## What is DSA

Data Structures and algorithms or DSA is a combination of two interrelated topics.

- Data Structures: Refers to the patterns in which we structure data sets, typically with the goal of being as efficient in it's storage and parsing as possible.
- Algorithms: Sets of steps that are meant to accomplish something. In the context of DSA, algorithms are used to perform operations on data, like parsing large amounts of data efficiently to match certain patterns, or to perform operations with said data in the most effective way possible.

As computers become more and more complex, the problems that we need to solve with them also become more and more complex. As such, unoptimized algorithms and data structures, can take large amounts of resources unnecessarily, increasing computation costs and time. With proper algorithm analysis skills, a programmer should be able to identify the bottlenecks in performance inside of a given operation, as well as potential sources of improvement.

It's because of this that data structures and algorithms remains one of the most important subjects of study for aspiring software engineers.

# A general overview of programming

The first chapter of the book, covers a summary of select computer science and math topics that will be of significance for later topics.

It begins by giving a more in depth overview of what's to come, and the importance of DSA studies, by means of proposing 2 real life examples of problems that are relatively easy to come up with straightforward programming solutions for, finding the "k-th" largest element in an array, and solving a word puzzle. In both cases, the book offers 2 potential solutions that are very direct and easy to visualize, however, it also presents the problem that said solutions bring to the table, the fact that they take unreasonable amounts of time to run for large data sets.

This further emphasizes the importance of DSA and proposes algorithm analysis as a potential solution for our troubles.

# Mathematics Review

This section goes over some of the relevant math topics for DSA study. As mathematics is not the main focus of this review, we can reduce the section to key concepts that we can explore later if we need refreshers on. Said key concepts are:

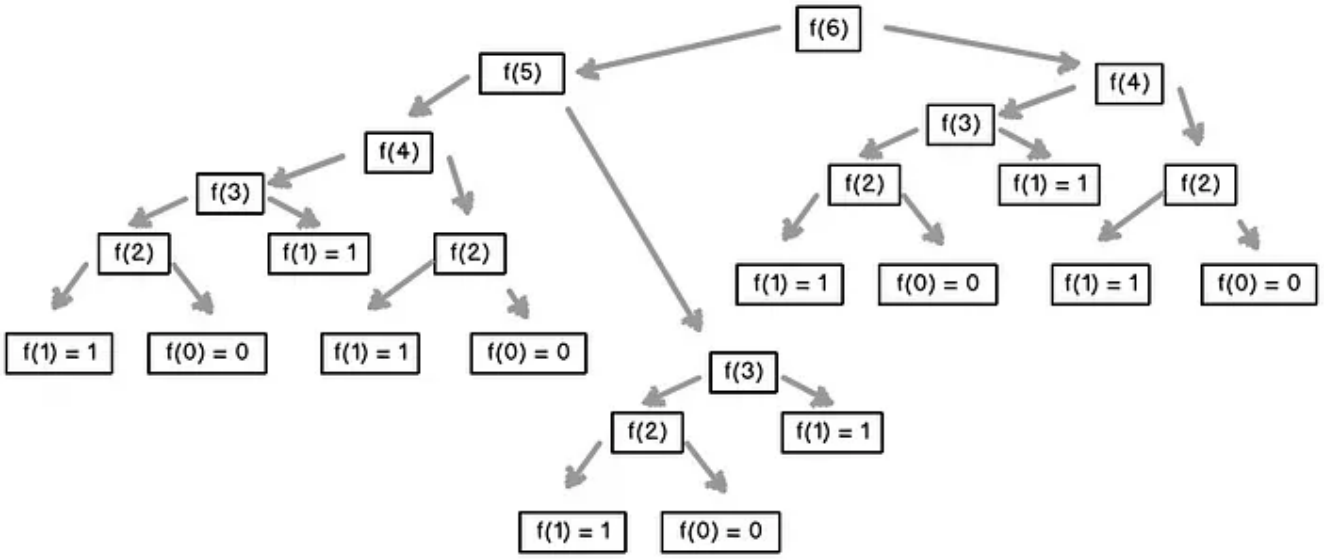
- Exponents and Exponent rules
- Logarithms
- Series
- Modular arithmetic
- Proofs by Induction, Contradiction and Counterexample

# Recursion

In this subchapter, the book explores the concept of recursion. Recursion is a mathematical concept that means "a problem defined in terms of itself", in other words, recursion refers to a mathematical expression, that has a simpler version of itself as part of it's definition. Professor Germain from the University of Utah, uses a very appropriate example to describe this concept, the mathematical formula of the Fibonacci sequence:

$$F(i) = F(i-1) + F(i-2)$$

In which we can clearly see how the formula uses itself to be defined, by inserting a number into it we can continue to operate on these numbers until we hit a minimum value, here's a good graphical representation provided by article writer der Merwe:



Of course, if not given a limit, the recursion can go forever, which is not ideal, as such it's important to understand the concept of a "base case", as introduced by the DSA Analysis in C++ book. This concept refers to the cases in which the result values can be known without need of recursion. We can see such an example if we code this function the following way:

```
int fib(int x){
        if(x < 2){
                return x;
        }else{
                return fib(x - 1) + fib(x - 2);
        }
}
```

The case value returns x when x is less than 2, considering the way in which the function will substract values in increments of only 1 and 2 at a time, this means our case values will return 1 and 0. After this, the return statements will resolve providing each iteration of our recursive function with increasingly higher values following the formula of the Fibonacci sequence.

There is however a couple of problems with recursion. It's very easy to mistakenly create circular logic by creating a situation in which f(n) either calls itself (as f(n) instead of a variation of f(n)) or in which some variation of f(n) that was arrived at from f(n) resolves to n again. In other words, recursion can be infinitely recursive if we don't adhere to these 2 rules:

- Always have a base case
- Always make progress towards the base case

Lastly, the book mentions a third important rule. Assume all recursive calls work. We can say this because whether a recursive function works or not can be proven with mathematical induction, in which it is assumed that given a base case, the smallest call given by any arbitrary positive integer is true, then proven for the next case. Meaning that if we can prove the function works, any mistakes will be logical rather than conceptual. This saves us a lot of work in debugging, since we don't have to slowly trace back every single recursive call in any given function call.

## Classes

A class is one of the key concepts of the object-oriented programming paradigm, it can be thought of as a template that allows us to create objects which store data and provide functionality to manipulate said data. This paradigm originates in the late 60's and early 70's but has become one of the most prevalent paradigms of programming even today, used in a wide variety of programs of all sizes and levels.

A class, allows us to define an object by it's members, composed of attributes and methods, which give the object state and functionality.

Object-oriented programming offers many benefits like modularity which provides us with the ability to keep the functionality of a program separated in small functional pieces that are more reusable and maintainable; abstraction, which provides a way for users and higher-level programmers to utilize our interfaces without having to concern themselves with the logic that lies underneath them; encapsulation, which aids security by means of protecting data from accidental or unauthorized manipulation; inheritance, which reduces code repetition and allows us to create systems that have logical links in their design as well as in their implementation; and polymorphism which allows us to abstract type-specific functionality, making programs more flexible.

The book attacks these concepts directly with a clear emphasis in functionality by means of analyzing the class "IntCell", which contains a single data member and methods to be able to access it and manipulate it.

## Initial Concepts

We'll tackle the concept of a constructor first. A constructor is a method that describes how to instantiate and object of the given class. We have several types of constructors, but for the time being we tackle the default constructor, which takes in no parameters and gives the data member a default value; and the parameterized constructor, which takes in an initial value and initializes the object's data member through the use of an initialization list.

An initialization list assigns the values to the appropriate members during object creation, not to be confused with a list of assignment operations also sometimes seen in constructors of this kind. The distinction can be important, as the code inside the brackets is run after the object's been initialized already, which on top of being less efficient it can also give problems if any of the data members are classified as constants, which would disallow initialization once the object has already been instantiated.

The IntCell class's parameterized constructor also uses the explicit keyword, which is important due to the fact that C++ by default handles object initialization through the use of implicit type conversions, which creates a temporary object with the new parameter assigned to it and then copies the values of said temporary object into the new one. By making it explicit this is prevented, as a one parameter constructor won't be able to create implicit temporary objects, thus enforcing stronger type safety and preventing subtle bugs that may or may not arise from unseen implicit type conversions.

When it comes to member functions there's a few types that stand out. Accessors and mutators are a pattern that has been used in many programs all throughout the different branches of programming. As the name implies, Accessors and mutators are methods that are used to access and manipulate data in objects that would normally be accesible due to encapsulation. This is also a good time to talk about a couple miscellaneous concepts, for example when it comes to encapsulation, access specifiers are used in classes to indicate whether or not certain parts of the definition should be accessible to the entire program, classes that inherit from it or only the specific instance itself, these are indicated with the keywords public, protected and private respectively. Lastly, the book marks the importance of defining the accessor method as a constant method through the use of the keyword const after the parenthesis in the declaration; this is done because the accessor method should not have the power to change the state of the data member.

Next, a common concept in OOP (Object-Oriented Programming) is the separation of interface and implementation. First of all, why separate the interface from the implementation in the first place. As Dr. Bruce Martin writes in J. Waldo's book:

> "In a distributed environment, allowing multiple implementation lattices for an interface is essential. Interfaces are global to the distributed environment, while implementations are local. For a distributed program that crosses process, machine and administrative boundaries, maintaining a single implementation of an interface is difficult, if not impossible."
> - Bruce Martin, 1993

What this means essentially is that in the modern scope of programming, systems will often require to support different implementations, with extended functionality, either through the use of different versions of certain software, or in situations in which the differences in the context that surrounds the program, like the hardware it's being run on or the different forms in which systems may fill dependencies. We can draw a very clear example in the context of videogame programming in the form of graphic API's. What happens when you want to write a game engine that runs in multiple platforms when

these platforms make use of proprietary API's (like DirectX for windows)? Well you could make 2 completely different versions of the engine AND the games that will use it, or you could create an interface that will be the same for all games and simply create different implementations that make use of different API's, then check which API's are available in the different platforms at the time of shipping or during installation and swap out the implementations. This allows us to support different kinds of platforms without having to create 2 completely different systems.

Now that we have settled the reasons why it's important, we can talk about the specifics of utilizing such a system. The book first and foremost introduces us to the concept of preprocessor commands, particularly it mentions the following commands:

- `#ifndef` which indicates to the compiler that the block of code to follow should not be compiled if the particular variable has already been defined.
- `#endif` which closes the block of code opened by the previous command.
- `#define` which defines a variable for the compiler that holds the continuing block of code.
- `#include` which includes a defined block of code into the current one.

These are used to properly link the different parts of our program into one cohesive sum. By including the interface in both the places it's needed and the place in which it's defined we can effectively separate our implementation without compromising the program, therefore boosting it's flexibility. It's important to only define a new block of code if it hasn't been defined before, as this causes problems for the compiler.

In terms of the implementation itself 2 more things need to be addressed, first because of the way scopes are handled in C++, we need to utilized the scope resolution operator [::] to indicate to our program that the following definitions belong to the member functions of our interface; second, that the signatures must match one to one, including parameters and specifiers.

To begin closing our section in classes we need to talk about the fact that classes are treated semantically as primitive types. What this means is that we can use the name of our class to declare new objects just like we would declare a new variable of a primitive type, like an int or a char. The reason this is important to mention is that confusion can easily arise when not treated as such, like when declaring a function that has said class as it's return type.

```
IntCell obj1; // Zero-parameter constructor call
```

```
IntCell obj2( ); // Function declaration
```

This is a common mistake beginner programmers make when trying to utilize the zero-parameter constructor of a class. Therefor modern C++ guidelines often push towards using the initializer list syntax with curly braces instead, which makes the code more uniform.

### Vector and string

In the C programming language arrays are not treated as first-class citizens, this can lead to a myriad of problems as not having access to the typical operations that other entities have greatly reduces the flexibility and usability of arrays. In addition to this, the built-in string type is simply implemented as an array of characters meaning it inherits all of the problems that C-style arrays have. Because of this, the STL introduces the std::vector and std::string classes, converting these 2 into first-class citizens, giving them the ability to be used with standard operators, hold information about themselves such as their length and add features such as size flexibility and extra security.

As an added bonus, the book makes reference to the range for loop syntax added in C++ 11, which allows us to declare a for loop that will access every item in a structure sequentially without requiring the iterators. It uses the following syntax:

```
int sum = 0;
for( auto x : squares )
        sum += x;
```

## C++ Details

This section of the book makes note of more miscellaneous concepts of C++ programming that are important for the understanding of different data structures and algorithms' implementation. For this summary they will be listed sequentially and briefly described:

### Pointers

A pointer is a variable that stores an address to another object's position in memory. It is very useful int he implementation of dynamic memory allocation as we will often require to access an unknown quantity of objects in unknown locations in memory. They are declared in with the following syntax:

```
int main( )
{
        IntCell *m;
        m = new IntCell{ 0 };
        m->write( 5 );
        cout << "Cell contents: " << m->read( ) << endl;

        delete m;
        return 0;
}
```

Couple of things to mark about pointer syntax, is that in order to dynamically allocate an object in C++, we need to use the "new" keyword, which reserves a new space in memory with the size of the object type. It is ***very important*** to stress the point that C++ (unlike other languages) does not have a garbage collector and unlike standard variables, values initialized with the "new", keyword are not automatically deleted once the scope for them ends. This is because the reason why scoped deletion happens in the first place, is that locally declared variables are stored in the stack section of the C++ memory model, as the scopes are finished, the last elements of the stack are liberated as well. However when declaring pointers, the only thing that is being pushed into the stack is the pointer itself (the variable that holds the address) ***not*** the value inside of the dynamically allocated address itself which resides in the heap and is not automatically deleted.

This is crucial to understand because what it means is that the pointer, which was our only way to access the dynamically allocated memory, will be freed once it's scope ends but not the memory we allocated, creating a situation in which the memory itself is still allocated but we no longer have a way to access it. This is known as a memory leak and in order to prevent them, we have to make sure that we use the delete keyword to signal for deletion any dynamically allocated memory segment we create ***or*** that we make sure to use memory safe pointers (or smart pointers) which include the code to free the resource in their own destructor method essentially giving ownership of dynamically allocated resources to a stack allocated object.

Lastly a couple more concepts to take into consideration is the related operators:

- Assignment and comparison operators (= ; == ) can be used to pass in a memory address to another pointer or compare if 2 pointers point to the same memory address.
- The member access operator (->) can be used to access a member of the object being pointed at.

- The address-of operator (&) can be used to return the memory location of an object.

## Lvalues, Rvalues and References

Before delving into references, the book defines lvalues and rvalues:

- Lvalue: Defines a non-temporary object. It can be modifiable or not (i.e. when using const).
- Rvalue: Defines a temporary object or is a value.
  The book provides this example:

```
vector<string> arr( 3 );
const int x = 2;
int y;
...
int z = x + y;
string str = "foo";
vector<string> *ptr = &arr;
```

In it we can see a group of lvalues such as "x", "y", "str" and "*ptr", as well as a group of rvalues such as "2" or "x + y"; In general we can refer to lvalues as expressions that we can save and (often) modify, where as rvalues are often raw data in the form of literals or expressions that are never saved or modified but can be used to define lvalues.

Understanding these concepts we can begin explaining references. An **lvalue reference** is one in which we transform an expression into another way to refer to the same object. This is useful when we need to directly manipulate an object but there is no convenient way of calling it directly. In the following example we can see it clearly in use:

```
auto & whichList = theLists[ myhash( x, theLists.size( ) ) ];
if( find( begin( whichList ), end( whichList ), x ) != end( whichList ) )
        return false;
whichList.push_back( x );
```

As we can see, calling the object that we are referencing with whichList, is extremely verbose, however simply declaring a new variable and assigning such value to it would not modify the original value but rather create a new one and modify it instead. As such if we need to access the value directly we have to use an lvalue reference.

As such many of the use cases for lvalue references will be avoiding copies of the same object and instead affecting the object directly, like inside of for range loops or when trying to modify an object inside a data structure directly while having a variable to call it individually.

Lvalue references are declared with an & after some type. As such these are sometimes confused with the usage of the address-off operator which is also expressed with an &. It's important to understand that the address-of operator returns a pointer to an address; the lvalue reference instead creates an alias to an object. They are not the same.

An **rvalue reference** on the other hand has the unique property of being also able to temporarily reference an rvalue.

Now to understand why rvalue references can be useful we need to dive deeper into parameter passing.

## Parameter passing

Parameters are typically passed in to functions by a call-by-value mechanism; that's to day that the value is copied into the parameter itself, creating a new variable with said value. However there are many different ways in which we could potentially want to use parameters in a function, many of which are ineffective or simply disfunctional when using call-by-value. As such we have different ways of passing values into functions:

- Call-by-value: The default one which involves copying the value into the parameter.
- Call-by-reference: Or call-by-lvalue-reference, which instead of passing in a copy of the value to the parameter, it sets the parameter as an alias for the original variables that are being passed in, which makes us able to change the original variables from within the context of our function.
- Call-by-reference-to-a-constant: Which can be used in situations in which we only need to "view" a value without modifying it. The reason why we don't simply make a call-by-value is because the copying operation can be extremely costly when handling large amounts of data, but we also can't simply use a call-by-reference because we run the risk of modifying a value that should not be modifiable. As such by making our reference constant through the use of the "const" specifier we can access the best of both worlds, preventing the costly operation without risking our original value being modified.
- Call-by-rvalue-reference: The methods above successfully handle every situation except one. What if we need to modify the data and create a new value to hold this modification without changing the original data but we are still handling large

amounts of data. That's where rvalue-references can come in handy, this is because typically it's easier to simply move an entire item rather than copying it, by using an rvalue-reference we temporarily hold access to the rvalue data that represents our original lvalue being passed into the function, except that now we can more effectively assign said rvalue to a new lvalue without having to go through the entire copy operation.

### Return Passing

Same as with parameters, there are several ways to return values from a function. The most straightforward, just as in the previous section, is the return-by-value method. In this the function simply returns a value of the appropriate type as an rvalue. However since rvalues cease to exist after they've been called, if the value that is being returned is an lvalue, it will be copied as part of the return sequence even if said lvalue will continue to exist past the functions termination. For this we can implement instead a different return type like return-by-constant-reference. This will allow the implementation to avoid unnecessary copying, although of course, the returned value will have to be assigned to another constant reference or else the copying will happen anyway. In C++11 however objects can define move semantics to prevent copying and allow a simple pointer change. Therefore if we implement such move semantics we can avoid going out of our way to prevent copying by playing with different types of references. Similarly we can use return-by-reference to keep access to the data of a class as the callers of a function.

### std::swap and std::move

These functions help sum up the concepts very well. This because std::swap can easily be implemented by making copies of the values before reassigning the original ones, however this comes with a very large cost when dealing with large data types, and even though this works, it's easy to see now that it would also be easier to simply move the pointers around rather than copying the entire values, as the pointers will always have the same size regardless of the size of the data type itself. This can be easily done by using static_cast to assign new variables with rvalue references and then simply using those to reassign our original variables. We can also use std::move for the same result.

```
void swap( vector<string> & x, vector<string> & y )
{
        vector<string> tmp = static_cast<vector<string> &&>( x );
        x = static_cast<vector<string> &&>( y );
        y = static_cast<vector<string> &&>( tmp );
}
```

```
void swap( vector<string> & x, vector<string> & y )
{
        vector<string> tmp = std::move( x );
        x = std::move( y );
        y = std::move( tmp );
}
```

### The big five

C++ has some premade definitions for 5 functions, the destructor, the copy constructor, the move constructor and the copy and move assignment operators.

- The default destructor applies the destructor on each data member, so any dynamically allocated data members that also have a destructor will be freed.
- The copy and move constructors build a new object of the same type and then copy or move a value into the new object depending on whether the value is an lvalue or an rvalue.
- The assignment operators work when used by 2 objects that have been previously constructed. They work similarly to the constructor, if the rhs value is an lvalue it will use the copy operator and if it is an rvalue it will use the move operator.

The default definitions for these 5 are perfectly fine in many cases, typically when objects are made of mostly primitive types and some objects. However if the data members contain pointers things can get complicated. The default constructor does not delete the objects that the pointer points towards, and the copy constructor and operator only copy the pointers instead of creating new pointers with new values, what we call a **shallow copy**. More often than not what we want is a **deep copy** instead, in which new pointers point to the copied values. If that's the case we need to implement the 3 ourselves. Doing so also removes the default move constructor and operator so we also have to define those 2.

```
~IntCell( ) = default; // Destructor
IntCell( const IntCell & rhs ) = default; // Copy constructor
IntCell( IntCell && rhs ) = default; // Move constructor
IntCell & operator= ( const IntCell & rhs ) = default; // Copy assignment
IntCell & operator= ( IntCell && rhs ) = default; // Move assignment
```

We can modify these as we see fit, but we do have to keep in mind that if we modify even one of them, it's good practice to modify all of them, as their functionality isn't guaranteed

(and for the move operators even gone). It's also interesting to point that the copy and move assignment operators return an lvalue-reference to allow chain assignments.

To prevent shallow copies in our operators we can follow this procedure (imagining that the stored value in IntCell is a pointer):

```
IntCell( const IntCell & rhs ) // Copy constructor
        { storedValue = new int{ *rhs.storedValue }; }

// option 1
IntCell & operator= ( const IntCell & rhs ) // Copy assignment
{
        IntCell copy = rhs;
        std::swap( *this, copy );
        return *this;
}

// option 2
IntCell & operator= ( const IntCell & rhs ) // Copy assignment
{
        if( this != &rhs )
                *storedValue = *rhs.storedValue;
        return *this;
}
```

In this we can see how the constructor initializes the new values with a newly allocated memory block that will contain a copy of the value in the original object (gotten by dereferencing rhs.storedValue). Then the copy assignment operation can either be done by using the same copy constructor to copy the values into a new, temporary item then swapped with the object in question; or by making sure that the values the pointers point to match.

The move operators can be done in a similar fashion but utilizing swaps and making sure to empty out our previous item.

```
IntCell( IntCell && rhs ) : storedValue{ rhs.storedValue }// Move constructor
        { rhs.storedValue = nullptr; }

IntCell & operator= ( IntCell && rhs ) // Move assignment
{
        std::swap( storedValue, rhs.storedValue );
```

```
        return *this;
}
```

### C-style arrays

The book briefly mentions C-style arrays. A C-style array is a pointer to a memory block that is as big as the size of the array as specified on initialization times the size of the data type to be stored. Since arrays are constant memory blocks, the variables they are assigned to are just pointers to said block, because of this, no other types of information, like the size of the array, are preserved when passing arrays into functions. The size of arrays must be known at allocation time, therefore in order to create a C-style array dynamically, we create a new pointer and make sure the size is known during initialization like such:

```
int *arr2 = new int[ n ];
delete [ ] arr2;
```

We also can't forget to delete it, arrays can be significantly more dangerous as the sizes can be pretty large. It's important to mention, that the vector and string classes are made by hiding these behaviour flaws under a class which helps ease the problems by implementing all sorts of checks and holding in additional data along with them, as well as handling on it's own the multiple allocations and deallocations. Because of this, in some rare cases it may be worth going back to C-style arrays, or even implementing our own kinds of dynamic array systems, in order to enhance performance.

## Templates

Templates in C++ are a functionality that allows us to create type independent logic. They work by defining a pattern rather than an actual implementation, then replacing the type descriptions with a generic typename (often T).

```
template <typename Comparable>
const Comparable & findMax( const vector<Comparable> & a )
{
        int maxIndex = 0;

        for( int i = 1; i < a.size( ); ++i )
                if( a[ maxIndex ] < a[ i ] )
                        maxIndex = i;
```

```
        return a[ maxIndex ];
}
```

As expressed in the example, findMax is a generic function that can take a constant reference to a vector of type "Comparable" (our generic typename), and returns a constant reference to a value of type "Comparable" as well. What this essentially means is that we can give the template function values of any type and it will attempt to perform the operations defined with them, then return a value of the same type.

```
int main( )
{
        vector<int> v1( 37 );
        vector<double> v2( 40 );
        vector<string> v3( 80 );
        vector<IntCell> v4( 75 );

        // Additional code to fill in the vectors not shown

        cout << findMax( v1 ) << endl; // OK: Comparable = int
        cout << findMax( v2 ) << endl; // OK: Comparable = double
        cout << findMax( v3 ) << endl; // OK: Comparable = string
        cout << findMax( v4 ) << endl; // Illegal; operator< undefined

        return 0;
}
```

Of course if the types don't match the necessary operations it will not perform as expected, because of this it might be smart to implement checks to aid debugging.

This, however, tends to be a problem when writing generic code, as it will mean that we will require lots of checks and even then it won't always work properly. An easy solution often seen in functional programming languages (such as Haskell) is the usage of functions as first-class citizens, to pass them as parameters into other functions. This however is not exactly possible in languages that do not support the paradigm. However in C++ a workaround can be crafted through the usage of what's known as a function object.

The basic idea is to create a specific object that holds the function itself, then by overloading the () operator one can create a similar syntax to one seen in functional programming.

```
class CaseInsensitiveCompare
{
        public:
                bool operator( )( const string & lhs, const string & rhs )
const
                        { return strcasecmp( lhs.c_str( ), rhs.c_str( ) ) < 0;
}
};
```

By turning functions into objects this way, we can essentially treat them as first-class citizens. Of course this is a rough example of it's implementation, but more refined implementations have been made and are part of STL libraries such as std::functional.

Templates can also be used to make classes that will use a generic typename to define it's data and function members and allow us to create objects that act differently depending on which types of data they are supposed to be operating with.

It's important to note, when using class templates, that the separation of interface and implementation has been historically problematic, which motivates many people to implement them as a single unit.

## Matrices

The last topic of chapter 1 is matrices. The STL does not have a matrix class on it's own. Therefore separate libraries or a self-made class is necessary when working with multi-dimensional arrays. The book explains the creation of a simple matrix using an std::vector that holds std::vectors of a certain type. The following code is given:

```
template <typename Object>
class matrix
{
        public:
                matrix( int rows, int cols ) : array( rows )
                {
                        for( auto & thisRow : array )
                        thisRow.resize( cols );
                }

                matrix( vector<vector<Object>> v ) : array{ v }
                        { }
                matrix( vector<vector<Object>> && v ) : array{ std::move( v ) }
```

```
                        { }

                const vector<Object> & operator[]( int row ) const
                        { return array[ row ]; }
                vector<Object> & operator[]( int row )
                        { return array[ row ]; }

                int numrows( ) const
                        { return array.size( ); }
                int numcols( ) const
                        { return numrows( ) ? array[ 0 ].size( ) : 0; }
        private:
                vector<vector<Object>> array;
};
```

As seen, the matrix implementation in itself is simple, the first array represents the "rows" of the matrix, the second one represents the "columns", the first constructor for the class creates a vector of size rows, where "rows" is the number of vectors of type object inside, then the constructor resizes the each vector of type object to "cols" size. The other 2 constructors are the class's copy and move constructors.

The operator overloads are overwritten following the idea that if we have a matrix m, then calling m[i] should return a vector in the "i-row" in such a way that `m[i][j]` should return the object in position "j" inside the vector in position "i" inside of the matrix.

Lastly the methods numrows and numcols are defined and simply return the number of "rows" and "columns" the implementation has. A ternary operator is used to prevent accessing a non-existing array, by checking if numrows is different than 0 (which would make the expression false in such a way that it returns 0 in the end), if it is then it will simply return the size of the vector in the first position (as all vectors are the same size).

Lastly the book decides important to mention that the "big five" that is to say, the default destructor, copy and move constructors and operators, are all unnecessary to implement here as std::vector already takes care of most of the work for us.

## References

A. Kumar, “What is DSA in computer science?,” Jul. 07, 2023. https://www.linkedin.com/pulse/what-dsa-computer-science-anurodh-kumar

M. A. Weiss, *Data Structures & Algorithm Analysis in C++*. 2012. [Online]. Available: http://ci.nii.ac.jp/ncid/BA4445754X

H. J. De St Germain, “Programming - Recursion.” https://users.cs.utah.edu/~germain/PPS/Topics/recursion.html

B. Van Der Merwe, “Recursive FiBoNNaCi method explained - Launch School - medium,” *Medium*, Dec. 07, 2021. [Online]. Available: https://medium.com/launch-school/recursive-fibonnaci-method-explained-d82215c5498e

“How did OOP evolve from procedural programming?,” *www.linkedin.com*, Feb. 06, 2024. https://www.linkedin.com/advice/0/how-did-oop-evolve-from-procedural

“IBM documentation.” https://www.ibm.com/docs/en/i/7.3?topic=only-explicit-conversion-operators-c11

J. Waldo, “The Evolution of C++: language design in the marketplace of ideas,” *Choice Reviews Online*, vol. 31, no. 08, pp. 31–4406, Apr. 1994, doi: 10.5860/choice.31-4406.

“What is a first-class citizen in computer science?,” *What Is a First-class Citizen in Computer Science?* | *Douglas Moura*, Oct. 08, 2022. https://douglasmoura.dev/en-US/what-is-a-first-class-citizen-in-computer-science

J. Chen, “CS 225 | Stack and Heap memory.” https://courses.engr.illinois.edu/cs225/fa2022/resources/stack-heap/

TylerMSFT, “Smart pointers (Modern C++),” Microsoft Learn, Aug. 03, 2021. https://learn.microsoft.com/en-us/cpp/cpp/smart-pointers-modern-cpp?view=msvc-170

TylerMSFT, “Lvalue reference declarator: &,” Microsoft Learn, Nov. 23, 2021. https://learn.microsoft.com/en-us/cpp/cpp/lvalue-reference-declarator-amp?view=msvc-170

TylerMSFT, “Function objects in the C++ Standard Library,” Microsoft Learn, Aug. 03, 2021. https://learn.microsoft.com/en-us/cpp/standard-library/function-objects-in-the-stl?view=msvc-170