Object-Oriented Programming
(in C++)

Arrays and Vectors

Professor Yi-Ping You (游逸平)
Department of Computer Science
http://www.cs.nctu.edu.tw/~ypyou/

Outline

- Introduction to Arrays
- Pointer-based Arrays
  - Declaring, Initializing, and Using Arrays
  - Passing Arrays to Functions
  - Multidimensional Arrays
- Object-based Arrays: `vector`
- Character Arrays
- Standard Library Class `string`

Array

- A collection of related data items of the same type
  - Structure or class: a collection of related data items of possibly different types
- Why do we need arrays?
  - Imagine keeping track of 10, or 100, or 1000 cars in memory
    - How would you name all the variables?
    - How would you process each of the variables?

```
int c1, c2, ..., c10;
```

- Two types of arrays
  - Pointer-based arrays (C-style)
  - Object-based arrays (`vector` from STL)

Outline

- Introduction to Arrays
- Pointer-based Arrays
  - Declaring, Initializing, and Using Arrays
  - Passing Arrays to Functions
  - Multidimensional Arrays
- Object-based Arrays: `vector`
- Character Arrays
- Standard Library Class `string`

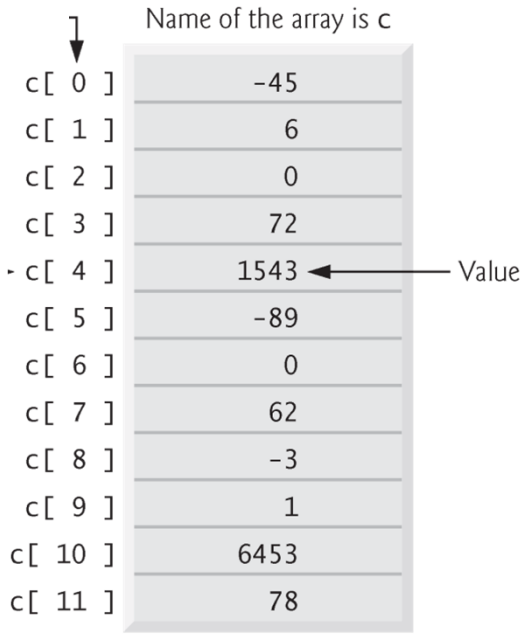
Declaring An Array (Pointer-based)

- An array is a consecutive group of memory locations
- Declaring an array:

An integer constant greater than zero

```
type arrayName[ arraySize ];
```

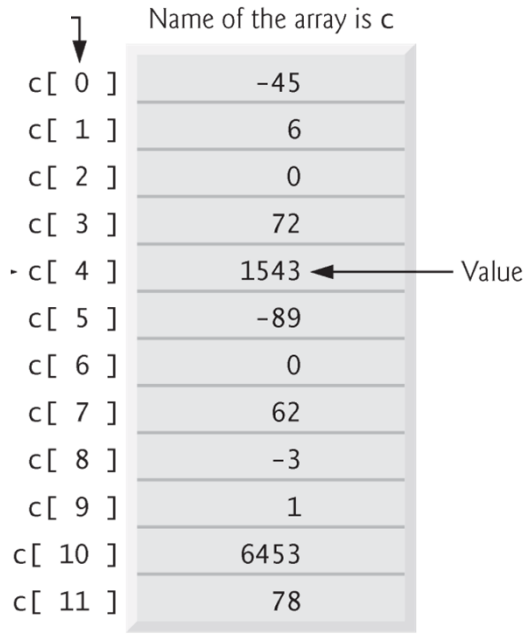
```
int c[ 12 ];
const int s = 10;
int a[ s ];
```



Array of 12 elements

Referencing An Array Element

- Use the array name and the position number of the particular element to specify the element
  - Position number is called subscript or index
    - The first element has subscript 0
    - The subscript must be integer or integer expression, and its value must be larger than or equal to 0
  - A subscripted array name is an lvalue
    - The lvalue of a variable is its address
    - The rvalue of a variable is its value



Array of 12 elements

Array Bounds Not Checked

- C/C++ has no array bounds checking
  - Referring to an element outside the array bounds is an execution-time logic error, not a syntax error
    - Undefined behavior
    - C++ simply exposes raw memory
    - A segmentation fault will occur if a memory address that your program doesn't have permission to access is accessed
- Bounds-checking is possible on class types
  - Will be discussed in the later lecture (Chapter 11)
- C++ offers the `std::vector` class template that provides bounds-check access when you want it (discussed later in this lecture)

Initializing An Array (1/3)

- Using a loop to initialize the array's elements
  - Declare array, specify number of elements
  - Use repetition statement to loop for each element
- Example

```
int n[10];

for ( int i = 0; i < 10; i++) {
  n[ i ] = 0;
}
```

Initializing An Array (2/3)

- With an initializer list
  - Initializer list
    - Items enclosed in braces (**{}**) and separated by commas
  - Examples

```
int n[5] = {10,20,30,40,50};
int m[5] = {10,20,30,40,50,60}; // error
```

```
int m[5] = {10,20,30};
```

    - The remaining elements are initialized to **zero**

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

    - Because array size is omitted in the declaration, the compiler determines the size of the array based on the size of the initializer list

Initializing An Array (3/3)

- Read the value for each element one by one from the keyboard
- Example

```
const int arraySize  = 5;
int array[arraySize];
for (int i = 0; i < arraySize; i++) {
  cout << "Please specify a value for element no."
       << i << endl;
  cin >> array[i];
}
```

Print An Array

- Use a for loop to access each element of the array and print one by one
- Example

```
const int arraySize = 5;
int array[arraySize] = { 1, 2, 4, 67, 3 };

for (int i = 0; i < arraySize; i++) {
   cout << array[i] << endl;
}
```

Passing Arrays to Functions (1/2)

- Functions that take arrays as arguments:

```
void modifyArray (int array [ ], int size);
```

- To pass an array argument to a function

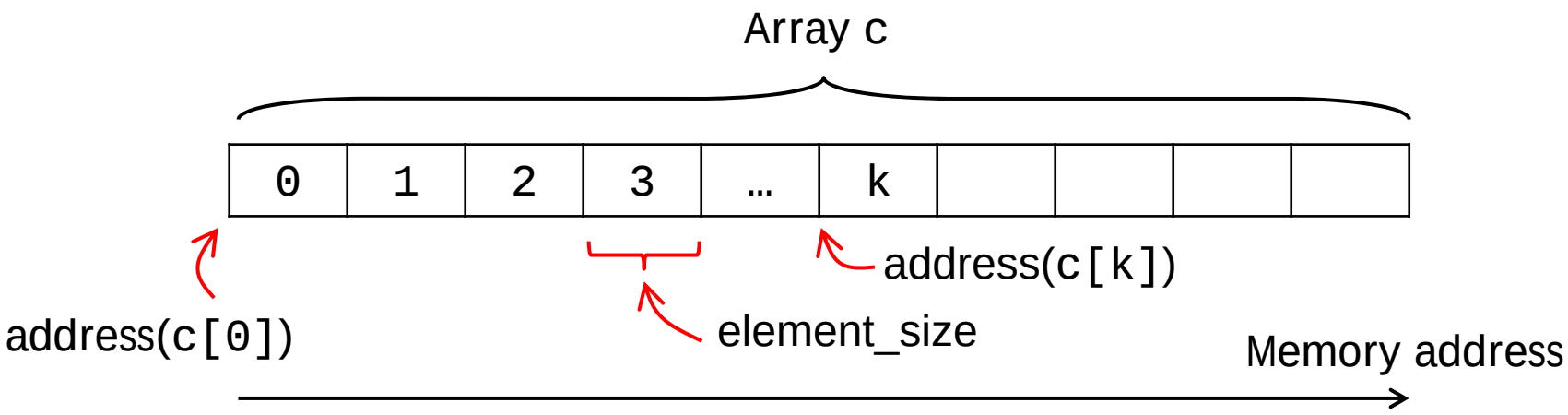
```
int hourlyTemperatures[ 24 ];
modifyArray( hourlyTemperatures, 24 );
```

- Array size is normally passed as another argument, so the function can process the specific number of elements in the array
  - Every `vector` object "knows" its own size

Passing Arrays to Functions (2/2)

- Arrays are passed by reference
  - Function call actually passes starting address of array
    - So function knows where array is located in memory
- Individual array elements are passed by value
- Supplement material
  - address(c[k]) = address (c[0]) + ((k-0) * element_size)

Defined by the type of the element

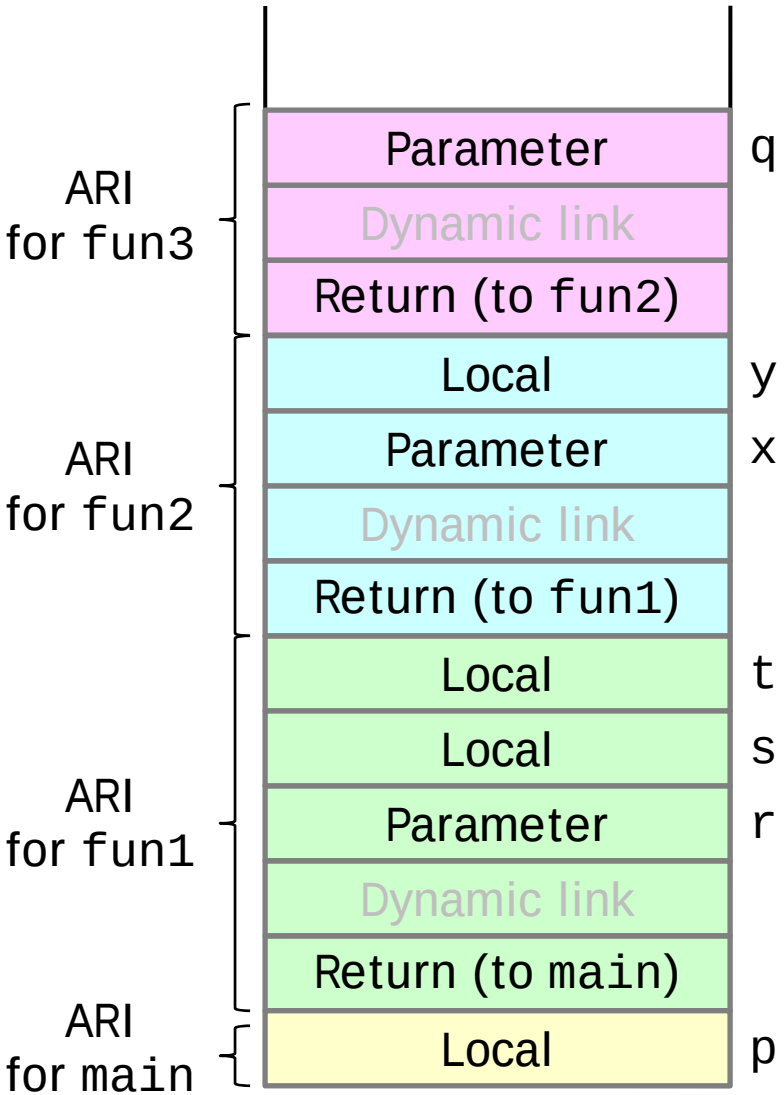


Why Passing Arrays by Reference?

- Passing by value would require copying each element to the activation record
- For large, frequently passed arrays, this is time consuming and require considerable storage for the copies of the array element

Supplement: How A Function Call Works

```
void fun1(float r) {
  int s, t;
  ...
  fun2(s);
  ...
}
void fun2(int x) {
  int y;
  ...
  fun3(y);
  ...
}
void fun3(int q) {
  ...
}
int main() {
  float p;
  ...
  fun1(p);
  ...
}
```



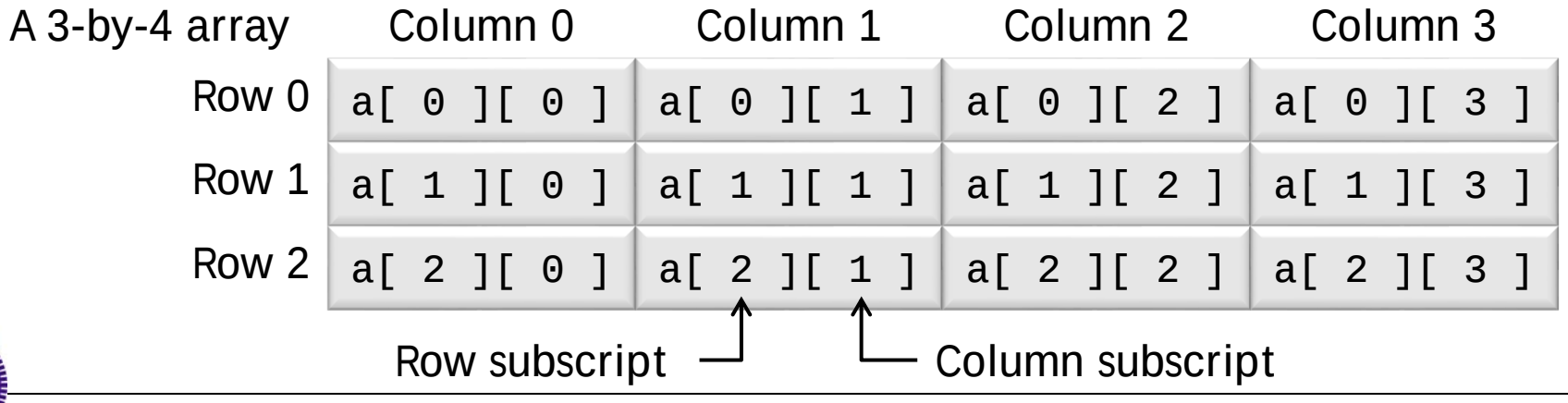
const Array Parameter

- Prevent modification of array values in the caller by code in the called function
- Elements in the array are constant in the function body
- Enables programmers to prevent accidental modification of data
  - The compiler will give error messages
- Using the qualifier const
- Example:

```
void funcOne(const int [ ]);
int a[] = {10,20,30};
funcOne(a);
```

Multidimensional Array

- Arrays with two or more dimensions are known as multidimensional arrays
- Multidimensional arrays with two dimensions
  - Called two dimensional or 2-D arrays
  - Represent tables of values with rows and columns
  - Elements referenced with two subscripts (`[x][y]`)
  - In general, an array with *m* rows and *n* columns is called an *m*-by-*n* array

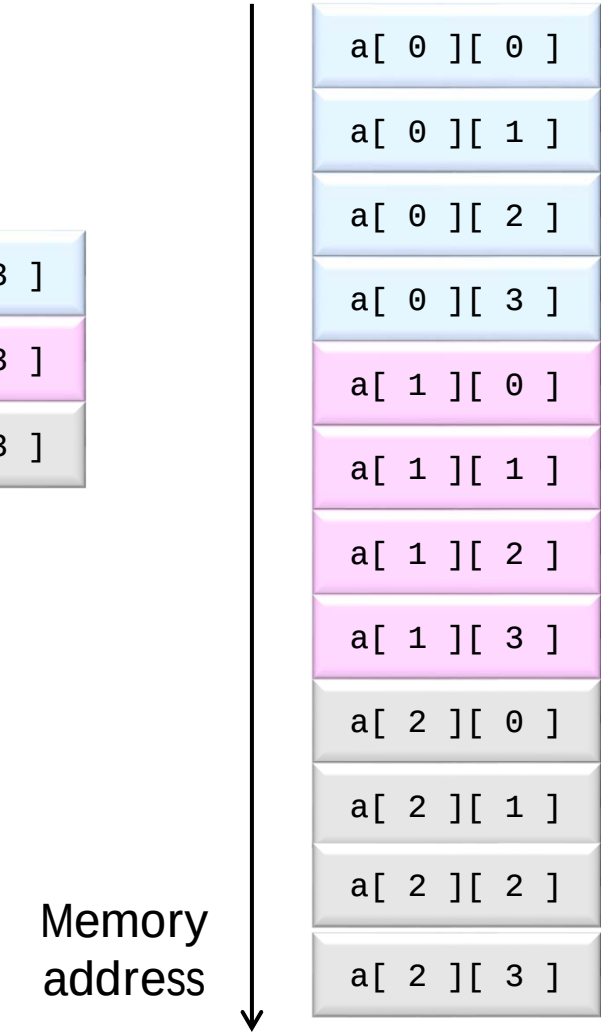


Row-Major-Order Arrays in C/C++

- Array elements are stored in the memory row by row

a[0][0]	a[0][1]	a[0][2]	a[0][3]
a[1][0]	a[1][1]	a[1][2]	a[1][3]
a[2][0]	a[2][1]	a[2][2]	a[2][3]

A 2-D array



Declaring & Initializing 2-D Arrays

- Declaring two-dimensional array b

```
int b[ 3 ][ 2 ] = { { 1, 2 }, { 3, 4 } };
```

  - 1 and 2 initialize b[0][0] and b[0][1]
  - 3 and 4 initialize b[1][0] and b[1][1]
  - 0 and 0 initialize b[2][0] and b[2][1] (implicitly)

	Column 0	Column 1
Row 0	1	2
Row 1	3	4
Row 2	0	0

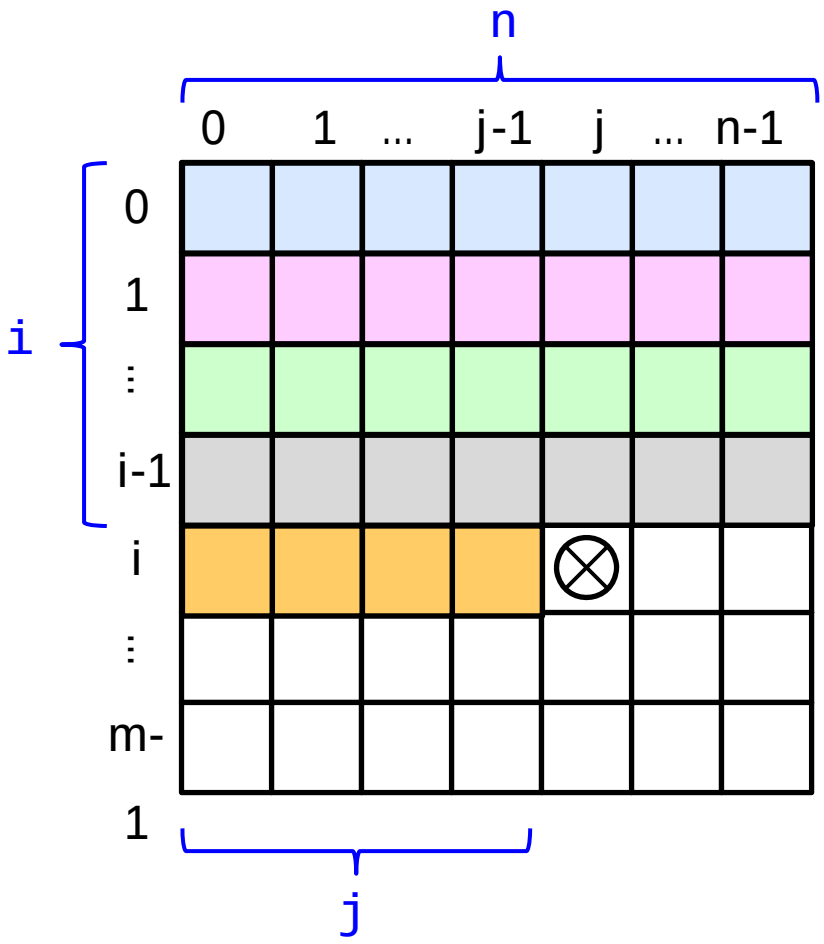
Passing Multidimensional Arrays to Functions

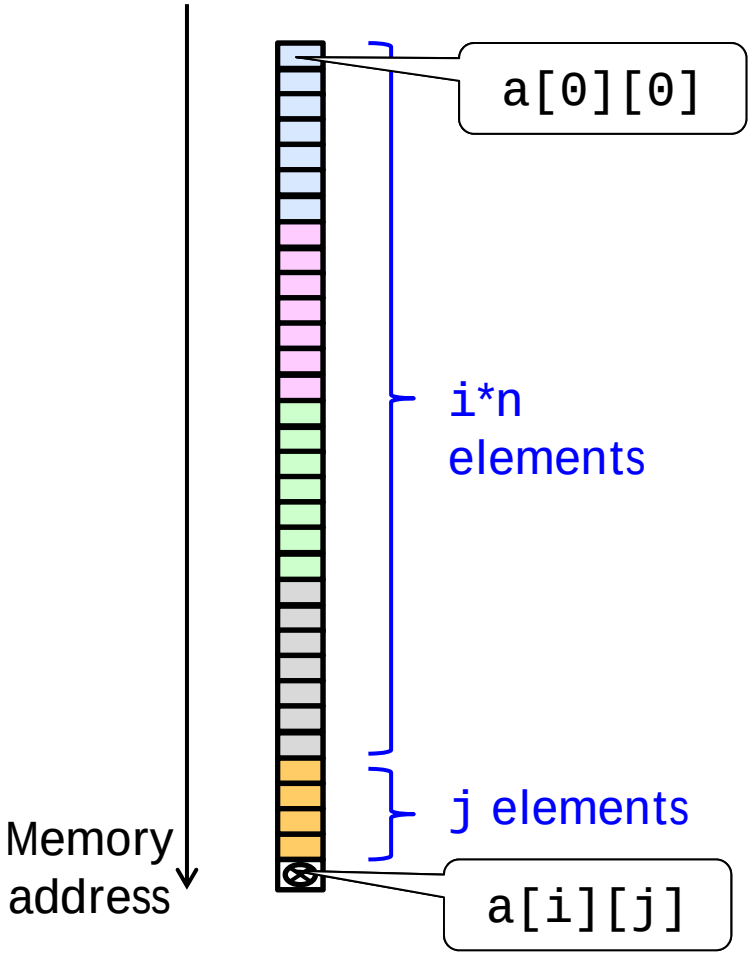
- Multidimensional array parameters
  - Size of first dimension is not required
    - As with a one-dimensional array
  - Size of subsequent dimensions are **required**
    - Compiler must know how many elements to skip to move to the second element in the first dimension
  - Example

```
void printArray( const int a[][ 3 ] );
```

Supplement: Locating an Element in a 2-D Array

- address(a[i][j])
  = address(a[0][0]) + ((i * n) + j) * element_size





What are the Array's Limitations?

- Array names are simply const pointers!
  - We can't meaningfully compare two arrays for equality
  - We can't assign one array to another, but must manually iterate over elements (if they are even the same size!)
- Statically allocated, so it can't grow with the data
- From the array itself, we don't know its size
  - It has to be tracked by the programmer

Outline

- Introduction to Arrays
- Pointer-based Arrays
  - Declaring, Initializing, and Using Arrays
  - Passing Arrays to Functions
  - Multidimensional Arrays
- Object-based Arrays: `vector`
- Character Arrays
- Standard Library Class `string`

`vector`: Object-based Array

- C-style pointer-based arrays have great potential for errors and are not flexible
  - No bounds check
  - Equality and relational operators do not apply to arrays
  - Array size is tracked by the programmer
- Class template `vector` (from C++ Standard Library) allows you to create a more powerful and less error-prone alternative to arrays
  - Standard class template `vector` is defined in header `<vector>` and belongs to namespace `std`

A Preview to Classes

- What is a class?
  - A user-defined, abstract data type
  - `struct` in C is also a user-defined data type, but `struct` has no operations

```
struct myStruct {
   int field1;
   int field2;
};

int main() {
   struct myStruct var;
   var.field1 = 10;
   var.field2 = 20;
}
```

```
class myClass {
   int member1;
   int member2;
   int func() { ... }
   void myClass() { ... }
};

int main() {
   myClass var;
   var.member1 = 10;
   var.member2 = 20;
   var.func();
}
```

An object

vector Class

```
namespace std {

  template <class T>
  class vector<T> {
    // creates an empty vector
    void vector() { ... }

    //creates vector of n default values
    void vector(int n) { ... }

    // return the number of elements
    size_t size() { return size }

    T *data;
    unsigned int size;
    ...
  };
  ...
}
```

Using `vector` (1/2)

- By default, all the elements of a `vector` object are set to 0
- `vectors` can be defined to store any data type
- `vector` member function `size` obtain the number of elements in the vector
- You can use square brackets, [], to access the elements in a vector
- `vector` objects can be compared with one another using the equality operators
- You can create a new `vector` object that is initialized with a copy of an existing `vector`

Using `vector` (2/2)

- You can use the assignment (=) operator with `vector` objects
- As with C-style pointer-based arrays, C++ does not perform any bounds checking when vector elements are accessed with square brackets
- Standard class template vector provides bounds checking in its member function `at`, which "throws an exception" (discussed in Chapter 16, Exception Handling) if its argument is an invalid subscript

Using `vector`: An Example

```
#include <vector>
using std::vector;

int main() {
  // creates an object-based array of five integers
  // which are initialized to zero
  vector<int> integers1(5);

  // integers1.size() returns the number of elements
  cout << integers1.size() << endl;

  integers1[3] = 89;    // no bounds checking
  integers1.at(3) = 89; // bounds checking

  // creates a vector by using integers1
  vector<int> integers2(integers1);

  return 0;
}
```

Using `vector` and `iterator`: An Example

```
#include <iostream>
#include <vector>
using std::vector;

int main() {
  vector<int> integers1(5);
  vector<int>::iterator it; // pointer into vector

  for(it = integers1.begin();
      it != integers1.end(); it++) {
    std::cout << ++(*it) << " ";
  }
  std::cout << std::endl;

  return 0;
}
```

```
1 1 1 1 1
```

Nested Type Name

- § 9.9, C++ Standard
  - Some call it a “member type”

```
class C {
  typedef int int_type; // as a nested typedef-name
  class inner_type { ... }; // as a nested class or struct
};

int_type       a1; // error, 'int_type' not known
C::int_type    a2; // OK
C::inner_type a3; // OK
```

- `iterator` in the previous slide is a nested type name in class template `vector<int>`

Outline

- Introduction to Arrays
- Pointer-based Arrays
  - Declaring, Initializing, and Using Arrays
  - Passing Arrays to Functions
  - Multidimensional Arrays
- Object-based Arrays: `vector`
- Character Arrays
- Standard Library Class `string`

Character Arrays (1/2)

- Using character arrays to store and manipulate strings
  - Arrays may be of any type, including `char`s
    - An array of characters:

```
char charArrays[] = {'H', 'i'};
```

  - We can store character strings in `char` arrays
  - Can be initialized using a string literal
    - Example

```
char string1[] = "Hi";
```

Equivalent to

```
char string1[] = { 'H', 'i', '\0' };
```

  - Array contains each character plus a special **string-termination** character called **the null character (`'\0'`)**

Character Arrays (2/2)

- Using character arrays to store and manipulate strings
  - Can also input a string directly into a character array from the keyboard using `cin` and >>

```
cin >> string1;
```

    - `cin` reads characters from the keyboard until the first white-space character is encountered – regardless of the array size
    - `cin >>` may read more characters than the array can store
  - A character array representing a null-terminated string can be output with `cout` and <<

```
cout << string1;
```

Outline

- Introduction to Arrays
- Pointer-based Arrays
  - Declaring, Initializing, and Using Arrays
  - Passing Arrays to Functions
  - Multidimensional Arrays
- Object-based Arrays: `vector`
- Character Arrays
- Standard Library Class `string`

Standard Library Class `string`

- Class built into C++
  - Available for anyone to use
  - Class `string`
    - Header **`<string>`**, namespace **`std`**
    - Example:

```
#include <string>
using std::string;
```

  - Character Arrays: strings
    - Header **`<cstring>`** (**`<string.h>`**)

Standard Library Class `string` (Cont.)

- Class `string`
  - Can initialize `string s1( "hi" );`
  - Output the string: `cout << s1;`
  - Take keyboard input: `cin >> s1;`
    - `cin` reads characters from the keyboard until the first white-space character is encountered
  - Read a line of text into a string

```
string s2; // s2 is empty
getline( cin, s2);
```

  - Assignment operator (=)
  - Concatenation (+) (Chapter 18) (Chapter 11, Operator Overloading)

Relational Operators

- Relational operators (Chapter 18)
  - ==, !=, >=, >, <=, <
  - Example : `if (s1 >= s2)`
- Compares string `s1` to string `s2`
  - Starts comparing the first character of each string. If they are equal to each other, it continues with the following pairs until the characters differ.
  - Comparing ASCII codes (ranging from 0-127, p. 936) of two characters
    - `'a' < 'z'` (97 < 122)
    - `'A' < 'a'` (65 < 97)

```
string foo = "alpha";
string bar = "beta";
if (foo < bar) cout << "foo is less than bar\n";
```

```
foo is less than bar
```

Substring Member Function `substr`

- `substr(int startIndex, int length)`
  - `s1.substr( 0, 14 );`
    - Starts at location 0, gets 14 characters
- `substr(int startIndex)`
  - `s1.substr( 15 );`
    - Substring beginning at location 15, to the end

More Functionalities

- operator `[]`
  - Access one character
  - No range checking (if subscript invalid)
  - Example: `char a = s1[0];`
- Member function `at`
  - Accesses one character
    - `Example: char b = s1.at( 10 );`
  - Has bounds checking, throws an exception if subscript is invalid

Empty String

- Member function `empty` tests whether the string is empty
  - Example:

```
if ( s1.empty() ) {
   ...
}
```

length & size

- `length()` and `size()` returns the same value
  - The number of characters in the string
  - Example:

```
string s1= “1234567” ;
int size = s1.size();
int length = s1.length();
```

Using `string`: An Example (1/5)

```
1  // Fig. 6.21: fig06_21.cpp
2  // Standard Library class string.
3  #include <iostream>
4  #include <string>
5  using namespace std;
6
7  int main()
8  {
9     string s1( "happy" ); // string object s1 initialized to "happy"
10    string s2( " birthday" ); // s2 initialized to " birthday"
11    string s3; // s3 is empty
12    string s4; // s4 is empty
13
14    // read a line of text into a string object
15    cout << "Enter a line of text: ";
16    getline( cin, s4 ); // read line of text into s4
17
18    // display each string's contents
19    cout << "s1 is \"" << s1 << "\"; s2 is \"" << s2
20       << "\"; s3 is \"" << s3 << "\"; s4 is \"" << s4 << "\"\n\n";
21
22    // determine the length of each string
23    cout << "s1 length " << s1.length() << "; s2 length " << s2.length()
24       << "; s3 length " << s3.length() << "; s4 length " << s4.length();
```

```
Enter a line of text: Using class string
s1 is "happy"; s2 is " birthday"; s3 is ""; s4 is "Using class string"
```

```
s1 length 5; s2 length 9; s3 length 0; s4 length 18
```

Using `string`: An Example (2/5)

```
25
26    // test equality and relational operators
27    cout << "\n\nThe results of comparing s2 and s1:"
28       << "\ns2 == s1 yields " << ( s2 == s1 ? "true" : "false" )
29       << "\ns2 != s1 yields " << ( s2 != s1 ? "true" : "false" )
30       << "\ns2 >  s1 yields " << ( s2 > s1 ? "true" : "false" )
31       << "\ns2 <  s1 yields " << ( s2 < s1 ? "true" : "false" )
32       << "\ns2 >= s1 yields " << ( s2 >= s1 ? "true" : "false" )
33       << "\ns2 <= s1 yields " << ( s2 <= s1 ? "true" : "false" );
34
35    // test string member function empty
36    cout << "\n\nTesting s3.empty():" << endl;
37
38    if ( s3.empty() )
39    {
40       cout << "s3 is empty; assigning s1 to s3;" << endl;
41       s3 = s1; // assign s1 to s3
42       cout << "s3 is \"" << s3 << "\"";
43    } // end if
44
45    // test string concatenation operator
46    cout << "\n\nAfter s1 += s2, s1 is ";
47    s1 += s2; // concatenate s2 to the end of s1
48    cout << s1;
```

```
The results of comparing s2 and s1:
s2 == s1 yields false
s2 != s1 yields true
s2 >  s1 yields false
s2 <  s1 yields true
s2 >= s1 yields false
s2 <= s1 yields true
```

```
Testing s3.empty():
s3 is empty; assigning s1 to s3;
s3 is "happy"
```

```
After s1 += s2, s1 is happy birthday
```

Using `string`: An Example (3/5)

```
49
50      // test string concatenation operator with string literal
51      cout << "\n\ns1 += \" to you\" yields" << endl;
52      s1 += " to you";
53      cout << "s1 is " << s1 << "\n\n";
54
55      // test string member function substr
56      cout << "The substring of s1 starting at location 0 for\n"
57         << "14 characters, s1.substr(0, 14), is:\n"
58         << s1.substr( 0, 14 ) << "\n\n"; // di
59
60      // test substr "to-end-of-string" option
61      cout << "The substring of s1 starting at\n"
62         << "location 15, s1.substr(15), is:\n"
63         << s1.substr( 15 ) << endl; // displays "to you"
64
65      // making a copy of a string
66      string s5( s1 ); // creates s5 as a copy of s1
67      cout << "\ns5 is " << s5;
68
69      // test the subscript operator to create lvalues
70      s1[ 0 ] = 'H'; // replaces h with H
71      s1[ 6 ] = 'B'; // replaces b with B
72      cout << "\n\ns1 after s1[0] = 'H' and s1[6] = 'B' is: " << s1;
```

```
s1 += " to you" yields
s1 is happy birthday to you
```

```
The substring of s1 starting at location 0 for
14 characters, s1.substr(0, 14), is:
happy birthday
```

```
The substring of s1 starting at
location 15, s1.substr(15), is:
to you
```

```
s5 is happy birthday to you
```

```
s1 after s1[0] = 'H' and s1[6] = 'B' is: Happy Birthday to you
```

Using `string`: An Example (4/5)

```
73
74      // test the subscript operator to create rvalues
75      cout << "\n\ns1[0] is " << s1[ 0 ] << "; s1[2] is " << s1 [ 2 ]
76         << "; s1[s1.length()-1] is " << s1[ s1.length() - 1 ];
77
78      // test subscript out of range with string member function "at"
79      cout << "\n\nAttempt to assign 'd' to s1.at( 30 ) yields:" << endl;
80      s1.at( 30 ) = 'd'; // ERROR: subscript out of range
81   } // end main
```

```
s1[0] is H; s1[2] is p; s1[s1.length()-1] is u
```

```
Attempt to assign 'd' to s1.at( 30 ) yields:
Platform specific error message will be dislayed
```

```
Enter a line of text: Using class string
s1 is "happy"; s2 is " birthday"; s3 is ""; s4 is "Using class string"

s1 length 5; s2 length 9; s3 length 0; s4 length 18

The results of comparing s2 and s1:
s2 == s1 yields false
s2 != s1 yields true
s2 >  s1 yields false
s2 <  s1 yields true
s2 >= s1 yields false
s2 <= s1 yields true
```

Using `string`: An Example (5/5)

```
Testing s3.empty():
s3 is empty; assigning s1 to s3;
s3 is "happy"

After s1 += s2, s1 is happy birthday

s1 += " to you" yields
s1 is happy birthday to you

The substring of s1 starting at location 0 for
14 characters, s1.substr(0, 14), is:
happy birthday

The substring of s1 starting at
location 15, s1.substr(15), is:
to you

s5 is happy birthday to you

s1 after s1[0] = 'H' and s1[6] = 'B' is: Happy Birthday to you

s1[0] is H; s1[2] is p; s1[s1.length()-1] is u

Attempt to assign 'd' to s1.at( 30 ) yields:
```

Platform specific error message will be dislayed

References

- Working Draft, Standard for Programming Language C++
  - http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2011/n3242.pdf
- C++ FAQ
  - http://www.parashift.com/c++-faq-lite/index.html