

# Slots 07-08-09

## Modules and Functions

Modules -  
C-Functions  
Scope of a variable

# Objectives

- Naturally, people divide a complex job into some smaller and simpler jobs. Each sub-job is expressed using a verb.
- Similarly, a program can be rather complex.
- How to divide a program into simpler parts and how to use them?

# Objectives

After studying this chapter, you should be able to:

- Define a C- module or C-function?
- Explain module's characteristics
- Implement C functions
- Use functions?
- Differentiate build-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

# 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.

# Contents

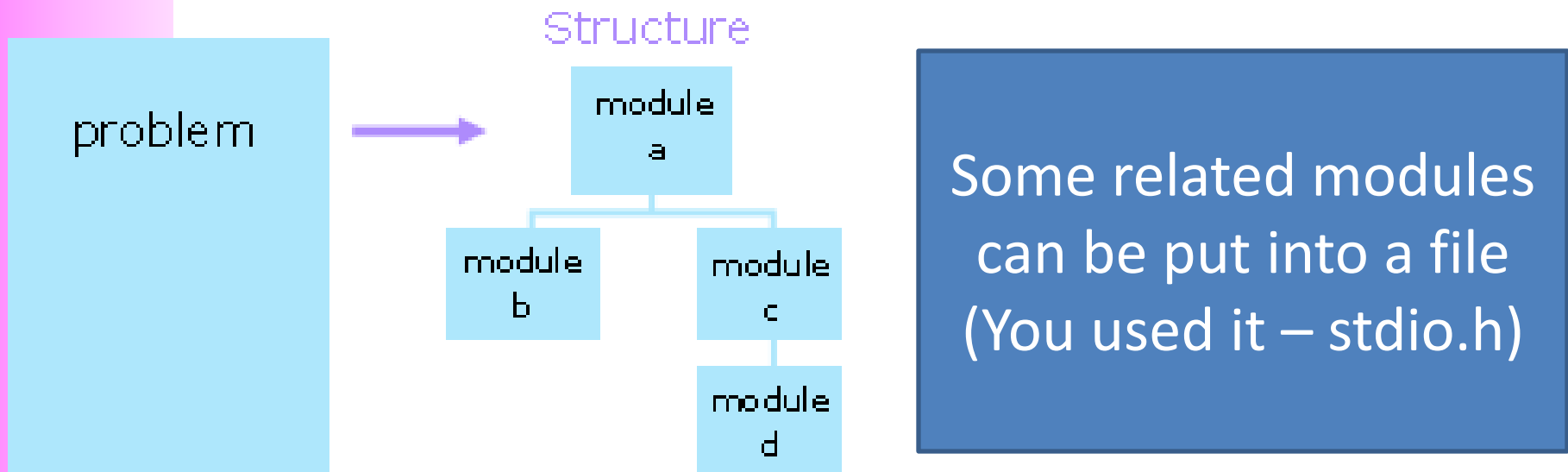
- What is a module?
- Characteristics of modules
- Hints for module identifying
- C-Functions and Modules
- How to Implement a function?
- How to use a function?
- What happen when a function is called?
- How to analyze a problem into functions?
- Implement a program using functions
- Extent and Scope of a variable
- Walkthroughs with Functions

# 1-What is a Module?

- (Software)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.
- ***To cook rice:***  
(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.



# Structure Design: An example

Develop a program that will accept a positive integer then sum of it's divisors is printed out.

Analyze	Code	Description
	<code>#include &lt;stdio.h&gt;</code>	Use modules in this file
<i>Divide the program into small tasks</i>	<code>int main</code> <code>{ int n; int s;</code>	Declare the main module and it's data
1- Accept n	<code>scanf("%d", &amp;n);</code>	Use a module scanf in the stdio.h
2- s = sum of it's divisors	<code>s = sumDivisors (n);</code>	Module will be implemented
3- Print out s	<code>printf("%d", s);</code>	Use a module printf in the stdio.h
4- Pause the program	<code>getchar();</code>	Use a module getchar in the stdio.h
	<code>}</code>	



## 2- Characteristics of Modules

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

All of them will be depicted in examples below.

## 3- Module identifying: Hints

In a structured design

- Each module has one entry point and one exit point,
- Each module is highly cohesive (performs a single task – tính gắn kết cao ), **code of a module focuses to the determined purpose and some related modules are put in one file.**
- Each module exhibits low coupling (sự phụ thuộc thấp). **In the best case, modules are independent.**

# 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-trùng lặp" - unrelated tasks → This module is not enough small → Separate smaller tasks in this task.
  - "logical- gom những việc liên quan lại" – This module contains some related tasks of which only one is performed - the module identifier suggests a choice → Separate them into separate smaller module, each smaller module for a choice.
  - "temporal-tạm thời" - 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

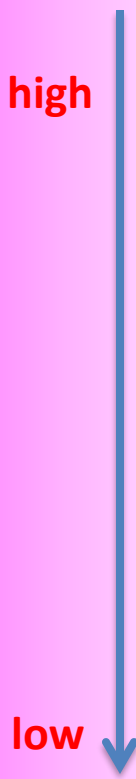
Some modules share the common data can be accepted if they perform their tasks sequentially.

# Module identifying : High Coupling

**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
    - "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



# 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

- *In C, we represent a module by a function.*
- A function may receive data and may return a value. Examples:
  - Print out divisors of the integer  $n \rightarrow n$  is data is accepted by the function and no value is returned.
    - $n=10 \rightarrow$  Print out values: 1, 2, 5
  - Sum of divisors of the integer  $n \rightarrow n$  is data is accepted by the function and a value is returned.
    - $n=10 \rightarrow 8$  is the return value
- The description of the internal logic of a function as the function's definition.

# Function Definitions: 4 parts

## Function header

```
returnType functionName (Type param1, Type param2, ...)
```

```
{
```

```
<code>
```

```
[return value; ]
```

```
}
```

**Function  
body**

**What is  
the  
result of  
the task**

**What is  
the  
name of  
the  
task?**

**To do this  
task, what  
are  
necessary  
data?**

**How  
does  
this task  
do?**

# Function syntax: 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 syntax: void function

- To identify a function that does not return any value, we specify **void** 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.

# void Function: Example

```
void printDivisors (int n)
{
    int i;
    for (i=1; i<= n/2; i++)
        if (n%i==0) printf("%d, ", i);
}
```

## Cases in which void functions can be selected:

- If you do this task, you realize that no value is needed after this task done.
- In the function body, the essential statements are printing data out.

# 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*

# 5-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 some functions

This function contains a sub-task → low cohesive.

```
// 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 a=1, b=5, c=7;  
// Calculate the average value of 3 integers  
double average()  
{  
    return (a+b+c)/3.0;  
}
```

This function accesses outside data  
→ rather coupling

Better

```
// Test whether an integer is a prime  
int isPrime( int n)  
{  
    int i;  
    for (i=2; i*i <=n; i++)  
        if (n%i==0) return 0;  
    return 1;  
}  
  
// Calculate the average value of 3 integers  
double average(int a, int b, int c)  
{  
    return (a+b+c)/3.0;  
}
```

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

## 6-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: **functionName (arg1, arg2,...);**

***Distinguish parameters and arguments***

**Parameters**: names of data in function implementation

**Arguments**: data used when a function is called

# Demonstration 1

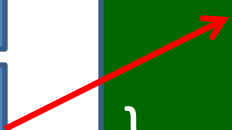
- Develop a program that will perform the following task in three times:
  - Accept a positive integer.
  - **Print out it's divisors → User-defined function.**

Print out divisors of the positive integer n

```
n=10
i=1 → n%i → 0 → Print out i
i=2 → n%i → 0 → Print out i
i=3 → n%i → 1
i=4 → n%i → 2
i=5 → n%i → 0 → Print out i
```

```
For i=1 .. n/2
    if (n%i ==0) Print out i;
```

```
void printDivisors ( int n)
{
    int i;
    for ( i=1; i<=n/2; i++)
        if (n%i==0) printf ( "%d, ", i );
}
```



# A function can be re-used.

## Demonstration 1

```
#include <stdio.h>
int main()
{
    int n, i;
    printf("\nInput n=");
    scanf("%d", &n);
    for (i=1; i<=n/2; i++)
        if (n%i==0) printf("%d, ", i);
    printf("\nInput n=");
    scanf("%d", &n);
    for (i=1; i<=n/2; i++)
        if (n%i==0) printf("%d, ", i);
    printf("\nInput n=");
    scanf("%d", &n);
    for (i=1; i<=n/2; i++)
        if (n%i==0) printf("%d, ", i);
    getchar();
    getchar();
}
```

C:\ K:\GiangDay\FU\OOP\BaiTap\demo1.exe

```
Input n=10
1, 2, 5,
Input n=15
1, 3, 5,
Input n=20
1, 2, 4, 5, 10, _
```

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

```
#include <stdio.h>
void printDivisors(int N)
{
    int i;
    for (i=1; i<=N/2; i++)
        if (N%i==0) printf("%d, ", i);
}
int main()
{
    int n, i;
    printf("\nInput n=");
    scanf("%d", &n);
    printDivisors(n);
    printf("\nInput n=");
    scanf("%d", &n);
    printDivisors(n);
    printf("\nInput n=");
    scanf("%d", &n);
    printDivisors(n);
    getchar();
    getchar();
}
```

parameter

Function  
Implemen  
tation

argument

Using  
function

C:\ K:\GiangDay\FU\OOP\BaiTap\demo1.exe

```
Input n=10
1, 2, 5,
Input n=15
1, 3, 5,
Input n=20
1, 2, 4, 5, 10, _
```

## Demonstration 2

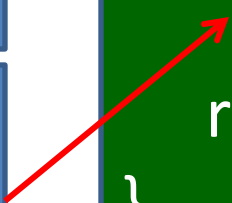
- Develop a program that will accept a positive integer then sum of it's divisors is printed out.

Sum of divisors of the positive integer n

```
n=10, S=0
i=1 → n%i → 0 → S= 0+1 =1
i=2 → n%i → 0 → S=1+2=3
i=3 → n%i → 1
i=4 → n%i → 2
i=5 → n%i → 0 → S= 3+5=8
```

```
S=0;
for i=1 .. n/2
    if (n%i ==0) S+=i;
return S;
```

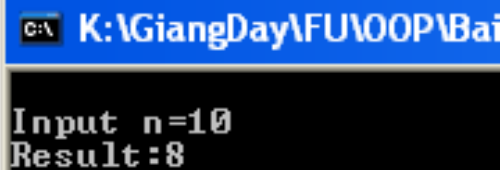
```
int sumDivisors ( int n )
{
    int S=0, i;
    for ( i=1; i<=n/2; i++)
        if (n%i==0) S +=i;
    return S;
}
```



# Demonstration 2

```
#include <stdio.h>
int sumDivisors(int N)
{   int i, S=0;
    for (i=1; i<=N/2; i++)
        if (N%i==0) S+=i;
    return S;
}
int main()
{   int n, sum;
    printf("\nInput n=");
    scanf("%d", &n);
    sum = sumDivisors(n);
    printf("Result:%d\n", sum);
    getchar();
    getchar();
}
```

**Code yourself**



```
C:\ K:\GiangDay\FU\OOP\Ba
Input n=10
Result:8
```

# Demonstration 3

```
#include <stdio.h>
int sumDivisors(int n )
{
    int i, S = 0;
    for ( i=1; i<= n/2; i++)
        if ( n%i !=0) S+=i;
    return S;
}
int main()
{
    int n;
    printf("\nInput a positive integer:");
    scanf("%d", &n);
    printf("Sum of divisors: %d\n", sumDivisors(n));
    getchar();
    getchar();
}
```

**Functions help maintaining the code easier.**

**Error source.**

**K:\GiangDay\FU\OOP\BaiTap\demo1.exe**

```
Input a positive integer:10
Sum of divisors: 7
```

**$N=10 \rightarrow 1+2+5 = 8$   
7 Is printed out. WHY?**

# Demonstration 4

Develop a program that will accept 3 resistances of a paralleled circuit and their equivalent is printed out.

```
#include <stdio.h>
double equivalent ( double r1, double r2, double r3)
{
    <your code>
}
int main()
{
    double R1, R2, R3, Z;
    printf("\nInput 3 resistances:");
    scanf("%lf%lf%lf", &R1, &R2, &R3);
    printf("Their equivalent: %lf\n", equivalent(R1, R2, R3));
    getchar();
    getchar();
}
```

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



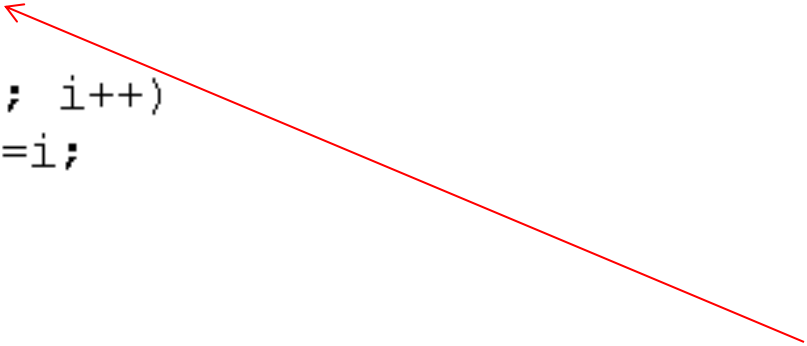
```
Input 3 resistances:3 3 3
Their equivalent: 1.000000
```



# Coercion When a Function is Called

- If there is a mismatch between the data type of an argument and the data type of the corresponding parameter, the compiler, wherever possible, coerces (sự ép kiểu) the value of the argument into the data type of the parameter.

```
#include <stdio.h>
int sumDivisors(int n )
{   int i, S = 0;
    for ( i=1; i<= n/2; i++)
        if ( n%i==0) S+=i;
    return S;
}
int main()
{   printf("Sum of divisors: %d\n", sumDivisors(15.7));
    getchar();
}
```



C:\K:\GiangDay\FU\OOP\BaiTap\demo1.exe

Sum of divisors: 9

# Function Prototypes

- Function prototypes describe the form of a function without specifying the implementation details → **Function declaration is put at a place and it's implementation is put at other.**
- 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.

```
returnType FuncName ( Type1 [ param1], Type2 [param2], ... ) ;
```

# Function Prototypes...

```
#include <stdio.h>

int main()
{
    int n;
    printf("Input a positive integer:");
    scanf ("%d", &n);
    printf("Result:%d\n",sumDivisors(n));
    getchar();
    getchar();
}

int sumDivisors(int n )
{
    int i, S = 0;
    for ( i=1; i<= n/2; i++)
        if ( n%i==0) S+=i;
    return S;
}
```

C:\ K:\GiangDay\FU\OOP\BaiTap\demo1.exe

Input a positive integer:12  
Result:16

The DEV C++ 4.9.9.2 compiler agrees user-defined functions which are implemented below the main function. Others, such as BorlandC++, do not.

OR

`int sumDivisors(int n);`

Prototype

```
#include <stdio.h>
int sumDivisors(int);
int main()
{
    int n;
    printf("Input a positive integer:");
    scanf ("%d", &n);
    printf("Result:%d\n",sumDivisors(n));
    getchar();
    getchar();
}

int sumDivisors(int n )
{
    int i, S = 0;
    for ( i=1; i<= n/2; i++)
        if ( n%i==0) S+=i;
    return S;
}
```

C:\ K:\GiangDay\FU\OOP\BaiTap\demo1.exe

Input a positive integer:12  
Result:16

It isn't recommended to take specific characteristics of the specific compilers. Use standard rules for making your program compiled easily in all compilers

# Function Prototypes...

```
1 #include <stdio.h>
2 // Prototype
3 int isPrime( int);
4 int main()
5 {   int n;
6     printf("Input n=:");
7     scanf("%d", &n);
8     if (isPrime(n)==1) printf("It's a prime.");
9     else printf("It's not a prime.");
10    getchar();
11    return 0;
12 }
```

Prototype: Acknowledge it

Use it. This position is marked.

Resources | Compile Log | Debug | Find Results | Close

Message

[Linker error] undefined reference to 'isPrime'  
ld returned 1 exit status

But it's implementation is missed!  
→ Can not update marks → Error

# 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`

myFunction.c

```
1 /* myfunc.c */
2 int isPrime( int n)
3 {   int i;
4     for (i=2; i*i <=n ; i++)
5         if (n%i==0) return 0;
6     return 1;
7 }
```

```
#include <stdio.h>
#include "myFunction.c"
int main()
{   int n;
    printf("Input a positive integer:");
    scanf ("%d", &n);
    if (isPrime(n)==1)printf("%d is a prime.\n", n);
    else printf("%d is not a prime.\n",n);
    getchar();
    getchar();
}
```

cmd K:\GiangDay\FU\OOP\BaiTap\demo1.exe

```
Input a positive integer:17
17 is a prime.
```

# Evaluating the isPrime(int) function

myFunction.c

```
1 /* myfunc.c */
2 int isPrime( int n)
3 {   int i;
4     for (i=2; i*i <=n ; i++)
5         if (n%i==0) return 0;
6     return 1;
7 }
```

2 exit points

→ It is not recommended.

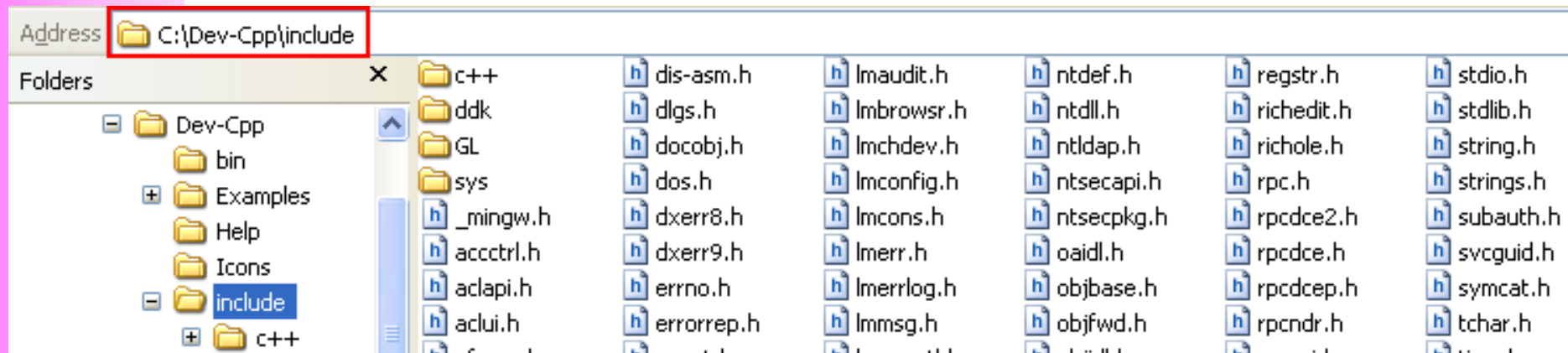
```
int isPrime(int n)/* Modified*/
{   int result=1, i;
    for (i=2; i*i<=n && result==1; i++)
        if (n%i==0) result=0;
    return result;
}
```

- Structure of a program code should be organized in a manner so that it is **understandable, testable and readily modifiable.**

→ It consists of simple logical constructs, each of which has **one entry point and one exit point.**

# The #include Directive

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



# 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?

- We use function calls to transfer execution control to functions. The syntax of a function call is

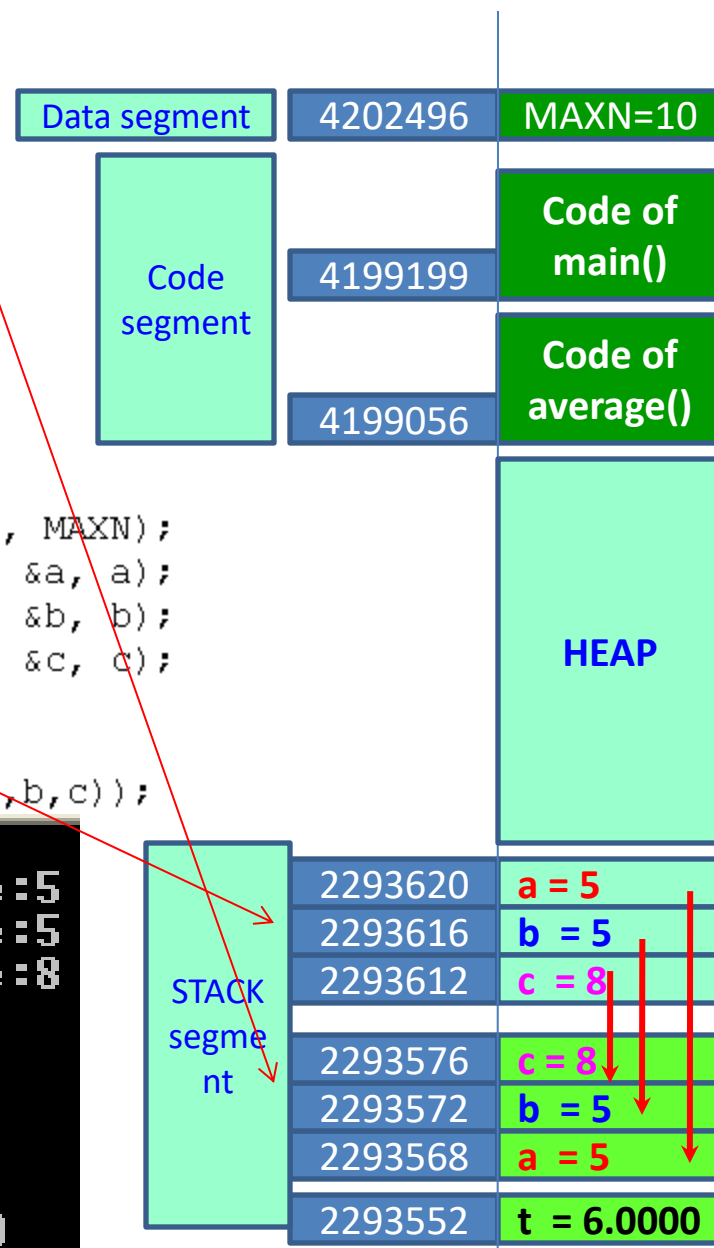
**functionIdentifier (argument1,argument2,...);**

- Once a function finishes executing its own instructions, it returns control to the point immediately following the initiating call.

# Memory Map when a function is called.

```
#include <stdio.h>
int MAXN=10;
double average (int a, int b, int c)
{ printf("Arg. a, address:%u, value:%d\n", &a, a);
  printf("Arg. b, address:%u, value:%d\n", &b, b);
  printf("Arg. c, address:%u, value:%d\n", &c, c);
  double t = (a+b+c)/3.0;
  printf("Var. t, address:%u, value:%lf\n", &t, t);
  return t;
}
int main()
{ int a= 5, b=5, c=8;
  printf("Var. MAXN, address:%u, value:%d\n", &MAXN, MAXN);
  printf("In main, var. a, address:%u, value:%d\n", &a, a);
  printf("In main, var. b, address:%u, value:%d\n", &b, b);
  printf("In main, var. c, address:%u, value:%d\n", &c, c);
  printf("Add. of main():%u\n", &main);
  printf("Add. of average(...):%u\n", &average);
  printf("Result returned to main: %lf\n", average(a,b,c));
}
```

```
Var. MAXN, address:4202496, value:10
In main, var. a, address:2293620, value:5
In main, var. b, address:2293616, value:5
In main, var. c, address:2293612, value:8
Add. of main():4199199
Add. of average(...):4199056
Arg. a, address:2293568, value:5
Arg. b, address:2293572, value:5
Arg. c, address:2293576, value:8
Var. t, address:2293552, value:6.000000
Result returned to main: 6.000000
```



# Memory Map when a function is called.

```
#include <stdio.h>
int MAXN=10;
double average (int a, int b, int c)
{ printf("Arg. a, address:%u, value:%d\n", &a, a);
  printf("Arg. b, address:%u, value:%d\n", &b, b);
  printf("Arg. c, address:%u, value:%d\n", &c, c);
  double t = (a+b+c)/3.0;
  printf("Var. t, address:%u, value:%lf\n", &t, t);
  return t;
}
int main()
{ int a= 5, b=5, c=8;
  printf("Var. MAXN, address:%u, value:%d\n", &MAXN, MAXN);
  printf("In main, var. a, address:%u, value:%d\n", &a, a);
  printf("In main, var. b, address:%u, value:%d\n", &b, b);
  printf("In main, var. c, address:%u, value:%d\n", &c, c);
  printf("Add. of main():%u\n", &main);
  printf("Add. of average(...):%u\n", &average);
  printf("Result returned to main: %lf\n", average(a,b,c));
  getchar();
  return 0;
}
```

Data segment	4202496	MAXN=10
Code segment	4199199	Code of main()
	4199056	Code of average()

2293620	a = 5
2293616	b = 5
2293612	c = 8
2293576	c = 8
2293572	b = 5
2293568	a = 5
2293552	t = 6.0000

## Mechanism for Calling function in C

- 1- Data of called function are allocated in the stack segment.
- 2- Copy values of arguments(calling function) to parameters (called function).
- 3- Control is moved to the called function and codes in the called function executes, result value of the called function is returned to the calling function and control is moved to the calling function
- 4- Remaining codes in the calling function execute.

# 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 its task, its memory block, allocated, is de-allocated.*

# In-class Exercise

A program for swapping two integers is implemented as this code.

Copy, paste, compile and run this program.

- Draw memory map
- Explain the result

```
#include <stdio.h>

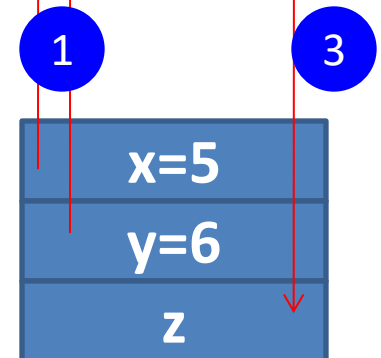
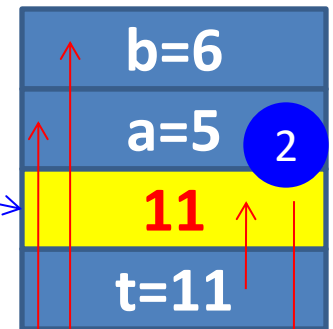
void swap( int a, int b)
{ int t;
  printf("In swap, var. a, add.:%u, value:%d\n", &a, a);
  printf("In swap, var. b, add.:%u, value:%d\n", &b, b);
  printf("In swap, var. t, add.:%u, value:%d\n", &t, t);
  t = a;
  a = b;
  b = t;
}

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);
  getchar();
  return 0;
}
```

# How does a return-function perform?

```
#include <stdio.h>
int sum( int a, int b)
{   int t= a+b;
    return t;
}
int main()
{   int x = 5, y= 6;
    int z = sum(x,y);
    printf("Result:%d\n", z);
    getchar();
    return 0;
}
```

When a return-value function executes, a memory block is allocated to store the result of the function

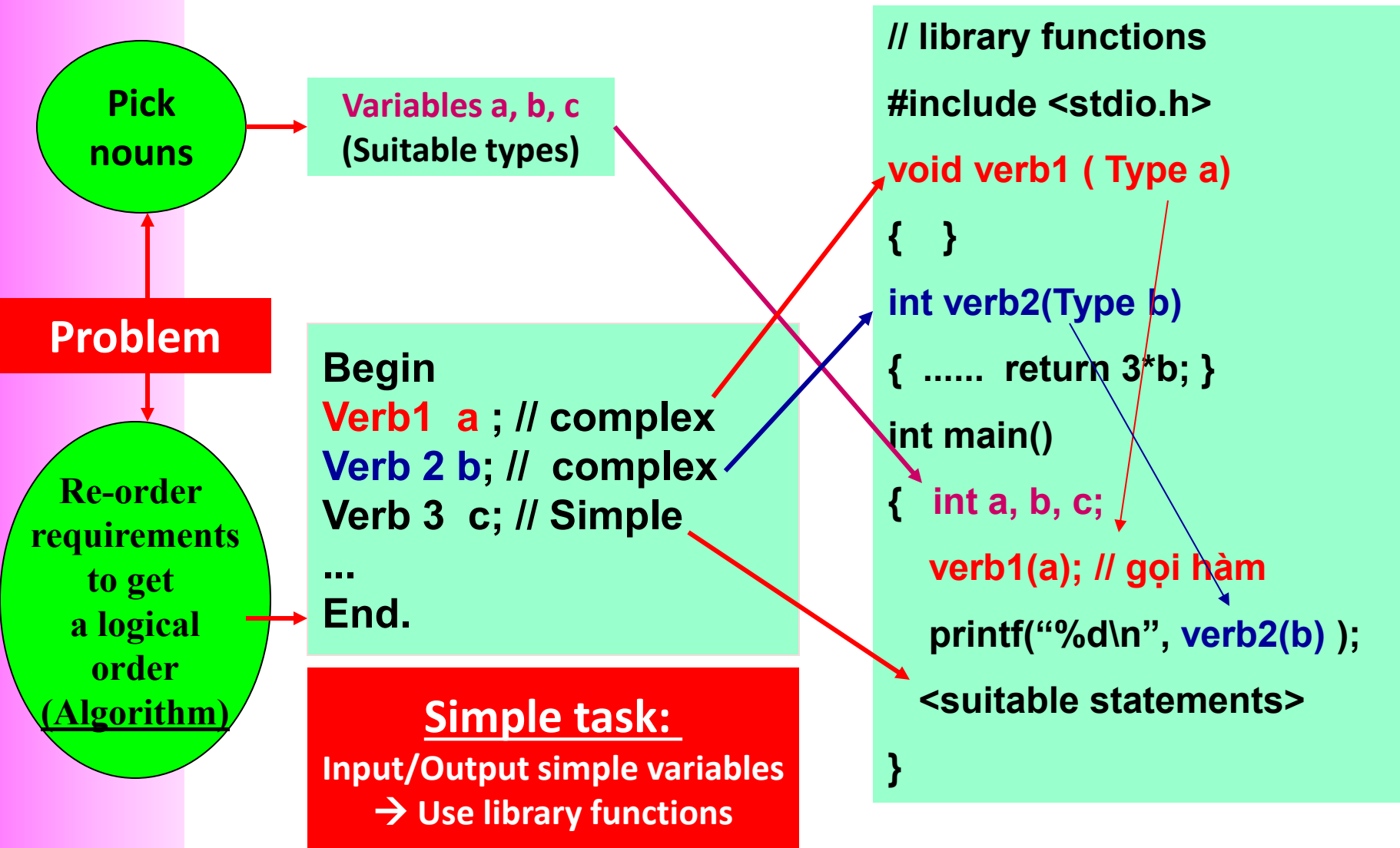


C:\K:\GiangDay\FU\OOP\BaiTap\testPrototype.exe

Result:11

- 1 - Copy arguments to function's parameters
- 2 - Code of the function executes to determine the result
- 3 - Copy the return value to the calling function

# 8-Analyse a program to functions



# 9- Implement a program using functions

**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.

Input: n=5

Output: 2, 3, 5, 7, 11

## Analysis

```
Function print_n_Primes (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;
    }
```



# Implement a program using functions ...

Develop a  
program that will  
print out **n** first  
primes.

Implement it.

```
1 #include <stdio.h>
2 int isPrime(int n)
3 {   int result=1, i;
4     for (i=2; i*i<=n && result==1; i++)
5         if (n%i==0) result=0;
6     return result;
7 }
8 void print_n_Primes(int n)
9 {   int count = 0; /* count primes printed */
10    int value =2; /* current value is considered */
11    while (count<n)
12    {   if (isPrime(value)==1)
13        {   printf("%d ", value);
14            count++;
15        }
16        value++;
17    }
18 }
19 int main()
20 {   int n;
21    printf("Input number of primes:");
22    scanf("%d",&n);
23    print_n_Primes(n);
24    getchar();
25    getchar();
26    return 0;
27 }
```

# Implement a program using functions ...

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

## Analysis

-Nouns: 2 integers  $\rightarrow$  int m,n

The greatest common divisor  $\rightarrow$  int G

The least common multiple  $\rightarrow$  int L

- Verbs:

- Begin

- Accept m, n  $\rightarrow$  simple

- G= Calculate the greatest common divisor of m,n  $\rightarrow$  function gcd

- L = Calculate the least common multiple of m,n  $\rightarrow$  function lcm

- Print out G, L  $\rightarrow$  simple

- End.

# Implement a program using functions ...

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

```
/* Caculate the gcd of two intergers */
int gcd (int value1, int value2)
{
    while (value1 != value2)
        if (value1 > value2) value1 -= value2;
        else value2 -= value1;
    return value1;
}
```

```
/* Caculate the lcm of two intergers */
int lcm (int value1, int value2)
{
    return (value1*value2)/gcd(value1, value2);
}
```

```
int main()
{
    int m, n, L, G;
    do
    {
        printf("Input 2 positives interger:");
        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);
    getchar(); getchar();
    return 0;
}
```

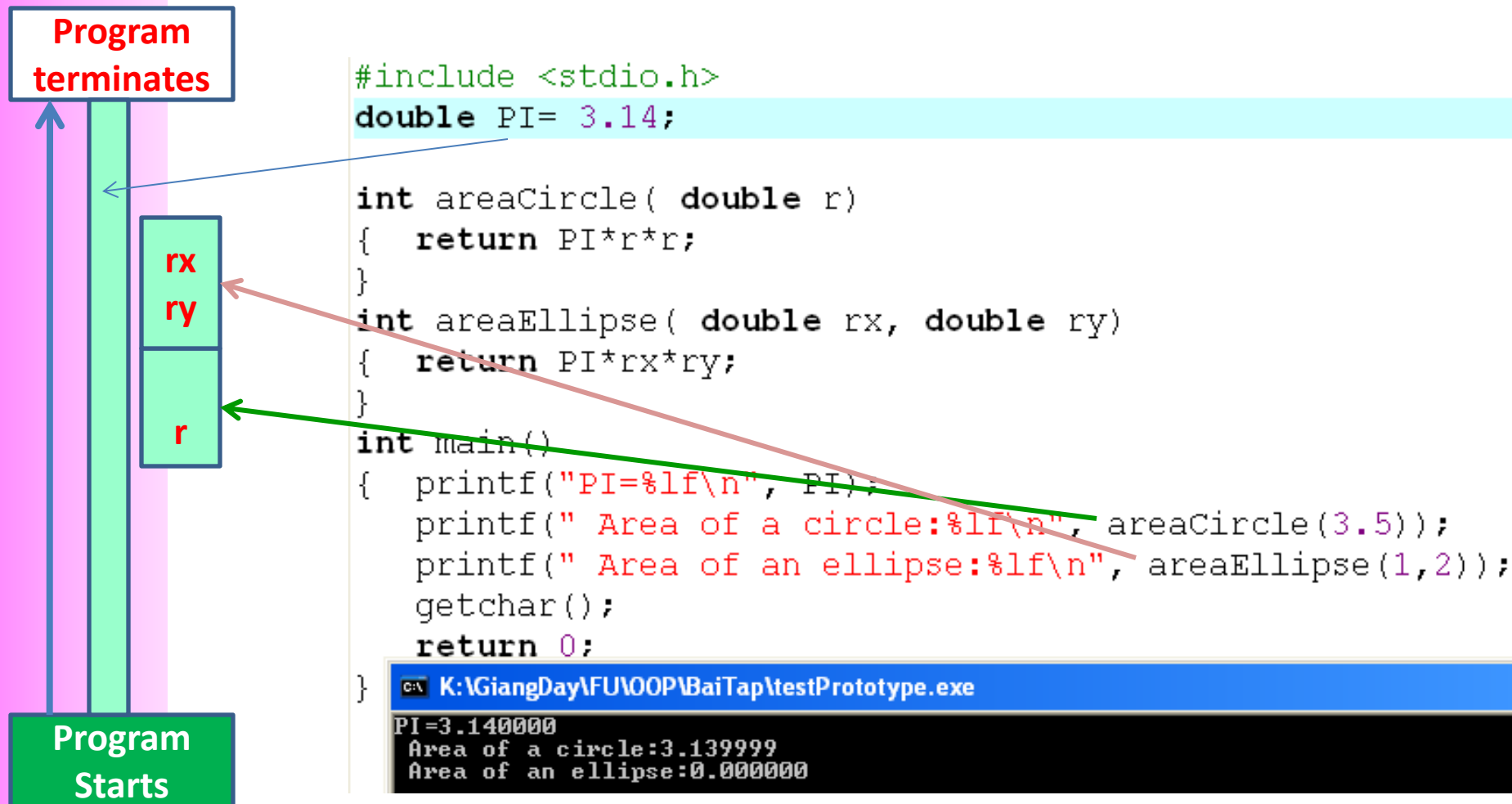
value1	value2
14	62
	62-14 = 48
	48-14 = 34
	34-14 = 20
	20-14 = 6
14-6 = 8	
8-6=2	
	6-2 = 4
	4-2= 2

```
Input 2 positives interger:-12 7
Input 2 positives interger:14 62
GCD:2, LCM:434
```

# 10- Extent and Scope of a variable

- ***Extent of a variable***: (tuổi thọ) Duration begins at the time the memory of this variable is allocated to the time this block is de-allocated.
- ***Scope of a variable***: (tầm vực) 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***: (biến toàn cục) 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***: (biến cục bộ) Variables declared inside a function → They are stored in the stack segment.

# Extent of Variables: Time-View



# Scope of Variables: Code-View

```
#include <stdio.h>

/* Caculate the gcd of two intergers */
int gcd (int value1, int value2)
{   while (value1 != value2)
    {   if (value1>value2) value1 -= value2;
        else value2 -= value1;
    }
    return value1;
}

/* Caculate the lcm of two intergers */
int lcm (int value1, int value2)
{   return (value1*value2)/gcd(value1, value2);
}

int main()
{   int m, n, L, G;
    do
    {   printf("Input 2 positives interger:");
        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);
    getchar(); getchar();
    return 0;
}
```

Local variables of the function **gcd** include:  
memory containing return value (int), value1, value2

Local variables of the function **lcm** include: memory containing return value (int), value1, value2

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

# Scope of Variables: Code-View

```
#include <stdio.h>
#include <conio.h>
int maxN=3;
void T(int a, int b)
{
    int k=0;
    k+=a;
    k+=b;
    int t=1;
    for (;t<k; t++) maxN+=t;
    printf("maxN=%d\n", maxN);
    printf("t+k=%d\n", t+k);
}
int main()
{
    int x=1, y=1;
    T(x, y);
    maxN=10;
    printf("maxN=%d", maxN);
    getch();
}
```

maxN

a, b

k

t

# 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
```

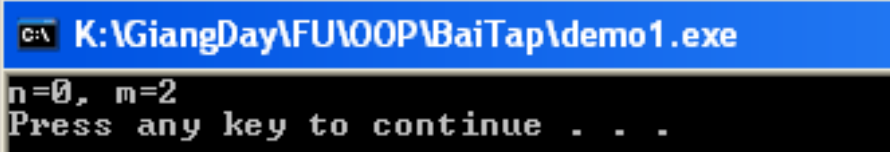
Two variables have the same name (input) but they are different because the inner variable has the narrower scope than the outer variable → **RULE: Local first, Global later**



# Extent and Scope of a variable: Visibility

## Values in non-initialized variables

```
#include <stdio.h>
#include <stdlib.h>
int n;
int main()
{   int m;
    printf("n=%d, m=%d\n", n, m);
    system("pause");
}
```



Clearing all memory blocks of a program before the program is loaded may spend a significant time. So, the mechanism is usually used is clearing the data segment only, but the stack segment is not. As a result, if variables are not initialized, a global variable is 0 but a local variable contains a non predictable value.

# 11- 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;
}
```

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

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

$t = 3*f(\dots) = 3*1 = 3$

# 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

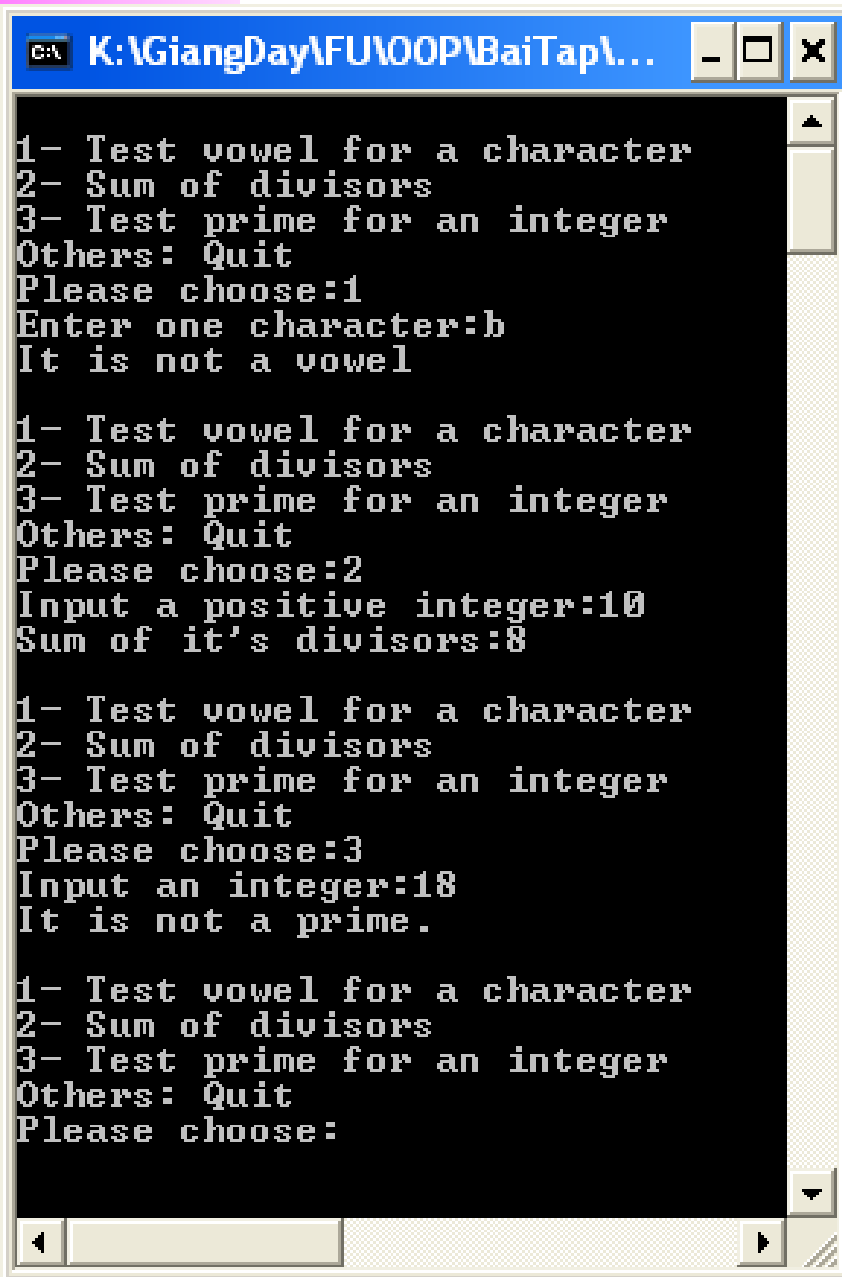
- Steps for implementing a function:
  - State the task clearly, verb is function name, nouns are parameters
  - Verb as find, search, calculate, count, check → return value function will return value. Other verbs: void function
  - Give parameters specific values, do the work manually, write down steps 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

- 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 and **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.

# Thank you

# Program using menu for some tasks



```
K:\GiangDay\FU\OOP\BaiTap\...  
1- Test vowel for a character  
2- Sum of divisors  
3- Test prime for an integer  
Others: Quit  
Please choose:1  
Enter one character:b  
It is not a vowel  
  
1- Test vowel for a character  
2- Sum of divisors  
3- Test prime for an integer  
Others: Quit  
Please choose:2  
Input a positive integer:10  
Sum of it's divisors:8  
  
1- Test vowel for a character  
2- Sum of divisors  
3- Test prime for an integer  
Others: Quit  
Please choose:3  
Input an integer:18  
It is not a prime.  
  
1- Test vowel for a character  
2- Sum of divisors  
3- Test prime for an integer  
Others: Quit  
Please choose:
```

Develop a C-program that allows user choose one task at a time:

- 1- Test whether a character is a vowel or not.
- 2- Print out sum of divisors of an integer.
- 3- Test whether an integer is a prime or not.

# Program using menu for some tasks

```
1 #include <stdio.h>
2 #include <math.h>
3 /* Function 1 : Test whether a character is a vowel */
4 int isVowel( int c)
5 {   return (c=='a' || c=='A' || c=='e' || c=='E' ||
6         c=='i' || c=='I' || c=='o' || c=='O' ||
7         c=='u' || c=='U');
8 }
9 /* Function 2: sum divisors of an integer */
10 int sumDivisors(int n)
11 {   int S=0, i;
12     for (i=1; i<=n/2; i++)
13         if (n%i==0) S+=i;
14     return S;
15 }
16 /* Function 3: Test whether an integer is prime or not */
17 int isPrime(int n)
18 {   int sqrtn = sqrt(n), i;
19     for (i=2; i<=sqrtn; i++)
20         if (n%i==0) return 0;
21     return 1;
22 }
```

```
int isPrime(int n)
{   int sqrtn= sqrt(n), result=1, i;
    for (i=2; i<=sqrtn && result==1; i++)
        if (n%i==0) result=0;
    return result;
}
```



# Program using menu for some tasks

```
23 /* Function for menu */
24 int menu()
25 { int choice;
26   printf ("\n1- Test vowel for a character");
27   printf ("\n2- Sum of divisors");
28   printf ("\n3- Test prime for an integer");
29   printf ("\nOthers: Quit");
30   printf ("\nPlease choose:");
31   scanf ("%d", &choice);
32   return choice;
33 }
34 int main()
35 { int userChoice;
36   int n; /* integer inputted */
37   char c; /* character inputted */
38   do
39   { userChoice = menu();
40     switch( userChoice)
41     { case 1: printf("Enter one character:");
42         fflush(stdin); /* clear the keyboard buffer*/
43         c= getchar();
44         if (isVowel(c)==1) printf("It is a vowel\n");
45         else printf("It is not a vowel\n");
46         break;
```

# Program using menu for some tasks

```
47     case 2: do
48         { printf ("Input a positive integer:");
49           scanf("%d", &n);
50         }
51         while (n<=0);
52         printf("Sum of it's divisors:%d\n", sumDivisors(n));
53         break;
54     case 3: printf ("Input an integer:");
55             scanf("%d", &n);
56             if (isPrime(n)==1) printf("It is a prime.\n");
57             else printf("It is not a prime.\n");
58             break;
59     default: printf("Good bye!");
60 }
61 }
62 while (userChoice>=1 && userChoice<4);
63 return 0;
64 }
```