function prototypes */
 int max2(int, int);
 int main()
{
      int x, y, z;
      printf("input three 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 three 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

Functions

- Function Definition
- Function Prototypes
- Function Flow
- Parameter Passing: Call by Value
- **Storage Scope of Variables**
- Functional Decomposition

Scope of Variables in a Function

- Scope of a variable
  - the section of code that can use the variable. In other words, the variable is visible in that section.
- Variables declared in a function is ONLY visible within that function. We call it **block scope**.
- Example below: variables **radius**, **pi** and **area** are **NOT** visible **outside** this function.

```
float areaOfCircle(float radius) {      // parameter – block scope
   const float pi = 3.14;               // const variable – block scope
   float area;                          // local variable – block scope
   area = pi*radius*radius;
   return area;
}
```

Local and Global Variables

- **Local variables**:
  - They are variables defined **inside** a function.
- **Global variables**:
  - They are variables defined **outside** the functions.

- Should **global variables** be used in your programs?
  - **Advantages** of using global variables:
    - simplest way of communication between functions
    - efficiency
  - **Disadvantages** of using global variables:
    - less readable program
    - more difficult to debug and modify

➢ **Strongly discouraged to use global variables** – instead you should use **parameter passing between functions** to achieve the same effect. So that **errors** will be **localized** within each function for easy debugging.

Local and Global Variables: Example

```
#include  <stdio.h>
int g_var = 5;                  // global variable –  has file scope
int fn1(int, int);
float fn2(float);

int main( ) {
     char reply;                // local - these two variables are only
     int num;                   // known inside main() function - block scope
     …
}
int fn1(int x, int y)           //  local x,y - formal parameters  are only
(                               //  known inside this function – block scope
     float fnum;                //  local - these two variables are known
     int temp;                  //  in this function only – block scope
     g_var += 10;
     …
}
float fn2(float n)  {
     float temp;                //  local and block scope
     …
}
```

Static Variables

- Static variables can be defined inside or outside a function using the **static** keyword.
  - The duration of a static variable is fixed.
  - Static variables are created at the **start** of the program and are destroyed only at the **end** of program execution. That is, they exist **throughout program execution** once they are created.
- If a **static** variable is defined and initialized, it is then **initialized once** when the storage is allocated. If a static variable is defined, but not initialized, it will be initialized to zero by the compiler.
- Static variables are very useful when we need to write functions that retain values between functions.
- We may use global variables to achieve the same purpose. However, **static variables** are preferable as they are **local variables** to the functions, and the shortcomings of global variables can be avoided.

Static Variables

```
#include <stdio.h>
void function();
int main()
{
   int i;
   for (i=0; i<3; i++)     // calling the fn three times
       function();
   return 0;
}
void function()
{
   static int static_var = 0;     /* static variable */
   int local_var = 0;              /* local variable */
   ++static_var;
   ++local_var;
   printf("Static variable: %d\n", static_var);
   printf("Local variable: %d\n", local_var);
}
```

Output

Static variable: 1
Local variable: 1
Static variable: 2
Local variable: 1
Static variable: 3
Local variable: 1

Note:

Local variable – the variable disappears after each function execution.

Static variable (like global) – the variable stays until the end of program execution.

Functions

- Function Definition
- Function Prototypes
- Function Flow
- Parameter Passing: Call by Value
- Storage Scope of Variables
- **Functional Decomposition**

Functional Decomposition

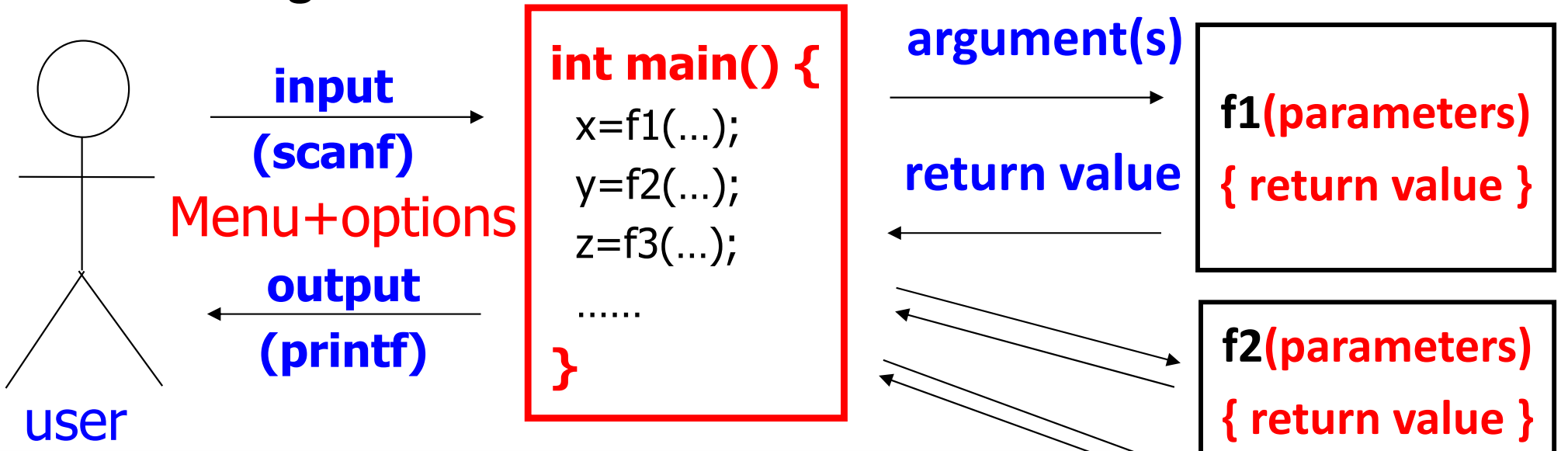
```
#include  <stdio.h>
#define  ...

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

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

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

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

- The **main()** function contains about 2000 lines of code which is difficult to read and debug.
- Functional decomposition refers to the top-down stepwise refinement technique that uses the divide-and-conquer strategy to produce smaller functions that are easier to understand. Smaller functions promote software reusability.

Functional Decomposition: Modular Design

The approach of designing programs as functional modules is called **modular design**.



- C uses **Procedural Programming** approach for program development:
  - The **main function** is responsible for interacting with user to read in or display data, and performing the actions by calling the other functions.
  - The **other functions**' input and output are passed using the arguments and return type via the main function.
- Easier to write and debug programs.

Thank you !!!