UESTC 1005 — Introductory Programming

Lecture 8 — Pointers et al

Hasan T Abbas

Hasan.Abbas@glasgow.ac.uk

"You are sad," the Knight said in an anxious tone: "let me sing you a song to comfort you."

Lecture Outline

- Pointers and Arrays
- Pointers and Functions
- Pointers to Pointers

But First a Recap ..

- A **pointer** is a variable that stores the address of another variable.
- Efficient memory management
- Direct access to data
- Dynamic memory handling

```c
#include <stdio.h> // which has definitions of printf function

int main() // void means nothing
{
    int x = 10;
    int *ptr = &x;
    printf("%d\n", *ptr);  // Outputs 10
    return 0;
}
```

Pointers and Arrays

- Array name is a constant pointer to the first element
- Syntax equivalence:

  `arr[i] == *(arr + i);`

```
int arr[] = {78, 81, 88};
printf("%d\n", arr[1]);        // Outputs 20
printf("%d\n", *(arr + 1));    // Also outputs 20
```

Traversing Arrays with Pointers

```c
int arr[] = {78, 81, 88, 46, 28, 72, 96, 87, 42, 98};
int *ptr = arr;
for (int i = 0; i < 10; i++) {
    printf("%d ", *(ptr + i));
}
```

Modifying Arrays via Pointers

- Change array elements using pointers
- Recall the concept of passing by reference

```
int arr[] = {78, 81, 88, 46, 28, 72, 96, 87, 42, 98};
int *ptr = arr;
for (int i = 0; i < 10; i++) {
    printf("%d ", *(ptr + i));
}
*(ptr + 4) = 20;  // Modifies arr[1]
for (int i = 0; i < 10; i++) {
    printf("%d ", *(ptr + i));
}
```

Flipping an Array 🐬

- Lets reverse the elements of an array

```
void reverseArray(int *arr, int size) {
    int *start = arr, *end = arr + size - 1;
    while (start < end) {
        int temp = *start;
        *start = *end;
        *end = temp;
        start++;
        end--;
    }
}
```

Pointers and Functions

Passing Variables by Reference

- Pointers allow a function to modify a variable directly 🔑

```
void increment(int *val) {
    (*val)++;
}

int x = 5;
increment(&x);
printf("%d\n", x);  // Outputs 6
```

🚧 Pointer to a Function ❗

- Just like a variable, a function is also stored in the memory
- A **pointer to a function** stores the address of a function.
- Function pointers are useful for passing functions as parameters to other functions

```
return_type (*pointer_name)(parameter_list);
```

```
int add(int a, int b) {
    return a + b;
}
func_ptr = &add;  // or simply func_ptr = add;
```

```
int result = func_ptr(10, 20);  // Calls the add function
printf("Result: %d\n", result);
```

Example - Function Pointer

```c
#include <stdio.h>

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

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

void executeOperation(int (*operation)(int, int), int x, int y) {
    printf("Result: %d\n", operation(x, y));
}

int main() {
    executeOperation(add, 10, 5);         // Outputs: Result: 15
    executeOperation(subtract, 10, 5);  // Outputs: Result: 5
    return 0;
}
```

Why Function Pointers

- Callbacks in Event Handling
- i.e. Pass a function to be executed on an event trigger.
- Dynamic Function Selection:
- i.e. Switch between multiple behaviours at runtime (e.g., add vs. subtract).
- Sorting with Custom Comparators:
- Pass comparison functions to sorting algorithms.

Callbacks in Event Handling

- Imagine a doorbell 🔔. When someone presses the doorbell (event), you want to trigger an action (function) like turning on a light or playing a sound.
- Callbacks in event handling work similarly.
- When this event happens, execute a specific function.

```c
void onClick() {
    printf("Button clicked!\n");
}

void simulateButtonClick(void (*callback)()) {
    // Simulate a button click
    callback();  // Call the provided function
}

int main() {
    simulateButtonClick(onClick);  // Register the onClick callback
    return 0;
}
```

Dynamic Function Selection

- Now imagine a calculator 🧮 where you can add, subtract, multiply, or divide.
- The action you want depends on what the user selects at runtime.
- With a callback, you can pass the desired operation (add, subtract, etc.) as a function and execute it dynamically.

Function Pointer Example

- Write a program that uses a function pointer to perform addition, subtraction, multiplication, and division based on user input.
- Widely used in frameworks, APIs, and system-level programming.

Const Pointers in Functions

- Sometimes, we would to protect certain data
- We can use `const` to prevent modifications

```
void printArray(const int *arr, int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
}
```

Pointers to Pointers

Pointers to Pointers 🤔

- A pointer storing the address of another pointer
- Allows functions to modify variables directly.

```
 54   55   56   57   58   59   60   61   62   63   64   65   66   67   68   69
+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+
|    | 58 |    |    | 62 |    | 55 |    | C  | h  | e  | n  | g  | d |  u  | \0 |
+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+
```

```c
const char *pString = "Chengdu";
const char **ppString = &pString;
const char ***pppString = &ppString;
```

Pointers to Pointers

- Accessing Data through a pointer to a pointer

```
int x = 100;
int *p = &x;
int **pp = &p;

printf("%d\n", **pp);  // Outputs 100
```

Putting it all Together 🍱

- Dynamic Memory Allocation
- Allows the creation of arrays whose size is determined at runtime.
- Allocates memory only as needed, avoiding wastage.

```
array -> | row1 | row2 | row3 |
          /        |        \
         [10, 20] [30, 40] [50, 60]
```