Programming for Engineers

UNIVERSITY AT ALBANY
State University of New York

Pointers

ICEN 200 – Spring 2018

Prof. Dola Saha

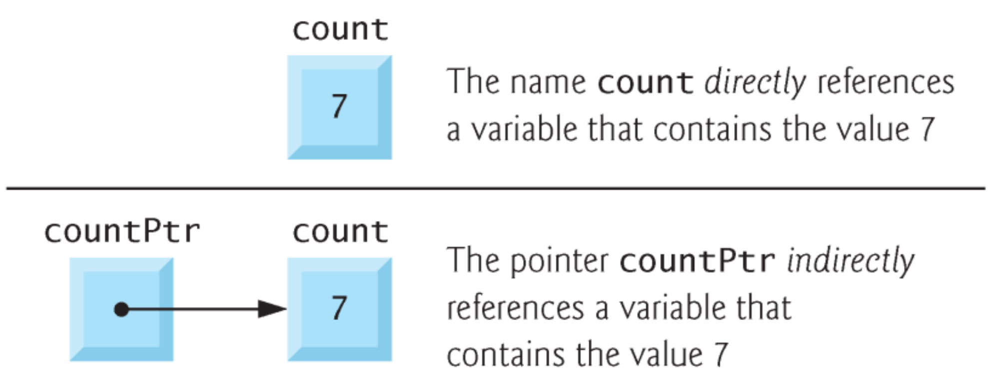
Pointers

- 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

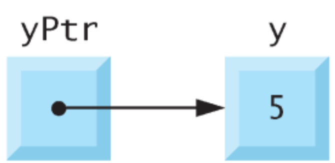
- Pointers should be initialized when they're defined or they can be assigned a value.
- A pointer may be initialized to `NULL`, `0` or an address.
- A pointer with the value `NULL` points to *nothing*.
- `NULL` is a *symbolic constant* defined in the <stddef.h> header (and several other headers, such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

```c
#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);
}
```

```
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

```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
}
```

Pass by reference – simulating with Pointer

```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
}
```

Pass by value (1)

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

```
int main(void)
{
    int number = 5;

    number = cubeByValue(number);
}
```

number: 5

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

n: undefined

Step 2: After `cubeByValue` receives the call:

```
int main(void)
{
    int number = 5;

    number = cubeByValue(number);
}
```

number: 5

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

n: 5

Pass by value (2)

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

```
int main(void)
{
    int number = 5;

    number = cubeByValue(number);
}
```

number: 5

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

`n * n * n` → 125

n: 5

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

```
int main(void)
{
    int number = 5;

    number = cubeByValue(number);
}
```

`cubeByValue(number)` → 125

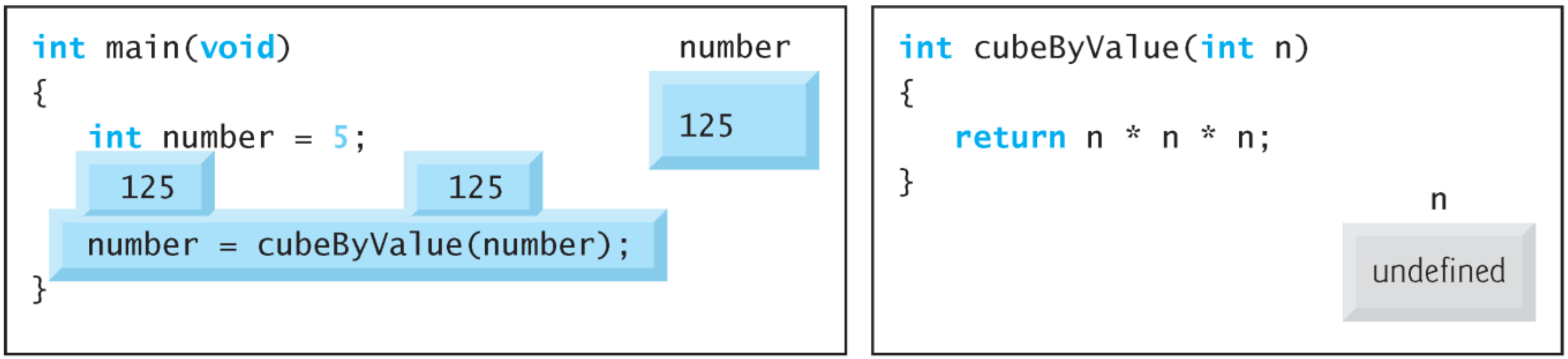
number: 5

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

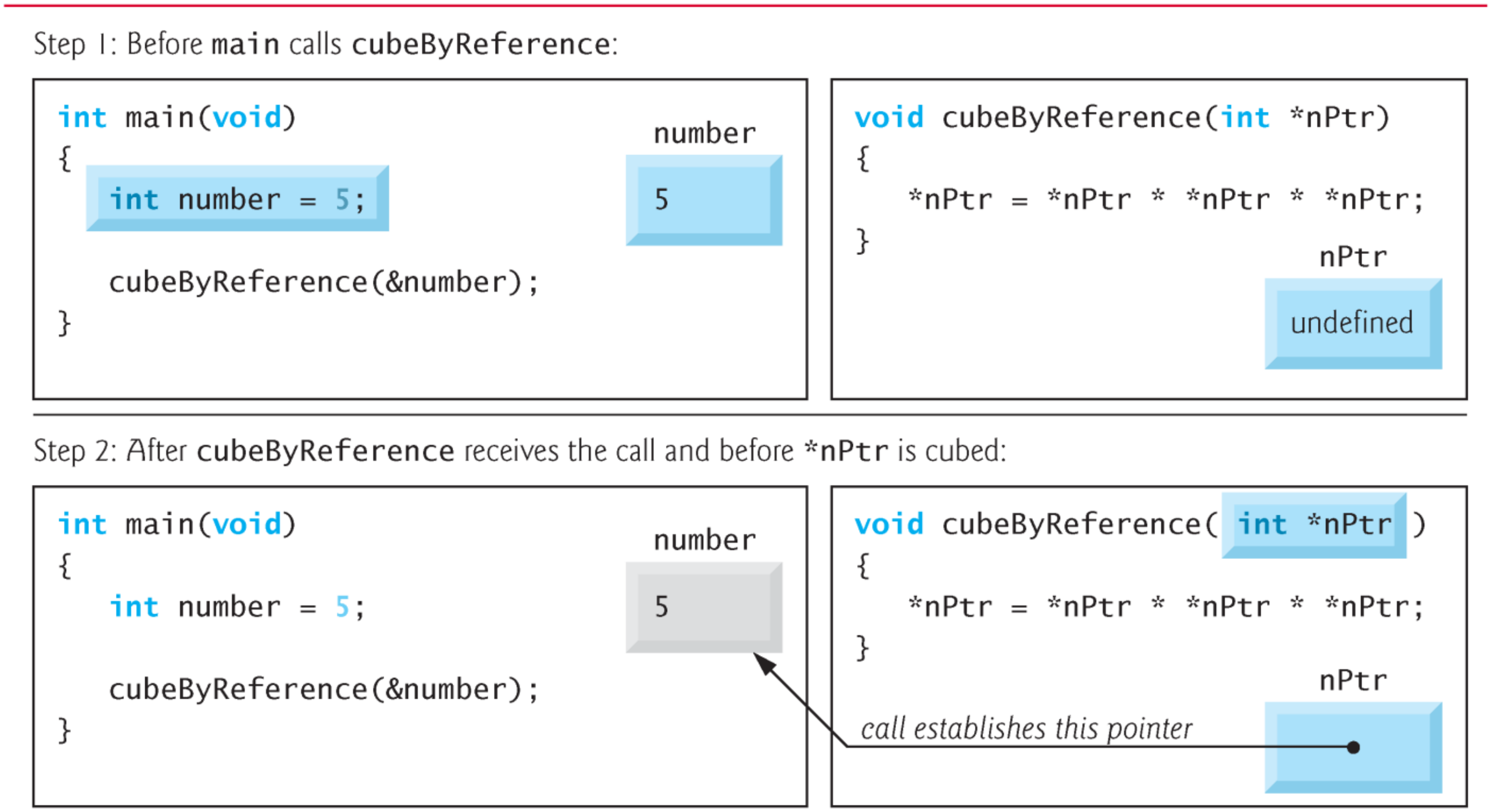
n: undefined

Pass by value (3)

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

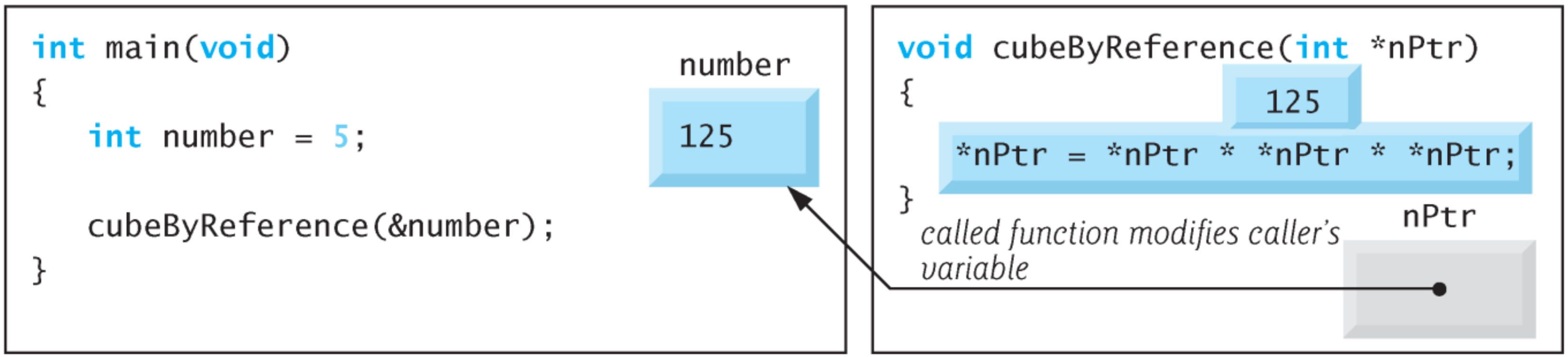


Pass by reference (1)



Pass by reference (2)

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



Determine Size of Data Types (1)

```
// Fig. 7.17: fig07_17.c
// Using operator sizeof to determine standard data type sizes.
#include <stdio.h>

int main(void)
{
   char c;
   short s;
   int i;
   long l;
   long long ll;
   float f;
   double d;
   long double ld;
   int array[20]; // create array of 20 int elements
   int *ptr = array; // create pointer to array

   printf("     sizeof c = %u\tsizeof(char)  = %u"
          "\n     sizeof s = %u\tsizeof(short) = %u"
          "\n     sizeof i = %u\tsizeof(int) = %u"
          "\n     sizeof l = %u\tsizeof(long) = %u"
          "\n    sizeof ll = %u\tsizeof(long long) = %u"
          "\n     sizeof f = %u\tsizeof(float) = %u"
```

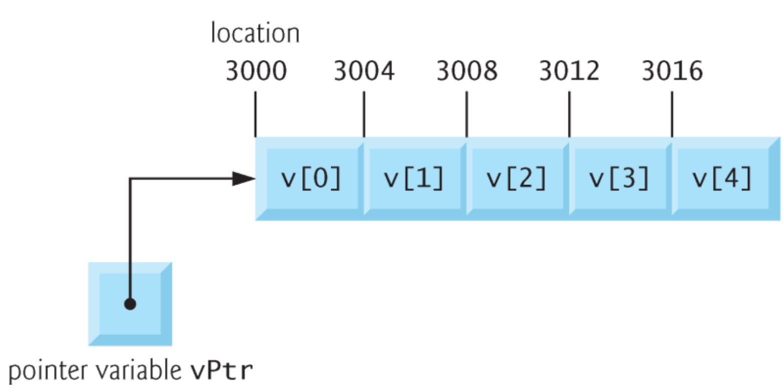
Determine Size of Data Types (2)

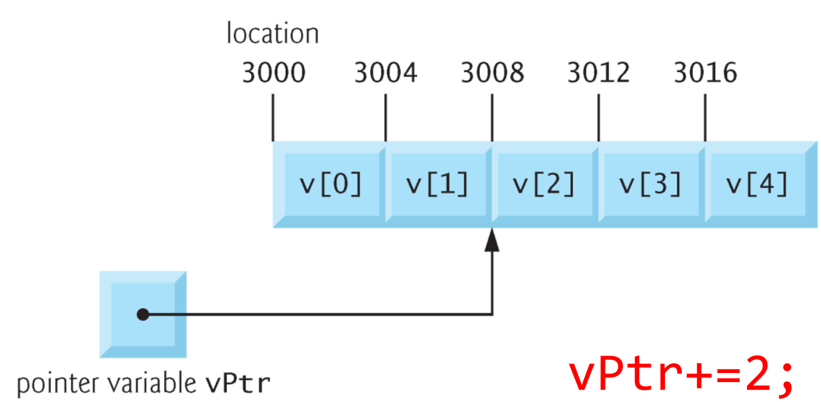
```c
                "\n     sizeof d = %u\tsizeof(double) = %u"
                "\n    sizeof ld = %u\tsizeof(long double) = %u"
                "\n sizeof array = %u"
                "\n   sizeof ptr = %u\n",
                sizeof c, sizeof(char), sizeof s, sizeof(short), sizeof i,
                sizeof(int), sizeof l, sizeof(long), sizeof ll,
                sizeof(long long), sizeof f, sizeof(float), sizeof d,
                sizeof(double), sizeof ld, sizeof(long double),
                sizeof array, sizeof ptr);
}
```

```
     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.





`vPtr+=2;`

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)

```
// Fig. 7.10: fig07_10.c
// Converting a string to uppercase using a
// non-constant pointer to non-constant data.
#include <stdio.h>
#include <ctype.h>

void convertToUppercase(char *sPtr); // prototype

int main(void)
{
   char string[] = "cHaRaCters and $32.98"; // initialize char array

   printf("The string before conversion is: %s", string);
   convertToUppercase(string);
   printf("\nThe string after conversion is: %s\n", string);
}

```

Access array elements by pointer (2)

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

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

Pointer Notation with Arrays (1)

```
// Fig. 7.20: fig07_20.cpp
// Using indexing and pointer notations with arrays.
#include <stdio.h>
#define ARRAY_SIZE 4

int main(void)
{
   int b[] = {10, 20, 30, 40}; // create and initialize array b
   int *bPtr = b; // create bPtr and point it to array b

   // output array b using array index notation
   puts("Array b printed with:\nArray index notation");

   // loop through array b
   for (size_t i = 0; i < ARRAY_SIZE; ++i) {
      printf("b[%u] = %d\n", i, b[i]);
   }

   // output array b using array name and pointer/offset notation
   puts("\nPointer/offset notation where\n"
        "the pointer is the array name");

```

Pointer Notation with Arrays (2)

```
    // loop through array b
    for (size_t offset = 0; offset < ARRAY_SIZE; ++offset) {
       printf("*(b + %u) = %d\n", offset, *(b + offset));
    }

    // output array b using bPtr and array index notation
    puts("\nPointer index notation");

    // loop through array b
    for (size_t i = 0; i < ARRAY_SIZE; ++i) {
       printf("bPtr[%u] = %d\n", i, bPtr[i]);
    }

    // output array b using bPtr and pointer/offset notation
    puts("\nPointer/offset notation");

    // loop through array b
    for (size_t offset = 0; offset < ARRAY_SIZE; ++offset) {
       printf("*(bPtr + %u) = %d\n", offset, *(bPtr + offset));
    }
}
```

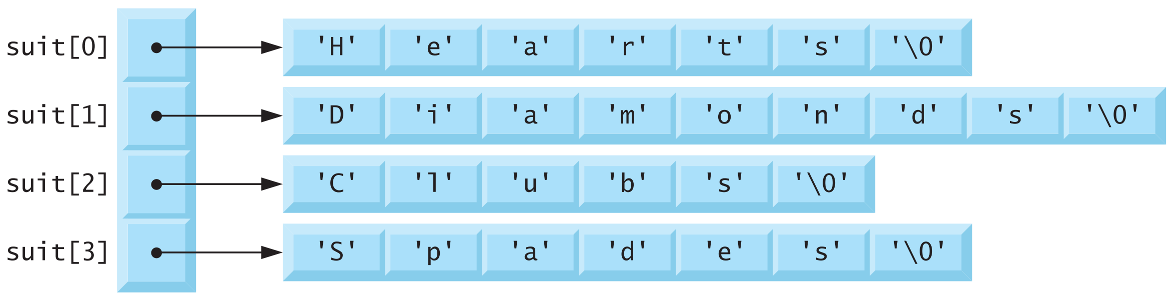
Pointer Notation with Arrays (3)

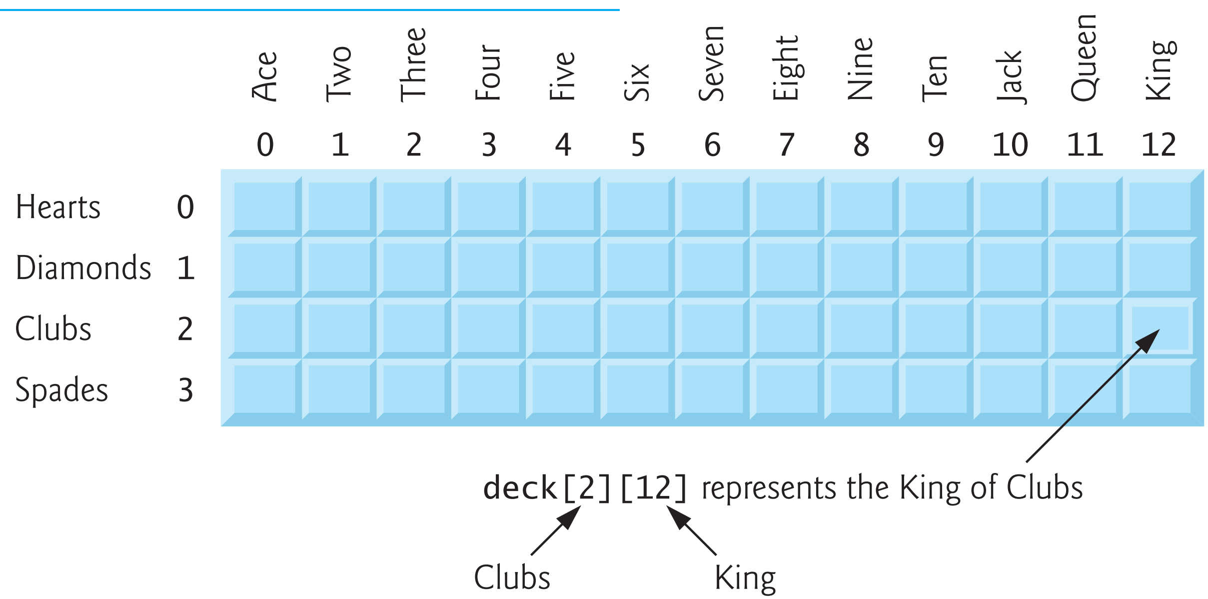
```
Array b printed with:
Array index notation
b[0] = 10
b[1] = 20
b[2] = 30
b[3] = 40

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

```
void
push(char stack[],     /* input/output - the stack */
     char item,        /* input - data being pushed onto the stack */
     int  *top,        /* input/output - pointer to top of stack */
     int  max_size)    /* input - maximum size of stack */
{
     if (*top < max_size-1) {
         ++(*top);
         stack[*top] = item;
     }
}

char
pop(char stack[],     /* input/output - the stack */
    int *top)         /* input/output - pointer to top of stack */
{
     char item;       /* value popped off the stack */

     if (*top >= 0) {
          item = stack[*top];
          --(*top);
     } else {
          item = STACK_EMPTY;
     }

     return (item);
}
```

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);`