CPT106

C++ Programming and Software Engineering II

Lecture 6 Arrays and Pointers

Dr. Xiaohui Zhu

Xiaohui.zhu@xjtlu.edu.cn

Room SD535

Outline

- Array
  - Array, multidimensional array
  - Pointers pointing to arrays
- Vector
  - Using vector
  - Passing vectors to functions
- Pointer
  - Pointer pointing to an object
  - **this** pointer
- Dynamic memory allocation
  - new and delete

1.1 Fundamentals of Array

- Why we need arrays?
- An *array* is a series of elements of the same type placed in continuous memory locations that can be individually referenced by adding an index to a unique identifier.
  - Example:

```
double dArr[10];
int iArr[]={1,2,3,4,5};
```

  - Addressing the elements of the array by its index:
    - The indices of the elements are numbered from 0 to 4;
    - In arrays the first index is always 0, independently of its length.

```
dArr[0] = 1.5;
b = iArr[1]+2;
```

1.1 Fundamentals of Array

- Declaration
  - Syntax:

```
data_type array_name[number_of_elements];
```

  - Example:

```
1.    double dArr[5];              // VALID
2.    int number; cin >> number;
      char student[number];        // INVALID
3.    const int SIZE=20;
      char name[SIZE];             // VALID
```

  - Notice: "**number_of_elements**" must be a constant value

1.1 Fundamentals of Array

- Initialisation
  - Syntax: **`type arr[number]={value1, value2, ...};`**
  - Example:

```
1.    int array1[3]={1,2,3};   //VALID
2.    int array2[3]={1,2,3,4};//INVALID
3.    int array3[3]={1, ,3};   //INVALID
4.    int array4[3]={1,2};     //VALID
5.    int array5[ ]={1,2,3};   //VALID
```

  - Rules:

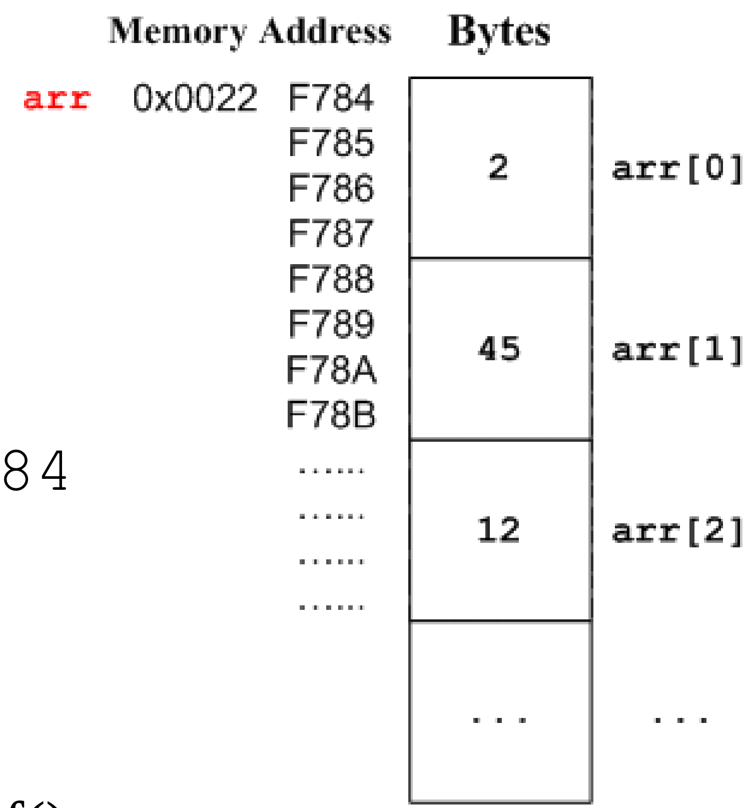
1. number of elements cannot exceed the dimension of array
2. assignment of values cannot jump
3. the omitted elements will be initialised to 0 by default
4. the dimension of the array can be omitted, and automatically assigned by counting the elements number

1.1 Fundamentals of Array

- Arrays in memory
  - Arrays are stored in memory as continuous drawers
  - Example:

```
int arr[]={2,45,12,6,23};
```

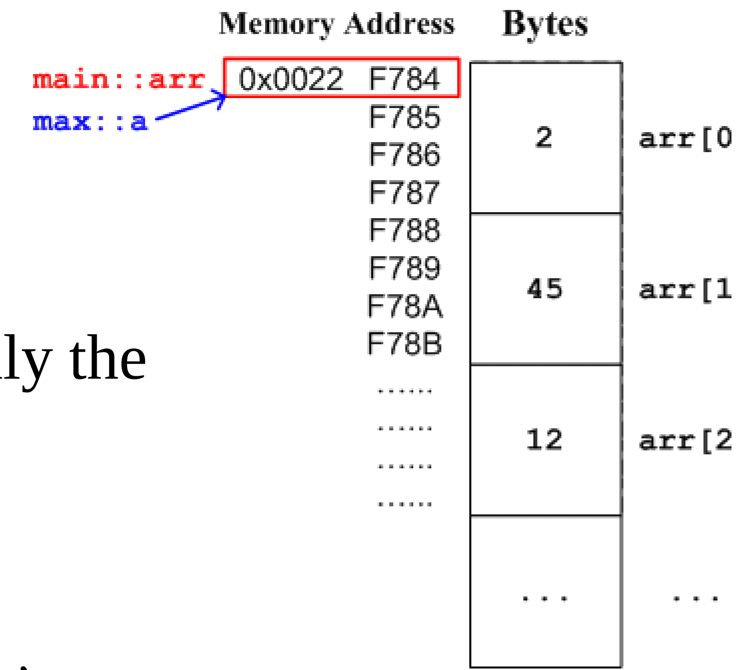
  - The value stored in **`arr`** is `0x0022F784 (hexadecimal number)`, the memory address of the first element.



  - To get the length of the array, use sizeof()

```
int size=sizeof(arr)/sizeof(int);
```

1.1 Pass Array to functions



- Pass array as argument
  - When pass an array to a sub-function, only the address of the first element is passed.
  - Therefore, the array length also has to be passed to the function:

    `max(arr, sizeof(arr)/sizeof(int))`

  - In the sub-function header, the array name **`a`** is followed by **`[]`**, to show it is an array rather than a normal integer.

    `int max(int a[], int size){ ...... }`

  - The value stored in **`a`** (in sub-function) is the same as **`arr`** (from main function), which is the memory address **`0x0022F784`**.
  - Modification of **`a`** in sub-function will influence the values in **`arr`**, which is in the calling function. (why?)

1.2 Multi-dimensional Array

arr2D [0][0]	arr2D [0][1]
arr2D [1][0]	arr2D [1][1]
arr2D [2][0]	arr2D [2][1]

- Declaration:
  - Syntax: **`data_type array_name[num_rows][num_columns];`**
  - Example: **`int arr2D[3][2];`**
- Initialisation: by sequence, or by row
  - Example:

```
1. int arr2D[3][2]={4,2,5,6};
2. int arr2D[3][2]={{4,2},{5},{6}};
```

4	2
5	6
0	0

By sequence

4	2
5	0
6	0

By row

- In memory, they are stored in continuous memory blocks row by row

Memory Address		
0x0022 F784	arr2D [0][0]	4
F788	arr2D [0][1]	2
F78C	arr2D [1][0]	5
F790	arr2D [1][1]	6
F794	arr2D [2][0]	...
F798	arr2D [2][1]	...

- Initialisation: Omit the num_row

```
3. int arr2D[2][2]={4,2,5,6};
4. int arr2D[3][2]={4,2,5,6,0,0};
5. int arr2D[2][3]={4,2,5,6,0,0};
```

4	2
5	6

4	2
5	6
0	0

4	2	5
6	0	0

1.2 Pass Multi-dimensional Array to functions

- Pass 2D array as argument
  - When pass an array to a sub-function, only the address of the first element is passed.
  - Similar to 1D array, the array dimensions have to be passed to the function (both **num_rows** and **num_columns**).
    - Example:

```
void disp(int a[3][2], int r, int c);        // VALID
```

  - The dimensions of the array should be announced in the function header:

```
void disp(int a[ ][2], int r, int c);        // VALID
void disp(int a[3][ ], int r, int c);        // INVALID
void disp(int a[ ][ ], int r, int c);        // INVALID
```

    - Omitting the **num_rows** is OK, but **num_columns** must be given.

1.3 Pointer points to an array

- An array name stores the address of the first element of the array, which can be regarded as a pointer
  - For 1D array:

```
int a[]={1,2,3,4,5};
int *ipa;
ipa=a;          // VALID
ipa=&a;         // INVALID
ipa=&a[0];      // VALID
```

  - For multi-dimensional array (2D):

```
int x[3][4] = {{10,20,30,40},
{50,60,70,80}, {90,100,110,120}};
int *p;
p=x;            // INVALID
p=&x[0];        // INVALID
p=&x[0][0];     // VALID
```

1.3 The pointer points to a 2D array

Array name – Initial address of the array:
int x[3][4] = {{…}, {…}, {…}};

Array pointer - A pointer points to an array:
int *p;
p=&x[0][0];

0012FF40 10	0012FF44 20	0012FF48 30	0012FF4C 40
0012FF50 50	0012FF54 60	0012FF58 70	0012FF5C 80
0012FF60 90	0012FF64 100	0012FF68 110	0012FF6C 120

Array Name: x

```
x = 0012FF40          // first row
*x = 0012FF40         // first element
**x = 10
*(*x+1) = 20
x+1 = 0012FF50        // second row
*(x+1) = 0012FF50     /* first element of the second row */
**(x+1) = 50
*(*(x+1)+1)=60
```

Array pointer: p=&x[0][0]

```
p = 0012FF40
*p = 10
p+1 = 0012FF44
*(p+1) = 20
*(p+5) = 60

**p                   // INVALID
**p+1                 // INVALID
*(*p+1)               // INVALID
```

2. Vector

- For traditional array, the array size must be determined during the declaration or initialisation stage.
  - Eg:
    ```
    1. int iArr[10];            // dimension given by constant
    2. int num[]={1,2,3}        // dimension is 3
    3. int size = 5;
       double dNum[size];       // INVALID, cannot be variable
    ```
  - And after the declaration and initialisation, the dimension of the array can no longer be changed.
- However, it is very often that the number of elements need to be determined during the execution, or we want to add another element to the array.
- ----------------- How ?   => using “vector”

2.1 Fundamental of Vector

- To use vector, include <vector> library at the beginning
- Declaration:
  - Syntax: **`vector<data_type> name(size);`**
  - Eg: 1. vector<int> iArr(10); // a vector with 10 elements

    2. int size = 5; // create with variable length

    vector<Complex> cNum(size); // is also VALID

    3. vector<int> empty; // VALID as an empty vector
- Initialisation:
  - Cannot be initialised as normal array

iVec	1	1	1	1	1

  - Eg: 1. vector<int> initial(3)={1,2,3}; // INVALID

    2. vector<int> iVec(5, 1); // VALID, 5 elements are all 1

    3. int arr[]={1, 2, 3};

    vector<int> iniTial(arr, arr+3); // VALID

2.1 Fundamental of Vector

- Accessing:
  - The same as normal array, using [] and index
  - Eg: 1. iVec[0]=10; // the first element is assigned to 10
    2. iVec.at(0)=10; // this way is also valid
  - **`Vec_name.front()`** returns the first element of the vector
  - **`Vec_name.back()`** returns the last element of the vector
  - Eg: 3. cout << iVec.front();
    4. iVec.front() = iVec.front() – iVec.back();
- Get the size
  - Get the number of elements in the vector container (**`unsigned int`**)
  - Eg: 1. cout << iVec.size();
    2. for (unsigned int i=0; i<iVec.size(); i++) {...}

2.2 Manipulating Vector

- Add element at the end: **push_back()**
  - Eg: int newInt;

    cin >> newInt; // input "5"

    iVec.push_back (newInt);

iVec	1	1	1	1	1	
iVec	1	1	1	1	1	5

- Delete element at the end: **pop_back()**
  - Eg: iVec.pop_back ();

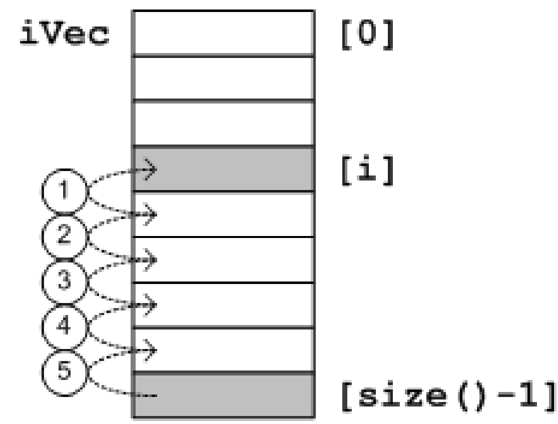
iVec	1	1	1	1	1
iVec	1	1	1	1	

  - The pop_back function does not return the element that is being removed. To know what that element is, you need to capture it first.
  - Eg: double last = iVec.back();

    iVec.pop_back ();

2.2 Manipulating Vector

- To remove an element in an ordered vector:
  – Move all elements following the element to be removed down (to a lower index) by one slot;
  – then delete the last element.



  – Code:

```
for (int i=pos; i<values.size()-1; i++)
      values[i]=values[i+1];
values.pop_back();
```

Consider how to insert an element to a specific position without influencing other elements' order.

2.3 Pass Vector to functions

- Passing a vector into a function **by value**
  - The function makes a local copy of all the content of the vector
  - Syntax:

```
return_type  func_name (vector<type> vec_name)
```

  - Example: a function that computes the average of all elements

```
double average(vector<double> values)
{   double sum = 0;
    for (unsigned int i=0; i<values.size(); i++)
        sum = sum + values[i];
    return sum/values.size();
}
```

```
int main()
{   double test[5]={1.2,-2.4,3,4.02,0};
    vector<double> value(test, test+5);
    cout <<"The average is "<<average(value)<<endl;
    return 0;
}
```

2.3 Pass Vector to functions

- Passing a vector into a function **by reference**
  - The function can modify the content of the vector
  - Syntax:

```
return_type  func_name (vector<type>& vec_name)
```

  - Example: a function that raises all elements by the given rate

```
void raise(vector<double>& values, double rate)
{    for (unsigned int i=0; i<values.size(); i++)
         values[i]=values[i]*(1+rate/100);
         // rate is in percentage
}
```

```
int main()
{    vector<double> savings(5);
     for (unsigned int i=0; i<savings.size(); i++)
         savings[i]=i*200;    // initialise
     raise(savings,5.2);      // call the function
     return 0;
}
```

2.3 Pass Vector to functions

- Use vector as a function **return value**
  - The function can return a vector
  - Syntax: `vector<type> func_name (parameter list)`
  - Example: a function that collects all values that fall within a range

```
vector<double> between(vector<double> values, double low, double high)
{   vector<double> result;
    for (unsigned int i=0; i<values.size(); i++)
        if(low<=values[i] && values[i]<=high)
            result.push_back(values[i]);
    return result;
}
```

```
int main()
{   ... // initialise a test vector
    vector<double> ranged = between(values, 1, 3);
    ... // output the values for check
    return 0; }
```

3.1 Pointer points to structure

- Pointer can be used to point to a structure

```
struct movies_t
{
  string title;
  int year;
};
```

  - Declare the pointer for **movies_t** type of structure

```
movies_t amovie;
movies_t * pmovie;
pmovie = &amovie;
```

  - the arrow operator (->) is a dereference operator that is used to access a member of an object to which we have a reference.

```
pmovie->title
(*pmovie).title
```

3.2 Pointer points to object

- Similarly, pointer can also be used to point to an object

```
class complexClass
{
  double x;
  double y;
public:
  double getX(){return x;}
  double getY(){return y;}
  void set(double a, double b){x=a;y=b;}
  Void add(complexClass *a)
  {    a->x = a->x + x;
       a->y = a->y + y;}
};
```

  – the arrow operator (->) performs the same as for structure pointer.

```
a->x
(*a).x
```

3.3. `this` Pointer

- **`this`** is a pointer that points to the object for which this function was called.
  - For example, function **`cNum.divide()`** will set the pointer **`this`** to the address of the object **`cNum`**.
  - In a method of a class, to call another method of the same class, use "**`this->method_name()`**"
  - For example:

```
complexClass complexClass :: complexClass(int x, int y)
{    this->x=x;
     this->y=y;
     double abs = this->abs()/sqrt(pow(divider.x,2)+pow(divider.y,2));
     double angle = this->angle()-atan2(divider.y,divider.x);
     result.x = abs*cos(angle);
     result.y = abs*sin(angle);
     return result;
}
```

4. Dynamic Memory Allocation

- Problem:
  - The number of variables in a programme cannot be fixed.
  - It is often necessary to use arrays with variable number of elements in a program depending on the number of input data.
- Limitation of using simple data type
  - The number of elements of an array must be constant.
  - The number of variables cannot be changed during programme execution.

```
int size=10;
float arr1[size];
```

X

```
const int size=10;
float arr1[size];
```

√

- **Solution:** Dynamic memory allocation using pointers

4.1 Allocation of Memory Address

- The operator **new** can allocate memory for a particular type of data and return the memory address.
- Syntax:

```
new datatype
```

  – Example:

```
double* pd1;
pd1 = new double;  *pd1 = 100.456;
```

  – At this point, a double variable has been dynamically allocated, but its name is unknown.

  – This variable can only be accessed through a pointer.

4.2 Dynamic Allocation of an Array

- Syntax:

```
double* pd2 = new double[n];
```

- Here n is an integer variable. Its value can be changed during the execution of the program.

- Example:

```
*pd2=1000.02;              // first element of array
*(pd2+1)=2435.08;          // second element of array
*(pd2+n-1)=43552.55;       // nth element
```

4.3 Dynamic Allocation of an Object

- Syntax:

```
double* pd2 = new className;
```

- Example:

```
Complex *ptr1 = new Complex;
```

  – To access, use "->" as for normal object pointer

```
ptr1-> display();
ptr1-> set(5,10);
```

  – It is also possible to create an array of classes using pointer

```
int N=5;
Complex *ptr2 = new Complex [N];
ptr2[0].show();
(ptr2+1)->set(5,10);
```

Enough Memory Space ?

- **Return of the expression:**
- **`pd2 = new double[n]`**

0	if no memory space
non-zero	if allocation successful

- **To check**

```
if(!(pd2 = new double[n]))
{
    cout<<"not enough memory space"<<endl;
    exit(1);
}
```

4.4 De-allocation of Memory Address

- The operator **delete**
- When a memory address is dynamically allocated by the **new** operator, no other program can make use of this address.
- It can only be de-allocated by the operator **delete**.

```
double* pd1= new double;
// To de-allocate a single variable
delete pd1;

double* pd2 = new double[n];
// To de-allocate an array
delete[ ] pd2;
```

An Example

```
#include<iostream>
using namespace std;
int main(void)
{
   int num;
   cout << "Type in the number of real values you are going to
input from the keyboard" << endl;
   cin >> num;  // The number of values to be input

   float *pvalue = new float [num];   //Allocate float array
   for (int i=0; i<num; i++)
   {
      cout<<“Key in value”<<(i+1)<<“ Please : ” << endl ;
      cin>>*(pvalue+i);  // Read in the ith value
   }
   ......
   delete [ ] pvalue;  //De-allocate memory after use
}
```

Summary

- In OOP, use **`new`** to dynamically allocate memory during processing the program
- The size of the dynamic memory allocation can be a variable (with value)
- The memory allocated by **`new`** can only be used by the specified program, so it must be de-allocated by **`delete`** before the end of the program. Otherwise, it will cause memory leak
- For dynamic array, the format of delete must be **`delete []`**. Otherwise, it still causes memory leak.