Programming Fundamentals

Module D - Functions

Le The Anh
anhlt161@fe.edu.vn

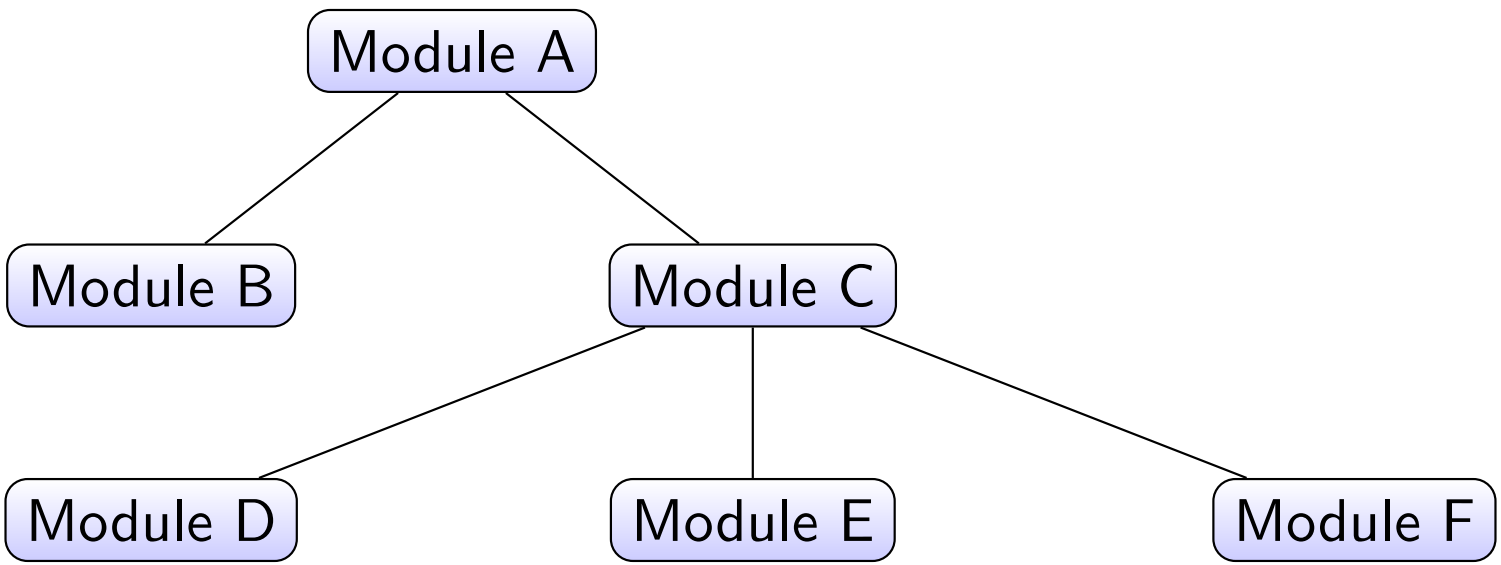
FPT UNIVERSITY

Objectives

1. Modularity
   - Structured Design
   - In-Class Practice
   - Functions
   - Prototypes
   - Style
2. Scope of a Variable
   - Local or Global Variables
   - Basic Validation
   - Visibility
   - Walkthroughs with Functions
3. Pointers
   - What is a Pointer
   - How to use Pointers
   - Pointer Arithmetic Operators
   - Pointer Comparison Operators

Structured Design

- In designing a program, we subdivide the problem conceptually into a set of design units. We call these design units modules. In subdividing the problem, we reduce the number of factors with which to deal simultaneously.
- Advantages of doing this:
  - Easy to understand the system
  - System maintenance is easy.
  - A module can be used many times, even in other tasks.



Module Designing

- We select each module so that it
  - is easy to upgrade,
  - contains a readable amount of code, and
  - can be used as a part of the solution to some other problem.
- In designing a module, we
  - select a descriptive identifier for the module,
  - identify the tasks to be performed within the module.
- In a structured design, each module should has
  - one entry point and one exit point,
  - a high cohesion and low coupling.

Cohesion

- Cohesion is a measure of the focus within a module.
- A module is highly cohesive if it performs a single task.
- A module is low in cohesion if it performs a collection of unrelated tasks.
- 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.

Degree of Cohesion

- low cohesion - generally unacceptable
  - “coincidental” - unrelated tasks
  - “logical” - related tasks of which only one is performed - the module identifier suggests a choice
  - “temporal” - multiple logically unrelated tasks that are only temporally related
- 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

Low Cohesion Example

- The degree of cohesion for a module that
  - calculates the PST and GST on a sale,
  - checks if a barcode is valid, and
  - raises an integer to the power of another integer
- The module has low cohesion. For an improved design, we identify each task as a separate module.

Coupling

- Coupling is a measure of the degree of inter-relatedness 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.
- In designing for low coupling, we ask what kind of data to avoid passing to the module.

Coupling Classification

The data classifications include (from low to high coupling):

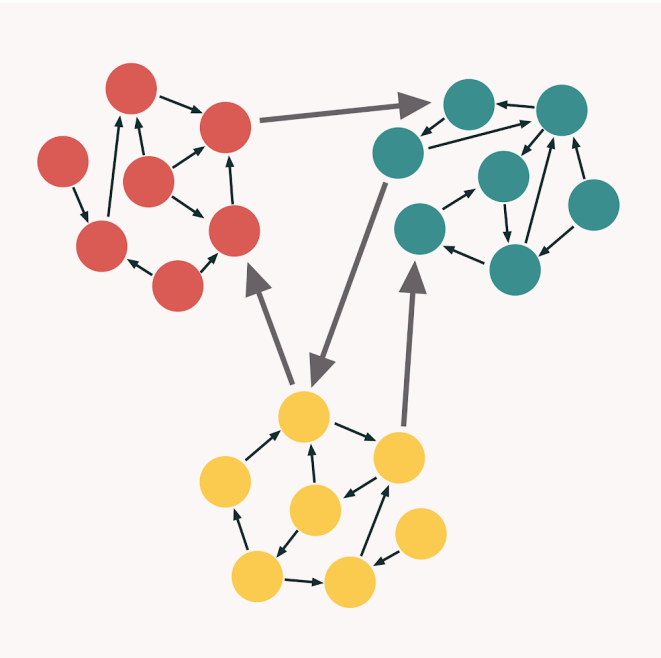
1. "data" - used by the module but not to control its execution
2. "control" - controls the execution of the module
3. "external" - part of an environment external to the module that controls its execution
4. "common" - part of a global set of data
5. "content" - accesses the internals of another module

Highly Coupled Example

- A control flag that tells a module whether to accept
  - integer input from the user, or
  - floating-point input from the user
- Controls the execution of the module from outside the module. A module that accepts such a control flag is highly coupled. For an improved design, we create two separate modules - one that accepts integer input and one that accepts floating-point input, neither of which receives any control flag.

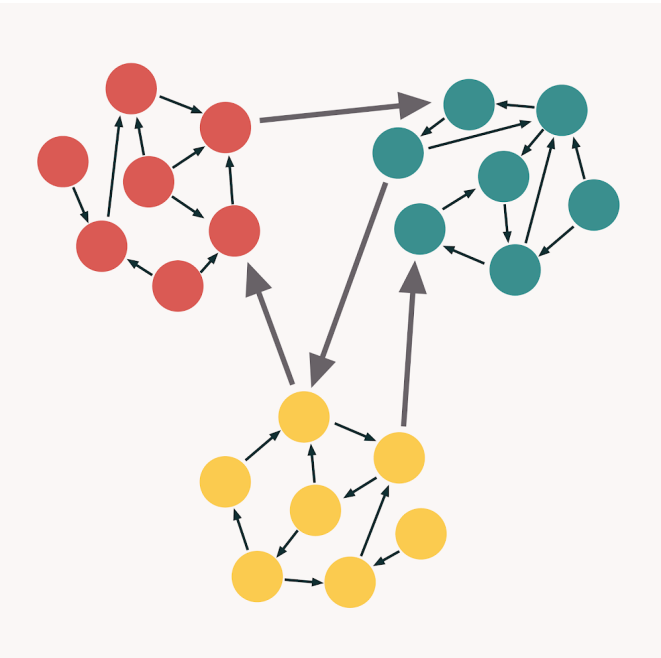
Cohesion vs Coupling

- Coupling is the measure of the degree of interdependence between the modules.
- Cohesion is a measure of the degree to which the elements of the module are functionally related.
- Coupling shows the relationships between modules while cohesion shows the relationships within the module.
- Good design = low coupling + high cohesion



Coupling and Cohension (Software Quality Metrics)

- 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.
- A highly cohesive module performs a single task, whereas a lowly cohesive module performs some unrelated tasks.
- Good design = low coupling + high cohesion



Code Example

```
void prime_counter(){
    int a, res=0;
    printf("Enter a number: ");
    scanf("%d", &a);
    for(int i=2; i < a; i++){
        int flag = 1;
        for(int j=2; j <= sqrt(i); j++){
            if (i%j == 0) {
                flag = 0;
                break;
            }
        }
        res += flag;
    }
    printf("Number of primes less than %d is %d", a, res);
}
```

Code Example

```
int is_prime(int i){
    for(int j=2; j <= sqrt(i); j++){
        if (i%j == 0) {
            return 0;
        }
    }
    return 1;
}

int prime_counter(int a){
    int res = 0;
    for(int i=2; i < a; i++){
        res += is_prime(i);
    }
    return res;
}
```

In-Class Practice

Design a program that either calculates the value of an integer raised to the power of an integer exponent or the arithmetic mean of a series of integers, depending upon a choice made by the user.

- List all of the tasks that the program should perform to solve this problem
- Identify the modules for the problem structure
- Check that each module is high in cohesion
- Check that each module is low in coupling

Functions

- In C, modules are implemented by functions.
- Functions may receive data and return a value.
- Function = {Header + Body}
  - The header includes the module name and the list of parameters that receive the module data enclosed within parentheses.
  - The body is the code block containing the module instructions and generates the return value.

Function Definition

- Syntax:

```
return_data_type function_name(data_type param_1,
                               data_type param_2, ...){
    /* declarations here */
    /* logical constructs here */
    return /* return expression here */;
}
```

  where

  - return_data_type: the data type of the value returned by the function.
  - function_name: the name of the module.
  - data_type, param_1, param_2: the data type and the names of the parameters.

- Example

```
int sum(int a, int b)
{
    return a + b;
}
```

void Functions

- 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 procedure in other languages.

```
void disp(){
    printf("I'm a 'void' function.");
}
```

Calls

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

```
function_name(arg_1,arg_2,...);
```

- Example:

```
c = sum(3, 7);
```

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

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.

```
int main()
{
    // do something
    return 0;
}
```

General Form of a C Program

```
Global declarations

return_data_type func_1(parameter list){
    statement sequence
}

return_data_type func_2(parameter list){
    statement sequence
}

...

main(parameter list){
    statement sequence
}
```

Pass By Value

- Pass-by-value: 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.
- Pass-by-reference: Lets the function knows the memory location of the parameter and therefore can directly modify the parameter.

```
void test(int a){
    a -= 1;
}

void main(){
    int a=1;
    test(a);
    printf("%d", a); // a = 1
}
```

Coercion

- 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 the value of the argument into the data type of the parameter.

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

int main(){
    int k1,k2,h;
    k1 = pow(2.6, 4 );
    printf("k = %d\n", k1); //45
    float x = pow(2.6, 4);
    printf("x = %f\n", x);  //45.697601
    h = 2.6;
    k2 = pow(h, 4);
    printf("k = %d\n", k2); //16
    return 0;
 }
```

In-Class Practice

- Code the function calls and headers for your design of the power/arithmetic-mean program.

Prototypes

- Function prototypes describe the form of a function without specifying the implementation details.
- To validate the number of arguments, the compiler needs to know the number of parameters.
- To coerce argument values, the compiler needs to know the data types of the parameters.
- To determine the data type of the call expression, the compiler needs to know the return data type of the function.
- A function prototype provides these three pieces of information.
- Function prototypes are placed near the head of our source file and before any function definitions.

```
return_data_type function_name(data_type param_1,
                               data_type param_2, ...);
```

Function Prototype

```c
#include <stdio.h>

int sum(int a, int b);
void disp(int a);

int main(){
    int a = 1, b = 2;
    int c = sum(a, b);
    disp(c);
    return 0;
}

int sum(int a, int b){
    return a + b;
}

void disp(int a){
    printf("%d", a);
}
```

Include Header Files

- We can collect function prototypes into a file and refer to the file in our program. Such a file is called a header file.
- 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 current directory
#include <filename> // in system directories
```

- Example:

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

void main()
{
    printf("sqrt(%d) = %.0lf", 4, sqrt(4)); // sqrt(4) = 2
}
```

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

Summary

- Modularity
  - Structured Design
  - In-Class Practice
  - Functions
  - Prototypes
  - Style

Q&A

Scope of a Variable

- The status of any variable in a program is described by two aspects: its extent and its scope.
- Its extent is that part of the program over which the variable has memory allocated for it; that is, the lifetime or storage duration of the variable.
- The scope of a variable is that part of a program over which the variable is visible; that is, the part over which expressions can access the variable.
- The extent of a variable sets the limits on its scope. Variables that are defined within any function are limited in scope to the function within which they are defined. They are invisible outside the function.

Local or Global Variables

- A variable that is defined within either a function or code block is a local variable.
- A variable that is declared outside any function including main is a global variable. The compiler allocates memory for global variables separately and that memory cannot be shared with any other variable. Global variables introduce a high degree of coupling, avoid using global variables altogether.

```c
#include <stdio.h>

int a = 1; // global variable

void func()
{
    int b = 2; // local variable
    printf("%d", a + b);
}

void main()
{
    func();
}
```

Basic Validation

- Recall our program to calculate the value of an integer raised to the power of another integer.

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

int main (void) {
    int base, exp, answer;

    base = 2;
    exp = 4;
    answer = power (base, exp);
    printf("%d^%d is %d\n", base, exp, answer);

    return 0;
}
```

Basic Validation

```c
/* power returns the value of base raised to
*  the power of exponent (base^exponent)
*  power assumes that the exponent is
*  non-negative
*/
int power (int base, int exponent) {
    int result, i;

    result = 1;
    for(i = 1; i <= exponent; i++)
        result = result * base;

    return result;
}
```

- Write a function to validate the input data to accept only integer numbers in a specific range.

Basic Validation

```c
int get_int(int min, int max)
{
    int n, ok;
    char c;

    do
    {
        ok = 0;
        printf("Enter an integer: ");
        if (scanf("%d%c", &n, &c) != 2 || c !='\n')
        {
            printf("Invalid input. Try again: \n");

            // flush the remaining characters
            fflush(stdin);
        }
        else if (n < min || n > max)
        {
            printf("%d is outside [%d, %d]\n", n, min, max);
        }
        else
            ok = 1;
    } while (ok == 0);

    return n;
}
```

Visibility

- A local variable is visible from its declaration to the end of the code block within which that variable was declared.

```c
#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
********************/
```

Visibility

- The declaration of one variable can refer to the value of a yet to be shadowed variable. For example,

```c
#include <stdio.h>

int main ( ) {
    int input;

    printf("Enter a value : ");
    scanf("%d", &input);
    if ( input > 10) {
        int input = input * 2;  /* 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 24
The value is 12
********************/
```

Walkthroughs with Functions

- The walkthrough table for a program with several functions groups the local variables under their parent function names.

-- function name --			int main()		
data type variable z	... ...	data type variable a	data type variable z	... ...	data type variable a
initial value	...	initial value	initial value	..	initial value
next value	...	next value	next value	...	next value
next value	...	next value	next value	...	next value
			next value	...	next value
next value	...	next value	next value	...	next value
next value	...	next value	next value	...	next value
			next value	...	next value
			next value	...	next value

Output: write the output here (line by line)

Summary

- Scope of a Variable
  - Local or Global Variables
  - Basic Validation
  - Visibility
  - Walkthroughs with Functions (read at home)
  - Practice (home work)

Q&A

Pointers

- Explain what a pointer is and where it is used
- Explain how to use pointer variables and pointer operators
- Assign values to pointers
- Explain pointer arithmetic
- Explain pointer comparisons

What is a Pointer?

- A pointer is a variable, which contains the address of a memory location of another variable.
- If one variable contains the address of another variable, the first variable is said to point to the second variable.
- A pointer provides an indirect method of accessing the value of a data item.
- Pointers can point to variables of other fundamental data types like int, char, or double or data aggregates like arrays or structures.

What are Pointers used for?

Some situations where pointers can be used are

- To return more than one value from a function
- To pass arrays and strings more conveniently from one function to another
- To manipulate arrays easily by moving pointers to them instead of moving the arrays itself
- To allocate memory and access it (Direct Memory Allocation)

Variable Address and Pointer

- Given a variable a, operator &a returns the address of a. We can use %p to print out the hexadecimal format of the memory address pointed by a pointer.

```
int a;
scanf("Enter an integer: %d", &a);
printf("Value of variable a is %d", a);
// Address of variable a is 0x7ffc1eec964c
printf("Address of variable a is %p", &a);
```

- Pointers are special variables used to store addresses rather than values. Declaration syntax:

```
data_type *pointer_name;

char *ch;
int *i;
float *f;
```

Pointer Operators

- Assign an address of a variable to a pointer

```
int a = 1, b = 2;
int *p;
p = &a;
```

- Get the variable pointed by a pointer

```
*p += 1;
*p = *p + b;
printf("%d\n", *p); // 4
```

- Assign a pointer to another pointer:

```
q = p;
printf("%p %p -> %d", p, q, *q);
```

- Set value of a pointer to `NULL` when it points to nothing. The `NULL` constant is defined in `<stdio.h>` or `<stddef.h>`

```
p = NULL;
```

Pass-by-Value vs. Pass-by-Reference

```c
#include <stdio.h>

// passing by value
void func_1(int a)
{
    a += 1;
}

// passing by reference
void func_2(int *a)
{
    *a += 1;
}

void main()
{
    int a = 1;

    func_1(a);
    printf("a = %d\n", a); // a = 1

    func_2(&a);
    printf("a = %d\n", a); // a = 2
}
```

Swap Function using Pointer

```c
#include <stdio.h>

void swap(int *a, int *b)
{
    int value = *a;
    *a = *b;
    *b = value;
}

int main()
{
    int a = 1, b = 2;
    swap(&a, &b);
    printf("%d %d", a, b);

    return 0;
}

// Output: 2 1
```

Pointer Arithmetic Operators

- Pointers in C support only two arithmetic operators: addition, subtraction. These operators change the memory address contained within a pointer, based on the pointer data type.

```
char ch = 'a', *c;
long long a = 1, *ll;
long double b = 2, *ld;

c = &ch;
ll = &a;
ld = &b;

printf("%lu %lu %lu", sizeof(char), sizeof(long long),
       sizeof(long double)); // 1 8 16

printf("%p %p\n", c, c+1); // 0x7ffced41076f 0x7ffced410770
printf("%p %p\n", ll, ll+1); // 0x7ffced410770 0x7ffced410778
printf("%p %p\n", ld, ld+1); // 0x7ffced410790 0x7ffced4107a0
```

Pointer Arithmetic Operators

```c
int n = 10;
int *a = &n;

printf("%p -> %d\n", a, *a); // 0x7ffdbb956fe4 -> 10
a = a + 1;
printf("%p, %lu\n", a, sizeof(int)); // 0x7ffdbb956fe8, 4
a = a - 1;
a += 2;
a -= 2;
a++; ++a;
a--; --a;

printf("%p -> %d", a, *a); // 0x7ffdbb956fe4 -> 10
```

Pointer Comparison

- Two pointers can be compared in a relational expression provided both the pointers are pointing to variables of the same type.
- Consider that ptr_a and ptr_b are 2 pointer variables, which point to data elements a and b. In this case the following comparisons are possible:

ptr_a < ptr_b	Returns true provided **a** is stored before **b**
ptr_a > ptr_b	Returns true provided **a** is stored after **b**
ptr_a <= ptr_b	Returns true provided **a** is stored before **b** or ptr_a and ptr_b point to the same location
ptr_a >= ptr_b	Returns true provided **a** is stored after **b** or ptr_a and ptr_b point to the same location.
ptr_a == ptr_b	Returns true provided both pointers ptr_a and ptr_b points to the same data element.
ptr_a != ptr_b	Returns true provided both pointers ptr_a and ptr_b point to different data elements but of the same type.
ptr_a == NULL	Returns true if ptr_a is assigned NULL value (zero)

Summary

- Pointer
  - Explain what a pointer is and where it is used
  - Explain how to use pointer variables and pointer operators
  - Assign values to pointers
  - Explain pointer arithmetic
  - Explain pointer comparisons

Q&A