Study Guide
ECS 20: Discrete Math for CS

Computer Science Tutoring Club
Fall 2018

December 29, 2018

Contents

Chapter 1

Introduction to ECS 36B

There are a number of overarching learning objectives for the ECS 36B class, listed on the CS department's website. The four main ones are:

1. Object-oriented programming in C++
2. Basic data structures and their use
3. Writing good programs of increased complexity and efficiency
4. Methods for debugging and verification

We will cover each of these concepts in separate chapters. Of the four, the first (object-oriented programming in C++) is usually focused on the most during the course, while the other three are often learned "along the way" through complex homework assignments.

As with all study guides, this is not meant to be a comprehensive review of the material and should not be used as your only study materials. We aim to present "just the highlights" of the course, which is hopefully helful for narrowing down exactly what you need to spend more time studying.

Chapter 2

Object-Oriented Programming in C++

In 36B, one of the primary topics is *object-oriented programming* (OOP), a programming paradigm supported by the C++ language (along with many others). As with most programming paradigms, OOP can be interpreted and taught in a variety of different ways, and you will probably want to make sure that you understand the interpretation provided by your professor in class.

In this chapter, OOP will be presented as a common *programing pattern* that has been formalized into the language itself. There are, however, many other approaches to presenting the concept of OOP, so please consider reading (insert links here).

2.1 Programming Patterns

Programming languages exist, at least partially, to allow human beings to effectively communicate commands to a computer. To that end, there are a number of concerns that make...

Consider the following loop pattern:

```
int i = 0;
while (i < 10) {
    // Do something with i
    i++;
}
```

This pattern of initialize-compare-increment-loop is extremely common, as it is the pattern to use whenever you want to increment or count through a certain finite number of things. It is so common, in fact, that a special syntax was introduced for loops like this: the for loop. Using a for loop, the above could be re-written like so:

```
for (int i = 0; i < 10; i++) {
    // Do something with i
}
```

Using the for loop, not only does the code become shorter, but the *intention* behind the code itself is easier to determine at a glance – we are performing some action (the loop body) over all integer i values from 0 until 10. In this way, even though the for loop does not actually expand the types of programs you could write (anything you could do with a for loop can be done with a while loop), it makes your code easier to read, understand, and maintain.

Similarly, many of the OOP features of C++ can be seen as language features that formalize particularly common patterns in regular C code, making their use easier to understand and maintain. Consider, for a first example, the following C code, which describes a simple dynamic array struct and associated helper methods in C:

```
struct IntegerList {
    int *data = NULL;
    int length = 0;
}

void append_to_integer_array(struct IntegerList *array,
                             int value) {
    array->length += 1;
    array->data = (int *)realloc(data,
                                 array->length * sizeof(int));
    array->data[array->length - 1] = value;
}

int get_item_integer_array(struct IntegerList *array,
                           int index) {
    if (index < 0 || index >= array->length) {
        printf("Error:␣array␣indexed␣out-of-bounds.\n");
        exit(1);
    }
    return array->data[index];
}

int main(void) {
    struct IntegerList array;

    append_to_integer_array(&array, 5);
    printf("%d\n", get_item_integer_array(&array, 0));
    return 0;
}
```

In this code, the helper methods are inherently tied to the struct; together, they conceptually describe the concept of an "integer array" that can be appended to or indexed. In fact, without the struct itself, the helper functions would be useless: their only purpose is to modify or read from the struct itself.

This is a very common pattern, and likely one that you have written yourself in ECS 36A. In C++, this pattern is formalized into the concept of a *class*. We will proceed step-by-step through some of the language features that C++ introduces to formalize this pattern.

2.2 Member Methods

The first thing we can do is *move the functions into the struct itself*, indicating that they are directly tied to the meaning of that struct:

```
struct IntegerList {
    int *data = NULL;
    int length = 0;

    void append(int value) {
        this->length += 1;
        this->data = (int *)realloc(data,
                                    this->length * sizeof(int));
        this->data[this->length - 1] = value;
    }

    int get(int index) {
        if (index < 0 || index >= this->length) {
            printf("Error: array indexed out-of-bounds.\n");
            exit(1);
        }
        return this->data[index];
    }
}

int main(void) {
    struct IntegerList array;

    IntegerList::append(&array, 5);
    printf("%d\n", IntegerList::get(&array, 0));

    return 0;
}
```

Now, we notice a number of interesting facts. First, the “append" and “get" methods no longer really need their postfixes, as it is now clear which data type they operate on. In fact, however, they seem to automatically pick up a “family name," as when we call member methods in this form we need to first specify the struct’s name (“IntegerArray") followed by two colons, then the name of the method itself (we will discuss this syntax more in a later section). Finally, we notice that (by default) all of the member methods of a class have an *implied* first argument, “this," which is a pointer to whatever struct the method “lives in." Here, “this" directly replaces the “array" parameter that we used previously.

Notably, when there are no naming collisions, “this->" is unnecessary in C++ member methods. In the above code, for example, all instances of “this->" could be removed without changing the meaning of the code.

2.3 Calling Syntax

The next thing we can do is change how we call these methods. If we know we’re calling a member method, then (in all cases we have seen so far) we can just look at the type of the first argument passed in to see which type of struct we are dealing with. In the example above, we see the call “IntegerArray::append(&array, 5)," which has a first argument of type “struct IntegerArray *". Thus, explicitly stating that we want to call **IntegerArray::**append is redundant, and we can instead use the C++ syntax “array.append(5)" which produces exactly the same effect while being less redundant and significantly easier to read and understand:

```
int main(void) {
    struct IntegerList array;

    array.append(5);
    printf("%d\n", array.get(0));

    return 0;
}
```

2.4 Access Keywords and Data Hiding

One fundamental concept in OOP is that of data hiding and the separation of interface and implementation. Essentially, we want objects to expose *interfaces*: a set of operations (such as appending to a list) that we can perform on (or with) that object, however, as the consumer of that object, we do not want to concern ourselves with the actualy implementation of it.

Ideally, this interface should be able to remain the same even as the underlying implementation changes for performance, maintanability, or feature reasons.

For example, the C programming language itself can be seen as just such an interface. Although the underlying hardware architecture (and instruction set exposed by the CPU) may change from computer to computer, the C language itself stays the same. This “abstraction" of the implementation details and separation of concerns between different levels of a system allows each level to be improved and optimized semi-independently, allowing for more rapid and robust improvement of the system as a whole.

Our IntegerArray class, as we have written it, currently stores the underlying data as a dynamically-allocated, contiguous array of memory locations. However, in the future, we may find that storing the array as a linked list is actually more efficient on our particular hardware achitecture or use case. In that case, we would need to replace our “data" pointer with a pointer to a linked list node, and update the contents of our “append" and “get" methods to perform the relevant operations on the linked-list structure. Importantly, however, the function *signatures* will not change: append, for example, still takes in the integer to add to the end of the list and still returns nothing. Thus, all of the code we have written in main will still work perfectly. This might look something like:

```
struct IntegerList {
    IntegerListNode *head = NULL;

    void append(int value) {
        // First find the end of the list
        for (IntegerListNode *node = this->head;
             node != NULL;
             node = node->next) {}
        // etc.
    }

    int get(int index) {
        IntegerListNode *node = this->head;
        while (node != NULL && index > 0) {
            node = node->next;
            index--;
        }
        if (node == NULL) {
            printf("Error: array indexed out-of-bounds.\n");
            exit(1);
        }
        return node->value;
    }
}

int main(void) {
    struct IntegerList array;
    array.append(5);
    printf("%d\n", array.get(0));

    return 0;
}
```

Notice that we did not need to change the definition of main at all.

However, there is one issue: if in main we had been *directly* reading from array.data, instead of going through member methods like ".get" like so:

```
int main(void) {
    struct IntegerList array;

    array.append(5);
    printf("%d\n", array.data[0]);

    return 0;
}
```

Then this change would break main, as there is no longer a "data" member on the IntegerArray struct. For this reason, we want ot ensure that consumers of our class always use the member methods to access data, and never directly "touch" the underlying implementation. In this case, we want to "hide" data and length, so that users of your class have to go through the append and get methods instead.

In C, you might do this by, say, prepending a "_" to a data member's name to specify that you should be carefully when using it. However, C++ provides a more explicit way to do this through the use of access keywords(?). Essentially, we can designate some of the members of our struct as "private," so that any attempt to directly access them outside of the struct itself will cause an error:

```
struct IntegerList {
    private:
        int *data = NULL;
        int length = 0;

    public:
        void append(int value) { /* ... */ }
        int get(int index) { /* ... */ }
}
```

```
int main(void) {
    struct IntegerList array;

    array.append(5);

    // This is OK:
    array.get(0);

    // This throws an error:
    array.data[0];

    return 0;
}
```

In this way, as long as we do not change the actualy method signatures, we can safely change the implementation of IntegerList without having to worry about whether we will be breaking any code that uses IntegerList.

2.5 “Class" Keyword

At this point, we have conceptually transitioned from defining a simple data structure with some associated helper methods to an entire, coherent conceptual *type* (or class) of object which has associated methods and an abstracted implementation. To highlight this, we can use the C++ keyword “class" instead of “struct," and drop the “struct" keyword when creating instances:

```
class IntegerList {
    private:
        int *data = NULL;
        int length = 0;

    public:
        void append(int value) {
            // ...
        }

        int get(int index) {
            // ...
        }
}

int main(void) {
    IntegerList array;
    array.append(5);
    printf("%d\n", array.get(0));

    return 0;
}
```

We note that this is primarily a cosmetic difference: the only meaningful effect of using “class" instead of “struct" here is that, by default, all class members are private (which is unimportant in this example, since we explicitly set the access levels of all members).