CODE, NOT JUST TO GET PAID BUT TO ENJOY

PRO SOFTC

PROGRAM IN C/C++ EFFORTLESSLY WITH DATA STRUCTURES

WRITTEN BY: RAO ZAEEM

2 DEC 2020

Table Of Contents

Chapter 1: Introduction

What Is SoftC

SoftC is a library for C/C++ built with 6500+ lines of code. It has a bunch of functions that you can use to make your work easier, and create complex data structures. With the power of nesting each other, you can do a lot, and store data effortlessly and access them easily. In this book I defined them all with complete details that I think will be necessary, Each data structure has its advantages and disadvantages which will be in their chapters. in this book, I assumed that you're familiar with C Basics, Pointers, Strings, Arrays, Dynamic Memory Allocation.

Idea Behind SoftC

You might be wondering why C? When I first started programming, C seems old and expired to me, but when I took CS50 Introduction to Computer Science on Edx by Harvard University, thanks to Prof David J. Malan that it was available for free, I learned about C. I read a book (which I borrow from my uncle) “Waite Group’s Turbo C Programming For The PC and Turbo C++” by Robert Lafore, and I just fell in love with it. I already knew Python, JavaScript, Dart, PHP but C impressed me with its power, performance, Memory Control, Pointers, freedom to allocate memory, and many more. After that when I start C++, to be honest, I didn’t like that (it didn’t mean that it’s bad at all, it has a large community and is very popular). it's Because I had to remember lots of things, understanding it's architecture is harder for me, C Seems pretty simple, faster, it only takes more effort. So why not make something that makes C Easy to program, first it was personal, then I decided to make this library free and share with everyone, so Without any further due let’s get right into it.

Setting Up

Go to the Github Repository (https://github.com/raozaeemshahid/SoftC) and clone that into your project directory, first make sure that these three libraries

<stdlib.h>, <stdio.h>, <limits.h> are included in your file, and then include "SoftC/SoftC.h". and you're ready to go. your initial file would look like this.

```c
#include <stdlib.h>
#include  <stdio.h>
#include <limits.h>
#include "SoftC/SoftC.h"

int main(void)
{
}
```

Chapter 2: String

Pointers of Char VS Array of Char Vs String of SoftC

when you learned C, you probably learned different methods to create a string, you can create a array of characters, declare it's size and assign it's characters one by one and put null character at the end. Or copy all characters from another pointer (precisely string) using strcpy or any other function. Or assign it a pointers (string written in double quotes). it's advantage is that you can change it's characters individually and it's disadvantages is that you cannot put string larger then you allocate size for it, secondly array is constant pointer, you can't change that. Another method you can use is declare the pointer of character, and assign it the pointer of a character (precisely string written in double quotes), it's reliable, you don't need to define size or maximum number of characters, it could be as long as number of bytes in memory, it's disadvantage is that that string is stored in read only memory, you can only access the characters by adding into pointer but cannot modify them, if you wanna change the string, you'll have to change the value of variable which is just a pointer. Mean while array is also the pointer to the first item, the only difference is that items of array is stored in a part of memory that can be modified by program, and that string (in quotes) that you assign to a pointer (of a char) is stored in read only memory that is managed by OS and cannot be modified by program.

This is enough recap of a string, if you couldn't understand the above paragraph so study it from other sources, this pointers and array concept is very important. now let's dive into how String of SoftC works, first I decided that I'll create a function that'll take a pointer of character (whether it's stored in read only or not, doesn't matter). It calculates it's size and then creates a array of that size and copy all characters, but there was a problem here, it's a Function right? so all of it's variable will vanished when its execution gets completed, so creating an array inside it will no longer work after that function ends, and it was giving error to declare static array, so I needed to create something that'll create string with the functionality of array (each character is accessible and modifiable), and no need to

worry the maximum size. so I used dynamic memory allocation, yes I used malloc function to allocate memory for array. I know it's insane. Programmer will have to free the string. but that's how it works, you can create assign string, then reassign it with any other string without worrying about length differences. I guess that's enough for now, let's learn about functions that you can use to make your work easier with Strings.

Functions of Strings

Functions for each data structure is written as a prototype in the source folder of each data structure, you can revise from there, although I'll explain all the functions in so many details, that you only need to read the name you will be able to recall complete functionality.

Characters

I combined the functions of character with functions of string, they're so much simple, let's take a quick overlook.

1) First we have CIsUpper, as by name, it is used to ask something, it takes character and return True (1) if that character is uppercase and false (0) if not.

2) Second we have CIsLower, as oppose to CIsUpper, it returns True (1) when character is lowercase and False (0) when character is uppercase.

3) Third we have CIsAlpha, it is also a question, if character is alphabet or not, it'll return True if char is in between a to z or A to Z, or return False if not.

4) Fourth we have CToUpper, it'll take a character, if that character is in uppercase it'll return that as it is, and if it's in lower case, it'll capitalize it and return, if it is not a alphabet, it'll do nothing and return that character.

5) Fifth we have CToLower, as oppose to CToUpper, it'll lowerize the character and return that, and do nothing if char is already in lowercase or not a alphabet.

That's all of the character function that is defined in SoftC, there could be many but I didn't do so, there are other libraries available to work on characters. I mainly focused on Data structures.

Strings

now let's dive into what I actually did in this data structure, I've discussed that I used malloc to create a string, there's a type for this named 'string', so if you wanna create a string, you don't need to write char *, just write string and it is same as character pointer, now let's see how you can work on strings with functions.

1) SAssign: First, we have the SAssign function, it'll take a string (doesn't matter if it is stored in read only memory or not) it'll allocate memory for that, copy all the characters and return you the address that allocated memory.

```
int main(void)
{
	string name;
	name = SAssign("Rao Zaeem Shahid");
	printf("\"%s\"\n", name);
	free(name)
}
```

Output:

"Rao Zaeem Shahid"

so that's how you create strings, call free at the program pass by the variable name and there will be no leaks, advantage of SAssign of using here is that you can modify characters individually, and then free it and reassign any other string without worrying about the size.

Code:

```
	string name;
	name = SAssign("Hello, Rao Zaeem Shahid");
	name[0] = 'H';
	name[1] = 'e';
	name[2] = 'y';
	name[3] = ',';
	name[4] = name[5] = ' ';
	printf("\"%s\"\n", name);
	free(name);
```

Output

```
"Hey,   Rao Zaeem Shahid"
```

After freeing the string, you can assign it with any other value, when you free the string, make sure that you don't use that string, because it's still pointing to the junk of memory but the content that was stored has been removed, so you may get garbage value, it could be anything that is stored from other programs or from OS at that time.

2) SReAssign: So Calling free again and again annoying right? so to skip this I defined another function named SReAssign, It is almost similar to SAssign, it just take on more argument, the first argument is the previous variable and other is string, it'll do nothing but call free to that junk of memory. so you can save a line of calling free and allocates new memory for the new string and return that.

Code

```
    string name = SAssign("Hey There,");
    printf("%s\n", name);
    name = SReAssign(name, "It's Rao Zaeem");
    printf("%s\n", name);
    name = SReAssign(name, "Isn't Programming Fun? I enjoy it a lot....");
    printf("%s\n", name);
    free(name);
```

Output:

```
Hey There,

It's Rao Zaeem

Isn't Programming Fun? I enjoy it a lot....
```

Run Valgrind to check any memory leaks, there should be no leaks. and that's how you can assign and reassign and string, it is as easy as character to pointer, with the functionality of Array of character, isn't that?

3) SLen: now we have the SLen function, as by name it'll take a string and return the length as long integer, it is alternative to strlen, you're probably familiar with that.

4) SSlice: In the list of functions, now we have SSlice, if you have a Python background, you may probably use string slicing there by string[starting_index: ending_index], well the same functionality is also available here, within the laws of C, let get started. it takes three arguments, first the string (it is not mandatory that string is assigned by SAssign function, it could be a pointer to read-only memory or array of character, doesn't matter), the second argument is an integer particularly the starting index from where you want to start slicing, and then the third argument is where you want to stop slicing, these indexes shouldn't be negative, or starting index shouldn't be greater than ending index, or it'll print an error message and do nothing. if any of the indexes is greater than the length of the string, then it'll be assumed as the last index. one more thing, as SAssign and SReAssign creates string by malloc, it'll also create a string by malloc, so store it in another variable, if you assign it back to the variable which you sliced, so the pointer to the old string will be lost you wouldn't be able to free that, there's another function to trim string that we will discuss later. back to SSlice, here's an example

Code:

```
string name = SAssign("Rao Zaeem Shahid");
string middleName = SSlice(name, 4, 9);
printf("\"%s\"\n", middleName);
free(name);
free(middleName);
```

Output:

```
"Zaeem"
```

5) SJoin: moving on, we have SJoin, as, by name, it joins two strings, pass it the string (array or pointer whatever you have), it'll create a new string by malloc, join both of the strings and create a new one, and returns you, again that is created by

malloc, so don't assign it back to variable or it'll leak memory, store it in a separate variable and then free, at least in the end. here's a quick example

Code:

```
string name = SAssign("Rao Zaeem ");
string lastName = "Shahid";
string fullName = SJoin(name, lastName);
printf("\"%s\"\n", fullName);
free(name);
free(fullName);
```

Output:

```
"Rao Zaeem Shahid"
```

as you can see, the name was created by SAssign so it needs to be free at the end, and the last name is a pointer to read-only memory so it'll be free by the program when it completes it's execution we don't need to do that (i did it just to show you that how a string is defined doesn't matter here, this function only need to read it). SJoin creates a new string so store it in separate variable and it also need to be free at least at the end of program. you can see there's space after the Zaeem in name, that is because that SJoin just join them, so if there's no space then both of these words will be merged together without space, and that looks weird.

7) SReverse: as by name, it'll reverse the string, it takes one argument, just give it the string, and now it's mandatory that string should array of characters or is created by SAssign, other wise it'll create segmentation fault. This function will modify the original string, so if you want to save the older version of string, then store that string in some other variable, but, you cannot copy string by assigning to other variable since it's just a pointer, then both of the variable will be pointing to the same string, you can use SAssign, pass it the older string and it'll also work to duplicate your string, then call SReverse function to string, here's a example

Code:

```
string name = SAssign("Rao Zaeem");
string copyName = SAssign(name);
SReverse(name);
printf("Previous String: %s\n", copyName);
```

```
printf("Reversed String: %s\n", name);
free(name);
free(copyName);
```

Output:

```
Previous String: Rao Zaeem

Reversed String: meeaZ oaR
```

8) STrim: you already used SSlice function, it's same, the only difference is that SSlice creates a new string (which needs to be free) and STrim don't create new string, instead it just modify the existing one, here's an example

Code:

```
string name = SAssign("Rao Zaeem");
STrim(name, 4, 9);
printf("\"%s\"\n", name);
free(name);
```

Output:

```
"Zaeem"
```

9) SToUpper: As by name, it is used to modify the string and capitalize all of it's character, if a character is already is uppercase then it'll do nothing, if char is in lowercase then it'll capitalize that, and if the character is not alphabet, it'll do nothing.

Code:

```
string name = SAssign("Rao Zaeem");
SToUpper(name);
printf("\"%s\"\n", name);
free(name);
```

Output:

```
"RAO ZAEEM"
```

10) SToLower: as oppose to SToUpper, it'll modify the existing string and lowerize all it's character,

Code:

```
string name = SAssign("RaO ZAEeM");
```

```
    SToLower(name);
    printf("\"%s\"\n", name);
    free(name);
```

Output:

```
"rao zaeem"
```

11) STitle: Same as title method of Python, it'll capitalize the first character of the first character of every word (more precisely that is separated by space). and lowerize every other character then the first character.

Code:

```
string name = SAssign("raO ZAEeM");
STitle(name);
printf("\"%s\"\n", name);
free(name);
```

Output:

```
"Rao Zaeem"
```

12) SCopy: next we have SCopy, it is used to copy the characters of one string to another string, but the destination (to where you're copying characters) should have enough memory allocated store that string, for instance, if you're copying string of 5 characters excluding null terminator, so the destination array should have at least memory for 6 char so that it could store that string, This function was designed to use by other function of other data structure that we'll see later in this book, I was not intended to explain it too, because SAssign and SReAssign is enough, but If you're familiar with strcpy function, then this is the alternative of strcpy. for demonstration, here's a quick review

Code:

```
    char name[6];
    SCopy(name, "Zaeem");
    printf("%s\n", name);
```

Output:

```
Zaeem
```

If you are wondering why would someone do that? like you could directly use that string which is second argument of SCopy function, right? here's the key point, that string which is the second argument of SCopy, is stored in read only memory, it's character is accessible but can't modifiable. The propose of copying it to array is to store in place where we can access that and modify as well. since SAssign and SReAssign made this easy for you, so you don't need to use SCopy in your code, just use SAssign and SReAssign and free them at last and don't worry. hope you are not angry at me because I explain everything and now you realize you don't need to use it.

13) SIsUpper: As by name, it takes one argument that is string for sure and return True (1) if all the alphabets are in uppercase and return false if any of alphabet is in lowercase, and ignore non alphabets.

14) SIsLower: As oppose to SIsUpper, it'll return True (1) when all the alphabets in in lowercase and false when any alphabet is in uppercase.

15) SIsSame: It takes two arguments, both of them are strings, if they're same it'll return True, if not then it'll return False.

16) SFindChar: As by name, it is used to find a character in a string, first pass it string and then character, it'll return the index of that character in that string if it finds. if it couldn't then it'll return -1.

17) SFindCharFrom: If you want to start searching from specific index, in case if the character is repeated more than once, and you want to find the second one, use this function, pass it string, character, and then index from where you want to start, for instance if index of first character is x, pass x+1 to find the next.

18) SFindStr: As same to SFindChar, it finds String inside String, first pass it the string in which you want to find, and then string which you want to find, it'll look for that string, if that string exist somewhere, then it'll return the index where that string starts inside that string. if it couldn't find, it'll return -1.

19) SCountChar: As by name, it counts the Character repeated in a string, it takes string and a character and return you the number of repetition of that character inside string.

If you read that chapter completely and understand everything until yet in this book, then i wanna Congratulate you.. you just mastered one Data Structure. let's move on to others

Chapter 3: Vector

Vector is data Structure used to store more than one value, specially when you don't know how many items need be stored, like if you get some data in runtime and need to be store somewhere, since you cannot declare variable in runtime, the idea of Data Structures comes in. If you've learned C++, you're probably familiar with Vector, if your background is Python, think it as List, but all the items of Vector should have same type, for instance you can't append int in Vector of characters, but store as many items as you want (not more than memory). there's another Data Structure named "List" (we'll see later in this book) in which you can store item of different data types. but each item of List takes 48 Bytes, while item of Vector take around 16 Bytes(but Only Long Double Item takes 32 bytes), such a huge difference. Now let's take a quick look at Vector Internals, it's nothing but Linked List. if you're familiar about it then you have idea how it works, another things you should know about is that vector is just a pointer like string. if you want to give it to some function by reference, you don't need to add ampersand with it, it is already a reference just pass variable name. without any further dues, Let's Get Started

Initializing Vector

There are 12 data types for vector, all of them have one starting point (structure) that's why all of them have one data type 'vector'. for now, we will not go any details. if you wanna create vector of characters, just call InitVectorOfChar. For short, call InitVectorOfShort. For int, call InitVectorOfInt. For long, use InitVectorOfLong. For long long, use InitVectorOfLLong. For float, use InitVectorOfFloat. For double, use InitVectorOfDouble. For long double, use InitVectorOfLDouble. For string, use InitVectorOfString. To create vector of vector (nested Vectors) use InitVectorOfVector. To create vector of List, use InitVectorOfList. To create vector of Dict (which we'll also see in upcoming chapters), use InitVectorOfDict. and whatever they return, assign that to a vector variable, now your vector is ready, you can store and access data from it. At the end of program or when you don't need to use the vector, delete the vector by

VDelete function, and it'll free of the memory, you don't need to worry anything about it. here's a quick example

Code:

```
vector myChars = InitVectorOfChar();
VDelete(myChars);
```

it outputs nothing, but your vector has been created and deleted in the background, you can work with vector in between these two lines, you can Insert Items, access them, find them, replace them, remove them, and many more. VDelete is same function for all kinds of vectors, but there are different function for initialization, if you wanna create vector for another type of items (like int or float), use Initialization function that is designed for that type. now Let's move on to working with them

Append Items

You can Append Items into vector by VAppend[typeof vector], if your vector is initialized for Char (mean you use InitVectorOfChar to create vector), use VAppendChar, to append character in the vector, it takes two argument first is your vector, and then the item you want to append. Similarly there are other functions for other types of vector like VAppendShort, VAppendInt, VAppendLong, VAppendLLong, VAppendFloat, VAppendDouble, VAppendLDouble, VAppendString, VAppendVector, VAppendList, VAppendDict, use them according to your vector type. It'll Append Item at the last of vector, to print the contents of the vectors, use Function logVector, give it vector, and it'll print your vector in a manner, and you can see all of the items of your vector, of course you can access items individually but we'll look at that later. now let's take a look at example

Code:

```
vector myChars = InitVectorOfChar();
VAppendChar(myChars, 'a');
VAppendChar(myChars, 'b');
VAppendChar(myChars, 'd');
logVector(myChars);
```

```
VDelete(myChars);
```

Output:

```
[0] a

[1] b

[2] d
```

in Left, the number in the square bracket is the index of the item in that vector. a totally new shape of C, right? it's syntax got completely changed. just stick with it and I'm sure it'll be lot of fun.

Accessing Item from Vector

if you wanna access item in a vector, well you can't use square brackets like you do with Array or with lists in Python, but there are some functions which you can use to make your job done, that is VGet[typeof vector], if your vector is of type, use VGetChar, if your vector is of type Short, use VGetShort, and same with all 12 types of vectors (VGetChar, VGetShort, VGetInt, VGetLong, VGetLLong, VGetFloat, VGetDouble, VGetLDouble, VGetString, VGetVector, VGetList, VGetDict). it takes two argument, first is your vector, and other is index, index as usual starts from 0.

Example:

```
vector myChars = InitVectorOfChar();
VAppendChar(myChars, 'a');
VAppendChar(myChars, 'b');

printf("First Item: %c\n", VGetChar(myChars, 0));
printf("Second Item: %c\n", VGetChar(myChars, 1));

VDelete(myChars);
```

Output:

```
First Item: a

Second Item: b
```

Inserting Character at desired position

VAppend[typeof vector] will Append vector at the last of vector, you've seen that, right? in case of inserting vector at desired position, not in the end but somewhere in between, you can use VInsert[typeof vector], it takes three argument, first the vector, then the item (of same type you created vector) and then the index on which you want to insert item. isn't that simple? here's a example.

Code:

```
vector myChars = InitVectorOfChar();
VAppendChar(myChars, 'a');
VAppendChar(myChars, 'b');
VAppendChar(myChars, 'd');

printf("Before Inserting: \n");
logVector(myChars);

VInsertChar(myChars, 'c', 2);

printf("After Inserting: \n");
logVector(myChars);

VDelete(myChars);
```

Output:

```
Before Inserting:
[0] a
[1] b
[2] d
After Inserting:
[0] a
[1] b
[2] c
[3] d
```

Length of a Vector

To get the length of vector, meaning getting the number of items in a vector, you can use function VLen, give it the vector and it'll return you long integer of it's length, you can use that to get the size of string but internally it's doing nothing, length of vector is already stored in your vector, vector is a structure, you can also access the size using myVector->size, and it'll also give you the size. take a look

Code:

```
vector myChars = InitVectorOfChar();
VAppendChar(myChars, 'a');
VAppendChar(myChars, 'b');
VAppendChar(myChars, 'd');
VInsertChar(myChars, 'c', 2);

printf("From Function: %ld\n", VLen(myChars));
printf("Direct Access: %ld\n", myChars->size);

VDelete(myChars);
```

Output:

```
From Function: 4

Direct Access: 4
```

Removing The Item

when you are working with vector, you probably gonna get into a situation when you need to remove a item from a vector for whatever reason, you can use VRemoveItem to do that, give it the vector and the index of item which you want to remove and it'll do you job. This is the only function for all kinds of vectors, you don't need to call different for vector of characters and no need to call different function for the vector of integers. Have fun.

Code:

```
vector myChars = InitVectorOfChar();
VAppendChar(myChars, 'a');
VAppendChar(myChars, 'b');
VAppendChar(myChars, 'd');
VInsertChar(myChars, 'c', 2);
VRemoveItem(myChars, 1);
```

```
logVector(myChars);

VDelete(myChars);
```

Output:

```
[0] a

[1] c

[2] d
```

see, it has remove the item on 1 index (second item) which was 'b', let's continue.

Popping the Last Item

In order to remove the last item from the vector, you can use VPop, give it the vector and it'll remove the last item of your vector, off course you can also do it by VRemoveItem(myVector, myVector->size - 1), both will work the same, but pop looks better to do that. and for Internals of Vector, Pop is a good choice.

Code:

```
vector myChars = InitVectorOfChar();
VAppendChar(myChars, 'a');
VAppendChar(myChars, 'b');
VAppendChar(myChars, 'c');
VAppendChar(myChars, 'd');

VPop(myChars);
logVector(myChars);

VDelete(myChars);
```

Output:

```
[0] a

[1] b

[2] c
```

Isn't that simple? interesting? and fun?

Comparing Two Vectors

To compare two Vector, and to see are they equal or not? you can use function VIsSame, give it two vector, it'll compare them. And return 1 (True) if both vector are same, and 0 (False) if not. Remember, it checks the fundamental types (Char, Short, Int, Long, Long Long, Float, Double, Long Double, String) if you have nested structure, for instance a vector has nested vector, then it'll go inside those vector items and check their Items. Example,

Code:

```
vector myChars = InitVectorOfChar();
VAppendChar(myChars, 'a');
VAppendChar(myChars, 'b');
VAppendChar(myChars, 'c');
VAppendChar(myChars, 'd');
VPop(myChars);

vector mySecondChar = InitVectorOfChar();
VAppendChar(mySecondChar, 'a');
VAppendChar(mySecondChar, 'b');
VAppendChar(mySecondChar, 'c');

if (VIsSame(myChars, mySecondChar))
{
	printf("Same\n");
}
else
{
	printf("Not Same\n");
}
VDelete(myChars);
VDelete(mySecondChar);
```

Output:

```
Same
```

if there is some other items in second vector, then it would print "Not Same". however I hope you're getting a idea how to use it. let's move on.

Copying Items from one vector to Another Vector

let's say you have two vectors, and you wanna insert all items of one vector into another, well you can loop for the size of vector one, Get it's all item and append to vector two, but that algorithm is little slow that I'm gonna show you, just use a function VCopy, first give it the destination vector, and then source vector, it'll take all items from source vector and append into destination vector.. so let's see an example.

Code:

```
vector myChars = InitVectorOfChar();
VAppendChar(myChars, 'a');
VAppendChar(myChars, 'b');
VAppendChar(myChars, 'c');
VAppendChar(myChars, 'd');

vector mySecondChar = InitVectorOfChar();
VCopy(mySecondChar, myChars);
VAppendChar(mySecondChar, 'e');

logVector(mySecondChar);
VDelete(myChars);
VDelete(mySecondChar);
```

Output:

```
[0] a
[1] b
[2] c
[3] d
[4] e
```

myChars remain the same, it still has 4 items, but we've created another vector, copy all items from myChars, and append one more item. now mySecondChar has 5 items.

Creating Copy of a Vector

if you wanna create a copy of a vector, well you could do by initializing a new vector, then copying all the items by VCopy functions, well that's corrrect, but

the same thing can be do in one line using VReCreate, it just do the same 2 steps as discussed above, but it looks pretty I guess. you can use VReCreate for creating a copy of any type of vector, it'll manage which type of vector to initialize and copying items. let me call a example for you.. an Example Please.

Code:

```
vector myChars = InitVectorOfChar();
VAppendChar(myChars, 'a');
VAppendChar(myChars, 'b');
VAppendChar(myChars, 'c');
VAppendChar(myChars, 'd');

vector mySecondChar = VReCreate(myChars);
logVector(mySecondChar);

VDelete(myChars);
VDelete(mySecondChar);
```

Output:

```
[0] a
[1] b
[2] c
[3] d
```

Replacing Item In a Vector

If there's need to replace an item in vector, well you can also it by Removing that item, and inserting new one into that place, that's correct, while the same thing can be done in one line using VReplace[typeof vector] , first it take vector, then the item that's gonna take that place, and then the index that's gonna get replaced.

Example:

```
vector myChars = InitVectorOfChar();

VAppendChar(myChars, 'a');
VAppendChar(myChars, 'b');
VAppendChar(myChars, 'f');
VAppendChar(myChars, 'd');
```

```
VReplaceChar(myChars, 'c', 2);

logVector(myChars);
VDelete(myChars);
```

Output:

```
[0] a
[1] b
[2] c
[3] d
```

for sure there are other function for different types of vector, you can check them all, but I'm lazy to check them, in fact I haven't even test them, I only tested the first one, if that is correct and the rest should be correct. because the code is copied into other function from first function when it gets success, just few minor changes. if you got any bugs. then I don't know what to do... avoid that function and don't used that. just kidding guys, you can create a issue on GitHub, or if you understand the code (fortunately or unfortunately) then make a pull request, or at least just send me an email raozaeem2018@gmail.com, help me to find the bug, your contribution will be appreciated. just describe how you got into trouble, and what goes wrong as much detail as you can. well, hope you'll do that if you find one.

Find Item in Vector

In order to find something in vector, you can use VFind[typeof vector], first give it the vector and then the item that you wanna find, if it finds that item inside the vector it'll return it's index, otherwise it'll return -1.

Find Item From a Specific Position in a Vector

If the item is repeated more than once in a vector, and you wanna locate the second one or further, then you can use VFindCharFrom or according to your vector type. Pass it vector, item, and the position from where you want to start searching. if it finds something, it'll return you the index, or it'll return -1.

Nested Structures

You probably know that the vector and string itself is nothing but a pointer, but their copy can be created by SAssign or VReCreate, remember? so whenever you insert them inside a vector, you're not inserting their reference, instead the append and insert function will create another copy of them then insert them. so you gotta delete the original string and vector by their deleting method, while that one copy inside the vector will be deleted when you remove item or delete the entire vector. On the other hand, when you get them by VGetVector or VGetString, you're not getting their copies instead you're getting their reference, if you change something in it, the item inside the vector will also be changed. These items that you get by VGetVector or VGetString are actually own by vector so you don't need to delete them either, vector will manage and delete them (when you delete the vector). Read this Nested Structures again, this is very important concept.

Memory it Takes

when you initialize a vector, it's gonna allocate 40 Bytes at heap, and keep incrementing as you insert more items and decrement as you remove items, each item Takes 16 Bytes except the Long Double, which takes 32 Bytes. on average in 1 MB of memory, 60,000+ items or 30,000+ long double can be stored. Note: This was calculated on my machine and purpose was only to give you a idea.

If you read everything until this book in this book and understand that, then I wanna Congratulate you that you have mastered one more Data Structure.

Chapter 4: Array

Array Of C VS Array Of SoftC

You're probably familiar with Arrays in C, you can create array of any data type, first tell it the Size of array and it allocate memory for that. the variable is a pointer to the first item, all item are stored in memory in row, they are accessible and modifiable by adding into pointer. I Hope you already know that.

Let's come to Array of SoftC, It is same as normal array, the difference is that it puts few more items in the array to keep track of the end, it puts three items at the end of array so that the program can look for those values and determine where the array has ended, but what if the same 3 bytes in row get inserted somewhere in array, we've ran into a problem, because that program will think this is the ending of the array and it'll not see anything farther then that. To solve this issue, we put 6 items instead of 3, so whenever the array is declared, 3 items will stay before the items of array, and 3 will always stay after the items, first three are exact copy of last three, so now the program can easily determine when the array has ended by looking for those indicators, it just needs to go into negative index to see the indicators, first indicator is array[-3], second indicator is array[-2], third indicator is array[-1], and then items starts from array[0], and keep going until it finds those indicator, now we don't need to hard code our program that these are indicators, go and look for them

Now, the indicator are dynamic, we can loop it, keep adding in pointer, once it finds them, break the loop. In case, if they get repeated somewhere in array, we can modify indicators without any hesitation, we know one of their copy is stored just behind the array, we only need subtract pointer (go onto negative index) to see what's their current value and then look for them in array, since we can modify them, they're programmed in a way that they can take every possible combination, and there could be more millions ways that they can arrange themselves, and inserting each combination in a array isn't that easy, if someone did that, by using GB's of their memory, then they can't go any further, program will stop inserting

more items and print that array is full. Those of you guys wondering what is happening, don't worry that whole thing is just for the information, you don't need to put those characters and modify them and check them, There are functions to do that and makes your work lot easier.

Now let's come to questions, how it gonna create array? and how it'll insert items? well remember when you declare array, you'll have to tell it the size, then you cannot put more items, right? but recall how we created Strings? we used malloc. YEAH..... we again used that same (useful or weird) technique to create array, now every time when item has to insert, it'll first create new array with (oldSize+1) size with malloc, copy the indicator, then all the items including new item, then last indicators, then check if the indicators don't get repeated somewhere, if yes then modify the indicators. if you have question if indicators has changed.. then how it'll check which are the real indicators and which one to change? right? when the insert function called, it calculated its size (before inserting new item). now think, oldSize is the index of old Indicators, Got it? now when one more item has inserted so the index of new indicator should start from (oldSize+1). but how it gonna check if indicators got repeated or not? it'll calculate new_size, and new_size should be equal to (oldSize+1) but if it is not, it means that indicators are repeated and has to changed, it'll modify the first copy of indicators, now (oldSize+1) is index of first indicator, (oldSize+2) is second indicator, and so on, it'll change them and keep changing until the result of new_size don't get equal to (oldSize+1), if indicators has changed their value in every possible way and that is exist in array, then it'll print "array is full" and free the new array and return the old array (that didn't have new item, so that will work fine) isn't that a good idea?

When it comes to string, its nothing but a pointer, so instead of creating it's 6 items as indicators, NULL pointer is enough, it starts pointers to characters from zero index, once it find NULL it means that array has ended, and the functions will not allow NULL to be inserted.

If this process gets completed, it'll free the old array and return the (newArray + 3) because we want items to start from array[0], but indicators were

inserted first and they get skip, and that's how they got negative index . Congrats it just created a array after a long industrial process, hope you understand how it all works, This architecture took me more than 30 hours trying different techniques, solving segmentation fault and lot more, I explain it just to give you idea how it works so you will be able to easily work with them. I think that's enough intro to arrays. let's continue.

Array VS Vector

When we have Vector, why would someone use Array? you have studied them both in details, Vector is a linked list, and Array is row items in memory, Appending into Vector is lot more faster (because it has stored the pointer of last node) while inserting at desired position is little bit slow (because it has to follow the pointers from the starting point to that node). Accessing items is also little bit slow because it has to follow the pointers from node to node for that index. lastly Vector takes too much memory, each item takes 16 Bytes (Long double take 32 bytes). let's talk about Array, all items of array is in row in memory, so we only need to add into pointer and we get our item, it's ultimately fast.. but inserting method is little bad, it take a little long but not too much at small scale. empty array takes memory of 6 items, if you have array of integers, then empty array takes 24 Bytes (if one int takes 4 Bytes) and each item will take 4 Bytes, it takes lot lesser memory than Vector, and accessing is lot more faster, but inserting isn't that sufficient. it's up to you which one you choose and which meets your requirement.

Working With Arrays

you've learned how arrays work, how they created, their pros and cons, now let's move to how you can work with them easily with functions in sufficient way.

Initializing Array

There are 9 types (char, short, int, long, long long, float, double, long double, string) that you can use to create array, all of them have different data types (charArray, shortArray, intArray, longArray, llongArray, floatArray, doubleArray, ldoubleArray, stringArray) and they have different function for initialization

(initCharArr, initShortArr, initIntArr, initLongArr, initLLongArr, initFloatArr, initDoubleArr, initLDoubleArr, initStringArr). you can't create Array of other data structure like Vector, List or Dict, I could add that functionality because they're just pointers like Strings so I didn't do so, you can use Vector to make them work. back to array let's see an example

Code:

```
charArray myChars = initCharArr();
DeleteChars(myChars);
```

and that's how you can create initialize and delete array, your array has been created and deleted in the background, You can work with your Array between these two lines, I could also delete that array by free(myChars-3) but this function looks pretty, there are other functions (DeleteChars, DeleteShorts, DeleteInts, DeleteLongs, DeleteLLongs, DeleteFloats, DeleteDoubles, DeleteLDoubles, DeleteStrings) for deleting array of other types of array. now let's move to appending items.

Appending Items

For each type of Array, there's another function (AppendChar, AppendShort, AppendInt, AppendLong, AppendLLong, AppendFloat, AppendDouble, AppendLDouble, AppendString), first they takes array as argument and then the item which is gonna append, and it'll do it's job. I will not go into full details, but you have gave an idea how it all works. Now whatever it return, assign it back to your array variable. and your array now has that item. you can print their contents using log functions (logChars, logShort, logInts, logLongs, logLLongs, logFloats, logDoubles, logLDoubles, logStrings) and they will print all items of array in a manner. or you can also use square brackets with index to access the item, if you go farther then your last item you'll find indicator, But use square bracket only to read the data, don't modify anything anywhere or you'll harm your own program. and it's up to you, and don't pass array to this function which was not created by initialization function, if you do so then it'll not able to determine the end, it'll look for the indicators that are negative index (which is garbage), and keep moving

until it's not found stop indicators. and then it'll create new array of all that garbage value, maybe it read your entire Memory and it couldn't find or the array gets so big that it cause heap overflow, that's really terrible. be careful. back to appending items, let's see by example how you can do it.

Code:

```
charArray myChars = initCharArr();
myChars = AppendChar(myChars, 'a');
myChars = AppendChar(myChars, 'b');
logChars(myChars);
DeleteChars(myChars);
```

Output:

```
[0] a
[1] b
```

When you call function, don't forget to assign it back to your variable, cause it created a new array and old one is deleted. again you can use square brackets to read the data but don't modify items of array using square brackets, use functions (which we'll discuss later) that are specially designed to do that. Another thing, when you insert append or insert a string in a array, you're inserting the reference. so if you change something in original string, that one string inside the array will also be changed.

Inserting Items at desired Position

In situation when you needed to insert the item at position not in the end somewhere in between the array, you can use Insert function (InsertChar, InsertShort, InsertInt, InsertLong, InsertLLong, InsertFloat, InsertDouble, InsertLDouble, InsertString) function to do that, first it takes array, then item, and index on which the item will be inserted. and whatever it return assign it back to your array variable and your job is done. here's example.

Code:

```
charArray myChars = initCharArr();
myChars = AppendChar(myChars, 'a');
myChars = AppendChar(myChars, 'c');
```

```
myChars = InsertChar(myChars, 'b', 1);

logChars(myChars);

DeleteChars(myChars);
```

Output:

```
[0] a

[1] b

[2] c
```

And that's how you insert item at any position in the Array, keep in mind how array works and don't do anything deliberately that will harm your computer.

Getting The Size Of Array

To get the size of Array, or get the number of items in array, you can use Size Function (SizeChars, SizeShorts, SizeInts, SizeLongs, SizeLLongs, SizeFloats, SizeDoubles, SizeLDoubles, SizeStrings), use one of them that matches you array type and it'll return you the long integer Size of array. take a look

Code:

```
charArray myChars = initCharArr();

myChars = AppendChar(myChars, 'a');
myChars = AppendChar(myChars, 'b');
myChars = AppendChar(myChars, 'c');

printf("Size is %ld\n", SizeChars(myChars));
DeleteChars(myChars);
```

Output:

```
Size is 3
```

Removing Item from Array

In order to remove at item from an array, you can use function Remove function (RemoveChar, RemoveShort, RemoveInt, RemoveLong, RemoveLLong, RemoveFloat, RemoveDouble, RemoveLDouble, RemoveString), it takes array

and index of the and it'll remove that item, it creates and returns new array so assign it back to your array variable. take a look at Example

Code:

```
charArray myChars = initCharArr();
myChars = AppendChar(myChars, 'a');
myChars = AppendChar(myChars, 'b');
myChars = AppendChar(myChars, 'b');
myChars = RemoveChar(myChars, 1);
logChars(myChars);
DeleteChars(myChars);
```

Output:

```
[0] a
[1] b
```

How's it going Guys? I don't know what are you feeling about it, maybe you liked it maybe you don't, or maybe you're super excited about it.. if yes then feel free to send me your feedback or any error at raozaeem2018@gmail.com, well let's continue our journey in mastering these Data Structures, believe me guys I'm also learning a lot while writing about these functions and architecture which was created by me. but it was so much that I forgot the usage of arrays. Just now, when I was writing the example, I call Append function and forgot to assign it back myChars, and myChars still pointing toward that old array that has garbage value, so Delete function creates error and I had no idea what went wrong.. apparently I found what went wrong within half an hour. but it annoyed me.

Popping Last Item In Array

When you're in need to remove the last item of array, again you can also do that by remove function, give it the of (size - 1), and that'll work the same, while the same thing you can also do by pop function (PopChar, PopShort, PopInt, PopLong, PopLLong, PopFloat, PopDouble, PopLDouble, PopString), just pass it the array and it'll remove the last item, and then it returns pointer to new array, assign that back to your array variable, Example,

Code:

```
charArray myChars = initCharArr();
myChars = AppendChar(myChars, 'a');
myChars = AppendChar(myChars, 'b');
myChars = AppendChar(myChars, 'c');
myChars = PopChar(myChars);

logChars(myChars);
DeleteChars(myChars);
```

Output:

```
[0] a
[1] b
```

Replacing The Item in Array (Changing Values of Items)

now let's come to method to modify an item of array, I warned you not to change value of your items directly by using square brackets because it might cause interruption the indicators, well you can do that by using Replace function (ReplaceChar, ReplaceShort, ReplaceInt, ReplaceLong, ReplaceLLong, ReplaceFloat, ReplaceDouble, ReplaceLDouble, ReplaceString), use anyone them that matches your array type, it takes three arguments, one is array, then new item that'll take place (or become the new value) and the index that's gonna get new value. here's example,

Code:

```
charArray myChars = initCharArr();
myChars = AppendChar(myChars, 'a');
myChars = AppendChar(myChars, 'd');
myChars = AppendChar(myChars, 'c');
myChars = ReplaceChar(myChars, 'b', 1);
logChars(myChars);
DeleteChars(myChars);
```

Output:

```
[0] a
[1] b
[2] c
```

Finding Items in Array

In Situation when program don't know the index of item but know the item itself, then Find function (FindChar, FindShort, FindInt, FindLong, FindLLong, FindFloat, FindDouble, FindLDouble, FindString) can give a index of that item if it founds in array, if it couldn't find, it'll return -1. here's example.

Code:

```
charArray myChars = initCharArr();
myChars = AppendChar(myChars, 'a');
myChars = AppendChar(myChars, 'b');
myChars = AppendChar(myChars, 'c');

logChars(myChars);
printf("b is at %ld\n", FindChar(myChars, 'b'));
DeleteChars(myChars);
```

Output:

```
[0] a

[1] b

[2] c

b is at 1
```

Start Finding Item From a Specific Position

Find function will find item from first position and will return the index of first match it found, but if the same item is repeated more than one time, since you cannot find the index of second item using Find function, well I'm here, you can use FindFrom functions (FindCharFrom, FindShortFrom, FindIntFrom, FindLongFrom, FindLLongFrom, FindFloatFrom, FindDoubleFrom, FindLDoubleFrom, FindStringFrom) to specify from where to start finding that, like if x is the index of first index, you can give it x+1 to find the index of second item. You have used that twice in Vector and String but I explain it again and I just don't why, anyway let's take an example.

Code:

```
charArray myChars = initCharArr();
myChars = AppendChar(myChars, 'a');
myChars = AppendChar(myChars, 'b');
```

```
    myChars = AppendChar(myChars, 'c');
    myChars = AppendChar(myChars, 'b');

    logChars(myChars);

    long Firstb = FindChar(myChars,'b');
    printf("Second b is at: ");
    printf("%ld\n", FindCharFrom(myChars, 'b', Firstb+1));
    DeleteChars(myChars);
```

Output:

```
[0] a

[1] b

[2] c

[3] b

Second b is at: 3
```

Joining Two Arrays

In Order to merge two array, well you gotta do that by looping over one array and appending all items into another array, because I had a function to do that but I removed that just now.

Comparing Two Array

You can Compare Array by looping and comparing both of their items one by one, but don't worry, you do it one line with isSame functions (isSameChars, isSameShorts, isSameInts, isSameLongs, isSameLLongs, isSameFloats, isSameDoubles, isSameLDoubles, isSameStrings), Give it two vectors, it'll return True (1) when both are same, or False (0) when they're not equal.

That's the end of Arrays, if you have read all the chapters until yet, then I wanna congratulate you, You've mastered one more Data Structure.

Chapter 5: List

List is Data Structure used to store more than one item, as compare to Vector, it's almost same but it can store data of different types in same List, while all the items of vector has to be same. different data types means you'll have to clarify which type of item you're inserting, and which type you're fetching. It makes everything clear, it's simple once you understand it. each item of List takes 48 Bytes, while each item of vector takes 16 Bytes or 32 hardly. so your requirements meet with Vector, then don't use List, save the memory and simply use Vector. Another thing List variable is a pointer like Vector or String or Array, so whenever you need to pass it to function, don't use ampersand. so without any further due, Let's see how to use them and how they work.

Initializing List

There's only one type list and only one function to initialize, which is initList, and your list is ready, sounds easy? and you can delete the list by LDelete function, let's take a look at an example.

Code:

```
list myList = initList();
LDelete(myList);
```

It outputs nothing, your list has been created and deleted, you can work with lists before deleting it, let's see what we can do with it.

Appending Items

There are different function to append items of different types, you can append any type of item in any list, you can append character by LAppendChar, it first takes List, and then item (which is character here, so pass it character) and it'll append that, similarly you can append short by LAppendShort, append int by LAppendInt, append long by LAppendLong, append long long by LAppendLLong, append float by LAppendFloat, append double by

LAppendDouble, append long double by LAppendLDouble, append string by LAppendString, append Vector by LAppendVector, Append List by LAppendList, append Dictionary by LAppendDict, you can create nested structure, nest them as much you want and make them more complex. there's no limit how far can you go. You can use logList to print all the items. let's see by example.

Code:

```
list myList = initList();
LAppendChar(myList, 'a');
LAppendInt(myList, 16);
LAppendFloat(myList, 5.8);
LAppendString(myList, "How's Going Guys?");

logList(myList);
LDelete(myList);
```

Output:

```
[0] <char>      a
[1] <int>       16
[2] <float>     5.800000
[3] <string>    How's Going Guys?
```

Similarly, you can Append more items, Even List too, Create List append some items into that list then append that list to another list, create a vector, append that vector to list, Experiment with them and that'll be lot of fun.

Getting the Type of Item in List

Well in order to access the item in a list, we first need to know what kind of data is stored at that index, because you can't get each type of item from just one function, it depends on item. there's function LGetType, that takes list and index and return the type of that item, it return in String, so you can compare it output by "char", "short", "int", "long", "long long", "float","double","long double", "string", "vector", "list", "dict" and then get your work done, I know it's lengthy but there was no other way. look at this example

Code:

```
list myList = initList();
LAppendChar(myList, 'a');
LAppendInt(myList, 16);
LAppendFloat(myList, 5.8);
LAppendString(myList, "How's Going Guys?");

string type = LGetType(myList, 1);
if  (SIsSame(type, "char"))
        printf("A char is stored at index 1\n");
else if  (SIsSame(type, "int"))
        printf("A Int is stored at index 1\n");
else
        printf("I didn't check all types because I'm lazy.\n");
LDelete(myList);
```

Output:

```
A Int is stored at index 1
```

well that's how you can determine what's store at that index, and now you can access the item at that index by a function which is our next topic.

Getting the Item from a vector

when you know what type item is stored at that location, you can get that by function for that type. For instance, if a character is stored at index 1, then you can get that item by LGetChar, similarly LGetShort to get short, LGetInt for int, LGetLong for longs, LGetLLong for Long Long, LGetFloat, LGetDouble, LGetLDouble, LGetString, LGetVector, LGetList, LGetDict. if you call wrong function and there's another type of item at that index, then it'll print that "we found %s item at %d index, you can't get that by this function". let's see a quick example.

Code:

```
list myList = initList();
LAppendChar(myList, 'a');
LAppendInt(myList, 16);
LAppendFloat(myList, 5.8);
LAppendString(myList, "How's Going Guys?");

string type = LGetType(myList, 1);
```

```
    if  (SIsSame(type, "char"))
        printf("%c is stored at index 1\n", LGetChar(myList, 1));
    else if (SIsSame(type, "int"))
        printf("%d is stored at index 1\n", LGetInt(myList, 1));
    else
        printf("I didn't check all types because I'm lazy.\n");
    LDelete(myList);
```

Output:

```
16 is stored at index 1
```

Inserting Item in List at Desired Position

If you wanna insert item somewhere in List not in the end specifically, then you can use LInsertChar, LInsertShort, LInsertInt, LInsertLong, LInsertLLong, LInsertFloat, LInsertDouble, LInsertLDouble, LInsertString, LInsertVector, LInsertList, LInsertDict. it takes three arguments, one is list, second is item, and third is index, let's take a look at example.

Code:

```
    list myList = initList();
    LAppendChar(myList, 'a');
    LAppendFloat(myList, 5.8);
    LAppendString(myList, "How's Going Guys?");
    LInsertInt(myList, 16, 1);

    logList(myList);
    LDelete(myList);
```

Output:

```
[0] <char>      a
[1] <int>       16
[2] <float>     5.800000
[3] <string>    How's Going Guys?
```

Getting the Size Of List

you can get the size of list by LLen function, or as we did in Vectors, you can also do something like myList->size and it'll be the size of List.

Code:

```
list myList = initList();
LAppendChar(myList, 'a');
LAppendFloat(myList, 5.8);
LAppendString(myList, "How's Going Guys?");

printf("Size Function: %ld\n", LLen(myList));
printf("Direct Access: %ld\n", myList->size);
LDelete(myList);
```

Output:

```
Size Function: 3
Direct Access: 3
```

Removing Item from List

Well as same same functionality with Vector and Array, you can remove item from List by LRemoveItem, it takes list and index and removes whatever item at that index, let's see an example

Code:

```
list myList = initList();
LAppendChar(myList, 'a');
LAppendFloat(myList, 5.8);
LAppendString(myList, "How's Going Guys?");

LRemoveItem(myList, 1);
logList(myList);

LDelete(myList);
```

Output

```
[0] <char>      a
[1] <string>    How's Going Guys?
```

Popping the last item from List

LPop is a function to pop the last item in List

Code:

```
list myList = initList();
LAppendChar(myList, 'a');
LAppendFloat(myList, 5.8);
LAppendString(myList, "How's Going Guys?");
LPop(myList);
logList(myList);
LDelete(myList);
```

Output:

```
[0] <char>      a
[1] <float>     5.800000
```

Replacing Item from List

Use LReplaceChar to replace character by an item, The type of item (which is gonna get replaced) doesn't matter. similarly you can use LReplaceShort, LReplaceInt, LReplaceLong, LReplaceLLong, LReplaceFloat, LReplaceDouble, LReplaceLDouble, LReplaceString, LReplaceVector, LReplaceList, LReplaceDict. let's see an example

Code:

```
list myList = initList();
LAppendChar(myList, 'a');
LAppendFloat(myList, 5.8);
LAppendString(myList, "How's Going Guys?");

LReplaceInt(myList, 2020, 1);
logList(myList);
LDelete(myList);
```

Output:

```
[0] <char>      a
[1] <int>       2020
[2] <string>    How's Going Guys?
```

Finding Item in List

In order to find a item in List, you'll need to clarify to which type of item you're looking for, like if you're finding a integer, there's separate function for that as LFindInt, similarly, LFindChar, LFindShort, LFindLong, LFindLLong, LFindFloat, LFindDouble, LFindLDouble, LFindString, LFindVector, LFindList and LFindDict to find other types of items, if it couldn't find any item, then it'll return -1.

Code:

```
list myList = initList();
LAppendChar(myList, 'a');
LAppendFloat(myList, 5.8);
LAppendString(myList, "How's Going Guys?");
logList(myList);

printf("<float> 5.8 is at %ld\n", LFindFloat(myList, 5.8));
LDelete(myList);
```

Output:

```
[0] <char>        a

[1] <float>       5.800000

[2] <string>      How's Going Guys?

<float> 5.8 is at 1
```

Finding Item from a Specific Position in List

You can do that by LFindCharFrom, LFindShortFrom, LFindIntFrom, LFindLongFrom, LFindLLongFrom, LFindFloatFrom, LFindDoubleFrom, LFindLDoubleFrom, LFindStringFrom, LFindVectorFrom, LFindListFrom, LFindDictFrom, Example

Code:

```
list myList = initList();
LAppendChar(myList, 'a');
LAppendInt(myList, 19);
LAppendString(myList, "How's Going Guys?");
LAppendInt(myList, 19);
logList(myList);
```

```
long FirstInt = LFindInt(myList, 19);
printf("Another <int> 19 is at: ");
printf("%ld\n", LFindIntFrom(myList, 19, FirstInt+1));
LDelete(myList);
```

Output:

```
[0] <char>      a
[1] <int>       19
[2] <string>    How's Going Guys?
[3] <int>       19
Another <int> 19 is at: 3
```

Comparing Two Lists

You can compare two List by accessing their items and comparing one by one, or you can use an alternative to do that, here's function LIsSame, you only need to pass it two lists and it'll compare their items, if items has nested structure then it'll go into those structures and compare their items too. if both lists are same that it'll return True (1) if not then it will return False (0). here's a example,

Code:

```
list myList = initList();
LAppendChar(myList, 'a');
LAppendInt(myList, 19);

list myList2 = initList();
LAppendChar(myList2, 'b');
LAppendInt(myList2, 18);

if (LIsSame(myList2, myList))
    printf("They are same\n");
else
    printf("They aren't same\n");
LDelete(myList);
```

Output:

```
They aren't same
```

for sure they're not same, both of their items are different.

Copying Items From One List To Another Lists

well you can copy all items of one list into another list by accessing all of its items, and appending into another list. but here's a short way to do that, you can use LCopy, it first takes destination list and then source list, and appends all items from source list to destination list. here's example

Code:

```
list myList = initList();
LAppendChar(myList, 'a');
LAppendInt(myList, 19);

list myList2 = initList();
LAppendChar(myList2, 'b');
LAppendInt(myList2, 18);
LCopy(myList, myList2);

logList(myList);
LDelete(myList);
```

Output:

```
[0] <char>      a
[1] <int>       19
[2] <char>      b
[3] <int>       18
```

Creating a Copy of List

You can also create a copy of your list by initializing new list and then copying all the items, but you can also do that in one line using LReCreate, it takes one list as a prototype. It first will initialize an empty list then copy all the items into it, here's an example:

Code:

```
list myList = initList();

LAppendChar(myList, 'a');
```

```
LAppendInt(myList, 19);

list myList2 = LReCreate(myList);

logList(myList2);
LDelete(myList);
```

Output:

```
[0] <char>      a

[1] <int>       19
```

See, it prints the contents of myList by logging the myList2, because it was created from the myList.

Nested Structures

You read a para something like that in the last of Vector chapter, the same concept also applies here, List is also pointer, but it's copy can be created by LReCreate, so whenever you insert it inside a List or Vector, you're not inserting it's reference, instead the append and insert function will create another copy of it then insert it. so you gotta delete the original list by LDelete, while that one copy inside the vector or List will be deleted when you delete that vector or list or just remove that item. On the other hand, when you get that by VGetList or LGetList you're not getting it's copy instead you're getting it's reference, if you change something in it, the item inside the vector or list will also be changed. These items that you get by LGetList or VGetList are actually own by the vector or list so you don't need to delete them either, they will manage and delete them (when you delete the vector or list). Read this Nested Structures again, this is very important concept.

With this, we have completed one more Data Structure, if you've read whole book until this page and understand everything, then I wanna congratulate you again that you have mastered one more Data Structure. now we have only one structure remaining named Dict. it's same as Python Dictionary, but within the laws of C. let's also explore that.

Chapter 6: Dictionary

Dictionary is a Structure used to Store data and access them very easily, unlike Vector and List where you gotta access element by their index, you gotta take of their index. In Dictionary there ain't such thing as index, each item has it's unique key (string). their order doesn't matter. their values are much likely to Lists but only accessible by key, each item item has unique string (key) from which it can be accessed, modified or removed. let's how you can do that.

Working With Dictionary

while key is everything in dictionary, so be careful about it this is case sensitive. we'll see how to initialize Dictionary, their item is called called identity, I don't have a reason why'd I choose this name. when I was writing those functions I couldn't think of anything else. each Identity has key (string) and it's item could have any type among all those 12 types. Happy? you can create new identities, remove existing one, check if dictionary has identity or not and many more. let's practically work with them.

Initializing Dictionary

There's type 'dict', you can use it to declare dictionary, and initialize dict by initDict, it's a simple function, whatever it return assign that to dict variable and it's ready, you can delete dictionary by DDelete, and it'll manage to free all of the memory that was allocated by dictionary. like this

Code:

```
dict Human = initDict();
DDelete(Human);
```

It outputs nothing, you dictionary has created in first, and deleted in second, you can work with it and use it before deleting it. so that's all. Or what else could be more simpler? I did it as much as simple as I could, but within the laws of C and my knowledge about it.

Adding Identities In Dictionary

If you wanna add a Identity, mean if you wanna add a key value pair, here's how, For instance if you wanna add a key and char pair (i have no idea why would someone identify character by string) so he can use DIdentifyChar, it first takes dict, then key for this identity, then the char, then this character can be identified by that key, similarly other functions, DIdentifyShort, DIdentifyInt, DIdentifyLong, DIdentifyLLong, DIdentifyFloat, DIdentifyDouble, DIdentifyLDoube, DIdentifyString, DIdentifyVector, DIdentifyList, DIdentifyDict to make other types items access by a key, and print the whole dictionary in a manner to debug by using logDict. let's understand better by an example

Code:

```
dict Human = initDict();

DIdentifyString(Human, "Name", "Rao Zaeem");
DIdentifyInt(Human, "Age", 16);
DIdentifyFloat(Human, "Height", 5.8);

logDict(Human);
DDelete(Human);
```

Output:

```
<string>      Name : Rao Zaeem
<int>         Age : 16
<float>       Height : 5.800000
```

Getting the Type Of Value in Dictionary

As you've seen in List, you can only get the item when you know the type of that item. You can get the type of value by it's key using function DGetValueType, give it dictionary and key, and it'll return string. and you can compare that. let's see how

Code:

```
dict Human = initDict();

DIdentifyString(Human, "Name", "Rao Zaeem");
DIdentifyInt(Human, "Age", 16);
```

```
DIdentifyFloat(Human, "Height", 5.8);

logDict(Human);

printf("Age has '%s' value\n", DGetValueType(Human, "Age"));
DDelete(Human);
```

Output:

```
<string>      Name : Rao Zaeem

<int>         Age : 16

<float>       Height : 5.800000

Age has 'int' value
```

Accessing The Values by key

You probably know which key has what type of value, right? if you want to get the value and you're sure it is string then use DGetString, give it the dictionary and key and it'll return you the item if that is string, if not then it'll print a error "message that this item has %s type", similarly DGetShort, DGetInt, DGetLong, DGetLLong, DGetFloat, DGetDouble, DGetLDouble, DGetVector, DGetList to get other types values. before moving on. I just want to realize you what can you do with Nesting. you can nest another dictionary in it, nest vector which also has nested structures. Amazing isn't that? back to accessing items, let's see example

Code:

```
dict Human = initDict();
DIdentifyString(Human, "Name", "Rao Zaeem");
DIdentifyInt(Human, "Age", 16);
DIdentifyFloat(Human, "Height", 5.8);

printf("Name: %s\n", DGetString(Human, "Name"));
printf("Age: %d\n", DGetInt(Human, "Age"));
printf("Height: %f\n", DGetFloat(Human, "Height"));

DDelete(Human);
```

Output:

```
Name: Rao Zaeem
```

Age: 16

Height: 5.800000

Getting the Number of Identities

you can get the number of Identities, in other words the length of your dictionary by DLen, and same using Pointer to Node, use myDict->size and that'll be the size of your dictionary. let's see by example

Code:

```
dict Human = initDict();

DIdentifyString(Human, "Name", "Rao Zayan");
DIdentifyInt(Human, "Age", 18);
DIdentifyFloat(Human, "Height", 5.7);
printf("Size Dict: %ld\n", DLen(Human));
printf("Direct Access: %ld\n", Human->size);

DDelete(Human);
```

Output:

Size Dict: 3

Direct Access: 3

Removing The Identities

In order to remove a identity, you can use DRemoveIdentity, give it the dictionary and key and it'll remove that identity, here's how

Code:

```
dict Human = initDict();

DIdentifyString(Human, "Name", "Rao Zaeem");
DIdentifyInt(Human, "Age", 16);
DIdentifyFloat(Human, "Height", 5.8);

DRemoveIdentity(Human, "Height");

logDict(Human);
DDelete(Human);
```

Output:

<string>	Name : Rao Zaeem

<int>	Age : 16

Checking for a Key

if you wanna know if the dictionary has a key or not? then you can use DHasKey function to check that, it'll return True(1) if dictionary has that key, if no then it'll return False (0), Here's Quick example for you

Code:

```
dict Human = initDict();
DIdentifyString(Human, "Name", "Rao Zaeem");
DIdentifyInt(Human, "Age", 16);

if  (DHasKey(Human, "Name"))
	printf("Name Found\n");
else
	printf("Name Not Found\n");

if  (DHasKey(Human, "Height"))
	printf("Height Found\n");
else
	printf("Height Not Found\n");
DDelete(Human);
```

Output:

Name Found

Height Not Found

Checking If Two Dictionaries has Same Keys

In need to make sure if two Dictionaries has same keys or not, you can use DHaveSameKeys, give it both the dictionaries as arguments and it'll return True if they both have same keys and false if they're not. here's example

Code:

```
dict Human1 = initDict();

DIdentifyString(Human1, "Name", "Rao Zaeem");
DIdentifyInt(Human1, "Age", 16);
```

```
dict Human2 = initDict();
DIdentifyString(Human2, "Name", "Rao Zayan");
DIdentifyInt(Human2, "Age", 18);

if (DHaveSameKeys(Human1, Human2))
    printf("Human1 and Human2 have same keys\n");
else
    printf("Human1 and Human2 don't have same key\n");
DDelete(Human1);
DDelete(Human2);
```

Output:

```
Human1 and Human2 have same keys
```

It'll not compare their values, it'll only check for Keys, both dictionaries should have same number of keys or it'll return false.

Copying Identities from One Dictionary to Another Dictionary

To Copy All Identities from one Dictionary to another Dictionary, you can use DCopyIdentities, first it takes destination dictionary then source dictionary, when a key is already exists in destination dictionary, then it'll over write that. mean it'll update the value of that key from source dictionary. I hope you're getting an idea, here's example

Code:

```
dict Human1 = initDict();
DIdentifyString(Human1, "Name", "Rao Zayan");
DIdentifyInt(Human1, "Age", 18);
DIdentifyFloat(Human1, "Height", 5.7);

dict Human2 = initDict();
DIdentifyString(Human2, "Name", "None");

DCopyIdentities(Human2, Human1);
logDict(Human2);

DDelete(Human1);
DDelete(Human2);
```

Output:

```
<string>      Name : Rao Zayan
<int>         Age : 18
<float>       Height : 5.700000
```

Comparing Two Dictionaries

You can compare Two Dictionaries by DIsSame, pass it two dictionaries and it'll return True or False. remember it'll check keys and values both, their order doesn't matter but both of the dictionaries should have same keys with same valueTypes with same values. then it'll return 1 (True). let's see example

Code:

```
dict Human1 = initDict();
DIdentifyString(Human1, "Name", "Rao Zayan");
DIdentifyInt(Human1, "Age", 18);
DIdentifyFloat(Human1, "Height", 5.7);

dict Human2 = initDict();
DIdentifyString(Human2, "Name", "None");
DCopyIdentities(Human2, Human1);

if (DIsSame(Human1, Human2))
    printf("Both Humans are same\n");
else
    printf("They're different\n");

DDelete(Human1);
DDelete(Human2);
```

Output:

```
Both Humans are same
```

Copying A Dictionary

To create a copy of dictionary, well you can create a initialize new dictionary and copy all items, I don't know why I always build something that is useless.. anyway, here's a way that you can do that in one line using DReCreate, it'll do the same thing of initializing and copying. let's see by example.

Code:

```
dict Human1 = initDict();
DIdentifyString(Human1, "Name", "Rao Zayan");
DIdentifyInt(Human1, "Age", 18);
DIdentifyFloat(Human1, "Height", 5.7);

dict Human2 = DReCreate(Human1);

logDict(Human2);

DDelete(Human1);
DDelete(Human2);
```

Output:

```
<string>      Name : Rao Zayan

<int>         Age : 18

<float>       Height : 5.700000
```

See, it prints the contents of Human1 while we log the Human2, because Human2 was created by Human1.

Nested Structures

You have read this concept twice, once in Chapter of Vector and another in Chapter of List, Dictionary is also the same, so these are 4 structures namely String, Vector, List And Dictionary, they're just a pointer. but when you nest them so their copies will be inserted. but when you get them, you'll get their reference. Now you've got the idea and be able to Create each kind of structure with them. I hope you enjoyed learning it.

With the end of Dictionaries, we've mastered all the Data Structures of SoftC, In case in any addition to Data structure in Future, re clone the repository from GitHub and it'll download the latest version of this book, this is always available free for you. I hope that the time you spent here will be worth it, and you've learned everything clearly and easily. Keep Practicing. Good Bye..

Best Regards

Rao Zaeem Shahid