Object-Oriented Programming in C++

Pre-Lecture 10: Advanced topics

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

Room 7.07, Schuster Building

Version: March 22, 2020

Prelecture 10

- C++ is supported by a rich library of functions and classes - the C++ standard library
- We have already made use of the C++ standard library
  - I.O: standard (keyboard and screen) `<iostream>`; file `<fstream>`; manipiulations `<iomanip>`
  - data structures: strings `<string>` and variable length arrays `<vector>`
- Standard library routines are declared in the `std::` namespace
- Can split library into 4 sections
  - 1 C standard library
  - 2 Standard Template Library
  - 3 I/O stream library
  - 4 Miscellaneous libaries
- On top of that one often uses the boost library–truly enormous, and very useful in many cases! (http://www.boost.org)

Standard library

C++ Standard library

- 1 C standard library: since C++ is super-set of C - no need to re-write some useful functions (e.g. in `cmath` and `string`)
- 2 Standard Template Library: useful set of template classes, e.g.
  - Container classes (standardised data structures such as `vector`, `list`, `deque`, `map`, `set` etc.)
  - Algorithms (`copy`, `find`, `sort`, `reverse` etc.)
  - Iterator (for defining own iterator class)
  - Numeric (including `complex`)
- 3 I/O Stream library: streams including `ostream` and `fstream`
- 4 Miscellaneous libraries: includes `string` class and `utility` library
- Have a look at: `http://www.cplusplus.com/reference` for full details
- Full details beyond scope of course - will instead provide a few examples (`vector` and `string` already covered)

C++ Standard library

complex numbers

- We wrote a basic class to handle complex numbers but C++ already provides one (sorry!)
- Part of the C++ STL
- Class template (specialized for `float`,`double`,`long double`)
  `template <class T> class complex;`
- Can do all the usual operations, e.g.

```
complex<double> z1(1,1),z2(1,-1); // two new complex numbers
cout<<"z1 = "<<z1<<endl; // (1,1)
cout<<"z2 = "<<z2<<endl; // (1,-1)
cout<<"z1 + z2 = "<<z1+z2<<endl; // (2,0)
cout<<"z1 * z2 = "<<z2*z2<<endl; // (0,-2)
cout<<"Real part of z1 = "<<z1.real()<<endl; // 1
cout<<"Imag part of z2 = "<<z2.imag()<<endl; // -1
cout<<"Modulus of z1 = "<<abs(z1)<<endl; // 1.41421
cout<<"Argument of z1 = "<<arg(z1)<<endl; // 0.
```

- See any STL reference book (or online) for further information

C++ Standard library

pairs

- A useful class when have close association of two object types (e.g. `double` and `string`)
- Defined in `utility` library `#include<utility>`
- Example

```
pair<double,string> obj1; // Declare pair object
obj1.first  = 1.; // Define first part of pair
obj1.second = "Object1"; // and second
cout<<"First part: "<<obj1.first<<endl;
cout<<"Second part: "<<obj1.second<<endl;
```

- Output

```
First part: 1
Second part: Object1
```

- Used for `map` class (see later)

C++ Standard library

container classes

- A set of class templates within STL
- Designed to hold many objects within a single container
- Three types
  - 1 Sequence containers: elements ordered in strict linear sequence: `vector`, `list`, `deque` (access time linear)
  - 2 Associative containers: elements accessed using a key (not position in sequence): `set`, `multiset`, `map`,`multimap`, `bitset`
  - 3 Container adapters: provide specific interface for some of the above (e.g., `queue`; not covered here)
- Also note many more in latest C++ standard!
- Elements of container classes accessed using iterators
- Let us look at a few examples

C++ Standard library

sequence containers

- Already met `vector` (a dynamic array)
- Vectors allocate contiguous memory allowing random access - strict linear relation between element and its memory address
- Another sequence container is the `list`, where each element instead uses two pointers (to previous and next element)
- Faster than vectors for adding, subtracting and organising elements (e.g. sorting)
- More awkward to find individual element (must use iterator or `search()` algorithm)

C++ Standard library

list example

```
1  // PL10/listdemo.cpp
2  // Application of the list container class
3  // Niels Walet. Last edited 03/12/2019
4  #include<iostream>
5  #include<list>
6  void print_list(std::list<int> &list_in)
7  {
8    std::cout<<"List contents: ";
9    for(auto li=list_in.begin();li!=list_in.end();++li)
10     std::cout<<*li<<" ";
11   std::cout<<std::endl;
12 }
13 int main()
14 {
15   std::list<int> my_list;
16   // Push some on the front
17   my_list.push_front(1);
18   my_list.push_front(2);
19   // and some on the back
20   my_list.push_back(3);
21   my_list.push_back(4);
22   print_list(my_list);
23   // Use iterator to identify current position in ↩
       list
24   std::list<int>::iterator li;
25   // Insert a new entry in middle of current list
26   li=my_list.begin();
27   for(int i{};i<2;i++) li++;
28   my_list.insert(li,5);
29   print_list(my_list);
30   // Sort list
31   my_list.sort();
32   print_list(my_list);
33   // Declare a second list
34   std::list<int> my_list2;
35   for(int i{};i<3;i++) my_list2.push_back(9-i);
36   print_list(my_list2);
37   // Merge two lists and re-sort
38   my_list.merge(my_list2);
39   my_list.sort();
40   print_list(my_list);
41   // Remove first and last entries
42   my_list.pop_front();
43   my_list.pop_back();
44   print_list(my_list);
45   return 0;
46 }
```

Listing 1 : selection of PL10/listdemo.cpp

C++ Standard library

list example

- Code outputs:

```
List contents: 2 1 3 4
List contents: 2 1 5 3 4
List contents: 1 2 3 4 5
List contents: 9 8 7
List contents: 1 2 3 4 5 7 8 9
List contents: 2 3 4 5 7 8
```

- Original list is line 1 (note 2 comes before 1)
- Line 2 after using `insert` with iterator
- Line 3 after sorting list
- Line 4 is 2nd list
- Line 5 is new merged and sorted list
- Line 6 is this new list with first and last element deleted

C++ Standard library

associative containers

- Associative containers allow one to find entries by association
- Example: the `map` class template - takes a pair of object types (key and data)
- Overloads `operator[]` to use key instead of position in array (memory)
- Useful method for accessing textual information stored with a key–a simplified database

C++ Standard library

map example

```
1  // PL10/mapdemo.cpp
2  // illustrates the use of the map containetr class
3  // Niels Walet. Last edited 03/12/2019
4  #include<iostream>
5  #include<string>
6  #include<utility>
7  #include<map>
8  // Use alias for our type of map
9  typedef std::map<int,std::string> ↩
       international_dial_codes;
10 void search_database(international_dial_codes &↩
       dial_codes, int code_search)
11 {
12   international_dial_codes::iterator dial_codes_iter;
13   dial_codes_iter = dial_codes.find(code_search);
14   if(dial_codes_iter != dial_codes.end())
15     std::cout<<"Found country for dial code "
16          <<code_search << " = "
17          <<dial_codes_iter->second<<std::endl;
18   else
19     std::cerr<<"Sorry, code " << code_search
20          <<" is not in database"<<std::endl;
21 }
22 int main()
23 {
24   // Using map associative container class
25   // (use key to access data)
26   // Example: international dial codes
27   international_dial_codes dial_codes;
28   // New entries using []
29   dial_codes[49] = "Germany";
30   dial_codes[44] = "United Kingdom";
31   // Can also insert a pair
32   dial_codes.insert(std::pair<int,std::string>(672,"↩
       Christmas Island"));
33   // How many entries so far?
34   std::cout<<"Size of database = "<<dial_codes.size()↩
       <<std::endl;
35   // Print out database - note sorted by codes!
36   international_dial_codes::iterator dial_codes_iter;
37   for(dial_codes_iter  = dial_codes.begin();
38       dial_codes_iter != dial_codes.end();
39       ++dial_codes_iter)
40     std::cout<<"Dial code: " << dial_codes_iter->↩
         first
41          <<", country: " << dial_codes_iter->second ↩
       << std::endl;
42   // What country has code 672? Let's find out (uses ↩
       iterator)
43   int code_search(672);
44   search_database(dial_codes,code_search);
45   // Again for a code not stored
46   code_search = 673;
47   search_database(dial_codes,code_search);
48   return 0;
```

Listing 2 : selection of PL10/mapdemo.cpp

C++ Standard library

- Code outputs

```
Size of database = 3
Dial code: 44, country: United Kingdom
Dial code: 49, country: Germany
Dial code: 672, country: Christmas Island
Found country for dial code 672 = Christmas Island
```

- Note order of output - `map` sorts data (incremental order) based on key
- As such, the `<` operator must be defined for key datatype (otherwise must define)

C++ Standard library

algorithms

- Containers are often well used with the `<algorithm>` header
- Things like `find` already seen
- But there are many more
- See, e.g., `http://www.cplusplus.com/reference/algorithm/`

Exceptions

Exception handling

- Detecting and handling exceptions (run-time errors) is a key part of writing any program
- Especially true when using dynamic memory management
- Could even be vital for mission-critical software
- C++ provides a neat method for this: `try`, `throw`, `catch`
- The `try` keyword is used to look for exceptions and `catch` to decide what to do with them

```
try
   {       // Our code being monitored goes here
   }
catch(int errorFlag)
   {       // Here we can take action
   }
```

- If an exception is detected, we `throw` it - program transfers immediately to `catch`

```
 if(error) throw(myErrorFlag);
```

- some of you may already have encountered the `noexcept` keyword

Exception handling

```
4  #include<iostream>
5  #include<cstdlib> //for exit
6  const int divide_flag(-1);
7  double divide(double x, double y)
8  {
9    if(x==0) throw divide_flag;
10   return y/x;
11 }
12 int main()
13 {
14   double x{3.},y{4.};
15   double result;
16   try {
17       result=divide(x,y);
18       std::cout<<"y/x = "<<result<<std::endl;
19       x=0;
20       result=divide(x,y);
21       std::cout<<"y/x = "<<result<<std::endl;
22     }
```

```
23   catch(int error_flag) {
24       if(error_flag == divide_flag) {
25       std::cerr<<"Error: divide by zero"<<std::endl;
26           exit(error_flag);
27         }
28     }
29   return 0;
30 }
```

Listing 3 : selection of PL10/exceptiondemo.cpp

output

```
y/x = 1.33333
```

Exception handling

notes

- You can have multiple catch statements for different datatypes

```
catch(int errorFlag) { ... }
catch(double exceptionDouble) { ... }
```

  where the appropriate one will be called (based on the type that is thrown)
- Make sure you implement an appropriate catch for every throw
- When exception thrown, program exits `try` construct - everything within there is reset
- If throwing an object instantiated from derived class, catch it first (before base class objects) otherwise can still be caught by base class `catch` statement

```
// Wrong - object from derivedClass will be caught by first catch
catch (baseClass B) { ... }
catch (derivedClass D) { ... }
```

Exception handling

another example

- You are advised to use exception handling when allocating memory
- If this fails an exception will be thrown of type `bad_alloc`

```
4  #include<iostream>
5  #include<memory>
6  int main() {
7    double *my_array;
8    try
9      {
10       my_array = new double[10000000000000000];
11     }
12   catch(bad_alloc memFail)
13     {
14       std::cerr<<"Memory allocation failure"<<std::endl;
15       return(1);
16     }
17   delete[] my_array;
18   return 0;
19 }
```

Listing 4 : selection of PL10/badalloc.cpp

Smart pointers

Smart pointers

Smart pointer: reminder

- Smart pointers are crucial to the RAII or Resource Acquisition Is Initialialization programming idiom. The main goal of this idiom is to ensure that resource acquisition occurs at the same time that the object is initialized, so that all resources for the object are created and made ready in one line of code.
- In practical terms, the main principle of RAII is to give ownership of any allocated resource—for example, dynamically-allocated memory or system object handles—to an lvalue object whose destructor contains the code to delete or free the resource and also execute any associated cleanup code.
- In most cases, when you initialize a raw pointer or resource handle to point to an actual resource, you can actually pass the pointer to a smart pointer immediately. In modern C++, raw pointers should only used in small code blocks of limited scope, loops, or helper functions where performance is critical and there is no chance of confusion about ownership.
- In your final project I would like to see **no** raw pointers, unless they never leave a small scope; otherwise use smart pointers!

Smart pointers

Smart pointer

C++ Standard Library Smart Pointers need the header `<memory>`.

Use these smart pointers as a first choice for encapsulating pointers to plain old C++ objects.

unique_ptr Allows exactly one owner of the underlying pointer. Use this as the default choice, unless you know for certain that you require a `shared_ptr`. Can be moved to a new owner, but not copied or shared. Replaces the older syntax `auto_ptr`, which is now deprecated. `unique_ptr` is small and efficient; the size is one pointer and it supports rvalue references for fast insertion and retrieval from STL collections.

shared_ptr Reference-counted smart pointer, as we have implemented ourselves earlier in the course. Use when you want to assign one raw pointer to multiple owners. The raw pointer is not deleted until all `shared_ptr` owners have gone out of scope or have otherwise given up ownership.

weak_ptr Special-case smart pointer for use in conjunction with `shared_ptr`. A `weak_ptr` provides access to an object that is owned by one or more `shared_ptr` instances, but does not participate in reference counting. Use when you want to observe an object, but do not require it to remain alive. Required in some cases to break circular references between `shared_ptr` instances.

Lambda functions

Lambda functions

- I will only give limited detail on lambda functions (technicallu, lamda closures). A very good longer discussion is at http://www.cprogramming.com/c++11/c++11-lambda-closures.html.
- What is a “lambda”? It is an anonymous (unnamed) function–there are other rather complex ways in C++ to do what a lambda does, but lambdas are elegant, and easy to understand

```
8  int main()
9  {
10   std::vector<int> v;
11   for (int i{}; i < 10; i++) v.push_back(i+1);
12   // Count the number of even numbers in the vector
13   int even_count = 0;
14   std::for_each(v.begin(), v.end(),
15         [&even_count] (int n)
16         {std::cout << n;
17           if (n % 2 == 0) {
18             std::cout << " is even " << std::endl;
19             ++even_count;
20           } else {
21             std::cout << " is odd " << std::endl;
22           }
23         });
24   // Print the count of even numbers to the console.
25   std::cout << "There are " << even_count
26         << " even numbers in the vector." << std::endl;
27 }
```

Listing 5 : selection of PL10/lambda1.cpp

Lambda functions

- The code use the `for_each` syntax as found in the algorithm header: it applies a function to each argument between begin and end.
- The lambda function definition (header) is on line 15,

```
15          [&even_count] (int n)
```

- The square bracket part says it has reference access to `eventCount` (i.e., can see and change its value)
- The parenthesis part shows that it takes an integer as argument
- Lines 16-22 are the function body
- The only alternative to this coding is using functors (see https://stackoverflow.com/questions/356950/what-are-c-functors-and-their-uses). That is so much more complicated that I don't even want to show it!

Lambda functions

- The take away message is that lambda and the standard library allow for very powerful programming!
- The important part is what and how variables are captured (made available) with the square brackets:
  - `[]` Capture nothing
  - `[&]` Capture any variable used by reference (so we don't have to specifiy!)
  - `[=]` Capture any variable used by value
  - `[=,&x]` Capture any variable used by value, but x by reference
  - `[x]` Capture x by value, don't capture anything else
- The return type of a lambda is void by default; for a simple return the compiler will work out what the return type is

```
[] ()  { return 1; }// an int
```

or use the return value syntax

```
[] () -> double { return 1; }
```

At the end

Course summary

- In this course you were introduced to the main concepts of Object Oriented Programming
- Key concepts to understand:
  - Classes (the rules) and objects (the instances)
  - Encapsulation - we control how data are used
  - Inheritance - creating class super-structures
  - Polymorphism - one interface, multiple methods
  - Class and function templates - structures with generic types
  - Organising code - multiple files, headers and namespaces
  - Good practice - comments; handling exceptions
- C++ the beginning - many languages use OOP (Java, C#, Python, Ruby, ...)
- And some are based on C++ (Rust, Go(lang), . . .)
- Standardised C++17 has been around for three years (most if not all of its new features already in latest compilers)
- but next update C++20 is on the horizon already!
- Remember: secret to good programming is practice