ST. CHARLES COMMUNITY COLLEGE

CPT-182 - Evening - Programming in C++

Lecture 3

Array, Vector, Iterator, String

Dayu Wang

- **Arrays**

  → Sometimes a list of data items should be stored.

  Store 200 students in 200 different variables would be ridiculous.

  → An array is a special variable having one name, but storing a list of data items, with each item being directly accessible.

  → Each item in an array is known as an element.

- **Declaration of Arrays**

```
1  int arr[5];  // Declare an array that stores 5 integers.
```

→ Syntax of declaring an array:

`Item_Type Var_Name[Num_of_Items];`

1) In C++, all the elements in an array have to be the same type.
2) You cannot alter the type of elements in an array after declaration.
3) The `Num_of_Items` must be a literal or const variable.

```
1  int num_of_items = 3;
2  int arr[num_of_items];  // (Error) You cannot put a variable in the [].
```

4) The size of the array (number of elements in the array) cannot be changed after declaration.

→ What if you would like to append an item to the end of the array?

You cannot do it for a regular array in C++.

- How arrays are stored in the memory?

→ Items in an array are physically connected in the memory.

```
1  int arr[5];
```

Memory Address	...	1000	1004	1008	1012	1016	...
Data		first	second	third	fourth	fifth	

Each integer requires 4 bytes to be stored in the memory.

→ The memory address of the elements are continuous in the memory.

→ Identifier arr will point to the first memory address of the array.

In this case, arr points to memory address 1000.

- How to initialize an array?

```
1  int arr[5] = { 1, 5, 3, -2, 28 };
```

→ Use {}, list all the items in the array separated by comma.


```
1  int arr[5] = { 1, 5, 3, -2 };
```

→ If the number of items given is less than the size of the array...

Memory Address	...	1000	1004	1008	1012	1016	...
Data		1	5	3	−2	dummy	

Items fill the first four memory cells.

```
1  int arr[] = { 1, 5, 3, -2 };
```

→ If you already listed all the items in the array, then the size of the array is optional. But the [] is a must.

arr will store 4 integers, which are 1, 5, 3 and −2.

```
1  int arr[3] = { 1, 5, 3, -2 };
```

→ Code will not compile if you give more items than the size of the array.

- Accessing Elements in Arrays

→ In an array, each element's position number is called the index.

→ The index enables direct access to any element.

```
1  int arr[4] = { 1, 5, 3, -2 };
```

Index	0	1	2	3
Value	1	5	3	−2

`arr[0]` will return 1.

`arr[1]` will return 5.

`arr[2]` will return 3.

`arr[3]` will return −2.

→ For an array of size `n`, its indices are from `0` to `n - 1`.

- **How to update an item in the array?**

```
1  int arr[5] = { 1, 5, 3, -2, 18 };
2  arr[1] = 10;
```

Memory Address	...	1000	1004	1008	1012	1016	...
Data		1	10	3	−2	18	
		arr[0]	arr[1]	arr[2]	arr[3]	arr[4]	

→ C++ arrays are mutable.

Data stored in memory address 1004 will be changed to 10.

- **What is the problem of the following code?**

```
1  int arr[] = { 1, 3, 5, 7, 9 };
2  arr[5] = 12;
```

No ~~Error~~

→ `[]` is called subscript operator.

→ For regular arrays, `[]` does not ~~check for boundaries~~.

- **How the subscript operator is executed?**

Memory Address	...	1000	1004	1008	1012	1016	...
Data		1	3	5	7	9	

→ `arr` points to memory address 1000.

When `arr[2]` is referenced, the system computes 1000 + 2 × 4 = 1008.

Then, it returns the value stored in memory address 1008.

→ When `arr[x]` is referenced, the system returns the value stored in memory address $1000 + 4x$.

When `arr[5]` is referenced, the system returns the value stored in memory address 1000 + 5 × 4 = 1020, even though the memory address is out of the bounds of the array.

Even `arr[1000]` will result in no ~~error~~ (in theory).

Memory Address	...	1000	1004	1008	1012	1016	...
Data		1	5	3	−2	18	
		arr[0]	arr[1]	arr[2]	arr[3]	arr[4]	

→ **What `arr[-3]` returns?**

In Python, negative index means counting from right to left.

There is no such mechanism in C++.

→ **`arr[-3]` will return the value stored in memory location 1000 + 4 × (−3) = 988, even though it is out of the boundaries of the array.**

- **If `arr` is an array, how to get the size of the array?**

In Python, we can use `len(arr)` to get the number of items in the list.

→ **In C++, there is no way to get the size of a regular array.**

→ **Therefore, you must create another variable to store the size of the array when you declare an array.**

```
1  int arr[5] = { 1, 2, 3, 4, 5 }, num_of_items = 5;
```

Every time when you need the size of the array, you need to reference `num_of_items`.

Since the size of a regular array cannot be changed once declared, the `num_of_items` is essentially a constant.

- **How to output an array?**

```
1  int arr[5] = { 1, 2, 3, 4, 5 };
2  cout << arr << endl;
```

```
Console  006FFDE4
```

→ **If you just use `cout << arr`, then it will output the memory address (hex number) of the beginning of the array.**

→ **Therefore, to output the array, you have to use loops.**

```
1  int arr[5] = { 1, 2, 3, 4, 5 };
2  for (int i = 0; i < 5; i++) {
3      cout << arr[i] << endl;
4  }
```

```
Console  1
         2
         3
         4
         5
```

→ **[Exercise] Suppose `arr[5] = { 1, 2, 3, 4, 5 }`, please write a program to output `arr` as the format `[1, 2, 3, 4, 5]`.**

```
1  int arr[5] = { 1, 2, 3, 4, 5 };
2  cout << '[';
3  for (int i = 0; i < 5; i++) {
4      cout << arr[i];
5      if (i != 4) { cout << ", "; }
6  }
7  cout << ']';
```

```
Console  [1, 2, 3, 4, 5]
```

- **What are the limitations of a regular C++ array?**

  → Size must be a constant at declaration.

  → Size cannot change after declaration.

  → Another variable is required to store the size of an array.

  → Operator [] does not ~~check for boundaries~~.

  → Besides [], there is no other operations on an array.

- **Vectors**

  → In C++, vector is an advanced data type for arrays.

  1) Element access/update using index is also supported.
  2) Boundary checking takes place when using the .at() function to access elements.
  3) Size can be changed dynamically at runtime.
  4) There is not a need to ~~use another variable~~ to store the size.
  5) Besides [], more operations can be performed on vectors.

  → Data type of the vector elements cannot be changed once the vector is declared.

- **Declaration of Vectors**

  → You must `#include <vector>` to use vectors.

  → `vector` is not ~~a primitive data type~~ in C++.

  Only integer, floating-point, and boolean types are primitive data types.

  Arrays, strings, vectors are not primitive data types.

```
1  vector<int> v1;  // Declare a vector "v1".
2  vector<int> v2(3);  // Declare a vector with size 3.
3  vector<int> v3(5, 0);  // A vector (size 5) with all elements to be 0
```

  → To declare a vector of some data type `Item_Type`, the syntax is:

  ```
  vector<Item_Type> Var_Name;
  ```

  1) There is no ~~[]~~ in vector declaration.
  2) If size is not specified at declaration, the vector has size **0** (empty).

  → If you want to specify size of the vector at declaration, syntax is:

  ```
  vector<Item_Type> Var_Name(Size);
  ```

  1) The `Size` needs to be placed in (), not ~~[]~~.
  2) The `Size` can be either a literal or a variable.

→ If you want to specify size and initialize all the elements in the vector at declaration, syntax is:

```
vector<Item_Type> Var_Name(Size, Initial_Value);
```

All the elements in the vector will be initialized with `Initial_Value`.

→ Vectors also support initialization similar to regular arrays.

```
1  vector<int> vec = { 1, 2, 3 };
```

→ However, the following are wrong.

```
1  vector<int> vec1(3) = { 1, 2, 3 };
2  vector<int> vec2(4, 0) = { 0, 0, 0, 0 };
```

If you want to use '{}' to initialize a vector, you cannot use () to give any information, even for a perfect match.

- **Access/Update Elements in Vectors**

→ There are two ways to reference an element in a vector.

1) Use the [] operator.

2) Use the .at() class-member function (recommended).


```
1  vector<int> vec = { 1, 5, 3, -2, 18 };
2  vec[1] = 10;
3  vec.at(3) = 2 * vec.at(1);
```

Memory Address	...	1000	1004	1008	1012	1016	...
Data		1	10	3	20	18	
		vec[0]	vec[1]	vec[2]	vec[3]	vec[4]	

In C++, [] does not ~~check for boundaries~~, even for vectors.

The .at() class-member function does check for boundaries.

- **How to return the size of the vector?**

→ We can use the .size() class-member function.

```
1  vector<int> vec = { 1, 5, 3, 2 };
2  cout << vec.size() << endl;
```

```
Console  4
```

- **What data type the .size() class-member function returns? Integer?**

  → The .size() class-member function returns a size type.

  In C++, we use size_t to represent size type.

```
1  vector<int> vec = { 1, 5, 3, -2, 18 };
2  size_t num_of_items = vec.size();
```

  → size_t is an alias of unsigned int.

  Therefore, .size() class-member function essentially returns a non-negative integer.

- **Why people would like to create alias for some data types?**

  → We know that a variable's name should make sense. It explains what the variable represents.

  ```
  int num_of_students;
  ```

  → If the data type can somehow also make sense, then the code's readability will be enhanced.

  ```
  size_t num_of_elements = vec.size();
  ```

  People know that num_of_elements represents the size of some data container.

- **Can we create our own alias?**

  → In C++, we can use the typedef statement.

  → [Example] 2-dimensional vector

```
1  #include <iostream>
2  #include <vector>
3
4  using namespace std;
5
6  typedef vector<vector<unsigned int>> Grid;
7  typedef vector<unsigned int> Row;
8
9  int main() {
10     Grid chessboard(8, Row(8, 0));
11     for (size_t i = 0; i < chessboard.size(); i++) {
12         for (size_t j = 0; j < chessboard.size(); j++) {
13             cout << chessboard[i][j] << ' ';
14         }
15         cout << endl;
16     }
17     system("pause");
18     return 0;
19 }
```

- **Appending a new item to the rear end of a vector**

  → **The biggest advantage of vectors over arrays is that size of a vector can dynamically change.**

  → **We can use the `.resize()` class-member function to change the size of a vector.**

```
1  vector<int> vec = { 1, 2, 3 };
2  vec.resize(6);  // Three "dummy" values will be added to the end.
```

If the new size is greater than the current size, then new elements are added to the end of the vector. All current elements are intact.

If the new size is smaller than the current size, then the vector will be truncated.

```
1  vector<int> vec = { 1, 2, 3 };
2  vec.resize(2);  // "3" will be deleted (truncation).
```

For the `.resize()` class-member function, the only argument is the size after resizing.

→ **The `.push_back()` class-member function**

```
1  vector<int> vec = { 1, 5, 3 };
2  vec.push_back(2);  // "vec" becomes [1, 5, 3, 2].
3  vec.push_back(7);  // "vec" becomes [1, 5, 3, 2, 7].
```

1) **The `.push_back()` class-member function appends a new element to the rear end of the vector.**
2) **It works similar to Python's `.append()` method for lists.**
3) **Using `.push_back()` class-member function, we can build a vector by keep appending elements to the rear end.**

→ **[Example] Storing values in the input file into a vector**

```
1  // Create a vector to store the values in the input file.
2  vector<int> vec;
3  // Read the values in the input file and store them in the vector.
4  int value;  // Stores the current value read from the input file.
5  while (fin >> value) { vec.push_back(value); }
```

→ The .pop_back() class-member function

```
1  vector<int> vec = { 1, 3, 5, 7, 9 };
2  vec.pop_back();  // "vec" becomes [1, 3, 5, 7].
3  vec.pop_back();  // "vec" becomes [1, 3, 5].
```

The .pop_back() class-member function removes the last element from the vector.

→ Can we insert an element at index i (may not at the rear end)?

Using index, we cannot do this directly in C++.

We need to use loops.

→ C++ provides another iteration mechanism to iterate through vectors.

Iterators

Using iterators, you can insert an item anywhere in the vector.

→ If you want to remove the element at index i from the vector, you can either "work around" (using loop), or use iterators.

→ The .pop_back() class-member function

```
1  vector<int> vec = { 1, 3, 5, 7, 9 };
2  vec.pop_back();  // "vec" becomes [1, 3, 5, 7].
3  vec.pop_back();  // "vec" becomes [1, 3, 5].
```

The .pop_back() class-member function removes the last element from the vector.

→ Can we insert an element at index i (may not at the rear end)?

Using index, we cannot do this directly in C++.

We need to use loops.

→ C++ provides another iteration mechanism to iterate through vectors.

Iterators

Using iterators, you can insert an item anywhere in the vector.

→ If you want to remove the element at index i from the vector, you can either "work around" (using loop), or use iterators.

- Quite a few functions in C++ are defined with iterators.

→ Iterator can be simply understood as "cursor" (position marker).

→ In C++, cursors (iterators) are on elements, not ~~between elements~~.

→ iterator class is defined inside the vector class.

→ [Example] Using iterator to find the maximum value in a vector.

```
1  int max_val = INT_MIN;  // Initialized to "negative infinity"
2  for (vector<int>::iterator it = vec.begin(); it != vec.end(); it++) {
3      if (*it > max_val) { max_val = *it; }
4  }
5  cout << "Max: " << max_val << endl;
```

vector<int>::iterator is the data type of an iterator of a vector of integers.

vec.begin() returns an iterator positioned at the first element of the vector vec.

vec.end() returns an iterator positioned just after the last element of the vector vec.

it++ advances the iterator, moving the iterator to the next element.

1) "++it" (prefix) advances the iterator, and returns the iterator after the advancement.
2) "it++" (postfix) advances the iterator, and returns a copy of the iterator before the advancement.
3) "--it" (prefix) moves the iterator to the previous element, and returns the iterator after the movement.
4) "it--" (postfix) moves the iterator to the previous element, and returns a copy of the iterator before the movement.

→ [Example] Using iterator to find the maximum value in a vector.

```
1  int max_val = INT_MIN;  // Initialized to "negative infinity"
2  for (vector<int>::iterator it = vec.begin(); it != vec.end(); it++) {
3      if (*it > max_val) { max_val = *it; }
4  }
5  cout << "Max: " << max_val << endl;
```

*it returns the element the iterator is on.

'*' (unary operator) is called dereferencing operator.

- **The .insert(), .erase(), and sort() functions on vectors**

```
1  vec = { 5, 4, 3, 2, 1 };
2  sort(vec.begin(), vec.end());  // "vec" becomes [1, 2, 3, 4, 5].
3  vec.insert(vec.begin(), 6);  // "vec" becomes [6, 1, 2, 3, 4, 5].
4  vec.insert(vec.begin() + 2, 0);  // "vec" becomes [6, 1, 0, 2, 3, 4, 5].
5  vec.erase(vec.end() - 2);  // "vec" becomes [6, 1, 0, 2, 3, 5].
```

vec.begin() + 2 returns an iterator positioned two elements after the first element.

vec.end() - 2 returns an iterator positioned one element before the last element (vec.end() is an iterator just after the last element).

To use the sort() function, you need to "#include <algorithm>".

- **Vector Copy**

```
1  vector<int> vec1 = { 1, 3, 5, 7, 9 };
2  vector<int> vec2 = vec1;
```

```
1  vector<int> vec1 = { 1, 3, 5, 7, 9 };
2  vector<int> vec2(vec1);  // Copy constructor
```

→ vec2 and vec1 are independent copies. Changing either one will not affect the other.

- **Vector Comparison**

→ Vector comparison uses the same mechanism as string comparison.

- **Strings**

→ A string is a sequence of characters.

→ A C string is just an array of characters.

```
1  char s[] = "Hello";
```

You can use a string literal to initialize a C string or assign a value to a C string.

There are 6 characters in the C string s.

Index	0	1	2	3	4	5
Character	'H'	'e'	'l'	'l'	'o'	'\0'

→ Because a string can be shorter than the character array, a C string must end with a special character known as a NULL character, written as '\0'.

Compiler automatically appends a NULL character.

→ A character array of size 20 can store strings of lengths 0 to 19. The longest string is 19, not ~~20~~, since the NULL character must be stored.

→ An array of characters ending with a NULL character is known as a NULL-terminated string.

```
1  char s[5] = "Hello";  // What's wrong?
```

→ Output streams automatically handle NULL-terminated strings, printing each character until reaching the NULL character that ends the string.

```
1  char s[] = "HelloWorld";
2  s[5] = '\0';
3  cout << s << endl;
```

```
Console  Hello
```

→ [Example] How to traverse a NULL-terminated string?

```
1  // Count how many capital letters in the string.
2  char s[100] = "HelloWorld";
3  unsigned int count = 0;
4  for (unsigned int i = 0; s[i] != '\0'; i++) {
5      if (s[i] >= 65 && s[i] <= 90) { count++; }
6  }
7  cout << count << endl;
```

You need to check whether it reaches the NULL character, instead of checking the size of the array.

→ The <cstring> library provides some functions to handle C strings.

strcmp(): Compare two strings.

strlen(): Return the size of the given string.

strcpy(): Copy the characters from source to destination.

strcat(): Append copy of string to another string.

- **C++ Strings**

→ #include <string>

→ C++ introduced its own string type to reduce those errors and increase programmer convenience.

You can use [] operator or .at() (recommended) class-member function to access characters in the string.

You can use .size() class-member function to get how many characters in the string. It returns a size_t.

C++ strings are not ~~NULL-terminated~~.

→ [Exercise] Please write a program that let the user enter a string, then output the number of digits the user just typed in.

Input: "Hello World" Expected Output: 0

Input: "33c3 f 2 00" Expected Output: 6

- **String Operations**

→ **String concatenation**

```
1  string s1 = "Hello", s2 = "World";
2  string s3 = s1 + s2;  // "s3" will be "HelloWorld".
```

→ **Substring function: .substr()**

```
1  string s1 = "HelloWorld";
2  string s2 = s1.substr(0, 5);  // "s2" is "Hello".
3  string s3 = s1.substr(5);  // "s3" is "World".
```

.substr(start, length)

It generates a string started from index start and has length length.

.substr(start)

It generates a string started from index start to the end of the original string.

→ **String functions:**

.push_back(): appends a character to the rear end of the string.

.pop_back(): removes the character at the rear end of the string.

- **C++ strings also support iterators.**

```
1  int main() {
2      string s;  // Stores the user input string.
3      unsigned int count = 0;  // Counts the number of digits.
4      cout << "Please enter a string: ";
5      getline(cin, s);
6      for (string::iterator it = s.begin(); it != s.end(); it++) {
7          if (isdigit(*it)) { count++; }
8      }
9      cout << "Number of digits: " << count << endl;
10     system("pause");
11     return 0;
12 }
```

You need to "#include <cctype>" to use function isdigit().

→ **[Important] C++ strings are mutable.**

Python strings are immutable.

```
1  string s = "abcd";
2  s.at(0) = 'x';  // "s" becomes "xbcd".
3  s.push_back('y');  // "s" becomes "xbcdy".
4  s.insert(s.begin() + 1, 'z');  // "s" becomes "xzbcdy".
```