PROGRAMMING METHODOLOGY
(PHƯƠNG PHÁP LẬP TRÌNH)

UNIT 14: Functions with Pointer Parameters

Unit 14: Functions with Pointer Parameters

Objectives:

- How to use pointers to return more than one value in a function

Reference:

- Chapter 5 Functions: Lessons 5.4 – 5.5

Unit 14: Functions with Pointer Parameters

1. Introduction (1/4)

- In Unit #4, we learned that a function may return a value, or it may not return any value at all (void function)
- Is it possible for a function to return 2 or more values?
- Does the following function f(*n*) return both 2*n* and 3*n*?

```
int f(int n) {
   return 2 * n;
   return 3 * n;
}
```

- No, f(*n*) returns only 2*n*.
- Once a return statement is executed, the function terminates immediately.

1. Introduction (2/4)

- Below is a program that swaps two variables:

```
#include <stdio.h>

int main(void) {
   int var1, var2, temp;

   printf("Enter two integers: ");
   scanf("%d %d", &var1, &var2);

   // Swap the values
   temp = var1;
   var1 = var2
   var2 = temp;

   printf("var1 = %d; var2 = %d\n", var1, var2);
   return 0;
}
```

```
Enter two integers: 72 9
var1 = 9; var2 = 72
```

Unit14_Swap_v1.c

1. Introduction (3/4)

- This is a modularised version of the previous program:

```
#include <stdio.h>

void swap(int, int);

int main(void) {
   int var1, var2;

   printf("Enter two integers: ");
   scanf("%d %d", &var1, &var2);

   swap(var1, var2);

   printf("var1 = %d; var2 = %d\n", var1, var2);
   return 0;
}

void swap(int para1, int para2) {
   int temp;
   temp = para1; para1 = para2; para2 = temp;
}
```

```
Enter two integers: 72 9
var1 = 72; var2 = 9
```

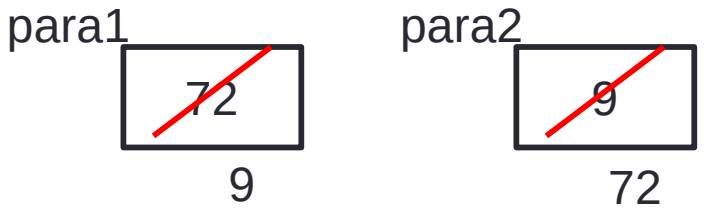
Unit14_Swap_v2.c

1. Introduction (4/4)

- What happens in Unit14_Swap_v2.c?
- It's all about pass-by-value and scope rule! (See Unit #4)

In main():

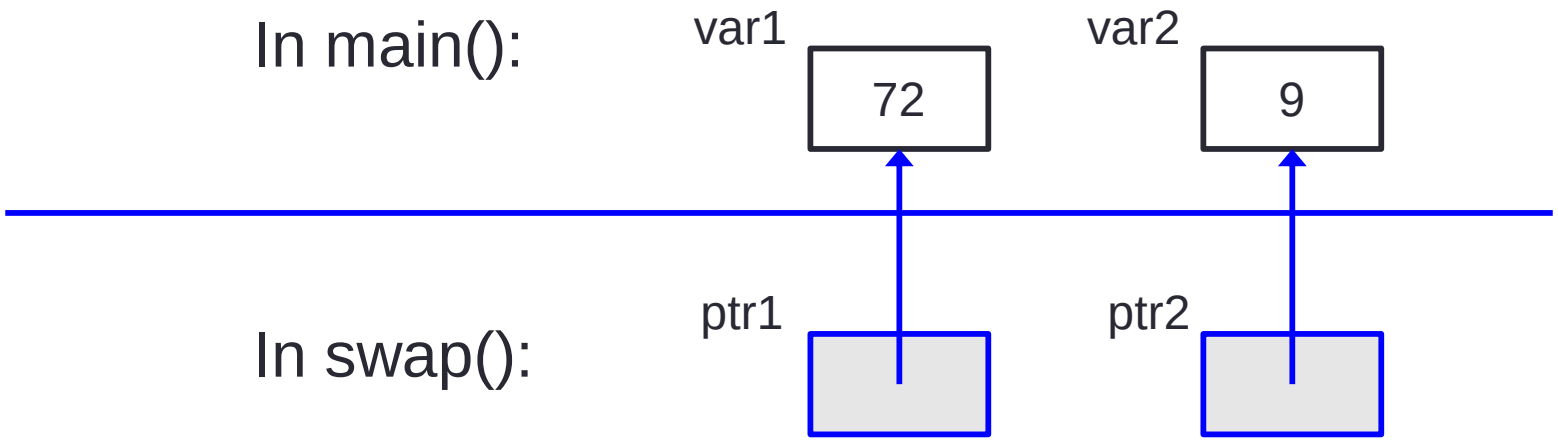
In swap():



- No way for swap() to modify the values of variables that are outside its scope (i.e. var1 and var2), unless...

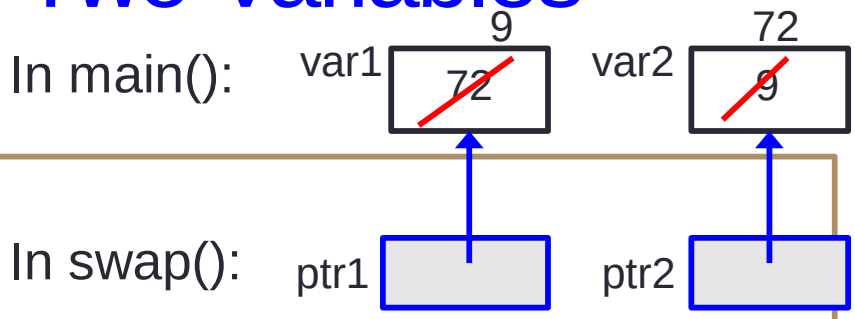
2. Functions with Pointer Parameters

- The only way for a function to modify the value of a variable outside its scope, is to find a way for the function to access that variable
- Solution: Use pointers! (See Unit #8)



2.1 Function to Swap Two Variables

- Here's the solution




```
#include <stdio.h>

void swap(int *, int *);

int main(void) {
   int var1, var2;

   printf("Enter two integers: ");
   scanf("%d %d", &var1, &var2);

   swap(&var1, &var2);

   printf("var1 = %d; var2 = %d\n", var1, var2);
   return 0;
}

void swap(int *ptr1, int *ptr2) {
   int temp;
   temp = *ptr1; *ptr1 = *ptr2; *ptr2 = temp;
}
```

Unit14_Swap_v3.c

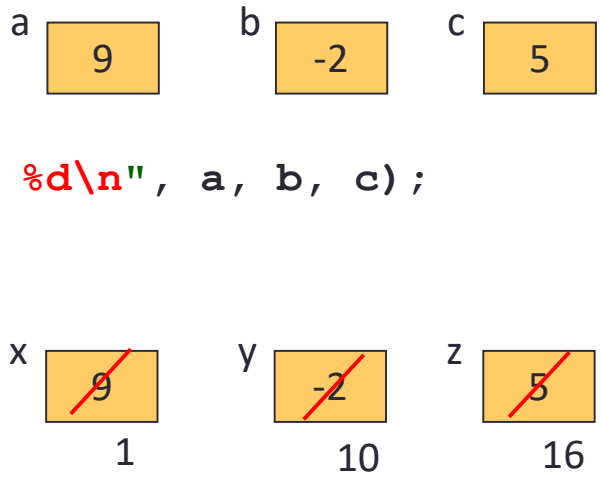
2.2 Examples (1/4)

Unit14_Example1.c

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

int main(void) {
    int a = 9, b = -2, c = 5;
    f(a, b, c);
    printf("a = %d, b = %d, c = %d\n", a, b, c);
    return 0;
}

void f(int x, int y, int z) {
    x = 3 + y;
    y = 10 * x;
    z = x + y + z;
    printf("x = %d, y = %d, z = %d\n", x, y, z);
}
```




```
x = 1, y = 10, z = 16
a = 9, b = -2, c = 5
```

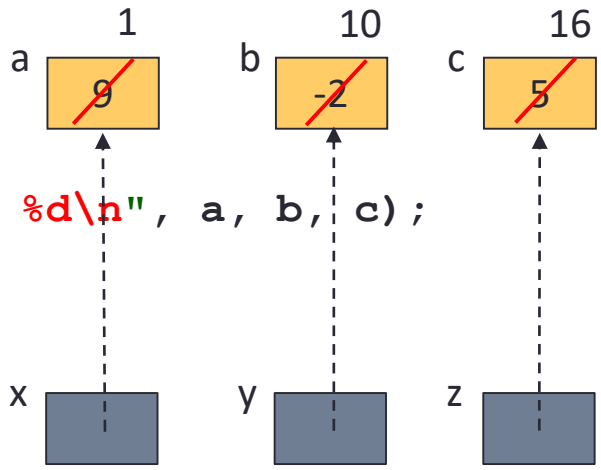
2.2 Examples (2/4)

Unit14_Example2.c

```
#include <stdio.h>
void f(int *, int *, int *);

int main(void) {
    int a = 9, b = -2, c = 5;
    f(&a, &b, &c);
    printf("a = %d, b = %d, c = %d\n", a, b, c);
    return 0;
}

void f(int *x, int *y, int *z)
{
    *x = 3 + *y;
    *y = 10 * *x;
    *z = *x + *y + *z;
    printf("*x = %d, *y = %d, *z = %d\n", *x, *y, *z);
}
```



*x is a, *y is b, and *z is c!

```
*x = 1, *y = 10, *z = 16
a = 1, b = 10, c = 16
```

2.2 Examples (3/4)

Unit14_Example3.c

```c
#include <stdio.h>
void f(int *, int *, int *);

int main(void) {
    int a = 9, b = -2, c = 5;
    f(&a, &b, &c);
    printf("a = %d, b = %d, c = %d\n", a, b, c);
    return 0;
}

void f(int *x, int *y, int *z)
{
    *x = 3 + *y;
    *y = 10 * *x;
    *z = *x + *y + *z;
    printf("x = %d, y = %d, z = %d\n", x, y, z);
}
```

Compiler warnings, because x, y, z are NOT integer variables!
They are addresses (or pointers).

2.2 Examples (4/4)

Unit14_Example4.c

```c
#include <stdio.h>
void f(int *, int *, int *);

int main(void) {
    int a = 9, b = -2, c = 5;
    f(&a, &b, &c);
    printf("a = %d, b = %d, c = %d\n", a, b, c);
    return 0;
}

void f(int *x, int *y, int *z)
{
    *x = 3 + *y;
    *y = 10 * *x;
    *z = *x + *y + *z;
    printf("x = %p, y = %p, z = %p\n", x, y, z);
}
```

Use %p for pointers.

Addresses of variables a, b and c.
(Values change from run to run.)

```
x = ffbff78c, y = ffbff788, z = ffbff784
a = 1, b = 10, c = 16
```

3. Design Issues

- We will discuss some design issues relating to the use of pointer parameters.
  - When should pointer parameters be avoided
  - Situations when the use of pointer parameters may violate cohesion

3.1 When Not to Use Pointer Parameters

- Both programs are correct, but which is preferred? Why?

(A)

Unit14_Print_v1.c

```
int main(void) {
  int num1 = 1, num2 = 2;
  print_values(num1, num2);
  return 0;
}

void print_values(int n1, int n2) {
  printf("Values: %d and %d", n1, n2);
}
```

(B)

```
int main(void) {
  int num1 = 1, num2 = 2;
  print_values(&num1, &num2);
  return 0;
}

void print_values(int *n1, int *n2) {
  printf("Values: %d and %d", *n1, *n2);
}
```

Unit14_Print_v2.c

- (B) does not allow calls like print_values(3, 4), print_values(a+b, c*d), etc., whereas (A) does.
- Use pointer parameters only if absolutely necessary.

3.2 Pointer Parameters vs Cohesion (1/6)

- Task: find the maximum value and average of an array
- 2 versions are shown
  - Version 1: Unit14_Max_and_Average_v1.c uses 2 functions to separately compute the maximum and average.
  - Version 2: Unit14_Max_and_average_v2.c uses a single function, with pointer parameters, to return both maximum and average.

3.2 Pointer Parameters vs Cohesion (2/6)

Unit14_Max_and_Average_v1.c

```c
#include <stdio.h>

int findMaximum(int [], int);
double findAverage(int [], int);

int main(void) {
   int numbers[10] = { 1, 5, 3, 6, 3, 2, 1, 9, 8, 3 };

   int max = findMaximum(numbers, 10);
   double ave = findAverage(numbers, 10);

   printf("max = %d, average = %.2f\n", max, ave);
   return 0;
}
```

3.2 Pointer Parameters vs Cohesion (3/6)

Unit14_Max_and_Average_v1.c

```
// Compute maximum value in arr
// Precond: size > 0
int findMaximum(int arr[], int size) {
   int i, max = arr[0];
   for (i=1; i<size; i++) {
      if (arr[i] > max)
         max = arr[i];
   }
   return max;
}

// Compute average value in arr
// Precond: size > 0
double findAverage(int arr[], int size) {
   int i;
   double sum = 0.0;
   for (i=0; i<size; i++)
      sum += arr[i];
   return sum/size;
}
```

3.2 Pointer Parameters vs Cohesion (4/6)

Unit14_Max_and_Average_v2.c

```c
#include <stdio.h>

void findMaxAndAverage(int [], int, int *, double *);

int main(void) {
   int numbers[10] = { 1, 5, 3, 6, 3, 2, 1, 9, 8, 3 };
   int max;
   double ave;

   findMaxAndAverage(numbers, 10, &max, &ave);

   printf("max = %d, average = %.2f\n", max, ave);
   return 0;
}
```

3.2 Pointer Parameters vs Cohesion (5/6)

```
// Compute maximum value and average value in arr
// Precond: size > 0
void findMaxAndAverage(int arr[], int size,
                       int *max_ptr, double *ave_ptr) {
   int i;
   double sum = 0.0;

   *max_ptr = arr[0];
   for (i=0; i<size; i++) {
      if (arr[i] > *max_ptr) {
         *max_ptr = arr[i];
      }
      sum += arr[i];
   }

   *ave_ptr = sum/size;
}
```

Unit14_Max_and_Average_v2.c

3.2 Pointer Parameters vs Cohesion (6/6)

- Which version is better?

Version 1	Version 2
Uses separate functions findMaximum() and findAverage()	Uses one function findMaxAndAverage()
No pointer parameter in functions	Uses pointer parameters in function
Functions are cohesive (refer to Week 3 Exercise 4: Cohesion) because each function does one task. Allows code reusability.	More efficient because overall one loop is used to compute the results, instead of two separate loops in version 1.

- Trade-off between cohesion and efficiency.
  - At this point, we shall value cohesion more.

4 Lab #3 Exercise #2: Subsequence (1/3)

- In this exercise, you are required to compute 3 values of the solution subsequence:
  - Sum
  - Interval
  - Start position
- As the topic on pointer parameters hasn't been covered then, you are told to use a 3-element array ans to hold these 3 values.
- This is only possible because the 3 values happen to be of the same type, i.e. int.
- As arrays are actually pointers, the function sum_subsequence() is able to put the 3 answers into the array ans

4 Lab #3 Exercise #2: Subsequence (2/3)

- We modify the function to return the 3 values through 3 pointers.

Old program

```
#include <stdio.h>

int scan_list(int []);
void sum_subsequence(int [], int, int []);

int main(void) {
   int list[10], size;
   int answers[3];    // stores the required answers

   size = scan_list(list);
   sum_subsequence(list, size, answers);

   printf("Max sum ...", answers[0], answers[1], answers[2]);
   return 0;
}

void sub_subsequence(int arr[], int size, int ans[]) {
   ...
}
```

4 Lab #3 Exercise #2: Subsequence (3/3)

- We modify the function to return the 3 values through 3 pointers.

New program

```
#include <stdio.h>

int scan_list(int []);
void sum_subsequence(int [], int, int *, int *, int *);

int main(void) {
   int list[10], size;
   int sum, interval, start;

   size = scan_list(list);
   sum_subsequence(list, size, &sum, &interval, &start);

   printf("Max sum ...", sum, interval, start);
   return 0;
}

void sub_subsequence(int arr[], int size, int *sum_ptr,
                     int *interval_ptr, int *start_ptr) {
   ...
}
```

Summary

- In this unit, you have learned about
  - Using pointer parameters in functions, to allow a function to modify the values of variables outside the function