Object Oriented Programming by C++

Advanced Data Types

Pointer, Vector and Array

2017. 8.

Sungwon Lee / Professor

Email: drsungwon@khu.ac.kr
Web: http://mobilelab.khu.ac.kr/

Modified & taught by Jinwoo Choi
in 2023 Spring
Email: jinwoochoi@khu.ac.kr
Webpage: https://sites.google.com/site/jchoivision/

Contents

- Pointer
- Pass by Pointer
- Higher-order Functions (Function Pointer)
- Vectors
- Arrays

Pointer

& Operator

Ordinarily we need not be concerned about where variables live in the computer's memory during a program's execution. The compiler generates machine code that takes care of those details for us. Some systems software like operating systems and device drivers need to access specific memory locations in order to interoperate with hardware. Systems programmers, therefore, must be able to write code that can access such lower-level detail. Developers of higher-level applications sometimes need to access the address of variables to achieve specialized effects.

Each byte in a computer's memory is numbered with a unique address. The first address is 0, and the locations are numbered sequentially up to some maximum value allowed by the operating system and hardware. A C++ variable is stored in memory, so each variable lives at a particular address.

If x is a variable of any type, the expression

```
&x          cout<<&x;
```

represents the *address* of x. The & operator is called the *address of* operator. Regardless of the type of x, the expression

```
&x
```

is really just a number—the numeric address of the variable's memory location, but except for some situations in systems programming, programmers rarely need to treat this value as a number.

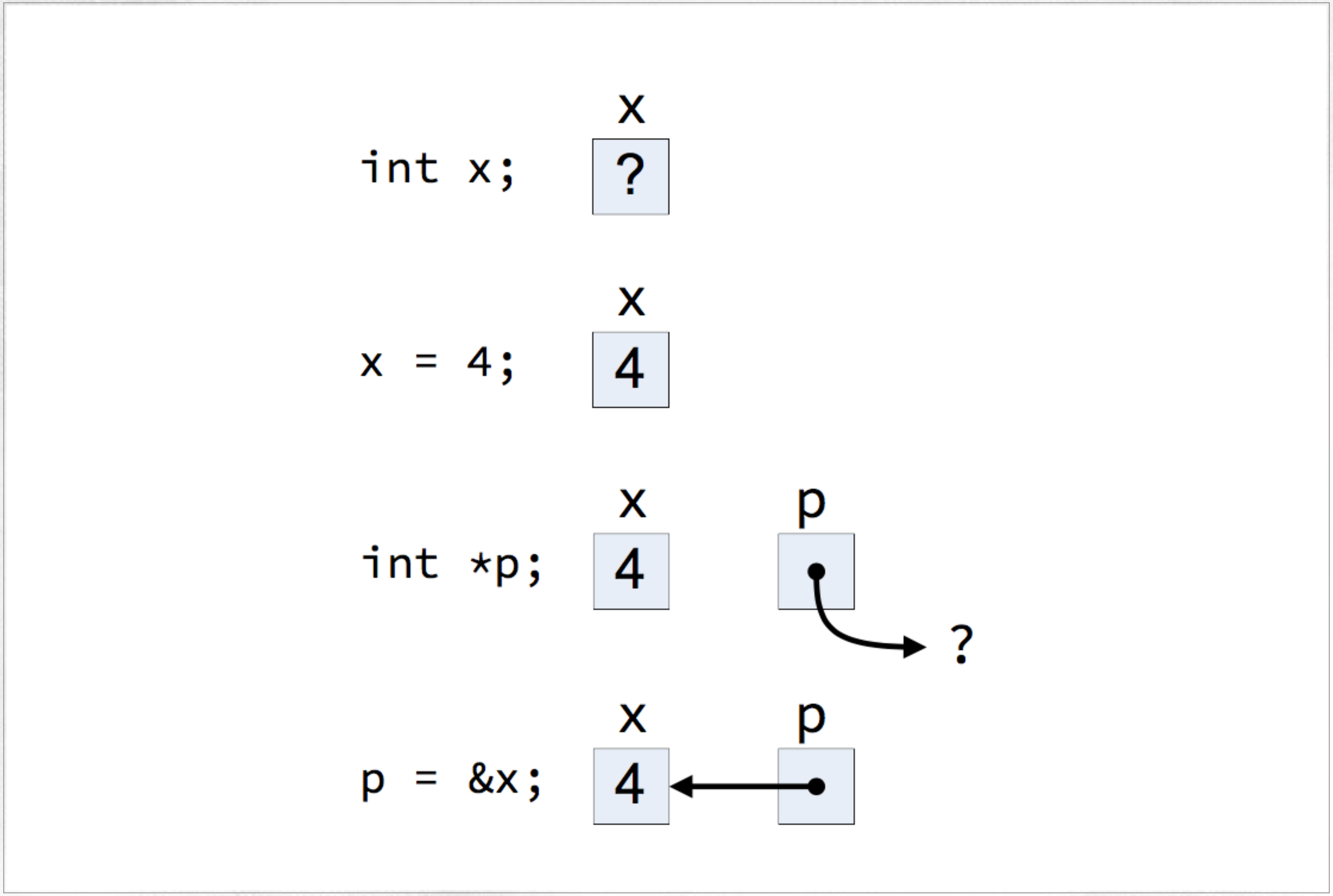
Pointer Variable

포인터: 어떤 변수의 주소를 저장하는 변수

Pointer variable stores the address of memory:

```
int *p;
```

Pointer Declaration and Assignment



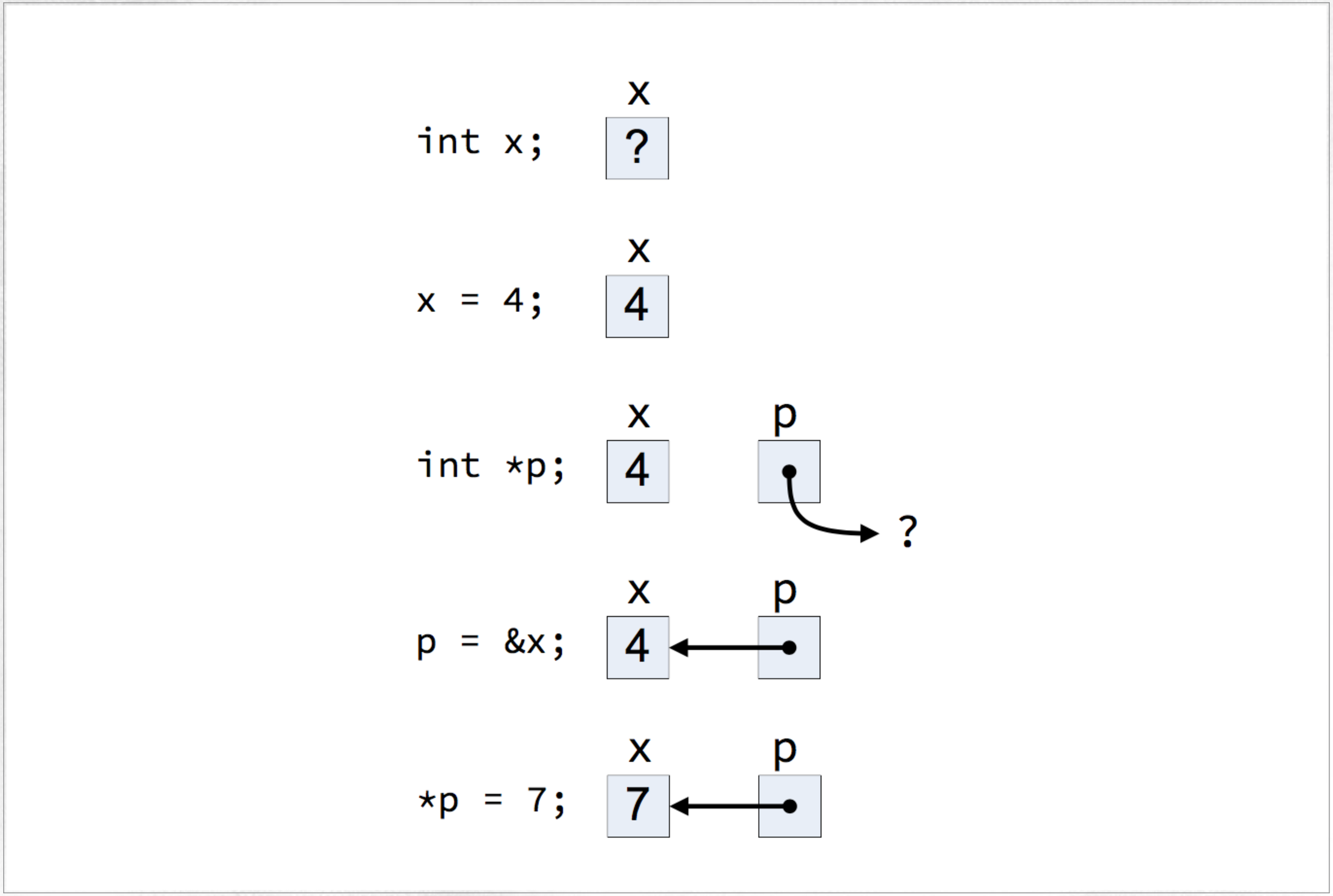
* Operator for Pointer Variable

*The * operator is the inverse of the & operator. & finds the address of a variable, and, when this address is assigned to a pointer variable like p, the * operator accesses the memory referenced by the pointer.*

```
cout << *p;
```

// Print the value at the memory address
// which is pointed by pointer p

Assignment via a Pointer



Example Software

```
#include <iostream>
using namespace std;

int main()
{
   int x = 4;
   int *p;

   cout << endl << "(1)[address of x: " << &x << "]" << "[value of x: " << x << "]";
   cout << endl << "(1)[pointed by p: " << p    << "]" << "[value of pointed p : " << *p <<"]";

   p = &x;

   cout << endl << "(2)[address of x: " << &x << "]" << "[value of x: " << x << "]";
   cout << endl << "(2)[pointed by p: " << p    << "]" << "[value of pointed p : " << *p <<"]";

   *p = 7;

   cout << endl << "(3)[address of x: " << &x << "]" << "[value of x: " << x << "]";
   cout << endl << "(3)[pointed by p: " << p    << "]" << "[value of pointed p : " << *p <<"]";

   return 0;
}
```

Example Software with Results

```
(1)[address of x: 0x7fff5fbff5e8][value of x: 4]
(1)[pointed by p: 0x7fff5fbff658][value of pointed p : -684363211]
(2)[address of x: 0x7fff5fbff5e8][value of x: 4]
(2)[pointed by p: 0x7fff5fbff5e8][value of pointed p : 4]
(3)[address of x: 0x7fff5fbff5e8][value of x: 7]
(3)[pointed by p: 0x7fff5fbff5e8][value of pointed p : 7]
```

```
#include <iost
using namesp

int main()
{
   int x = 4;
   int *p;

   cout << endl << "(1)[address of x: " << &x << "]" << "[value of x: " << x << "]";
   cout << endl << "(1)[pointed by p: " << p    << "]" << "[value of pointed p : " << *p <<"]";

   p = &x;

   cout << endl << "(2)[address of x: " << &x << "]" << "[value of x: " << x << "]";
   cout << endl << "(2)[pointed by p: " << p    << "]" << "[value of pointed p : " << *p <<"]";

   *p = 7;

   cout << endl << "(3)[address of x: " << &x << "]" << "[value of x: " << x << "]";
   cout << endl << "(3)[pointed by p: " << p    << "]" << "[value of pointed p : " << *p <<"]";

   return 0;
}
```

Example Software with Results

```
(1)[address of x: 0x7fff5fbff5e8][value of x: 4]
(1)[pointed by p: 0x7fff5fbff658][value of pointed p : -684363211]
(2)[address of x: 0x7fff5fbff5e8][value of x: 4]
(2)[pointed by p: 0x7fff5fbff5e8][value of pointed p : 4]
(3)[address of x: 0x7fff5fbff5e8][value of x: 7]
(3)[pointed by p: 0x7fff5fbff5e8][value of pointed p : 7]
```

```
#include <iost
using namesp

int main()
{
   int x = 4;
   int *p;

   cout << endl << "(1)[address of x: " << &x << "]" << "[value of x: " << x << "]";
   cout << endl << "(1)[pointed by p: " << p    << "]" << "[value of pointed p : " << *p <<"]";

   p = &x;

   cout << endl << "(2)[address of x: " << &x << "]" << "[value of x: " << x << "]";
   cout << endl << "(2)[pointed by p: " << p    << "]" << "[value of pointed p : " << *p <<"]";

   *p = 7;

   cout << endl << "(3)[address of x: " << &x << "]" << "[value of x: " << x << "]";
   cout << endl << "(3)[pointed by p: " << p    << "]" << "[value of pointed p : " << *p <<"]";

   return 0;
}
```

Passing Address of Variable for Function Call

Listing 10.18: swapwithpointers.cpp

```
#include <iostream>

/*
 *  swap(a, b)
 *      Interchanges the values of memory
 *      referenced by its parameters a and b.
 *      It effectively interchanges the values
 *      of variables in the caller's context.
 */
void swap(int *a, int *b) {
   int temp = *a;
   *a = *b;
   *b = temp;
}

/*
 *  main
 *      Interchanges the values of two variables
 *      using the swap function.
 */
int main() {
    int var1 = 5, var2 = 19;
    std::cout << "var1 = " << var1 << ", var2 = " << var2 << '\n';
    swap(&var1, &var2);
    std::cout << "var1 = " << var1 << ", var2 = " << var2 << '\n';
}
```

Passing Address of Variable for Function Call

```
#include <iostream>
using namespace std;

void swap(int *a, int *b)
{
   int temp = *a;

   std::cout << "a = " << a << " ( " << *a << " ), “ << "b = " << b << " ( " << *b << " )" << '\n';

   *a = *b;
   *b = temp;

   std::cout << "a = " << a << " ( " << *a << " ), “ << "b = " << b << " ( " << *b << " )" << '\n';
}

int main()
{
   int var1 = 5, var2 = 19;

   std::cout << "var1 = " << var1 << " ( " << &var1 << " ), “
             << "var2 = " << var2 << " ( " << &var2 << " )" << '\n';

   swap(&var1, &var2);

   std::cout << "var1 = " << var1 << " ( " << &var1 << " ), “
             << "var2 = " << var2 << " ( " << &var2 << " )" << '\n';
}
```

Passing Address of Variable for Function Call

```
var1 = 5 ( 0x7fff5fbff64c ), var2 = 19 ( 0x7fff5fbff648 )
a = 0x7fff5fbff64c ( 5 ), b = 0x7fff5fbff648 ( 19 )
a = 0x7fff5fbff64c ( 19 ), b = 0x7fff5fbff648 ( 5 )
var1 = 19 ( 0x7fff5fbff64c ), var2 = 5 ( 0x7fff5fbff648 )
```

```
#include <iostream>
using namespace std;

void swap(int *a, int *b)
{
   int temp = *a;

   std::cout << "a = " << a << " ( " << *a << " ), “ << "b = " << b << " ( " << *b << " )" << '\n';

   *a = *b;
   *b = temp;

   std::cout << "a = " << a << " ( " << *a << " ), “ << "b = " << b << " ( " << *b << " )" << '\n';
}

int main()
{
   int var1 = 5, var2 = 19;

   std::cout << "var1 = " << var1 << " ( " << &var1 << " ), “
             << "var2 = " << var2 << " ( " << &var2 << " )" << '\n';

   swap(&var1, &var2);

   std::cout << "var1 = " << var1 << " ( " << &var1 << " ), “
             << "var2 = " << var2 << " ( " << &var2 << " )" << '\n';
}
```

Function Pointer

- Function Pointer
  - Pointer variable can store the address of Function
  - Function pointer can point the functions with same input and output types
  - REMEMBER the calling style~!

Listing 10.20: arithmeticeval.cpp

```
#include <iostream>

int add(int x, int y) {
    return x + y;
}

int multiply(int x, int y) {
    return x * y;
}

int evaluate(int (*f)(int, int), int x, int y) {
    return f(x, y);
}

int main() {
    std::cout << add(2, 3) << '\n';
    std::cout << multiply(2, 3) << '\n';
    std::cout << evaluate(&add, 2, 3) << '\n';
    std::cout << evaluate(&multiply, 2, 3) << '\n';
}
```

Function Pointer

- Function Pointer
  - Function pointers are not restricted to function parameters.
  - Given the definition of add above, the following code fragment is legal:

```
// Declare func to be a pointer to a function that accepts
// two integer parameters and returns an integer result
int (*func)(int, int);

// Assign add to func
func = add;

// Call add through func
std::cout << func(7, 2) << '\n';
```

  - Higher-order functions via function pointers provide a powerful tool for developing flexible programs. With functions as parameters we can dynamically customize the behavior of a function, essentially “plugging in” new functionality by passing in different functions.

Multiple variables with Linear ordering

- What is Vector
  - A vector in C++ is ***an object that manages a block of memory*** that can hold multiple values simultaneously; a vector, therefore, represents a ***collection of values***
  - A vector has a name, and we may ***access the values it contains via their position*** within the block of memory managed by the vector
  - A vector stores a sequence of values, and the values must ***all be of the same type***
  - A collection of values all of the same type is said to be ***homogeneous***.

Vector declaration

- Vector is an Standard C++ Object
  - Likes console input & output, file and string
  - Step.1: Header including

```
#include <vector>
```

  - Step.2: Namespace definition

```
using namespace std; // or using std::vector statement
```

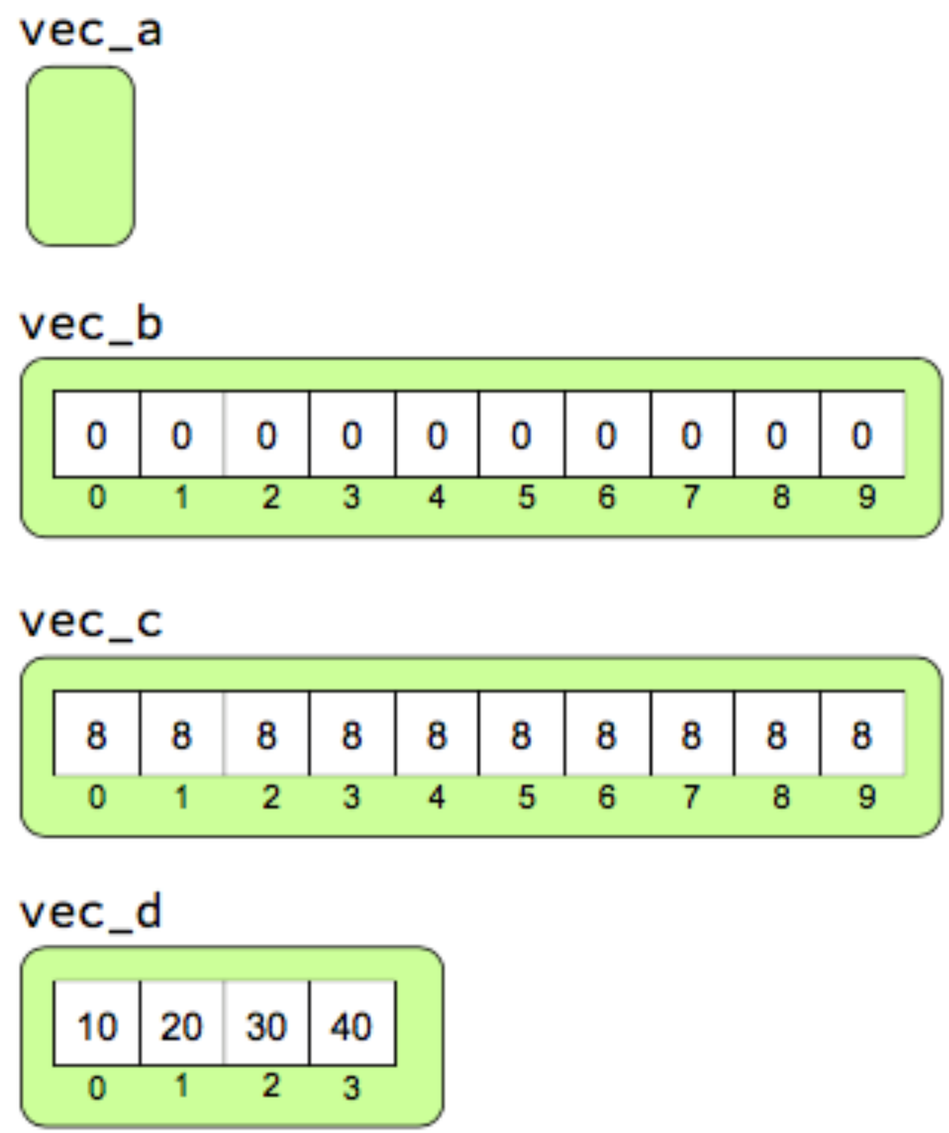
  - Step.3: Make your own vector variable

```
vector<int> vec_a; // vector with name only
vector<int> vec_b(10); // vector with initial size
vector<int> vec_c(10, 8); // vector with initial size and initial value
vector<int> vec_d{ 10, 20, 30, 40 }; // vector with specific values
```

Vector declaration in Computer memory

- Vector is an linear storage
  - Vectors store their elements in a contiguous block of memory
  - For example, the memory occupied by the element at index 2 follows immediately after the memory occupied by the element at index 1 and immediately before the element at index 3

```
vector<int> vec_a;
vector<int> vec_b(10);
vector<int> vec_c(10, 8);
vector<int> vec_d{ 10, 20, 30, 40 };
```



Important note for C++ Version

The vector initializer list syntax:

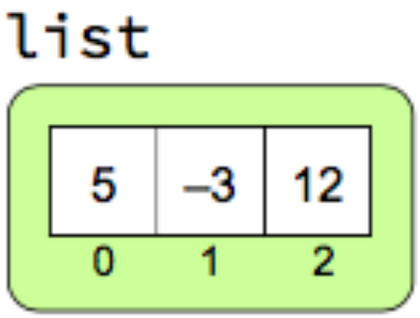
```
std::vector<int> vec_d{10, 20, 30, 40, 50};
```

is a C++11 language feature and, therefore, is not be supported by older C++ compilers; in particular, the C++ compilers available for Visual Studio editions prior to Visual Studio 2013 do not support this initializer list syntax.

Vector element indexing and usage

Figure 11.2 The numbers below the boxes represent indices, or positions, within the vector. Note that the first element in a vector is found at index 0, not 1.

```
vector<int> list(3);
list[0] = 5;
list[1] = -3;
list[2] = 12;
```




```
std::vector<int> list(3);  // Declare list to be a vector of three int
list[0] = 5 ;              // Make the first element 5
list[1] = -3 ;             // Make the second element -3
list[2] = 12 ;             // Make the last element 12
std::cout << list[1] << '\n'; // Print the element at index 1
```

0 to Size-1

Vector element indexing and usage

An element of a vector accessed via its index behaves just like a variable of that type; for example, suppose we declare `nums` as

```
std::vector<double> nums(10);
```

This declaration specifies a collection of 10 double-precision floating-point elements. The following code fragment shows how we can manipulate some of those elements:

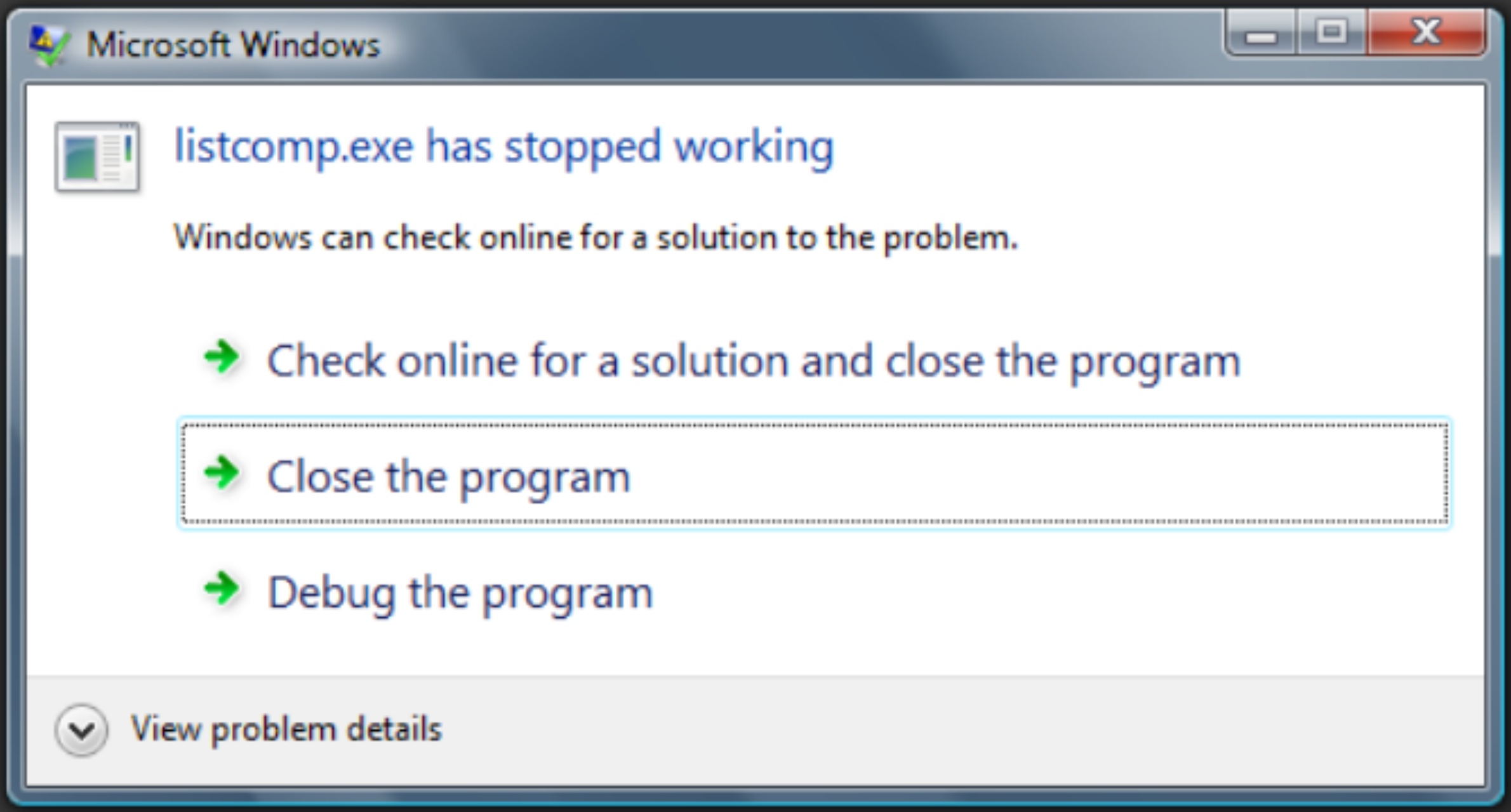
```
// Print the fourth element
std::cout << nums[3] << '\n';
// The third element is the average of the first and last elements
nums[2] = (nums[0] + nums[9])/2;
// Assign elements at indices 1 and 4 from user input
std::cin >> nums[1] >> nums[4];
```

Note that the `<<` operator associated with the `std::cout` output stream object is not designed to work with vector objects as a whole:

```
std::cout << nums << '\n';    // <--- Will not compile!
```

0 to Size-1

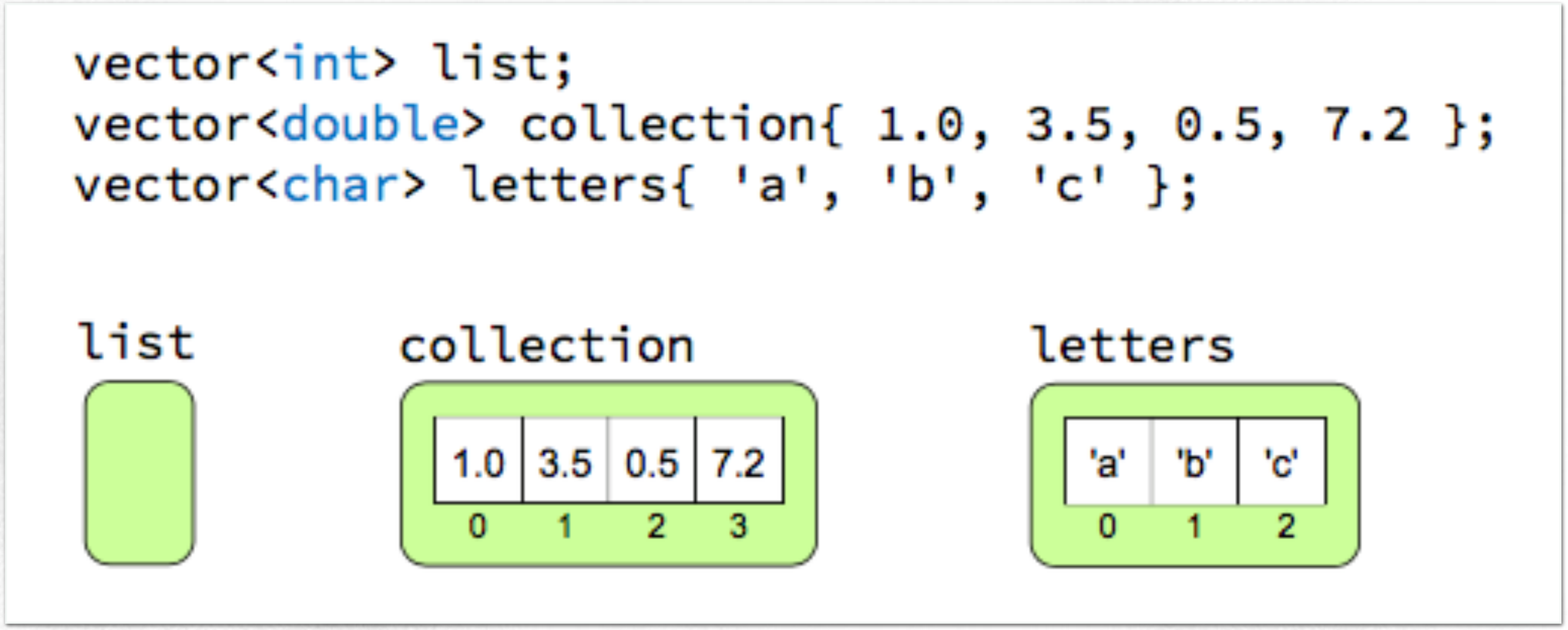
Wrong indexing? (RUN-TIME ERROR)



Vector element types

- Vectors may hold any valid C++ data type
- The following code fragment declares three vectors of differing types:

```
std::vector<int> list;
std::vector<double> collection{ 1.0, 3.5, 0.5, 7.2 };
std::vector<char> letters{ 'a', 'b', 'c' };
```



Traversing a Vector

Listing 11.3: vectoraverage.cpp

```
#include <iostream>
#include <vector>

int main() {
    double sum = 0.0;
    const int NUMBER_OF_ENTRIES = 5;
    std::vector<double> numbers(NUMBER_OF_ENTRIES);

    std::cout << "Please enter " << NUMBER_OF_ENTRIES << " numbers: ";
    // Allow the user to enter in the values.
    for (int i = 0;  i < NUMBER_OF_ENTRIES;  i++) {
        std::cin >> numbers[i];
        sum += numbers[i];
    }
    std::cout << "The average of ";
    for (int i = 0;  i < NUMBER_OF_ENTRIES - 1;  i++)
        std::cout << numbers[i] << ", ";
    // No comma following last element
    std::cout << numbers[NUMBER_OF_ENTRIES - 1] << " is "
              << sum/NUMBER_OF_ENTRIES << '\n';
}
```

Traversing a Vector (Simplified Grammar)

Listing 11.5: foreachexample.cpp

```
#include <iostream>
#include <vector>

int main() {
    // Declare a vector of ten numbers
    std::vector<double> vec(10);

    // Allow the user to populate the vector
    std::cout << "Please enter 10 numbers: ";
    for (double& elem : vec)
        std::cin >> elem;

    // Print the vector's contents
    for (double elem : vec)
        std::cout << elem << '\n';
}
```

elem is replaced with vec[0] to [9] for each repetition.

Reference variable (&) can be used to change value.

Vector methods (!! Vector is object !!)

- `push_back`—inserts a new element onto the back of a vector
- `pop_back`—removes the last element from a vector
- `operator[]`—provides access to the value stored at a given index within the vector
- `at`—provides bounds-checking access to the value stored at a given position within the vector
- `size`—returns the number of values currently stored in the vector
- `empty`—returns true if the vector contains no elements; returns false if the vector contains one or more elements
- `clear`—makes the vector empty.

```
std::vector<int> list;   // Declare list to be a vector
list.push_back(5);       // Add 5 to the end of list
list.push_back(-3);      // Add -3 to the end of the list
list.push_back(12);      // Add 12 to the end of list
list.pop_back();         // Removes 12 from the list
list.pop_back();         // Removes -3 from the list
```

Vector methods (!! Vector is object !!)

Listing 11.6: vectoroutofbounds2.cpp

```
#include <iostream>
#include <vector>

int main() {
    const int SIZE = 3;
    std::vector<int> a{5, 5, 5};
    // Print the contents of the vector
    std::cout << "a contains ";
    for (int i = 0; i < SIZE; i++)
        std::cout << a.at(i) << " ";
    std::cout << '\n';
    // Change all the 5s in vector a to 8s
    for (int i = 0; i <= SIZE; i++)   // Bug: <= should be <
        a.at(i) = 8;
    // Reprint the contents of the vector
    std::cout << "a contains ";
    for (int i = 0; i < SIZE; i++)
        std::cout << a.at(i) << " ";
    std::cout << '\n';
}
```

Vector methods (!! Vector is object !!)

Listing 11.6: vectoroutofbounds2.cpp

```
#include <iostream>
#include <vector>

int main() {
    const int SIZE = 3;
    std::vector<int> a{5, 5, 5};
    // Print the contents of the vector
    std::cout << "a contains ";
    for (int i = 0; i < SIZE; i++)
        std::cout << a.at(i) << " ";
    std::cout << '\n';
    // Change all the 5s in vector a to 8s
    for (int i = 0; i <= SIZE; i++)   // Bug: <= should be <
        a.at(i) = 8;
    // Reprint the contents of the vector
    std::cout << "a contains ";
    for (int i = 0; i < SIZE; i++)
        std::cout << a.at(i) << " ";
    std::cout << '\n';
}
```

Enhanced like:

```
for (unsigned i = 0; i < list.size(); i++)
    std::cout << list[i] << " ";
std::cout << '\n';
```

Passing Vector (default: pass by value) 1/2

Listing 11.7: vectortofunc.cpp

```
#include <iostream>
#include <vector>

/*
 *  print(v)
 *     Prints the contents of an int vector
 *     v is the vector to print
 */
void print(std::vector<int> v) {
    for (int elem : v)
        std::cout << elem << " ";
    std::cout << '\n';
}

/*
 *  sum(v)
 *     Adds up the contents of an int vector
 *     v is the vector to sum
 *     Returns the sum of all the elements
 *     or zero if the vector is empty.
 */
int sum(std::vector<int> v) {
```

Passing Vector (default: pass by value) 2/2

```
    int result = 0;
    for (int elem : v)
        result += elem;
    return result;
}

int main() {
    std::vector<int> list{ 2, 4, 6, 8, };
    // Print the contents of the vector
    print(list);
    // Compute and display sum
    std::cout << sum(list) << '\n';
    // Zero out all the elements of list
    int n = list.size();
    for (int i = 0; i < n; i++)
        list[i] = 0;
    // Reprint the contents of the vector
    print(list);
    // Compute and display sum
    std::cout << sum(list) << '\n';
}
```

Passing Vector (why not: pass by Reference)

Listing 11.7: vectortofunc.cpp

```
#include <iostream>
#include <vector>

/*
 *  print(v)
 *      Prints the contents of an int vector
 *      v is the vector to print
 */
void print(std::vector<int> v) {
    for (int elem : v)
        std::cout << elem << " ";
    std::cout << '\n';
}

/*
 *  sum(v)
 *      Adds up the conte
 *      v is the vector t
 *      Returns the sum o
 *      or zero if the ve
 */
int sum(std::vector<int> v) {
```

if you want to modify the values in 'v':

```
void print(      std::vector<int>& v) {
    for (int elem : v)
        std::cout << elem << " ";
    std::cout << '\n';
}
```

Returning Vector (1/2)

Listing 11.9: primelist.cpp

```
#include <iostream>
#include <vector>
#include <cmath>

/*
 *  print(v)
 *     Prints the contents of an int vector
 *     v is the vector to print; v is not modified
 */
void print(const std::vector<int>& v) {
    for (int elem : v)
        std::cout << elem << " ";
    std::cout << '\n';
}

/*
 *  is_prime(n)
 *     Determines the primality of a given value
 *     n an integer to test for primality
 *     Returns true if n is prime; otherwise, returns false
 */
bool is_prime(int n) {
    if (n < 2)
        return false;
    else {
        bool result = true;  // Provisionally, n is prime
        double r = n, root = sqrt(r);
        // Try all possible factors from 2 to the square
        // root of n
        for (int trial_factor = 2; result && trial_factor <= root;
              trial_factor++)
            result = (n % trial_factor != 0);
        return result;
    }
}

/*
```

Due to the fact that vectors may contain a large number of elements, you usually should pass vectors to functions as reference parameters rather than value parameters:

- If the function *is* meant to modify the contents of the vector, declare the vector as a non-`const` reference (that is, omit the `const` keyword when declaring the parameter).
- If the function is *not* meant to modify the contents of the vector, declare the vector as a `const` reference.

It generally is safe to return a vector by value from a function if that vector is declared local to the function. Modern C++ compilers generate optimized code that avoid the overhead of copying the result back to the caller.

Returning Vector (2/2)

```
 *  primes(begin, end)
 *      Returns a vector containing the prime
 *      numbers in the range begin...end.
 *      begin is the first number in the range
 *      end is the last number in the range
 */
std::vector<int> primes(int begin, int end) {
    std::vector<int> result;
    for (int i = begin; i <= end;  i++)
        if (is_prime(i))
            result.push_back(i);
    return result;
}

int main() {
    int low, high;
    std::cout << "Please enter lowest and highest values in "
              << "the range: ";
    std::cin >> low >> high;
    std::vector<int> prime_list = primes(low, high);
    print(prime_list);
}
```

Multidimensional Vectors

- Two-dimensional (2D) vector

```
std::vector<std::vector<int>> a(2, std::vector<int>(3));
```

- Step.1:
  - There is a one-dimensional (1D) vector
  - It has three integer elements
- Step.2:
  - 2D vector's element is 1D vector
  - Thus, a has two 1D vectors, and each 1D vector has 3 integers

Multidimensional Vector manipulation

- Two-dimensional (2D) vector can be understood by:
  - Row
  - Column

std::vector<std::vector<int>> a(2, std::vector<int>(3));

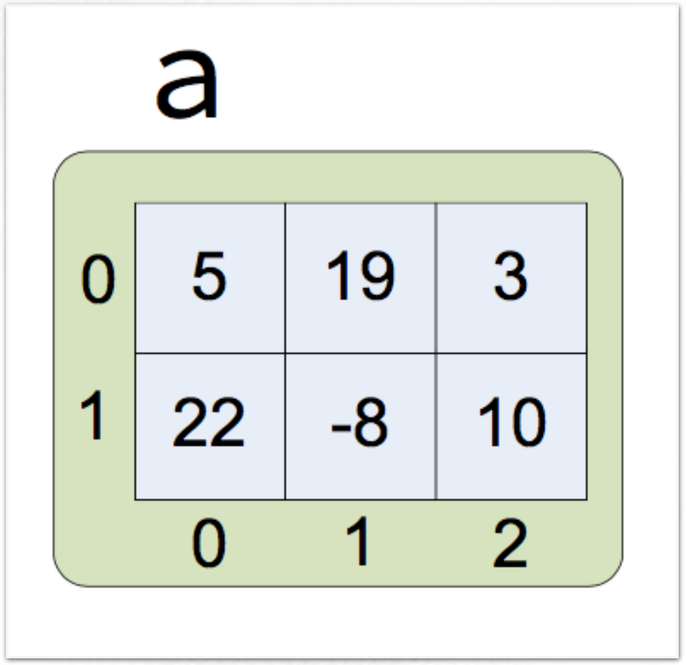
```
std::vector<std::vector<int>> a(2, std::vector<int>(3));
a[0][0] =  5;
a[0][1] = 19;
a[0][2] =  3;
a[1][0] = 22;
a[1][1] = -8;
a[1][2] = 10;
```

Multidimensional Vector manipulation

- Two-dimensional (2D) vector can be understood by:
  - Row
  - Column

```
std::vector<std::vector<int>> a(2, std::vector<int>(3));
```

```
std::vector<std::vector<int>> a{{ 5, 19,  3},
                                {22, -8, 10}};
```



Multidimensional Vector in Loop

The following function prints the contents of a 2D vector of doubles:

```
void print(const std::vector<std::vector<double>>&  m) {
    for (unsigned row = 0; row < m.size(); row++) {
        for (unsigned col = 0; col < m[row].size(); col++)
            std::cout << std::setw(5) << m[row][col];
        std::cout << '\n';
    }
}
```

We can use range-based for statements to simplify the code:

```
void print(const std::vector<std::vector<double>>&  m) {
    for (const std::vector<double>& row : m) {  // For each row
        for (int elem : row)   // For each element in a row
            std::cout << std::setw(5) << elem;
        std::cout << '\n';
    }
    return sum;
}
```

Simplifying Multi-Vector Declaration

```
using Matrix = std::vector<std::vector<double>>;
```

Given the `using` statement defining the new type name `Matrix`, we may express the parameter for the `print` function more simply:

```
void print(const Matrix& m) {
    for (const std::vector<double>& row : m) {  // For each row
        for (int elem : row)    // For each element in a row
            std::cout << std::setw(5) << elem;
        std::cout << '\n';
    }
    return sum;
}
```

We can take advantage of the type inference capability of C++11 to simplify the `print` function even further (see Section 3.10):

```
void print(const Matrix& m) {
    for (auto row : m) {  // For each row
        for (int elem : row)    // For each element in a row
            std::cout << std::setw(5) << elem;
        std::cout << '\n';
    }
    return sum;
}
```

More than 2D

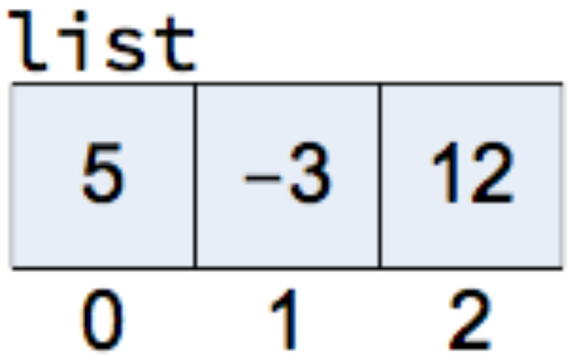
WHY NOT ?

```
matrix[x][y][z][t] = 1.0034;
```

What is Array?

- 40+ years old C language technology
  - Not an object
  - No object functions
  - Directly mapped to the specific memory and its values
  - New declaration grammar
  - Same manipulation grammar
  - Size can be changed (dynamic array) or can't (static array)

```
int list[3];
list[0] = 5;
list[1] = -3;
list[2] = 12;
```



Array declaration by elements & Function (1/2)

Listing 11.11: arraytofunc.cpp

```
#include <iostream>

/*
 *  print(a, n)
 *      Prints the contents of an int array
 *      a is the array to print
 *      n is the size of the array
 */
void print(int a[], int n) {
    for (int i = 0; i < n; i++)
        std::cout << a[i] << " ";
    std::cout << '\n';
}

/*
 *  sum(a, n)
 *      Adds up the contents of an int array
 *      a is the array to sum
 *      n is the size of the array
 */
int sum(int a[], int n) {
```

No size() function,
we should send its size.

Array declaration by elements & Function (2/2)

```
    int result = 0;
    for (int i = 0; i < n; i++)
        result += a[i];
    return result;
}

int main() {
    int list[] = { 2, 4, 6, 8 };
    // Print the contents of the array
    print(list, 4);
    // Compute and display sum
    std::cout << sum(list, 4) << '\n';
    // Zero out all the elements of list
    for (int i = 0; i < 4; i++)
        list[i] = 0;
    // Reprint the contents of the array
    print(list, 4);
    // Compute and display sum
    std::cout << sum(list, 4) << '\n';
}
```

Passing Array is 'Pass by Reference'

Listing 11.12: cleararray.cpp

```
#include <iostream>

/*
 *  print(a, n)
 *     Prints the contents of an int array
 *     a is the array to print
 *     n is the size of the array
 */
void print(int a[], int n) {
    for (int i = 0; i < n; i++)
        std::cout << a[i] << " ";
    std::cout << '\n';
}

/*
 *  clear(a, n)
 *     Makes all the elements of array a zero
 *     a is the array to zero out
 *     n is the size of the array
 */
void clear(int a[], int n) {
    for (int i = 0; i < n; i++)
        a[i] = 0;
}

int main() {
    int list[] = { 2, 4, 6, 8 };
    // Print the contents of the array
    print(list, 4);
    // Zero out the array
    clear(list, 4);
    // Reprint the contents of the array
    print(list, 4);
}
```

Why we use **const** ? (again)

As a general rule, all functions that accept array parameters should declare the array contents as `const` unless they need to modify the elements of the array. For example, use

```
int sum(const int a[], int n)
```

rather than

```
int sum(int a[], int n)
```

if the `sum` function must be able to examine the contents of the array but is not intended to modify its contents.

Pointers and Arrays

Listing 11.13: pointerprint.cpp

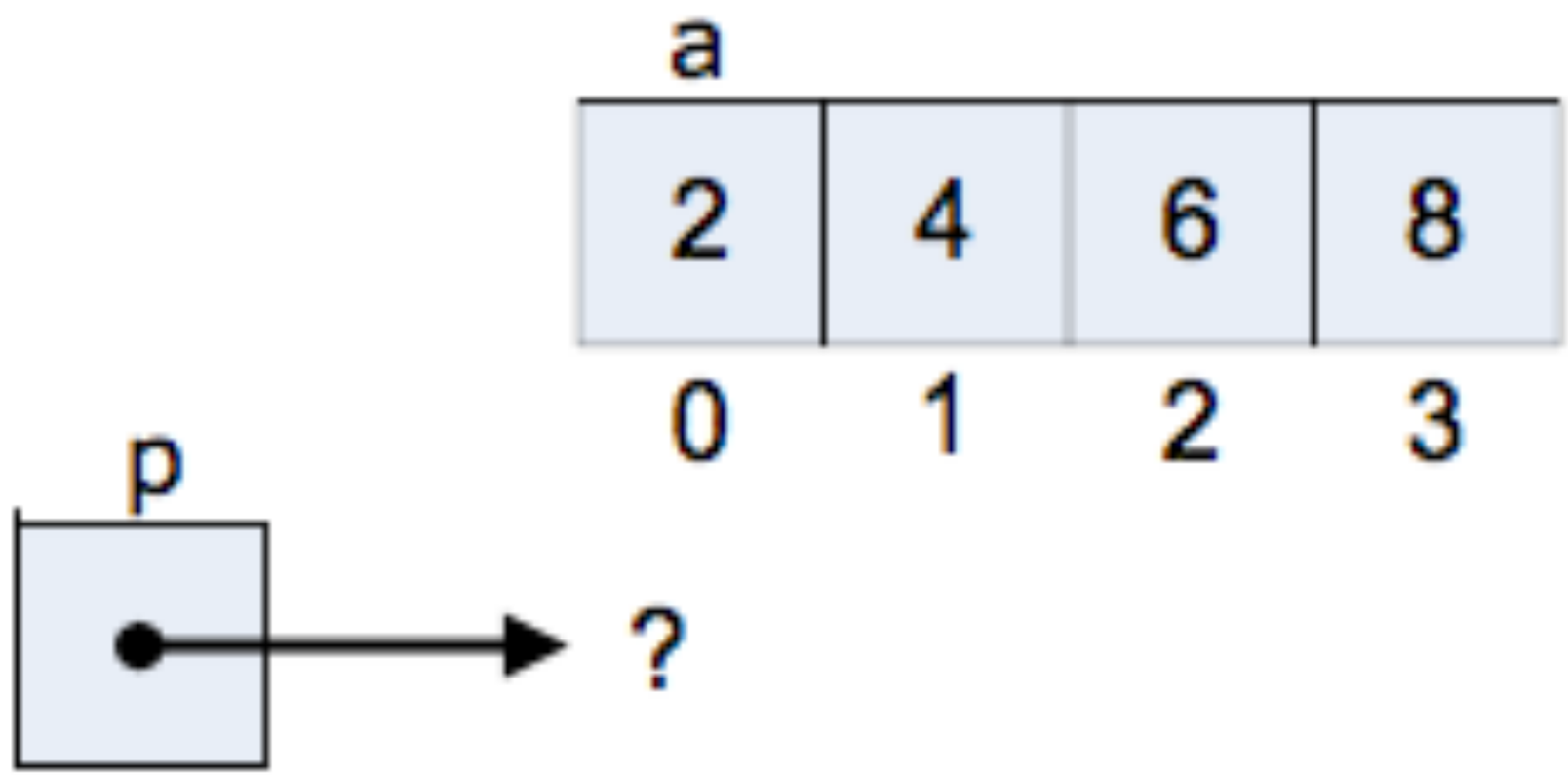
```
#include <iostream>

int main() {
    int a[] = { 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 },
        *p;

    p = &a[0];  // p points to first element of array a

    // Print out the contents of the array
    for (int i = 0; i < 10; i++) {
        std::cout << *p << ' ';  // Print the element p points to
        p++;  // Increment p so it points to the next element
    }
    std::cout << '\n';
}
```

NEXT 'ELEMENT' !!

(don't be confused)

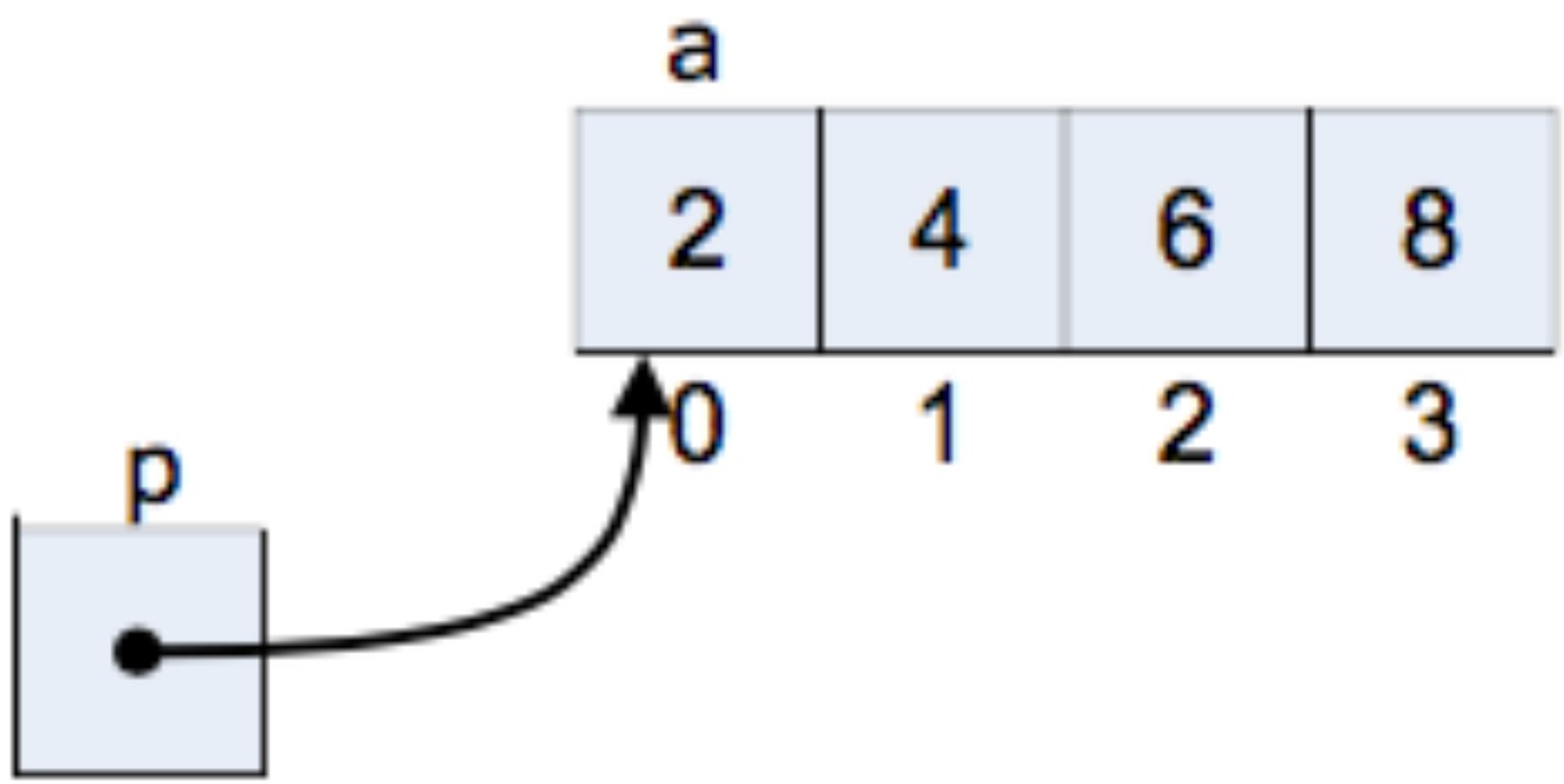
Pointer to Array (1/4)

```
int a[] = { 2, 4, 6, 8 }, *p
```



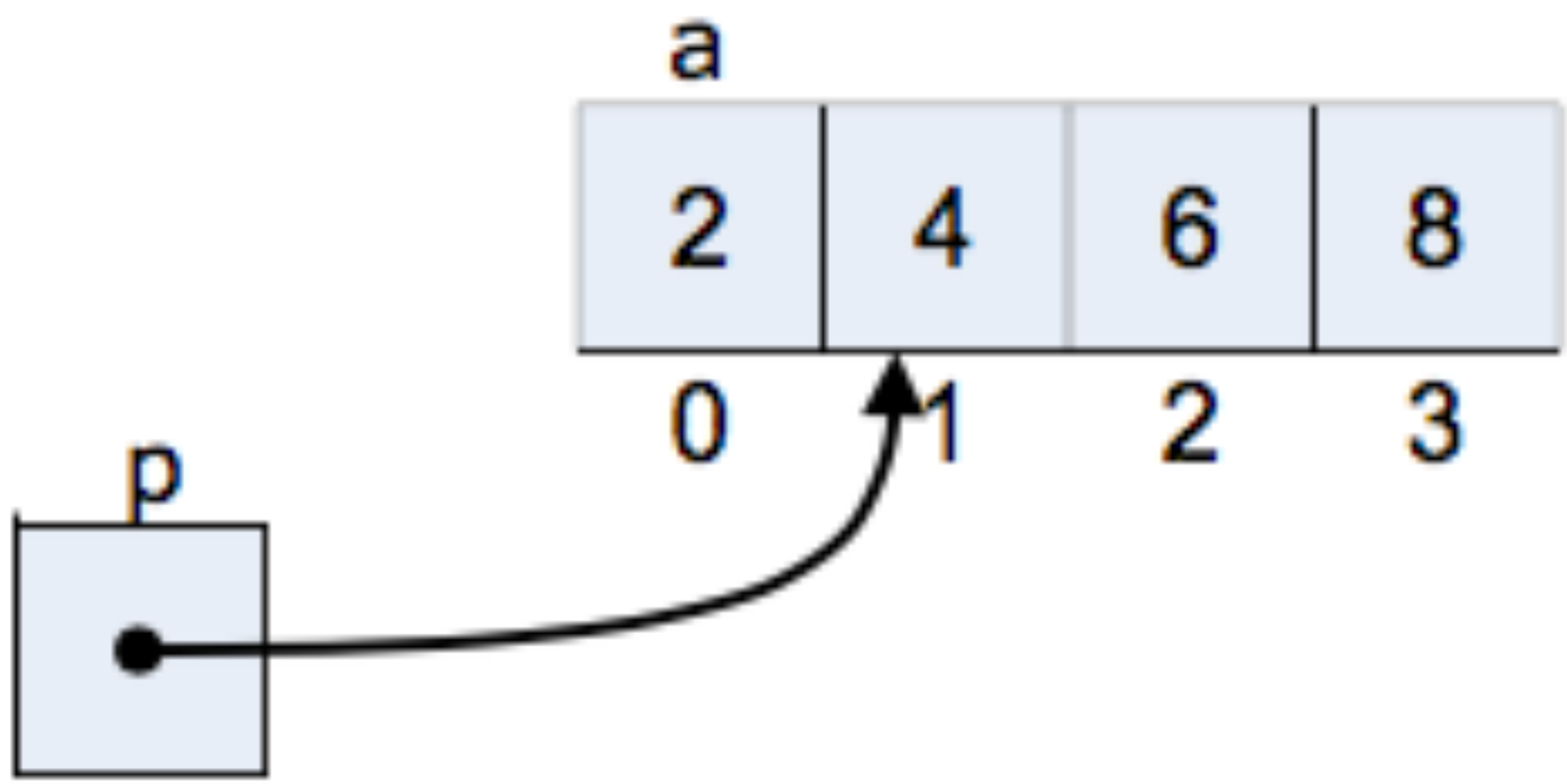
Pointer to Array (2/4)

```
p = a;
```



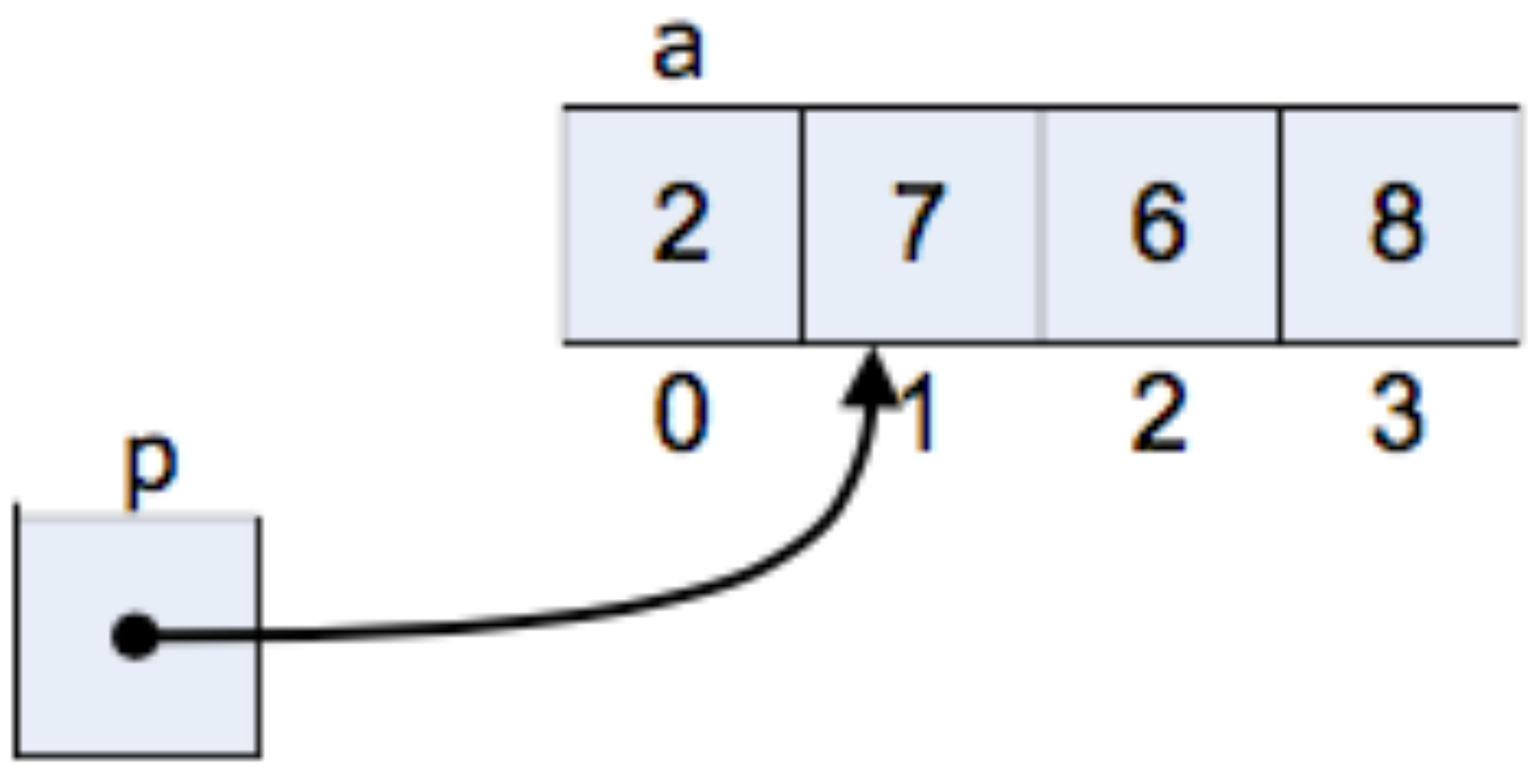
Pointer to Array (3/4)

```
p++;
```



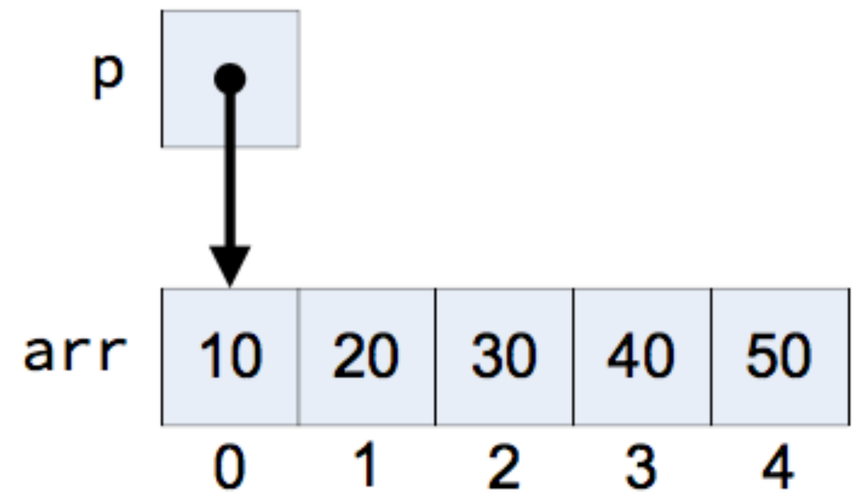
Pointer to Array (4/4)

```
*p = 7;
```

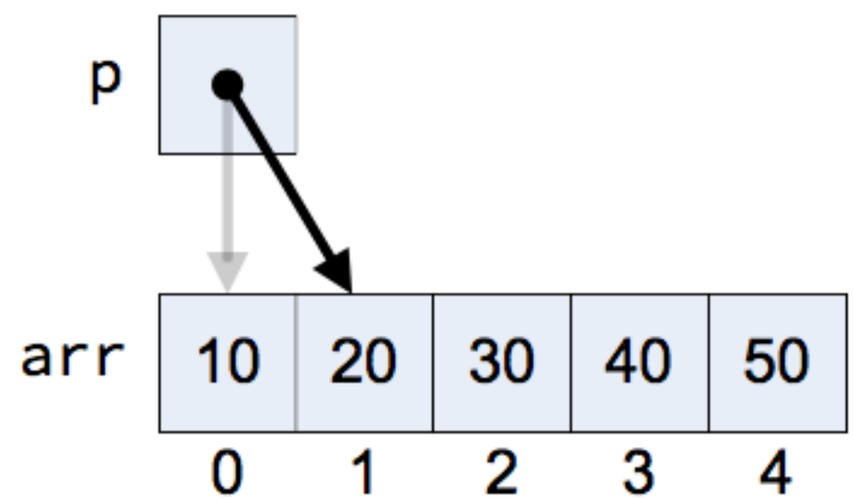


Pointer to Array (again)

```
int arr[] = {10, 20, 30, 40, 50};
int *p = arr;
```




```
p++;
```



Pointer to Array Example

Listing 11.14: pointerprint2.cpp

```
#include <iostream>

int main() {
    int a[] = { 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 },
        *begin, *end, *cursor;

    begin = a;       // begin points to the first element of array a
    end = a + 10;    // end points to just after the last element

    // Print out the contents of the array
    cursor = begin;
    while (cursor != end) {
        std::cout << *cursor << ' ';  // Print the element
        cursor++;  // Increment cursor so it points to the next element
    }
    std::cout << '\n';
}
```

Different look, but Same meaning

Sometimes pointer notation is used to represent an array parameter to a function. The array print function that begins

```
void print(const int a[], int n)
```

could instead be written

```
void print(const int *a, int n)
```

where a is a pointer to an array. The compiler treats the two forms identically in the machine language it produces.

Dynamic Array (with *new* & *delete* statement)

- Static array size is fixed
- Dynamic array size can be dynamically allocated and deleted
  - new is used for dynamic memory allocation

```
double *numbers;
numbers = new double[size]
```

  - delete is used for dynamic memory release

```
delete[] numbers;
```

> You should develop the habit of ensuring that every use of the new operator has an associated call of delete. If the delete statement is omitted from Listing 11.18 (flexiblearrayaverage.cpp), the program in this case likely will behave no differently since most operating systems reclaim all the dynamic memory a program holds when the program finishes executing. Serious problems arise in longer running programs that must allocate and deallocate dynamic memory over time. Programs that run for extended periods of time, like web servers and operating system kernels, can crash due to failures to properly deallocate memory. The condition is known as a *memory leak*—the program overtime allocates more and more memory via new but never releases the memory back with delete. Eventually the program uses up all of the available heap space and crashes.

Dynamic Array by Example

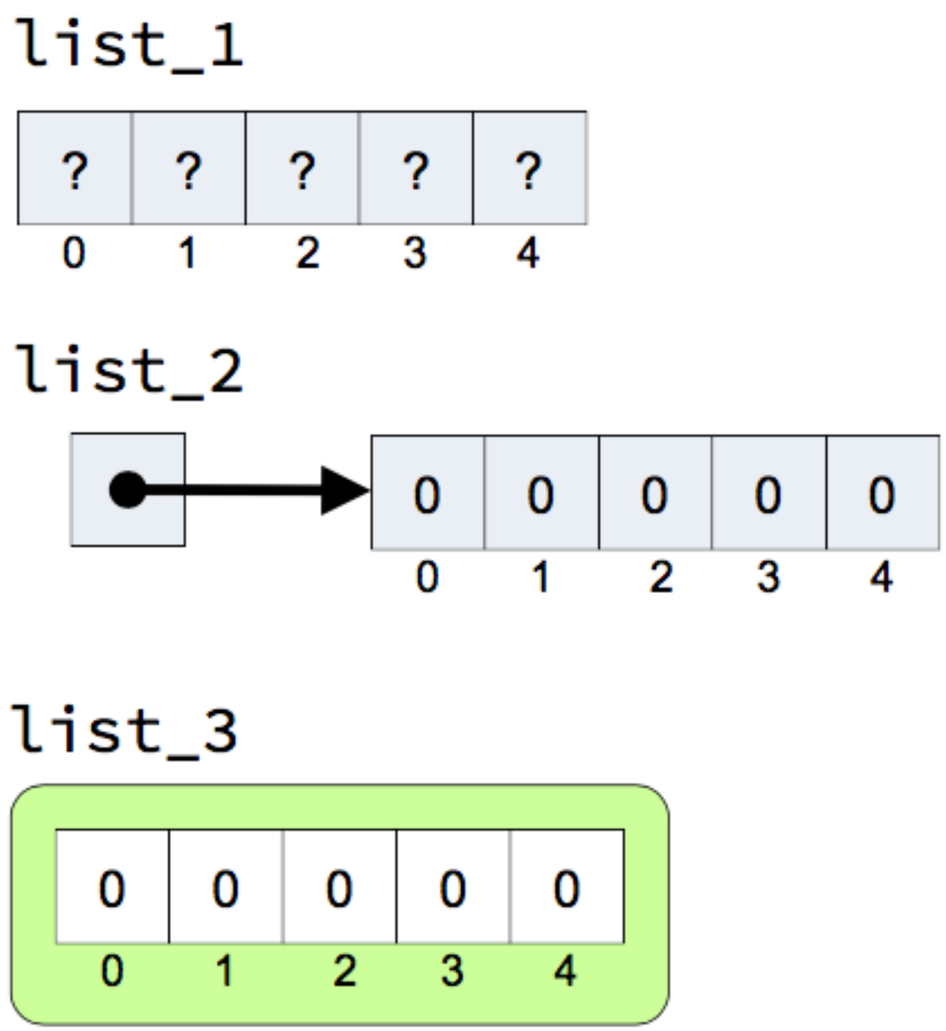
Listing 11.18: flexiblearrayaverage.cpp

```
#include <iostream>

int main() {
    double sum = 0.0;
    double *numbers;  // Note: numbers is a pointer, not an array
    int size;  // Actual number of entries

    // Get effective size of the array
    std::cout << "Please enter number of values to process: ";
    std::cin >> size;

    if (size > 0) {  // Nothing to do with no entries
        std::cout << "Please enter " << size << " numbers: ";
        // Allocate the exact size needed
        numbers = new double[size];  // Dynamically allocated array
        // Allow the user to enter in the values.
        for (int i = 0;  i < size;  i++) {
            std::cin >> numbers[i];
            sum += numbers[i];
        }
        std::cout << "The average of ";
        for (int i = 0;  i < size - 1;  i++)
            std::cin >> numbers[i] << ", ";
        // No comma following last element
        std::cout << numbers[size - 1] << " is "
                  << sum/size << '\n';
        delete [] numbers;  // Free up the space held by numbers
    }
}
```

[] can be used with pointer

Vector vs. Dynamic Array vs. Static Array

```
int list_1[5];
int *list_2 = new int[5];
vector<int> list_3(5);
```



Multidimensional Array by Example (1/2)

Listing 11.20: twodimarray.cpp

```
#include <iostream>
#include <iomanip>

const int ROWS = 3,
          COLUMNS = 5;

// The name Matrix represents a new type
// that means a ROWS x COLUMNS
// two-dimensional array of double-precision
// floating-point numbers.
using Matrix = double[ROWS][COLUMNS];

// Allow the user to enter the elements of a matrix
void populate_matrix(Matrix m) {
    std::cout << "Enter the " << ROWS << " rows of the matrix.\n";
    for (int row = 0; row < ROWS; row++) {
        std::cout << "Row #" << row << " (enter " << COLUMNS << " values):";
        for (int col = 0; col < COLUMNS; col++)
            std::cin >> m[row][col];
    }
}
```

Multidimensional Array by Example (2/2)

```
// We declare m constant because printing a matrix should not
// change it.
void print_matrix(const Matrix m) {
    for (int row = 0; row < ROWS; row++) {
        for (int col = 0; col < COLUMNS; col++)
            std::cout << std::setw(5) << m[row][col];
        std::cout << '\n';
    }
}

int main() {
    // Declare the 2D array
    Matrix mat;
    // Populate the array
    populate_matrix(mat);
    // Print the array
    print_matrix(mat);
}
```

C String and Pointer to Array

Listing 11.21: findchar.cpp

```
#include <iostream>

bool find_char(const char *s, char ch) {
    while (*s != '\0')  { // Scan until we see the null character
        if (*s == ch)
            return true;   // Found the matching character
        s++;  // Advance to the next position within the string
    }
    return false;   // Not found
}

int main() {
    const char *phrase = "this is a phrase";
    // Try all the characters a through z
    for (char ch = 'a'; ch <= 'z'; ch++) {
        std::cout << '\'' << ch << '\''<< " is ";
        if (!find_char(phrase, ch))
            std::cout << "NOT ";
        std::cout << "in " << '\"' << phrase << '\"' << '\n';
    }
}
```

Notice

Programmers must manage primitive arrays and associated pointers very carefully **because array and pointer misuse is a common source of difficult to find and repair bugs within programs**. Vectors effectively take care of much of the memory management problems that plague raw arrays. Additionally, vectors provide functionality and convenience that arrays cannot. As we have seen, it is easy to adapt a vector to a context that requires an array.

Object Oriented Programming by C++

Sungwon Lee / Professor

Email: drsungwon@khu.ac.kr

Web: http://mobilelab.khu.ac.kr/