Course
ELEC-A7151
Course materials
Your points
Code
Code Vault
Course Pages
MyCourses
Teams Channel

This course has already ended.
The latest instance of the course can be found at: Object oriented programming with C++: 2023 Autumn

‹ 3 I/O streams Course materials 5 Classes and objects ›
ELEC-A7151 / Module 1: Basics / 4 Strings and vectors

4 Strings and vectors¶

- **You can download the template for the programming tasks of the module as a `zip` file from `this link`.**

Contents

The C language provided the programmer only the simple basic datatypes, and anything more complex (such as string manipulation, or linked lists) needed to be implemented by the programmer, or acquired from some external library. The C++ Standard Library provides ready-made classes needed for operating on common types data, such as a **string** class for character strings, or **vector** class for vector of multiple data items. Following the information hiding principles, the programmer does not need to worry about operating a contents inside the *string* or *vector* object, but operates the objects using the functions provided in the public interface. This way should be able to avoid some of the familiar bugs from C, such as careless management of pointers and memory.

4.1 Strings¶

C++ Primer: Chapter 3.2 (String)

The **string** type contains a variable-length sequence of characters. In C we would have strings through `char *` (char pointer), where programmer needs to mind about space allocation for the string, and proper termination of the string. On the C programming course, errors in string management were one of the most common reason of failures. For C++, however, **string** is an abstract data type that takes care of space allocation, safe initialization and management of the string, on behalf of the programmer. Because string is part of the the C++ standard library, it belongs to the std namespace. Strings are defined in a separate "*string*" header.

Declaring a string object looks rather similar to declaring variables in C:

```
std::string s1;  // an empty string
std::string s2 = "Hello";
std::string s3 = s2;
```

In the above example, **s1** is an empty string. Different from C, it **is initialized**, and one can assume the string to be always empty after the variable declaration. An initial value can be assigned together with variable declaration, as done for **s2**. **s3** is assigned based on the content of **s2**. The content of string **s2** is *copied* to string **s3**.

The basics operators can be redefined to work in a new way with C++ objects. This is called *operator overloading*, and will be discussed in more detail later. For example, with *string* objects, the '+' operator can be used to concatenate two strings. You can study the following code in your local development environment by modifying and testing it as you wish.

```
1#include <iostream>
2#include <string>
3
4int main(void) {
5   std::string s1 = "Hello";
6   std::string s2;
7   std::string s3;
8
9   s2 = "world";
10   s3 = s1 + " " + s2;
11   std::cout << s3 << std::endl;
12
13   return 0;
14}
```

When a string needs to be read from the user, the **cin** stream can be used as shown below:

```
1#include <iostream>
2#include <string>
3
4int main(void) {
5   std::string s1;
6   std::cin >> s1;
7   std::cout << "String was: " << s1 << std::endl;
8   unsigned int length = s1.length();
9   std::cout << "Length of the string is: " << length << std::endl;
10
11   return 0;
12}
```

The above reads a string from user into *s1*, and then outputs it to standard output, changing the line in the end. For input streams, the characters until the first whitespace (space, tab, newline, etc.) are read into the string.

In addition, the **length** function is used to get the length of the string that was given by the user. The *length* function is defined as a member of the string class, and therefore it must always be called for a particular object representing the class, here `s1.length()` (more about member functions in section 5, "Classes and Objects"). The **string** class also includes other functions for operating on the string, see the link to cppreference below.

Strings can also be passed as function arguments and return values, as any other data type:

```
1#include <string>
2#include <iostream>
3
4int LongerLength(const std::string& a, const std::string& b) {
5   if (a.size() > b.size())
6      return a.size();
7   else
8      return b.size();
9}
10
11int main(void) {
12   std::string s1 = "Hei";
13   std::string s2 = "Hello";
14   std::cout << LongerLength(s1, s2);
15
16   return 0;
17}
```

The above program uses two *strings* as the arguments of function *LongerLength*, that returns the length of the longer of the two *strings*. The *strings* are passed as **references**, i.e., the string object is not copied with the function call, but the function operates on the original *strings* in the *main* function (more about references later). The **const** declarations in front of the function arguments also tell, that the *LongerLength* function is not going to modify the *strings*.

Full reference of string class

> Full reference of the string class in [cppreference]. Note that string is a basic_string with char as template type. Templates are explained in the following modules, but the basic_string page roughly applies to string. This includes all the functions and members.

4.2 Vectors¶

C++ Primer: Chapter 3.3 (Vector)

Vector is a list of objects of a certain type that can be dynamically resized. It is a bit like a dynamic array in C, but C++ provides an abstract data type for handling a collection of objects, that is easier and safer to use than C arrays. The Vector implementation takes care of memory management and needed data structures, so the programmer using the vector type does not need to worry about those.

The vector type definition requires an additional specification about what type of objects are stored in the vector. The stored object type is indicated inside pointy brackets ("<" and ">").

Below there are three vectors, for storing

1. integer numbers;
2. strings; and
3. Car - type objects (we assume that in our include header "Car.hpp" we have defined the "Car" type).

```
#include <vector>
#include <string>
#include <Car.hpp>

std::vector<int> numbers = { 1, 2, 3 }; // stores the integers 1, 2 and 3
std::vector<std::string> words; // empty vector that stores strings
std::vector<Car> automobiles; // empty vector that stores Cars
```

Vectors *words* and *automobiles* are empty by default, but we set the initial content of vector *numbers* to contain integers 1, 2 and 3.

New members can be added to the vector using the **push_back** function. Here is an example how:

```
1#include <vector>
2#include <iostream>
3
4int main(void) {
5   std::vector<int> numbers;
6
7   numbers.push_back(5);
8   numbers.push_back(7);
9
10   std::cout << "Size: " << numbers.size() << std::endl;
11
12   return 0;
13}
```

After using the *push_back* function to add integers 5 and 7 to the vector *numbers*, the example outputs the number of objects in the *numbers* vector using the *size* function.

Vectors can be used as function arguments or return values like any other type or class. Below example has function *LargestNumber* that finds the largest value in an int vector:

```
1#include <vector>
2#include <iostream>
3
4int LargestNumber(const std::vector<int>& v) {
5   int largest = -1000; // hmm... -1000 is the smallest number
6   for (unsigned int i = 0; i < v.size(); i++) {
7      if (v[i] > largest)
8         largest = v[i];
9   }
10   return largest;
11}
12
13int main(void) {
14   std::vector<int> numbers = { 1, 2, 3 };
15
16   numbers.push_back(5);
17   numbers.push_back(7);
18
19   std::cout << "Size: " << numbers.size() << std::endl
20      << "Largest: " << LargestNumber(numbers) << std::endl;
21
22   return 0;
23}
```

The *LargestNumber* function takes one *argument*, a reference to int-type vector. The argument is *const*, i.e., the function cannot modify the vector content. The main function shows how the function is called. You can try and see what our little program prints out.

Similarly to C arrays, a subscript operator (`[ ]`) can be used to access a particular element in vector. Use of the subscript operator is not safe when accessing out of bounds elements, and like in C, using invalid index causes a run-time failure. Therefore one needs to be aware of the vector's current length before using it.

Reference of vector class

> Full reference of the vector class.

4.3 Programming Tasks¶

Points **10 / 10** My submissions **5** Deadline Friday, 8 October 2021, 19:59 To be submitted alone

This course has been archived (Saturday, 17 December 2022, 19:59).

Vectors

Objective: Practice basic handling of the vector type.

In this exercise you will need to implement the following three functions that operate on given *int* - type vectors.

- `GetMin` that will return the smallest value stored in the vector.
- `GetMax` that will return the largest value stored in the vector.
- `GetAvg` that will return the average of the values stored in the vector. The returned value will be of *double* type, even though the input values are *int*

Instructions on how to run and test your programs locally are available in Getting Started Module.

vector.cpp

Choose File No file chosen

Submit

Points **15 / 15** My submissions **4** Deadline Friday, 8 October 2021, 19:59 To be submitted alone

This course has been archived (Saturday, 17 December 2022, 19:59).

Vector of strings

Objective: Practice use of C++ strings and vectors.

Implement a program that stores and removes given *strings* from a vector, as commanded by the user. You should implement a simple command line interface that implements the following.

In the beginning, program should print `Commands: ADD, PRINT, REMOVE,QUIT`, followed by a newline. Then print `Enter a command:` followed by a newline. Then the program should read the command from the user and do the following:

- **ADD**: Add a given string to the vector. The program should first prompt `Enter a name:` (without trailing space) followed by a newline. Then it reads the name from the user and adds it to a the vector. Finally, it prints: `Number of names in the vector:`, a newline, the size of the vector, and finally newline. The ADD functionality is implemented in the function `Adder`.
- **REMOVE**: Removes the last string from the vector. This operation is implemented in the function `Remover`. The function should print the removed string in the following way: `Removing the last element:`, followed by a newline, then the string and a newline.
- **PRINT**: Outputs all stored strings, each on a separate line (e.g., followed by newline character). This operation is implemented in function `Printer`.
- **QUIT**: Exit the program.

Note: Adder function needs to be implemented and give full points before any other tests can be run.

In addition to the above three functions, you need to implement the function *CMDReader* that parses the commands and calls the appropriate functions. The detailed function interfaces can be found in the file *vector_strings.hpp*.

Here is an example of an session (Highlighted lines are input):

```
Commands: ADD, PRINT, REMOVE, QUIT
Enter a command:
   ADD
Enter a name:
   Erkki
Number of names in the vector:
   1
Enter a command:
   ADD
Enter a name:
   Tiina
Number of names in the vector:
   2
Enter a command:
   PRINT
Erkki
Tiina
Enter a command:
   REMOVE
Removing the last element:
Tiina
Enter a command:
   QUIT
```

Instructions on how to run and test your programs locally are available in Getting Started Module.

vector_strings.cpp

Choose File No file chosen

Submit

‹ 3 I/O streams Course materials 5 Classes and objects ›