20.x — Chapter 20 summary and quiz

Chapter Review

Another chapter down! 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.

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.

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 integers (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>
#include <iterator>

// 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(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, static_cast<int>
(std::size(array)) - 1) };
        // 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;
}
```

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` returns 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.