TRƯỜNG ĐẠI HỌC FPT

Modules and Functions

Review

- **Logic constructs** = Statements can be used in a program.
  - 3 basic constructs: Sequence, selection constructs (if, if…else, ?:), Iteration constructs (for/ while/ do … while)
- **Walkthrough**
  - Code are executed by ourself
  - Tasks in a walkthrough: a record of the changes that occur in the values of program variables and listing of the output, if any, produced by the program.
  - Debug program

Objective to learn Modules and Functions

After studying this section, you should be able to:

- **Code Organization**: Programs are structured, making them easier to read and maintain.
- **Reusability**: Functions allow reuse of code, reducing duplication.
- **Debugging and Testing**: Isolated functions simplify locating and fixing bugs.
- **Maintainability**: Modular code supports easy updates and changes.
- **Abstraction**: Functions hide implementation details, focusing on functionality.
- **Scalability**: Modular design enables handling larger and more complex systems.
- **Collaboration**: Teams can work on separate modules independently.

Objective

- Define a C-module or C-function?
- Explain module’s characteristics
- Implement C functions
- Use functions?
- Differentiate built-in and user-defined functions
- Explain mechanism when a function is called
- Analyze a problem into functions
- Implement a program using functions
- Understand extent and scope of a variable

Contents

1- What is a Module?

What is a Module?

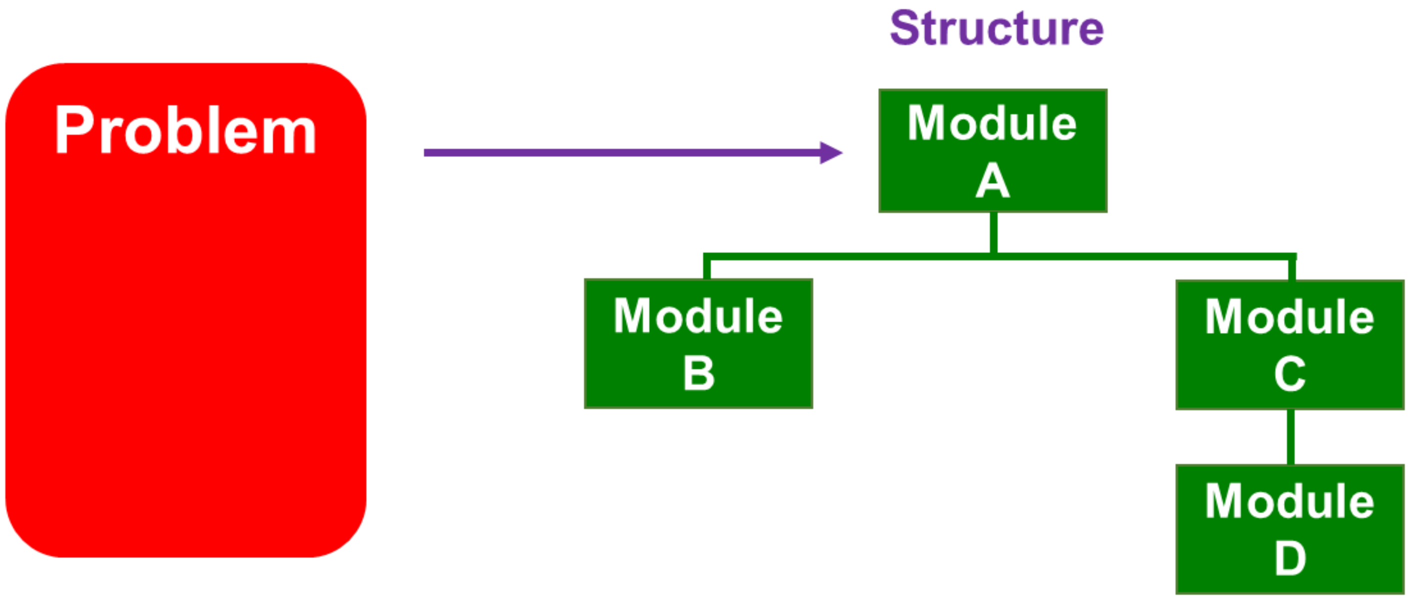
- Module is **a portion of a program** that carries out a specific small function and may be used alone or combined with other **modules** to create a program.
- Natural thinking: A large task is divided into some smaller tasks.
- Example: To cook rice we divide it into small tasks

(1) Clean the pot; (2) Measure rice; (3) Washing rice; (4) add water;

(5) Boil; (6) Keep hot 10 minutes.

Modules: Structure Design

- In designing a program, we subdivide the problem conceptually into a set of design units. We call these design units as **modules**. In subdividing the problem, we reduce the number of factors with which to deal simultaneously.



Some related modules can be put into a file (You used it – stdio.h)

Structure design - Example

- Problem: Develop a program that will accept a positive integer then sum of it’s divisors is printed out.
- Solution:

Analyze	Code	Description
	#include <stdio.h> #include <stdlib.h>	Use modules in this file
Divide the program into small tasks:	int main { int n; int s;	Declare the main module and it’s data
Task 1 - Accept n	scanf(“%d”, &n);	Use a module **scanf** in the **stdio.h**
Task 2 - s = sum of it’s divisors	s = sumDivisors (n);	Module will be implemented
Task 3 - Print out s	printf(“%d”, s);	Use a module **printf** in the **stdio.h**
Task 4 - Pause the program	system(“pause”);	Use a module **system** in the **stdlib.h**
	}	

2 - Characteristics of Modules

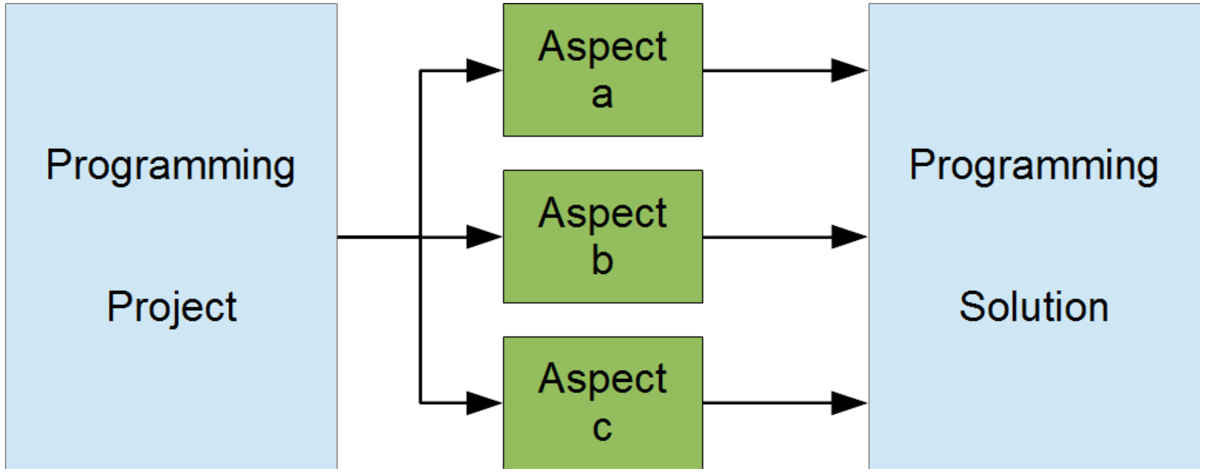
Characteristics of Modules

Characteristics	Reason
It is easy to upgrade and maintain	It contains a small group of code lines for a **SPECIFIC** task.
It can be re-used in the same program	It has a identified name (a descriptive identifier) and can be used more than one time in a program.
It can be re-used in some programs	if it is stored in an outside file (library file), it can be used in some programs.

3 - Module identifying: Hints

Modules Design Principles

- We can sub-divide a programming project in different ways.
- Select our modules so that each one focuses on a narrower aspect of the project.
- Our objective is to define a set of modules that simplifies the complexity of the original problem.



Modules Design Principles (cont.)

- **Some general guidelines for defining a module include:**
  - The module is easy to upgrade
  - The module contains a readable amount of code
  - The module may be used as part of the solution to some other problem
- **For a structured design, we stipulate that:**
  - Each module has one entry point and one exit point
  - Each module is **highly cohesive**
  - Each module exhibits **low coupling**

Module Identifying

Lowly cohesive

An input operation in a processing module is not encouraged.

➔ All the code in a module focus to the purpose of the module

```
#include <stdio.h>
int  n ;

Module for summing divisors of n
{   accept n
    sum of it's divisors
}

Module for printing out divisors of n
{   accept n
    print out it's divisors
}

int main ()
{
   access  n
}
```

High coupling

Some modules access a common data is not encouraged.

➔ All modules should be self-contained (independent)

Module identifying - Cohesion

- Cohesion is a measure of the focus within a module.
- A module performs a single task → highly cohesive.
- A module performs a collection of unrelated tasks → low cohesion.
- In designing a cohesive module, we ask whether a certain task belongs:
  - The reason to include it is that it is related to the other tasks in some particular manner.
  - A reason to exclude it is that it is unrelated to the other tasks.
- How to identify modules? → If you still use a verb to describe a task then a module is identified.

Module identifying - Degrees of cohesion

Low cohesion → generally unacceptable

- **Coincidental** - unrelated tasks

  → This module is not enough small → Separate smaller tasks in this task.

- **Logical** - This module contains some related tasks of which only one is performed

  → Separate them into separate smaller module, each smaller module for a choice.

- **Temporal** - multiple logically unrelated tasks that are only temporally related

  → Separate them into separate smaller module although they are temporal

One module for two tasks:
- Sum divisors of the integer n
- Print out divisors of the integer n

→

In the case of the operation for summing of n is not used, this module can not be applied.

Module identifying - Degrees of cohesion

High cohesion - generally acceptable

- **Communicational** - the tasks share the same data

  → All tasks are carried out each time.

- **Sequential** - multiple tasks in a sequentially dependent relationship.

  → Output of one task serves as input to another task - the module identifier suggests an assembly line.

- **Functional** - performs a single specific task

  → The module identifier suggests a precise verb phrase

Some modules share the common data can be accepted if they perform their tasks sequentially.

Module identifying - High Coupling

High Coupling - It's not advisable

- **Coupling** is a measure of the **degree of interrelatedness** of a module to its referring module(s).
- A module is low in coupling if it performs its tasks on its own.
- A module is highly coupled if it shares that performance with some other module including the referring module → **It should not be used**.
- In designing for low coupling, we ask what kind of data to avoid passing to the module.

Module identifying - Coupling classification

- The data classifications include:

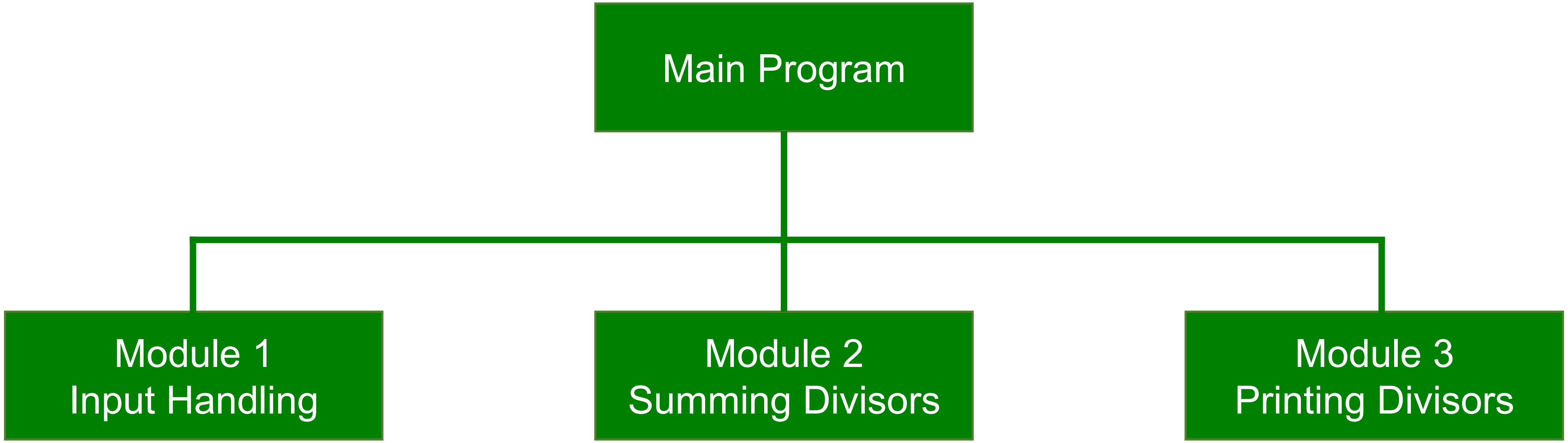
high

- **Data** - used by the module but not to control its execution → ***Data in/dependant***
- **Control** - controls the execution of the module → ***Control in/dependent***
- **External** - part of an environment external to the module that controls its execution
- **Common** - part of a global set of data
- **Content** - accesses the internals of another module

low

Example - High Cohesion & Low Coupling

- Problem: Develop a program that will accept a positive integer then sum of it’s divisors is printed out.
- Modules design:



Example - High Cohesion & Low Coupling (cont.)

- **Module 1: Input Handling**

  → This module is responsible for reading the value of **n** from the user.

- **Module 2: Summing Divisors**

  → This module calculates the sum of all divisors of a given integer.

- **Module 3: Printing Divisors**

  → This module prints all divisors of a given integer.

- **Main Program**

  → This module coordinates the other modules. It uses input, calculates the sum of divisors, and prints the divisors.

Example - High Cohesion & Low Coupling (cont.)

module_demo.c

```c
#include <stdio.h>

// Module 1: Input Handling
int getInput() {
    int n;
    printf("Enter a number: ");
    scanf("%d", &n);
    return n;
}

// Module 2: Summing Divisors
int sumOfDivisors(int n) {
    int sum = 0;
    for (int i = 1; i <= n; i++) {
        if (n % i == 0) {
            sum += i;
        }
    }
    return sum;
}
```

```c
// Module 3: Printing Divisors
void printDivisors(int n) {
    printf("Divisors of %d are: ", n);
    for (int i = 1; i <= n; i++) {
        if (n % i == 0) {
            printf("%d ", i);
        }
    }
    printf("\n");
}

// Main Program
int main() {
    // Input
    int n = getInput();

    // Processing
    int sum = sumOfDivisors(n);

    // Output
    printf("The sum of divisors of %d is: %d\n", n, sum);
    printDivisors(n);

    return 0;
}
```

Output

D:\MonHoc\PRF192\ThucHanh\module_demo.exe

```
Enter a number: 12
The sum of divisors of 12 is: 28
Divisors of 12 are: 1 2 3 4 6 12

--------------------------------
Process exited after 19.82 seconds with return value 0
Press any key to continue . . .
```

Example Explain - High Cohesion & Low Coupling

- **High Cohesion**: Each module has a single, well-defined purpose
  - **getInput**: Handles user input.
  - **sumOfDivisors**: Calculates the sum of divisors.
  - **printDivisors**: Displays all divisors.
- **Low Coupling**:
  - Each module operates independently and communicates only via function calls and return values.
  - No global variables are shared between modules.
- **Reusability**: Each module can be reused in other programs without modification
- **Ease of Maintenance**: Modifications to one module (e.g., changing the input method) won't affect others.

Module identifying : How to create them?

- If you still use a verb to describe a task then a module is identified.
- In practice:
  - List all of the tasks (verbs) that the program should perform to solve this problem.
  - Identify the modules (verbs) for the problem structure
  - Check that each module is **high in cohesion** (each basic task is a module)
  - Check that each module is **low in coupling** (modules are independent)

4 - C-Functions and Modules

C-Functions and Modules

- In C, we represent a module by a function.
- A function is a block of reusable code designed to perform a specific task. A function may receive data and may return a value.
- Example:
  - Print out divisors of the integer **n** → **n** is data is accepted by the function and no value is returned.

    n =12 → Print out values: 1, 2, 3, 4, 6, 12
  - Sum of divisors of the integer **n** → **n** is data is accepted by the function and a value is returned.

    n =12 → 28 is the return value
- The description of the internal logic of a function as the function's definition.

Function Definitions

- Function Syntax:

What is the result of the task

Function header

```
returnType   functionName ([ Type   param1, Type  param2, …])
{
      <code>
     [return value; ]
}
```

Function body

How does this task do?

What is the name of the task?

To do this task, what are necessary data?

Function Definitions - Example

return DataType

Function Identifier

Parameters

```
double average (int a, int b, int c)
{
        double result;
        result = (a+b+c)/3. ;
        return result;
}
```

Body:
Logical construct

Review:
(a+b+c)/3 → integer

Review:
(a+b+c)/3.0 → double

Review
3.0 and 3. are the same
3.3500 = 3.35
3.30 = 3.3
3.0 = 3.

Function Definitions - Example (cont.)

function_syntax.c

```c
#include <stdio.h>

/*
* Funtion: average
* ------------------------------------
*   Computes  the average of 3 integers
*   parameters: a, b, c are integer numbers
*   returns: the average of 3 integers
*/
double average(int a, int b, int c)
{
    double result;
    result = (a+b+c)/3.;
    return result;
}

int main(){
    // Implemeting function
    double avgNumbers = average(5, 8, 7);
    printf("The average of the numbers: %lf", avgNumbers);

    return 0;
}
```

D:\MonHoc\PRF192\ThucHanh\function_syntax.exe

```
The average of the numbers: 6.666667
--------------------------------
Process exited after 0.0824 seconds with
return value 0
Press any key to continue . . .
```

Function syntax: void function

- To identify a function that does not return any value, we specify **void** keyword for the return data type and exclude any expression from the return statement:
  - Alternatively, we can omit the **return** statement altogether.
- A function that does not return a value is called a subroutine or procedure in other languages.
- Syntax:

```
void functionName ([ Type param1, Type param2, …. ])
{
        // Statements
}
```

void function - Example

```c
/* Example 1: Count down an integer n to 1
 * countDown.c
 */

#include <stdio.h>

// void function with parameter
void countDown(int n){
    while (n > 0)
    {
        printf("%d ", n);
        n--;
    }
}

int main(){
    // Inplemeting function
    countDown(10);

    return 0;
}
```

```
10 9 8 7 6 5 4 3 2 1
--------------------------------
Process exited after 0.09705 seconds with return value 0
Press any key to continue . . .
```

```c
/* Example 2: Alphabet A - Z
 * alphabet.c
 */

#include <stdio.h>

// void function without parameter
void alphabet()
{
    char letter = 'A';

    do {
        printf("%d ", letter);
        letter++;
    } while (letter != 'Z');
}

int main(){
    // Inplemeting function
    alphabet();

    return 0;
}
```

```
65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89
--------------------------------
Process exited after 0.08573 seconds with return value 0
Press any key to continue . . .
```

main function

- The **main()** function is the function to which the operating system transfers control at the start of execution.
- **main()** returns a value to the operating system upon completing execution. C compilers assume an **int** where we don't provide a return data type.
- The operating system typically accepts a value of **0** as an indicator of success and may use this value to control subsequent execution of other programs.
- **main() is the entry point of a C- program**

```c
#include <stdio.h>

// definition other functions

int main(){
    // statements

    return 0;
}
```

5 - How to implement a function?

How to implement a function?

State the task clearly: Verb + nouns (Objects)

Verbs:
Find,
Compute,
Count,
Check

Others

int | long | ...
void

```
functionName(Type param1,  Type param2)
{
   <steps of  processing>
   return [ Expression];
}
```

Give values to the parameters;
Carry out the work with yourself;
Write down steps;
Translate steps to C;

A task is described clearly if the receiver does not need to ask any thing

Evaluate the functions

This function contains a sub-task → low cohesive.

```c
#include <stdio.h>
// Test whether an integer is a prime or not
int isPrime(int n){
    printf("Input n=");
    scanf("%d", &n);
    int i;
    for(i=2; i*i<=n; i++){
        if(n%i==0){
            return 0;
        }
    }
    return 1;
}

int main(){

    int n;
    if(isPrime(n)==1){
        printf("n is prime.\n");
    }else{
        printf("n is not prime.");
    }
    return 0;
}
```

fix →

Better

```c
#include <stdio.h>
// Test whether an integer is a prime or not
int isPrime(int n){
    int i;
    for(i=2; i*i<=n; i++){
        if(n%i==0){
            return 0;
        }
    }
    return 1;
}

int main(){

    int n;
    printf("Input n=");
    scanf("%d", &n);
    if(isPrime(n)==1){
        printf("n is prime.\n");
    }else{
        printf("n is not prime.\n");
    }
    return 0;
}
```

Functions for testing will return 1 for true and 0 for false.
Common algorithm in testing is checking all cases which cause FALSE. TRUE is accept when no case cause FALSE

Evaluate the functions

This function accesses outside data → rather coupling

```c
#include <stdio.h>

int a=5, b=10, c=2;
// Computes the average of 3 integer numbers
double average(){
    return (a+b+c)/3.0;
}

int main(){
    printf("Average of 3 integer numbers: %.2lf", average());
    return 0;
}
```

fix

```c
#include <stdio.h>
// Computes the average of 3 integer numbers
double average(int a, int b, int c){
    return (a+b+c)/3.0;
}
int main(){
    int x = 5, y = 10, z = 2;
    printf("Average of 3 integer numbers: %.2lf", average(x,y,z));
    return 0;
}
```

Better

6 - How to use a function?

How to use a function?

- In C, you can use either the built-in library functions or your own functions.
- If you use the built-in library functions, your program needs to begin with the necessary include file.
- Syntax for using a function:

```
functionIdentifier (argument1, argument2, …);
```

- Distinguish parameters and arguments
  - **Parameters**: names of data in function implementation
  - **Arguments**: data used when a function is called

Practice 1

- Develop a program that will perform the following task in three times:
  - Accept a positive integer.
  - **Print out it's divisors**
- Analyze: Print out divisors of the positive integer **n**

```
n=6
i=1 → n%i → 0 → Print out i
i=2 → n%i → 0 → Print out i
i=3 → n%i → 0 → Print out i
i=4 → n%i → 1
i=5 → n%i → 1
i=6 → n%i → 0 → Print out i
```

solution

```
for i=1 … n
    if (n%i ==0) print out i;
```

User-defined function

```
void  printDivisors(int n)
{
    int i;
    for ( i=1; i<=n; i++)
        if (n%i==0) printf("%d, ", i );
}
```

Practice 1 - Solution

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

int main(){
    int n, i;

    printf("\nInput n=");
    scanf("%d", &n);
    for(i=1; i<=n; i++){
        if(n%i==0){
            printf("%d, ", i);
        }
    }

    printf("\nInput n=");
    scanf("%d", &n);
    for(i=1; i<=n; i++){
        if(n%i==0){
            printf("%d, ", i);
        }
    }

    printf("\nInput n=");
    scanf("%d", &n);
    for(i=1; i<=n; i++){
        if(n%i==0){
            printf("%d, ", i);
        }
    }

    printf("\n");
    system("pause");
    return 0;
}
```

repeat

Same output

D:\MonHoc\PRF192\ThucHanh\practi...

```
Input n=6
1, 2, 3, 6,
Input n=5
1, 5,
Input n=10
1, 2, 5, 10,
Press any key to continue . . .
```

A function can be re-used.

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

void printDivisors(int n){
    int i;
    for(i=1; i<=n; i++){
        if(n%i==0){
            printf("%d, ", i);
        }
    }
}

int inputN(){
    int n;
    printf("\nInput n=");
    scanf("%d", &n);
    return n;
}

int main(){
    int i;

    for(i=1; i<=3; i++){
        int n;
        n = inputN();
        printDivisors(n);
    }

    printf("\n");
    system("pause");
    return 0;
}
```

parameter

User-defined function

Call function

argument

What do you think if the program will perform this task 20 times?

Exercise 1

- Develop a program that will accept a positive integer then sum of it's divisors is printed out.
- Requirement: Implementation of User-defined function
- *Hint*: Sum of divisors of the positive integer n

```
n=6, S=0
i=1 → n%i → 0 → S= 0+1=1
i=2 → n%i → 0 → S= 1+2=3
i=3 → n%i → 0 → S= 3+3=6
i=4 → n%i → 1
i=5 → n%i → 1
i=6 → n%i → 0 → S=6+6=12
```

Exercise 2

- Develop a program that will accept 3 parallel circuit resistances and their equivalent is printed out.
- Complete the program below:

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

double equivalent(double r1, double r2, double r3){
    // Your code
}

int main(){
    double R1, R2, R3, Z;

    printf("Input 3 resistances: ");
    scanf("%lf%lf%lf", &R1, &R2, &R3);

    printf("Their equivalent: %.2lf\n", equivalent(R1, R2, R3));

    system("pause");
    return 0;
}
```

$1/Z = 1/r1 + 1/r2 + 1/r3 \rightarrow Z = ?$

D:\MonHoc\PRF192\ThucHanh\exercise2_function.exe

```
Input 3 resistances: 100 200 300
Their equivalent: 54.55
Press any key to continue . . .
```

Function Prototypes

- Function prototypes describe the form of a function without specifying the implementation details
- Function prototypes declaration is put at a place and it's implementation is put at other.
- **Syntax**: **returnType functionIdentifier ([Type1 param1, Type2 param2, . . .]);**
- When the program is compiled:
  - Step 1: The compiler acknowledges this prototype (return type, name, order of data types in parameters) and marks places where this function is used and continues the compile process.
  - Step 2: If the function is detected, the compiler will update the marks in the previous step to create the program. Else, an error is thrown.

Function Prototypes - Example

```c
1  /*  prototypes.c
2   *  Computes the odd number: 1 -> n
3   *  Author: ThoPN3
4   *  Date: yyyy/MM/dd
5   */
6  #include <stdio.h>
7
8  int main(){
9      int n, result;
10
11     printf("Input n=");
12     scanf("%d", &n);
13
14     result = sumOddNumbers(n);
15     printf("Sum the odd numbers is: %d", result);
16
17     return 0;
18 }
19
20 int sumOddNumbers(int n){
21     int i, sum = 0;
22     for(i=1; i<=n; i++){
23         if(i%2!=0){
24             sum += i;
25         }
26     }
27     return sum;
28 }
```

Compiler (2) | Resources | Compile Log | Debug | Find Results | Console | Close

Line	Col	File	Message
		D:\MonHoc\PRF192\ThucHanh\proto...	In function 'main':
14	11	D:\MonHoc\PRF192\ThucHanh\prototype.c	[Warning] implicit declaration of function 'sumOddNumbers' [-Wimplicit-function-declaration]

```c
1  /*  prototypes.c
2   *  Computes the odd number: 1 -> n
3   *  Author: ThoPN3
4   *  Date: yyyy/MM/dd
5   */
6  #include <stdio.h>
7
8  // Function Prototype definition
9  int sumOddNumbers(int n);
10
11 int main(){
12     int n, result;
13
14     printf("Input n=");
15     scanf("%d", &n);
16
17     result = sumOddNumbers(n);
18     printf("Sum the odd numbers is: %d", result);
19
20     return 0;
21 }
22
23 int sumOddNumbers(int n){
24     int i, sum = 0;
25     for(i=1; i<=n; i++){
26         if(i%2!=0){
27             sum += i;
28         }
29     }
30     return sum;
31 }
```

```
Input n=10
Sum the odd numbers is: 25
```

The #include directive

- We use the **#include** directive to instruct the compiler to insert a copy of the header file into our source code.
- Syntax:

```
#include "filename"   // in user directory
#include <filename> // in system directory
```

- Example:

myFuntions.c

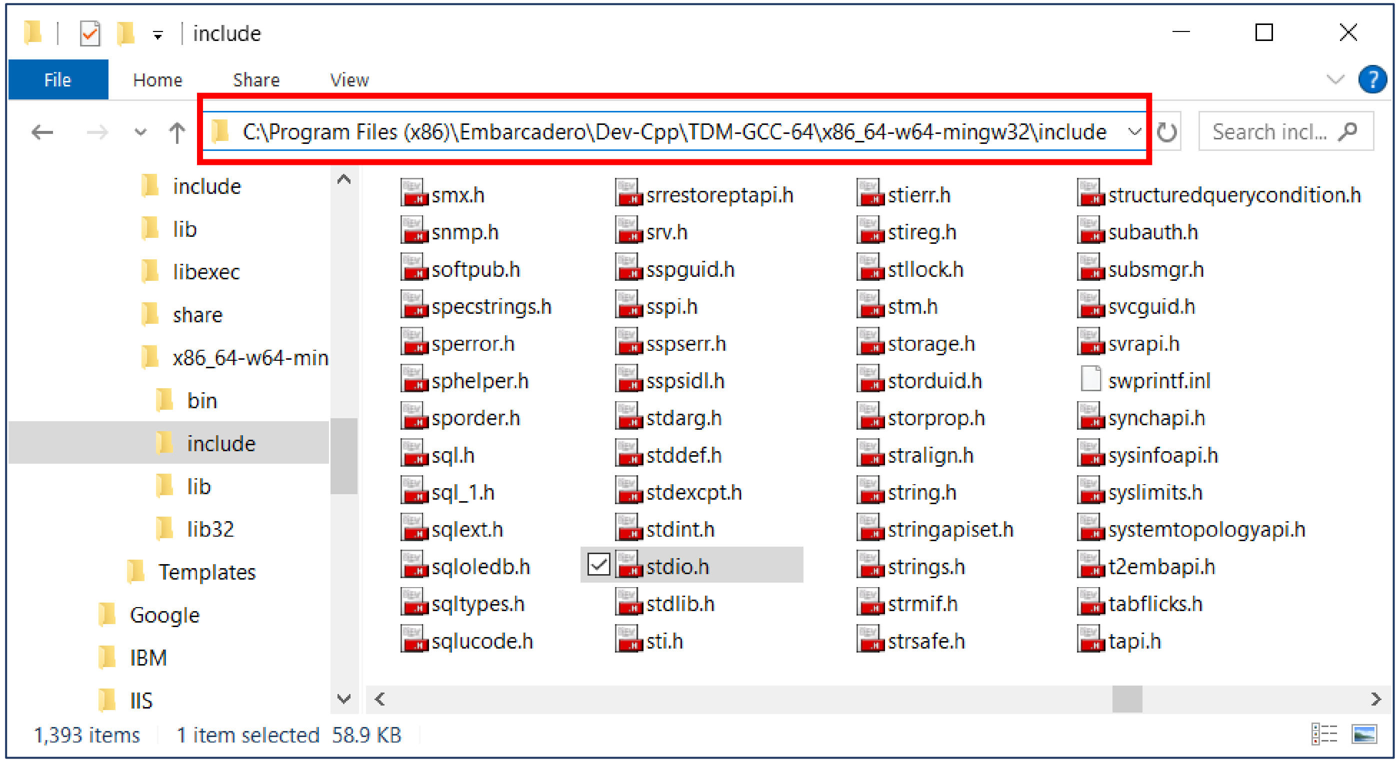
```c
int sumOddNumbers(int n){
    int i, sum = 0;
    for(i=1; i<=n; i++){
        if(i%2!=0){
            sum += i;
        }
    }
    return sum;
}
```

```c
#include <stdio.h>
#include "myFuntions.c"

int main(){
    int n, result;

    printf("Input n=");
    scanf("%d", &n);

    result = sumOddNumbers(n);
    printf("Sum the odd numbers is: %d", result);

    return 0;
}
```

```
Input n=10
Sum the odd numbers is: 25
```

The #include directive (cont.)

- System directory: The **include** directory of the select programming environment (such as Dev-Cpp)



Function Programming Style

For style, we

- Declare a prototype for each function definition
- Specify the return data type in each function definition
- Specify void for a function with no parameters
- Avoid calling the main function recursively
- Include parameter identifiers in the prototype declarations
- Use generic comments and variables names so that we can use the function in a variety of applications without modifying its code

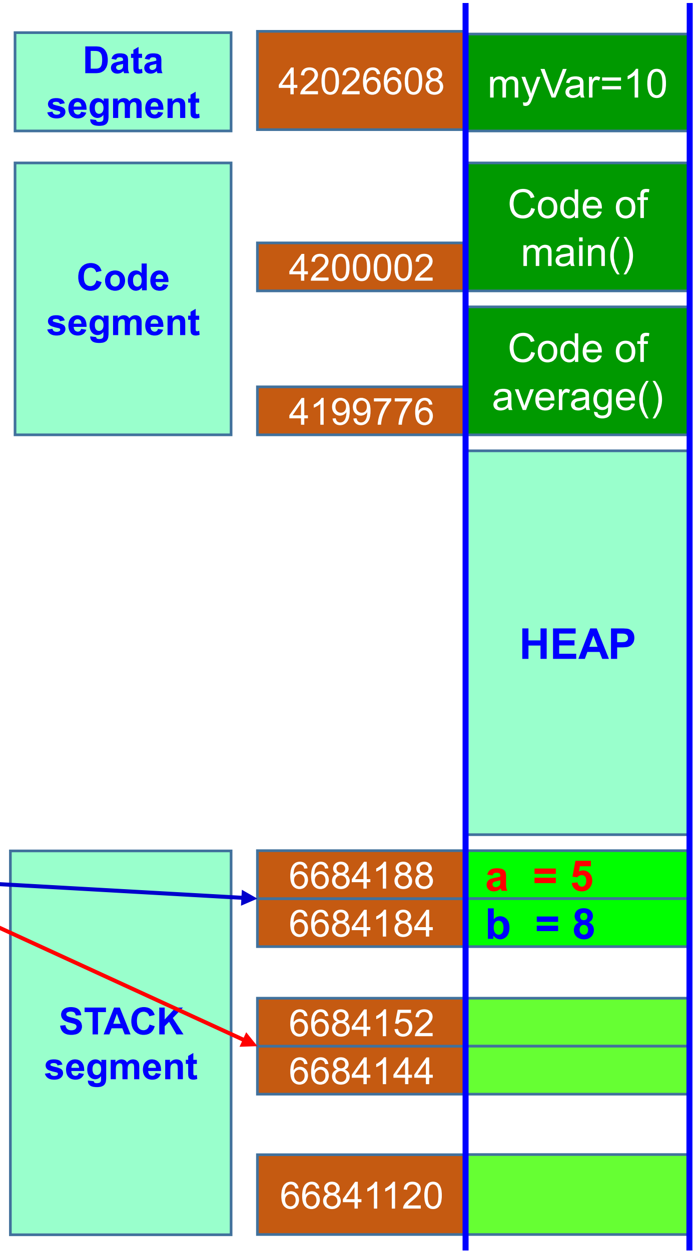
7 - What happen when a function is called?

Memory map when a function is called

```c
#include <stdio.h>

int myVar = 10;

double average(int a, int b){
    double result;
    result = (a+b)/2.0;

    printf("\nIn average function\n");
    printf("%-15s %-15s %-15s\n", "Name", "Address", "Value");
    printf("-----------------------------------------\n");
    printf("%-15s %-15u %-15d\n", "a", &a, a);
    printf("%-15s %-15u %-15d\n", "b", &b, b);
    printf("%-15s %-15u %-15lf\n", "result", &result, result);

    return result;
}

int main(){
    int a=5, b=8;

    printf("In main function\n");
    printf("%-15s %-15s %-15s\n", "Name", "Address", "Value");
    printf("-----------------------------------------\n");
    printf("%-15s %-15u %-15d\n", "myVar", &myVar, myVar);
    printf("%-15s %-15u %-15d\n", "a", &a, a);
    printf("%-15s %-15u %-15d\n", "b", &b, b);

    printf("Address of main(): %u\n", &main);
    printf("Address of average(...): %u\n", &average);
    printf("Result returned to main: %lf", average(a, b));

    return 0;
}
```

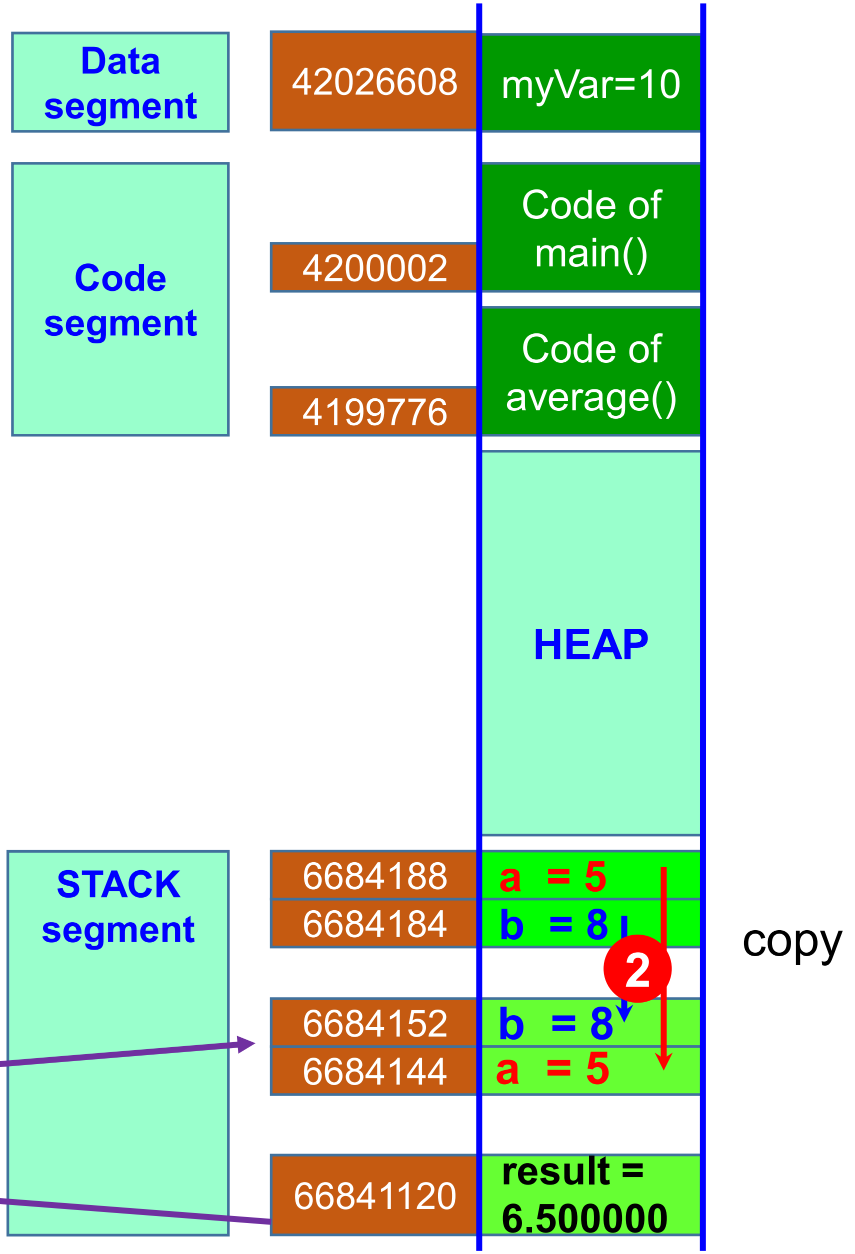
Memory mapping

```
In main function
Name            Address         Value
-----------------------------------------
myVar           4206608         10
a               6684188         5
b               6684184         8
Address of main(): 4200002
Address of average(...): 4199776

In average function
Name            Address         Value
-----------------------------------------
a               6684144         5
b               6684152         8
result          6684120         6.500000
Result returned to main: 6.500000
```



Memory map when a function is called

```c
#include <stdio.h>

int myVar = 10;

double average(int a, int b){
    double result;
    result = (a+b)/2.0;

    printf("\nIn average function\n");
    printf("%-15s %-15s %-15s\n", "Name", "Address", "Value");
    printf("-----------------------------------------\n");
    printf("%-15s %-15u %-15d\n", "a", &a, a);
    printf("%-15s %-15u %-15d\n", "b", &b, b);
    printf("%-15s %-15u %-15lf\n", "result", &result, result);

    return result;
}

int main(){
    int a=5, b=8;

    printf("In main function\n");
    printf("%-15s %-15s %-15s\n", "Name", "Address", "Value");
    printf("-----------------------------------------\n");
    printf("%-15s %-15u %-15d\n", "myVar", &myVar, myVar);
    printf("%-15s %-15u %-15d\n", "a", &a, a);
    printf("%-15s %-15u %-15d\n", "b", &b, b);

    printf("Address of main(): %u\n", &main);
    printf("Address of average(...): %u\n", &average);
    printf("Result returned to main: %lf", average(a, b));

    return 0;
}
```



Pass by value

- C-language uses the "**pass by value**" only when passing arguments to called functions.
- The function receives copies of the data supplied by the arguments in the function call (*The compiler allocates space for each parameter and initializes each parameter to the value of the corresponding argument in the function call*).
- So, anything passed into a function call is unchanged in the caller's scope when the function returns
  - Parameters and arguments stored in different addresses
  - Although they have the same names, they are still different
- When a called function completes it's task, it's memory block, allocated, is de-allocated.

Exercise 3

- A program for swapping two integers is implemented as this code.
- Rewrite, compile and run this program.
  - Draw memory map
  - Explain the result

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

void swap( int a, int b)
{   int t;
    printf("In swap, var. a, add.:%u, value:%d\n", &a, a);
    printf("In swap, var. b, add.:%u, value:%d\n", &b, b);
    printf("In swap, var. t, add.:%u, value:%d\n", &t, t);
    t = a;
    a = b;
    b = t;
}
int main()
{   int x = 5, y = 7;
    printf("In main, var. x, add.:%u, value:%d\n", &x, x);
    printf("In main, var. y, add.:%u, value:%d\n", &y, y);
    printf("Addr. of main(): %u\n", &main);
    printf("Addr. of swap(...): %u\n", &swap);
    swap (x, y);
    printf("After swapping x and y\n");
    printf("x=%d, y=%d\n", x, y);

    system("pause");
    return 0;
}
```

8 - Analyse a program to functions

Analyse a program to functions

Pick nouns → **Variables a, b, c (Suitable types)**

Problem

Re-order requirements to get a logical order (Algorithm) →

Begin
Verb1 a ; // complex
Verb 2 b; // complex
Verb 3 c; // Simple
...
End.

Simple task:
Input/Output simple variables → Use library functions

```
// library functions
#include <stdio.h>
void verb1 ( Type a)
{   }
int verb2(Type b)
{ ......  return 3*b; }
int main()
{
    int a, b, c;
    verb1(a); // call function
    printf("%d\n", verb2(b) );
   <suitable statements>
}
```

TRƯỜNG ĐẠI HỌC FPT

9 - Implement a program using functions

Implement a program using functions

- Example: Develop a program that will print out the n first primes.

Analysis
- Nouns: the integer n → int n
- Verbs:
  - Begin
  - Accept n → simple
  - Print n first primes → function
  - End.

Analysis

```
Function printNPrimes (int n)
    int count = 0;
    int value = 2;
    while (count <n)
    {
            if ( value is a prime → function
            {
                        count = count +1;
                        print out value; → simple
            }
            value = value +1;
    }
```

Result:

Input: n=5
Output: 2, 3, 5, 7, 11

Implement a program using functions (cont.)

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

int isPrime(int n){
    int i, flag = 1;

    for(i=2; i*i<=n && flag==1; i++){
        if(n%i==0){
            flag=0;
        }
    }
    return flag;
}

void printNPrimes(int n){
    int count=0; // count primes printed
    int value=2; // current value is considered
    while(count<n){
        if(isPrime(value)==1){
            printf("%d ", value);
            count++;
        }
        value++;
    }
}
```

continue

```c
int main(){
    int n;
    printf("Input number of primes:");
    scanf("%d", &n);
    printNPrimes(n);

    printf("\n");
    system("pause");
    return 0;
}
```

Compile & Run

D:\MonHoc\PRF192\ThucHanh\printNPrimes.exe

```
Input number of primes:5
2 3 5 7 11
Press any key to continue . . .
```

Exercise 4

- Develop a program that will accept two positive integers then print out the greatest common divisor and the least common multiple of them.
- *Hint*:

<u>Analysis</u>

- Nouns: 2 integers → int m, n
  The greatest common divisor → int G
  The least common multiple → int L
- Verbs:
  - Begin
  - Accept m, n → simple
  - G = Calculate the greatest common divisor of m,n → function **gcd**
  - L = Calculate the least common multiple of m,n → function **lcm**
  - Print out G, L → simple
  - End.

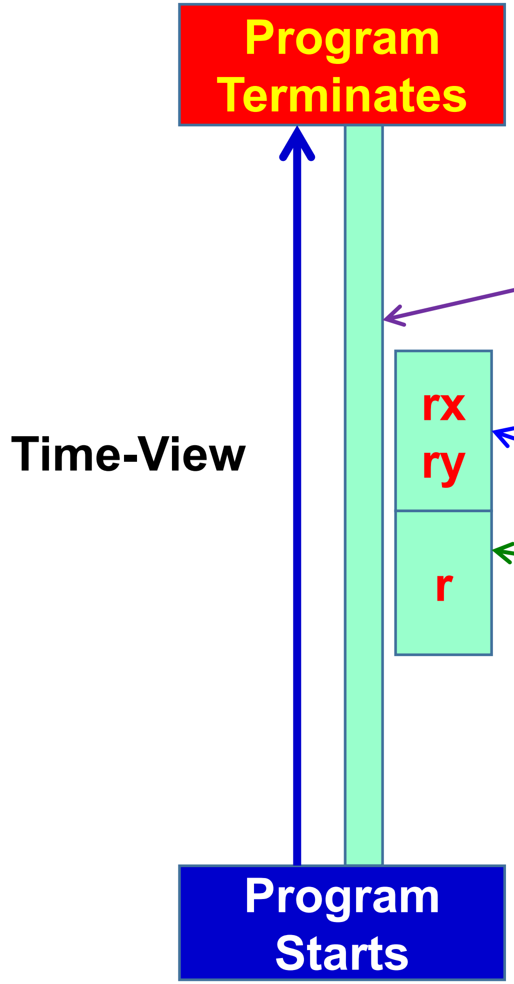
TRƯỜNG ĐẠI HỌC FPT

10 - Extent and Scope of a variable

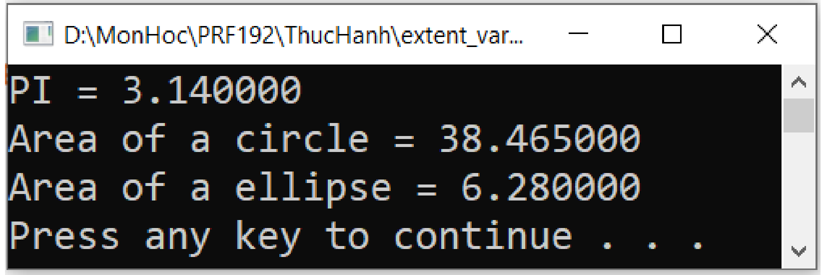
Extent and Scope of a variable

- **Extent of a variable**: Duration begins at the time the memory of this variable is allocated to the time this block is de-allocated.
- **Scope of a variable**: The code block between the line which this variable is declared and the close brace of this block. In it’s scope, the variable is visible (means that accessing to this variable is valid).
- **Global Variables**: Variables declared outside of all functions → They are stored in the data segment. If possible, do not use global variables because they can cause high coupling in functions.
- **Local Variables**: Variables declared inside a function → They are stored in the stack segment.

Extent of Variables: Time-View




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

double PI = 3.14;

double areaCircle(double radius){
    return PI * radius * radius;
}

double areaEllipse(double rx, double ry){
    return PI * rx * ry;
}

int main(){
    printf("PI = %lf\n", PI);
    printf("Area of a circle = %lf\n", areaCircle(3.5));
    printf("Area of a ellipse = %lf\n", areaEllipse(1, 2));

    system("pause");
    return 0;
}
```



Scope of Variables: Code-View

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

// Find the greatest common divisor of two integers
int gcd(int a, int b) {
    while (b != 0) {
        int temp = b;
        b = a % b;
        a = temp;
    }
    return a;
}

// Find the least common multiple of two integers
int lcm(int a, int b) {
    return (a * b) / gcd(a, b);
}

int main(){
    int m, n, G, L;

    do{
        printf("Input 2 positives integer: ");
        scanf("%d%d", &m, &n);
    }while(m<=0 || n<=0);

    G = gcd(m, n);
    L = lcm(m,n);

    printf("GCD = %d, LCM = %d\n", G, L);

    system("pause");
    return 0;
}
```

Local variables of the function **gcd** include: memory containing return value (int), a, b

Local variables of the function **lcm** include: memory containing return value (int), a, b

Local variables of the function **main** include: memory containing return value (int), m, n, G, L

D:\MonHoc\PRF192\ThucHanh\UCLN_BCNN...

```
Input 2 positives integer: 12 18
GCD = 6, LCM = 36
Press any key to continue . . .
```

Extent and Scope of a variable: Visibility

```
#include <stdio.h>

int main ( ) {
    int input;

    printf("Enter a value : ");
    scanf("%d", &input);
    if ( input > 10) {
        int input = 5;  /* POOR STYLE */

        printf("The value is %d\n", input);
    }
    printf("The value is %d\n", input);

    return 0;
}
```

```
Enter a value : 12
The value is 5
The value is 12
```

Two variables have the same name (input) but they are different because the inner variable has the narrower scope than the outer variable → **RULE: Local first, Global later**

11 - Walkthroughs with Functions

Walkthroughs with Functions

- Given the following function and a case of using it. What is the value of the variable **t** when the function terminates?

```
int f( int a, int b, int c)
{
     int t= 2*(a+b-c)/5;
    return t;
}
```

```
int x = 5, y= 6, z= 7;
int t = 3*f(y,x,z);
```

y=6	x=5	z=7	f(a,b,c)
a	b	c	t
6	5	7	2*(6+5-7)/5 =1

t = 3*f(…) = 3*1 = 3

Exercise 5

- Write a C program that will accept a non-negative integer then print out whether this number is power of 2 or not.
- *Hint*: **Analysis**
  - Variable: long n;
  - Operation: Check a long integer n whether it is power of 2 or not (named isPower2)

    return ((n & (n-1))==0);
  - main function:

    Do

        accept n;

    While (n<=0)

    if (isPower2(n)==1) Print out " It is power of 2"

    else print out " It is not power of 2"

n	n binary	n&(n-1)
1	0000 0001	0000 0001 0000 0000 0000 0000
2	0000 0010	0000 0010 0000 0001 0000 0000
4	0000 0100	0000 0100 0000 0011 0000 0000
8	0000 1000	0000 1000 0000 0111 0000 0000
16	0001 0000	0001 0000 0000 1111 0000 0000

Exercise 6

- Write a C program that will
  - Accept 3 integers m, d, y that represent a date.
  - Print out they are valid or not.
  - Attention:
    - The February in a leap year will have 29 days.
    - If Y is a leap year then (Y%4==0 && Y%400 !=0) || (Y%100==0)

Summary

- Module: A portion of a program that carries out a specific function and may be used alone or combined with other **modules** to create a program.
- Advantages of modules: It is easy to upgrade and it can be re-used
- C-function is a module
- A function is highly cohesive if all it’s statements focus to the same purpose
- Parameters make a function low coupling
- 4 parts of a function: Return type, function name, parameters, body
- Syntax for a function:

```
returnType functionName( Type param1, Type param2, …)
{
        <<statements>>
}
```

Summary (cont.)

- Steps for implementing a function:
  - State the task clearly, verb is function name, nouns are parameters
  - Verb as find, search, calculate, count, check → return value function will return value. Other verbs: void function
  - Give parameters specific values, do the work manually, write down steps done, translate steps to C statement
- Simple tasks: input/output some single value → Basic task → Library functions
- C-language uses the pass-by-value in passing parameters → The called function can not modify this arguments.
- Simple tasks: input/output some single values → Basic tasks → Library functions
- C-language uses the pass-by-value in passing parameters → The called function can not modify it's arguments.

Summary (cont.)

- Function prototype is a function declaration but it's implementation is put at another place.
- Syntax for a function prototype:

  returnType functionName (parameterType,,,)

- Compiler will compile a program containing function prototype in three:
  - **Step 1**: Acknowledges the function template and marks places where this function is called.
  - **Step 2:** Update marks with function implementation if it is detected.
- Use a system library function: #include<file.h>
- Use **user-defined function in outside file**: #include “filename”
- Extent of a variable begins at the time this variable is allocated memory to the time this memory is de-allocated.
- Scope of a variable begins at the line in which this variable is declared to the closing brace containing it.