# Lecture 9: Pointer

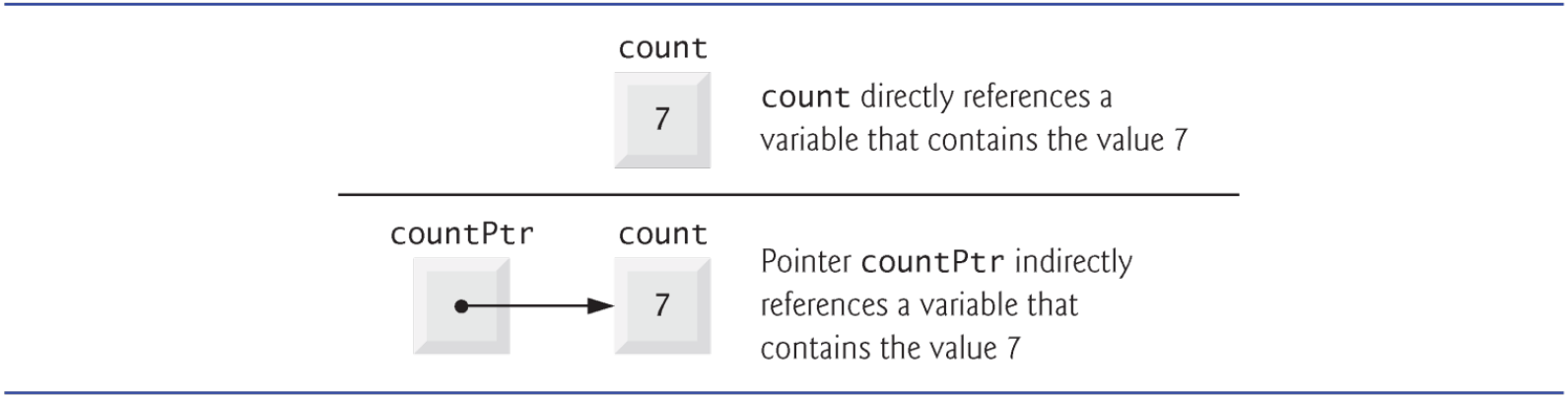
Dr. Tsung-Wei Huang
Department of Electrical and Computer Engineering
University of Utah, Salt Lake City, UT

# Learning Objective

- What pointers are.
- The similarities and differences between pointers and references, and when to use each.
- To use pointers to pass arguments to functions by reference.
- The close relationships between pointers and arrays.
- To use arrays of pointers.
- Basic pointer-based string processing.
- To use pointers to functions.

# Introduction

- A pointer contains the memory address of a variable.
- The variable name directly references a value, and a pointer indirectly references a value.
- Referencing a value through a pointer is called indirection.



# Declaring a Pointer

- **The declaration**
  - `int *countPtr, count;`

  **declares the variable `countPtr` to be of type `int *` (i.e., a pointer to an `int` value)**
  - Read as "`countPtr` is a pointer to `int`."
  - Variable `count` in the preceding declaration is declared to be an `int`, not a pointer to an `int`.
  - Each variable being declared as a pointer must be preceded by an asterisk (*).
- **When * appears in a declaration, it isn't an operator; rather, it indicates that the variable being declared is a pointer.**
- **Pointers can be declared to point to objects of any data type.**

# Value of a Pointer

- Pointers should be initialized either when they're declared or in an assignment.
- A pointer may be initialized to 0, NULL or an address of the corresponding type.
- A pointer with the value 0 or NULL points to nothing and is known as a null pointer.
- The value 0 is the only integer value that can be assigned directly to a pointer variable.

# Pointer Operator

- **The address operator (&) is a unary operator that obtains the memory address of its operand.**
  - Cannot be applied to constants or to expressions that do not result in references
- **Assuming the declarations**

```
int y = 5; // declare variable y
int *yPtr; // declare pointer variable yPtr
```

**the following statement assigns the address of the variable `y` to pointer variable `yPtr`.**

```
yPtr = &y; // assign address of y to yPtr
```

# Pointer Operator (cont'd)

- Assume the integer variable `y` stored at memory location 600000 and pointer variable `yPtr` stored at memory location 500000.
- The * operator, commonly referred to as the indirection operator or dereferencing operator, returns a synonym for the object to which its pointer operand points.
  - Called dereferencing a pointer
- `yPtr = 600000; *yPtr = 5;`

# Example

```
1
2  // Pointer operators & and *.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     int a; // a is an integer
9     int *aPtr; // aPtr is an int * which is a pointer to an integer
10
11    a = 7; // assigned 7 to a
12    aPtr = &a; // assign the address of a to aPtr
13
14    cout << "The address of a is " << &a
15       << "\nThe value of aPtr is " << aPtr;
16    cout << "\n\nThe value of a is " << a
17       << "\nThe value of *aPtr is " << *aPtr;
18    cout << "\n\nShowing that * and & are inverses of "
19       << "each other.\n&*aPtr = " << &*aPtr
20       << "\n*&aPtr = " << *&aPtr << endl;
21 } // end main
```

The & and * operators are inverses of one another.

```
The address of a is 0012F580
The value of aPtr is 0012F580

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

Showing that * and & are inverses of each other.
&*aPtr = 0012F580
*&aPtr = 0012F580
```

# Pass by Pointer (address)

- There are three ways in C++ to pass arguments to a function—pass-by-value, pass-by-reference with reference arguments and pass-by-reference with pointer arguments.
- We explain pass-by-reference with pointer arguments (pass by pointer)
- Pointers, like references, can be used to modify the variables in the caller or to pass pointers to large data objects to avoid the overhead of being passed by value.
- In C++, you can use pointers and the indirection operator (*) to accomplish pass-by-reference.

# Recap: Call By Value

```
1
2  // Pass-by-value used to cube a variable's value.
3  #include <iostream>
4  using namespace std;
5
6  int cubeByValue( int ); // prototype
7
8  int main()
9  {
10    int number = 5;
11
12    cout << "The original value of number is " << number;
13
14    number = cubeByValue( number ); // pass number by value to cubeByValue
15    cout << "\nThe new value of number is " << number << endl;
16 } // end main
17
18 // calculate and return cube of integer argument
19 int cubeByValue( int n )
20 {
21    return n * n * n; // cube local variable n and return result
22 } // end function cubeByValue
```

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

# How to do it with Call by Pointer?

```
1
2  // Pass-by-reference with a pointer argument used to cube a
3  // variable's value.
4  #include <iostream>
5  using namespace std;
6
7  void cubeByReference( int * ); // prototype
8
9  int main()
10 {
11    int number = 5;
12
13    cout << "The original value of number is " << number;
14
15    cubeByReference( &number ); // pass number address to cubeByReference
16
17    cout << "\nThe new value of number is " << number << endl;
18 } // end main
19
20 // calculate cube of *nPtr; modifies variable number in main
21 void cubeByReference( int *nPtr )
22 {
23    *nPtr = *nPtr * *nPtr * *nPtr; // cube *nPtr
24 } // end function cubeByReference
```

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

# Using const with Pointer

- **A nonconstant pointer to nonconstant data**
  **ex:** `int *myPtr = &x;`
  - Both the address and the data can be changed
- **A nonconstant pointer to constant data (Fig. 7.10)**
  **ex:** `const int *myPtr = &x;`
  - Modifiable pointer to a `const int` (data are not modifiable)
- **A constant pointer to nonconstant data (Fig. 7.11)**
  **ex:** `int *const myPtr = &x;`
  - Constant pointer to an `int` (data can be changed, but the address cannot)
- **A constant pointer to constant data (Fig. 7.12)**
  **ex:** `const int *const Ptr = &x;`
  - Both the address and the data are not modifiable

# Example of const pointer

```
1
2  // Attempting to modify data through a
3  // nonconstant pointer to constant data.
4
5  void f( const int * ); // prototype
6
7  int main()
8  {
9     int y;
10
11    f( &y ); // f attempts illegal modification
12 } // end main
13
14 // xPtr cannot modify the value of constant variable to which it points
15 void f( const int *xPtr )
16 {
17    *xPtr = 100; // error: cannot modify a const object
18 } // end function f
```

# Example of const pointer

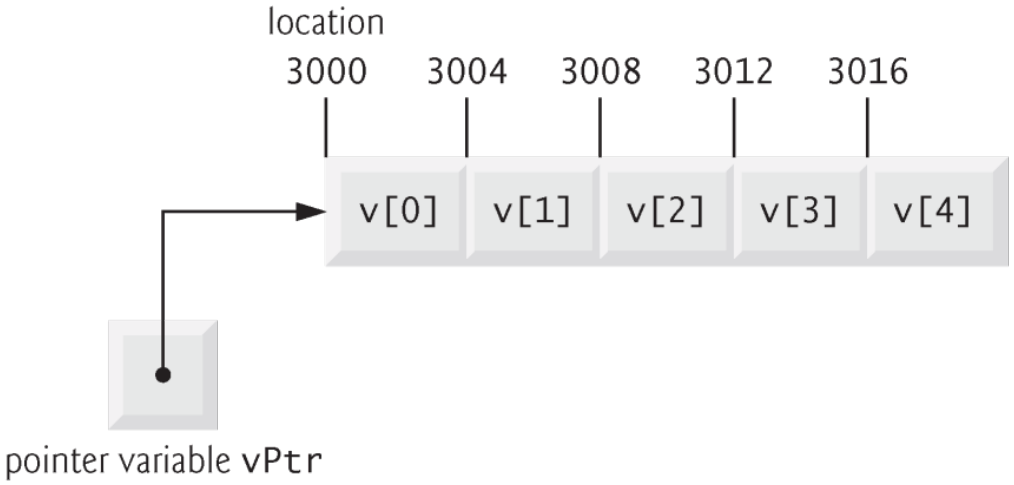
```
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     int x = 5, y;
9
10    // ptr is a constant pointer to a constant integer.
11    // ptr always points to the same location; the integer
12    // at that location cannot be modified.
13    const int *const ptr = &x;
14
15    cout << *ptr << endl;
16
17    *ptr = 7; // error: *ptr is const; cannot assign new value
18    ptr = &y; // error: ptr is const; cannot assign new address
19 } // end main
```

# Pointer Expression and Arithmetic

- **Pointers are valid operands in arithmetic expressions, assignment expressions and comparison expressions.**
- **pointer arithmetic—certain arithmetic operations may be performed on pointers:**
  - increment (++)
  - 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 of the same type

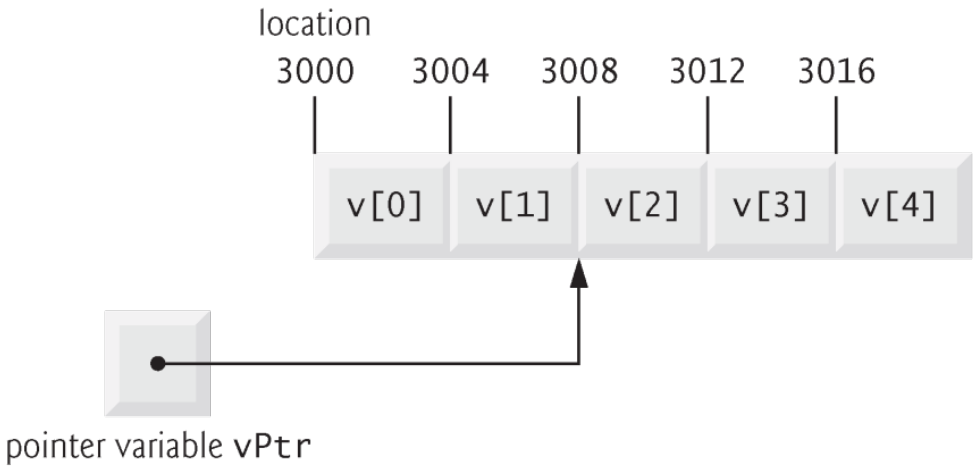
# Pointer Expression and Arithmetic

- Assume that array `int v[5]` has been declared and that its first element is at memory location 3000.
- Assume that pointer `vPtr` has been initialized to point to `v[0]` (i.e., the value of `vPtr` is 3000).



# Pointer Expression and Arithmetic

- **In conventional arithmetic, the addition 3000 + 2 yields the value 3002.**
  - Not true for pointer arithmetic
- **When an integer is added to, or subtracted from, a pointer, the pointer is incremented or decremented by that integer times the size of the object to which the pointer refers**
  - The number of bytes depends on the object's data type.



# Pointer Expression and Arithmetic

- Pointer variables pointing to the same array may be subtracted from one another.
- For example, if `vPtr` contains the address 3000 and `v2Ptr` contains the address 3008, the statement
  - `x = v2Ptr - vPtr;`
- would assign to `x` the number of array elements from `vPtr` to `v2Ptr`—in this case, 2.
- Pointer arithmetic is meaningless unless performed on a pointer that points to an array.

# sizeof Operator

- The unary operator sizeof determines the size of an array (or of any other data type, variable or constant) in bytes during program compilation.
- When applied to the name of an array, the sizeof operator returns the total number of bytes in the array as a value of type size_t.
- When applied to a pointer parameter in a function that receives an array as an argument, the sizeof operator returns the size of the pointer in bytes—not the size of the array.

# Example

```
1
2   // Sizeof operator when used on an array name
3   // returns the number of bytes in the array.
4   #include <iostream>
5   using namespace std;
6
7   size_t getSize( double * ); // prototype
8
9   int main()
10  {
11     double array[ 20 ]; // 20 doubles; occupies 160 bytes on our system
12
13     cout << "The number of bytes in the array is " << sizeof( array );
14
15     cout << "\nThe number of bytes returned by getSize is "
16        << getSize( array ) << endl;
17  } // end main
18
19  // return size of ptr
20  size_t getSize( double *ptr )
21  {
22     return sizeof( ptr );
23  } // end function getSize
```

# sizeof Operator

- The number of elements in an array also can be determined using the results of two `sizeof` operations.
- Consider the following array declaration:
  - `double realArray[ 22 ];`
- To determine the number of elements in the array, the following expression (which is evaluated at compile time) can be used:
  - `sizeof realArray / sizeof( realArray[ 0 ] )`
- The expression determines the number of bytes in array `realArray` and divides that value by the number of bytes used in memory to store the array's first element.

# Bytes of Types

```
1
2   // Demonstrating the sizeof operator.
3   #include <iostream>
4   using namespace std;
5
6   int main()
7   {
8      char c; // variable of type char
9      short s; // variable of type short
10     int i; // variable of type int
11     long l; // variable of type long
12     float f; // variable of type float
13     double d; // variable of type double
14     long double ld; // variable of type long double
15     int array[ 20 ]; // array of int
16     int *ptr = array; // variable of type int *
17
18     cout << "sizeof c = " << sizeof c
19        << "\tsizeof(char) = " << sizeof( char )
20        << "\nsizeof s = " << sizeof s
21        << "\tsizeof(short) = " << sizeof( short )
22        << "\nsizeof i = " << sizeof i
```

# Bytes of Types

```
23          << "\tsizeof(int) = " << sizeof( int )
24          << "\nsizeof l = " << sizeof l
25          << "\tsizeof(long) = " << sizeof( long )
26          << "\nsizeof f = " << sizeof f
27          << "\tsizeof(float) = " << sizeof( float )
28          << "\nsizeof d = " << sizeof d
29          << "\tsizeof(double) = " << sizeof( double )
30          << "\nsizeof ld = " << sizeof ld
31          << "\tsizeof(long double) = " << sizeof( long double )
32          << "\nsizeof array = " << sizeof array
33          << "\nsizeof ptr = " << sizeof ptr << endl;
34    } // end main
```

```
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 f = 4     sizeof(float) = 4
sizeof d = 8     sizeof(double) = 8
sizeof ld = 8    sizeof(long double) = 8
sizeof array = 80
sizeof ptr = 4
```

# Relationship between Pointers and Array

- **The array name (without a subscript) is a constant pointer to the first element of the array.**
  - Although it's a pointer, it cannot be modified in arithmetic expressions.
- **Pointers can be used to do any operation involving array subscripting.**
- **Assume the following declarations:**

```
int b[ 5 ]; // create 5-element int array b
int *bPtr; // create int pointer bPtr
```

- **We can set `bPtr` to the address of the first element in array `b` with the statement**

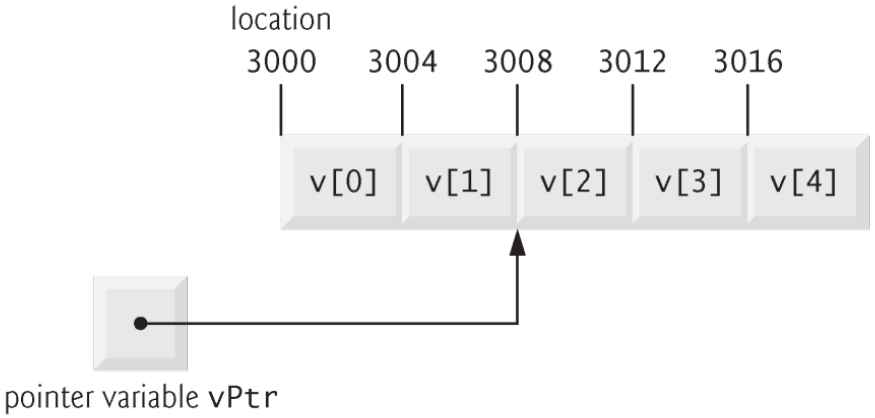
```
bPtr = b; // assign address of array b to bPtr
```

- **equivalent to**

```
bPtr = &b[ 0 ]; // also assigns address of
array b to bPtr
```

# Relationship between Pointers and Array

- **Array element `b[ 2 ]` can alternatively be referenced with the pointer expression**
  - `*( bPtr + 2 )`
- **The `2` in the preceding expression is the offset to the pointer.**
- **This notation is referred to as pointer/offset notation.**
  - The parentheses are necessary, because the precedence of * is higher than that of +.



# Relationship between Pointers and Array

- **Pointers can be subscripted exactly as arrays can.**
- **For example, the expression `bPtr[ 1 ]` refers to the array element `b[ 1 ]`; this expression uses pointer/ subscript notation.**
- **In this section, four notations are discussed for referring to array elements to accomplish the same task :**
  - Array subscript notation,
  - Pointer/offset notation with the array name as a pointer,
  - Pointer subscript notation, and
  - Pointer/offset notation with a pointer

# Relationship between Pointers and Array

```
1  // Fig. 7.18: fig07_18.cpp
2  // Using subscripting and pointer notations with arrays.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     int b[] = { 10, 20, 30, 40 }; // create 4-element array b
9     int *bPtr = b; // set bPtr to point to array b
10
11    // output array b using array subscript notation
12    cout << "Array b printed with:\n\nArray subscript notation\n";
13
14    for ( int i = 0; i < 4; i++ )
15       cout << "b[" << i << "] = " << b[ i ] << '\n';
16
17    // output array b using the array name and pointer/offset notation
18    cout << "\nPointer/offset notation where "
19       << "the pointer is the array name\n";
20
21    for ( int offset1 = 0; offset1 < 4; offset1++ )
22       cout << "*(b + " << offset1 << ") = " << *( b + offset1 ) << '\n';
```

# Relationship between Pointers and Array

```
23
24      // output array b using bPtr and array subscript notation
25      cout << "\nPointer subscript notation\n";
26
27      for ( int j = 0; j < 4; j++ )
28         cout << "bPtr[" << j << "] = " << bPtr[ j ] << '\n';
29
30      cout << "\nPointer/offset notation\n";
31
32      // output array b using bPtr and pointer/offset notation
33      for ( int offset2 = 0; offset2 < 4; offset2++ )
34         cout << "*(bPtr + " << offset2 << ") = "
35            << *( bPtr + offset2 ) << '\n';
36   } // end main
```

```
Array b printed with:

Array subscript notation
b[0] = 10
b[1] = 20
b[2] = 30
b[3] = 40
```

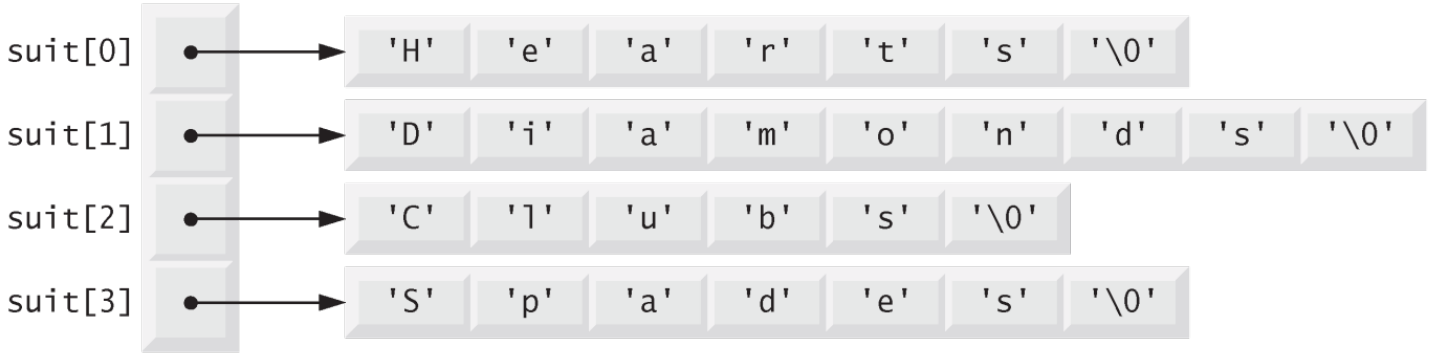
# Relationship between Pointers and Array

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

Pointer subscript 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
```

# Pointer-based String

- **Character constant**
  - An integer value represented as a character in single quotes.
  - The value of a character constant is the integer value of the character in the machine's character set.
- **A string is a series of characters treated as a single unit.**
  - May include letters, digits and various special characters such as +, -, *, /and $.
- **String literals, or string constants, in C++ are written in double quotation marks**
- **A pointer-based string is an array of characters ending with a null character ( \0 ).**
  - Ex: "happy" → 'h', 'a', 'p', 'p', 'y', '\0'

# Pointer-based String




**cout << suit[0]; → print out "Hearts"**

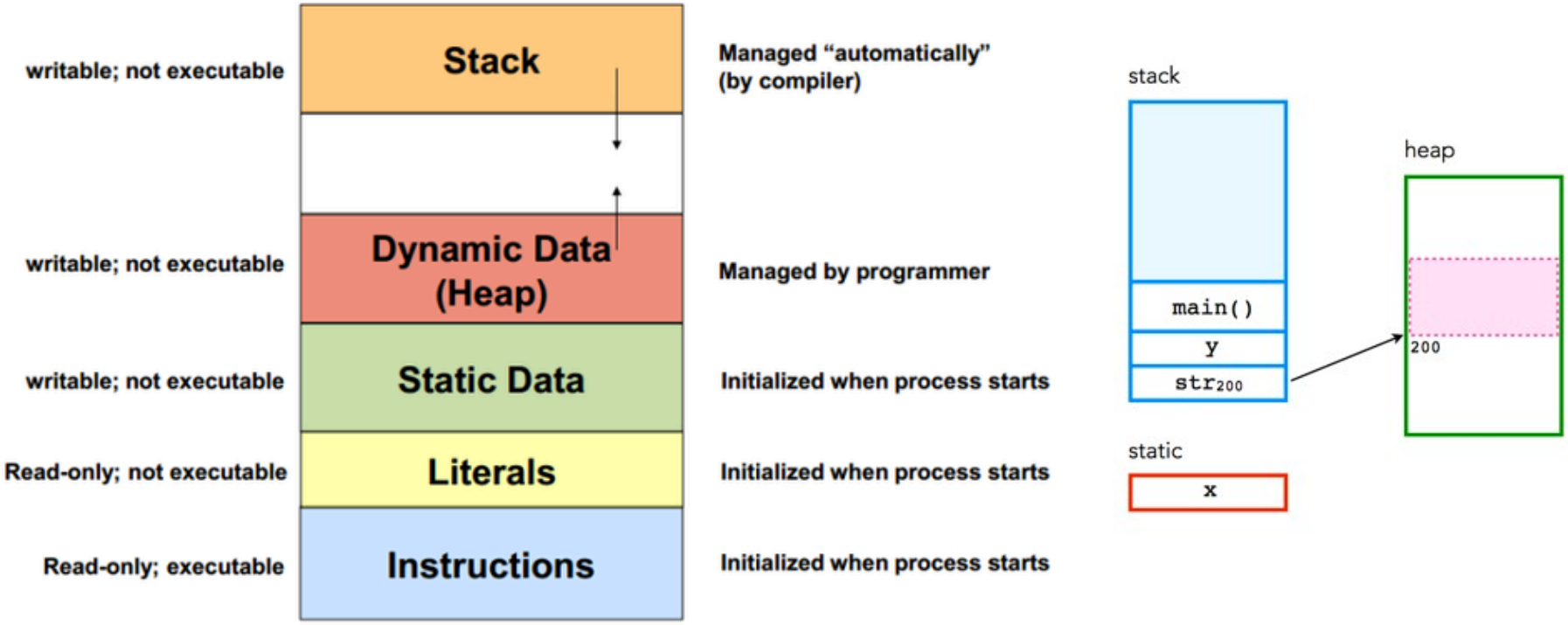
**cout << suit[2]; → print out "Clubs"**

**What will be printed by the following program?**

```
for (int i=0; i<10; i++) {
        idx = rand()%4;
        cout << suit[idx];
}
```

# Memory Layout

- Stack vs Heap



# Stack

- **The place where *arguments* of a function call are stored**
- **The place where *registers* of the calling function are saved**
- **The place where *local data* of called function is allocated**
  - *Automatic* data
- **The place where called function leaves *result* for calling function**
- **Supports recursive function calls**
- **…**

# Stack

- Imagine the following program:–

```
int factorial(int n){
  if (n <= 1)
    return (1);
  else
    int y = factorial(n-1);
    return (y * n);
}
```

- Imagine also the caller:–

```
int x = factorial(100);
```

- What does compiled code look like?

# Compiled Code: Caller

```
int x = factorial(100);
```

- Put the value "100" somewhere that *factorial* function can find
- Put the current program counter somewhere so that *factorial* function can return to the right place in *calling* function
- Provide a place to put the result, so that *calling function* can find it

# Compiled Code: Factorial Function

- Save the *caller*'s registers somewhere
- Get the argument *n* from the agreed-upon place
- Set aside some memory for local variables and intermediate results – i.e., *y, n - 1*

- Do whatever *factorial* was programmed to do

- Put the result where the *caller* can find it
- Restore the *caller*'s registers
- Transfer back to the program counter saved by the *caller*

# Somewhere?

- So that *caller* can provide as many arguments as needed (within reason)?
- So that *called routine* can decide at run-time how much temporary space is needed?

- So that *called routine* can call any other routine, potentially recursively?

# Answer: a Stack

- ***Stack*** **– a linear data structure in which items are added and removed in *last-in, first-out* order.**

- **Calling program**
  - *Push* arguments & return address onto stack
  - After return, *pop* result off stack

# Stack with Called Routine

## ❑ Called routine

- *Push* registers and return address onto stack
- *Push* temporary storage space onto stack

- Do work of the routine

- Pop registers and temporary storage off stack
- Leave result on stack
- Return to address left by calling routine

# Stack

- **All modern programming languages require a stack**
  - Fortran and Cobol did not (non-recursive)
- **All modern processors provide a designated *stack pointer* register**
- **All modern process address spaces provide room for a stack**
  - Able to grow to a large size
  - May grow upward or downward

# Heap

- **A place for allocating memory that is not part of *last-in, first-out* discipline**
- **I.e., dynamically allocated data structures that survive function calls**
  - E.g., strings in C
  - **`new`** objects in C++, Java, etc.

# Allocate Memory from Heap

- ***malloc*() – POSIX standard function**
  - Allocates a chunk of memory of desired size
  - Remembers size
  - Returns pointer
- ***free* () – POSIX standard function**
  - Returns previously allocated chunk to heap for reallocation
  - Assumes that pointer is correct!

# Allocate Memory from Heap

- ***malloc*() – POSIX standard function**
  - Allocates a chunk of memory of desired size
  - Remembers size
  - Returns pointer
- ***free* () – POSIX standard function**
  - Returns previously allocated chunk to heap for reallocation
  - Assumes that pointer is correct!
- ***Storage leak* – failure to *free* something**

# Heap in Modern Systems

- **Many modern programming languages require a heap**
  - C++, Java, etc.
  - *NOT* Fortran
- **Typical process environment**
  - Heap grows toward stack — but never shrinks!
- **Multi-threaded environments**
  - All threads *share* the same heap
  - Data structures may be passed from one thread to another.

# How to Detect Memory Leak?


https://valgrind.org/

# Summary

- Pointer
- Call by value, address, reference
- Memory layout

# LAB Today

- Practicing the use of pointer arithmetic