```
1   /*****************************************************************************
2
3   Simple examples STL collection basics.
4
5   Very Short Version:
6    Search for #TODO marks with questions and answer them. (Create answer doc.) :)
7
8   Short Version:
9    The #TODO marks are questions - answer them in order in a lab notes document. As
10   you go save your changes and commit to your repo as evidence of your work.
11
12  Longer Version (recommended steps):
13      1. Put this file in your repo folders and commit with a message like "No changes
14         yet" or similar.
15      2. Open this file in your IDE that *must* support debugging. (VS, CLion etc)
16      3. Build and run this file. Make sure this works before going on.
17      4. Create a lab-notes document for your answers. (Paste the questions provided
18         if you want, or discover them as you go.) Save and commit.
19      5. Work your way through each of the file #TODO Q. marks in order.
20         - Work one section at a time. There are if (false) { ... } blocks to change to
21           if (true) get the code to run.
22         - Suggest committing your lab notes to your repo as you do each section (minimum).
23           This will make a good history of evidence for you!
24         - Suggest turning each finished section back to "false" (as it will probably
25           speed up compiling time, depending on your setup.)
26         - This code file may not change much as you go (simple false/true changes) so you
27           may not need to commit it much (depending on what extra changes you decide to make).
28
29  There is a simple text file with the questions listed if that helps you with your lab notes.
30
31  Written by Clinton Woodward <cwoodward@swin.edu.au>
32   for COS30031 Games Programming
```

```
33
34  This file is for your personal study use only and must not be shared or made
35   publicly available.
36
37  Updates
38   2020-07-08: Cleanup, new questions and comment help.
39
40  NOTE: this example code uses C++11 extensions ("auto" in particular)
41  so you may need to tell your compiler to use at least this standard with a
42  flag, for example, -std=C++11 or similar.
43
44  */
45
46
47  #include <iostream>
48  #include <array>
49  #include <vector>
50  #include <stack>
51  #include <queue>
52  #include <list>
53  #include <algorithm>
54
55  using namespace std;
56
57  struct Particle {
58      int x, y;
59  };
60
61  class ParticleClass {
62  public:
63      int x, y;
64
65      // ParticleClass(); // works, but random (not set) initial values
66      ParticleClass() {
67          x = 0;
68          y = 0;
69          cout << " - ParticleClass() (default) constructor called" << endl;
70      }; // default, called by collections
71      ParticleClass(int x, int y) {
72          this->x = x;
73          this->y = y;
74          cout << " - ParticleClass(x,y) constructor called" << endl;
75      }
76
77      void show() const {
78          cout << " - ParticleClass: (" << x << ", " << y << ")" << endl;
79      }
80
81      ~ParticleClass() {
82          cout << " - ~ParticleClass() destructor called" << endl;
```

```
83          }
84      };
85
86      // Forward declaration of simple functions
87      void array_demo_1();
88      void array_demo_2();
89      void array_demo_3();
90
91      void stack_demo();
92
93      void queue_demo();
94
95      void list_demo();
96
97      void vector_demo();
98
99      void showIntArray(const array<int, 3> &arr);
100
101     int main() {
102         // Uncomment each of the _demo function to investigate!!
103
104         //array_demo_1();
105         // Q.1 Questions are inside array_demo_1 - answer them there.
106
107         //array_demo_2();
108         // Q.2 In array_demo_2, explain what a4(a1) does
109
110         //array_demo_3();
111         // Q.3 No questions for array_demo_3, it's just a demo of Struct/Class use with array.
112
113         //stack_demo();
114         // Q.4 How do we (what methods) add and remove items to a stack?
115         // Q.5 A stack has no no [] or at() method - why?
116
117         //queue_demo();
118         // Q.6 What is the difference between a stack.pop() and a queue.pop() ?
119
120         //list_demo();
121         // Q.7 Can we access a list value using and int index? Explain.
122         // Q.8 Is there a reason to use a list instead of a vector?
123
124         vector_demo();
125         // Q.9 Was max_size and size the same? (Can they be different?)
126         // Q.10 Which ParticleClass constructor was called?
127         // Q.11 Were the ParticleClass instances deleted? If so, how?
128         // Q.12 Was the vector instance deleted? If so, how do you know this?
129         // Q.13 Your IDE might suggest to use emplace_back instead of push_back. What does this mea
130
131         cout << 'c' << endl;
132         return 0;
```

```
    return 0;
}

void showIntArray(const array<int, 3> &arr) {
    // #TODO: Extension: Apparently const prevents a copy - quicker performance.
    //  Is this true? How could you tell or what ref/url supports your view?
    cout << " - array<int, 3> contents: ";
    for (int i = 0; i < arr.size(); i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
}

void array_demo_1() {
    // std::array
    // A templated class for "fixed size" arrays (with known internal buffer)
    // - prevents "decay" usage into a pointer (unlike [] types)
    // - maintains array size for us (fixed)
    // - bounds checking
    // - C++ container operations size, begin, end ...
    //   (except size-changing push/pop etc)
    // - can be passed *by value* to a function (others can't)
    //
    // methods?
    //  - iterators: begin, end, rbegin, fend
    //  - capacity: size, max_size, empty
    //  - access: front, back, [], at()
    //  - modifiers: swap
    //
    // Note
    //  - the at() supports bounds checking, [] does not
    //  - the at(index) is range protected (but slower due to getter)

    cout << " << std:array demos!>>" << endl;
    // simple quick std::array example
    array<int, 3> a1 = {8, 77, -50}; // initializer list
    // Q.1.1 What do the < and > mean or indicate?
    // Q.1.2 Why don't we need to write std:array here? (Is this good?)
    // Q.1.3 Explain what the int and 3 indicate in this case?

    if (false) {
        cout << "a1 address: " << hex << &a1 << endl;
        cout << dec; // put back to decimal mode (after being in hex)
        cout << "a1 size: " << a1.size() << endl;
        // Note: array size is fixed when created, so max_size == size
        cout << "a1 max_size: " << a1.max_size() << endl;
        // access (read/write) of elements? using [index]
        cout << "reading a1[0]: " << a1[0];
        cout << "altering ...";
        a1[0] = 42; // the answer
```

```
182             cout << " it is now: " << a1[0] << endl;
183         }
184 
185         // Show contents using for, iterator and foreach
186         if (false) {
187             // ... using plain-old for loop (int i position)
188             cout << "a1 contents using plain-old for loop: ";
189             for (int i = 0; i < a1.size(); i++) {
190                 cout << a1[i] << " ";
191             }
192             cout << endl;
193 
194             // ... using explicit C++ std templated iterator
195             cout << "a1 contents using a templated iterator: ";
196             array<int, 3>::iterator itr;
197             for (itr = a1.begin(); itr < a1.end(); itr++) {
198                 cout << *itr << " ";
199             }
200             cout << endl;
201 
202             // ... using auto to get iterator (whew - much easier)
203             cout << "a1 contents using auto provided template iterator: ";
204             for (auto itr2 = a1.begin(); itr2 < a1.end(); itr2++) {
205                 cout << *itr2 << " ";
206             }
207             cout << endl;
208             // Q.1.4 In the code above, what is the type of itr2?
209 
210             //
211             cout << "a1 contents using auto & for-each iterator: " << endl;
212             for (auto &v : a1)
213                 cout << v << " ";
214             cout << endl;
215             // Q.1.5 In the code above, what is the type of v?
216             // Q.1.6 In the code above, what does the & mean in (auto &v : a1)
217 
218             // pass to a function (by value, using const to ensure it is not copied)
219             showIntArray(a1);
220         }
221 
222         // Q.1.7 Try this. Why does a1[3] work but at(3) does not?
223         if (false)
224         {
225             // access of array by [index] is not range protected (BAD)
226             cout << "What is at [3]? (out of bounds) " << a1[3] << endl;
227             cout << "What is at(3)? (out of range exception) " << a1.at(3) << endl;
228 
229         // let's use some other container methods
230         cout << "front() == " << a1.front() << endl;
231         cout << "back() == " << a1.back() << endl;
```

```
232         //cout << "empty() == " << a1.emtpy() << endl; // Hmm! empty() work? try it
233
234         // #TODO: Extension. Create examples of swap() and fill()
235         // a1.swap(s2) and a1.fill(value) also.
236
237         // iterator for loop
238         // Q.1.8 auto is awesome. What is the actual type of v that it works out for us?
239         cout << "Using for with iterator ... " << endl;
240         for (auto v = a1.begin(); v != a1.end(); v++)
241             cout << " " << v;
242         cout << endl;
243
244         // iterator for-each loop
245         // Q.1.9 auto is still awesome. What is the actual type of v here?
246         cout << "Using for-each (ranged) iterator ... " << endl;
247         for (auto &v : a1)
248             cout << " " << v;
249         cout << endl;
250
251         // sort?
252         sort(a1.rbegin(), a1.rend());
253         cout << "Reverse Sort() on a1, now ..." << endl;
254         showIntArray(a1);
255         // Q.1.10 How would you do a forward (not reverse) sort?
256         reverse(a1.begin(), a1.end());
257         cout << "Forward Sort() on a1, now..." << endl;
258         showIntArray(a1);
259
260         // multidimensional array (note the dimension order)
261         array<array<int, 2>, 4> a_2d = {{{1, 2}, {3, 4}, {5, 6}, {7, 8}}};
262         cout << "2d array access a_2d[2][0] == " << a_2d[2][0] << endl;
263         // Hmm ... Vote - are multi-dimensional arrays pretty to create?
264
265         cout << " done." << endl;
266         }
267     }
268
269     void array_demo_2() {
270         if (true) {
271             // array of 5 ints, must state size
272             array<int, 5> a1;
273             array<int, 4> a2 = {-4, 2, 7, -100};
274
275             cout << "a1 " << hex << &a1 << " " << a1.size() << endl;
276             cout << "a2 " << hex << &a2 << " " << a2.size() << endl;
277
278             // new array via copy
279             auto a3 = a2; // this is a copy
280             // if auto doesn't work (C++11 extension) either configure your compiler
```

```
281             // or state the type explicitly. (VS2010+ should support it, etc)
282             // - array<int, 4> a3 = a2; // equivalent to auto
283             // - array<int, 4> z1 = a1; // compile error - different length
284 
285             cout << "a3 " << hex << &a3 << " " << a3.size() << endl;
286             auto a4(a1); // this works too
287             cout << "a4 " << hex << &a4 << " " << a4.size() << endl;
288         }
289     }
290 
291     void array_demo_3() {
292         if (true) {
293             // Array of struct Particles
294             array<Particle, 3> a1; // random/not initialised values
295             array<Particle, 3> a2{}; // initialised values to 0, can write = {} also
296 
297             // old school for loop (clear, simple, quick)
298             // note: initial values may be random - struct has no default initialiser
299             cout << "a1 array of Particles ..." << endl;
300             for (int i = 0; i < a1.size(); i++)
301                 cout << " - Particle: " << i << " (" << a1[i].x << ", " << a1[i].y << ")" << endl;
302 
303             cout << "a2 array of Particles, initialised, using for-each ..." << endl;
304             for (auto &p: a2)
305                 cout << " - Particle: (" << p.x << ", " << p.y << ")" << endl;
306         }
307 
308         if (true) {
309             array<ParticleClass, 3> a1;
310 
311             cout << "Show a1 array of ParticleClass instance details ... " << endl;
312             for (auto &p: a1)
313                 cout << " - ParticleClass: (" << p.x << ", " << p.y << ")" << endl;
314 
315             cout << "Show a1 array of ParticleClass instance details using show() ... " << endl;
316             for (auto &p: a1)
317                 p.show();
318         }
319     }
320 
321     void stack_demo() {
322         // stack (LIFO, container adaptor)
323         // - empty, size, back, push_back, pop_back (standard container)
324         // - top, push, pop (no [] or at() ...)
325         // - will use a deque if container type not specified
326         stack<int> s1;
327         // push some values onto the stack, last on top()
328         cout << "Stack (LIFO) ... " << endl;
329         for (int i = 0; i < 5; ++i) s1.push(i);
330 
```

```
331         cout << "Removing stack elements with pop() ...";
332         while (!s1.empty()) {
333             cout << ' ' << s1.top(); // last added (newest)
334             s1.pop();
335         }
336         cout << endl;
337     }
338 
339     void queue_demo() {
340         // queue (FIFO, container adaptor)
341         // - empty, size, back, push_back, pop_back (standard container)
342         // - front, back, push, pop (no [] or at() ...)
343         // - will use a deque if container type not specified
344         queue<int> q1;
345         // push some values onto the stack, last on top()
346         cout << "Queue (FIFO) ... " << endl;
347         for (int i = 0; i < 5; ++i) q1.push(i);
348 
349         cout << "Removing queue elements with pop() ...";
350         while (!q1.empty()) {
351             cout << ' ' << q1.front();
352             q1.pop(); // front (first, or oldest), not last
353         }
354         cout << endl;
355     }
356 
357     void list_demo() {
358         // std::list
359         // A sequence container (internally, a double-linked list)
360         // - specialised for constant time insert/erase at any position
361         // - good at insert, extract, move but uses iterator (not uint index)
362         // - house-keeping overhead (link details)
363         // - iteration in either direction
364         list<int> l1;
365         list<int>::iterator it;
366 
367         // set some initial values:
368         cout << "List (double-linked list) ... " << endl;
369         for (int i = 1; i <= 5; ++i) l1.push_back(i); // 1 2 3 4 5
370 
371         cout << " - list contains:";
372         for (auto &i: l1) cout << " " << i;
373         cout << endl;
374 
375         // modify
376         cout << "Insert using iterator access (end() - 1)" << endl;
377         it = l1.end();
378         --it;
379         l1.insert(it, 77);
```

```
380
381         // show inserted element
382         cout << " - list contains:";
383         for (auto &i: l1) cout << " " << i;
384         cout << endl;
385
386         // sort?
387         cout << "Sort list (using default compare) ... " << endl;
388         l1.sort();
389         cout << " - list contains:";
390         for (auto &i: l1) cout << " " << i;
391         cout << endl;
392     }
393
394     void vector_demo() {
395         // std::vector
396         // A templated class for "dynamic size" arrays
397         // - maintains array size for us, (can use pointer offset still)
398         // - bounds checking and resize/memory management (+overhead cost)
399         // - C++ container operations (size, begin, end ... )
400
401         // methods?
402         //  - iterators: begin, end, rbegin, rend (+const iterators)
403         //  - capacity: size, max_size, empty, resize, shrink_to_fit, capacity, reserve
404         //  - access: front, back, [], at()
405         //  - modifiers: assign, emplace*, insert, erase, emplace_back*,
406         //               push_back, pop_back, clear, swap
407
408         if (true) {
409             // simple quick vector example
410             vector<int> v1 = {8, 77, -50}; // initializer list
411             cout << "v1 address: " << hex << &v1 << endl;
412             cout << dec; // put back to decimal mode (after being in hex)
413             cout << "v1 size: " << v1.size() << endl;
414             // vector size is not fixed, so max_size <> size (typically)
415             cout << "v1 max_size: " << v1.max_size() << endl;
416
417         }
418
419         if (true) {
420             vector<ParticleClass> v1;
421
422             v1.push_back(ParticleClass(1, 2));
423             v1.push_back(ParticleClass(3, 4));
424             v1.push_back(ParticleClass(5, 6));
425
426             cout << "Show v1 vector of ParticleClass instance details using show() ... " << endl;
427             for (auto &p: v1)
428                 p.show();
429         }
```

```
430
431     }
```