7.x — Chapter 7 comprehensive quiz

BY ALEX ON DECEMBER 4TH, 2015 | LAST MODIFIED BY ALEX ON MAY 19TH, 2018

Chapter 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 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 parameter is a function parameter that has a default value provided. If the caller doesn't explicitly pass in a value for the default parameter, 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.

Quiz time!

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.

Hide Solution

```
double max(double x, double y);
```

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

Hide Solution

```cpp
void swap(int &x, int &y);
```

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).

Hide Solution

```cpp
// Note: array can't be const in this case, because returning a non-const reference to a const element would be a const violation.
int& getLargestElement(int *array, int length);
```

2) What’s wrong with these programs?

a)

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

Hide Solution

doSomething() returns a reference to a local variable that will be destroyed when doSomething terminates.

b)

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

Hide Solution

function sumTo () has no termination condition. Variable value will eventually go negative, and the function will loop infinitely until the stack overflows.

c)

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

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

Hide Solution

The two divide functions are not distinct, as they have the same name and same parameters. There is also a potential divide by 0 issue.

d)

```cpp
#include <iostream>

int main()
{
    int array[100000000];

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

    return 0;
}
```

Hide Solution

The array is too large to be allocated on the stack. It should be dynamically allocated.

e)

```cpp
#include <iostream>

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

    return 0;
}
```

Hide Solution

argv[1] may not exist. If it does, argv[1] is a string argument, and can’t be converted to an integer via assignment.

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:

```cpp
// 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 otherwise
int binarySearch(int *array, int target, int min, int 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 something wrong with your code!\n";
    }

    return 0;
}
```

3a) 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.

Hide Solution

```cpp
#include <cassert>

// 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 otherwise
int binarySearch(int *array, int target, int min, int max)
{
        assert(array); // make sure array exists

    while (min <= max)
    {
        // implement this iteratively
        int midpoint = min + ((max-min) / 2); // this way of calculating midpoint avoids overflow

        if (array[midpoint] > target)
        {
            // if array[midpoint] > target, then we know the number must be in the lower half of the array
            // we can use midpoint - 1 as the upper index, since we don't need to retest the midpoint next iteration
            max = midpoint - 1;
        }
        else if (array[midpoint] < target)
        {
            // if array[midpoint] < target, then we know the number must be in the upper half of the array
            // we can use midpoint + 1 as the lower index, since we don't need to retest the midpoint next iteration
            min = midpoint + 1;
        }
        else
            return midpoint;
    }

    return -1;
}
```

3b) Write a recursive version of the binarySearch function.

Hide Solution

```
#include <cassert>

// 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 otherwise
int binarySearch(int *array, int target, int min, int max)
{
        assert(array); // make sure array exists

    // implement this recursively

    if (min > max)
        return -1;

    int midpoint = min + ((max-min) / 2); // this way of calculating midpoint avoids overflow

    if (array[midpoint] > target)
    {
        return binarySearch(array, target, min, midpoint - 1);
    }
    else if (array[midpoint] < target)
    {
        return binarySearch(array, target, midpoint + 1, max);
    }
    else
        return midpoint;
}
```

8.1 -- Welcome to object-oriented programming

Index

7.14 -- Ellipsis (and why to avoid them)

Share this:

Facebook Twitter Google Pinterest

C++ TUTORIAL | PRINT THIS POST

111 comments to 7.x — Chapter 7 comprehensive quiz

« Older Comments 1 2

Vamshi malreddy
July 6, 2018 at 4:55 pm · Reply