

Array, String, Vector, Structure

Topics

- Arrays Hold Multiple Values
- Accessing Array Elements
- Inputting and Displaying Array Contents
- Array Initialization
- Processing Array Contents
- Using Parallel Arrays
- Arrays as Function Arguments
- Two-Dimensional Arrays
- String
- Array and String
- Vector
- Structure

Array

If we are required to write a program to sort 50 grades for a class of students, how can we write that program?

```
int grade1, grade2, grade3, ..., grade50;
```

50 variables!

Array

- An **array** (数组) offers a solution to this problem
- Array is a derived data type

- It itself is not a type
- Every element in the array has **same type**
- Declared using [] operator

```
const int ISIZE = 50;
```

```
int tests[ISIZE];
```

- For example:

```
int grade[50];
```

instead of

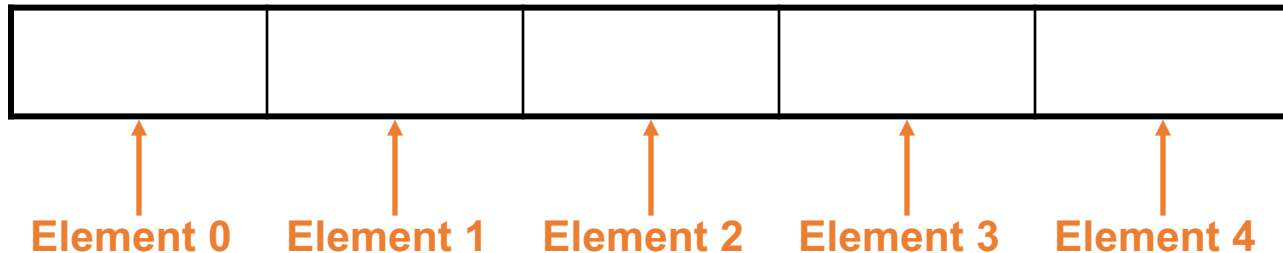
```
int grade1, grade2, grade3, ..., grade50;
```

Array Storage in Memory

The definition

```
int tests[SIZE]; // ISIZE = 50
```

allocates the following memory



Values are stored in **adjacent** memory locations

Array Terminology

In the definition `int tests[SIZE] ;`

- `int` is the data type of the array elements
- `tests` is the name of the array
- `SIZE`, in `[SIZE]`, is the size declarator. It shows the number of elements in the array.
- The size of an array is the number of bytes allocated for it
*(number of elements) * (bytes needed for each element)*

Array Terminology Examples

Examples:

Assumes **int** uses 4 bytes and **double** uses 8 bytes

```
const int ISIZE = 5, DSIZE = 10;
```

```
int tests[ISIZE]; // holds 5 ints, array  
                  // occupies 20 bytes
```

```
double volumes[DSIZE]; // holds 10 doubles  
                       // array is 80 bytes
```

Accessing Array Elements

- Each array element has a subscript, used to access the element.
- Subscripts start at 0



Accessing Array Elements

Array elements (accessed by array name and subscript) can be used as regular variables

tests



```
tests[0] = 79;  
cout << tests[0];  
cin >> tests[1];  
tests[2] = tests[0] + tests[1];  
cout << tests; // illegal due to  
               // missing subscript
```

Inputting and Displaying Array Contents

cout and **cin** can be used to display values from and store values into an array

```
const int ISIZE = 5;  
  
int tests[ISIZE];  
  
cout << "Enter first test score ";  
cin  >> tests[0];
```

Array Subscripts

- Array subscript can be an integer constant, integer variable, or integer expression

• Examples:	<u>Subscript is</u>
<code>cin >> tests[3];</code>	int constant
<code>cout << tests[i];</code>	int variable
<code>cout << tests[i+j];</code>	int expression

What happens if subscript is not an integer???

Inputting and Displaying All Array Elements

To access each element of an array

- Use a loop (usually a **for** loop)
- Let loop control variable be array subscript
- A different array element will be referenced each time through loop

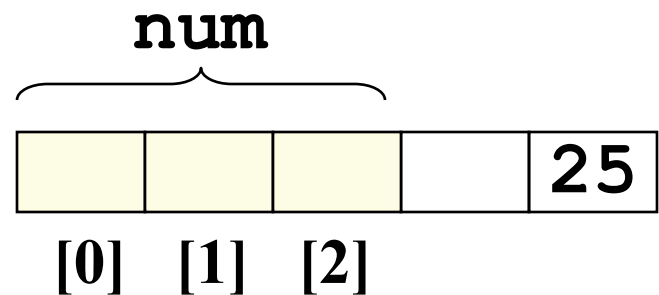
```
for (i = 0; i < 5; i++)  
    cout << tests[i] << endl;
```

Note difference between array location and contents!!!

No Bounds Checking

- There are no checks in C++ that an array subscript is in range – so no messages
- An invalid array subscript can cause program to overwrite other memory
- Example:

```
const int ISIZE = 3;  
int i = 4;  
int num[ISIZE];  
num[i] = 25;
```



Off-By-One Errors

- Most often occur when a program accesses data one position beyond the end of an array, or misses the first or last element of an array.
- Don't confuse the ordinal number of an array element (first, second, third) with its subscript (0, 1, 2)

Array Initialization

- Can be initialized during program execution with assignment statements

```
tests[0] = 79;
```

```
tests[1] = 82;
```

- Can be initialized at array definition with an initialization list

```
const int ISIZE = 5;
```

```
int tests[ISIZE] = {79,82,91,77,84};
```

Partial Array Initialization

- If array initialized at definition with fewer values than size of array, remaining elements set to 0 or **NULL**

```
int tests[ISIZE] = {79, 82};
```

79	82	0	0	0
----	----	---	---	---

- Initial values used in order; cannot skip over elements to initialize noncontiguous range

Implicit Array Sizing

- Can set array size by size of the initialization list

```
short quizzes[]={12,17,15,11};
```

12	17	15	11
----	----	----	----

- Must use either array size declarator or initialization list when array is defined

```
short quizzes[4]={12,17,15,11};
```

Processing Array Contents

- Array elements can be
 - treated as ordinary variables of same type as array
 - used in arithmetic operations, in relational expressions, etc.
- Example:

```
if (principalAmt[3] >= 10000)
    interest = principalAmt[3] * intRate1;
else
    interest = principalAmt[3] * intRate2;
```

Using Increment & Decrement Operators with Array Elements

When using `++` and `--` operators, don't confuse the element with the subscript

```
tests[i]++; // adds 1 to tests[i]
```

```
tests[i++]; // increments i, but has  
            // no effect on contents  
            // of tests
```

Copying One Array to Another

- Cannot copy with an assignment statement:

```
tests2 = tests; //won't work
```

- Must instead use a loop to copy element-by-element:

```
for (int indx=0; indx < ISIZE; indx++)  
    tests2[indx] = tests[indx];
```

Are Two Arrays Equal?

- Like copying, cannot compare in a single expression:

```
if (tests2 == tests)
```

- Use a while loop with a boolean variable:

```
bool areEqual=true;
int indx=0;
while (areEqual && indx < ISIZE)
{
    if(tests[indx] != tests2[indx])
        areEqual = false;
    index++;
}
```

Sum, Average of Array Elements

- Use a simple loop to add together array elements

```
float average, sum = 0;  
for (int tnum=0; tnum< ISIZE; tnum++)  
    sum += tests[tnum];
```

- Once summed, average can be computed

```
average = sum/ISIZE;
```

Largest Array Element

- Use a loop to examine each element & find largest element (*i.e.*, one with largest value)

```
int largest = tests[0];  
for (int tnum = 1; tnum < ISIZE; tnum++)  
{   if (tests[tnum] > largest)  
    largest = tests[tnum];  
}  
cout << "Highest score is " << largest;
```

- A similar algorithm exists to find the smallest element

Partially-Filled Arrays

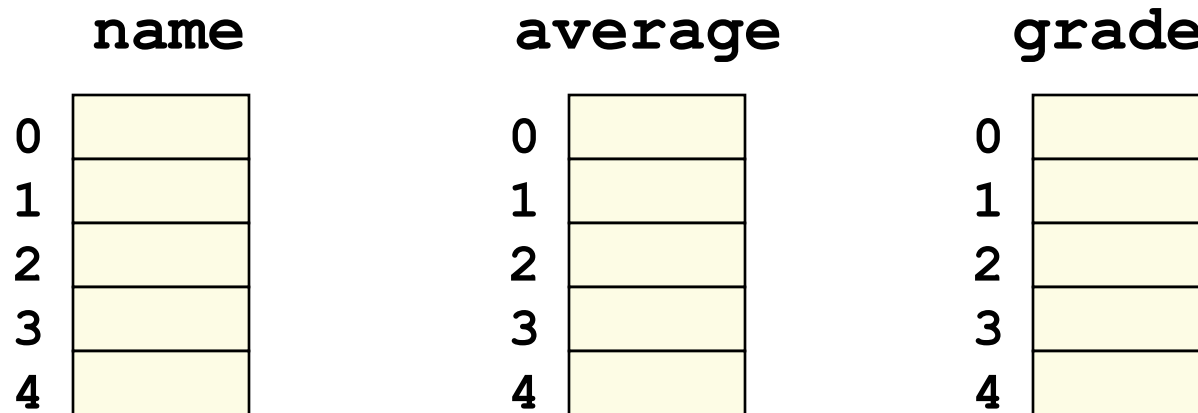
- Exact number of data (and, therefore, array size) may not be known when a program is written.
- Programmer makes best estimate for maximum amount of data, sizes arrays accordingly.
- Programmer must also keep track of how many array elements are actually used

Using Parallel Arrays

- Parallel arrays: two or more arrays that contain related data
- Subscript is used to relate arrays
 - elements at same subscript are related
- The arrays do not have to hold data of the same type

Parallel Array Example

```
const int ISIZE = 5;  
  string name[ISIZE];    // student name  
  float average[ISIZE];  // course average  
  char grade[ISIZE];     // course grade
```



Parallel Array Processing

```
const int ISIZE = 5;
    string name[ISIZE];    // student name
float average[ISIZE]; // course average
char grade[ISIZE];    // course grade
...
for (int i = 0; i < ISIZE; i++)
    cout << " Student: " << name[i]
        << " Average: " << average[i]
        << " Grade: "    << grade[i]
        << endl;
```

Two-Dimensional Arrays

- Can define one array for multiple sets of data
- Like a table in a spreadsheet
- Use two size declarators in definition

```
int exams[4][3];
```



Number
of rows

An orange oval containing the text "Number of rows" has an orange arrow pointing from it to the number 4 in the array definition above.



Number
of cols

An orange oval containing the text "Number of cols" has an orange arrow pointing from it to the number 3 in the array definition above.

First number
ALWAYS
represents
Rows!!

Two-Dimensional Array Representation

```
int exams[4][3];
```

	columns		
r o w s	exams[0][0]	exams[0][1]	exams[0][2]
	exams[1][0]	exams[1][1]	exams[1][2]
	exams[2][0]	exams[2][1]	exams[2][2]
	exams[3][0]	exams[3][1]	exams[3][2]

Use two subscripts to access element

```
exams[2][2] = 86;
```

Initialization at Definition

- Two-dimensional arrays are initialized row-by-row

```
int exams[2][2] = { {84, 78},  
                    {92, 97} };
```

84	78
92	97

- Can omit inner { }
- What will this do?

```
int exams[2][2] = { 84, 78};
```

2D Array Traversal

- Traversal: to visit every element in a data structure
- Use **nested loops**, one for row and one for column, to visit each array element.
 - Outer loop determines if by row or column
- Accumulators can be used to sum (process) elements row-by-row, column-by-column, or over the entire array.

Example: 2D array

```
// Fill array by rows
int A[3][5],r,c; //3 rows,5 columns
for (r=0;r<3;++r)
    for (c=0;c<5;c++)
        { cin >> A[r][c]; }
```


Example: 2D array

```
// Print array by rows
int A[3][5],r,c;//3 rows,5 columns
for (r=0;r<3;++r)
    {for (c=0;c<5;c++)
        { cout << A[r][c];}
    cout << '\n' ;
}
```

Example: 2D array

```
// Sum & Avg values in 3x5 array
int A[3][5], r, c, sum = 0;
float avg;
for (r=0; r<3; ++r)
    {for (c=0; c<5; c++)
        {sum += A[r][c]}
    }
avg = sum/15.0;
```

Processing by Columns

```
// Fill array by columns
int A[3][5],r,c; //3 rows
                        //5 columns
for (c=0;r<5;++c)
    for (r=0;c<3;r++)
        { cin >> A[r][c]; }
```

String

String Declaration

- A string is a sequence of characters
 - E.g., "abcd", "", " "
- Three ways to declare a string variable
 - Array
 - Pointer
 - Class

String Class

- Class
 - Format
 - `string s1, s2;`
 - `s1` and `s2` are two string variables.
 - Here `string` is not a data type, it is a **class** that will be introduced later.
 - The code preprocessor must include

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

String Initialization

```
string s1;  
string s2("hello");  
string s3 = "World";
```

S1 is given a null string as the initial value.
S2's initial value is "Hello".
S3's initial value is "World".

String Input

- Read a string
 - `cin >> str;`
 - `cin` read a string and stops at a space
 - For example

```
cin >> s1 >> s2;
```

- If the input is

I am a UIC student

- "I" will be assigned to `s1` and "am" will be assigned to `s2`.

String Output

- Output a string
 - `cout << str;`

An Example

```
#include <string>
#include<iostream>
using namespace std;
int main ()
{
    string word1 = "Hello ", word2 = " World!";
    cout << word1 << word2 << endl;
    return 0;
}
```

Get a Line

- **Use** `getline`
- **For example**

```
string strline;  
getline(cin, strline); // read until Enter  
getline(cin, strline, '?'); // read until the symbol '?'
```

Class Exercise

```
#include <string>
#include<iostream>
using namespace std;
int main ()
{
    string strline;
    getline(cin, strline); // read until Enter
    cout << strline << endl;
    getline(cin, strline, 'a');// read until the symbol 'a'
    cout << strline << endl;
    return 0;
```

```
}
```

Assume input:

I am a UIC student

I am a UIC student

Output??

String Operation

- Length of a string
- Locate part of a string
- Modification/Assignment
- Compare
- Find a position

Length of a String

- `str.length()`
 - Calculate the total number of characters in a string `str`.
 - For example: `str.length()`
 - If `str` is "abc", `str.length()` will return a value 3.
- `str.empty()`
 - Check if a string is empty or not. If empty, return true; otherwise false.
 - For example: `str.empty()`
 - If `str` is "abcde", `str.empty()` will return false.

Locate Part of AString

- `str.at(i)`
 - Locate the character at the position `i` of the string `str`.
 - For example: `c = str.at(2)`
 - If `str` is "abcde", `str.at(2)` will return a char `'c'`.
- `str.substr(pos, len)`
 - Locate the string in `str`, starting from the position `pos` and with the length `len`.
 - For example: `str.substr(3, 2)`
 - If `str` is "abcde", `str.substr(3, 2)` will return a string "de".

Modification/Assignment

- `str1 + str2`
 - return concatenation of `str1` and `str2`.
 - **For example:** `str3 = str1 + str2;`
 - If `str1` is "abc", `str2` is "def", `str3` is "abcdef"
- `str1 = str2`
 - **Assign** `str2` to `str1`
 - **For example:** `str3 = str1;`
 - If `str1` is "abc", `str3` is "abc"

Modification/Assignment

- `str1.insert(pos, str)`
 - Insert the string `str` into `str1` at the position `pos`.
 - For example: `str1.insert(2, str2)`
 - If `str1` is "abc", `str2` is "def", then `str1` will be changed to "abdefc"
- `str1.erase(pos, len)`
 - Remove the string starting from `pos` with a length `len` from `str1`.
 - For example: `str1.erase(2, 3);`
 - If `str1` is originally "abcdefg", the resulting `str1` is "abfg".

Compare

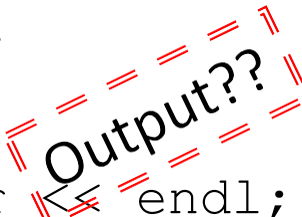
- `==`, `<=`, `>=`, `<`, `>`, `!=`
 - All the relational operators can be used to compare strings.
 - Strings are compared based on ASCII code starting from the first character.
 - Comparison stops when the characters in two strings are different.
 - For example
 - `"Hello"` and `"Hello "`: which one is bigger?
 - `"He llo"` and `"Hello"`: which one is bigger?
 - `"Hello world"` and `"Hello World"`: which one is bigger?

Find

- `str.find(str1)`
 - Find the position of the first occurrence of `str1` in `str`.
 - For example: `str.find(str1)`
 - If `str` is "abcdecfc dg", `str1` is "cd", then `str.find(str1)` will return a value 2.
- `str.find(str1, pos)`
 - Find the position of the first occurrence of `str1` in `str`, starting from `pos`.
 - For example: `str.find(str1, 4)`
 - If `str` is "abcdecfc dg", `str1` is "cd", then `str.find(str1, 4)` will return a value 7.

Class Exercise

```
#include <string>
#include<iostream>
using namespace std;
int main ()
{
    string str("HelloWorld, Hey"), str1("He");
    string str2;
    if(!str1.empty()) {
        cout << "str1's length is " <<str1.length();
        if (str.find(str1)>=0)
            cout << str1 << " can be found in " << str << endl;
    }
    str2 = str.substr(5, 5);
    cout << str2 << " is extracted from " << str << endl;
    return 0;
}
```



After Class

- Think and practice to find the answers to the following questions:
 - In `str.at(i)`, if `i` is out of range of valid index of `str`, what will happen? For example, `str.at(6)` while `str` is "abc".
 - In `str.find(str1)`, if `str1` does not exist in `str`, what is the result? For example, `str.find("oo")` while `str` is "abc".

Array and String

- Astring can be declared using array.
 - For example
 - `char str[20];`
 - `str` is a string variable, it can store at most 20 characters
 - The string **must** end with `'\0'`.

Array and String

In C/C++ one way of dealing with strings is an array of type **char**.
In C every string has the special feature that the final character of the string is the "null" character denoted `\0`.

```
char dog[5]={ 'b', 'o', 'x', 'e', 'r'}; // not a string
char cat[5]={ 'f', 'a', 't', 's', '\0'}; // a string
```

The null character at the end tells different routines when to stop.

The method shown above looks tedious to use. A different way:

```
char bird[10] = "Mr. Cheeps";
/* \0 is understood at the end. */
char name[] = "John Chrispell";
/* Let the compiler count */
```

Note if you initialize to much space all space at the end of the array will be denoted by the null character.

Example:

```
char cArrayStr1[] = "hello";    // String, can "cout"
char cArray[] = {'h', 'e', 'l', 'l', 'o'};    // not string, cannot "cout"
char cArrayStr2[] = {'h', 'e', 'l', 'l', 'o', '\0'};    // same as cArrayStr1
```

Char Array and String

The string class in C++ requires the use of the string header file, and is part of the `std` namespace, and has a `string` type object that is similar to a character array. Consider the following `strtype1.cpp` example:

```
#include <iostream>
#include <string>           // make string class available
int main(){
    using namespace std;
    char charr1[20];        // create an empty array
    char charr2[20] = "jaguar"; // create an initialized array
    string str1;            // create an empty string object
    string str2 = "panther"; // create an initialized string

    cout << "Enter a kind of feline: ";
    cin >> charr1;
    cout << "Enter another kind of feline: ";
    cin >> str1;           // use cin for input
    cout << "Here are some felines:\n";
    cout << charr1 << " " << charr2 << " "
        << str1 << " " << str2 // use cout for output
        << endl;
    cout << "The third letter in " << charr2 << " is "
        << charr2[2] << endl;
    cout << "The third letter in " << str2 << " is "
        << str2[2] << endl;    // use array notation

    return 0;
}
```


Char Array and String

```
#include <iostream>
#include <string>           // make string class available
#include <cstring>          // C-style string library
int main(){

    using namespace std;
    char charr[20];
    string str;

    cout << "Length of string in charr before input: "
         << strlen(charr) << endl;
    cout << "Length of string in str before input: "
         << str.size() << endl;
    cout << "Enter a line of text:\n";
    cin.getline(charr, 20);    // indicate maximum length
    cout << "You entered: " << charr << endl;
    cout << "Enter another line of text:\n";
    getline(cin, str);        // cin now an argument; no length specifier
    cout << "You entered: " << str << endl;
    cout << "Length of string in charr after input: "
         << strlen(charr) << endl;
    cout << "Length of string in str after input: "
         << str.size() << endl;

    return 0;
}
```

Vector

Vectors

- Holds a set of elements, like an array
- Flexible number of elements - can grow and shrink
 - No need to specify size when defined
 - Automatically adds more space as needed
- Defined in the Standard Template Library (STL)
 - Covered in a later chapter
- Must include **vector** header file to use vectors

```
#include <vector>
```

Vectors

- Can hold values of any type
 - Type is specified when a vector is defined

```
vector<int> scores;  
vector<double> volumes;
```
- Can use `[]` to access elements

Defining Vectors

- Define a vector of integers (starts with 0 elements)
`vector<int> scores;`
- Define **int** vector with initial size 30 elements
`vector<int> scores(30);`
- Define 20-element **int** vector and initialize all elements to 0
`vector<int> scores(20, 0);`
- Define **int** vector initialized to size and contents of vector **finals**
`vector<int> scores(finals);`

Growing a Vector's Size

- Use **push_back** member function to add an element to a full array or to an array that had no defined size

```
// Add a new element holding a 75  
scores.push_back(75);
```

- Use **size** member function to determine number of elements currently in a vector

```
howbig = scores.size();
```

Removing Vector Elements

- Use **pop_back** member function to remove last element from vector

```
scores.pop_back();
```

- To remove all contents of vector, use **clear** member function

```
scores.clear();
```

- To determine if vector is empty, use **empty** member function

```
while (!scores.empty()) ...
```

Structure

Structure

- To represent one item, we can declare single **variables**
 - e.g., `int grade;`
- To represent several items of the same type, we can declare an **array**
 - e.g., `int grade[100];`

If we have 100 students, each student's information includes **name, age, gender, and major**

How to deal with this???

An Example

- A birthday consists of 3 parts: `year`, `month`, and `day`.
- We can define them in this way

```
int year = 2007;
```

```
int month = 11;
```

```
int day = 13;
```

- These 3 variables are logically related, we cannot see that the above declarations are for a birthday.
- It would be better if they can be grouped **together**.

An Example

A structure called `birthday`

```
struct birthday {  
    int year;        // 1st member variable  
    int month;       // 2nd member variable  
    int day;         // 3rd member variable  
};                  // don't forget the semicolon!
```

- `birthday` now is a new type that can be used like `int`, `char`, ...
- `birthday` has three members.

struct and array

- A structure is a **collection** of **related data items** of same or different types, usually contribute to one object
 - e.g.,
 - **Student**: student id, name, major, gender, start year, ...
 - **BankAccount**: account number, name, currency, balance, ...
 - **AddressBook**: name, address, telephone number, ...
- Indicated by keyword **struct**
- An **array** contains only **data of the same type**, usually the data in an array do not have coherent relationship.

struct Variable Declarations

- To declare a variable of `struct` type

```
struct date {  
    int year;  
    int month;  
    int day;  
};
```

```
date birthday;
```

Accessing the Members of a Structure

- A member of a structure is accessed by specifying the **variable name**, followed by a **period**, and then the **member name**

```
struct date {  
    int year;  
    int month;  
    int day;  
};  
date today;  
  
cin >> today.month >> today.day;  
  
if(today.month == 1 && today.day == 1)  
    cout << "Happy new year";
```

struct-to-struct Assignment

```
struct studentRecord{  
    string name;  
    int id;  
    string dept;  
    char gender;  
};  
studentRecord student1, student2;  
  
student1.name = "Tom Hanks";  
student1.id = 12345;  
student1.dept = "COMP";  
student1.gender = 'M';  
  
student2 = student1;
```