C++ Programming

Week 4:

Arrays, Strings and Vectors

Dr. Owen Chen
Cary Chinese School
Director of Math and Computer Science

2023 Summer

Week 4: Agenda

- Review C++ operators
- Review Flow Control
- Review Homework
- Arrays
- Strings
- Vectors

Operators Precedence

Precedence	Operator	Description	Associativity
1	a++ a--	Suffix/postfix increment and decrement	Left-to-right
2	++a --a ! ~	Prefix increment/decrement, Logical/Bitwise Not	Right-to-left
3	a*b a/b a%b	Multiplication, division, and remainder	Left-to-right
4	a+b a-b	Addition and subtraction	Left-to-right
5	≪ ≫	Bitwise left shift and right shift	Left-to-right
6	< <= > >=	Relational operators	Left-to-right
7	== !=	Equality operators	Left-to-right
8	&	Bitwise AND	Left-to-right
9	^	Bitwise XOR	Left-to-right
10	|	Bitwise OR	Left-to-right
11	&&	Logical AND	Left-to-right
12	||	Logical OR	Left-to-right

if **and** if-else

```
int num = 10;

if (num < 5)
   cout << "The number is less than 5. " << endl;
else
   cout << “else condition”;

if (num == 5 ) {
   cout << "The number is 5." << endl;
} else {
   cout << "The number is not 5." << endl;
}
```

if-else if-else

```
if (num < 5)
   cout << "The number is less than 5." << endl;
else if (num > 20)
   cout << "The number is greater than 20." << endl;
else if (num > 10)
   cout << "The number is in range (10, 20]." << endl;
else
   cout << "The number is in range [5, 10]." << endl;
```

Relational Expressions

Operator name	Example
equal to	`a == b`
not equal to	`a != b`
less than	`a < b`
greater than	`a > b`
less than or equal to	`a <= b`
greater than or equal to	`a >= b`

Logical Expressions

- **An operand is not `bool`, it will be converted to `bool` implicitly.**

Operator name	Symbol-like operator	Keyword-like operator	Example
negation	!	not	!a
AND	&&	and	a && b
Inclusive OR	||	or	a || b

- **Precedence: `!` > `&&` > `||`**

- **What's the value of the follow expressions?**

```
if(-2 && true)
   cout << "The condition is true." << endl;
if(not -2)
   cout << " (!-2) is true, really?" << endl;
```

for and while Loops

- for

```
for ([init]; [cond]; [increment]) {
    ...
}
```

To use when number of iterations is known

- while

```
while (cond) {
    ...
}
```

To use when number of iterations is not known

- do while

```
do {
...
} while (cond);
```

To use when number of iterations is not known, but there is at least one iteration

while loop

while.cpp

Syntax :

```
while ( expression ) {
   //...
}
```

If the condition is true, the statement (loop body) will be executed.

```
int num = 10;
While (num > 0) {
   cout << "num = " << num << endl;
   num--;
}
```

do-while loop

while.cpp

The test takes place after each iteration in a do-while loop.
The test takes place before each iteration in a while loop.

```
int num = 10;
do {
   cout << "num = " << num << endl;
   num--;
} while (num > 0);
```

for loop

for.cpp

- **Syntax:**

```
for (init-clause; cond-expression; iteration-expression)
    loop-statement
```

- **Example:**

```
int sum = 0;
for(int i = 0; i < 10; i++)
{
    sum += i;
    cout << "Line " << i << endl;
}
cout << "sum = " << sum << endl;
```

`switch` Statement

Execute one of several statements, depending on the value of an expression.
`break` prevents executing some following statements. Don't forget `break`!
More similar to `goto`, not `if-else if-else`

```
switch (input_char)
{
   case 'a':
                  x = 'a';
                  break;
   case 'A':
      cout << "Move left." << endl;
      break;
   case 'd':
   case 'D':
      cout << "Move right." << endl;
      break;
   default:
      cout << "Undefined key." << endl;
      break;
}
```

switch.cpp

Review Homework for last week

1) Go over Week 3 Class Notes.

2) Is the following program legal? If so, what values are printed?

```
int i = 100, sum = 0;
for (int i = 0; i != 10; ++i)
sum += i;
cout << i << " " << sum << endl;
```

3) Write a program that uses a while to sum the numbers from 50 to 100.

4) Write a C++ program using a for loop to print out the first 20 Fibonacci numbers like this:

First 20 Fibonacci Numbers:
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765

5) Write a C++ program to print all printable symbols and their corresponding ASCII codes.
The first printable symbol with the smallest ASCII code is ! and its ASCII code is 33 and the last printable symbol with the largest ASCII code of 126 is ~

Review Homework for last week

1) Go over Week 3 Class Notes.

2) Is the following program legal? If so, what values are printed?

```
int i = 100, sum = 0;
for (int i = 0; i != 10; ++i)
sum += i;
cout << i << " " << sum << endl;
```

Solutions:

- homework3_loop.cpp
- fibonacci_numbers.cpp
- print_ascii_table.cpp

3) Write a program that uses a while to sum the numbers from 50 to 100.

4) Write a C++ program using a for loop to print out the first 20 Fibonacci numbers like this:

First 20 Fibonacci Numbers:
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765

5) Write a C++ program to print all printable symbols and their corresponding ASCII codes.
The first printable symbol with the smallest ASCII code is ! and its ASCII code is 33 and the last printable symbol with the largest ASCII code of 126 is ~

Today's Topics:

Strings, Arrays and Vectors

Strings

What is a String?

- Strings are essentially collections of characters.
- Examples:
    - "Hello World",
    - "My name is Jason"

- They're enclosed in a double “quote” or a single ‘quote’.

- **In C++, we have two types of strings:**

1) std::string

- The std::string class that's provided by the C++ Standard Library is the preferred method to use for strings.

2) C-style Strings

- These are strings derived from the C programming language and they continue to be supported in C++.
- These "collections of characters" are stored in the form of arrays of type char that are null-terminated (the \0 null character).
- C-style strings are relatively unsafe and not recommended.

C++ `string`

A **`string`** is a variable-length sequence of characters. To use the `string` type, we must include the `string` header. Because it is part of the library, `string` is defined in the `std` namespace. Our examples assume the following code:

```
#include <string>
using std::string;
```

This section describes the most common `string` operations;

Defining and Initializing `strings`

Each class defines how objects of its type can be initialized. A class may define many different ways to initialize objects of its type. Each way must be distin- guished from the others either by the number of initializers that we supply, or by the types of those initializers. Table 3.1 lists the most common ways to initialize `strings`. Some examples:

Table 3.1: Ways to Initialize a `string`

`string s1`	Default initialization; `s1` is the empty string.
`string s2(s1)`	`s2` is a copy of `s1`.
`string s2 = s1`	Equivalent to `s2(s1)`, `s2` is a copy of `s1`.
`string s3("value")`	`s3` is a copy of the string literal, not including the null.
`string s3 = "value"`	Equivalent to `s3("value")`, `s3` is a copy of the string literal.
`string s4(n, 'c')`	Initialize `s4` with `n` copies of the character `'c'`.

C-style strings

A C-style string (null-terminated strings/arrays of characters) is a series of characters stored in bytes in memory.
This kind of strings can be declared as follows

```
char rabbit[16] = {'P', 'e', 't', 'e', 'r'};
char bad_pig[9] = {'P', 'e', 'p', 'p', 'a', ' ', 'P', 'i', 'g'}; //a bad one!
char good_pig[10] = {'P', 'e', 'p', 'p', 'a', ' ', 'P', 'i', 'g', '\0'};
```

initchar.cpp

- size_t strlen(const char *str);

  Returns the number of characters, the first NULL will not be included.

```
char name[10] = {'Y', 'u', '\0', 'S', '.', '0'};
cout << strlen(name) << endl;
```

C-Style String manipulation and examination

Copy

```
char* strcpy( char* dest, const char* src );
```

```
char *strncpy(char *dest, const char *src, size_t count);
```

Concatenate: appends a copy of src to dest

```
char *strcat( char *dest, const char *src );
```

Compare

```
int strcmp( const char *lhs, const char *rhs );
```

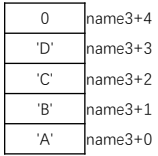
stringop.cpp

String literals

It isn't convenient to initial a string character by character. String literals can help.

```
char name1[] = "North Carolina";
char name2[] = "North "    " Carolina";
char name3[] = "Carolina"; //how many bytes for the array?
```

```
const wchar_t[] s5 = L"ABCD";
const char16_t[] s9 = u"ABCD"; //since C++11
const char32_t[] s6 = U"ABCD"; //since C++11
```



C++ string class

C++ string class provides functions to manipulate and examinate strings.

```
std::string str1 = "Hello";
std::string str2 = "SUSTech";
std::string result = str1 + ", " + str2;
```

Different types of strings

```
std::string
std::wstring
std::u8string //(C++20)
std::u16string //(C++11)
std::u32string //(C++11)
```

stdstring.cpp

Example: read a string from user: hello_name.cpp

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

int main(){
    string name;
    cout << "Please enter your name:";
    cin >> name;
    cout << "Hello, " << name << "!" << endl;
    return 0;
}
```

Reading and Writing `strings`

We use the `iostream` library to read and write values of built-in types such as `int`, `double`, and so on.

We use the same IO operators to read and write `strings`:

```
string s;
cin >> s;
cout << s << endl;
```

`string` Operations

Table 3.2: `string` Operations

`os << s`	Writes `s` onto output stream `os`. Returns `os`.
`is >> s`	Reads whitespace-separated string from `is` into `s`. Returns `is`.
`getline(is, s)`	Reads a line of input from `is` into `s`. Returns `is`.
`s.empty()`	Returns `true` if `s` is empty; otherwise returns `false`.
`s.size()`	Returns the number of characters in `s`.
`s[n]`	Returns a reference to the `char` at position `n` in `s`; positions start at 0.
`s1 + s2`	Returns a `string` that is the concatenation of `s1` and `s2`.
`s1 = s2`	Replaces characters in `s1` with a copy of `s2`.
`s1 == s2`	The `strings` `s1` and `s2` are equal if they contain the same characters.
`s1 != s2`	Equality is case-sensitive.
`<, <=, >, >=`	Comparisons are case-sensitive and use dictionary ordering.

`string` cctype functions

Table 3.3: `cctype` Functions	
`isalnum(c)`	`true` if `c` is a letter or a digit.
`isalpha(c)`	`true` if `c` is a letter.
`iscntrl(c)`	`true` if `c` is a control character.
`isdigit(c)`	`true` if `c` is a digit.
`isgraph(c)`	`true` if `c` is not a space but is printable.
`islower(c)`	`true` if `c` is a lowercase letter.
`isprint(c)`	`true` if `c` is a printable character (i.e., a space or a character that has a visible representation).
`ispunct(c)`	`true` if `c` is a punctuation character (i.e., a character that is not a control character, a digit, a letter, or a printable whitespace).
`isspace(c)`	`true` if `c` is whitespace (i.e., a space, tab, vertical tab, return, newline, or formfeed).
`isupper(c)`	`true` if `c` is an uppercase letter.
`isxdigit(c)`	`true` if `c` is a hexadecimal digit.
`tolower(c)`	If `c` is an uppercase letter, returns its lowercase equivalent; otherwise returns `c` unchanged.
`toupper(c)`	If `c` is a lowercase letter, returns its uppercase equivalent; otherwise returns `c` unchanged.

Example: string + operator: hello_name2.cpp

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

int main(){
    string name;
    string greet = "Hello, ";
    string greet_end = "!\n";
    cout << "Please enter your name:";
    cin >> name;
    cout << greet + name + greet_end;
    return 0;
}
```

C-Style String

```
char str[] = "c string";

char str[9] = "c string";
char str[] = {'c', ' ', 's', 't', 'r', 'i', 'n', 'g', '\0’};
char str[9] = {'c', ' ', 's', 't', 'r', 'i', 'n', 'g', '\0'};
```

Arrays

- An array is a contiguously allocated block of memory
- Fixed number of elements
- Its element type can be any fundamental type (int, float, bool, etc), structure, class, pointer, enumeration,

array.cpp

```cpp
#include <iostream>
using namespace std;
int main()
{
    int num_array1[5]; //uninitialized array, random values
    int num_array2[5] = {0, 1, 2, 3, 4}; //initialization
    for(int i = 0; i < 5; i++)
        cout << num_array1[i] << " ";
    cout << endl;
    for(int i = 0; i < 5; i++)
        cout << num_array2[i] << " ";
    cout << endl;
    return 0;
}
```

Variable-length arrays

If the length is not an integer constant expression, the array will be a variable-length one.

variable-array.cpp

```
int len = 1;
while ( len < 10 )
{
    int num_array2[len]; //variable-length array
    cout << "len = " << len;
    cout << ", sizeof(num_array2)) = "
         << sizeof(num_array2) << endl;
    len ++;
}
```

Arrays of unknown size

The number is not specified in the declaration.

```
int num_array[ ] = {1, 2, 3, 4}; // the type of num_array is "array of 4 int"
```

The arguments of a function

```
float array_sum(float values[], size_t length);
```

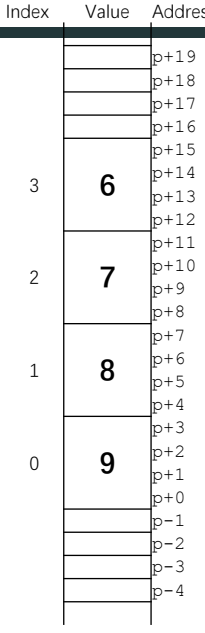
```
float array_sum(float *values, size_t length);
```

Element accessing

index-bound.cpp

```
int array1[4] = {9,8,7,6};
int array2[4];
array2 = array1; //error!
array2[0] = array1[0]; //okay
array2[1] = array1[1]; //okay
array2[2] = array1[2]; //okay
array2[3] = array1[3]; //okay


int num_array[5];
for(int idx = -1; idx <= 5; idx++)
   num_array[idx] = idx * idx;
for(int idx = -1; idx <= 5; idx++)
   cout << num_array[idx] << endl;
```



- **Arrays are not objects in C/C++ (different with Java);**
- **Arrays can be regarded as addresses**

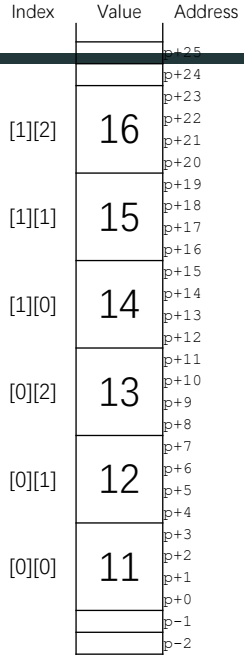
Multidimensional arrays

```
int mat[2][3] = {{11,12,13}, {14,15,16}};
for (int r = 0; r < rows; r++)
{
   for(int c = 0; c < cols; c++)
      cout << mat[r][c] << ",";
   cout << endl;
}
```

md-array.cpp

Arrays of unknown bound

```
void init_2d_array(float mat[][], //error
          size_t rows, size_t cols)
void init_2d_array(float mat[][3],
          size_t rows, size_t cols)
```



const **Arrays**

```
const float PI = 3.1415926f;
PI += 1.f; // error
const float values[4] = {1.1f, 2.2f, 3.3f, 4.4f};
values[0] = 1.0f; // error
```

Used as function arguments

```
float array_sum(const float values[], size_t length)
{
   float sum = 0.0f;
   for (int i = 0; i < length; i++)
   {
      sum += values[i];
      //values[i] = 0; //error
   }
   return sum;
}
int main()
{
   float values[4] = {1.1f, 2.2f, 3.3f, 4.4f};
   float sum = array_sum(values, 4);
}
```

const-array.cpp

Vector

- A **`vector`** is a collection of objects, all of which have the same type. Everyobject in the collection has an associated index, which gives access to that object. A `vector` is often referred to as a **container** because it "contains" other objects.

- To use a `vector`, we must include the appropriate header. In our examples, we also assume that an appropriate `using` declaration is made:

```
#include <vector>
using std::vector;
```

- A `vector` is a **class template**. C++ has both class and function templates.

Vector Initialization

- **Code: vector.cpp**

Table 3.4: Ways to Initialize a `vector`

`vector<T> v1`	`vector` that holds objects of type `T`. Default initialization; `v1` is empty.
`vector<T> v2(v1)`	`v2` has a copy of each element in `v1`.
`vector<T> v2 = v1`	Equivalent to `v2(v1)`, `v2` is a copy of the elements in `v1`.
`vector<T> v3(n, val)`	`v3` has `n` elements with value `val`.
`vector<T> v4(n)`	`v4` has `n` copies of a value-initialized object.
`vector<T> v5{a,b,c...}`	`v5` has as many elements as there are initializers; elements are initialized by corresponding initializers.
`vector<T> v5 = {a,b,c...}`	Equivalent to `v5{a,b,c...}`.

Adding Elements to a `vector`

- **Code: vector.cpp**

```
vector<int> v2; // empty vector
for (int i = 0; i < 100; i++) {
   v2.push_back(i);
 }
```

Vector Operations

Table 3.5: vector Operations	
`v.empty()`	Returns `true` if `v` is empty; otherwise returns `false`.
`v.size()`	Returns the number of elements in `v`.
`v.push_back(t)`	Adds an element with value `t` to end of `v`.
`v[n]`	Returns a reference to the element at position `n` in `v`.
`v1 = v2`	Replaces the elements in `v1` with a copy of the elements in `v2`.
`v1 = {a,b,c...}`	Replaces the elements in `v1` with a copy of the elements in the comma-separated list.
`v1 == v2`	`v1` and `v2` are equal if they have the same number of elements and each element in `v1` is equal to the corresponding element in `v2`.
`v1 != v2`	
`<, <=, >, >=`	Have their normal meanings using dictionary ordering.

Range-based for Loop

C++11 introduces the **range-based for loop** to simplify the verbosity of traditional `for` loop constructs. They are equivalent to the `for` loop operating over a range of values, but **safer**

The range-based `for` loop avoids the user to specify start, end, and increment of the loop

```
for (int v : { 3, 2, 1 }) // INITIALIZER LIST
    cout << v << " ";     // print: 3 2 1

int values[] = { 3, 2, 1 };
for (int v : values)      // ARRAY OF VALUES
    cout << v << " ";     // print: 3 2 1

for (auto c : "abcd")     // RAW STRING
    cout << c << " ";     // print: a b c d
```

Range-based for Loop

Range-based for loop can be applied in three cases:

- Fixed-size array `int array[3]`, `"abcd"`
- Branch Initializer List `{1, 2, 3}`
- Any object with `begin()` and `end()` methods

```
std::vector vec{1, 2, 3, 4};

for (auto x : vec) {
    cout << x << ", ";
// print: "1, 2, 3, 4"
```

```
int matrix[2][4];
for (auto& row : matrix) {
    for (auto element : row)
        cout << "@";
    cout << "\n";
}
// print: @@@@
//        @@@@
```