Object-Oriented Programming in C++

Pre-Lecture 3:

Prof Niels Walet (Niels.Walet@manchester.ac.uk)

Room 7.07, Schuster Building

Version: February 2, 2020

Prelecture 3

Outline

In today's lecture we will introduce just two powerful features of C++ in the standard library:

- Strings - store and manipulate groups of characters
- Vectors - store and manipulate groups of any object (but at the moment numbers)

Part 1:
Strings

Strings

The string (an array of characters) is a vital feature of any programming language

- Examples of usage:
  - Names of files to read and write data
  - Parsing text (e.g. counting the number of words, plagiarism checking)
  - Storing human-readable information in databases (e.g. names and addresses, degree courses)
- The historical way to implement strings is by defining, using and manipulating arrays of characters. This is a bit awkward, but you may still meet codes that use this.
- But there are better ways!

Strings as arrays

Strings: char[]

- To define a string as an array of characters, you use an array of type `char`

```
const size_t no_char{100}; // Size of array
char string1[no_char]; //fixed length array to store string
// Or we can do it this way
char *string2;
string2 = new char[no_char];
```

- These character arrays are known as null-terminated strings
- The last character (one byte long) is used for the null character (signifies end of string)
- Null character gets added automatically (but the array must have enough space for the extra character)

Strings: char[]

- Manipulation of `char[]` is done via function calls
- For this we need to include the C string library header (at top of program)

```
#include<cstring>
```

- To define string, need to make use of the `strcpy()` function
- For first two examples

```
strcpy(string1,"This is string1");
strcpy(string2,"This is string2");
```

- Character arrays can be inserted into `cout`

```
std::cout<<string1<<std::endl;
```

Strings: char[]

Common functions to manipulate strings

- Compare two strings with `strcmp`

```
//comparisons
std::cout<<"comparing string1 and string2: ";
if(strcmp(string1,string2))
  std::cout<<"strings are not equal"<<std::endl;
else
  std::cout<<"strings are equal"<<std::endl;
```

Strings: char[]

Common functions to manipulate strings

- Join two strings together with strcat (2nd added to 1st)

```
strcat(string1,string2);
```

- Copy strings with strcpy (2nd copied to first)

```
strcpy(string2,string1);
```

- Determine length of string with strlen

```
std::cout<<"Length of string2 = "<<strlen(string2)<<" "<<strlen("")<<↩
  std::endl;
```

Strings: char[] (complete)

```
4  #include<iostream>
5  #include<cstring>
6  int main()
7  {
8    const size_t no_char{100}; // Size of array
9    char string1[no_char]; //fixed length array to store string
10   // Or we can do it this way
11   char *string2;
12   string2 = new char[no_char];
13   // Or we can initialize our array at the same time
14   char string3[] = "This is string3";
15   // fill arrays with characters by calling strcpy
16   strcpy(string1,"This is string1");
17   strcpy(string2,"This is string2");
18   // Print out strings
19   std::cout<<string1<<std::endl;
20   std::cout<<string2<<std::endl;
21   std::cout<<string3<<std::endl;
22   //comparisons
23   std::cout<<"comparing string1 and string2: ";
24   if(strcmp(string1,string2))
25     std::cout<<"strings are not equal"<<std::endl;
26   else
27     std::cout<<"strings are equal"<<std::endl;
28   std::cout<<"comparing string1 with itself :";
29   if(strcmp(string1,string1))
30     std::cout<<"strings are not equal"<<std::endl;
31   else
32     std::cout<<"strings are equal"<<std::endl;
```

Strings: char[] (output)

```
33    // joining
34    strcat(string1,string2);
35    std::cout<<"Joined string: "<<string1<<std::endl;
36    //copying
37    strcpy(string2,string1);
38    std::cout<<"Copied string: "<<string2<<std::endl;
39    //length
40    std::cout<<"Length of string2 = "<<strlen(string2)<<" "<<strlen("")<<std::endl;
41    return 0;
42  }
```

```
This is string1
This is string2
This is string3
comparing string1 and string2: strings are not equal
comparing string1 with itself :strings are equal
Joined string: This is string1This is string2
Copied string: This is string1This is string2
Length of string2 = 30 0
```

Strings the C++ way

Strings: The C++ standard library way

- C++ provides a `string` datatype[1]
- String manipulation with `string` easier and more powerful: can grow or shrink when required
- To start using strings, we need to include the header

```
#include<string>
```

(so we do not use `cstring` anymore!!)

- We can then define a string as follows

```
string my_first_string{"Hello, world!"};
```

- Its length is worked out as follows

```
std::cout << "Length of string = "<<my_first_string.length()<<std::↩
  endl;
```

- We can also print out a character within string (like an array)

```
std::cout<<"2nd character in string is "<<my_first_string[1]<<std::↩
  endl;
```

[1] actually, it's a class, more next week...

Strings: The standard library way

- Strings can also be defined from the input stream, `cin`. Note: no need to specify length of string in advance!
- Simple way is as usual:

```
std::cout<<"Enter a phrase: ";
std::cin>>input_string;
```

- This line only extracts one word as the input terminates at first whitespace character (leaving the rest of the text and newline in buffer)
- For phrases/sentences, use function `getline` (declared in the `string` header)

```
getline(std::cin,input_string);
```

Note: Need to use `ignore()` when mixing `getline()` and `cin>>`.

- Strings can be compared like other simple data types

```
// match
if(my_first_string == my_second_string)
  std::cout<<"Strings match!"<<std::endl;
```

Strings: The standard library way

- Joining two strings is easy - just add together (overload "+")

```
string joined_string{my_first_string + my_second_string};
std::cout<<"Joined string: "<<joined_string<<std::endl;
```

- Can also append

```
my_first_string +=  my_second_string;
std::cout<<"Appended string: "<<my_first_string<<std::endl;
```

- Extracting substrings (e.g. words) is also easy using `substr`
- Arguments to `substr` are position of the first character (minimum is zero) and (optional) length of substring

```
const size_t first{18};
const size_t last{22};
string a_word = joined_string.substr(first,last-first+1);
```

Strings: The standard library way

```
// PL3/cstringstream.cpp
// Demonstration of strings using the modern syntax
// Niels Walet, last updated 04/12/2019
#include<iostream>
#include<string>
using std::string;
int main()
{
  string my_first_string{"Hello, world!"};
  std::cout<<my_first_string<<std::endl;
  //length
  std::cout << "Length of string = "<<my_first_string.length()<<std::endl;
  //individual character
  std::cout<<"2nd character in string is "<<my_first_string[1]<<std::endl;
  //input
  string input_string;
  std::cout<<"Enter a phrase: ";
  std::cin>>input_string;
  std::cin.ignore();
  std::cout<<"Enter a phrase: ";
  getline(std::cin,input_string);
  //
  string my_second_string{" C++ rocks!"};
  // match
  if(my_first_string == my_second_string)
    std::cout<<"Strings match!"<<std::endl;
  //join
  string joined_string{my_first_string + my_second_string};
  std::cout<<"Joined string: "<<joined_string<<std::endl;
```

Strings: The standard library way

```
30   //append
31   my_first_string +=  my_second_string;
32   std::cout<<"Appended string: "<<my_first_string<<std::endl;
33   // extract
34   const size_t first{18};
35   const size_t last{22};
36   string a_word = joined_string.substr(first,last-first+1);
37   std::cout<<"Extracting characters "<<first<<"-"<<last<<" from joined string: "<<a_word<<std::endl;
38   return 0;
39 }
```

Listing 1 : selection of PL3/cppstring.cpp

Use input

```
Dumbo!
I am Niels
```

Gives output

```
Hello, world!
Length of string = 13
2nd character in string is e
Enter a phrase: Enter a phrase: Joined string: Hello, world! C++ rocks!
Appended string: Hello, world! C++ rocks!
Extracting characters 18-22 from joined string: rocks
```

String Streams

Strings: Streams

- Crucial requirement when dealing with strings: incorporating non-string data (formatting)
- Example: a filename with an integer identifier
- How do we do that?

Strings: Streams

- C++ allows us to use string streams

```
4  #include<iostream>
5  #include<string>
6  #include<sstream>
7  using namespace std;
8  int main()
9  {
10   int file_index{123};
11   ostringstream output_stream;
12   output_stream << "FileData." << file_index;
13   string output_filename{output_stream.str()};
14   std::cout<<output_filename<<std::endl;
15   output_stream.str(""); //clear stream content
16   return 0;
17 }
```

Listing 2 : selection of PL3/cppstringstream.cpp

- Note this can be done as many times as you like - previously inserted data will not be lost! To extract the string from a stringstream we append `.str()` to the stringstream variable (we call the "class function" `str()`)
- We can also use the `str()` function to replace the stringstream "buffer"; this is particularly useful for clearing it

```
ofss.str(""); // empty output string stream buffer
```

Part 2:
Arrays and Vectors

Arrays

Arrays

- Another crucial feature of a programming language is the use of arrays - lists of numbers.
- Obvious use: store a set of indexed (consecutive) data
- Usage is straightforward, e.g.

```
6  int main()
7  {
8    const size_t n_a{5};
9    double a[n_a];
10   for(size_t i{};i<n_a;i++) a[i]=static_cast<double>(i+1);
11   for(size_t i{};i<n_a;i++) std::cout<<"a["<<i<<"] = "<<a[i]<<std::endl↩
       ;
12   return 0;
13 }
```

Listing 3 : selection of PL3/carray.cpp

which yields

```
a[0] = 1
a[1] = 2
a[2] = 3
a[3] = 4
a[4] = 5
```

- One potentially major disadvantage: once arrays are defined their size cannot be

Vectors

C++ standard library extension: vectors

- C++ provides an extension to do this (and much more): vectors

```
3  // Niels Walet, last updated 04/12/2019
4  #include<iostream>
5  #include<vector>
6  #include<string>
7  int main()
8  {
9      const size_t n_a{5};
10     std::vector<double> a(n_a); //length 5 vector of type double
11     std::vector<double> b;  // empty vector (stores doubles)
12     std::vector<int> c(3,1); //length 3 vector of integers, with all ↩
       elements 1
13     std::vector<std::string> names(3); //vector with 3 strings
14     std::vector<std::vector<double> > dd(3); //vector containing 3 ↩
       empty double vectors
15     dd[0]=a;// set the first element of dd to the vector a
16 }
```

Listing 4 : selection of PL3/vector.cpp

- A vector is more like an array than what we call a vector in physics!

C++ standard library extension: vector functions

A number of useful standard functions are part of the vector class:

- `empty()`: A boolean function that tests whether the vector is empty
- `size()`: An integer function; returns length of vector
- `push_back(..)`: Use this to add an element at the end of a vector (extend)
- `pop_back(..)`: Use this to delete element at the end of a vector (shrink)
- `clear()`: Use this to empty the contents of a vector

C++ standard library extension: vector example

```
9   const size_t n_a{5};
10  std::vector<double> a(n_a); //length 5 vector of type double
11  std::vector<double> b;  // empty vector (stores doubles)
12  std::vector<int> c(3,1); //length 3 vector of integers, with all elements 1
13  std::vector<std::string> names(3); //vector with 3 strings
14  std::vector<std::vector<double> > dd(3); //vector containing 3 empty double vectors
15  dd[0]=a;// set the first element of dd to the vector a
16  if(!(a.empty())) std::cout<<"a is not empty"<<std::endl;
17  if(b.empty()) std::cout<<"b is empty"<<std::endl;
18  std::cout<<"a has size: "<<a.size()<<std::endl;
19  std::cout<<"b has size: "<<b.size()<<std::endl;
20  std::cout<<"c has size: "<<c.size()<<std::endl;
21  std::cout<<"names has size: "<<names.size()<<std::endl;
22  std::cout<<"dd has size: "<<names.size()<<std::endl;
23  names[0] = "Vector";
24  names[1] = "of";
25  names[2] = "strings";
26  for(size_t i{}; i<names.size(); ++i)
27    std::cout<<"names["<<i<<"] = "<<names[i]<<std::endl;
28  b.push_back(1.5);
29  b.push_back(3.0);
30  b.push_back(4.5);
31  std::cout<<"b now has size "<<b.size()<<std::endl;
32  for(size_t i{}; i<b.size(); ++i)
33    std::cout<<"b["<<i<<"] = "<<b[i]<<std::endl;
34  b.pop_back();
35  std::cout<<"b now has size "<<b.size()<<std::endl;
36  b.clear();
37  if(b.empty()) std::cout<<"b is now empty"<<std::endl;
38  return 0;
39 }
```

Listing 5 : selection of PL3/vector2.cpp

C++ standard library extension: vector example output

```
a is not empty
b is empty
a has size: 5
b has size: 0
c has size: 3
names has size: 3
dd has size: 3
names[0] = Vector
names[1] = of
names[2] = strings
b now has size 3
b[0] = 1.5
b[1] = 3
b[2] = 4.5
b now has size 2
b is now empty
```

Iterators

C++ standard library extension: iterators

- An alternative way to access vector values is to use an iterator
- An iterator is a special type of pointer to a vector element

```
4  #include<iostream>
5  #include<vector>
6  int main()
7  {
8    std::vector<double> vector_double;
9    vector_double.push_back(4.5);
10   vector_double.push_back(1.5);
11   vector_double.push_back(3.0);
12   std::vector<double>::iterator vector_begin{vector_double.begin()};
13   std::vector<double>::iterator vector_end{vector_double.end()};
14   std::vector<double>::iterator vector_iterator;
15   for(vector_iterator=vector_begin;
16       vector_iterator<vector_end;
17       ++vector_iterator)
18     std::cout<<*vector_iterator<<std::endl;
19   return 0;
20 }
```

Listing 6 : selection of PL3/vector3.cpp

```
4.5
1.5
3
```

output

C++ standard library extension: iterators

- Iterators seem more cumbersome but they apply more widely than just to vectors (e.g., to strings and much more)
- More importantly, they are used in some useful functions
- Example: `sort` and `reverse` order of previous vector (in the `algorithm` header)

```
8  int main()
9  {
10   std::vector<double> double_vector;
11   // Set values of vector by pushing
12   double_vector.push_back(4.5);
13   double_vector.push_back(1.5);
14   double_vector.push_back(3.0);
15   std::vector<double>::iterator vector_begin{double_vector.begin()};
16   std::vector<double>::iterator vector_end{double_vector.end()};
17   // Sort data in ascending order
18   sort(vector_begin, vector_end);
19   std::cout<<"Sorted data:"<<std::endl;
20   std::vector<double>::iterator vector_iterator;
21   for(vector_iterator=vector_begin;vector_iterator<vector_end;++↩
         vector_iterator)
22     std::cout<<*vector_iterator<<std::endl;
23   // Reverse order
24   reverse(vector_begin, vector_end);
25   std::cout<<"Reverse sorted data:"<<std::endl;
26   for(vector_iterator=vector_begin;vector_iterator<vector_end;++↩
         vector_iterator)
27     std::cout<<*vector_iterator<<std::endl;
28   return 0;
29 }
```

Listing 7 : selection of PL3/vector4.cpp

```
Sorted data:
1.5
3
4.5
Reverse sorted data:
4.5
3
1.5
```

output

Conclusions

C++ standard library extension

- Some of you may know (or have guessed) already: vectors and strings are examples of C++ objects
- We met some objects before (`cout`, `cin` and file streams)
- Objects are made from two elements
  - Data and their organisation
  - The operations that can be performed on the objects (e.g. calculate size of a vector)
- These rules are collected together within a class
- The vector and string classes are pre-defined as part of the C++ Standard Library [2]
- In the next lecture, we will learn how to write our own classes.

[2] The vector class, known as a container class, is part of the Standard Template Library (STL) that also includes lists, sets and maps

The END