C Programming for Engineers

Pointers

ICEN 360– Spring 2017

Prof. Dola Saha

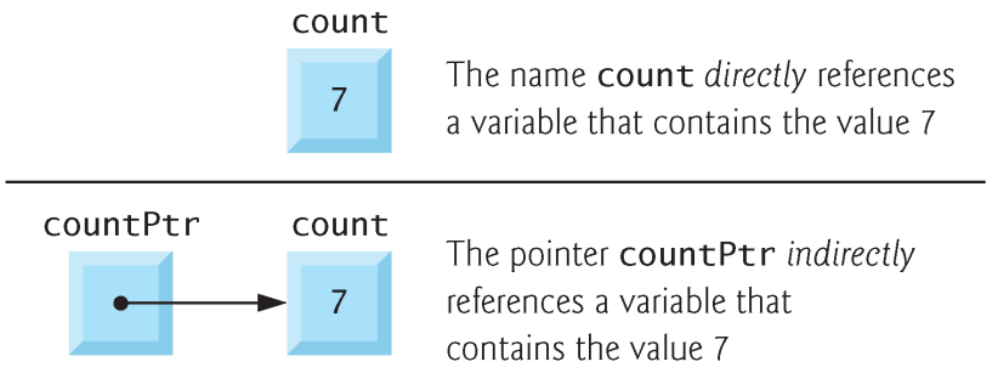
Pointers

- Pointers are variables whose values are *memory addresses*.
- A variable name *directly* references a value, and a pointer *indirectly* references a value.
- Referencing a value through a pointer is called indirection.

Declaring Pointers

- Pointers 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 variable `count` is defined to be an `int`, *not* a pointer to an `int`.



Initializing Pointers

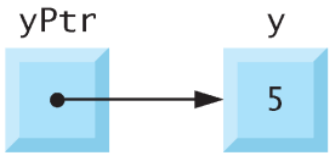
- Pointers should be initialized when they're defined or they can be assigned a value.
- A pointer may be initialized to `NULL`, `0` or an address.
- A pointer with the value `NULL` points to *nothing*.
- `NULL` is a *symbolic constant* defined in the <stddef.h> header (and several other headers, such as <stdio.h>).
- Initializing a pointer to `0` is equivalent to initializing a pointer to `NULL`, but `NULL` is preferred.
- When `0` is assigned, it's first converted to a pointer of the appropriate type.
- The value `0` is the *only* integer value that can be assigned directly to a pointer variable.

Pointer Operator

- The &, or address operator, is a unary operator that returns the address of its operand.
- Example definition
  - ```
    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.



Graphical Representation

Memory Representation
    ```

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.
- Example:
  - `printf("%d", *yPtr);`

  prints the value of variable that `yPtr` is pointing to

  In this case it is `y`, whose value is 5.
- Using * in this manner is called dereferencing a pointer.

Using & and *

```
3   #include <stdio.h>
4
5   int main(void)
6   {
7      int a = 7;
8      int *aPtr = &a; // set aPtr to the address of a
9
10     printf("The address of a is %p"
11            "\nThe value of aPtr is %p", &a, aPtr);
12
13     printf("\n\nThe value of a is %d"
14            "\nThe value of *aPtr is %d", a, *aPtr);
15
16     printf("\n\nShowing that * and & are complements of "
17            "each other\n&*aPtr = %p"
18            "\n*&aPtr = %p\n", &*aPtr, *&aPtr);
19  }
```

```
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
```

Pass by value

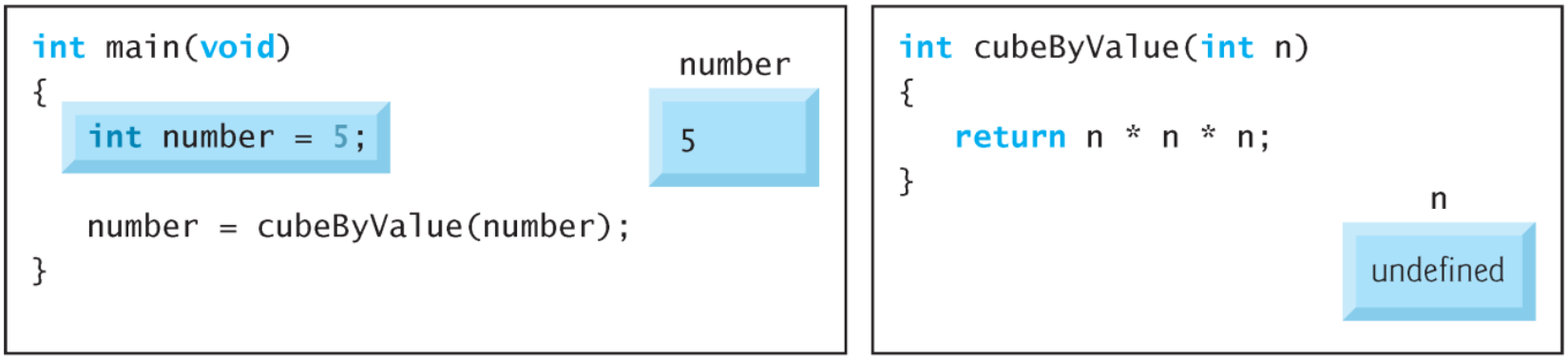
```
1  // Fig. 7.6: fig07_06.c
2  // Cube a variable using pass-by-value.
3  #include <stdio.h>
4
5  int cubeByValue(int n); // prototype
6
7  int main(void)
8  {
9     int number = 5; // initialize number
10
11    printf("The original value of number is %d", number);
12
13    // pass number by value to cubeByValue
14    number = cubeByValue(number);
15
16    printf("\nThe new value of number is %d\n", number);
17 }
18
19 // calculate and return cube of integer argument
20 int cubeByValue(int n)
21 {
22    return n * n * n; // cube local variable n and return result
23 }
```

Pass by reference – simulating with Pointer

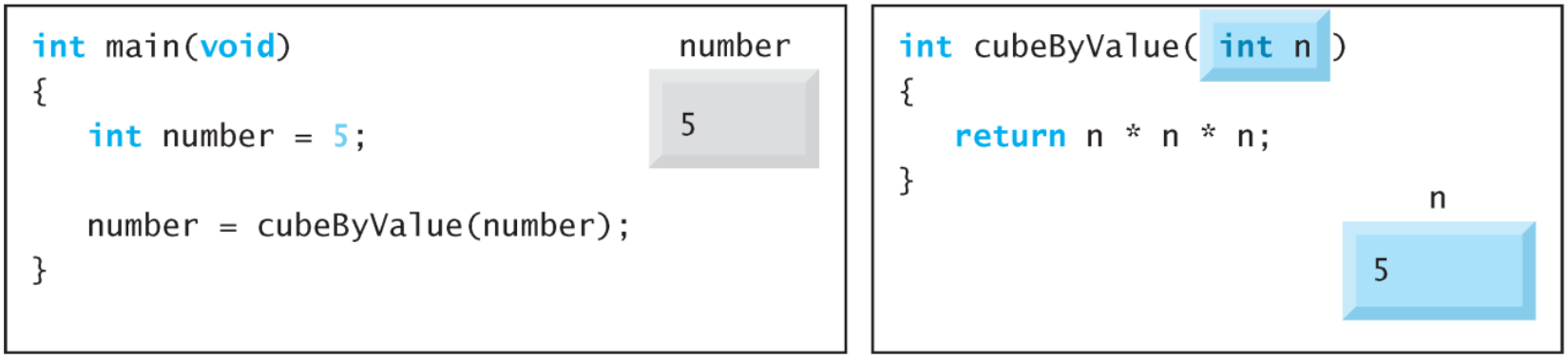
```
1  // Fig. 7.7: fig07_07.c
2  // Cube a variable using pass-by-reference with a pointer argument.
3
4  #include <stdio.h>
5
6  void cubeByReference(int *nPtr); // function prototype
7
8  int main(void)
9  {
10    int number = 5; // initialize number
11
12    printf("The original value of number is %d", number);
13
14    // pass address of number to cubeByReference
15    cubeByReference(&number);
16
17    printf("\nThe new value of number is %d\n", number);
18 }
19
20 // calculate cube of *nPtr; actually modifies number in main
21 void cubeByReference(int *nPtr)
22 {
23    *nPtr = *nPtr * *nPtr * *nPtr; // cube *nPtr
24 }
```

Pass by value (1)

Step 1: Before `main` calls `cubeByValue`:

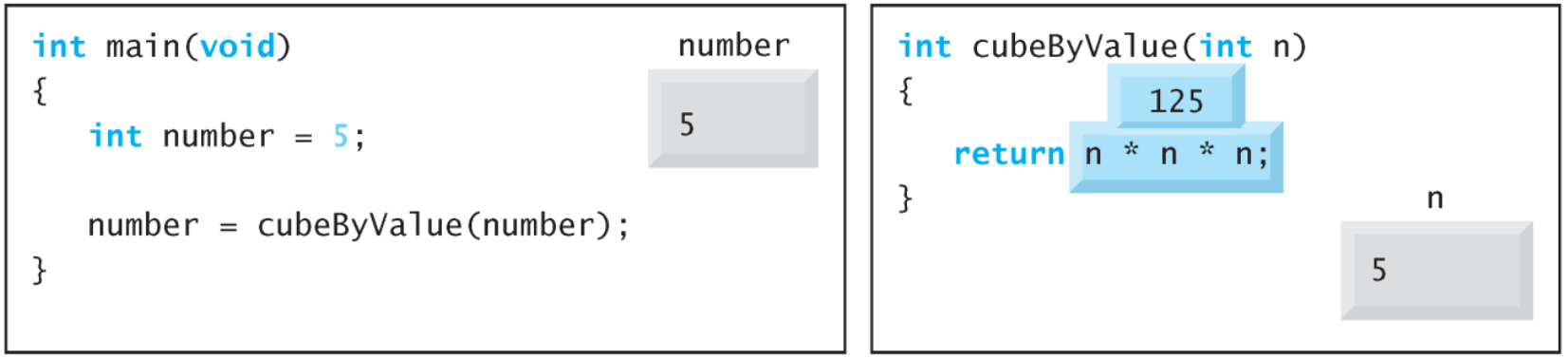


Step 2: After `cubeByValue` receives the call:

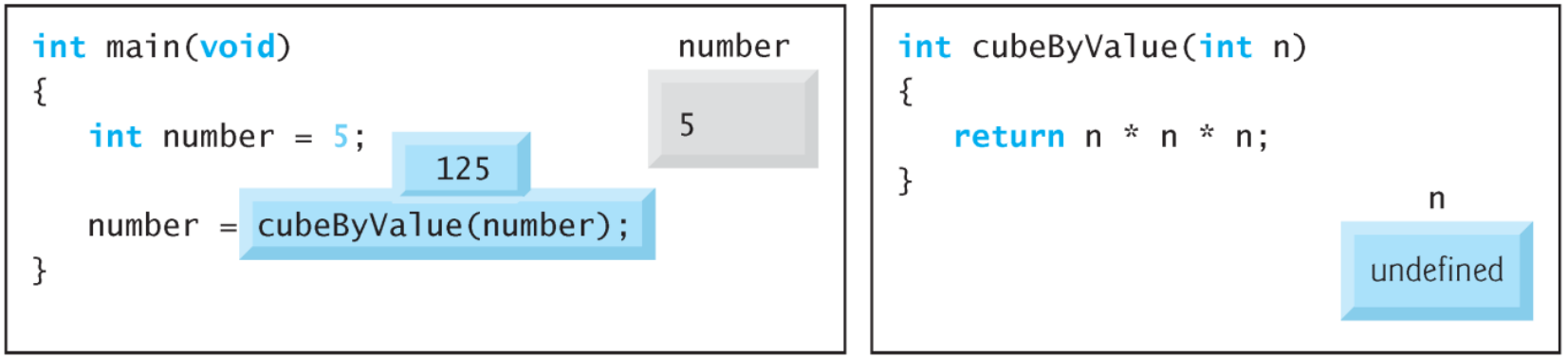


Pass by value (2)

Step 3: After `cubeByValue` cubes parameter `n` and before `cubeByValue` returns to `main`:

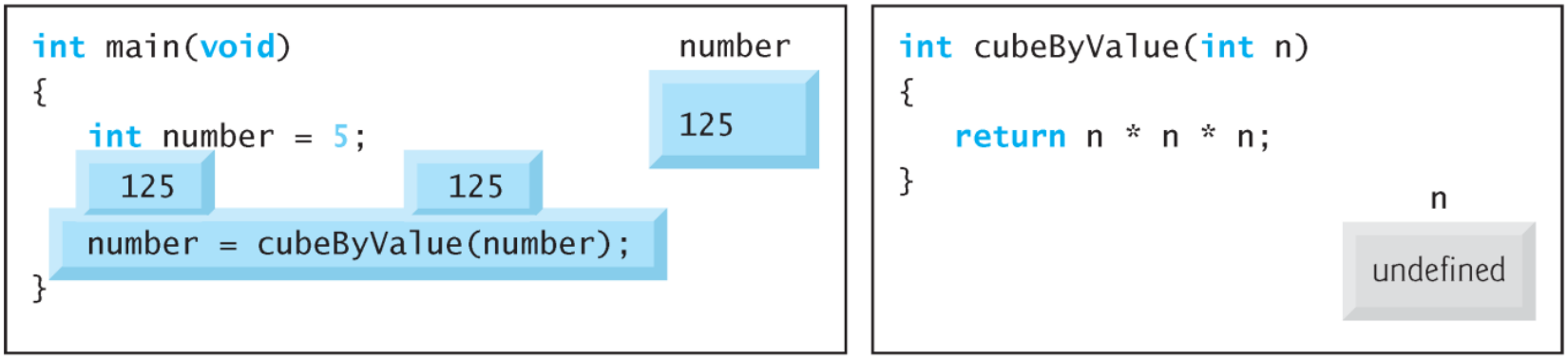


Step 4: After `cubeByValue` returns to `main` and before assigning the result to `number`:



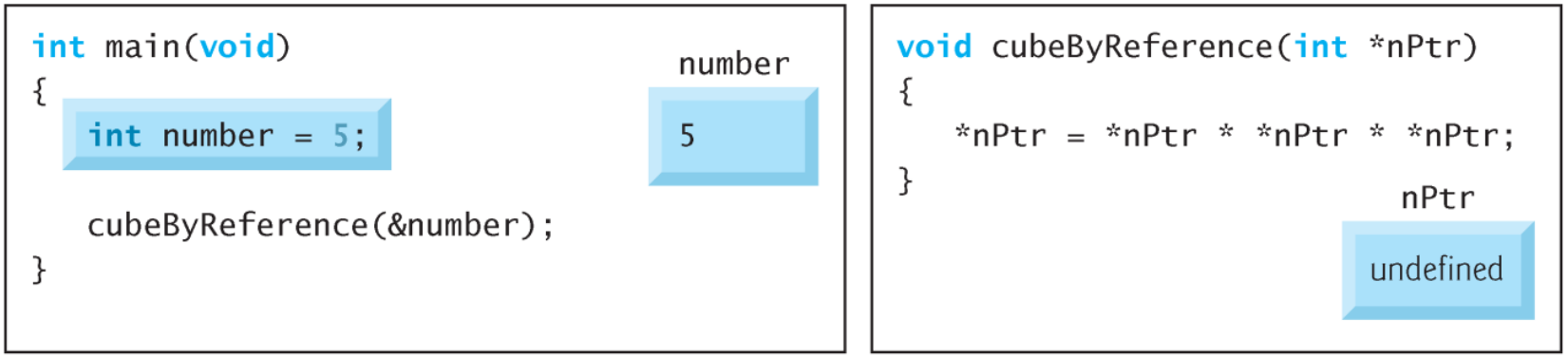
Pass by value (3)

Step 5: After `main` completes the assignment to `number`:

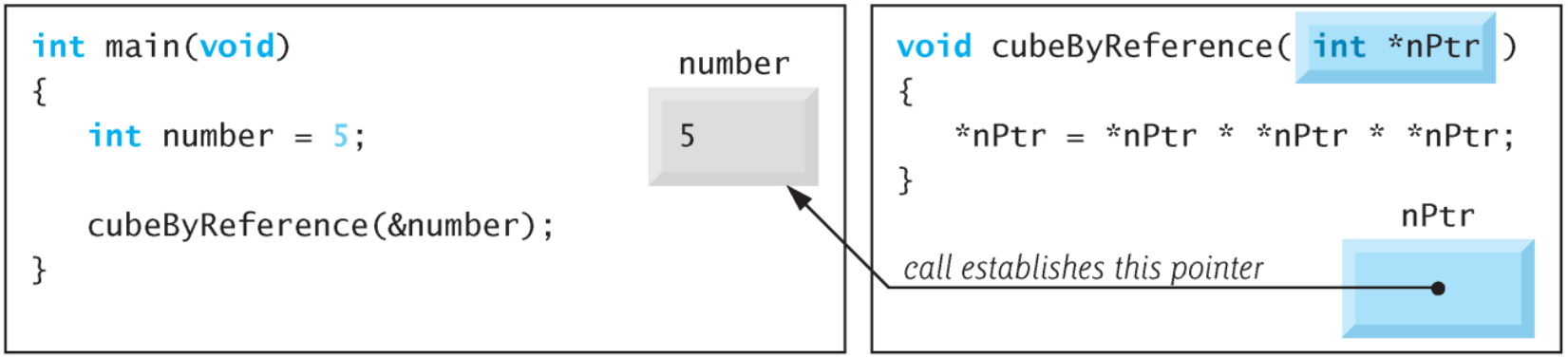


Pass by reference (1)

Step 1: Before `main` calls `cubeByReference`:

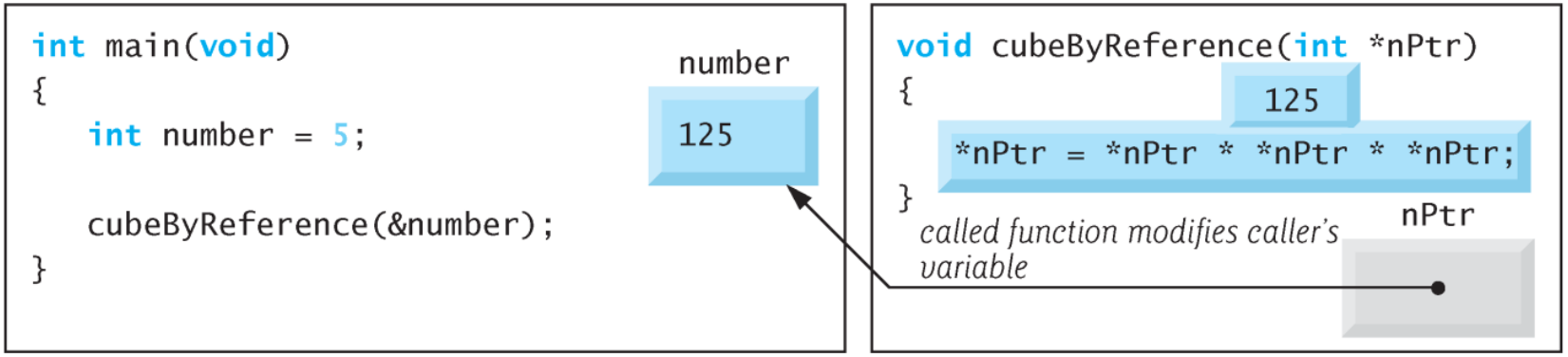


Step 2: After `cubeByReference` receives the call and before `*nPtr` is cubed:



Pass by reference (2)

Step 3: After `*nPtr` is cubed and before program control returns to `main`:



Determine Size of Data Types (1)

```
1  // Fig. 7.17: fig07_17.c
2  // Using operator sizeof to determine standard data type sizes.
3  #include <stdio.h>
4
5  int main(void)
6  {
7     char c;
8     short s;
9     int i;
10    long l;
11    long long ll;
12    float f;
13    double d;
14    long double ld;
15    int array[20]; // create array of 20 int elements
16    int *ptr = array; // create pointer to array
17
18    printf("    sizeof c = %u\tsizeof(char)  = %u"
19           "\n     sizeof s = %u\tsizeof(short) = %u"
20           "\n     sizeof i = %u\tsizeof(int) = %u"
21           "\n     sizeof l = %u\tsizeof(long) = %u"
22           "\n    sizeof ll = %u\tsizeof(long long) = %u"
23           "\n     sizeof f = %u\tsizeof(float) = %u"
```

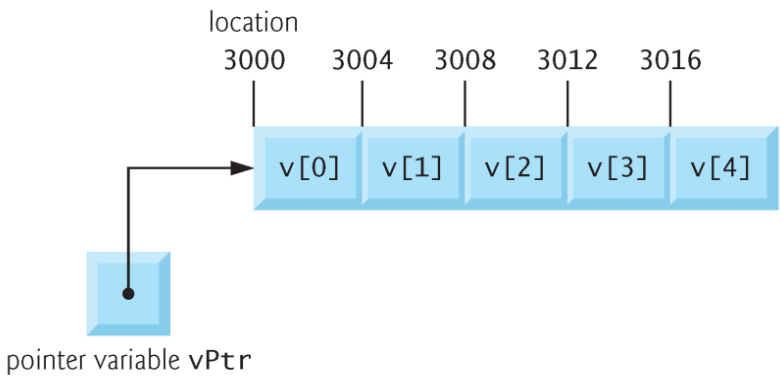
Determine Size of Data Types (2)

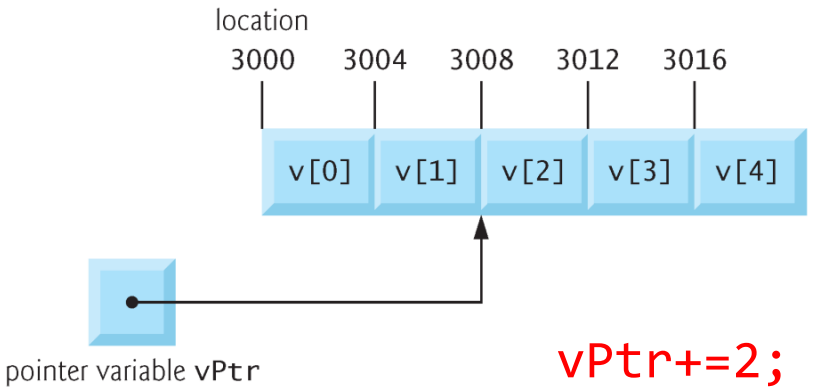
```
24              "\n     sizeof d = %u\tsizeof(double) = %u"
25              "\n    sizeof ld = %u\tsizeof(long double) = %u"
26              "\n sizeof array = %u"
27              "\n   sizeof ptr = %u\n",
28              sizeof c, sizeof(char), sizeof s, sizeof(short), sizeof i,
29              sizeof(int), sizeof l, sizeof(long), sizeof ll,
30              sizeof(long long), sizeof f, sizeof(float), sizeof d,
31              sizeof(double), sizeof ld, sizeof(long double),
32              sizeof array, sizeof ptr);
33  }
```

```
     sizeof c = 1       sizeof(char)  = 1
     sizeof s = 2       sizeof(short) = 2
     sizeof i = 4       sizeof(int) = 4
     sizeof l = 4       sizeof(long) = 4
    sizeof ll = 8       sizeof(long long) = 8
     sizeof f = 4       sizeof(float) = 4
     sizeof d = 8       sizeof(double) = 8
    sizeof ld = 8       sizeof(long double) = 8
sizeof array = 80
  sizeof ptr = 4
```

Pointer Arithmetic

- A pointer may be
  - *incremented* (++) or *decremented* (--),
  - an integer may be *added* to a pointer (+ or +=),
  - an integer may be *subtracted* from a pointer (- or -=)
  - one pointer may be subtracted from another—this last operation is meaningful only when *both* pointers point to elements of the *same* array.
- When an integer n is added to or subtracted from a pointer
  - Pointer is incremented or decremented by that integer times the size of the object to which the pointer refers.





Pointer and Array

- Arrays and pointers are intimately related in C and often may be used interchangeably.
- An *array name* can be thought of as a constant pointer.
- Pointers can be used to do any operation involving array indexing.
- Set `bPtr` equal to the address of the first element in array `b` with the statement
  - `bPtr = b;`
- Address of the array's first element:
  - `bPtr = &b[0];`

Pointer and Array

- Array element `b[3]` with pointer expression
  - `*(bPtr + 3)`
  - The `3` in the expression is the offset to the pointer.
- This notation is referred to as pointer/offset notation.
- Address of `b[3]` can be referenced as
  - `&b[3]`
  - `(bPtr+3)`

Access array elements by pointer (1)

```
1  // Fig. 7.10: fig07_10.c
2  // Converting a string to uppercase using a
3  // non-constant pointer to non-constant data.
4  #include <stdio.h>
5  #include <ctype.h>
6
7  void convertToUppercase(char *sPtr); // prototype
8
9  int main(void)
10 {
11    char string[] = "cHaRaCters and $32.98"; // initialize char array
12
13    printf("The string before conversion is: %s", string);
14    convertToUppercase(string);
15    printf("\nThe string after conversion is: %s\n", string);
16 }
17
```

Access array elements by pointer (2)

```
18  // convert string to uppercase letters
19  void convertToUppercase(char *sPtr)
20  {
21     while (*sPtr != '\0') { // current character is not '\0'
22        *sPtr = toupper(*sPtr); // convert to uppercase
23        ++sPtr; // make sPtr point to the next character
24     }
25  }
```

```
The string before conversion is: cHaRaCters and $32.98
The string after conversion is: CHARACTERS AND $32.98
```

Classroom Assignment

- Write a function encrypt() and a function decrypt() to change each of the characters in a string shift by a given value.

- Sample output where the values are shifted by 5.

```
String given = This is confidential information
String encrypted = Ymnx%nx%htsknijsynfq%nsktwrfynts
String decrypted = This is confidential information
```

Pointer Notation with Arrays (1)

```
1  // Fig. 7.20: fig07_20.cpp
2  // Using indexing and pointer notations with arrays.
3  #include <stdio.h>
4  #define ARRAY_SIZE 4
5
6  int main(void)
7  {
8     int b[] = {10, 20, 30, 40}; // create and initialize array b
9     int *bPtr = b; // create bPtr and point it to array b
10
11    // output array b using array index notation
12    puts("Array b printed with:\nArray index notation");
13
14    // loop through array b
15    for (size_t i = 0; i < ARRAY_SIZE; ++i) {
16       printf("b[%u] = %d\n", i, b[i]);
17    }
18
19    // output array b using array name and pointer/offset notation
20    puts("\nPointer/offset notation where\n"
21          "the pointer is the array name");
22
```

Pointer Notation with Arrays (2)

```
23      // loop through array b
24      for (size_t offset = 0; offset < ARRAY_SIZE; ++offset) {
25         printf("*(b + %u) = %d\n", offset, *(b + offset));
26      }
27
28      // output array b using bPtr and array index notation
29      puts("\nPointer index notation");
30
31      // loop through array b
32      for (size_t i = 0; i < ARRAY_SIZE; ++i) {
33         printf("bPtr[%u] = %d\n", i, bPtr[i]);
34      }
35
36      // output array b using bPtr and pointer/offset notation
37      puts("\nPointer/offset notation");
38
39      // loop through array b
40      for (size_t offset = 0; offset < ARRAY_SIZE; ++offset) {
41         printf("*(bPtr + %u) = %d\n", offset, *(bPtr + offset));
42      }
43   }
```

Pointer Notation with Arrays (3)

```
Array b printed with:
Array index notation
b[0] = 10
b[1] = 20
b[2] = 30
b[3] = 40

Pointer/offset notation where
the pointer is the array name
*(b + 0) = 10
*(b + 1) = 20
*(b + 2) = 30
*(b + 3) = 40

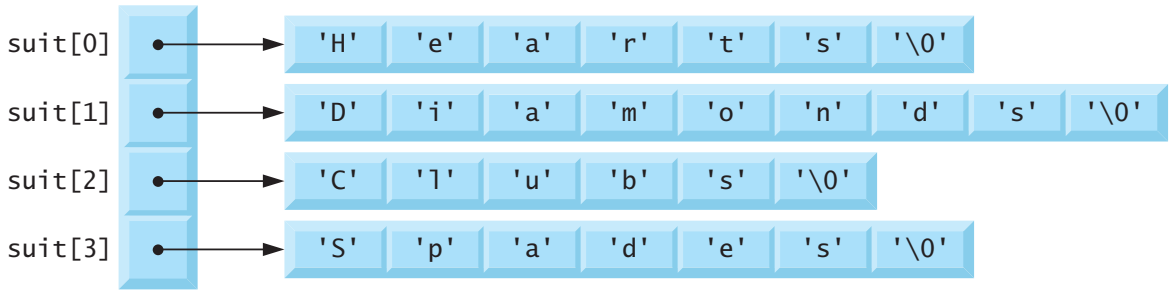
Pointer index notation
bPtr[0] = 10
bPtr[1] = 20
bPtr[2] = 30
bPtr[3] = 40

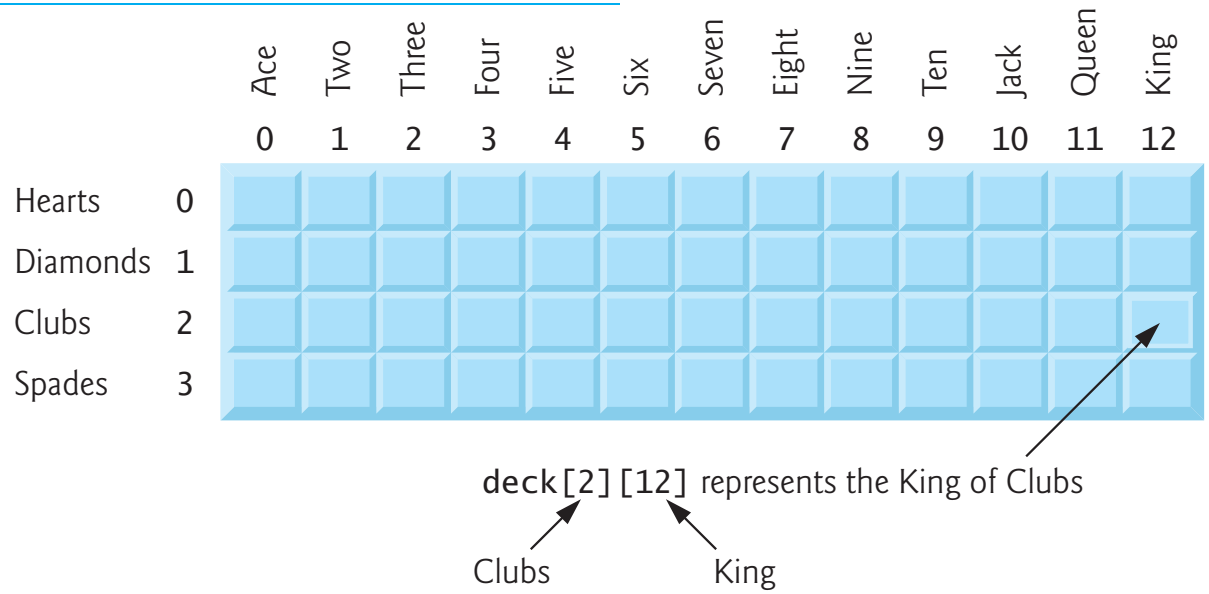
Pointer/offset notation
*(bPtr + 0) = 10
*(bPtr + 1) = 20
*(bPtr + 2) = 30
*(bPtr + 3) = 40
```

Array of Pointers

- Arrays may contain pointers

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





Pointers to Functions

- A pointer to a function contains address of function in the memory.

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

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

```
// invoke function at location choice in array f and pass
// choice as an argument
(*f[ choice ])( choice );
```

Stack - Push and Pop with Pointers

```
1.  void
2.  push(char stack[],     /* input/output - the stack */
3.       char item,        /* input - data being pushed onto the stack */
4.       int  *top,        /* input/output - pointer to top of stack */
5.       int  max_size)    /* input - maximum size of stack */
6.  {
7.       if (*top < max_size-1) {
8.           ++(*top);
9.           stack[*top] = item;
10.      }
11. }
12.
13. char
14. pop(char stack[],     /* input/output - the stack */
15.     int *top)         /* input/output - pointer to top of stack */
16. {
17.      char item;       /* value popped off the stack */
18.
19.      if (*top >= 0) {
20.           item = stack[*top];
21.           --(*top);
22.      } else {
23.           item = STACK_EMPTY;
24.      }
25.
26.      return (item);
27. }
```

Calculate Execution Time

```
#include <time.h>
clock_t start, end;
start = clock();
// Write the code that needs to be timed
end = clock();
double time_taken = ((double)(end-start)) /
                            CLOCKS_PER_SEC;
printf("The time taken for this program is %lf\n",
                            time_taken);
```