```
1  //Sample provided by Fabio Galuppo
2  //http://member.acm.org/~fabiogaluppo
3  //fabiogaluppo@acm.org
4  //November 2013
5
6  #include <iostream>
7  #include <sstream>
8
9  #include <algorithm>
10 #include <functional>
11 #include <type_traits>
12 #include <tuple>
13 #include <string>
14 #include <limits>
15
16 #include <cstring>
17 #include <cassert>
18 #include <ctime>
19
20 #include <vector>
21 #include <array>
22 #include <deque>
23 #include <initializer_list>
24 #include <array>
25 #include <unordered_map>
26
27 #include <random>
28 #include <memory>
29
30 #include <chrono>
31 #include <thread>
32 #include <future>
33 #include <atomic>
34
35 //Supported C++ 11 features in Visual C++ 2013
36 //http://msdn.microsoft.com/library/vstudio/hh567368(v=vs.120).aspx
37
38 ////////////////////////
39 //1. Raw Literal Strings
40 void Feature1()
41 {
42     const char* html = R"(
43     <html>
44     <body>
45     <h1>This is Raw Literal String in C++</h1>
46     <h2>This is Raw Literal String in C++</h2>
47     <h3>This is Raw Literal String in C++</h3>
48     <h4>This is Raw Literal String in C++</h4>
49     <h5>This is Raw Literal String in C++</h5>
50     <h6>This is Raw Literal String in C++</h6>
51     </body>
52     </html>
53     )";
54
55     const wchar_t* wstr = LR"(Hello, World!!!)";
56
57     std::cout << "content: " << html << "length: " << std::strlen(html) << std::endl;
58     std::wcout << "content: " << wstr << " length: " << std::wcslen(wstr) << std::endl;
59 }
60
61 ////////////////////////
62 //1. Default Template Arguments for Function Templates
63 //2. Range-based for-loop (range-for)
64 //3. Initializer Lists/Uniform Initialization
65 template<typename Container>
66 void container_display(const Container& xs)
```

```
67 {
68     std::cout << "{ ";
69     for (auto& x : xs)
70         std::cout << x << " ";
71     std::cout << "}" << std::endl;
72 }
73
74 template <typename Container,
75     typename Predicate = std::less<typename Container::value_type >>
76     void container_sort(Container& c, Predicate p = Predicate())
77 {
78         std::sort(std::begin(c), std::end(c), p);
79     }
80
81 void Feature2()
82 {
83     std::vector<int> xs{ 1, 2, 3, 4, 5 };
84
85     container_sort(xs, std::greater<int>());
86     container_display(xs);
87
88     container_sort(xs);
89     container_display(xs);
90 }
91
92 ////////////////////////
93 //1. Delegating Constructs
94 //2. Brace Initialization
95 //3. Initialization List
96 //4. Aliases
97 //5. Explicitly Defaulted and Deleted Functions
98 template <typename Container>
99 struct random_access_container_wrapper
100 {
101     using size_type = typename Container::size_type;
102     using const_reference = typename Container::const_reference;
103     using reference = typename Container::reference;
104     //or
105     //typedef typename Container::size_type size_type;
106     //typedef typename Container::const_reference const_reference;
107     //typedef typename Container::reference;
108
109     random_access_container_wrapper(std::string description, size_type size) :
110         description(description), c(Container(size))
111     {
112     }
113
114     random_access_container_wrapper(const char* description) :
115         random_access_container_wrapper(description, 64)
116     {
117     }
118
119     random_access_container_wrapper(size_type size) :
120         random_access_container_wrapper("", size)
121     {
122     }
123
124     template<typename U>
125     random_access_container_wrapper(std::initializer_list<U> il) :
126         random_access_container_wrapper(il.size())
127     {
128         std::copy(il.begin(), il.end(), c.begin());
129     }
130
131     const_reference operator[](size_type pos) const
132     {
```

```
133         return *(c.begin() + pos);
134     }
135
136     reference operator[](size_type pos)
137     {
138         return *(c.begin() + pos);
139     }
140
141     using self = random_access_container_wrapper;
142
143     random_access_container_wrapper(const random_access_container_wrapper&) = delete;
144     //or
145     //self(const self&) = delete;
146
147     self& operator=(const self&) = delete;
148
149     /*random_access_container_wrapper() :
150     random_access_container_wrapper("")
151     {
152     }*/
153     self() : self("")
154     {
155     }
156
157     ~random_access_container_wrapper() = default;
158
159 private:
160     Container c;
161     std::string description;
162 };
163
164 template<typename T>
165 using racw_with_vector = random_access_container_wrapper<std::vector<T>>;
166
167 struct color_pod
168 {
169     signed short r, g, b;
170 };
171
172 //noncopyable
173 struct color_type
174 {
175     using value_type = signed short;
176
177     color_type(const value_type r, const value_type g, const value_type b)
178         : r(r), g(g), b(b)
179     {
180     }
181
182     ~color_type()
183     {
184     }
185
186     const value_type R() const { return r; }
187
188     const value_type G() const { return g; }
189
190     const value_type B() const { return b; }
191
192     color_type() = delete;
193     color_type(const color_type&) = delete;
194     color_type& operator=(const color_type&) = delete;
195
196 private:
197     const value_type r, g, b;
198 };
```

```
199
200 void Feature3()
201 {
202     random_access_container_wrapper<std::deque<int>> xs;
203     xs[0] = 1;
204
205     random_access_container_wrapper<std::vector<int>> ys("cannot be resized", 3);
206     ys[0] = 1; ys[1] = 2; ys[2] = 3;
207
208     random_access_container_wrapper<std::vector<double>> zs{ 10, 20, 30 };
209
210     /* random_access_container_wrapper<std::deque<int>> ws(xs); */
211
212     racw_with_vector<char> vs{ 'a', 'b', 'c' };
213
214     const color_pod amber{ 255, 126, 0 };
215     const color_pod eletric_lime{ 204, 255, 0 };
216     const color_type emerald{ 80, 200, 120 };
217     const color_type golden{ 255, 215, 0 };
218 }
219
220 ////////////////////////
221 //1. Non-Static Data Member Initializers
222 struct color
223 {
224     using value_type = signed short;
225
226     const static value_type MIN = 0;
227     const static value_type MAX = 255;
228
229     value_type r = 255;
230     value_type g = 255;
231     value_type b = 255;
232     std::string name = "white";
233
234     color(std::string name, value_type r, value_type g, value_type b) :
235         name(name), r(r), g(g), b(b)
236     {
237     }
238
239     color() = default;
240     ~color() = default;
241     color(const color&) = default;
242     color& operator=(const color&) = delete;
243 };
244
245 const color make_COLOR(const char* name,
246                        const color::value_type r = color::MIN,
247                        const color::value_type g = color::MIN,
248                        const color::value_type b = color::MIN)
249 {
250     return color(name, r, g, b);
251 }
252
253 const color make_WHITE() { return color{}; }
254
255 const color make_BLACK() { return make_COLOR("black"); }
256
257 void Feature4()
258 {
259     //except if there's no default member values
260     //const color RED   { 255, 0, 0 };
261     //const color GREEN { 0, 255, 0 };
262     //const color BLUE  { 0, 0, 255 };
263
264     const auto white = make_WHITE();
```

```
265     const auto black = make_BLACK();
266     const auto red = make_COLOR("red", white.r | black.r);
267 }
268
269 ////////////////////////
270 //1. Eliminating undesirable conversion
271 double double_exchange(double& x, double newValue)
272 {
273     auto oldValue = x;
274     x = newValue;
275     return oldValue;
276 }
277
278 double double_exchange(double& x, float newValue) = delete;
279
280 void Feature5()
281 {
282     double d = 100.0;
283     double old_d = double_exchange(d, 200.0);
284     //old_d = double_exchange(d, 300.0f);
285 }
286
287 ////////////////////////
288 //1. Variadic Templates
289 template <typename T, typename... Ts> struct List
290 {
291     T head;
292     List<Ts...> tail;
293 };
294
295 template<typename T> struct List<T>
296 {
297     T head;
298 };
299
300 void println_all() { std::cout << std::endl; }
301
302 template<typename Arg, typename... Args>
303 void println_all(Arg a, Args... args)
304 {
305     unsigned int size = sizeof...(args);
306     if (size >= 0)
307     {
308         std::cout << a << " ";
309         println_all(args...);
310     }
311 }
312
313 template<typename... Items>
314 void println_all_uniform(Items... items)
315 {
316     //Items must be uniform (same type for all args)
317     auto xs = { items... };
318     for (auto& x : xs)
319         std::cout << x << " ";
320     std::cout << std::endl;
321 }
322
323 void Feature6()
324 {
325     List<int, int, int> xs;
326     xs.head = 1;
327     xs.tail.head = 2;
328     xs.tail.tail.head = 3;
329     //xs.tail.tail.tail ... invalid
330
```

```
331     List<int, int, int, int> ys {1};
332     List<int, int, int> ys1 = ys.tail;
333     List<int, int> ys2 = ys1.tail;
334     List<int> ys3 = ys2.tail;
335 
336     println_all(1, 2.0, "3", '4');
337 
338     println_all_uniform(1.f, 2.f, 3.f, 4.f);
339 }
340 
341 ////////////////////////
342 //1. Strongly Typed Enums
343 //2. Lambdas
344 using ushort = unsigned short;
345 
346 enum class Suit : ushort
347 {
348     Diamond, Club, Heart, Spade
349 };
350 
351 enum class Rank : ushort
352 {
353     Ace = 1,
354     Two, Three, Four,
355     Five, Six, Seven,
356     Eight, Nine, Ten,
357     Jack,Queen, King
358 };
359 
360 const char* to_str(const Rank rank)
361 {
362     switch (rank)
363     {
364     case Rank::Ace: return "A";
365     case Rank::Two: return "2";
366     case Rank::Three: return "3";
367     case Rank::Four: return "4";
368     case Rank::Five: return "5";
369     case Rank::Six: return "6";
370     case Rank::Seven: return "7";
371     case Rank::Eight: return "8";
372     case Rank::Nine: return "9";
373     case Rank::Ten: return "10";
374     case Rank::Jack: return "J";
375     case Rank::Queen: return "Q";
376     case Rank::King: return "K";
377     default: return "";
378     }
379 }
380 
381 const char* to_str(const Suit suit)
382 {
383     switch (suit)
384     {
385     case Suit::Diamond: return "D";
386     case Suit::Club: return "C";
387     case Suit::Heart: return "H";
388     case Suit::Spade: return "S";
389     default: return "";
390     }
391 }
392 
393 struct Card
394 {
395     Card(Rank r, Suit s) :
396         R_(r), S_(s)
```

```
397     {
398     }
399
400     Card(const Card& that) :
401         R_(that.R_), S_(that.S_)
402     {
403     }
404
405     const Card& operator=(const Card& that)
406     {
407         if (this != &that)
408         {
409             R_ = that.R_;
410             S_ = that.S_;
411         }
412
413         return *this;
414     }
415
416     const Suit get_suit() const { return S_; }
417
418     const Rank get_rank() const { return R_; }
419
420     friend static std::ostream& operator<<(std::ostream& out, const Card& c)
421     {
422         out << to_str(c.R_) << "(" << to_str(c.S_) << ")";
423         return out;
424     }
425
426     template<typename Integral>
427     static Card make_Card(Integral rank, Integral suit)
428     {
429         static_assert(std::is_integral<Integral>::value,
430             "Integral must be an integral type");
431         static_assert(std::is_convertible<Integral, ushort>::value,
432             "Integral must be an convertible type to unsigned short");
433         //assert(0U <= rank  && rank <= 3U);
434         //assert(1U <= suit  && suit <= 13U);
435         //or
436         assert(static_cast<ushort>(Rank::Ace) <= rank  && rank <= static_cast<ushort>(Rank::King));
437         assert(static_cast<ushort>(Suit::Diamond) <= suit  && suit <= static_cast<ushort>(Suit::Spade)) ↙
    ;
438         return Card(static_cast<Rank>(rank), static_cast<Suit>(suit));
439     }
440
441 private:
442     Suit S_;
443     Rank R_;
444 };
445
446 typedef std::vector<Card> deck_type;
447
448 deck_type make_Standard_52()
449 {
450     deck_type temp;
451
452     for (ushort r = static_cast<ushort>(Rank::Ace); r <= static_cast<ushort>(Rank::King); ++r)
453         for (ushort s = static_cast<ushort>(Suit::Diamond); s <= static_cast<ushort>(Suit::Spade); ++s)
454             temp.push_back(Card::make_Card(r, s));
455
456     return temp;
457 }
458
459 void cout_deck(const deck_type& d)
460 {
461     for (auto c : d)
```

```
462         std::cout << c << " ";
463     std::cout << std::endl;
464 }
465
466 void Feature7()
467 {
468     auto deck_52 = make_Standard_52();
469
470     auto by_Suit = [](const Card& lhs, const Card& rhs){
471         return lhs.get_suit() < rhs.get_suit();
472     };
473
474     auto by_Rank = [](const Card& lhs, const Card& rhs){
475         return lhs.get_rank() < rhs.get_rank();
476     };
477
478     cout_deck(deck_52);
479     //then
480     //std::sort(deck_52.begin(), deck_52.end(), by_Suit);
481     //std::stable_sort(deck_52.begin(), deck_52.end(), by_Rank);
482     //then
483     std::stable_sort(deck_52.begin(), deck_52.end(), by_Suit);
484     cout_deck(deck_52);
485     //then
486     std::random_shuffle(deck_52.begin(), deck_52.end());
487     cout_deck(deck_52);
488 }
489
490 ////////////////////////
491 //1. chrono
492 void Feature8()
493 {
494     auto now = std::chrono::system_clock::now();
495     auto now_time = std::chrono::system_clock::to_time_t(now);
496     std::cout << std::ctime(&now_time);
497
498     using _clock = std::chrono::system_clock;
499     std::chrono::time_point<_clock> start, end;
500     start = _clock::now();
501     std::this_thread::sleep_for(std::chrono::seconds(3));
502     end = _clock::now();
503     std::chrono::duration<double> elapsed_seconds = end - start;
504     std::time_t delta = _clock::to_time_t(end);
505     std::cout << std::ctime(&delta);
506 }
507
508 ////////////////////////
509 //1. random
510 //2. static_cast
511 //3. array
512 //4. unordered_map
513 //5. unique_ptr
514 template<typename Integral>
515 struct random_functor
516 {
517     static_assert(std::is_integral<Integral>::value, "Integral must be an integral type");
518     static_assert(std::is_signed<Integral>::value, "Integral must be an signed integral type");
519
520     using RndEngine = std::default_random_engine;
521     using RndDistr = std::uniform_int_distribution<Integral>;
522
523     random_functor(Integral min_inclusive, Integral max_inclusive,
524         unsigned int seed = static_cast<unsigned>(std::time(nullptr))) :
525         Engine_(new RndEngine(seed)),
526         Rnd_(new RndDistr(min_inclusive, max_inclusive))
527     {
```

```
528     }
529
530     random_functor(Integral max_exclusive,
531         unsigned int seed = static_cast<unsigned>(std::time(nullptr))) :
532         Engine_(new std::RndEngine(seed)),
533         Rnd_(new RndDistr(0, max_exclusive - 1))
534     {
535     }
536
537     //[min; max] or [0; max)
538     auto operator()() const -> Integral { return (*Rnd_)(*Engine_); }
539
540     random_functor(const random_functor&) = delete;
541     random_functor& operator=(const random_functor&) = delete;
542
543 private:
544     std::unique_ptr<RndEngine> Engine_;
545     std::unique_ptr<RndDistr> Rnd_;
546 };
547
548 void fill_random(int min_inclusive, int max_inclusive, unsigned int size, std::unordered_map<int, int>& ↙
      xs)
549 {
550     random_functor<int> rnd(min_inclusive, max_inclusive);
551
552     for (unsigned int i = 0; i < size; ++i)
553     {
554         int x = rnd();
555         auto iter = xs.find(x);
556         if (iter != xs.end())
557             iter->second++;
558         else
559             xs.emplace(x, 1);
560     }
561 }
562
563 void fill_random(int min_inclusive, int max_inclusive, unsigned int size, std::vector<int>& xs)
564 {
565     random_functor<int> rnd(min_inclusive, max_inclusive);
566
567     for (unsigned int i = 0; i < size; ++i)
568         xs.push_back(rnd());
569 }
570
571 void Feature9()
572 {
573     unsigned int seed = static_cast<unsigned int>(std::chrono::system_clock::now().time_since_epoch(). ↙
      count());
574     std::default_random_engine generator(seed);
575     std::normal_distribution<double> distribution(0.0 /* mean */, 1.0 /* stddev */);
576
577     std::array<double, 5> arr;
578     for (int i = 0; i < 5; ++i)
579         arr[i] = distribution(generator);
580
581     std::array<std::array<double, 10>, 4> mat;
582     unsigned int i = 0;
583     for (auto& x : { std::make_tuple(0.0, 0.4), std::make_tuple(0.0, 1.0),
584         std::make_tuple(0.0, 2.2), std::make_tuple(-2.0, 0.7) })
585     {
586         std::normal_distribution<double> distr(std::get<0>(x) /* mean */, std::get<1>(x) /* stddev */);
587         for (unsigned int j = 0; j < mat[i].size(); ++j)
588             mat[i][j] = distr(generator);
589         ++i;
590     }
591
```

```
592     std::unordered_map<int, int> xs;
593     fill_random(1, 5, 1000, xs);
594 }
595 
596 ////////////////////////
597 //1. thread
598 //2. atomic
599 struct stop_watch
600 {
601     stop_watch() :
602         Start_(now()) {}
603 
604     std::chrono::milliseconds elapsed_ms() const
605     {
606         return std::chrono::duration_cast<std::chrono::milliseconds>(now() - Start_);
607     }
608 
609     std::chrono::nanoseconds elapsed_ns() const
610     {
611         return std::chrono::duration_cast<std::chrono::nanoseconds>(now() - Start_);
612     }
613 
614     void restart() { Start_ = now(); }
615 
616 private:
617     static std::chrono::high_resolution_clock::time_point now()
618     {
619         return std::chrono::high_resolution_clock::now();
620     }
621 
622     std::chrono::high_resolution_clock::time_point Start_;
623 };
624 
625 void Feature10()
626 {
627     std::vector<int> xs;
628     fill_random(1, 5, 100000, xs);
629 
630     std::atomic<int> count;
631 
632     auto find_elem = [&count](std::vector<int>::const_iterator start,
633         std::vector<int>::const_iterator finish, int elem){
634 
635         std::stringstream ss;
636         ss << "finding in thread #" << std::this_thread::get_id() << std::endl;
637 
638         for (auto i = start; i != finish; ++i)
639             if (*i == elem) ++count;
640 
641         std::cout << ss.str();
642     };
643 
644     stop_watch sw;
645     std::thread t(find_elem, xs.cbegin(), xs.cend(), 1);
646     t.join();
647     auto elapsed_ms = sw.elapsed_ms().count();
648 
649     std::cout << "found " << count << " occurence(s) from " << "1" << std::endl;
650     std::cout << elapsed_ms << " ms" << std::endl;
651 
652     sw.restart();
653     int find_this_elem = 2;
654     count = 0;
655 
656     std::thread t1(find_elem, xs.cbegin(), xs.cbegin() + xs.size() / 2, find_this_elem);
657     find_elem(xs.cbegin() + xs.size() / 2, xs.cend(), find_this_elem);
```

```
658     t1.join();
659
660     elapsed_ms = sw.elapsed_ms().count();
661
662     std::cout << "found " << count << " occurence(s) from " << find_this_elem << std::endl;
663     std::cout << elapsed_ms << " ms" << std::endl;
664 }
665
666 ////////////////////////
667 //1. async
668 //2. future
669 //3. promise
670 void Feature11()
671 {
672     std::promise<int> prom;
673     auto prom_fut = prom.get_future();
674
675     std::thread t([](std::future<int>& f) {
676         std::cout << "waiting for future value..." << std::endl;
677         int x = f.get();
678         std::cout << "value has arrived = " << x << std::endl;
679     }, std::ref(prom_fut));
680
681     std::this_thread::sleep_for(std::chrono::seconds(4));
682
683     prom.set_value(100);
684
685     t.join();
686
687
688     std::vector<std::future<std::tuple<unsigned int, short, int>>> results;
689
690     const unsigned int NUMBER_OF_TASKS = 4;
691     std::cout << "waiting for results..." << std::endl;
692     for (unsigned int i = 0; i < NUMBER_OF_TASKS; ++i)
693     {
694         //async
695         results.push_back(std::async([i]
696         {
697             unsigned int seed = (i + 10) * static_cast<unsigned int>(std::time(nullptr));
698             random_functor<short> rnd(1, std::numeric_limits<short>::max(), seed);
699
700             random_functor<int> rnd_sleep(1, 4, seed);
701             int sleep = rnd_sleep();
702             std::this_thread::sleep_for(std::chrono::seconds(sleep));
703             unsigned int thread_id = std::this_thread::get_id().hash();
704
705             return std::make_tuple(thread_id, rnd(), sleep);
706         }));
707     }
708
709     for (auto& result : results)
710     {
711         auto r = result.get();
712         std::cout << "thread #" << std::get<0>(r)
713             << " result = " << std::get<1>(r)
714             << " delay = " << std::get<2>(r) << " s" << std::endl;
715     }
716 }
717
718 int main()
719 {
720     Feature1();
721     //Feature2();
722     //Feature3();
723     //Feature4();
```

```
724     //Feature5();
725     //Feature6();
726     //Feature7();
727     //Feature8();
728     //Feature9();
729     //Feature10();
730     //Feature11();
731 }
```