COSC 151: Intro to Programming: C++

Chapter 3
Strings, Vectors and Arrays

Chapter 3: Strings, Vectors, and Arrays

- **Objectives**
  - We should be able to
    - Understand the **`string`** library type
    - Understand the **`vector`** library type
    - Be introduced to **`auto`** – get used to it
    - Use **`string`** and **`vector`** for our programs
    - Reinforce using types with member functions and data (like our **`Sales_item`**) class
    - **Iterators** – a general purpose way to traverse collections of objects

C3: Library string Type

- A string is a variable length sequence of characters
- There are many ways to initialize a string

```
#include <string>        // include the necessary header
std::string s1;          // default initialization, s1 is the empty string
std::string s2 = s1;     // s2 is a copy of s1
std::string s3 = “hiya”; // s3 os a copy pf the string literal
std::string s4(10, 'c'); // s4 is cccccccccc
```

C3: Operations on string objects

- We can read and write strings through io stream objects

```
#include <string>        // include the necessary header
#include <iostream>      // include the header for io streams

int main()
{
   std::string s;               // initialize, empty string s
   std::cin >> s;               // read white-space terminated string from cin
   std::cout << s << std::endl; // write the string to cout
   return 0;
}
```

C3: Operations on string objects (contd)

- **Common string operations**

```
std::string s1;
std::string s2;

// ...
os << s1 // writes string s1 to output stream os
is >> s2 // reads whitespace separated string from input stream is to s2
getline(is, s1) // reads entire line (up to newline) from is, stores in s1
s1.empty() // returns true if s1 is the empty string
s.size() // returns the number of characters in s1
s[n]; // returns a reference to the character in s at potition n
s1 + s2; // returns a string that is the concatenation of s1 and s2
s1 = s2; // replaces the content of s1 with a copy of s2
s1 == s2; // returns true if s1 and s2 have the same content (case-sensitive)
s1 != s2; // returns true is s1 and s2 have different content (case-sensitive)
// <, <=, >, >= supported for case-senstive and lexagraphic ordering
```

C3: string Reading

- **Stream insertion operator is whitespace delimited**

```
string s;
cin >> s;  // read s, user enters “Hello World!” + ENTER
cout << s; // prints s, “Hello”, reading stopped at whitespace
```

- **Use getline to read an entire line (up to a newline character)**

```
string s;
getline(std::cin, s);  // read s, user enters “Hello World!” + ENTER
cout << s; // prints s, “Hello World!”, reading continues until new line
```

- **Reading an unknown number of strings**

```
string s;
while(cin >> s)  // read s, until conversion fails (stop with Ctrl+Z)
   cout << s;
```

C3: string::empty()

- **The empty() function returns a bool, true if the string is empty, false otherwise**

```
std::string line;
while(getline(std::cin, line) // read all lines...
{
   // here we use the logical not operator (operator!), it returns
   // the inverse of the bool value of its operand
   if(!line.empty())
   {
      // we only get here if line is NOT empty
      std::cout << line << std::endl;
   }
}
```

C3: `string::size()` and `string::size_type`

- **The `size()` function returns the number of characters in the `string`**
  - The type returned by `size()` is `std::string::size_type`
  - This return type is long and good programmers are lazy
- **When the compiler “knows“ the type, we can ask it to use the correct type for us, using the keyword `auto`.**
  - Get used to `auto`, and start using it

```
std::string text;

int s = text.size() // may work, technically incorrect, compiler may WARN
unsigned int s = text.size() // may work, technically incorrect

std::string::size_type s = text.size(); // correct, but long (and we're lazy!!!!)
auto s = text.size(); // correct, terse, wonderful!
```

C3: Iterating a string

- **When iterating over an entire string, use range-based for.**

```
for(declaration : expression)
   statement

std::string str = “some string”;

// print the characters in str one character to a line
for(auto c : str)
   std::cout << c << std::endl;
```

- **To change the characters of a string, process with range-based for, with a reference as the loop variable type**

```
std::string str = “some string”;

// convert each character in str to upper case
for(auto& c : str)
   c = topupper(c);

std::cout << str << std::endl; // prints “SOME STRING”
```

C3: Partial Processing string

- **Sometimes we only want to process certain elements of a string, not iterate it.**
- **String supports random-access of content via the subscript operator (operator[])**

```
// A very simply function for ensuring a string starts
// with a capital letter and ends in a period
void ensure_sentence(std::string& sentence) // take sentence as reference so
                                            // it can be changed
{
    if(!sentence.empty())
    {
       // sentence has at least one character
       // make sure the first letter is  upper case
       sentence[0] = toupper(sentence[0]);

       auto last = sentence.size() - 1; // get the last character of the sentence
                                        // remember indexes are 0 based, so if a
                                        // sentence has 10 characters, the last
                                        // index is 9 (hence the -1)

       if(!ispunct(sentence[last]))
       {
          // someone forgot the punctuation, just append a period
          sentence += '.';
       }
   }
}

std::string my_sentence = “hello, everyone”;
ensure_sentence(my_sentence);
std::cout << my_sentence << std::endl; // prints “Hello, everyone.”
```

C3: Iterating via subscript operator

- **It is possible to iterate a string using the subscript operator**
  - In fact, *this is very common*. Range-based for was introduced only recently, so there is quite a lot of code that still uses this (though inferior) form.
  - This form is still useful if *you don't intend to process all characters* in the string

```
std::string s;

// The make uppercase logic from previous...
for(std::string::size_type index = 0; index < s.size(); ++index)
{
   s[index] = toupper(s[index]);
}
```

C3: Library `vector` Type

- **A `vector` is a collection of objects, all of the same type**
- **`vector` is a class template**
  - Can be thought of as instructions to the compiler for generating classes or functions
  - Have some special syntax that we will learn more about later...

C3: Defining and Initializing a vector

- **Like string, there are many ways to define and intialize a vector.**

```
#include <vector>        // include the necessary header
using std::vector;       // using directive allows us to skip the std:: prefix

vector<int> v1;           // default initialization, empty container of ints
vector<Sales_item>;       // empty vector of Sales_item objects
vector<vector<string>> s; // vectors can contain vectors

vector<int> ivec;           // empty vector of ints
vector<int> ivec2 = ivec;   // copy elements of ivec into ivec2
vector<string> svec = ivec; // ERROR:  svec holds strings not ints

vector<int> ivec3(10, 5);   // has 10 elements with value 5
```

C3: vector List Initialization

- **A vector can be initialized with a list of values, but note the difference when using list initialization {} and calling a specific constructor ().**

```
vector<int> ivec(3, 7);      // 3 elements of value 7
vector<int> ivec2{3, 7};     // 2 elements values, 3 and 7

vector<string> svec = {“C++”, “is”, “fun”}; // 3 elements, “C++”,”is”,”fun”
```

C3: Adding elements to vector

- We can add elements to a vector at any time
- Elements are added using push_back()

```
vector<int> input_values; // starts off empty...

input_values.push_back(14);
input_values.push_back(9213344);
// vector now has two elements, 14 and 9213344
```

- You can add an unknown number of elements using a loop

```
vector<int> input_values; // starts off empty...

int iv = 0;
while(std::cin >> iv) // read in values input by the user
{
   input_values.push_back(iv); // store them in the vector
}
```

C3: Common vector Operations

- Common vector operations

```
std::vector<int> v1;
std::vector<int> v2;

// ...
v1.empty() // returns true if v1 has no elements
v1.size()  // returns the number of elements in v1
v1[n];     // returns a reference to the element in v1 at potition n
v1 = v2;   // replaces the content of v1 with a copy of v2
v1 = {1,...};// replaces the content of v1 with the elements in the list
v1 == v2; // returns true if s1 and s2 have the same content
v1 != v2; // returns true is s1 and s2 have different content
// <, <=, >, >= supported for lexagraphic ordering
```

C3: `vector::size()` and `vector::size_type`

- **The `size()` function returns the number of elements in the `vector`**
  - The type returned by `size()` is `std::vector<T>::size_type`
  - This return type is long and good programmers are lazy
- **When the compiler “knows“ the type, we can ask it to use the correct type for us, using the keyword `auto`.**
  - Get used to `auto`, and start using it

```
std::vector<int> v;

int s = v.size() // may work, technically incorrect, compiler may WARN
unsigned int s = v.size() // may work, technically incorrect

std::vector<int>::size_type s = v.size(); // correct, but long (and we're lazy!!!!)
auto s = v.size(); // correct, terse, wonderful!
```

C3: Iterating a vector

- **When iterating over an entire vector, use range-based for.**

```
for(declaration : expression)
   statement

std::vector<int> v = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

// print the numbers in v one number to a line
for(auto n : str)
   std::cout << n << std::endl;
```

- **To change the elements of a vector, process with range-based for, with a reference as the loop variable type**

```
std::vector<int> v = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

// subtract 1 from each elements
for(auto& n : str)
   n -= 1;

for(auto n : str)                 // prints 0, 1, 2... one number per line
   std::cout << n << std::endl;
```

C3: Iterating via subscript operator

- Like string, vector supports random-acess via the subscript operator (operator[])
- We can use this for processing specific elements of the vector

```
std::vector<int> v = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
v[0] = 10;
v[3] = 30;
```

- We can also use it for iterating

```
std::vector<int> v = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

// print elements one per line
for(std::vector<int>::size_type idx = 0; idx < v.size(); ++idx)
{
   std::cout << v[idx] << std::endl;
}
```

C3: Subscriping does not add Elements

- **In the context of both `vector` and `string`, subscripting does not add elements**

```
std::vector<int> v;
std::string s;

v[0] = 10; // This will cause a crash!
s[0] = 10; // Also will cause a crash (or throw an exception...)
```

C3: Iterators

- **Iterators are a fundamental part of the standard library**
- **Iterators represent a range**
  - Begin – the first elements
  - End – one past the last element
  - [begin, end)
- **All of the standard library container types (including `string` and `vector`) support iterators**

C3: Iterators (contd)

- **Iterators allow traversal of a range, and allow access to the currently denoted element**

```
std::vector<int> v = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

auto it = v.begin();  // get an iterator to the first element
auto it2 = v.end();

*it += 6;  // v[0] now equals 7
++it;      // move to the next element
*it -= 1;  // v[1] now equals 1
--it;      // move to the previous element
std::cout << *it;  // prints 7

if(it == it2) // iterators can be compared for equality
{
   std::cout << “Empty!” << std::endl; // won't print, we know it's not empty
}

if(it != it2) // iterators can be compared for inequality
{
   std::cout << “Not empty!” << std::endl;
}
```

C3: Iterators (contd)

- **Iterators can be used to iterate a range**
  - Hopefully, that was obvious from the name

```cpp
std::vector<int> v = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

for(auto it = v.begin(); // it is the “first element”
    it != v.end();       // continue until it is one past the last element
    ++it)                // increment the loop variable (iterator)
{
   std::cout << *it << std::endl;
}
```

C3: Iterators (contd)

- **Iterators have specific types**

```
std::vector<int> iv = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
std::vector<double> dv = {1.0, 2.10, 3.210, 4.3210};
std::string s = “Hello, everyone!”;

// ivit is of type std::vector<int>::iterator
auto ivit = iv.begin();

// dvit is of type std::vector<double>::iterator
auto dvit = dv.begin();

// sit is of type std::string::iterator
auto sit = s.begin();

if(ivit == dvit) // compile error, types incompatible
{
}

if(sit == ivit) // compile error, types incompatible
{
}
```

C3: Iterators (contd)

- **Some operations invalidate iterators**
  - Especially operations that modify the size of a range

```
std::vector<int> iv = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
std::string s = “Hello, everyone!”

auto ivit = iv.begin();   // grab iterator to first element

for(int i = 0; i < 5; ++i)  // move ahead 5 elements...
   ++ivit;

ivit.resize(3);            // remove some elements, ivit invalidated

auto svit = s.begin();
s += “ Today is a good day.”;  // add some elements, svit invalidated
```

- **Takeaway: Don't modify the size of a range while iterating it**
  - This rule also applies for range-based for

C3: Pointers, Arrays, Multidimensional Arrays, etc.

- **These are advanced topics that are (mostly) no longer necessary in modern C++.**
- **Instead of a raw array, use `std::array`**

```
int arr[10];  // an array of 10 integers
std::array<int, 10> arr;  // the modern way to express the above, prefer it
```

- **Multidimensional arrays, arrays decaying to pointers, sending array elements**
  - You may need to understand this when looking at legacy code
  - For your own sanity, don't write this stuff in new code *when you can avoid it*

Final Thoughts

- **`string` objects are used to store sequences of characters**
- **`vector` is a dynamically resizing container of objects all of the same type**
- **Use `auto` for type deduction. It will save you time and effort.**
- **Use range-based for loops when you need to fully iterate a container**
  - `string`, `vector`, both!
- **Use subscript operator for random access to container elements**
  - `string`, `vector`, both!

Final Thoughts (contd)

- **Iterators represent a range of values [begin, end)**
- **Use iterators to access elements of a range**
  - `string`, `vector`, both!
  - And more – we will learn
- **Some operations invalidate iterators**
  - Be careful when caching (holding) iterators