Programming for Engineers

Lecture 8: Function Pointers

Course ID: EE057IU

Instructor: Dr. Hieu Nguyen

Assistant Professor, School of Electrical Engineering

International University – VNU-HCM

Ho Chi Minh City, Fall 2024

Email: nthieu@hcmiu.edu.vn

Office Hours: Wed 1:00 PM – 4:00 PM and Thu 8:30 AM – 11:00 AM

Introduction to Function Pointers

What are Function Pointers?

- **Definition**: Variables store the address of the function
- **Purpose**: Allow dynamic calling of different functions at runtime
- **Key benefit**: Increase the flexibility and modularity of the code

Declaring syntax

- **General syntax:** return_type (*pointer_name)(parameter_types)
- **Example:** void (*$myFunc$)(int, float);
- **Explanation**: declares $myFunc$ as a pointer to a function taking int and $float$ and return $void$

Function Pointers

- **Key concepts**
  - Array Name as Address: Represents the memory address of the first element.
  - Function Name as Address: Represents the starting address of the function's code.
- **Pointers to Functions**
  - A pointer to a function holds the memory address of the function.
  - Capabilities:
    - Pass to and return from functions.
    - Store in arrays.
    - Assign to other function pointers (same type).
    - Compare for equality or inequality.

Using Function Pointers

- **Declaring**
  - int (*$myFunc$)(int, float);

  myFunc is a pointer

int *$myFunc$(int a, float b);

header of a function that return a pointer to an integer

- **Assigning**
  - $myFunc$ = &targetFunction; //address of the pointer

- **Calling**
  - $myFunc$(5, 3.2); // function call through pointer

In-class example

```c
#include <stdio.h>

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

int main() {
    int (*addPtr)(int, int) = add;
    printf("Result: %d", addPtr(10, 5));  // Output: 15
    return 0;
}
```

"**addPtr** is a pointer, which is pointing to a **function** containing two int arguments and it return an int"

"**addPtr** pointer takes the address of add() function

```
Result: 15

=== Code Execution Successful ===
```

In-class example

```c
#include <stdio.h>

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

int main() {
    int (*funcPtr)(int, int);
    funcPtr = add;  // No need for & here
    printf("Result: %d\n", funcPtr(5, 3));  // Calls `add` function
    return 0;
}
```

```c
#include <stdio.h>

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

int main() {
    int (*funcPtr)(int, int);
    funcPtr = &add;  // Using & is valid, but unnecessary
    printf("Result: %d\n", funcPtr(5, 3));  // Calls `add` function
    return 0;
}
```

```
Result: 8


=== Code Execution Successful ===
```

In-class example

```c
#include <stdio.h>

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

int subtract(int a, int b) {
    return a - b;
}

int main() {
    int (*operation)(int, int);

    operation = add;
    printf("Addition: %d\n", operation(10, 5));

    operation = subtract;
    printf("Subtraction: %d\n", operation(10, 5));

    return 0;
}
```

```
Addition: 15
Subtraction: 5


=== Code Execution Successful ===
```

Function Pointers - **Sorting in Ascending or Descending Order**

```c
// fig07_17.c
// Multipurpose sorting program using function pointers.
#include <stdio.h>
#define SIZE 10

// prototypes
void bubbleSort(int work[], size_t size, int (*compare)(int a, int b));
int ascending(int a, int b);
int descending(int a, int b);

int main(void) {
   // initialize unordered array a
   int a[SIZE] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };

   printf("%s", "Enter 1 to sort in ascending order,\n"
          "Enter 2 to sort in descending order: ");
   int order = 0;
   scanf("%d", &order);

   puts("\nData items in original order");

   // output original array
   for (size_t counter = 0; counter < SIZE; ++counter) {
      printf("%5d", a[counter]);
   }

   // sort array in ascending order; pass function ascending as an
   // argument to specify ascending sorting order
   if (order == 1) {
      bubbleSort(a, SIZE, ascending);
      puts("\nData items in ascending order");
   }
   else { // pass function descending
      bubbleSort(a, SIZE, descending);
      puts("\nData items in descending order");
   }

   // output sorted array
   for (size_t counter = 0; counter < SIZE; ++counter) {
      printf("%5d", a[counter]);
   }

   puts("\n");
}

// multipurpose bubble sort; parameter compare is a pointer to
// the comparison function that determines sorting order
void bubbleSort(int work[], size_t size, int (*compare)(int a, int b)) {
   void swap(int *element1Ptr, int *element2ptr); // prototype

   // loop to control passes
   for (int pass = 1; pass < size; ++pass) {
      // loop to control number of comparisons per pass
      for (size_t count = 0; count < size - 1; ++count) {
         // if adjacent elements are out of order, swap them
         if ((*compare)(work[count], work[count + 1])) {
            swap(&work[count], &work[count + 1]);
         }
      }
   }
}

// swap values at memory locations to which element1Ptr and
// element2Ptr point
void swap(int *element1Ptr, int *element2Ptr) {
   int hold = *element1Ptr;
   *element1Ptr = *element2Ptr;
   *element2Ptr = hold;
}

// determine whether elements are out of order for an ascending order sort
int ascending(int a, int b) {
   return b < a; // should swap if b is less than a
}

// determine whether elements are out of order for a descending order sort
int descending(int a, int b) {
   return b > a; // should swap if b is greater than a
}
```

Function Pointers - **Sorting in Ascending or Descending Order**

```
Enter 1 to sort in ascending order,
Enter 2 to sort in descending order: 1

Data items in original order
    2    6    4    8   10   12   89   68   45   37
Data items in ascending order
    2    4    6    8   10   12   37   45   68   89
```

```
Enter 1 to sort in ascending order,
Enter 2 to sort in descending order: 2

Data items in original order
    2    6    4    8   10   12   89   68   45   37
Data items in descending order
   89   68   45   37   12   10    8    6    4    2
```

Fig. 7.17 | Multipurpose sorting program using function pointers. (Part 3 of 3.)

Function Pointers – **Using Function Pointers to Create a Menu-Driven System**

```c
// fig07_18.c
// Demonstrating an array of pointers to functions.
#include <stdio.h>

// prototypes
void function1(int a);
void function2(int b);
void function3(int c);

int main(void) {
   // initialize array of 3 pointers to functions that each take an
   // int argument and return void
   void (*f[3])(int) = {function1, function2, function3};

   printf("%s", "Enter a number between 0 and 2, 3 to end: ");
   int choice = 0;
   scanf("%d", &choice);

   // process user
   while (choice >= 0 && choice < 3) {
      // invoke function at location choice in array f and pass
      // choice as an argument
      (*f[choice])(choice);

      printf("%s", "Enter a number between 0 and 2, 3 to end: ");
      scanf("%d", &choice);
   }

   puts("Program execution completed.");
}

void function1(int a) {
   printf("You entered %d so function1 was called\n\n", a);
}

void function2(int b) {
   printf("You entered %d so function2 was called\n\n", b);
}

void function3(int c) {
   printf("You entered %d so function3 was called\n\n", c);
}
```

Alternative code:

```c
if (choice == 0) {
    function1(choice);
} else if (choice == 1) {
    function2(choice);
} else if (choice == 2) {
    function3(choice);
}
```

```
Enter a number between 0 and 2, 3 to end: 0
You entered 0 so function1 was called

Enter a number between 0 and 2, 3 to end: 1
You entered 1 so function2 was called

Enter a number between 0 and 2, 3 to end: 2
You entered 2 so function3 was called

Enter a number between 0 and 2, 3 to end: 3
Program execution completed.
```

Fig. 7.18 | Demonstrating an array of pointers to functions. (Part 2 of 2.)

In-class exercise

Build a calculator to perform one of the four operator:

Sum, Subtract, Multiplication, Division

Let the user define 2 numbers and the operator to perform.

Use function pointer, array of pointers. Do not use switch/case, if/else