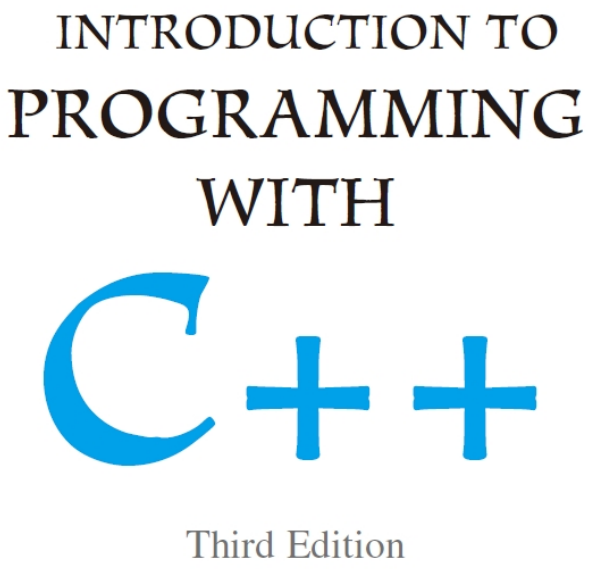
Introduction to Programming with C++

Chieh-Sen (Jason) Huang

Department of Applied Mathematics

National Sun Yat-sen University



Contents are based on book by Y. Daniel Liang

Array

- C++ and most other high-level languages provide a data structure, the array, which stores a fixed-size sequential collection of elements of the same type.

- An array is used to store multiple values of the same type. An element in an array can be accessed using an index.

```
elementType arrayName[SIZE];
```

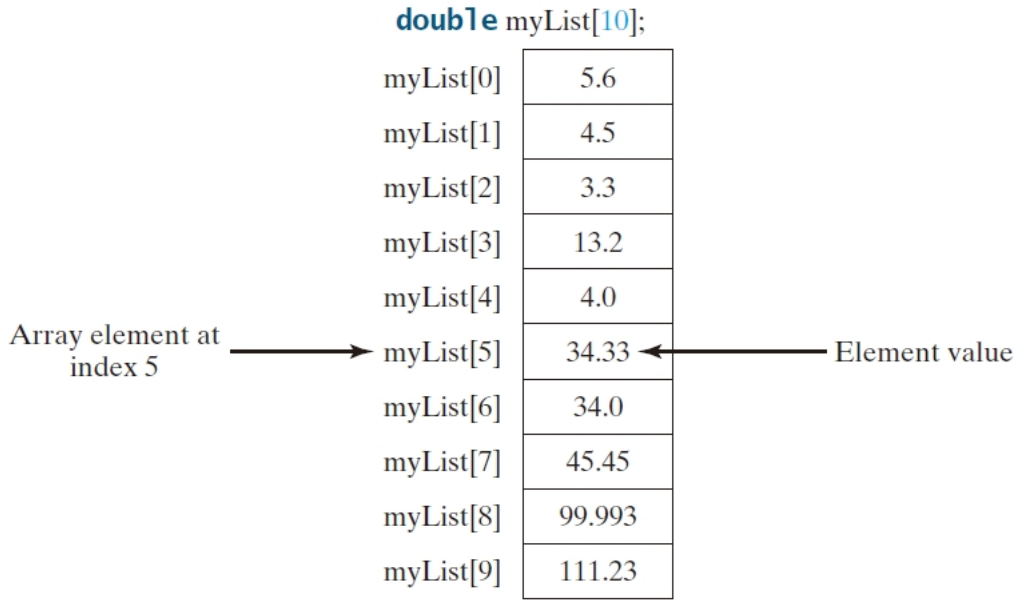
  1. The elementType can be any data type, and all elements in the array will have the same data type.
  2. The SIZE, known as array size **declarator**, must be an expression that evaluates to a constant integer greater than zero.

```
double myList[10];
```

- The compiler allocates the space for 10 double elements for array myList.

Array

```
myList[0] = 5.6;
myList[1] = 4.5;
myList[2] = 3.3;
myList[3] = 13.2;
myList[4] = 4.0;
myList[5] = 34.33;
myList[6] = 34.0;
myList[7] = 45.45;
myList[8] = 99.993;
myList[9] = 111.23;
```



- The array size used to declare an array must be a constant expression in standard C++.

```
int size = 4;
double myList[size]; // Wrong

const int SIZE = 4;
double myList[SIZE]; // Correct
```

Array

- The array elements are accessed through the integer index.
- Array indices are 0-based; that is, they run from 0 to arraySize-1.
- The first element is assigned the index 0. This is the C.S. curse.
- We, mathematicans, like to count thing from 1.
- Accessing array elements using indexes beyond the boundaries (e.g., myList$[-1]$ and myList$[10]$) causes an out-of-bounds error.
- Out of bounds is a serious error. C++ will simply access the content in that memory location. Unfortunately, the C++ compiler does not report it.

```
for (int i = 0; i < 10; i++){
  myList[i] = i;
}
```

Array Initializers

- C++ has a shorthand notation, known as the array initializer, which declares and initializes an array in a single statement.

```
elementType arrayName[arraySize] = {value0, value1, ..., valuek};
```

e.g.

```
double myList[4] = {1.9, 2.9, 3.4, 3.5};
```

- C++ allows you to omit the array size when declaring and creating an array using an initializer.

```
double myList[] = {1.9, 2.9, 3.4, 3.5};
```

Processing Array

- When processing array elements, you will often use a for loop.

```
int i;
int ary[5];

cout<< "Please input 5 integers:\n";
for(i = 0; i < 5; i++){
  cout << "No." << i + 1 << ": ";
  cin >> ary[i];
}

int sum = 0;       //sum
for(i = 0; i < 5; i++)
  sum += ary[i];
cout<< "The sum is " << sum <<"\n";
```

Processing Array

```
int ninzu = 0 ;  //number of failing students. List7_2a.cpp
int tensu[6];   //score array
int rakudai[6];  //index of failing students
```

```
  cout << " Please input 6 grades: \n";
  for( i = 0 ; i < 6 ; i++){
    cout << "No " << i+1 << " "; //There is no number "0"
    cin >> tensu[i];

    if( tensu[i] < 60){
      rakudai[ninzu] = i; //record the index
      ninzu++;
    }
  }

  cout<< " There are " << ninzu
       << " students who failed. They are \n";

  for( i = 0 ; i < ninzu ; i++)
   cout << "NO: " << rakudai[i] + 1 << " with grade "
        << tensu[rakudai[i]] << "\n";
```

```
tensu[] = {65, 12, 45, 62, 99, 3}
rakudai[]={1, 2, 5}
```

```
tensu[rakudai[0]] = 12, tensu[rakudai[1]] = 45, tensu[rakudai[2]] = 3
```

Passing Arrays to Functions

- C++ uses pass-by-value to pass array arguments to a function.
- There are important differences between passing the values of variables of primitive data types and passing arrays.
- For an argument of a primitive type, the argument's value is passed.
- For an argument of an array type, the value of the argument is the **starting memory address** to an array; this value is passed to the array parameter in the function.
- Semantically, it can be best described as **pass-by-sharing**, that is, the array in the function is the same as the array being passed.
- Thus, if you change the array in the function, you will see the change outside the function.

Passing Arrays to Functions

- `int t[]` specifies that the parameter is an integer array of any size.

- Therefore, you can pass any integer array to invoke this function.

- Normally you should also pass its size in another argument, so that the function knows how many elements are in the array. Otherwise, you will have to hard code this into the function.

```
// List7_3a.cpp
   double ans = avg(test);// Invoke the function
   cout << "The average of the grades is " << ans << "\n";

double avg(int t[])
{
   double sum = 0;
   for(int i = 0; i < 5; i++){
     sum += t[i];
   }
   return sum/5;
}
```

- We could also pass pointer `*t` using `double avg(int *t)`.

Returning Arrays from Functions

- Can you return an array from a function using a similar syntax?

```
// Return the reversal of list
int[] reverse(const int list[], int size)
```

- This is not allowed in C++.

- You can circumvent this restriction by passing two array arguments in the function:
  `void reverse(const int list[], int newList[], int size)`

- See, List7_7.cpp.

- Bubble sort, see List7_4a.cpp.

- Move to Chapter 11.