Chapter 7: Pointers
===================

* Pointers are variables that contain memory addresses as their values.
* A variable name directly references a value
* A pointer indirectly references a value.

```
               countPtr               count
                +-----+              +-----+
                |213f-+------------->|  7  |
                +-----+              +-----+
                  0176                 213f
```

* The declaration:

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

* The "*" only applies to "countPtr" in the declaration, it indicates that the variable being declared is a pointer.
* Pointers can be declared to point to objects of any data type (including user-defined).

* A pointer may be initialized to "0", "NULL", or an address.
* A pointer with the value "NULL" points to nothing.
* The value "0" is the only integer value that can be assigned directly to a pointer variable.

Pointer Operators
----

* The "&", or address operator, is a unary operator that returns the address of its operand.
* E.g.

```
        int y = 5;
        int *yPtr;
        ....
        yPtr = &y;
        ....
```

* This statement assigns the adress of the variable "y" to pointer variable "yPtr".
* The operand of the address operator must be a variable; it cannot be constants, expressions, or variables declared with the storage class "register".

* The "*" 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.
* E.g.

```
        printf("%d", *yPtr);
```

Calling Functions by Address
----

* All function calls in C are call by value.
* Many functions require the capability to
    1. modify one or more variables in the caller
    2. pass a pointer to a large data object to avoid the overhead of passing the object call by value (which requires making a copy of the object).
* In C, when calling a function with arguments that should be modified, the addresses of the arguments are passed.
* 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.

* E.g.

```
        #include <stdio.h>
        int cubeByValue(int);
        main()
        {
            int number = 5;

            printf("The original value of number is %d\n", number);
            number = cubeByValue(number);
            printf("The new value of number is %d\n", number);
            return 0;
        }

        int cubeByValue(int n)
        {
            return n * n * n;
        }
```

* E.g.

```
        #include <stdio.h>
        int cubeByReference(int *);
        main()
        {
            int number = 5;

            printf("The original value of number is %d\n", number);
            cubeByReference(&number);
            printf("The new value of number is %d\n", number);
            return 0;
        }

        int cubeByReference(int *nPtr)
        {
            *nPtr = (*nPtr) * (*nPtr) * (*nPtr);
        }
```

* The compiler does not differentiate between a function that receives a pointer and a function that receives a single-subscripted array.

* That is, the following two forms are (almost) the same,

```
        int cubeByReference(int *nPtr)
        {
            .....
        }
```

And

```
        int cubeByReference(int nPtr[])
        {
            .....
        }
```

Using the "Const" Qualifier with Pointers
===

* The "const" qualifier enables the programmer to inform the compiler that the value of a particular variable should not be modified.

* Always award a function enough access to the data in its parameters to accomplish its specified task, but no more.

* An array name is a constant pointer to the beginning of the array.

* All data in the array can be accessed and changed by using the array name and array index.

* E.g.

```
        char string[] = "characters";
        char character = 'C';
        ....
        string = &character;        /* error */
        ....
```

* Pointers that are declared "const" must be initialized when they are declared.
* A constant pointer:

```
        char string[] = "characters";
```

* A non-constant pointer to a contant data:

```
        const char *string;
```

* A constant pointer to non-constant data:

```
        int * const ptr = &x;
```

* A constant pointer to a constant data:

```
        const int *const ptr = &x;
```

"sizeof" operator

* C provides the special unary operator "sizeof" to determine the size in bytes of an array (or any other data type) during program compilation.
* E.g.

```
        #include <stdio.h>
        main()
        {
            float array[20];
            printf("The number of bytes in the array is %d\n",
                        sizeof(array));
            return 0;
        }
```

* To determine the number of elements in the array,

```
        arraysize = sizeof(array) / sizeof(double);
```

* Operator "sizeof" can be applied to any variable name, type, or constant.
* When applied to a variable name (that is not an array name) or a constant, the number of bytes used to store the specific type of variable or constant is found.
* E.g.

```
        #include <stdio.h>
        main()
        {
             printf("        sizeof(char) = %d\n"
                    "       sizeof(short) = %d\n"
                    "         sizeof(int) = %d\n"
                    "        sizeof(long) = %d\n"
                    "       sizeof(float) = %d\n"
                    "      sizeof(double) = %d\n"
                    " sizeof(long double) = %d\n",
                    sizeof(char), sizeof(short), sizeof(int),
                    sizeof(long), sizeof(float), sizeof(double),
                    sizeof(long double));
             return 0;
        }
```

Pointer Expressions and Pointer Arithmetic

* Pointers can be valid operands in arithmetic expressions, assignment expressions, and comparison expressions.

* A pointer may be
    1. incremented (++)
    2. decremented (--)
    3. an integer adds to a pointer (+ or +=)
    4. an integer substracts from a pointer (- or -=)
    5. one pointer substracts from another

* When an integer is added to (or substracted from) a pointer, the pointer is not simply incremented (or decremented) by the integer, but by that integer times the size of the object to which the pointer refers.
* Therefore,

```
        a = v[20];
```

and

```
        a = *(v+20);
```

is the same, no matter what the type of array "v" is.
* Pointer arithmetic is meaningless unless performed on an array.

* A pointer can be assigned to another pointer if both pointers are of the same type.
* Otherwise, a cast operator must be used to convert the pointer on the right of the assignment to the pointer type on the left of the assignment.
* The exeception to this rule is the pointer to "void" (i.e., "void *") which is a generic pointer that can represent any pointer type.
* A pointer to "void" cannot be dereferenced.

* Pointers can be compared using equality and relational operators.
* This can be meaningless unless the pointers point to members of the same array.

The Relationship between Pointers and Arrays
----

* An array name can be thought of as a constant pointer.
* Pointers can be used to do any operation involving array indexing.
* E.g.

```
        int b[5], *bPtr;
        ....
        bPtr = b;      /* or  bPtr = &b[0];     */
        ....
        b[3] = 10;     /* or  *(bPtr + 3) = 10; */
                       /* or  *(b + 3) = 10;    */
        ....
        bPtr[1] = 2;   /* or  b[1] = 2;         */
                       /* or *(bPtr + 1) = 2;   */
        ....
        bPtr = &b[3];  /* or  bPtr = bPtr + 3;  */
        ....
```

* Question: Then "bPtr[1]" = ??
* The "3" in the above expression is the offset to the pointer.
* When the pointer points to the beginning of an array, the offset value is identical to the array index.
* E.g.

```
        #include <stdio.h>
        main()
        {
            int i, offset, b[]={10,20,30,40};
            int *bPtr = b;
            printf("Array b printed with:\n"
                   "Array indexing notation\n");
            for (i=0; i<4; i++)
                printf("b[%d] = %d\n", i, b[i]);

            printf("\nPointer/offset notation where \n"
                   "the pointer is the array name\n");
            for (offset=0; offset<4; offset++)
                printf("*(b + %d) = %d\n", offset, *(b+offset));

            printf("\nPointer index notation\n");
            for (i=0; i<4; i++)
                printf("bPtr[%d] = %d\n", i, bPtr[i]);

            printf("\nPointer/offset notation\n");
            for (offset=0; offset<4; offset++)
                printf("*(bPtr + %d) = %d\n", offset,
                                        *(bPtr+offset));
            return 0;
        }
```

* E.g.

```
        #include <stdio.h>
        void copy1(char *, const char *);
        void copy2(char *, const char *);
        main()
        {
            char string1[10], *string2 = "Hello";
                 string3[10], string4[] = "Good Bye";

            copy1(string1, string2);
            printf("string1 = %s\n", string1);

            copy2(string3, string4);
            printf("string3 = %s\n", string3);
            return 0;
        }

        void copy1(char *s1, const char *s2)
        {
            int i;
            for (i=0; s1[i] = s2[i]; i++);
        }

        void copy2(char *s1, const char *s2)
        {
            for (; *s1 = *s2; s1++, s2++);
        }
```

Arrays of Pointers
----

* Arrays may contain pointers.
* A common use of such a data structure is to form an array of strings.
* Each entry in an array of strings is actually a pointer to the first character of a string.
* E.g.

```
        char *suit[4] = {"Hearts", "Diamonds", "Clubs", "Spades"};
```

* Each of these strings is stored in memory as a NULL-terminated character string.

* Although it appears as though these strings are being placed in the "suit" array, only pointers are actually stored in the array.
* Thus, even though the "suit" array is fixed in size, it provides access to character strings of any length.
* E.g.

```
        #include <stdio.h>
        #include <stdlib.h>
        #include <time.h>

        void shuffle(int [][13]);
        void deal(const int [][13], const char *[], const char *[]);

        main()
        {
            char *suit[4] = {"Hearts", "Diamonds", "Clubs",
                             "Spades"};
            char *face[13] = {"Ace", "Deuce", "Three", "Four",
                              "Five", "Six", "Seven", "Eight",
                              "Nine", "Ten", "Jack", "Queen",
                              "King"};
            int deck[4][13] = {0};

            srand(time(NULL));
            shuffle(deck);
            deal(deck, face, suit);
            return 0;
        }

        void shuffle(int wDeck[][13])
        {
            int card, row, column;

            for (card=1; card<=52; card++)
            {
                row = rand() % 4;
                column = rand() % 13;
                while (wDeck[row][column]!=0)
                {
                    row = rand() % 4;
                    column = rand() % 13;
                }
                wDeck[row][column] = card;
            }
        }

        void deal(const int wDeck[][13], const char *wFace[],
                  const char *wSuit[])
        {
            int card, row, column;
            for (card=1; card<=52; card++)
                for (row=0; row<4; row++)
                    for (column=0; column<13; column++)
                        if (wDeck[row][column] == card)
                            printf("%5s of %-8s%c",
                                    wFace[column], wSuit[row],
                                    (card%2)==0?'\n':'\t');
        }
```

Pointer to Functions

----

* A pointer to a function contains the address of the function in memory.
* A function name is the starting address in memory of the code that performs the function's task.
* Pointers to functions can be
    1. passed to functions
    2. returned from functions
    3. stored in arrays
    4. assigned to other function pointers
* E.g.

```
        void bubble(int *work, const int size,
                            int (*compare)(int, int))
        {
            ......
            if ((*compare)(work[count], work[count+1]))
            ......

        }
```

* The parameter:

```
        int (*compare)(int, int)
```

tells "bubble" to expect a parameter that is a pointer to a function that receives two integer parameters and returns an integer result.