Introduction to Object-Oriented Programming

COMP2011: Array — a Collection of Homogeneous Objects

Dr. Cecia Chan
Dr. Cindy Li
Dr. Brian Mak

Department of Computer Science & Engineering
The Hong Kong University of Science and Technology
Hong Kong SAR, China
What is an Array?

- **Array** is a collection of homogeneous objects: objects of the same type. e.g. a collection of int, char, double, . . ., or user-defined types.

- **Exception**: The array elements cannot be reference variables.
Motivation from Programming Point of View

- A function to sort 3 integers can be:
  ```cpp
  void sort_3_int(int& x, int& y, int& z);
  ```

- A function to sort 6 integers can be:
  ```cpp
  void sort_6_int(int& u, int& v, int& w, int& x, int& y, int& z);
  ```

- How about a function to sort 10,000 integers? Are you going to create variable names for the 10,000 different integers?

- **Array** is designed to solve this problem: you only need one identifier name to address all the 10,000 integers, and there is a way to refer to each of them.

- It can solve problems like: read a list of student names, and sort them in alphabetical order.

- In an **Excel file**, each column/row is basically an array so that you can do some common operations (like average, max, min, count) on it.
Part I

1-Dimensional Array
Syntax: **Definition of a 1D Array**

\[ \langle \text{data-type} \rangle \ \langle \text{array-name} \rangle \ [ \langle \text{size} \rangle ] ; \]

- \( \langle \text{size} \rangle \) should be a **positive constant**. It can be a constant expression too.

**Examples**

```cpp
int number[10000]; // an array of 10,000 uninitialized integers

const int NUM_STUDENTS = 335;
char gender[NUM_STUDENTS]; // an array of 335 char
float score[NUM_STUDENTS + 1]; // an extra element to hold the mean

int n = 3;
double x[n]; // compilation error on VC++: size is NOT a constant

int value[-4]; // compilation error: array size cannot be -ve
```
A 1D array is an ordered list of elements.

Successive elements are stored in contiguous memory.

To access an element, use the subscript operator \([\ ]\) with an array index.

For an array of size \(N\), the indices run from 0, 1, 2, \ldots, \(N - 1\).

Each array element is treated like a regular variable:
- you may assign a value to it
- you may assign its value to another variable
- you may pass it by value or reference to a function

( if \(x\) is an int array, \text{sizeof(int)} = 4 )

<table>
<thead>
<tr>
<th>(x[0])</th>
<th>(x[1])</th>
<th>(x[2])</th>
<th>(x[N−1])</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\cdots)</td>
<td>(23)</td>
<td>(588)</td>
<td>(19)</td>
</tr>
<tr>
<td>(1000)</td>
<td>(1004)</td>
<td>(1008)</td>
<td>()</td>
</tr>
<tr>
<td>(\cdots)</td>
<td>(6)</td>
<td>(1000 + 4(N−1))</td>
<td>()</td>
</tr>
</tbody>
</table>
Array works particularly well with loops: e.g. use a for-loop to access and manipulate each array element in turn.

This is not a coincidence, but part of the C++ language design.

Examples

```cpp
int y; // A regular int variable
int x[3]; // An array of 3 int numbers
x[0] = 34; // Array indices start from zero in C++
x[1] = 289;
x[2] = 75; // Index of the last element is 2 NOT 3!
y = x[2]; // Now both y and x[2] are 75
max(x[2], x[0]); // Pass array elements by value
swap(x[1], x[0]); // Pass array elements by reference

for (int j = 0; j < 3; j++)
x[j] *= 3; // Triple each element of an array
```
Example: Manipulate an Array of Scores using for Loop

```cpp
#include <iostream> /* array-mean.cpp */
using namespace std;

int main()
{
    const int NUM_STUDENTS = 5;
    float score[NUM_STUDENTS];
    float sum_score = 0; // Don't forget initializing the sum

    for (int j = 0; j < NUM_STUDENTS; ++j)
    {
        cin >> score[j];
        sum_score += score[j]; // Accumulate the scores
    }

    cout << "mean score = " << sum_score/NUM_STUDENTS << endl;
    return 0;
}
```
```cpp
#include <iostream> /* array-max.cpp */
using namespace std;

int main()
{
    const int NUM_STUDENTS = 5;
    float score[NUM_STUDENTS];
    // Read in the first student’s score. Assume #student >= 1
    cin >> score[0];
    float max_score = score[0]; // A good way to initialize max score

    for (int j = 1; j < NUM_STUDENTS; ++j)
    {
        cin >> score[j];
        if (max_score < score[j])
            max_score = score[j];
    }

    cout << "max score = " << max_score << endl;
    return 0;
}
```
C++ compiler does not automatically check that an array index is out of bound.

That is, for an array of size $N$, the compiler won’t check if it is subscripted with an index between 0 and $N - 1$, neither at compile-time nor run-time.

There is no compilation error for the following codes:

```cpp
int x[10]; x[-2] = 5; x[100] = 9;
```

When the codes are run, $x[-2] = 5;$ will put the value 5 to the memory space which is $2 \times 4$ bytes (size of 2 int) before the array $x$. Similarly, $x[100] = 9;$ will put the value 9 to the memory space which is $90 \times 4$ bytes beyond the array.

This is a common cause of the run-time error called segmentation fault — your program trespasses into memory locations that do not belong to it.
Just like any local variable, when an array is defined, its elements are not initialized automatically.

Syntax: Define and Initialize a 1D Array Simultaneously

```
<data-type> <array-name> [<<size>] = \{ <value_0>, <value_1>, \ldots, <value_{size-1}> \};
```

- If there are fewer values than the array size, the unspecified values will be zeros.
- It is a compilation error if there are more values than the array size.
- If you leave out the array size in the array initialization, the compiler will count the number of initializing values and uses that as the array size.
- Once defined, you cannot assign values to an array using the initialization syntax.
Example: Array Initialization

```c
int a[5] = {1, 2, 3, 4, 5};
/* Same as

    int a[5];
*/

int b[5] = {1, 2};    // => 1, 2, 0, 0, 0
int c[5] = {};        // => 0, 0, 0, 0, 0
int d[] = {1, 2, 3};  // Compiler determines the size=3 automatically
int e[3];

// Compilation error:
// can’t assign values to an array using the { } syntax
e = {5, 6, 7};

// Compilation error: can’t declare an array of references
double x = 1.5, y = 2.5, z = 3.5;
int& s[] = {x, y, z};
```
Common Mis-uses of an Array

While each array element can be treated as a simple variable, the whole array, as represented by the array identifier, cannot.

```cpp
int x[] = {1, 2, 3, 4, 5};
int y[] = {6, 7, 8, 9, 0};
int z[5];

/* Incorrect way */
// Cannot assign to array elements using the initialization syntax
x = {5, 4, 3, 2, 1};

x = 8; // x is not an integer! Its elements are.
x += 2; // x is not an integer! Its elements are.
x = y; // No assignment between 2 arrays
z = x + y; // Cannot +, -, *, / on the array, but only its elements

/* Correct way; what does each for-statement do? */
for (int j = 0; j < 5; ++j) x[j] = 5 - j;
for (int j = 0; j < 5; ++j) x[j] = 8;
for (int j = 0; j < 5; ++j) x[j] += 2;
for (int j = 0; j < 5; ++j) x[j] = y[j];
for (int j = 0; j < 5; ++j) z[j] = x[j] + y[j];
```
Examples: Arrays as Function Arguments

/* function header */
float mean_score(float score[], int size) { ... }
float max_score(float score[], int size) { ... }

/* inside the main() */
float score[NUM_STUDENTS];
mean_score(score, NUM_STUDENTS);
max_score(score, NUM_STUDENTS);

Since the array identifier alone does not tell us about its size, a function that operates on an array needs at least 2 input arguments:
- the array identifier
- the array size (of type int)
#include <iostream> /* array-mean-max-fcn.cpp */
using namespace std;

float mean_score(float score[], int size) {
    float sum_score = 0.0; // Don’t forget initializing the sum to 0
    for (int j = 0; j < size; j++)
        sum_score += score[j]; // Accumulate the scores
    return sum_score/size;
}

float max_score(float score[], int size) {
    // Initialize the max score to that of the first student
    float max_score = score[0];
    for (int j = 1; j < size; j++)
        if (max_score < score[j])
            max_score = score[j];
    return max_score;
}
```c++
int main()
{
    const int NUM_STUDENTS = 5;
    float score[NUM_STUDENTS];

    for (int j = 0; j < NUM_STUDENTS; j++)
        if (!(cin >> score[j])) return -1;

    cout << "mean score = " << mean_score(score, NUM_STUDENTS) << endl;
    cout << "max score = " << max_score(score, NUM_STUDENTS) << endl;
    return 0;
}
```
While a regular variable may be passed to a function by value or reference, an array variable is always passed by value. However, although the array variable is passed by value, its elements are effectively passed by reference! Any change to an array element inside the function will persist even after the function returns. Just like a regular variable, you pass an array to a function simply by its variable name. e.g.

```
max_score(score, NUM_STUDENTS);
```
```cpp
#include <iostream>  /* array-add-rotate.cpp */
using namespace std;

void array_add(int x[], int y[], int z[], int size)
{
    for (int j = 0; j < size; j++)
        z[j] = x[j] + y[j];
}

void circular_rotation(int x[], int size)
{
    int item_0 = x[0];  // Save the first element before rotation
    for (int j = 1; j < size; j++)
        x[j-1] = x[j];  // Rotate up
    x[size - 1] = item_0;  // Fix the last element
}

void array_print(int x[], int size)
{
    for (int j = 0; j < size; j++)
        cout << x[j] << 't';
    cout << endl;
}
```

```cpp
int main()
{
    int a[] = {1, 2, 3, 4};
    int b[] = {11, 12, 13, 14};
    int c[4];

    array_add(a, b, c, 4);
    array_print(c, 4);
    cout << endl;

    for (int k = 0; k < 4; k++)
    {
        circular_rotation(a, 4);
        array_print(a, 4);
    }

    return 0;
}
```
Just like simple constants, an array of constants can be made using the keyword “const”.

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

It defines 4 integer constants: \( x[0] \), \( x[1] \), \( x[2] \), and \( x[3] \) are all of the type `const int`.

Like simple constants, a `constant array`

- must be `initialized` when it is defined.
- once defined, its elements `cannot` be modified.

One main use of constant array is in the definition of the formal parameters of a function: to disallow modification of the `elements` of an array passed to a function, declare that array constant using `const`.

- inside the function, the array is `read-only`.
- however, the original array in the caller is still `writable`. 
#include <iostream> /* const-array-mean-max-fcn.cpp */
using namespace std;

float mean_score(const float score[], int size)
{
    float sum_score = 0.0; // Don’t forget initializing the sum to 0
    for (int j = 0; j < size; j++)
        sum_score += score[j]; // Accumulate the scores
    return sum_score/size;
}

float max_score(const float score[], int size)
{
    // Initialize the max score to that of the first student
    float max_score = score[0];
    for (int j = 1; j < size; j++)
        if (max_score < score[j])
            max_score = score[j];
    return max_score;
}
Example: Prevent Modification by Constant Array ..

```cpp
int main()
{
    const int NUM_STUDENTS = 5;
    float score[NUM_STUDENTS];

    for (int j = 0; j < NUM_STUDENTS; j++)
        if (!(cin >> score[j]))
            return -1;

    cout << "mean score = " << mean_score(score, NUM_STUDENTS) << endl;
    cout << "max score = " << max_score(score, NUM_STUDENTS) << endl;
    return 0;
}
```
Part II

Multi-dimensional Array
In general, an array can be multi-dimensional.
C++ 2-dimensional Array

Syntax: **Definition** of a 2D Array

\[
\text{<data-type> <array-name> [ <size_1> ] [ <size_2> ] ;}
\]

```cpp
int a[2][3] = {1,2,3,4,5,6}; // sizeof(int) = 4
```

![Diagram showing 2D array layout and memory addresses]

- `a[0][0]` is at memory address 1000
- `a[1][0]` is at memory address 1004
- `a[0][1]` is at memory address 1008
- `a[1][1]` is at memory address 1012
- `a[0][2]` is at memory address 1016
- `a[1][2]` is at memory address 1020

ROW 0:
- `a[0][0]` to `a[0][2]`

ROW 1:
- `a[1][0]` to `a[1][2]`
A **2D array** can be initialized in 2 ways:
- **row by row**, or
- **like a 1D array** since the array cells are actually stored linearly in the memory.

### Examples

```c
/* Initialize row by row */
int point[5][2] = { // An int array with 5 rows and 2 columns
    {1, 1},
    {2, 4},
    {3, 9},
    {4, 16},
    {5, 25}
};

/* Initialize using the fact that the cells of a 2D array actually are stored linearly in the memory */
int point[5][2] = { 1,1, 2,4, 3,9, 4,16, 5,25 };`
#include <iostream>    /* File: 2d-array-fcn.cpp */
#include <cmath>
using namespace std;

float distance(float x1, float y1, float x2, float y2)
{
    float x_diff = x1 - x2, y_diff = y1 - y2;
    return sqrt(x_diff*x_diff + y_diff*y_diff);
}

void print_2d_array(const float a[][3], int num_rows, int num_columns)
{
    for (int i = 0; i < num_rows; i++)
    {
        for (int j = 0; j < num_columns; j++)
        {
            cout << a[i][j] << 't';
        }
        cout << endl;
    }
```c
void compute_all_distances(
    const float point[][2], float dist[][3], int num_points)
{
    for (int i = 0; i < num_points; i++)
        for (int j = 0; j < num_points; j++)
            dist[i][j] = distance(point[i][0], point[i][1],
                                    point[j][0], point[j][1]);
}

int main()
{
    float dist[3][3];  // Distances between any pairs of points
    float point[3][2]  // (x, y) coordinates of 3 points
                       = { {1.0, 1.0} , {2.0, 2.0} , {4.0, 3.0} };

    compute_all_distances(point, dist, 3);
    print_2d_array(dist, 3, 3);
    return 0;
}
```
Syntax: Definition of an N-dimensional Array

\[
\langle \text{data-type} \rangle \ \langle \text{array-name} \rangle \ [\langle \text{size}_1 \rangle] [\langle \text{size}_2 \rangle] \cdots [\langle \text{size}_N \rangle] ;
\]

```cpp
type array_name [size_1][size_2] ... [size_N];
```

```cpp
int a[2][2][2] = {1,2,3,4,5,6,7,8}; // sizeof(int) = 4
```

Diagram of a 2x2x2 array with its memory layout.
Remarks on Multi-dimensional Array

- Although conceptually a 2D array is like a matrix, and a 3D array is like a cube, the elements of a multi-dimensional array are stored linearly in the memory (just like a 1D array).

- In C++, the elements of a multi-dimensional array are stored in row-major order: row by row.

- There are programming languages (e.g. FORTRAN) that store multi-dimensional array elements in column-major order: column by column.

- In row-major order, the last dimension index runs fastest, while the first dimension index runs slowest.

- If a multi-dimensional array is used in a C++ function, all dimensions other than the first dimension must be specified in its declaration in the function header.
Part III

C String: Special 1D Character Array

'h' 'k' 'u' 's' 't' '\0'
In general, one cannot deal with the whole array at once, but has to deal with each array element, one at a time because a sequence of, e.g., integers, do not represent a new object.

A char array is different: a sequence of chars may be interpreted as a word or sentence or paragraph or even a novel!

C++ follows C’s trick of representing a character string by a 1D character array with the end-marker ’\0’.

Just add the null character ’\0’ (ASCII code = 0) after the last character of the string you need.

'h' 'k' 'u' 's' 't' '0'}
For a string of length $N$, add '\0' as the $(N+1)$th element of its char array.

Now if everyone writes functions of char arrays that represents strings with the above understanding, then one doesn’t need to pass the size of such char arrays to their functions!

C++ allows another notation using the double quotes. e.g.

```
“hkust” = ’h’ ’k’ ’u’ ’s’ ’t’ ’\0’
```
Example: C String

```cpp
#include <iostream> /* File: c-string.cpp */
using namespace std;

int main()
{
    char s1[6] = {'h', 'k', 'u', 's', 't', 'z'};

    // At this point, s1 is still a simple char array
    for (int j = 0; j < 5; j++)
        cout << s1[j];
    cout << endl;
    s1[5] = '\0'; // Now, s1 is a C string
    cout << s1 << endl;

    // Another notation for initializing literal constant strings
    char s2[20] = {'h', 'k', 'u', 's', 't', '\0'};
    cout << "s2 = " << s2 << endl;
    char s3[20] = "hkust"; cout << "s3 = " << s3 << endl;
    return 0;
}
```
```cpp
#include <iostream>    /* File: c-string-fcn.cpp */
using namespace std;
const char NULL_CHAR = '\0';

int str_len(const char s[]) 
{
    int j;
    for (j = 0; s[j] != NULL_CHAR; j++)
        ;
    return j;
}

int str_concatenate(const char s1[], const char s2[], char s[])
{
    int j;
    for (j = 0; s1[j] != NULL_CHAR; j++)
        s[j] = s1[j];   // Copy s1 to s

    for (int k = 0; s2[k] != NULL_CHAR; k++, j++)
        s[j] = s2[k];   // Copy s2 after s1

    s[j] = NULL_CHAR;   // Make s a C String
    return j;
}
```

```c
int main()
{
    char a[20] = "Albert";
    char b[20] = "Einstein";
    char c[20];
    int length;

    cout << "length of string a = " << str_len(a) << endl;
    cout << "length of string b = " << str_len(b) << endl;

    length = str_concatenate(a, b, c);
    cout << "After concatenation: "
         << c << " of length " << length << endl;

    return 0;
}
```
#include <iostream>  /* File: str-array.cpp */
using namespace std;

void print_strings(const char s[][16], int num_of_strings)
{
    for (int j = 0; j < num_of_strings; j++)
        cout << s[j] << " ";
    cout << endl;
}

int main()
{
    // 5 C-strings, each having a max. length of 15 char
    const char word[5][16] = {
        "hong kong",
        "university",
        "of",
        "science",
        "technology"
    };

    print_strings(word, 5);
    return 0;
}
cin will skip all white spaces before reading data of the required type until it sees the next white space. White spaces are any sequence of ’ ’, ’\t’ and ’\n’.

For `char x; cin >> x;`, if the input is “hkust”, cin will skip all the leading white spaces, and gives ’h’ to x.

The same is true for reading a C string.

For `char x[20]; cin >> x;`, if the input is “hkust”, cin will skip all the leading white spaces, and gives “hkust” to x.

Thus, cin is not good at reading multiple words or even a paragraph including possibly the newline. Instead, use:

```
cin.getline(char s[], int max-num-char, char terminator);
```

cin.getline() will stop when either (max-num-char - 1) characters are read, OR, the terminating character terminator is seen. The terminating character is removed from the input stream but is not read into the string.

The C-string terminating null character is automatically inserted at the end of the read string.
```cpp
#include <iostream> /* File: read-str.cpp */
using namespace std;

int main()
{
    const int MAX_LINE_LEN = 1000;
    char s[MAX_LINE_LEN+1];

    // Read until the newline character (default)
    cin.getline(s, MAX_LINE_LEN+1, '\n');
    cout << s << endl;

    // Read until the character 'W'
    cin.getline(s, sizeof(s), 'W');
    cout << s << endl;

    return 0;
}
```
```cpp
#include <iostream> /* File: palindrome.cpp */

using namespace std;

bool palindrome(const char x[]) {
    int j = 0; // An index reading the array from top (left)
    int k = strlen(x) - 1; // An index reading the array from bottom (right)
    for ( ; j < k; ++j, --k)
        if (x[j] != x[k])
            return false;

    return true;
}

int main()
{
    const int MAX_LINE_LEN = 255;
    char whole_line[MAX_LINE_LEN+1];

    while (cin.getline(whole_line, MAX_LINE_LEN+1, '\n'))
        cout << boolalpha << palindrome(whole_line) << endl;
    return 0;
}
```