7.x — Chapter 7 comprehensive quiz

BY ALEX ON DECEMBER 4TH, 2015 | LAST MODIFIED BY NASCARDRIVER ON JANUARY 26TH, 2020

Quick Summary

Another chapter down! The next chapter is the best one, and you're almost there! There's just this pesky quiz to get past...

Function arguments can be passed by value, reference or address. Use pass by value for fundamental data types and enumerators. Use pass by reference for structs, classes, or when you need the function to modify an argument. Use pass by address for passing pointers or built-in arrays. Make your pass by reference and address parameters const whenever possible.

Values can be returned by value, reference, or address. Most of the time, return by value is fine, however return by reference or address can be useful when working with dynamically allocated data, structs, or classes. If returning by reference or address, remember to make sure you're not returning something that will go out of scope.

Inline functions allow you to request that the compiler replace your function call with the function code. You should not need to use the inline keyword because the compiler will generally determine this for you.

Function overloading allows us to create multiple functions with the same name, so long as each function is distinct in the number or types of parameters. The return value is not considered when determining whether an overload is distinct.

A default argument is a default value provided for a function parameter. If the caller doesn't explicitly pass in an argument for a parameter with a default value, the default value will be used. You can have multiple parameters with default values. All parameters with default values must be to the right of non-default parameters. A parameter can only be defaulted in one location. Generally it is better to do this in the forward declaration. If there are no forward declarations, this can be done on the function definition.

Function pointers allow us to pass a function to another function. This can be useful to allow the caller to customize the behavior of a function, such as the way a list gets sorted.

Dynamic memory is allocated on the heap.

The call stack keeps track of all of the active functions (those that have been called but have not yet terminated) from the start of the program to the current point of execution. Local variables are allocated on the stack. The stack has a limited size. std::vector can be used to implement stack-like behavior.

A recursive function is a function that calls itself. All recursive functions need a termination condition.

A syntax error occurs when you write a statement that is not valid according to the grammar of the C++ language. The compiler will catch these. A semantic error occurs when a statement is syntactically valid, but does not do what the programmer intended. Two common semantic errors are logic errors, and violated assumptions. The assert statement can be used to detect violated assumptions, but has the downside of terminating your program immediately if the assertion statement is false.

Command line arguments allow users or other programs to pass data into our program at startup. Command line arguments are always C-style strings, and have to be converted to numbers if numeric values are desired.

Ellipsis allow you to pass a variable number of arguments to a function. However, ellipsis arguments suspend type checking, and do not know how many arguments were passed. It is up to the program to keep track of these details.

Lambda functions are functions that can be nested inside other functions. They don't need a name and are very useful in combination with the algorithms library.

Quiz time

Question #1

Write function prototypes for the following cases. Use const if/when necessary.

a) A function named max() that takes two doubles and returns the larger of the two.

Show Solution

b) A function named swap() that swaps two integers.

Show Solution

c) A function named getLargestElement() that takes a dynamically allocated array of integers and returns the largest number in such a way that the caller can change the value of the element returned (don't forget the length parameter).

Show Solution

Question #2

What's wrong with these programs?

a)

```
int& doSomething()
{
    int array[]{ 1, 2, 3, 4, 5 };
    return array[3];
}
```

Show Solution

b)

```
int sumTo(int value)
{
    return value + sumTo(value - 1);
}
```

Show Solution

c)

```
float divide(float x, float y)
{
    return x / y;
}

double divide(float x, float y)
{
    return x / y;
}
```

Show Solution

d)

```
#include <iostream>

int main()
{
    int array[100000000]{};

    for (auto x: array)
        std::cout << x << ' ';

    std::cout << '\n';

    return 0;
}
```

Show Solution

e)

```
#include <iostream>

int main(int argc, char *argv[])
{
    int age{ argv[1] };
    std::cout << "The user's age is " << age << '\n';

    return 0;
}
```

Show Solution

Question #3

The best algorithm for determining whether a value exists in a sorted array is called binary search.

Binary search works as follows:

- Look at the center element of the array (if the array has an even number of elements, round down).
- If the center element is greater than the target element, discard the top half of the array (or recurse on the bottom half)
- If the center element is less than the target element, discard the bottom half of the array (or recurse on the top half).
- If the center element equals the target element, return the index of the center element.
- If you discard the entire array without finding the target element, return a sentinel that represents "not found" (in this case, we'll use -1, since it's an invalid array index).

Because we can throw out half of the array with each iteration, this algorithm is very fast. Even with an array of a million elements, it only takes at most 20 iterations to determine whether a value exists in the array or not! However, it only works on sorted arrays.

Modifying an array (e.g. discarding half the elements in an array) is expensive, so typically we do not modify the array. Instead, we use two integer (min and max) to hold the indices of the minimum and maximum elements of the array that we're interested in examining.

Let's look at a sample of how this algorithm works, given an array { 3, 6, 7, 9, 12, 15, 18, 21, 24 }, and a target value of 7. At first, min = 0, max = 8, because we're searching the whole array (the array is length 9, so the index of the

last element is 8).

- Pass 1) We calculate the midpoint of min (0) and max (8), which is 4. Element #4 has value 12, which is larger than our target value. Because the array is sorted, we know that all elements with index equal to or greater than the midpoint (4) must be too large. So we leave min alone, and set max to 3.
- Pass 2) We calculate the midpoint of min (0) and max (3), which is 1. Element #1 has value 6, which is smaller than our target value. Because the array is sorted, we know that all elements with index equal to or lesser than the midpoint (1) must be too small. So we set min to 2, and leave max alone.
- Pass 3) We calculate the midpoint of min (2) and max (3), which is 2. Element #2 has value 7, which is our target value. So we return 2.

Given the following code:

```
#include <iostream>

// array is the array to search over.
// target is the value we're trying to determine exists or not.
// min is the index of the lower bounds of the array we're searching.
// max is the index of the upper bounds of the array we're searching.
// binarySearch() should return the index of the target element if the target is found, -1 oth
int binarySearch(const int *array, int target, int min, int max)
{

}

int main()
{
    constexpr int array[]{ 3, 6, 8, 12, 14, 17, 20, 21, 26, 32, 36, 37, 42, 44, 48 };

    // We're going to test a bunch of values to see if they produce the expected results
    constexpr int numTestValues{ 9 };
    // Here are the test values
    constexpr int testValues[numTestValues]{ 0, 3, 12, 13, 22, 26, 43, 44, 49 };
    // And here are the expected results for each value
    int expectedValues[numTestValues]{ -1, 0, 3, -1, -1, 8, -1, 13, -1 };

    // Loop through all of the test values
    for (int count{ 0 }; count < numTestValues; ++count)
    {
        // See if our test value is in the array
        int index{ binarySearch(array, testValues[count], 0, 14) };
        // If it matches our expected value, then great!
        if (index == expectedValues[count])
            std::cout << "test value " << testValues[count] << " passed!\n";
        else // otherwise, our binarySearch() function must be broken
            std::cout << "test value " << testValues[count] << " failed.  There's something w
    }

    return 0;
}
```

a) Write an iterative version of the binarySearch function.

Hint: You can safely say the target element doesn't exist when the min index is greater than the max index.

Show Solution

b) Write a recursive version of the binarySearch function.

Show Solution

Tip

`std::binary_search` return true if a value exists in a sorted list.
`std::equal_range` returns the iterators to the first and last element with a given value.

Don't use these functions to solve the quiz, but use them in the future if you need a binary search.


8.1 -- Welcome to object-oriented programming


Index


7.16 -- Lambda captures

C++ TUTORIAL | PRINT THIS POST

185 comments to 7.x — Chapter 7 comprehensive quiz

« Older Comments 1 2 3

Suyash
January 25, 2020 at 3:42 pm · Reply

Here's my implementation of one of the simplest but in some ways, the most useful algorithm, Binary Search... Reminded me of my first algorithms class... It is one of the most important algorithms that I believe, every programmer should know... Thanks for including it on the quiz...

a) Iterative Version

```
// array is the array to search over.
// target is the value we're trying to determine exists or not.
// min is the index of the lower bounds of the array we're searching.
// max is the index of the upper bounds of the array we're searching.
// binarySearch() should return the index of the target element if the target is found, -1
int binarySearch(const int* array, int target, int min, int max)
{
    assert(array && "Value Error: Array does not exist.");

    constexpr int not_found{ -1 };
    constexpr int no_of_terms{ 2 };

    while (min <= max)
    {
        const int mid{ (max + min) / no_of_terms };

        // When `target` is found
        if (target == array[mid])
            return mid;
        // When `target` is less than the value found at `mid` of `array`
        else if (target < array[mid])
            max = mid - 1;
        // When `target` is greater than the value found at `mid` of `array`
        else
            min = mid + 1;
    }

    // `target` is not found
    return not_found;
}
```

b) Recursive Version

```
// array is the array to search over.
// target is the value we're trying to determine exists or not.
// min is the index of the lower bounds of the array we're searching.
// max is the index of the upper bounds of the array we're searching.
// binarySearchRecursion() should return the index of the target element if the target is f
int binarySearchRecursion(const int* array, int target, int min, int max)
{
    assert(array && "Value Error: Array does not exist.");

    constexpr int not_found{ -1 };
    constexpr int no_of_terms{ 2 };

    // Base Case 1 (When `target` is not found)
    if (min > max)
        return not_found;

    const int mid{ (max + min) / no_of_terms };

    // Base Case 2 (When `target` is found)
    if (target == array[mid])
        return mid;
    // When `target` is less than the value found at `mid` of `array`
    else if (target < array[mid])
        max = mid - 1;
    // When `target` is greater than the value found at `mid` of `array`
    else
        min = mid + 1;

    // Recursive Case
```

```
30      binarySearchRecursion(array, target, min, max);
31  }
```

One question, I saw how you calculated midpoint... In what scenario would the conventional method of finding midpoint ((a + b) / 2) would lead to an overflow?

Referring to this line in your code:

```
1  int midpoint{ min + ((max-min) / 2) }; // this way of calculating midpoint avoids overflow
```

Will wait for your reply...

nascardriver
January 26, 2020 at 1:55 am · Reply

Hey!

Your algorithms look good!

Your midpoint calculation fails whenever (min + max) > INT_MAX.
Say the range of an `int` is [-1000, 1000] and `int` safely wraps around.

```
1  min = 600
2  max = 800
3
4  (max + min) / 2 = -600 / 2 = -300
5  min + ((max - min) / 2) = 600 + ((200) / 2) = 700
```

Suyash
January 26, 2020 at 2:41 am · Reply

Wow, thank you for introducing techniques that could help us prevent subtle bugs from occurring in our code...

This makes it this exercise even more valuable for everyone...

Unfortunately, this is my last chapter for the time being (due to me being busy as my current project is nearing deployment)... I will most likely come back by mid February...

Thank you to both you & @Alex for your wonderful contributions to this site... This site, in my humble opinion, is definitely one of the best places to learn modern C++ on the Internet...

P.S:And, who knows, if everything goes well, I'll be back sooner...

nascardriver
January 26, 2020 at 2:46 am · Reply

Good luck with the deployment! See you soon :)

Ged
December 13, 2019 at 6:31 am · Reply

```
1  assert(array); // make sure array exists
```

Could you give an example when it will work. Cause when I try to create an example, my compiler just throws out errors.

nascardriver
December 13, 2019 at 6:49 am · Reply

If your compiler is throwing errors, you're doing something else wrong.
`assert(array)` is successful whenever `array != nullptr`.
What error are you getting?

Ged
December 13, 2019 at 7:29 am · Reply

No I mean my program works, but I have no idea how to make my array nullptr in the example so the assert would work. Cause there is constexp. When i try to initialize it with nullptr, because it constexpr, it won't allow it, I am unable to modify it because of the const.

The only way it can work is when I do this

```
1 int* array = nullptr;
2 int binarySearch(int* array, int target, int min, int max)
```

But it has no const, so my question would be do we really need assert() when we are using const?

nascardriver
December 13, 2019 at 7:33 am · Reply

```
1 binarySearch(nullptr, 0, 0, 0);
```

You won't ever get a `nullptr` when you use static arrays, but this can happen when you dynamically allocate the array.

Wallace
December 6, 2019 at 11:40 am · Reply

Hi Alex and nascardriver,

For exercise 3a, I get an error when starting from that example code:

No matching function for call to 'binarySearch'
1. Candidate function not viable: no known conversion from 'const int [15]' to 'int *' for 1st argument

This is with Clang set to strict C++17 in Xcode. Adding const to the function declaration's array parameter like this solves it:

```
1 int binarySearch(const int *array, int target, int min, int max)
```

Is there any reason the lesson's starting code doesn't have that const?

Also, there's a minor typo early in the summary: "Most of time."

Thanks for this great resource!

nascardriver
December 7, 2019 at 3:28 am · Reply

`binarySearch`'s `array` should be `const`, thanks for pointing out the error! It worked without `const` before I made `main`'s `array` `constexpr`.

Type fixed, thanks!

hersel99
November 8, 2019 at 11:30 am · Reply

Hi! For quiz 3b I come with a (hopefully working) more simplified version:

```
int binarySearch(const int *array, int target, int min, int max)
{
    if (min <= max)
    {
        int midpoint { (max-min)/2 + min };

        if (array[midpoint] < target)
            min = midpoint+1;
        else if (array[midpoint] > target)
            max = midpoint-1;
        else
            return midpoint;
    }
    else
        return -1;

    return binarySearch(array, target, min, max);
}
```

Samira Ferdi
October 28, 2019 at 7:35 pm · Reply

Hi, Alex and Nascardriver! What do you think?

```
#include <iostream>
#include <vector>

using index_t = unsigned int;
index_t getMidIndex(const unsigned int minIndex, const unsigned int maxIndex)
{
    return ( maxIndex - minIndex ) / 2;
}

//needleIndex_t is integer instead of unsigned integer
//because find() would returned -1 if there is no needle in the haystack.
using needleIndex_t = int;
using haystack_t    = std::vector<int>;
needleIndex_t find(const int needle, const haystack_t &haystack)
{
    unsigned int minIndex{ 0 };
    unsigned int maxIndex{ haystack.size() - 1 };

    //Needle is out of the range of haystack
    if( needle < haystack[minIndex] || needle > haystack[maxIndex] )
        return -1;

   while( getMidIndex(minIndex, maxIndex) )
    {
        unsigned int midIndexPosition{
            getMidIndex(minIndex, maxIndex) + minIndex
        };

        if( needle > haystack[midIndexPosition] )
```

```
            minIndex = midIndexPosition;

        else if( needle < haystack[midIndexPosition] )
            maxIndex = midIndexPosition;

        else
            return static_cast<int>( midIndexPosition);
    }

    //If there is no middle index
    //(the haystack's element less than or equal to 2 elements)
    if(needle == haystack[minIndex])
        return static_cast<int>(minIndex);

    else if( needle == haystack[maxIndex])
        return static_cast<int>(maxIndex);

    //The needle is not found!
    return -1;
}

int main()
{
    std::vector<int> haystack{
        3, 6, 8, 12, 14, 17, 20, 21, 26, 32, 36, 37, 42, 44, 48
    };

    // We're going to test a bunch of values to see if they produce the expected results
    const int numTestValues = 9;
    // Here are the test values
    int testValues[numTestValues] = { 0, 3, 12, 13, 22, 26, 43, 44, 49 };
    // And here are the expected results for each value
    int expectedValues[numTestValues] = { -1, 0, 3, -1, -1, 8, -1, 13, -1 };

    // Loop through all of the test values
    for (int count=0; count < numTestValues; ++count)
    {
        // See if our test value is in the array
        int index = find(testValues[count], haystack);
        // If it matches our expected value, then great!
        if (index == expectedValues[count])
             std::cout << "test value " << testValues[count] << " passed!\n";
        else // otherwise, our binarySearch() function must be broken
             std::cout << "test value " << testValues[count] << " failed.  There's somethin
    }


    return 0;
}
```

nascardriver
October 29, 2019 at 2:38 am · Reply

Hi!

- Initialize your variables with brace initializers.
- Limit your lines to 80 characters in length for better readability.
- You have `index_t`, but you're not using it everywhere.
- Since you're using `index_t` to index the vector, `index_t` should be the same as the vector's index type. `using index_t = haystack_t::size_type;`.

- Inconsistent formatting. Use your editor's auto-formatting feature.
- The calls in line 23 and 26 produce the same result every time. You can use a `while (true)` loop and `break` or declare the midIndex outside of the loop.

Samira Ferdi
October 29, 2019 at 6:20 pm · Reply

Hi, Nascardriver! Thanks for your reply!

So, this is my change! (full code)

```
1  #include <iostream>
2  #include <vector>
3
4  //needleIndex_t is integer instead of unsigned integer
5  //because find() would returned -1 if there is no needle in the haystack.
6  using needleIndex_t = int;
7  using haystack_t = std::vector<int>;
8  needleIndex_t find(const int needle, const haystack_t &haystack)
9  {
10     using index_t = haystack_t::size_type;
11     index_t minIndex{ 0 };
12     index_t maxIndex{ haystack.size() - 1 };
13
14     //The needle is out of the range of haystack
15     if( needle < haystack[minIndex] || needle > haystack[maxIndex] )
16         //The needle is not found!
17         return -1;
18
19     while(minIndex <= maxIndex)
20     {
21         index_t midIndex{ minIndex + ( (maxIndex - minIndex) / 2 ) };
22
23         if( needle > haystack[midIndex] )
24             minIndex = midIndex + 1;
25
26         else if( needle < haystack[midIndex] )
27             maxIndex = midIndex - 1;
28
29         else
30             return static_cast<int>(midIndex);
31     }
32
33     //The needle is not found!
34     return -1;
35 }
36
37 void binarySearchTest(const haystack_t &haystack, const int *testValues,
38                       const int *expectedValuesIndex, const int numTestValues)
39 {
40     // Loop through all of the test values
41     for (int count{ 0 }; count < numTestValues; ++count)
42     {
43         // See if our test value is in the array
44         int index{ find(testValues[count], haystack) };
45
46         // If it matches our expected value, then great!
47         if ( index == expectedValuesIndex[count] )
48             std::cout << "test value " << testValues[count] << " passed!\n";
49
50         // otherwise, our binarySearch() function must be broken
```

```
51          else
52          {
53              std::cout << "test value " << testValues[count]
54                        << " failed.  There's something wrong with your code!\n";
55          }
56      }
57  }
58
59  int main()
60  {
61      std::vector<int> haystack
62      {
63          3, 6, 8, 12, 14, 17, 20, 21, 26, 32, 36, 37, 42, 44, 48
64      };
65
66      //===== TESTING CODES =====
67      // We're going to test a bunch of values to see if they produce the expected
68      const int numTestValues{ 9 };
69      // Here are the test values
70      int testValues[numTestValues] { 0, 3, 12, 13, 22, 26, 43, 44, 49 };
71      // And here are the expected results for each value
72      int expectedValuesIndex[numTestValues] { -1, 0, 3, -1, -1, 8, -1, 13, -1 };
73
74      binarySearchTest(haystack, testValues, expectedValuesIndex, numTestValues);
75
76      return 0;
77  }
```

nascardriver
October 30, 2019 at 3:36 am · Reply

Line 61: That's a `haystack_t`.
The rest looks good.
Line 12 is dangerous. If the haystack is empty, the integer underflows (because -1 isn't unsigned).

Samira Ferdi
October 30, 2019 at 6:19 pm · Reply

Hi, Nascardriver! Thanks for replying!

Yes, your right! Line 12 is dangerous if haystack is empty. So, I just added

```
1  assert(haystack.size() && "[ERROR] The haystack cannot be empty!");
```

at the top in find() and binarySearchTest. But is it necessary that I have to added assertion in main() right after haytack definition?

And I found that if find() work with vector, the while's loop expression should be:

```
1  while( static_cast<int>(minIndex) <= static_cast<int>(maxIndex) )
```

Deals with unsigned int is horrible pain! I understand now why other language throw out unsigned int. Do unsigned int has significant in C++, so C++ keeps using it?

nascardriver
October 31, 2019 at 12:47 am · Reply

> I just added
There's also `std::vector::empty`

```
assert(!haystack.empty() /* ... */)
```

> is it necessary that I have to added assertion in main
No. `main` has no problems if the vector is empty.

> the while's loop expression should be
Comparing unsigned to unsigned is fine.

> Do unsigned int has significant in C++
Sometimes you need that extra bit to save memory, eg. networking, or you want to represent a 32 bit rgba color, which uses up all bits but signedness doesn't make sense. And bitwise operations make more sense if all bits have the same meaning.

Samira Ferdi
October 31, 2019 at 5:13 pm · Reply

Hi, Nascardriver! Thanks a lot to reply my question!

If I remove this code

```
if( needle < haystack[minIndex] || needle > haystack[maxIndex] )
{
    //Needle is not found!
    return -1;
}
```

and while loop's expression is unsigned like this

```
while(minIndex <= maxIndex)
```

If I test this then the program is crash because maxIndex's value wrap around just to figure out 0 is exist or not!

nascardriver
November 1, 2019 at 12:37 am · Reply

> If I remove this code
Then `midIndex` can be 0 and line 27 wraps it. With the check, it should work, right?

hellmet
October 27, 2019 at 4:09 am · Reply

Thank you, Alex and Nascardriver, for the great detailed tutorials, amazing quizzes and patiently clarifying the many questions I had! Really had a lot of fun along the way!
Onwards and forwards to OOP!

Avijit Pandey
October 17, 2019 at 9:16 am · Reply

Hi! Here is my attempt to the recursive version for the 3rd problem.

```cpp
#include <iostream>
#include <cassert>
int binarySearch(int *array, int target, int min, int max)
{
    assert(array && "array not found.");
    int cValue = min + (max-min)/2;
    //Base Case
    if(array[cValue] == target)
        return cValue;
    if(min > max)
        return -1;

    //Recursive Assumption
    if(array[cValue] > target)
        return binarySearch(array, target, min, cValue-1);
    else if(array[cValue] < target)
        return binarySearch(array, target, cValue + 1, max);
}

int main()
{
    int array[] = { 3, 6, 8, 12, 14, 17, 20, 21, 26, 32, 36, 37, 42, 44, 48 };

    // We're going to test a bunch of values to see if they produce the expected results
    const int numTestValues = 9;
    // Here are the test values
    int testValues[numTestValues] = { 0, 3, 12, 13, 22, 26, 43, 44, 49 };
    // And here are the expected results for each value
    int expectedValues[numTestValues] = { -1, 0, 3, -1, -1, 8, -1, 13, -1 };

    // Loop through all of the test values
    for (int count=0; count < numTestValues; ++count)
    {
        // See if our test value is in the array
        int index = binarySearch(array, testValues[count], 0, 14);
        // If it matches our expected value, then great!
        if (index == expectedValues[count])
             std::cout << "test value " << testValues[count] << " passed!\n";
        else // otherwise, our binarySearch() function must be broken
             std::cout << "test value " << testValues[count] << " failed.  There's somethin
    }

    return 0;
}
```

I wanted to ask why you chose not to include the last condition(that the array can no longer be halved) as the base case.
Also, please point out any mistakes that I'm not noticing.

nascardriver
October 18, 2019 at 12:01 am · Reply

- Initialize your variables with brace initializers.
- Inconsistent formatting. Use your editor's auto-formatting feature.
- Line 16 is always true. The way your code is now, `binarySearch` is missing a `return`-statement.
- Line 10 could be `min >= max`. If `min == max`, then the array segment's length is 1 and you know that this last element is not the target because of line 8.

Checking equality at the beginning is wasteful. It's unlikely that this check is true as long as the array segment's length is not 1. It's much more likely that the target is to the left or right of the mid point, so

those checks should run first.

Anastasia
August 16, 2019 at 6:20 am · Reply

Hi!
I'm not sure whether it belongs here (sorry if it doesn't), but I wanted to put binary search in a context in order to see better how it works. So I made it play hi-lo game we've coded in chapter 5 quiz (it wins always).

```
#include <iostream>
#include <vector>
#include <random>
#include <ctime>

namespace mersenne {

    std::mt19937 random { static_cast<std::mt19937::result_type>(std::time(nullptr)) };

}

namespace hi_lo {

    enum {
        MIN_VALUE = 1,
        MAX_VALUE = 100
    };

    enum class Guess {
        TOO_HIGH,
        TOO_LOW,
        CORRECT
    };

    int generateSecretNumber()
    {
        std::uniform_int_distribution<> random_number { MIN_VALUE, MAX_VALUE };
        return random_number(mersenne::random);
    }

    static const int secret_number { generateSecretNumber() };
    static constexpr int num_tries { 7 };
}

void fillWithValues(std::vector<int> &vec)
{
    vec.reserve(hi_lo::MAX_VALUE - hi_lo::MIN_VALUE);

    for (int value { hi_lo::MIN_VALUE }; value <= hi_lo::MAX_VALUE; ++value)
        vec.push_back(value);
}

hi_lo::Guess checkGuess(const int guess)
{
    if (guess > hi_lo::secret_number) {
        std::cout << "Your guess is too high." << '\n';
        return hi_lo::Guess::TOO_HIGH;
    }

    else if (guess < hi_lo::secret_number) {
```

```
51          std::cout << "Your guess is too low." << '\n';
52          return hi_lo::Guess::TOO_LOW;
53      }
54
55      else {
56          std::cout << "Correct! ";
57          return hi_lo::Guess::CORRECT;
58      }
59  }
60
61  bool playWithBS(const std::vector<int> &values)
62  {
63      int min { 0 };
64      int max { static_cast<int>(values.size() - 1) };
65      int try_num { 1 };
66
67      while (try_num <= hi_lo::num_tries) {
68          int center { (max - min) / 2 + min };
69          int guess {
70              static_cast<int>(
71              values.at(
72              static_cast<std::vector<int>::size_type>(
73              center)))
74          };
75          std::cout << "Guess #" << try_num << ": " << guess << '\n';
76
77          switch (checkGuess(guess)) {
78              case hi_lo::Guess::TOO_HIGH:
79                  max = center - 1;
80                  break;
81
82              case hi_lo::Guess::TOO_LOW:
83                  min = center + 1;
84                  break;
85
86              case hi_lo::Guess::CORRECT:
87                  return true;
88          }
89
90          ++try_num;
91      }
92
93      return false;
94  }
95
96  void playHiLo()
97  {
98      std::cout << "Let's play a game. I'm thinking of a number. You have " <<
99                  hi_lo::num_tries << " tries to guess what it is (" << hi_lo::MIN_VALUE <<
100                 << hi_lo::MAX_VALUE << ")." << '\n';
101
102     std::vector<int> values {};
103     fillWithValues(values);
104
105     if (playWithBS(values))
106         std::cout << "You win!" << '\n';
107
108     else
109         std::cout << "You lose! The correct number was " << hi_lo::secret_number <<
110                     '\n';
111 }
112
```

```
int main()
{
    playHiLo();
    return 0;
}
```

Line 69 (`int guess { ... };`) is ridiculous, is there a better way to handle/avoid those casts?

nascardriver
August 16, 2019 at 7:01 am · Reply

Hi!

> Line 69 (`int guess { ... };`) is ridiculous, is there a better way to handle/avoid those casts?
You don't need the cast in line 70. Your vector is a vector of int, you're casting int to int. You only need the cast for the index.

Your entire vector is not needed. It's a contiguous list of numbers, `values[i] = (i + 1)`, you can calculate that on the fly.

```
int guess{ center + 1 };
```

It's still a binary search if you do this, but with an imaginary list.

Anastasia
August 16, 2019 at 7:32 am · Reply

> You don't need the cast in line 70. Your vector is a vector of int, you're casting int to int. You only need the cast for the index.
I could swear my compiler kept complaining about types untill I did what I did with line 69. You're right though, I see it now.

> Your entire vector is not needed.
But what if I want to make it guess from a different set of numbers (say from 150 to 500)? The way I did it, I'd only need to change MIN_VALUE and MAX_VALUE respectively.

nascardriver
August 16, 2019 at 7:41 am · Reply

```
// 1 was magic
int guess{ center + hi_lo::MIN_VALUE };
// I'm a wizard
```

Anastasia
August 16, 2019 at 7:48 am · Reply

Yes, I see. All this stuff with indexes was useless altogether, BS could have used the values directly. Oh well, I guess my abstract thinking needs a lot of work still. Thanks!

A
July 20, 2019 at 7:02 pm · Reply

Hello,
I've checked my code against the solution several times, and can't seem to find anything wrong, yet each time I run it, it states each test value passed but the last. Am I missing something?

```
#include <iostream>
#include <cassert>

int binarySearch(int *array, int target, int min, int max) {
    assert(array);

    while (min <= max) {
        int midpointValue{min + ((max - min) / 2)};
        if (array[midpointValue] > target)
            max = midpointValue - 1;
        else if (array[midpointValue] < target)
            min = midpointValue + 1;
        else
            return midpointValue;
    }
    return -1;
}

int main() {
    int array[] {3, 6, 8, 12, 14, 17, 20, 21, 26, 32, 36, 37, 42, 44, 48};

    // Testing 9 different values against above array
    const int numTestValues{9};
    int testValues[numTestValues] {0, 3, 12, 13, 22, 26, 43, 44, 49};
    // Expected results of each value
    int expectedValues[numTestValues] {-1, 0, 3, -1, -1, 8, -1, 13, 1};

    for (int count{}; count < numTestValues; ++count) {
        // Check if test value is in array
        int index{binarySearch(array, testValues[count], 0, 14)};
        // Returns true if matched expected value
        if (index == expectedValues[count])
            std::cout << "Test value " << testValues[count] << " passed!\n";
        else
            std::cout << "Test value " << testValues[count] << " failed.\n";
    }
    return 0;
}
```

nascardriver
July 21, 2019 at 12:34 am · Reply

Change line 35 to

```
std::cout << "Test value " << testValues[count] << " failed: " << index << '\n';
```

to see what your `binarySearch` is saying. Compare the result to what you expected (Manually search 49 in the array) and what your `expectedValues` says it should be.

A
July 21, 2019 at 7:44 am · Reply

Ah, I see what I did wrong. I was also misinterpreting what the output was meant to be. Thank you for your help!

mmp52
July 3, 2019 at 3:03 am · Reply

Hey, this might seem stupid but why didn't he use const double's when defining max Function? Aren't we supposed to use const everywhere if we don't manipulate the input?

nascardriver
July 3, 2019 at 3:06 am · Reply

Use const for references and pointers. It doesn't matter for return/copy-by-value variables, since modifying them doesn't affect the caller/callee.

mmp52
July 5, 2019 at 6:26 am · Reply

thanks!

« Older Comments 1 2 3