Programming Design

Arrays

Ling-Chieh Kung

Department of Information Management
National Taiwan University

Variables and arrays

- Today we introduce **arrays**.
  - A collection of variables of the same type.
  - An array variable is of an **array type**, a **nonbasic data type**.
- There are many nonbasic data types:
  - Arrays.
  - Pointers.
  - Self-defined data types (e.g., classes).
- Before we introduce arrays, let's talk more about variables and basic data types.

Outline

- **More about variables**
  - Constant variables
  - Casting among basic data types
- Single-dimensional arrays
- Multi-dimensional arrays

Constant variables

- Sometimes we want to use a variable to store a particular value.
  - In a program doing calculations regarding circles, the value of π may be used **repeatedly**.
  - We do not want to write many `3.14` throughout the program! Why?
  - We may declare `pi = 3.14` once and then use `pi` repeatedly.
- In this case, this variable is actually a **symbolic constant**.
  - We want to prevent it from being **modified**.

Constant variables

- A **constant** is one kind of variables.
- To declare a constant, use the key word **`const`**:
  - `const int a = 100;`
  - All further assignment operations on a constant generate compilation errors.
  - That is why we must **initialize** a constant.
- It is suggested to use **capital characters** and **underlines** to name constants. This distinguishes them from usual variables.
  - `const double PI = 3.1416;`
  - `const int MAX_LEVEL = 5;`
  - Some people use lowercase characters and underlines.

Casting

- Variables are **containers**.
- Variables of different types are containers of different **sizes/shapes**.
  - **`long`** ≧ **`int`** ≧ **`short`**.
  - "Shapes" of **`int`** and **`float`** are different (though sizes are identical).
- A big container may store a small item. A big item must be "cut" to be stored in a small container.
  - So are variables of different types.

```
short s = 100;
int i = s; // 100
i = 100000;
s = i; // -31072
```

```
double d = 5; // d = 5.0
int s = 5.5; // s = 5
```

Casting

- Changing the type of a variable or literal is called **casting**.
- There are two kinds of casting:
  - **Implicit casting**: from a small type to a large type.
  - **Explicit casting**: from a large type to a small type.
- When implicit casting occurs, there is no value of precision loss.
  - The system does that automatically.
  - The value of that variable or literal does not change.
  - There is no need for a programmer to indicate how to implicitly cast one small type to a large type.
- To cast a large type to a small type, a programmer is responsible for indicating **how to do it** explicitly.

Explicit casting

- Suppose we want to store 5.6 to an integer:
  - `int a = 5.6;` is not good.
  - `int a = static_cast<int>(5.6);` is better.
- To cast basic data types, we use **`static_cast`**:

```
static_cast<type>(expression)
```

  - When a float or double is cast to an integer value (and there is no value loss), the fractional part is **truncated**.
- In the example above, both statements makes **`a`** equal 5.
  - Then why bothering?

Explicit casting

- Explicit casting is to indicate the **way** of casting we want.
  - For basic types, there is only one way to cast a large type to a small type.
  - For more complicated types, however, there may be **multiple**.
- There are four different explicit casting operators.
  - **`static_cast`**, **`dynamic_cast`**, **`reindivter_cast`**, and **`const_cast`**.
  - For basic data types, **`static_cast`** is enough.
- By explicitly indicating how to cast:
  - This is to make sure that, at the run time, the program runs as we expect.
  - This is also to notify other programmers (or the future ourselves).
- Explicit casting also allows for a temporary change of types (see below).

Good programming style

- There is an old way of explicit casting:

```
(type) expression
```

  - For example, `int a = (int) 5.6;` .
- Try to avoid it!
  - This operation includes all four possibilities, and we have no idea which one will be performed at the run time.
- If possible, try to modify your variable declaration to avoid casting.

Casting for division

- Let's try this program:
- The **division** operator returns an integer if both operands (numerator and denominator) are integers.
- How to get our desired results?
  - If allowed, we may change the data types of the operands.
  - If not allowed, we may cast the operands **temporarily**.

```
int d1 = 10;
int d2 = 3;
cout << d1 / d2 << "\n";

double d3 = 10;
int d4 = 3;
cout << d3 / d4 << "\n";

int d5 = 10;
double d6 = 3;
cout << d5 / d6 << "\n";
```

Casting for division

- Which one works?

```
int d1 = 10;
int d2 = 3;
cout << static_cast<double>(d1 / d2);
```

```
int d1 = 10;
int d2 = 3;
cout << static_cast<double>(d1) / d2;
```

- Casting can be a big issue when we work with nonbasic data types.
- At this moment, just be aware of fractional and integer values.

Outline

- More about variables
- **Single-dimensional arrays**
- Multi-dimensional arrays

Set of similar variables

- Suppose we want to write a program to store five students' scores.
- We may declare 5 variables.
  - `int score1, score2, score3, score4, score5;`
- What if we have 500 students? How to declare 500 variables?
- Even if we have only 5, we are unable to write a loop to process them.

```
for (int i = 0; i < 5; i++)
{
  cout << score1; // and then?
  cout << scorei; // error!
}
```

Why arrays?

- An array is a collection of variables with **the same type**.
- To declare five integer variables for scores, we may write:

```
int score[5];
```

  - These variables are declared with **the same array name** (**score**).
  - They are distinguished by their **indices**.

```
cout << score[2];
```

An array is a type

- Arrays are often used with loops.
  - Quite often the loop counter is used as the array index.

```
int score[5];
for (int i = 0; i < 5; i++)
  cin >> score[i];
for (int i = 0; i < 5; i++)
  cout << score[i] << " ";
```

- An array is also a (nonbasic) **type**.
  - The type of **`score`** is an "integer array" (of length 5).
  - What is this?

```
cout << score;
```

- We will go back to this when we introduce pointers.
  - For now, just treat an array as a sequence of variables.

Array declaration

- The grammar for declaring an array is

```
data type array name[number of elements];
```

- E.g., `int score[5];`
  - This is an integer array with five elements (the **array length/size** is 5).
  - Each **array element** itself is a **variable**.
  - The **index** starts at **0**! They are `score[0]`, `score[1]`, …, and `score[4]`.
- It occupies 4 bytes * 5 = 20 **continuous** bytes.
  - Try `cout << sizeof(score);`!

Address	Identifier	Value
0x20c648	score[0]	?
0x20c64c	score[1]	?
0x20c650	score[2]	?
0x20c654	score[3]	?
0x20c658	score[4]	?

Memory

An example

- We have written a program for 5 scores:

```
int score[5];
for (int i = 0; i < 5; i++)
  cin >> score[i];
for (int i = 0; i < 5; i++)
  cout << score[i] << " ";
```

- If we have 500 students:

```
int score[500];
for (int i = 0; i < 500; i++)
  cin >> score[i];
for (int i = 0; i < 500; i++)
  cout << score[i] << " ";
```

Array initialization

- Arrays are not initialized automatically.

```
int array[100];

for (int i = 0; i < 100; i++)
{
  cout << array[i] << " ";
  if (i % 10 == 9)
    cout << "\n";
}
```

Array initialization

- Various ways of initializing an array:
  - `int dayInMonth[12] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};`
  - `int dayInMonth[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};` (size of **dayInMonth** will be 12)
  - `int dayInMonth[12] = {31, 28, 31};` (**nine** 0s)
  - `int dayInMonth[3] = {1, 2, 3, 4};` (error!)
- To initialize all elements to 0:
  - `int score[500] = {0};` (500 0s)

The boundary of an array

- In C++, it is **allowed** for one to "go outside an array".
  - No compilation error!
  - **May or may not** generate a **run time error**: If our program try to access a memory space allocated to another program, the operating system will terminate our program.
  - The result is **unpredictable**.
- A programmer must be aware of array bounds by herself/himself.

```
int array[100] = {0};

for (int i = 0; i < 500; i++)
{
  cout << array[i] << " ";
  if (i % 10 == 9)
    cout << "\n";
}
```

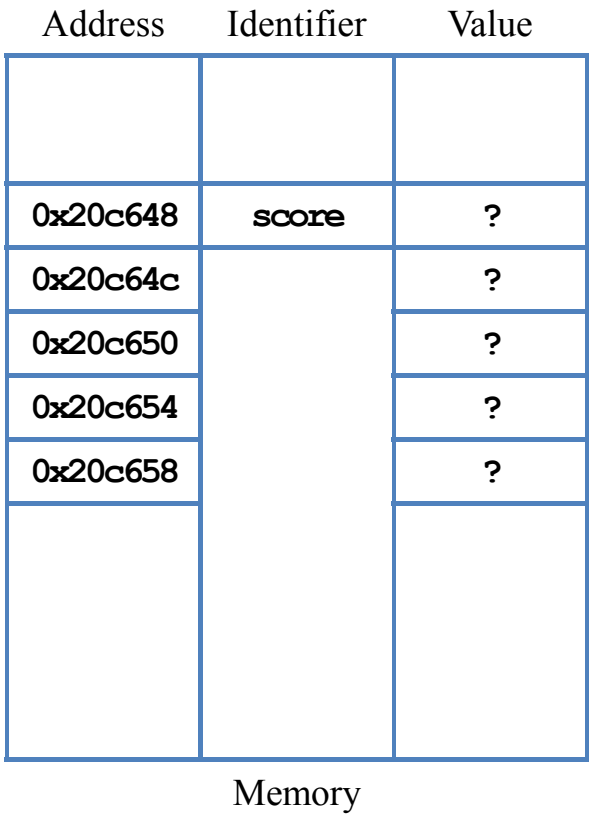
Memory allocation for arrays

- So what happens when we declare or access an array?
- When we declare an array:

```
int score[5];
```

  - The system allocates memory spaces according to the type and length.
  - The array variable indicates the **beginning address** of the space.

```
cout << score; // 0x20c648
```

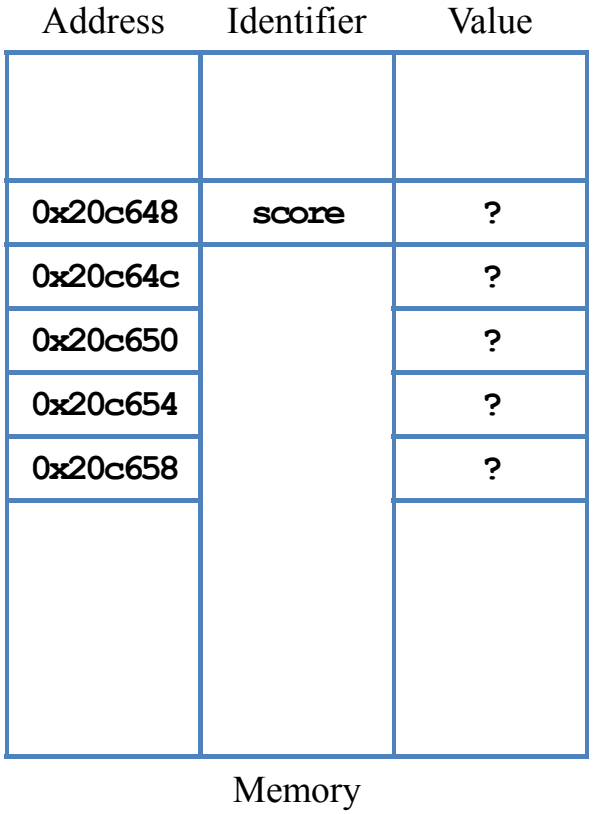


Memory indexing for arrays

- When we access an array element:
  - The array index indicates the amount of **offset** for accessing a memory space.
  - **`score[i]`** means to take the variable stored at "starting from **`score`**, offset by **`i`** units".

```
cout << score + 2; // 0x20c650
```

- So **`score[i]`** is **always accepted** by the compiler for any value of **`i`**.
  - Always be careful when using arrays!



Finding the array length

- Sometimes we are given an array whose size is not known by us.
- One way of finding the **array length** is to use **`sizeof`**.
  - It returns the total number of bytes allocated to that array.
- Suppose the array is named score, its length equals

```
sizeof(score) / sizeof(score[0]);
```

  - **`sizeof(score)`** is the total number of bytes allocated to the array.
  - **`sizeof(score[0])`** is the number of bytes allocated to the first element.

Finding the array length

- Example: Let's print out all elements in an array:

```
int array[] = {1, 2, 3};
int length = sizeof(array) / sizeof(array[0]);
for(int i = 0; i < length; i++)
  cout << array[i] << " ";
```

- When using **sizeof** to count the length of, e.g., an integer array:
  - Use **sizeof(a) / sizeof(a[0])**.
  - Do not use **sizeof(a) / sizeof(int)**.
- Why?

Example: finding the maximum

- How to find the **maximum** among many numbers?
- Suppose we want to write a program that:
  - Asks the user to input 10 numbers.
  - Once 10 numbers are input, prints out the maximum.

```
float value[10] = {0};
for (int i = 0; i < 10; i++)
  cin >> value[i];

// and then?
```

Example: finding the maximum

- Now the task is to find the maximum in **`value`**.
- In many cases, we write an **algorithm** to complete a task.
  - An algorithm is a step-by-step procedure that completes a given task.
- When designing an algorithm, we typically write **pseudocodes** first.
  - A description of steps in words organized in a program structure.
  - To ignore the details of implementations.
- How to find the maximum?
  - Compare the first two and find the larger one.
  - Use it to be compare with the third one.
  - And so on.

Example: finding the maximum

- One pseudocode for finding the maximum in a set is:

Given a vector A of n numbers:
for i from 0 to $n - 1$
  find the larger between A_i and A_{i+1}
  put the larger one at A_{i+1}
output A_n

- What some drawbacks of this implementation (or algorithm)?

- Implementation:

```
// value: a size-10 float array
for (int i = 0; i < 9; i++)
{
  if (value[i] > value [i + 1])
  {
    float temp = value[i + 1];
    value[i + 1] = value[i];
    value[i] = temp;
  }
}
cout << value[9];
```

Example: finding the maximum

- Let's record the current maximum at some other place:

```
float value[10] = {0};
for (int i = 0; i < 10; i++)
  cin >> value[i];

float max = value[0];
for (int i = 1; i < 10; i++)
{
  if (value[i] > max)
    max = value[i];
}
cout << max;
```

Good programming style

- It is suggested to declare a **constant** and use it to:
  - Declare an array.
  - Control any loop that traverses the array.
- Why?

```
const int VALUE_LEN = 10;

float value[VALUE_LEN] = {0};
for (int i = 0; i < VALUE_LEN; i++)
  cin >> value[i];

float max = value[0];
for (int i = 1; i < VALUE_LEN; i++)
{
  if (value[i] > max)
    max = value[i];
}
cout << max;
```

Things you cannot (should not) do

- Suppose you have two arrays **a1** and **a2**.
  - Even if they have the same length and their elements have the same type, you **cannot** write **a1 = a2**. This results in a syntax error.
  - You also **cannot** compare two arrays with **==**, **>**, **<**, etc. Why?
- **a1** and **a2** are just two **memory addresses**!
- To copy one array to another array, use a loop to copy each element **one by one**.

```
int a1[5] = {1, 2, 3, 4, 5};
int a2[5] = {0};

// a2 = a1; // error!
for (int i = 1; i < 5; i++)
{
  a2[i] = a1[i];
}
```

Things you cannot (should not) do

- Although allowed in Dev-C++, you should not declare an array with its length being a **nonconstant** variable.
  - This creates a syntax error in some compilers.
  - In ANSI C++, the length of an array must be **fixed** when it is declared.
  - To dynamically determine the array length:
- The index of an array variable should be an **integer**.
  - Some compiler allows a fractional index (casting is done automatically).

```
// DO NOT do this
int x = 0;
cin >> x;
// very bad!
int array[x];
array[2] = 3; // etc.
```

```
// Do this
int x = 0;
cin >> x;
// good!
int* array = new int[x];
array[2] = 3; // etc.
```

Outline

- More about variables
- Single-dimensional arrays
- **Multi-dimensional arrays**

Two-dimensional arrays

- While a one-dimensional array is like a **vector**, a two-dimensional array is like a **matrix** or **table**.
- Intuitively, a two-dimensional array is composed by **rows** and **columns**.
  - To declare a two-dimensional array, we should specify the numbers of rows and columns.

```
data type array name[rows][columns];
```

- As an example, let's declare an array with 3 rows and 7 columns.

```
double score[3][7];
```

Two-dimensional arrays

- `double score[3][7];`

	0	1	2	3	4	5	6
0	[0][0]	[0][1]	[0][2]				
1	[1][0]				[*x*][*y*]		
2	[2][0]						

  - **`score[0][0]`** is the 1st and **`score[0][1]`** is the 2nd. What are *x* and *y*?
- We may initialize a two-dimensional array as follows:
  - `int score[2][3] = {{4, 5, 6}, {7, 8, 9}};`
  - `int score[2][3] = {4, 5, 6, 7, 8, 9}; // 2 can be omitted.`

Example: matrix addition

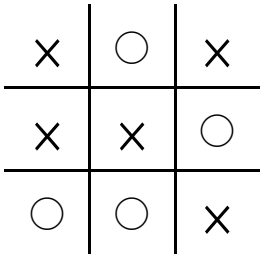
- Let's write a program to do matrix addition.

```
int a[2][3] = {{1, 2, 3}, {1, 2, 3}};
int b[2][3] = {{4, 5, 6}, {7, 8, 9}};
int c[2][3] = {0};

for (int i = 0; i < 2; i++)
{
  for (int j = 0; j < 3; j++)
    c[i][j] = a[i][j] + b[i][j];
}
```

Example: tic-tac-toe

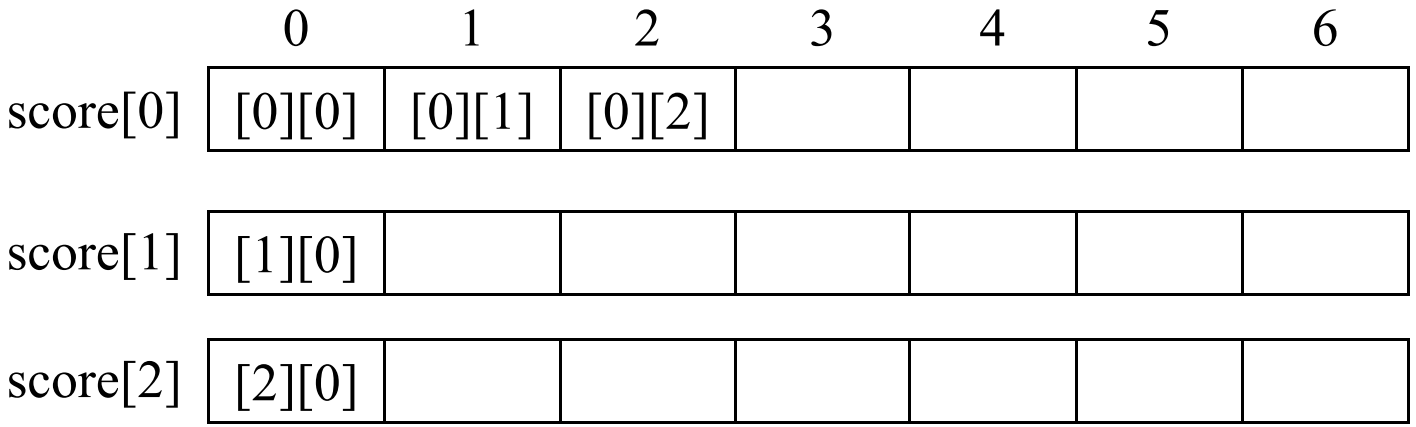
- Let's write a program to detect the winner of a tic-tac-toe game:

```
int a[3][3] = {{1, 0, 1}, {1, 1, 0}, {0, 0, 1}};

for (int i = 0; i < 2; i++)
{
  if (a[i][0] == a[i][1] && a[i][1] == a[i][2])
  {
    cout << a[i][0] << endl;
    break;
  }
}
// then check for columns and diagonals
```



Embedded one-dimensional arrays

- Two-dimensional arrays are not actually rows and columns.
- A two-dimensional array is actually **several** one-dimensional arrays.



- Try this:

```
int a[2][3];
cout << a << " " << a[0] << " " << a[1] << endl;
```

Embedded one-dimensional arrays

- `int a[2][3];`
  - `a[0][0]` is the first element.
  - `a[0][1]` is the second element.
  - `a[1][0]` is the **fourth** element.
- Two dimensional arrays are stored **linearly**.
  - And still **consecutively**.
- Try this:

```
int a[2][3];
cout << a << " " << a[0] << endl;
cout << a[1] << " " << a + 1 << endl;
cout << sizeof(a) << " " << sizeof(a[0]) << endl;
```

Address	Identifier	Value
0x20c648	a[0]	?
0x20c64c		?
0x20c650		?
0x20c654	a[1]	?
0x20c658		?
0x20c65c		?

Memory

Embedded one-dimensional arrays

- So for a two dimensional array **`score`**:
  - **`score[0]`** is the ____th one-dimensional array.
  - **`score[0][j]`** is the ____th element of the ____th one-dimensional array.
  - **`score[i]`** is the ____th one-dimensional array.
- Which description is more accurate?
  - There is an array having three rows and seven columns.
  - There is an array having three rows, each having seven elements.
- All these one-dimensional arrays must be of **the same length**.
  - Two-dimensional arrays with various row lengths can be built with pointers.

Multi-dimensional arrays

- We may have arrays with even higher dimensions.
  - `char threeDim[3][4][5];`
  - `Int eightDim[3][4][5][6][1][7][4][8];`
- Difficult to imagine and use.