Dynamische Container C

Lukas Momberg [11141259]
Dennis Goßler [11140150]

Inhaltsverzeichnis

1. Einleitung

Vorwort:

Um Daten zu speichern werden Container allerart benötigt.
In der Programmierspreche C kann dies oftmals eine Herausforderung sein, da es nur sehr primitive Arten der Speicherung als Standard enthält.

Da wir in Zukunft uns auch weiterhin mit der Sprache C / C++ intensiv weiterbilden wollen, haben wir ein paar Lösungen für die Programmiersprache C entwickelt. (Listen, Dictionary, Stack, Queue, LinkedList und eine String Implementierung auf Basis der Liste)

1. Einleitung

Grundlegende Herangehensweise:

Alle unsere Container benutzen eine sehr ähnliche Grundlegende Struktur.
Ein Beispiel wäre hierfür, dass sich in jeder Collection eine .h Datei befindet, die ein Struck des jeweiligen Containers enthält.

```
typedef struct List_
{
    void** Content;
    size_t SizeOfSingleElement;
    unsigned int Size;
}List;
```

```
typedef struct Dictionary_
{
    size_t SizeOfKey;
    size_t SizeOfValue;
    unsigned int Size;

    DictionaryTreeItem* Root;
}Dictionary;
```

1. Einleitung

Grundlegende Herangehensweise:

Des weiteren benutzen unsere Container Voidpointer um auf die zu speienden Elemente zu zeigen.

Zusätzlich finden Sie Kommentare zu jeder Funktion die eine Speicherstruktur hat.

```
// Removes Element at index
// @param list: Address of list object
// @param index: position of the object that will be removed
// @return CollectionError: Errorcode that contains information about the operation.
/* --------------------
* returns: CollectionNoError
*    CollectionEmpty
*    CollectionArrayIndexOutOfBounds
*/
CollectionError ListItemRemove(List* list, unsigned int index);
```

1. Einleitung

Grundlegende Herangehensweise:

Wenn eine Funktion auf eine Collection fehlschlägt, können folgende Fehlercodes zurückgegeben:

```
CollectionError
{
	// Successful / No Error
	CollectionNoError,

	// Some internal allocation failed
	CollectionOutOfMemory,

	// There is no data to pull.
	CollectionEmpty,

	// Specified index points not to any data.
	CollectionArrayIndexOutOfBounds,

	// Specified 'ElementSize' was Zero.
	CollectionNoElementSizeSpecified,

	// Specified 'data' is Null-Pointer
	CollectionElementIsNullPointer,

	//Dictionary specific
	CollectionKeyAlreadyExists

}
```

2.1 Grundcontainer [List]

List:

Ziel:

Unsere Implementierung der Liste zielt darauf ab, Elemente schnell und einfach einer Liste hinzuzufügen.

Möglichkeiten:

Nach der Initialisierung der Liste gibt es die Möglichkeit, mit der „ListItemAdd“ Funktion, Elemente der List hinzuzufügen oder mit „ListItemRemove“ Elemente zu löschen. Die Liste wird automatisch, wenn nötig, vergrößert.

Alle möglichen Funktionen finden Sie auf den folgenden Seiten. (siehe List [Alle Funktionen])

2.1 Grundcontainer [List]

List:

Größenverwaltung:

Wir ein weiteres Element einer vollen Liste[1] hinzugefügt, wird diese automatisch vergrößert und alle neuen freien Positionen werden mit 0x00 vollgeschrieben.

$$Größe_{neu} = 1.5 * (Größe_{alt} + 1)$$

1: Wenn alle Speicherpositionen ungleich 0x000 sind.

2.1 Grundcontainer [List]

List:

Diese Abbildung soll verdeutlichen wie sich die Daten in der List, bei anwenden von Funktionen, verhalten.

Schritt	Pos .0	Pos.1	Pos.2	Pos.3	Pos.4	...	Eingabe	Ausgabe
0 – Voher	Null						-	-
1 – Allocation	\0	\0	\0	\0		...	4	-
2 – ADD	A	\0	\0	\0		...	A	-
3 – INSERT	A	\0	D	\0		...	2, D	-
4 – ADD	A	C	D	\0		...	C	-
5 – GET	A	C	D	\0		...	2	D
6 – REMOVE	A	C	\0	\0		...	2	-
7 – GET	A	C	\0	\0			2	NULL
7 – Deconstruct	Null						-	-

2.1 Grundcontainer [List]

List [Alle Funktionen]:

Funktionsname	Rückgabewert	Parameter	Nutzen
ListInitialize	void	List* list, unsigned int count, size_t sizeOfSingleElement	Inizialisierung des Containers
ListDestruction	void	List* list	Gibt Speicher frei
ListItemInsertAt	CollectionError	List* list, unsigned int indexValue, void* value	Fügt/ ersetzt das Element beim Index
ListItemAdd	CollectionError	List* list, unsigned int index, void* out	Fügt Wert zur Liste hinzu
ListItemGet	CollectionError	List* list, void* value	Nimmt einen Wert aus dem Container
ListItemRemove	CollectionError	List* list, unsigned int index	Entfernt ein Wert und gibt den Speicher frei
ListClear	CollectionError	List* list	Setzt alle Werte auf NULL

2.1 Grundcontainer [List]

List [Beispielcode 1/2]:

```
List exampleList = EMPTYLIST;
ListInitialize(&exampleList, 4, sizeof(char*));

char* outputChar;
char* testString1 = calloc(2, sizeof(char));
memcpy(testString1, "A", 1 * sizeof(char));

char* testString2 = calloc(3, sizeof(char));
memcpy(testString2, "BB", 2 * sizeof(char));

char* testString3 = calloc(4, sizeof(char));
memcpy(testString3, "DDD", 3 * sizeof(char));
```

2.1 Grundcontainer [List]

List [Beispielcode 2/2]:

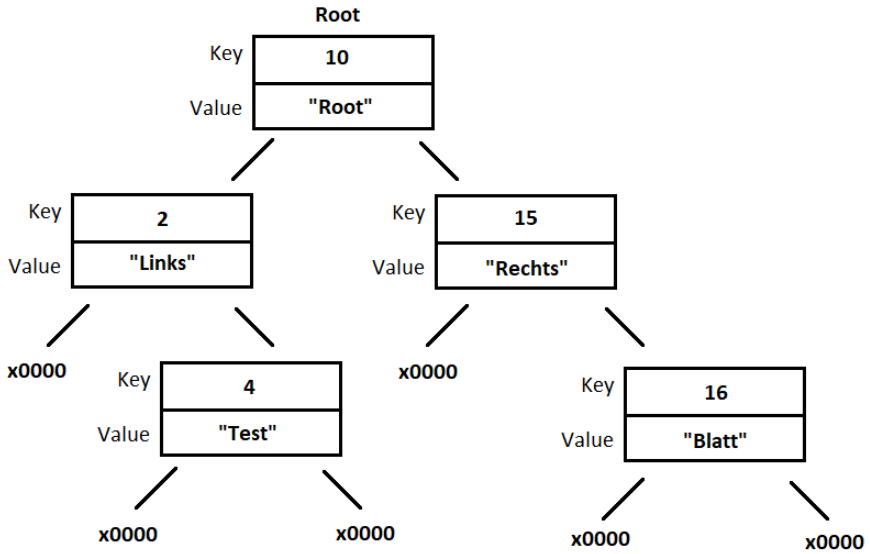
```
// Add functions
ListItemAdd(&exampleList, testString1);
ListItemInsertAt(&exampleList, 2, testString3);
ListItemAdd(&exampleList, testString2);

// Get value
ListItemGet(&exampleList, 2, &outputChar);
// Remove value
ListItemRemove(&exampleList, 2);
// Output is Null
ListItemGet(&exampleList, 2, &outputChar);

ListDestruction(&exampleList);
```

2.2 Grundcontainer [Dictionary]

Dictionary:

Erklärung:

Oftmals müssen Daten Schnell abrufbar sein. Hierfür haben wir ein Dictionary implementiert welches Daten in Form eines ‚Key' und einer zugehörigen Wert (Value) in einer Baumstruktur speichert.



Diese Abbildung zeigt wie die Daten im Dictionary verwaltet werden.

2.2 Grundcontainer [Dictionary]

Dictionary:

Funktionsweise:

Das Dictionary wurde als ein Binärer Suchbaum implementiert. Dieser besitzt pro Eintrag einen ‚Key' und eine ‚Value'. Beim Einfügen wird von der Wurzel ab entschieden, ob der jeweilige ‚Key' größer oder kleiner ist. Wird eine passende Stelle gefunden wird der Datensatz dort gespeichert.

Key Duplikate sind nicht zulässig.

2.2 Grundcontainer [Dictionary]

Dictionary [Alle Funktionen]:

Funktionsname	Rückgabewert	Parameter	Nutzen
DictionaryInitialize	void	Dictionary* dictionary, unsigned int sizeOfKey, unsigned int sizeOfValue	Inizialisierung des Containers
DictionaryDestruction	void	Dictionary* list	Gibt Speicher frei
DictionaryContainsKey	char	Dictionary* dictionary, void* key	Gibt 0 / 1 zurück, ob Item vorhanden ist
DictionaryAdd	CollectionError	Dictionary* dictionary, void* key, void* value	Fügt Wert zum Dictionary hinzu
DictionaryGet	CollectionError	Dictionary* dictionary, void* key, void* out	Nimmt einen Wert aus dem Container
DictionaryRemove	CollectionError	Dictionary * list, unsigned int index	Entfernt ein Wert und gibt ihn frei

2.2 Grundcontainer [Dictionary]

Dictionary [Beispielcode 1/2]:

```
Dictionary exampleDic = EMPTYDICTIONARY;
DictionaryInitialize(&exampleDic, sizeof(int*), sizeof(char*));

char* outputAddress;

int t0 = 10, t1 = 15, t2 = 2, t3 = 4, t4 = 16;

// Add functions
DictionaryAdd(&exampleDic, &t0, "Root");
DictionaryAdd(&exampleDic, &t1, "Rechts");
DictionaryAdd(&exampleDic, &t2, "Links");
DictionaryAdd(&exampleDic, &t3, "Test");
DictionaryAdd(&exampleDic, &t4, "Blatt");
```

2.2 Grundcontainer [Dictionary]

Dictionary [Beispielcode 2/2]:

```
// Get value
DictionaryGet(&exampleDic, &t4, &outputAddress);

// Remove value
DictionaryRemove(&exampleDic, &t2);

DictionaryDestroy(&exampleDic);
```

2.3 Grundcontainer Stack

Daten werden gestapelt, der ältere Wert wird vom neuern verdeckt.
Es kann immer ein Wert hinzugefügt werden, beim Entfernen wird der neueste Wert entfernt,
First In Last Out (FILO) Prinzip.
Diese Vorgänge sind sehr schnell und sind generell sicher.

Schritt	0x01	0x02	0x03	0x04	0x05	...	Eingabe	Ausgabe
0 – Voher	Null						-	-
1 – Allocation	\0	\0	\0	\0	\0	...	-	-
2 – Push	A	\0	\0	\0	\0	...	A	-
3 – Push	A	B	\0	\0	\0	...	B	-
4 – Push	A	B	C	\0	\0	...	C	-
5 – Pull	A	B	\0	\0	\0	...	-	C
6 – Pull	A	\0	\0	\0	\0	...	-	B
7 – Pull	Null						-	A

2.3 Grundcontainer Stack

Stack [Alle Funktionen]:

Funktionsname	Rückgabewert	Parameter	Nutzen
StackInitialize	void	Stack* stack unsigned int sizeOfSingleElement	Inizialisierung des Containers
StackClear	void	Stack* stack	Löschen aller Daten des Containers
StackPush	CollectionError	Stack* stack void* element	Fügt Wert zum Container hinzu
StackPull	CollectionError	Stack* stack void* element	Nimmt einen Wert aus dem Container

2.3 Grundcontainer Stack

Stack [Beispeilcode]:

```
char x = 0;
CollectionError collectionError;
Stack stack;
StackInitialize(&stack, sizeof(char));

//---<Add>----------------------------------
char a[] = "Hello";

StackPush(&stack, &a[0]); // Add 'H'
StackPush(&stack, &a[1]); // Add 'e'
StackPush(&stack, &a[2]); // Add 'l'
StackPush(&stack, &a[3]); // Add 'l'
StackPush(&stack, &a[4]); // Add 'o'
//-----------------------------------------

//---<Pull>---------------------------------
StackPull(&stack, &x); // x now contains 'o'
StackPull(&stack, &x); // x now contains 'l'
StackPull(&stack, &x); // x now contains 'l'
StackPull(&stack, &x); // x now contains 'e'
StackPull(&stack, &x); // x now contains 'H'
//-----------------------------------------

//---<Clear>--------------------------------
collectionError = StackPull(&stack, &x); // Will return that the List is Empty
//-----------------------------------------
```

2.4 Grundcontainer Queue

Daten werden in einer Liste gespeichert, der älteste Wert wird hier entnommen. First in First out (FIFO). Da Daten von Vorne entnommen werden entsteht ungenutzter Speicher.
Hier ist zu Achten, dass dieser Speicher möglichst freigegeben wird, natürlich ist eine umbauen des Speichers bei jedem Zugriff nicht unbedingt Sinnvoll.

Schritt	0x01	0x02	0x03	0x04	0x05	...	Eingabe	Ausgabe
0 – Voher	Null						-	-
1 – Allocation	\0	\0	\0	\0	\0	...	-	-
2 – Push	A	\0	\0	\0	\0	...	A	-
3 – Push	A	B	\0	\0	\0	...	B	-
4 – Push	A	B	C	\0	\0	...	C	-
5 – Pull	A	B	C	\0	\0	...	-	A
6 – Pull	A	B	C	\0	\0	...	-	B
7 – Pull	Null						-	C

2.4 Grundcontainer Queue

Queue [Alle Funktionen]:

Funktionsname	Rückgabewert	Parameter	Nutzen
QueueInitialize	void	Queue* queue unsigned int sizeOfSingleElement	Inizialisierung des Containers
QueueClear	void	Queue* queue	Löschen aller Daten des Containers
QueuePush	CollectionError	Queue* queue void* element	Fügt Wert zum Container hinzu
QueuePull	CollectionError	Queue* queue void* element	Nimmt einen Wert aus dem Container

2.4 Grundcontainer Queue

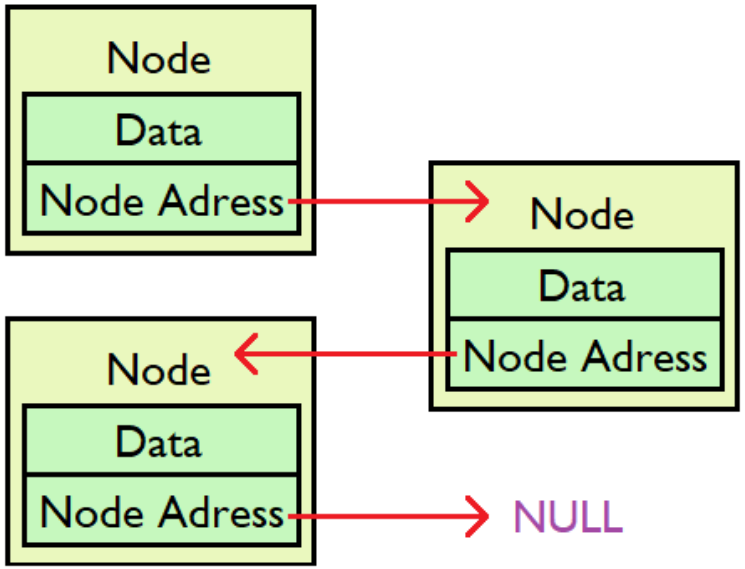
Queue [Beispielcode]:

```
char x = 0;
CollectionError collectionError;
Queue queue;
QueueInitialize(&queue, sizeof(char));

//---<Add>-----------------------------------------------
char a[6] = "ABCDE";

collectionError = QueuePush(&queue, &a[0]); // Add 'A' with optional errorCheckValue

QueuePush(&queue, &a[1]); // Add 'B'
QueuePush(&queue, &a[2]); // Add 'C'
QueuePush(&queue, &a[3]); // Add 'D'
QueuePush(&queue, &a[4]); // Add 'E'
//-------------------------------------------------------

//--<Pull>----------------------------------------------

    QueuePull(&queue, &x); // x will contain 'A'
    QueuePull(&queue, &x); // x will contain 'B'
    QueuePull(&queue, &x); // x will contain 'C'
    QueuePull(&queue, &x); // x will contain 'D'
    QueuePull(&queue, &x); // x will contain 'E'
//-------------------------------------------------------

//---<Clear<---------------------------------------------
collectionError = QueuePull(&queue, &x);
//-------------------------------------------------------
```

2.5 Grundcontainer [LinkedList]

Daten werden in Ketten-Elementen gespeichert. Jedes zwischen Element kennt seinen nächsten Nachbarn. Durch diese Kette kann man jedes Element ansprechen. Das Letze Element hat immer einen Null Wert, da dieser der Letze Wert ist und keinen nächsten Wert besitzt.

Schritt	Nr.1	Nr.2	Nr.3	Nr.4	Nr.5	...	Eingabe	Ausgabe
1	Null						-	-
	Null					...		
2 - Push	A					...	A	-
	Null					...		
3 - Push	A					...	B	-
	->	B				...		
		Null				...		
4 - Push	A					...	C	-
	->	B				...		
		->	C			...		
			Null			...		
5 – Pull(1)	A					...	-	B
	->	C				...		
		Null				...		



2.5 Grundcontainer [LinkedList]

LinkedList [Alle Funktionen]:

Funktionsname	Rückgabewert	Parameter	Nutzen
LinkedListInitialize	void	LinkedList* linkedList unsigned int sizeOfElement	Inizialisierung des Containers
LinkedListClear	void	LinkedList* linkedList	Löschen aller Daten des Containers
LinkedListInsert	CollectionError	LinkedList* linkedList unsigned int index void* element	Fügt Wert zum Container hinzu
LinkedListAddToEnd	CollectionError	LinkedList* linkedList void* element	Fügt Wert am Ende des Containers an
LinkedListRemoveAtIndex	CollectionError	LinkedList* linkedList unsigned int index void* element	Entnimmt Wert aus gegebener Position
LinkedListGetElement	CollectionError	LinkedList* linkedList unsigned int index void* element	Liest Wert aus gegebener Position
LinkedListGetNode	CollectionError	LinkedList* linkedList unsigned int index LinkedListNode** linkedListNode	Liest Knoten aus gegebener Position
LinkedListGetLastElement	CollectionError	LinkedList* linkedList LinkedListNode** linkedListNode	Liest letzten Knoten

2.5 Grundcontainer [LinkedList]

LinkedList [Beispielcode]:

```
LinkedList linkedList;

LinkedListInitialize(&linkedList, sizeof(int));

//---<Insert>-------------------------------------------
int data[] = {10,20,30,40,50,60};

LinkedListAddToEnd(&linkedList, &data[0]);
LinkedListAddToEnd(&linkedList, &data[1]);
LinkedListAddToEnd(&linkedList, &data[2]);
LinkedListAddToEnd(&linkedList, &data[3]);
LinkedListAddToEnd(&linkedList, &data[4]);
LinkedListAddToEnd(&linkedList, &data[5]);
//------------------------------------------------------

//---<Read>---------------------------------------------
int extractedData[6];

// Get elements form index
LinkedListGetElement(&linkedList, 0, &extractedData[0]);
LinkedListGetElement(&linkedList, 1, &extractedData[1]);
LinkedListGetElement(&linkedList, 2, &extractedData[2]);
LinkedListGetElement(&linkedList, 3, &extractedData[3]);
LinkedListGetElement(&linkedList, 4, &extractedData[4]);
LinkedListGetElement(&linkedList, 5, &extractedData[5]);
//------------------------------------------------------
```

3.1 Generelle Container [String]

String:

Ziel:

Ziel hinter diesem Container war es zu zeigen wie leicht es ist unsere Container zu erweitern und oder zu verändern.

Möglichkeiten:

Da unser ‚String Container' auf unser Liste aufbaut, sieht dieser Container der List sehr ähnlich. Es wurden zusätzliche Funktionalitäten hinzugefügt, wie zum Beispiel, dass dem Conatiner ein weiterer String angehangen werden kann (Concat Funktion).

3.1 Generelle Container [String]

String [Alle Funktionen]:

Funktionsname	Rückgabewert	Parameter	Nutzen
StringInitialize	void	String* string, char* inputString	Inizialisierung des Containers
StringDestruction	void	String* string	Gibt Speicher frei
StringCharInsertAt	CollectionError	String* string, unsigned int indexValue, char value	Fügt/ ersetzt einen Char beim Index
StringCharAdd	CollectionError	String* string, char addChar	Fügt einen Char an die nächste freie Stelle hinzu
StringCharGet	CollectionError	String* string, unsigned int index, char* out	Gibt ein Char aus dem String zurück
StringConcat	CollectionError	String* string, char* addString	Vereint ein String Objekt mit einem char*
StringGetFullString	char*	String* list	Gibt den String als char* zurück. Muss wieder an den Ram freigeben!

3.1 Generelle Container [String]

String [Beispiel 1/2]:

```
char* testChar = calloc(12, sizeof(char));
memcpy(testChar, "Test String", 11 * sizeof(char));

String testString = EMPTYSTRING;

StringInitialize(&testString, testChar);

char* outputChar;
```

3.1 Generelle Container [String]

String [Beispiel 2/2]:

```
//Gets 't'
StringCharGet(&testString, 3, &outputChar);
//Adds '!' to the end
StringCharAdd(&testString, '!');

//Gets '!'
StringCharGet(&testString, 11, &outputChar);
StringConcat(&testString, "TEST");

char* output = StringGetFullString(&testString);

StringDestruction(&testString);
free(output);
```

4. Tests

Erläuterung:

Da wir schnell festgestellt haben, dass es sehr schwierig ist unsere Collections auf mehreren Plattformen fortlaufend zu testen, haben wir uns dafür entscheiden ein eigenes Testsystem zu schreiben.

Funktionsweise:

Unsere Testfunktionen nehmen zwei Werte auf. Einer welcher der erwartetet Wert ist und welcher Wert zurückgeliefert wird. Am Ende bekommt man eine Auflistung wie viele Test geklappt oder fehlgeschlagen sind.

Die Testfunktionen unterstützen Integer, String und CollectionError Werte.

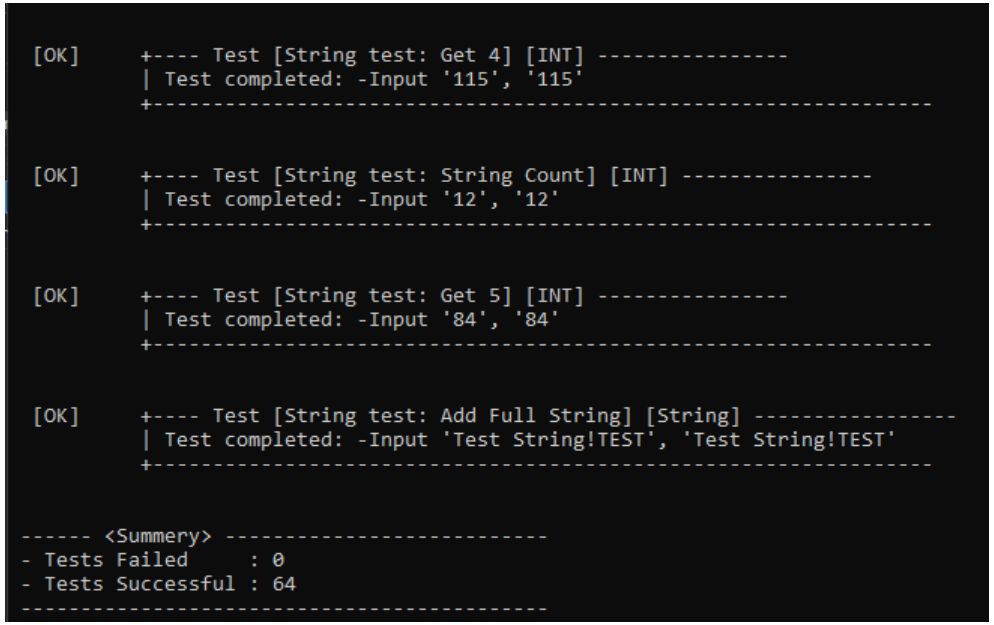
```
void test_int(int expectedInput, int input, char* name);
void test_int_Failure(int expectedInput, int input, char* name);

void test_string(char* expectedInput, char* input, char* name);
void test_string_Failure(char* expectedInput, char* input, char* name);
```

4. Tests

Memory leak:

Unser Projekt wurde vollständig auf Memory Leaks geprüft.

Unsere Tests:



5. Kompilierung

Windows:

Öffne das Projekt in Visual Studio und klicke das Test-Projekt und wähle es als Start-Projekt aus. Ohne diese Einstellung wird möglicherweise versuch eine .lib Datei zu öffnen, dort wird dann ein Fehler angezeigt.

Linux:

Navigiere zu der makefile. Dort befindet sich auch ein Skript das zum kompilieren genutzt werden kann. Es erstellt einen bin Order für Temporäre Daten, danach kompiliert es und startet das fertige Programm danach.
Vergiss nicht die Skript-Datei ausführbar zu machen mit chmod +x FileName.sh.
Dann zum Öffnen ./Filename.sh

Der genutzte Code ist auch auf GitHub zu finden

https://github.com/DennisGoss99/InPr_DynamicCollections