

DATA STRUCTURE & PROGRAMMING II

Chapter 7- Recursive function



Lecture overview

□ Overall lectures

1. Introduction to algorithm
2. Basic data types and statements
3. Control structures and Loop
4. Array
5. Data structure
6. Sub-programs

7. *Recursive*



8. Pointers
9. Linked Lists
10. Stacks and Queues
11. Sorting algorithms
12. Trees

C++

Outline

- Review on Function
- What is recursive function?
- How to use recursive
- Examples and Practices

Review on Function

Function

□ What and Why?

- A function is a block of code that performs a specific task. A function may return a value.
- Why function?
 - ✓ A large and complex program can be divided into smaller programs
 - ✓ assign task and work in team is easy
 - ✓ Program is easy to understand, fix error, and maintain
 - ✓ Reusable code
 - ✓ A function can be call again and again anytime and anywhere in the program

Function

□ Type of function

- There are two types of functions in C programming (likewise, in C++):

1. Standard library functions

- Also called: built-in function, or existing function
- For example, see Table 1

Library	Provided functions
stdio.h	printf(), scanf()
math.h	pow(), sqrt(), ceil(), floor(), cos(), sin()

2. User-defined functions

- Also called: user-custom function
- For example, a user (programmer) defines his/her own function to do something

Syntax C language

❑ Create Function with returning value

```
type functionName(type parameter1){  
    ... ..  
    ... ..  
    return value;  
}
```

a) Type of value returning
from the function

d) Return value
It must be the same data type as a)

c) Parameter of a function
It has parameter type and parameter name
If have more than one, separate them by comma

b) Name of function
it should start with a verb

```
int sumTwoNumber(int n1, int n2){  
    int s;  
    s=n1+n2;  
    return s;  
}
```

Ex1: Create a function to do
summation of two numbers

Syntax C language

❑ Create Function with no returning value

```
void functionName(type parameter1){  
    ... ..  
    ... ..  
    return value;  
}
```

a) Void means no returning value from the function

d) No need to *return value* since we use *void* in a)

c) Parameter of a function

It has parameter type and parameter name
If have more than one, separate them by comma

b) Name of function

it should start with a verb

```
void greetMessage(char name[20]){  
    printf("Hi, %s", name);  
    printf("Welcome back!");  
}
```

Ex1: Create a function to display a greet message for a person

Example of functions

□ Returning Vs. Non-returning value function

- Function with returning value

```
int sum(int a, int b){  
    int res=0;  
  
    res=(a+b)*2;  
    n++;  
    return res;  
}
```

- Function with no returning value

```
void display(int a){  
    printf("Hello %d\n", a);  
    printf("This function has no  
    returning value");  
    printf("use the keyword void");  
}
```

Syntax C language

❑ How to execute the created function?

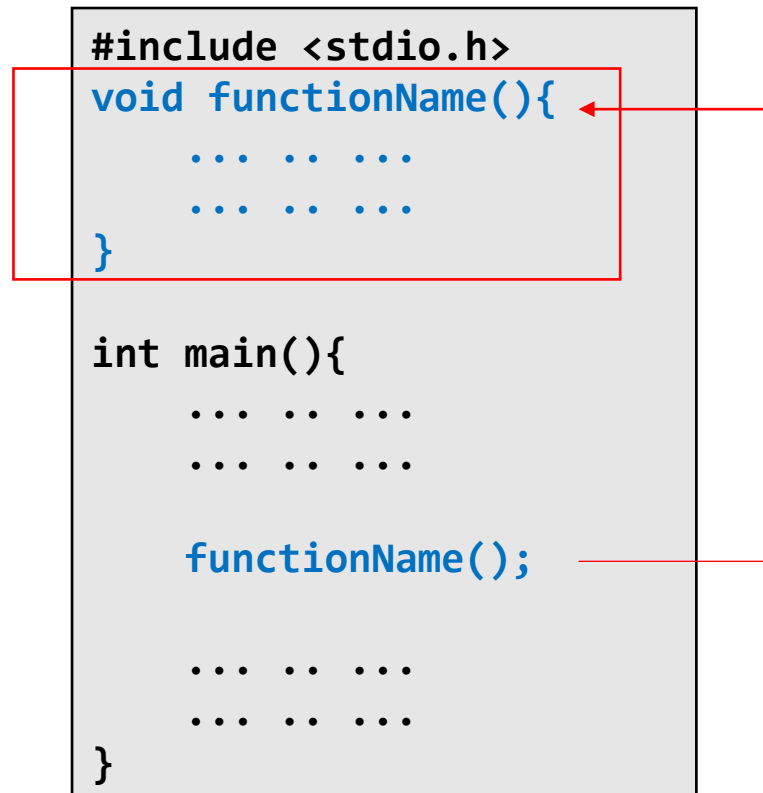
- The execution of a C program begins from the **main()** function.
- Our user-defined function can be executed by calling it from **main()** function

```
#include <stdio.h>
void functionName(){
    ... ..
    ... ..
}

int main(){
    ... ..
    ... ..

    functionName();

    ... ..
    ... ..
}
```

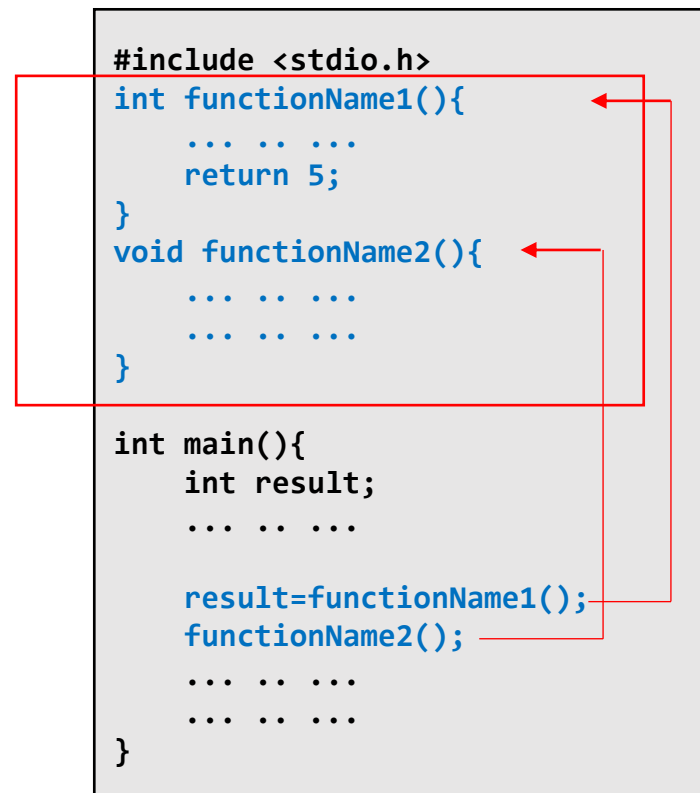
A diagram illustrating a C program where a user-defined function is called from the main function. A red box highlights the definition of 'void functionName()' at the top. A red line extends from this box down to a call 'functionName();' inside the 'int main()' function, showing the execution flow from the definition to the call.

```
#include <stdio.h>
int functionName1(){
    ... ..
    return 5;
}
void functionName2(){
    ... ..
    ... ..
}

int main(){
    int result;
    ... ..

    result=functionName1();
    functionName2();

    ... ..
    ... ..
}
```

A diagram illustrating a C program where multiple user-defined functions are called from the main function. Red boxes highlight the definitions of 'int functionName1()' and 'void functionName2()'. Red lines extend from these boxes down to their respective calls 'result=functionName1();' and 'functionName2();' inside the 'int main()' function, showing the execution flow.

Function

□ Global Vs. Local variable

- **Local variable** a variable that creates inside a function

- ✓ Its value can not be accessed from outside the function
- ✓ Its value will be destroyed after the function finish executing

- **Global value** is variable that creates outside the function below all library inclusion

- ✓ Its value can be accessed from any functions
- ✓ Its value will be destroyed after the whole program finish executing

The diagram illustrates the scope of variables in a C program. It features a code block with the following C code:

```
#include <stdio.h>

int n=0;
char name[10];
int sum(int a, int b){
    int res=0;

    res=(a+b)*2;
    n++;
    return res;
}

int main(){
    int num=10;
    int res=2;

    printf("*Start program \n");
    printf("%d", sum(2,5));
    res = sum(2,5);
    printf("%d", res);
    printf("%d", n);
}
```

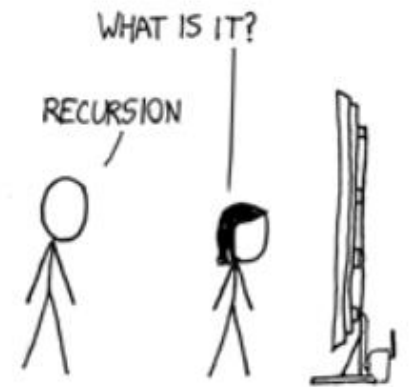
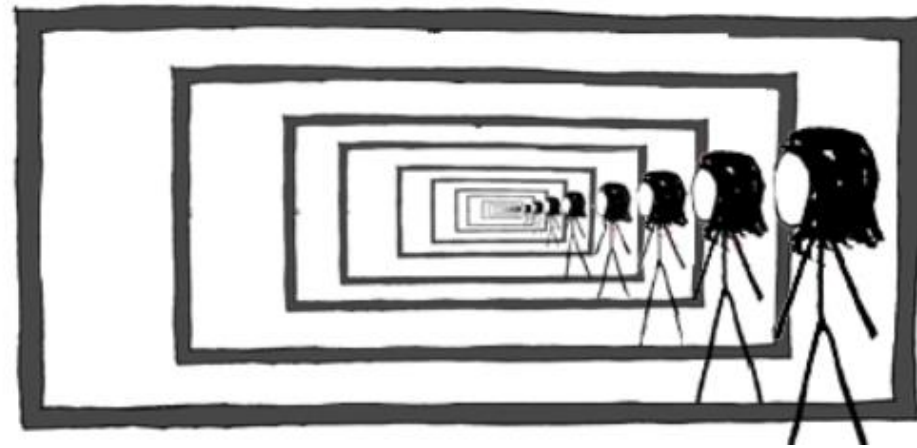
Annotations and their targets:

- Global variables** (red arrow) points to the global variables `int n=0;` and `char name[10];`.
- User-defined Function** (blue arrow) points to the `sum` function definition.
- Local variables** (red arrow) points to the local variables `int res=0;` inside the `sum` function and `int num=10;` and `int res=2;` inside the `main` function.
- Call created function** (red arrow) points to the call `sum(2,5)` inside the `main` function.

Output:

```
*Start program
14
14
2
```

Recursive Function



Recursive

□ Definition

- A **recursion** is an algorithm which calls itself with smaller (or simpler) input values
- In programming languages, ***recursion*** is the ability given to a function to call itself
- To write a recursive program, the following steps are needed

1. Find a recursive decomposition of the problem (find cases which allow to call itself again)
 - Express solution in general case
2. Find the stop condition (find base case)
 - Check that the stop condition is reached after a finite number of recursive calls in all cases

Recursive

□ Definition

- **Recursive function** is a function that calls itself repetitively until certain condition is satisfied.
It works like a loop
- Component of recursive function
 - A condition to determine when the function should call itself again (keep repetition) [repeat case or general case]
 - A condition to determine when the function should not call itself again (stopping condition) [base case]

Remark: When call itself again, we should update its parameter in order to reach the stopping condition

- **If-else** statement
- A **return** statement

Recursive

□ Example of recursion in factorial

- The factorial of a positive integer n , denoted by
 - $n! = 1*2*...*(n-1)*n = (n-1)! * n$
- To calculate $n!$, we just know how to calculate $(n-1)!$ then multiple by n
- The subproblem $(n-1)!$ is the same as the initial problem, BUT a simpler case since $n-1 < n$

Recursive

□ Example of recursion in calculating a factorial number

■ Calculate factorial

- $n! = 1$ if $n \leq 1$
- $n! = ((n-1)!) * n$ if $n > 1$

How does it work

✓ fac(4) ←

✓ return fac(3)*4=24 ←

✓ return fac(2)*3=6 ←

✓ return fac(1)*2=2 ←

✓ return 1 ←

```
Function fac(n: integer): integer
Begin function
    if(n<=1) then
        return 1
    else
        return (fac(n-1))*n
    end if
End function
```

An example of a recursive function to calculate a factorial number

Recursive

□ Example of recursion in Suit Fibonacci

- Suit Fibonacci

- $U_n = 1$ if $n \leq 2$
- $U_n = U_{n-1} + U_{n-2}$ if $n \geq 3$

```
Function fibonacci(n: integer): integer
Begin function
    if(n==1 OR n==2) then
        return 1
    else
        return fibonacci(n-1)+fibonacci(n-2)
    end if
End function
```

An example of a recursive function to calculate the n^{th} element of suit fibonacci

Recursive

□ Example of recursion Summation

■ Sum

- $F(n) = n + f(n-1)$ when $n > 1$
- $F(n) = 1$ when $n = 1$

```
Function sum(n: integer): integer
Begin function
    if(??) then
        return ??
    else
        return ??
    end if
End function
```

Using recursive to find summation

$$S = 1 + 2 + 3 + \dots + n$$

An example of a recursive function to
computer summation

Recursive

□ Remark

- **Base case** is a case that it does not call to the main problem. It stops recursive
- In each call of recursive, it should tend to reach the base case.



Recursive

□ Practice: Write a C++ program using recursive to

1. Calculate the summation $(2 + 4 + 6 + \dots + n)/n$ using recursive where n is a number input by a user
2. Display the message Hello 1, Hello 2, ..., Hello n , where n is an input given from user.
3. Count the number of digits in an integer input by a user using recursive

Recursive: Direct and Indirect

□ Definition

- Direct recursive
 - Subproblem uses the main problem to call for defining the result
- Indirect recursive
 - Subproblem A calls subproblem B
- Example

$f(x)=1$	if $x=1$ or $x=2$
$f(x)=f(x-1)*f(x-2)$	if $x>2$

Direct recursive

$f(x)=1$	if $x=1$
$f(x)=g(x)+2$	if $x>1$
$g(x)=0$	if $x=2$
$g(x)=f(x)-1$	if $x>2$

Indirect recursive

Q & A

Practices

□ Write a subprogram using recursive to

1. Calculate summation of $S = 1^2 + 2^2 + 3^2 + \dots + n^2$, where n is an integer number input by user
2. Calculate power n of an integer m denoted by m^n

Hint: $m^n = \underbrace{m * m * \dots * m}_n$

3. Calculate the multiplication between 2 positive integers using this method:

$$m * n = \underbrace{m + m + \dots + m}_n$$

4. Find the minimum value of an array

Practice

□ Write a subprogram (not use loop) using recursive to

5. Display n stars (*) where n is a positive integer number input by a user
6. Compute the sum of all elements in an array, where the array and its size are given as parameters

Hint: The subprogram looks like: `function sum(ar[]: integer): integer`

7. Check how many times a character appears in a string
8. Find the sum of digits of a number

Q & A