

Programming Design

Variables and 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

- **Basic data types**
- Constants and casting
- Single-dimensional arrays
- Multi-dimensional arrays

Data types, literals, and variables

- Recall that in C++, each variable must have its **data type**.
 - It tells the system how to **allocate** memory spaces and how to **interpret** those 0s and 1s stored there.
 - It will also determine how **operations** are performed on the variable.
- Here we introduce **basic** (or built-in or primitive) data types.
 - Those provided as part of the C++ standard.
 - We will define our own data types later in this semester.
- Before we start, let's distinguish **literals** from **variables**.
 - Literals: items whose contents are **fixed**, e.g., 3, 8.5, and “Hello world”.
 - Variables: items whose values may **change**.

Basic data types

- The ten C++ basic data types (bytes information comes from the instructor's compiler):

Category	Type	Bytes	Type	Bytes
Integers	bool	1	long	4
	char	1	unsigned int	4
	int	4	unsigned short	2
	short	2	unsigned long	4
Fractional numbers	float	4	double	8

- Basic type names are all keywords.
- Number of bytes are compiler-dependent.

int

- **int** means an integer.
- In Dev-C++ 5.11 for Windows 10:
 - An integer uses 4 bytes to store from -2^{31} to $2^{31} - 1$.
 - **unsigned** (4 bytes): from 0 to $2^{32} - 1$.
 - **short** (2 bytes): from -32768 to 32767 .
 - **long**: the same as **int**.
- The C++ standard only requires a compiler to ensure that:
 - The space for a **long** variable \geq the space for an **int** one.
 - The space for an **int** variable \geq the space for a **short** one.
- **short** and **long** just create integers with different “lengths”.
 - In most information systems this is not an issue.

Limits of int

- The limits of C++ basic data types are stored in `<climits>`.

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

int main()
{
    cout << INT_MIN << " " << INT_MAX << "\n";

    return 0;
}
```

- For information, see, e.g., <http://www.cplusplus.com/reference/climits/>.

sizeof

- We may use the **sizeof** operator to know the size of a variable or a type.

```
cout << "int " << sizeof(int) << "\n";
cout << "char " << sizeof(char) << "\n";
cout << "bool " << sizeof(bool) << "\n";

short s = 0;
cout << "short int " << sizeof(s) << "\n";
long l = 0;
cout << "long int " << sizeof(l) << "\n";

cout << "unsigned short int " << sizeof(unsigned short) << "\n";
cout << "unsigned int " << sizeof(unsigned) << "\n";
cout << "unsigned long int " << sizeof(unsigned long) << "\n";
```

- Dev-C++ (and some other compilers) offers **long long** for 8-byte integers.

Overflow

- Be aware of **overflow**!
 - Storing a value that is “too large” to a variable.

```
int i = 0;
short sGood = 32765;

while(i < 10)
{
    short sBad = sGood + i;
    cout << sGood + i << " " << sBad << "\n";
    i = i + 1;
}
```

Overflow

看板	Hate
作者	Xtaiwansoul (.)
標題	[超] 大富翁我
時間	Sun Dec 8 06:39:37 2013

我好不容易冷靜下來

事情是這麼開始的，我玩了手機版的大富翁4fun
玩著玩著把錢夫人幹掉，金貝貝住進了醫院之後身上只剩500塊
我踩到大財神，和所有對手收2600元
正當我認為贏的那一刻我發現一件很奇怪的事情

金貝貝身上有21億!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
金貝貝身上有21億啊!!!!!!!!!!!!!!
我看了我數了兩次是真的 21億!!!
我身上一百多萬我以為我超強超威
金貝貝從街頭魯蛇突然變成三代公務員!!!!

好險我發現金貝貝不會存錢，我就開始想辦法用卡片逆轉
內心上演小宅男戰勝大鯨魚的戲碼

我用了均富卡(所有人現金平分，我拿了他十億)
還用了查稅卡(一次抽對方20%現金)

就當我辛辛苦苦的抽了幾次之後他終於剩下6億
(踩到我房子才8000塊也要用免費卡)

我查稅按下去之後 我破產!!!!!!!!!!!!!!
我破產啊!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

- Why? Why 2.1 billions?
- $2^{31} - 1 = 2147483647$.

Source: <http://disp.cc/m/tread.php?id=115-70Zv>

char

- **char** means a character.
 - Use **one byte** (0 to 255) to store English characters, numbers, and symbols.
 - Cannot store, e.g, Chinese characters.
- It is also an **integer**!
 - These characters are encoded with the **ASCII code** in most PCs.
 - ASCII = American Standard Code for Information Interchange.
 - See the ASCII code mapping in your textbook.
- Nevertheless, avoid doing arithmetic on **char**.

Example: ASCII code table 1

- This program prints out common symbols and their ASCII codes.

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

int main()
{
    for(int c = 33; c <= 126; c++)
    {
        cout << c << " ";
        char cAsChar = c;
        cout << cAsChar << "\n";
    }

    return 0;
}
```

Example: ASCII code table 2

- Here is another version.
- What is the difference?

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

int main()
{
    cout << " 0 1 2 3 4 5 6 7 8 9\n";
    cout << " 3      ";
}
```

```
for(int c = 33; c <= 126; c++)
{
    if(c % 10 == 0)
    {
        if(c / 10 <= 9)
            cout << " ";
        cout << c / 10;
    }
    char cAsChar = c;
    cout << " " << cAsChar;
    if(c % 10 == 9)
        cout << "\n";
}
return 0;
}
```

Literals in char type

- Use single quotation marks to make your **char** literal.
 - **char c = 'c';**
 - **char c = 99;**
- Some wrong ways of marking a character:
 - Wrong: **char c = "c";**
 - Wrong: **char c = 'cc';**
- More about **char** will be discussed when we talk about **casting** and **strings**.

float and double

- **float** and **double** are used to declare fractional numbers.
 - Can be **5.0**, **-6.2**, etc.
 - Can be **16.25e2** ($1.625 * 10^3$ or 1625), **7.33e-3** (0.00733), etc.
- They follow the IEEE floating point standards.
 - **float** uses 4 bytes to store values between 1.4×10^{-45} and 3.4×10^{38} .
 - **double** uses 8 bytes to store values between 4.9×10^{-324} and 1.8×10^{308} .
- Dev-C++ (and some other compilers) offers **long double** as a 16-bytes floating point data type.

Precision can be a big issue

- Consider the following program:
- Nothing special.

```
#include<iostream>
#include<cmath> // for sqrt()
using namespace std;

int main()
{
    for(int i = 0; i < 100; i++)
    {
        float f = sqrt(i);
        cout << f << " " << f * f << " ";
        cout << "\n";
    }

    return 0;
}
```


Precision can be a big issue

- How about this:
- As modern computers store values in bits, most **decimal fractional numbers** can only be **approximated**.

– 3	1	1	.	0	0	0	0
– 3.375	1	1	.	0	1	1	0
– 3.4375	1	1	.	0	1	1	1
– 3.4?							

```
int bad = 0;
for(int i = 0; i < 100; i++)
{
    float f = sqrt(i);
    cout << f << " " << f * f << " ";

    if(f * f != i)
    {
        cout << "!!!";
        bad++;
    }
    cout << "\n";
}
cout << "bad precision: " << bad;
```

Precision can be a big issue

- Let's see how big the errors are:

```
#include<iostream>
#include<cmath>
#include<iomanip> // for setprecision()
using namespace std;

int main()
{
    for(int i = 0; i < 100; i++)
    {
        float f = sqrt(i);
        cout << f << " " << setprecision(10) << f * f << " ";
        cout << "\n";
    }
    return 0;
}
```

Precision can be a big issue

- Remedy: “imprecise” comparisons.

```
if(abs(f * f - i) > 0.0001)
{
    cout << "!!!";
    bad++;
}
```

- The error tolerance can be neither too large nor too small.
 - It should be set according to the property of your own problem.
- To learn more about this issue, study *Numerical Methods*, *Numerical Analysis*, *Scientific Computing*, etc.
- In this course, we will play with only integers for most of the time.

bool

- A **bool** variable uses 1 byte to record one Boolean value: true or false.
 - Two literals: **true** and **false**.
 - 7 bits are wasted.
 - All non-zero values are treated as true.
- **bool** variables play an important role in control statements!

```
bool b = 0;
cout << b << "\n";

b = 1;
cout << b << "\n";

b = 10;
cout << b << "\n";

b = 0.1;
cout << b << "\n";

b = -1;
cout << b << "\n";
```

Outline

- Basic data types
- **Constants and casting**
- 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** \geq **int** \geq **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:

```
int grade1 = 0, grade2 = 0;  
cin >> grade1 >> grade2;  
cout << (grade1 + grade2) / 2;
```

- The **division** operator returns an integer if both operands (numerator and denominator) are integers.
- How to get our desired results?
 - If appropriate, we may change the data types of the operands.

```
double grade1 = 0, grade2 = 0;
```

- If not appropriate, we may cast the operands **temporarily**.

Casting for division

- Which one works?

```
int grade1 = 0, grade2 = 0;  
cin >> grade1 >> grade2;  
cout << static_cast<float>((grade1 + grade2) / 2);
```

```
int grade1 = 0, grade2 = 0;  
cin >> grade1 >> grade2;  
cout << static_cast<float>(grade1 + grade2) / 2;
```

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

Casting a character to an integer

- Try to explain the following program:

```
char c = 254;  
int a = 10;  
cout << c + a; // 8. Why?
```

- Avoid doing arithmetic on **char**.

Outline

- Basic data types
- Constants and casting
- **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**.

```
cin >> score[2];
```

```
cout << score[2] + score[3];
```

An array is a type

- Arrays are often used with loops.
 - Quite often the loop counter is used as the array index.
- An array is also a (nonbasic) **type**.
 - The type of **score** is an “integer array” (of length 5).
 - What is this?
- We will go back to this when we introduce pointers.
 - For now, just treat an array as a sequence of variables.

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

```
cout << score;
```

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 $4 \text{ bytes} \times 5 = 20$ **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 daysInMonth1[12] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
int daysInMonth2[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
cout << sizeof(daysInMonth2); // 4 * 12 = 48
int daysInMonth3[12] = {31, 28, 31}; // nine 0s
int daysInMonth4[3] = {1, 2, 3, 4}; // error!
```

- To initialize all elements to 0:

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

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

Example: inner product

- This program calculates the inner product of two given 4-dimensional vectors.
- Do these exercises at home:
 - Modify the program to allow a user to decide the values of the two vectors.
 - Write a program that calculate the sum of two vectors.

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

int main()
{
    int a[4] = {1, 2, 3, 4};
    int b[4] = {4, 3, 2, 1};

    int ip = 0;
    for(int i = 0; i < 4; i++)
        ip += a[i] * b[i];
    cout << ip << "\n";

    return 0;
}
```


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 (Hexadecimal)
```

Address	Identifier	Value
0x20c648	score	?
0x20c64c		?
0x20c650		?
0x20c654		?
0x20c658		?

Memory

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!

Address	Identifier	Value
0x20c648	score	?
0x20c64c		?
0x20c650		?
0x20c654		?
0x20c658		?

Memory

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?
- We want to write a program that:
 - Asks the user to input 10 numbers.
 - Once 10 numbers are input, prints out the maximum.
- 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.

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

Example: finding the maximum

- Let's record the current maximum at **max**:

```
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 that 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 = 0; i < 5; i++)
{
    a2[i] = a1[i];
}
```

Things you cannot (should not) do

- Although allowed in Dev-C++, you should not declare an array whose length is **nonconstant**.
 - 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:
 - We will talk about this a few weeks later.
- The index of an array variable should be **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

- Basic data types
- Constants and casting
- 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 ?

Two-dimensional arrays

- We may initialize a two-dimensional array as follows:

```
int score1[2][3] = {{4, 5, 6}, {7, 8, 9}};  
int score2[][3] = {4, 5, 6, 7, 8, 9}; // 2 can be omitted.
```

```
int score3[2][3] = {{4, 5}, {7, 8, 9}};  
cout << score3[0][2]; // 0
```

```
int score4[2][3] = {4, 5, 7, 8, 9};  
cout << score4[0][2]; // 7
```

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
```

X	O	X
X	X	O
O	O	X

Example: matrix additions

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

```
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++) // matrix addition
    for(int j = 0; j < 3; j++)
        c[i][j] = a[i][j] + b[i][j];

for(int i = 0; i < 2; i++) // print out c
{
    for(int j = 0; j < 3; j++)
        cout << c[i][j] << " ";
    cout << "\n";
}
```


Example: matrix multiplications

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

```
int a[2][3] = {1, 1, 1, 2, 2, 2};  
int b[3][4] = {1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3};  
int c[2][4] = {0};  
  
for(int i = 0; i < 2; i++)  
    for(int j = 0; j < 4; j++)  
        for(int k = 0; k < 3; k++)  
            c[i][j] += a[i][k] * b[k][j];  
  
// print out c
```

Embedded one-dimensional arrays

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

	0	1	2	3	4	5	6
score [0]	[0][0]	[0][1]	[0][2]				
score [1]	[1][0]						
score [2]	[2][0]						

- 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] << "\n";
cout << a[1] << " " << a + 1 << "\n";
cout << sizeof(a) << " " << sizeof(a[0]) << "\n";
```

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[i][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 (but hard to use).
 - `int threeDim[2][3][4];`
 - This is an array of $2 \times 3 \times 4 = 24$ integers.
 - They together occupies $24 \times 8 = 192$ bytes (in a continuous space in the memory).
 - `threeDim` is still the address of the first element `threeDim[0][0][0]`.

Address	Identifier	Value
0x20c600	<code>threeDim[0][0][0]</code>	?
0x20c604	<code>threeDim[0][0][1]</code>	?
⋮		⋮
0x20c65c	<code>threeDim[1][2][3]</code>	?

Memory

Multi-dimensional arrays

threeDim

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

threeDim[1]

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

threeDim[1][1]

[1][1][0]	[1][1][1]	[1][1][2]	[1][1][3]
-----------	-----------	-----------	-----------