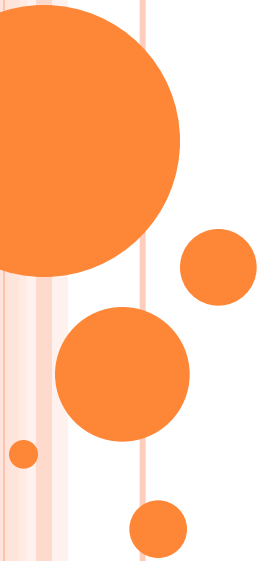
Chapter 7 Notes – **Pointers**

C How to Program, 6th Edition

Deitel & Deitel



SWE110 Lecture Note

Instructor: Zachi Baharav

(Slides credit: Thien An Nguyen (An))

Objectives

- Pointers
- Lab: we will continue to work on Assignment 5, lab2. However, the first part of assignment 5 should cover enough for you to work on the homework part.
- Require reading: Chapter 7
- Tutoring: if you need tutor, send Joey a request for private tutor session(s). It’s free.

7.1 – Introduction

- Pointer is one of the most powerful yet difficult to master features.
- It is an alternative for function parameters passing.
- However, it is the only way to access storage on the heap.

7.2 – Pointer Variable Definitions and Initialization

- Variables are memory locations where you can change their contents.
- Pointers are variables that contain the address of variables.

```
float *fptr;                    // declare a pointer to float variables
char *cptr;                     // a pointer to characters
int *iptr;                      // declare a pointer to int variables

int inum = 5;                   // an int variable
iptr = &inum;                   // iptr is a pointer to int variables

int *ptr2 = &inum;              // combine declare and initialize

char array[] = {'f', 'u', 'n'};    // an array
char *aptr = array;             // direct access to modify it
```

- In another words: a pointer is an address variable. It stores only the address

7.3 – Pointer Operators

- With a variable, you can directly access its value.

```
int inum = 5;                   // declare and initialize a variable
printf("%d", inum);             // direct access to fetch its content
inum++;                         // direct access to modify it
```

- With a pointer, you have to indirectly access the data of the variable that it is pointing to.

```
int inum = 5;                   // declare and initialize a variable
int *iptr = &inum;              // declare and initialize a pointer

printf("%d", *iptr);            // indirect access to fetch its content
*iptr = 10;                     // indirect access to modify it

int inum2 = 10;                 // declare and initialize a variable
iptr = &inum2;                  // change iptr
```

7.3 – Example

```
#include <stdio.h>

int main()
{
  int ivar;       // declare a variable of integers
  int *iptr;      // declare a pointer to int variables

  ivar = 10;      // init. the int variable with an int
  iptr = &ivar;   // init. the int pointer with address of an int

  printf("The address of ivar is %p\n", &ivar);
  printf("The value of iptr is %p\n\n", iptr);

  printf("The content of ivar is %d\n", ivar);
  printf("The content of the variable iptr pointing to is %d\n\n", *iptr);

  return 0;
}
```

7.3 – Another Example

```
#include <stdio.h>

int main()
{
  int ivar, ivar2, ivar3; // declare a variables of integers
  int *iptr;              // declare a pointer to int variables

  ivar = 10;              // init. the int variable with an int
  ivar2 = 11;             // init. the int variable with an int
  ivar3 = 12;             // init. the int variable with an int

  iptr = &ivar;  // init. the int pointer with address of an int
  printf("Current value of ivar is %d\n", *iptr);

  iptr = &ivar2; // point iptr to a different address of type int
  printf("Current value of ivar2 is %d\n", *iptr);

  iptr = &ivar3; // point iptr to a different address of type int
  printf("Current value of ivar3 is %d\n", *iptr);

  return 0;
}
```

7.4 – Parameter Passing

- Consider the following program:

```
#include <stdio.h>

// This function attempts to double the value of its input parameter
void doubleNum(int num)
{
  num = num * 2;
}

int main()
{
  int ivar;                  // declare an int variable
  ivar = 10;                 // init ivar

  doubleNum(ivar);           // call-by-value

  printf("The content of ivar is %d\n", ivar);

  return 0;
}
```

7.4 – Parameter Passing

- Consider the following program:

```
#include <stdio.h>

// This function attempts to double the value of its input parameter
void doubleNum(int num)
{
  num = num * 2;
}

int main()
{
  int ivar;                  // declare an int variable
  ivar = 10;                 // init ivar

  doubleNum(ivar);           // call-by-value

  printf("The content of ivar is %d\n", ivar);

  return 0;
}
```

Sample Output:

```
The content of ivar is: 10
```

Quick Check

- Was doubleNum attempt to double its input parameter success? Explain the result.

Quick Answer

- Was doubleNum attempt to double its input parameter success? Explain the result.

- No.
- In main, we call doubleNum and pass ivar to it.
- At doubleNum, ivar value is copied into a local variable name num.
- Inside doubleNum, it doubles num's content. It does not modify ivar.
- After doubleNum exit, num goes away, and ivar was not modified. So, it displays 10.

7.4 – Parameter Passing

- Consider the following changes:

```
#include <stdio.h>

// This function double the value of its input parameter
void doubleNum(int *num)
{
  *num = *num * 2;          // indirect access to ivar
}

int main()
{
  int ivar;                 // declare an int variable
  ivar = 10;                // init ivar

  doubleNum(&ivar);         // call-by-reference

  printf("The content of ivar is %d\n", ivar);

  return 0;
}
```

7.4 – Parameter Passing

- Consider the following changes:

```
#include <stdio.h>

// This function double the value of its input parameter
void doubleNum(int *ptr)
{
  *ptr = *ptr * 2;
}

int main()
{
  int ivar;                  // declare an int variable
  ivar = 10;                 // init ivar

  doubleNum(&ivar);          // call-by-reference

  printf("The content of ivar is %d\n", ivar);

  return 0;
}
```

Sample Output:

```
The content of ivar is: 20
```

Quick Check

- Was doubleNum doubles its input parameter success this time? Explain the result.

Quick Answer

- Was doubleNum doubles its input parameter success this time? Explain the result.

- Yes.
- In main, we call doubleNum with the address of ivar.
- At doubleNum, the address of ivar is copied into a local variable name ptr.
- Inside doubleNum, it indirectly doubles the content of ivar using `ptr`.
- After doubleNum exit, ptr goes away. ivar was modified. So, it displays 20.

7.4 – Parameter Passing

- Consider the following program:

```
#include <stdio.h>

// This function attempts to swap the value of two variables
void swap(int num1, int num2)
{
  int temp = num1;
  num1 = num2;
  num2 = temp;
}

int main()
{
  int ivar, ivar2;          // declare 2 int variables

  ivar = 10;                // init ivar
  ivar2 = 20;               // init ivar2

  swap(ivar, ivar2);        // call-by-value

  printf("The content of ivar is %d\n", ivar);
  printf("The content of ivar2 is %d\n", ivar2);

  return 0;
}
```

7.4 – Parameter Passing

- Consider the following program:

```
#include <stdio.h>

// This function attempts to swap the value of two variables
void swap(int num1, int num2)
{
  int temp = num1;
  num1 = num2;
  num2 = temp;
}

int main()
{
  int ivar, ivar2;          // declare 2 int variables

  ivar = 10;                // init ivar
  ivar2 = 20;               // init ivar2

  swap(ivar, ivar2);        // call-by-value

  printf("The content of ivar is %d\n", ivar);
  printf("The content of ivar2 is %d\n", ivar2);

  return 0;
}
```

Sample Output:

```
The content of ivar is: 10
The content of ivar2 is: 20
```

Quick Check

- Why did the value of ivar and ivar2 remained unchanged?

Quick Answer

- Why did the value of ivar and ivar2 remained unchanged?

- When `ivar` and `ivar2` are passed to the function swap, their values are copied into swap local variables `num1` and `num2`.
- By the end of swap, num1 = 20, num2 = 10. However, **num1 and num2 are swap's local variables**. `ivar` and `ivar2` were never been modified.

7.4 – Pointers as Input Parameters

```
#include <stdio.h>

// This function swap the value of its input parameters
void swap(int *ptr1, int *ptr2)
{
  int temp = *ptr1;
  *ptr1 = *ptr2;
  *ptr2 = temp;
}

int main()
{
  int ivar, ivar2;          // declare 2 int variables

  ivar = 10;                // init ivar
  ivar2 = 20;               // init ivar2

  swap(&ivar, &ivar2);      // call-by-reference

  printf("The content of ivar is %d\n", ivar);
  printf("The content of ivar2 is %d\n", ivar2);

  return 0;
}
```

7.4 – Pointers as Input Parameters

```
#include <stdio.h>

// This function swap the value of its input parameters
void swap(int *ptr1, int *ptr2)
{
  int temp = *ptr1;
  *ptr1 = *ptr2;
  *ptr2 = temp;
}

int main()
{
  int ivar, ivar2;          // declare 2 int variables

  ivar = 10;                // init ivar
  ivar2 = 20;               // init ivar2

  swap(&ivar, &ivar2);      // call-by-reference

  printf("The content of ivar is %d\n", ivar);
  printf("The content of ivar2 is %d\n", ivar2);

  return 0;
}
```

Sample Output:

```
The content of ivar is: 20
The content of ivar2 is: 10
```

Quick Check

- How did the changes in swap and the parameter passing effect `ivar` and `ivar2`?

Quick Answer

- How did the changes in swap and the parameter passing effect `ivar` and `ivar2`?
- Notice in main, when the swap is called, the address of `ivar` and `ivar2` are passed instead of the their contents.
- At the beginning of swap, it copies the address of `ivar` and `ivar2` into `ptr1` and `ptr2`.
- Since `ptr1` and `ptr2` are pointers, we have to indirectly access the memory locations they are pointing to… thus, `ivar` and `ivar2` are modified.
- After swap exit, `ptr1` and `ptr2` are no longer exist. However, `ivar` and `ivar2` retain their values.

7.4 – Parameter Passing Summary

- There are two ways to pass arguments to a function:
  - `Call-by-value`: a copy of the variables are passed into the calling functions. As the result, changes inside the function through the copies are not effected on the original copy of the variables.
  - `Call-by-reference`: the address of the variables are passed into the calling function. The function receives the references as pointer notations. Changes inside the function through the pointers are changes on the original variables.
- Arrays are passed into function via `call-by-reference`.

7.4 – Parameter Passing Summary

- In chapter 6, arrays as input parameters can be declared as one of the following:

```
void functionx(int arrayname[], int size)
{
  ...              // code in the function
}
```

```
void functionx(int arrayname[4], int size)
{
  ...              // code in the function
}
```

```
void functionx(int *arrayname, int size)
{
  ...              // code in the function
}
```

```
#define LEN 4
int main()
{
  int arr[] = {2, 1, 6, 5};
  functionx(arr, LEN);     // code in the function
  return 0;
}
```

Using const Qualifier with Pointers

- There are cases when you do not want the calling function to modify your variables. call-by-value would be the way to go.
- There are cases when you do want the calling function to modify your variables. call-by-reference would be the way to go.
- The `const` qualifier did not exist in early versions of C. It was added later. This introduces six possibilities as in the following slides.

Using const Qualifier with Pointers

- First, let's look into the combination of const and call-by-value:

```
// This function attempts to double the values of the array elements
void doubleElements(int arr[], int len)
{
  int i;
  len = 0;                          // accidentally modified

  for ( i = 0; i < len; i++){
    arr[i] = arr[i] * 2;
  }
}

int main()
{
  int i;
  int iarr[4] = {5, 3, 2, 1};       // declare and init. an array
  doubleElements(iarr, 4);          // call-by-reference, call-by-value
  for ( i = 0; i < 4; i++){         // print the array
    printf("%d ", iarr[i]);
  }
  return 0;
}
```

Using const Qualifier with Pointers

- The sample output does not double the array elements because the loop in the doubleElements failed.

```
// This function attempts to double the values of the array elements
void doubleElements(int arr[], int len)
{
  int i;
  len = 0;                          // accidentally modified

  for ( i = 0; i < len; i++){       // failed to execute the loop
    arr[i] = arr[i] * 2;
  }
}

int main()
{
  int i;
  int iarr[4] = {5, 3, 2, 1};       // declare and init. an array
  doubleElements(iarr, 4);          // call-by-reference, call-by-value
  for ( i = 0; i < 4; i++){         // print the array
    printf("%d ", iarr[i]);
  }
  return 0;
}
```

Sample Output:

```
5, 3, 2, 1
```

Using const Qualifier with Pointers

- Second, making `len` a `const` will prevent any accidental modifications to it.

```
// This function doubles the values of the array elements
void doubleElements(int arr[], const int len)
{
  int i;
  len = 0;                          // compile error. Take it out!

  for ( i = 0; i < len; i++){
    arr[i] = arr[i] * 2;
  }
}

int main()
{
  int i;
  int iarr[4] = {5, 3, 2, 1};       // declare and init. an array
  doubleElements(iarr, 4);          // call-by-reference, call-by-value
  for ( i = 0; i < 4; i++){         // print the array
    printf("%d ", iarr[i]);
  }
  return 0;
}
```

Using const Qualifier with Pointers

- Third, a `non-const` pointer to `non-const` data. The data can be modified:

```
// This function prints the values of the input array
void printArray(int *arr, const int len)
{
  int i;
  for ( i = 0; i < len; i++){
    printf("%d ", arr[i]);
    arr[i] = arr[i] + 2;          // un-wanted data modification
  }
}

int main()
{
  int i;
  int iarr[4] = {5, 3, 2, 1};      // declare and init. an array
  printArray(iarr, 4);    // call-by-reference, call-by-value

  return 0;
}
```

Using const Qualifier with Pointers

- Fourth, a `non-const` pointer to a `const` data. It prevents its data from being modified:

```
// This function prints the values of the input array
void printArray(const int *arr, const int len)
{
  int i;
  for ( i = 0; i < len; i++){
    printf("%d ", arr[i]);
    arr[i] = arr[i] + 2;            // compile error
  }
}

int main()
{
  int i;
  int iarr[4] = {5, 3, 2, 1};      // declare and init. an array
  printArray(iarr, 4);    // call-by-reference, call-by-value

  return 0;
}
```

Using const Qualifier with Pointers

- Again, `non-const` pointers to `non-const` data. The pointer can be modified:

```
// This function doubles the values of the array elements
void doubleElements(int *arr, const int len)
{
  int i;
  arr = &arr[1];                    // un-wanted modification, this result
  for ( i = 0; i < len; i++){       // in stack memory corruption
    arr[i] = arr[i] * 2;
  }
}

int main()
{
  int i;
  int iarr[4] = {5, 3, 2, 1};       // declare and init. an array
  doubleElements(iarr, 4);          // call-by-reference, call-by-value

  return 0;
}
```

Using const Qualifier with Pointers

- Fifth, a `const` pointers to `non-const` data. It prevent the pointer from being modified:

```
// This function doubles the values of the array elements
void doubleElements(int const *arr, const int len)
{
  int i;
  arr = &arr[1];                    // a compile error
  for ( i = 0; i < len; i++){
    arr[i] = arr[i] * 2;            // it's ok to modify the data
  }
}

int main()
{
  int i;
  int iarr[4] = {5, 3, 2, 1};       // declare and init. an array
  doubleElements(iarr, 4);          // call-by-reference, call-by-value

  return 0;
}
```

Using const Qualifier with Pointers

- Sixth, making `const` pointers to `const` data. The pointers and data cannot be modified:

```
// This function doubles the values of the array elements
void printArray(const int const *arr, const int len)
{
  int i;
  for ( i = 0; i < len; i++){
    printf("%d ", arr[i]);
    arr[i] = -1;                    // compile error
  }
}

int main()
{
  int i;
  int iarr[4] = {5, 3, 2, 1};      // declare and init. an array
  printArray(iarr, 4);    // call-by-reference, call-by-value

  return 0;
}
```

7.7 – sizeof Operator

- This operator returns the size in bytes of a data type or a data structure (i.e. an array).

```
int main()
{
  int ivar;
  char cvar;
  int iarr[4];

  printf("size of ivar: %d", sizeof(ivar));          // 4
  printf("size of cvar: %d", sizeof(cvar));          // 1
  printf("size of arr: %d", sizeof(arr));            // 14
  printf("size of float: %d", sizeof(float));        // 4

  return 0;
}
```

7.8 – Pointer Expressions and Arithmetic

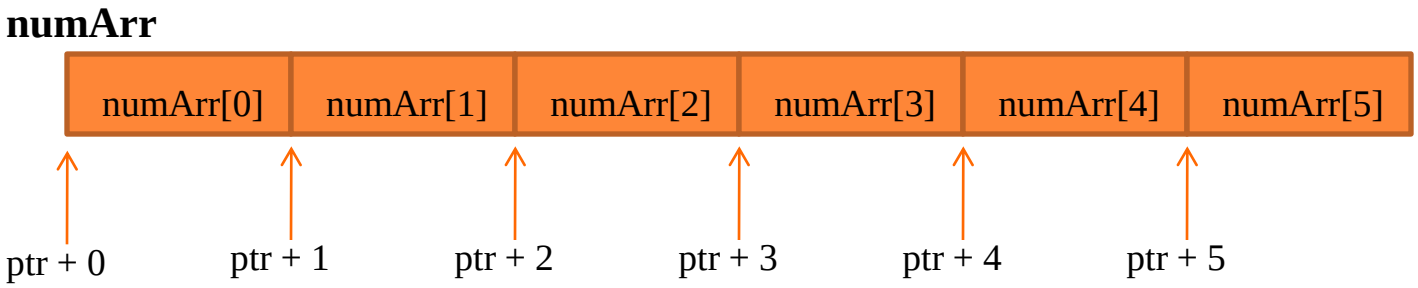
- Pointers do not have to point to a single value. They can point to different cells of an array.

```
int main()
{
  int numArr[6] = {3, 2, 4, 1, -1, 0}; // declare array numArr
  int *ptr = numArr;      // ptr points to the first element of the array
  int value;

  ptr = &numArr[4];       // points to the 5th element
  value = numArr[4];      // value = -1
  value = *ptr;           // value = -1

  ptr = numArr;           // points to the 1st element
  value = numArray[0];    // value = 3
  value = *ptr;           // value = 3

  ptr = &numArr[2];       // points to the 3rd element
  value = numArray[2];    // value = 4
  value = *ptr;           // value = 4

  return 0;
}
```

7.8 – Pointer Expressions and Arithmetic

- Pointer arithmetic were performed based on the size of the type of the array. For example, ptr is pointing to the 1st element of the int array. ptr++ would result in 4 bytes being added into ptr. Thus, moving it to point to the next element in the array.

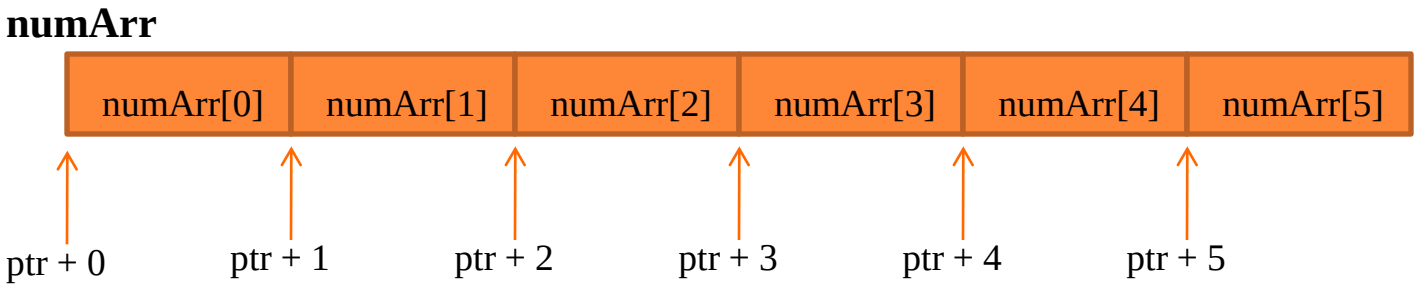
```
int numArr[6];
int *ptr = numArr;          // points to the 1st element of the array
ptr++;                      // points to element numArr[1]
```



7.8 – Pointer Expressions and Arithmetic

- In the following code fragment, ptr += 2 result in 8 bytes being added into the current location, which moves ptr to point to the third element in the array.
- The ptr += 4 would result in 16 bytes added into the current ptr which moves ptr to the end of the array.

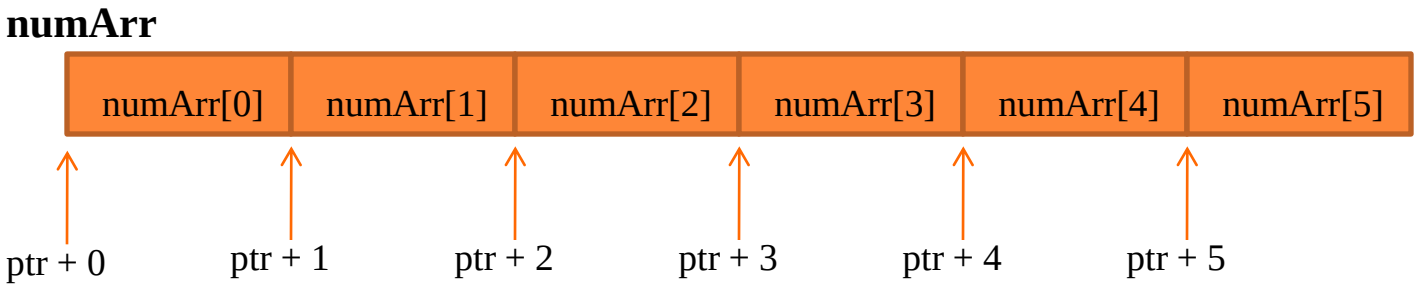
```
int numArr[6];
int *ptr = numArr;
ptr += 2;                  // points to element numArr[2]
ptr += 4;                  // points to the end of the array.
printf("%d", *ptr);        // unexpected, incorrect output
```



7.8 – Relationship between Array and Pointers

- The following pointer arithmetics are equivalent to accessing the array via the indexes:

```
int main()
{
  int numArr[6] = {3, 2, 4, 1, -1, 0}; // declare array numArr
  int *ptr = numArr;      // ptr points to the first element of the array

  if (numArr[1] == *(ptr + 1))
     printf("Hi!");                  // true. Hi!
  if (numArr[5] == *(ptr + 5))
      printf("Here 2");              // true. Here 2

  return 0;
}
```



7.10 – Arrays of Pointers

- The following declarations declare arrays of pointers to strings:

```
int main()
{
  char *arr[2] = {"Hello", "Hi"};  // array of pointers to char array

  printf("%s", arr[0]);            // Hello
  printf("%s", arr[1]);            // Hi

  return 0;
}
```

7.10 – Arrays of Pointers

- The following declarations are equivalent with the previous declaration:

```
int main()
{
  char *arr[2];                    // array of pointers to char array
  arr[0] = "Hello";
  arr[1] = "Hi";

  printf("%s", arr[0]);            // Hello
  printf("%s", arr[1]);            // Hi

  return 0;
}
```

In class

- Bubble sort example.