Lecture#2

C Pointers Introduction

Pointer Operators

Pass by Reference

CENG 102- Algorithms and Programming II,

2024-2025, Spring

Contains materials from:
P. Deitel, H. Deitel, ‘’C How to Program with an Introduction to C++’’, 8th edition, Pearson

Lecture 3.1

C Pointers Introduction

7.1 Introduction

- In this chapter, we discuss one of the most powerful features of the C programming language, the pointer.
- Pointers enable programs to simulate **pass-by-reference**, to pass functions between functions, and to create and manipulate **dynamic data structures**, i.e., data structures that can grow and shrink at execution time, such as linked lists, queues, stacks and trees.

7.2 Pointer Variable Definitions and Initialization

- Pointers are variables whose values are *memory addresses*.
- Normally, a variable directly contains a specific value.
- A pointer, on the other hand, contains an *address* of a variable that contains a specific value.
- In this sense, a variable name *directly* references a value, and a pointer *indirectly* references a value (Fig. 7.1).
- Referencing a value through a pointer is called indirection.

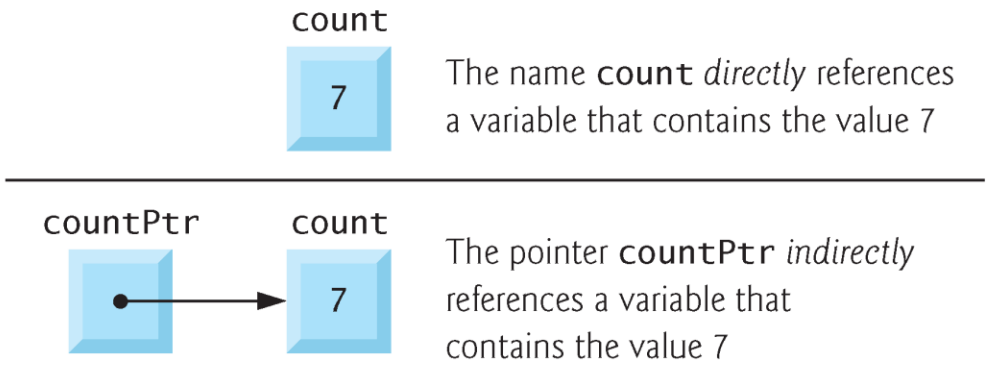

Fig. 7.1 | Directly and indirectly referencing a variable.

7.2 Pointer Variable Definitions and Initialization (cont.)

Declaring Pointers

- Pointers, like all variables, must be defined before they can be used.
- The definition

```
int *countPtr, count;
```

  specifies that variable `countPtr` is of type `int *` (i.e., a pointer to an integer).
- The definition is read (right to left),
  - "`countPtr` is a pointer to `int`"

    or
  - "`countPtr` points to an object of type `int`."
- Also, the variable `count` is defined to be an `int`, *not* a pointer to an `int`.

7.2 Pointer Variable Definitions and Initialization (cont.)

- The * applies *only* to `countPtr` in the definition.
- When * is used in this manner in a definition, it indicates that the variable **being defined is a pointer**.
- Pointers can be defined to point to objects of any type.
- To prevent the ambiguity of declaring pointer and non-pointer variables in the same declaration as shown above, you should always declare only one variable per declaration.

Common Programming Error 7.1

The asterisk (*) notation used to declare pointer variables does not *distribute to all variable names in a declaration. Each pointer must be declared with the * prefixed to the name; e.g., if you wish to declare* `xPtr` *and* `yPtr` *as* `int` *pointers, use* `int *xPtr, *yPtr;`.

Good Programming Practice 7.1

We prefer to include the letters `Ptr` *in pointer variable names to make it clear that these variables are pointers and need to be handled appropriately.*

7.2 Pointer Variable Definitions and Initialization (cont.)

Initializing and Assigning Values to Pointers

- Pointers can be initialized when they're defined, or they can be assigned a value.
- A pointer may be initialized to `NULL`, `0` or an address.
- A pointer with the value `NULL` or `0` points to *nothing*.
- `NULL` is a *symbolic constant* defined in the `<stddef.h>` header (and several other headers, such as `<stdio.h>`).

7.2 Pointer Variable Definitions and Initialization (cont.)

- Initializing a pointer to `0` is equivalent to initializing a pointer to `NULL`, but `NULL` is preferred.
- The value `0` is the *only* integer value that can be assigned directly to a pointer variable.
- When `0` is assigned, it's first converted to a pointer of the appropriate type.

Error-Prevention Tip 7.1

Initialize pointers to prevent unexpected results.

Lecture 3.2

C Pointer Operators

7.3 Pointer Operators

- The &, or address operator, is a unary operator that returns the address of its operand.
- For example, assuming the definitions
  - ```
    int y = 5;
    int *yPtr;
    ```

  the statement
  - `yPtr = &y;`

  assigns the *address* of the variable `y` to pointer variable `yPtr`.
- Variable `yPtr` is then said to "point to" `y`.
- Figure 7.2 shows a schematic representation of memory after the preceding assignment is executed.
    ```

7.3 Pointer Operators (Cont.)

Pointer Representation in Memory

- Figure 7.3 shows the representation of the pointer in memory, assuming that integer variable `y` is stored at location `600000`, and pointer variable `yPtr` is stored at location `500000`.
- The operand of the address operator must be a variable; the address operator *cannot* be applied to constants or expressions.

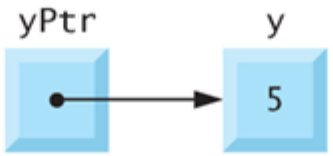

Fig. 7.2 | Graphical representation of a pointer pointing to an integer variable in memory.

Fig. 7.3 | Representation of `y` and `yPtr` in memory.

7.3 Pointer Operators (Cont.)

The Indirection (*) Operator

- The unary * operator, commonly referred to as the indirection operator or dereferencing operator, returns the *value* of the object to which its operand (i.e., a pointer) points.
- For example, the statement
  - `printf("%d", *yPtr);`

  prints the value of variable y, namely 5.
- Using * in this manner is called dereferencing a pointer.

Common Programming Error 7.2

Dereferencing a pointer that has not been properly initialized or that has not been assigned to point to a specific location in memory is an error. This could cause a fatal execution-time error, or it could accidentally modify important data and allow the program to run to completion with incorrect results.

7.3 Pointer Operators (Cont.)

Demonstrating the & and * Operators

- Figure 7.4 demonstrates the pointer operators `&` and *.
- The **`printf`** conversion specifier `%p` outputs the memory location as a *hexadecimal* integer on most platforms.
- Notice that the *address* of `a` and the *value* of `aPtr` are identical in the output, thus confirming that the address of `a` is indeed assigned to the pointer variable `aPtr`
- The & and * operators are complements of one another—when they're both applied consecutively to `aPtr` in either order, the same result is printed.
- Figure 7.5 lists the precedence and associativity of the operators introduced to this point.

```c
// Fig. 7.4: fig07_04.c
// Using the & and * pointer operators.
#include <stdio.h>

int main(void)
{
   int a = 7;
   int *aPtr = &a; // set aPtr to the address of a

   printf("The address of a is %p"
          "\nThe value of aPtr is %p", &a, aPtr);

   printf("\n\nThe value of a is %d"
          "\nThe value of *aPtr is %d", a, *aPtr);

   printf("\n\nShowing that * and & are complements of "
          "each other\n&*aPtr = %p"
          "\n*&aPtr = %p\n", &*aPtr, *&aPtr);
}
```

Fig. 7.4 | Using the & and * pointer operators. (Part 1 of 2.)

```
The address of a is 0028FEC0
The value of aPtr is 0028FEC0

The value of a is 7
The value of *aPtr is 7

Showing that * and & are complements of each other
&*aPtr = 0028FEC0
*&aPtr = 0028FEC0
```

Fig. 7.4 | Using the & and * pointer operators. (Part 2 of 2.)

Operators	Associativity	Type
`()` `[]` `++` *(postfix)* `--` *(postfix)*	left to right	postfix
`+` `-` `++` `--` `!` `*` `&` *(type)*	right to left	unary
`*` `/` `%`	left to right	multiplicative
`+` `-`	left to right	additive
`<` `<=` `>` `s>=`	left to right	relational
`==` `!=`	left to right	equality
`&&`	left to right	logical AND
`||`	left to right	logical OR
`?:`	right to left	conditional
`=` `+=` `-=` `*=` `/=` `%=`	right to left	assignment
`,`	left to right	comma

Fig. 7.5 | Precedence and associativity of the operators discussed so far.

Lecture 3.3

Pass by Reference

7.4 Passing Arguments to Functions by Reference

- There are two ways to pass arguments to a function—pass-by-value and pass-by-reference.
- Many functions require the capability to *modify variables in the caller* or to pass a pointer to a large data object to avoid the overhead of passing the object by value (which incurs the time and memory overheads of making a copy of the object).
- In C, you use **pointers** and the **indirection operator to *simulate* pass-by-reference (actually it passes the address by value, but you can use that address as a reference to modify the content of the variable)**.

7.4 Passing Arguments to Functions by Reference (Cont.)

- **When calling a function with arguments that should be modified, the *addresses* of the arguments are passed.**
- This is normally accomplished by applying the address operator (&) to the variable (in the caller) whose value will be modified.
- As we saw in Chapter 6, **arrays are *not* passed using operator & because C automatically passes the starting location in memory of the array (the name of an array is equivalent to `&arrayName[0]`).**
- When the address of a variable is passed to a function, the indirection operator (*) may be used in the function to modify the value at that location in the caller's memory.

7.4 Passing Arguments to Functions by Reference (Cont.)

Pass-By-Value

- The programs in Fig. 7.6 and Fig. 7.7 present two versions of a function that cubes an integer—`cubeByValue` and `cubeByReference`.
- Figure 7.6 passes the variable `number` by value to function `cubeByValue`
- The `cubeByValue` function cubes its argument and passes the new value back to `main` using a `return` statement.
- The new value is assigned to `number` in `main`

```c
// Fig. 7.6: fig07_06.c
// Cube a variable using pass-by-value.
#include <stdio.h>

int cubeByValue(int n); // prototype

int main(void)
{
   int number = 5; // initialize number

   printf("The original value of number is %d", number);

   // pass number by value to cubeByValue
   number = cubeByValue(number);

   printf("\nThe new value of number is %d\n", number);
}

// calculate and return cube of integer argument
int cubeByValue(int n)
{
   return n * n * n; // cube local variable n and return result
}
```

Fig. 7.6 | Cube a variable using pass-by-value. (Part 1 of 2.)

```
The original value of number is 5
The new value of number is 125
```

Fig. 7.6 | Cube a variable using pass-by-value. (Part 2 of 2.)

7.4 Passing Arguments to Functions by Reference (Cont.)

Pass-By-Reference

- Figure 7.7 passes the variable `number` by reference—the address of `number` is passed—to function `cubeByReference`.
- Function `cubeByReference` takes as a parameter a pointer to an `int` called `nPtr`.
- The function *dereferences* the pointer and cubes the value to which `nPtr` points, then assigns the result to `*nPtr` (which is really `number` in `main`), thus changing the value of `number` in `main`.

```c
// Fig. 7.7: fig07_07.c
// Cube a variable using pass-by-reference with a pointer argument.

#include <stdio.h>

void cubeByReference(int *nPtr); // function prototype

int main(void)
{
   int number = 5; // initialize number

   printf("The original value of number is %d", number);

   // pass address of number to cubeByReference
   cubeByReference(&number);

   printf("\nThe new value of number is %d\n", number);
}

// calculate cube of *nPtr; actually modifies number in main
void cubeByReference(int *nPtr)
{
   *nPtr = *nPtr * *nPtr * *nPtr; // cube *nPtr
}
```

Fig. 7.7 | Cube a variable using pass-by-reference with a pointer argument. (Part 1 of 2.)

```
The original value of number is 5
The new value of number is 125
```

Fig. 7.7 | Cube a variable using pass-by-reference with a pointer argument. (Part 2 of 2.)

7.4 Passing Arguments to Functions by Reference (Cont.)

- A function receiving an *address* as an argument must define a *pointer parameter* to receive the address.
- For example, in Fig. 7.7 the header for function `cubeByReference` is:
  - `void cubeByReference(int *nPtr)`
- The header specifies that `cubeByReference` *receives* the *address* of an integer variable as an argument, stores the address locally in `nPtr` and does not return a value.
- The function prototype for `cubeByReference` contains `int *` in parentheses. (Names included in prototype for documentation purposes are ignored by the C compiler.)

Error-Prevention Tip 7.2

Use pass-by-value to pass arguments to a function unless the caller explicitly requires the called function to modify the value of the argument variable in the caller's environment. This prevents accidental modification of the caller's arguments and is another example of the principle of least privilege.

7.4 Passing Arguments to Functions by Reference (Cont.)

- For a function that expects a one-dimensional array as an argument, the function's prototype and header can use the pointer notation shown in the parameter list of function `cubeByReference`.
- The compiler does not differentiate between a function that receives a pointer and one that receives a one-dimensional array.
- When the compiler encounters a function parameter for a one-dimensional array of the form `int b[]`, the compiler converts the parameter to the pointer notation `int *b`.
- The two forms are interchangeable.

7.4 Passing Arguments to Functions by Reference (Cont.)

```c
#include <stdio.h>

void printArray1(int arr[], int size);
void printArray2(int *arr, int size);

int main() {
    int numbers[] = {1, 2, 3, 4, 5};

    printArray1(numbers, 5);
    printArray2(numbers, 5);
}

void printArray1(int arr[], int size)
{
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

void printArray2(int *arr, int size)
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}
```

```
1 2 3 4 5
1 2 3 4 5
```

What is the output?

```c
#include <stdio.h>

int main()
{
    int *ab;
    int m;
    m=17;

    printf("The address of m : %p\n", &m);
    printf("The value of m : %d\n\n", m);

    ab=&m;

    printf("The address of ab : %p\n", ab);
    printf("The value of ab : %d\n\n", *ab);

    m=35;

    printf("The address of ab : %p\n", ab);
    printf("The value of ab : %d\n\n", *ab);

    *ab=8;

    printf("The address of m : %p\n", &m);
    printf("The value of m : %d", m);
}
```

The Output

```
The address of m : 0x7fff22a7181c
The value of m : 17

The address of ab : 0x7fff22a7181c
The value of ab : 17

The address of ab : 0x7fff22a7181c
The value of ab : 35

The address of m : 0x7fff22a7181c
The value of m : 8
```

Example 1

A C program that sums two integer values using pointers.

```c
#include <stdio.h>

int main()
{
    int number1 = 5, number2 = 7, *ptr1, *ptr2, sum;

    ptr1 = &number1;
    ptr2 = &number2;

    sum = *ptr1 + *ptr2;

    printf("Sum: %d\n", sum);
}
```

Example 2

A C program that sums two integer values using call by reference.

```c
#include <stdio.h>

int sumTwoValues(int *, int *);

int main()
{
    int number1 = 5, number2 = 7, sum;

    sum = sumTwoValues(&number1, &number2);

    printf("Sum: %d\n", sum);
}

int sumTwoValues(int *n1, int *n2)
{
    int sum = *n1 + *n2;
    return sum;
}
```

Self-Practice Exercise

Write a C program code that finds the maximum of two numbers using pointers.