Lighthouse Financial Technology *2019 Feb 21*

Question 1

Given a type, return true if it is a pointer, false otherwise. All *3* methods below are verified with *Visual Studio*.

```
// Method 1 : LHFT’s asnwer, but I add static, or we can’t access traits without instantiating traits obj.
template <typename T> struct is_pointer     { static const bool value = false; };
template <typename T> struct is_pointer<T*> { static const bool value = true;  };
std::cout << (is_pointer<std::string >::value? “a ptr” : ”not a ptr”);
std::cout << (is_pointer<std::string*>::value? “a ptr” : ”not a ptr”);

// Method 2
template <typename T> struct is_pointer     { enum { value = false }; };
template <typename T> struct is_pointer<T*> { enum { value = true  }; };
std::cout << (is_pointer<std::string >::value? “a ptr” : ”not a ptr”);
std::cout << (is_pointer<std::string*>::value? “a ptr” : ”not a ptr”);

// Method 3
template <typename T> struct is_pointer     { typedef std::false_type value; };
template <typename T> struct is_pointer<T*> { typedef std:: true_type value; };
std::cout << (std::is_same<is_pointer<std::string >::value, std::true_type>::value? “a ptr” : ”not a ptr”);
std::cout << (std::is_same<is_pointer<std::string*>::value, std::true_type>::value? “a ptr” : ”not a ptr”);
```

Basically the idea is template specialization. In fact, template class `std::is_pointer<>` is offered by STL.

Question 2

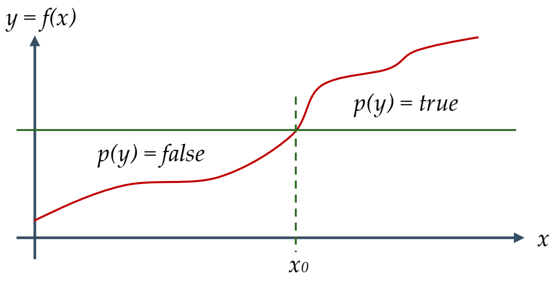
Given a vector of sorted integers, and a target integer value, return the index of the element that equals to the target, and *-1* if no entry can be matched. Make the implementation as simple as possible.

```
int search(const std::vector<int>& vec, int target)
{
      unsigned long begin = 0;                // inclusive
      unsigned long end = vec.size();         // exclusive

      while(begin < end) // This loop is simple, as it can handle two special cases automatically.
      {
            // Case 1 : when begin = end-1, then middle = begin
            // Case 2 : when begin = end-2, then middle = begin+1
            unsigned long middle = (begin + end)/2;

            if      (vec[middle] == target)   return  middle;
            else if (vec[middle] <  target)   begin = middle+1;
            else                              end = middle;
      }
      return -1;  // quit when no element left
}
```

Why is it so hard to get this implementation everytime I come across this question? This is because I always tend to think in a recursive way, which is definitely not necessary. Beside it is too ambitious to store only a pointer to the middle (while omitting begin and end), if we have to omit begin and end pointers, then the function should read both pointers as input, rather than the vector itself. Finally above implementation works when vector size is *1* or *2*, thus no special case needed.

Question 3

Given a monotonic increasing function $f : \Re \to \Re$ and a predicate function $p : \Re \to \{true, false\}$, find the minimum value x_0 such that $p(f(x_0))$ is true if $p(y) = true$ when y is greater than some threshold and $x \in \{0, 10^9\}$.



The answer is also bisection. How many iterations does it take (suppose we want to achieve ±0.5 accuracy)?

$$\#iteration = \log_2 10^9 = \log_2 1G = \log_2 K^3 = \log_2 2^{30} = 30$$

```
bool within_tolerance(double x, double y) { return fabs(x-y) < 0.5; }

double search(double (*f)(double)) // I try to use exactly the same style as previous question.
{
      double xmin = 0;
      double xmax = 1e9;

      while(within_tolerance(xmin, xmax))
      {
            double x = (xmin + xmax)/2;
            double y0 = f(x-0.5);
            double y1 = f(x+0.5);

            if      (!p(y0) &&  p(y1))   return  middle;
            else if (!p(y0) && !p(y1))   xmin = x;
            else if ( p(y0) &&  p(y1))   xmax = x;
            else throw std::exception("The functor is not monotonic increasing.");
      }
      throw std::exception("The root is not within x range [0,1e9].")
}
```

Do gradient search like Newton method, secant method or Brent method offer faster convergence?

Question 4

What is implementation of `std::map`? It is a binary search tree.

What are the differences between binary tree and binary search tree? The latter is a subset of the former, the former refers to tree having at most two children per node, whereas the later is sorted, so that keys in the LHS subtree are smaller than the key of current node and keys in the RHS subtree are greater than that of current node.

Why should binary search tree be self balancing? Height of tree can be minimised with self balancing, search time can be minimised and approximated as $log_2 N$, where N is the number of nodes in the tree.

Question 5

What is the problem with the following code (it can be compiled)? Given that `delete` is called appropriately inside `f`.

```
f(new X, new Y); // f(X* x, Y* y) { ... delete x; delete y; }
```

It is OK in most cases, however there is memory leakage when construction of `Y` throws, then `X` is leaked.

Question 6

- What is cache friendly programming?
- What is memory barrier?
- What is virtual memory addressing and page fault?

Page fault is an exception raised when a process accesses a memory page, which is not mapped by memory management unit (MMU) into the virtual address space of the process.

Question 7

How does backward propagation work in neural network? Decomposition of derivatives into partial derivatives …

Question 10 : Least recently used *(LRU)* map

Least recently used map is a map which applies least recently used replacement algorithm to ensure a limited maximum number of entries in the map, in short *LRU* is an algorithm that erase least recently used entries. For example, consider a map with maximum size *3*, it has two functions `void put(const K& k, const V& v)` and `const V& get(const K& k)`.

```
map.put(A,v1);     // (empty)         push back   resulting state (A,v1)
map.put(B,v2);     // (not full)      push back   resulting state (A,v1), (B,v2)
map.put(C,v3);     // (not full)      push back   resulting state (A,v1), (B,v2), (C,v3)
map.put(D,v4);     // pop front       push back   resulting state (B,v2), (C,v3), (D,v4)
map.get(C);        // pop existing    push back   resulting state (B,v2), (D,v4), (C,v3)
map.put(E,v5);     // pop front       push back   resulting state (D,v4), (C,v3), (E,v5)
map.put(D,v6);     // pop front       push back   resulting state (C,v3), (E,v5), (D,v6)
map.put(A,v7);     // pop front       push back   resulting state (E,v5), (D,v6), (A,v7)
map.put(D,v8);     // pop existing    push back   resulting state (E,v5), (A,v7), (D,v8)
map.get(B);        // get fail                    resulting state (E,v5), (A,v7), (D,v8)
map.get(E);        // pop existing    push back   resulting state (A,v7), (D,v8), (E,v5)
map.put(E,v9);     // pop existing    push back   resulting state (A,v7), (D,v8), (E,v9)
```

How should we implement the *LRU* algorithm? The following implementation is tested in online C++ compiler.

```
template<typename K, int N> struct lru_list
{
    auto add_entry(const K& key, std::optional<K>& del_entry)
    {
        list.push_front(key);
        if (list.size() > N)
        {
            del_entry = std::make_optional<K>(list.back());
            list.pop_back();
        }
        return list.begin();
    }

    auto use_entry(typename std::list<K>::iterator iter)
    {
        list.push_front(*iter);
        list.erase(iter);
        return list.begin();
    }

    std::list<K> list;
};

template<typename K, typename V, int N> struct lru_map
{
    struct cell
    {
        V value;
        typename std::list<K>::iterator iter;
    };

    void set(const K& key, const V& value)
    {
        auto iter = map.find(key);
        if (iter == map.end())
        {
            std::optional<K> del_entry;
            map[key] = cell{value, recently_used.add_entry(key, del_entry)};
            if (del_entry) map.erase(*del_entry);
        }
        else
        {
            iter->second.value = value;
            iter->second.iter  = recently_used.use_entry(iter->second.iter);
        }
    }

    std::optional<V> get(const K& key)
    {
        auto iter = map.find(key);
        if (iter == map.end())
        {
            return std::nullopt;
        }
        else
        {
            auto output = std::make_optional<V>(iter->second.value);
            iter->second.iter = recently_used.use_entry(iter->second.iter);
            return output;
        }
    }

    std::map<K, cell> map;
    lru_list<K, N> recently_used;
};
```

Basic ideas include :

- `lru_map` stores key-value pair and reference to `lru_list` via `list<K>::iterator`
- `lru_list` stores recently used sequence and reference to `lru_map` via `K`
- `lru_map` offers `set` and `get` functions
- `lru_list` offers `add_entry` and `use_entry` functions

	input	output
`lru_list<>::add_entry`	newly-added-key	updated `list<K>::iterator` and to-be-deleted key
`lru_list<>::use_entry`	`list<K>::iterator`	updated `list<K>::iterator`

Question 11

What is `std::optional<T>` in the above implementation? This is a C++17 helper template.

Question 12

What is move semantics? What are `std::move` and `std::forward`?

Question 13

In *TDMS pricing manager*, how is threadpool implemented? As there is no thread pool in STL, can we use `promise`, `future` and `async_task` instead?

Question 14

In *TraderRun disruptor pattern*, there are lockfree ring buffer, what are the synchronization mechanisms between producer and consumer? My answer is that : producer's next-write and consumer's next-read are both atomic variables, no explicit synchronization is needed, memory barrier is used, while consumer thread keeps polling to check for unprocessed ticks. Yet the interviewer commented that this is totally out of the question, he was expecting answers like conditional variable, he did also asked what other classes does conditional variable have to work with? And what happens inside the waiting function on conditional variable? Conditional variable has to work with mutex, when consumer tries to consume, finding that there is nothing to consume, it then waits on conditional variable and passing a mutex to producer, so that producer can enter the critical session and produces.

Interviewer did also challenge my understanding about `atomic` by dropping me questions like : why `atomic` is needed for next-read or next-write indices in lockfree buffer as integer is already atomic? What happens inside the call `fetch_and_add` on an `atomic` variable? Name all multithreading synchronization mechanisms. Here are the answers from wiki : spinlock, mutex, semaphore, reader writer lock, conditional variable and memory barrier.

Question 15 : Design a market data broadcast module

Design a datafeed broadcasting component for multiple securities and multiple trading algos in multiple machines, there are 5000 different securities 10 algos, each algo observes several securities, one machine may run more than one algos, do not transfer duplicated data to the same machine (2 algos on the same machine observing overlapping sets of securities).

Do we really need 5000 ring buffers?

No, with inheritance, we can merge them into one. Like the one in `YLib`.