Report for the final project of IP January 2025

Annique Quarles van Ufford, Mafalda Candal, Julia Romocea, Zsófia Sándor
Deadline: 30th Jan 23:59

Task description

This project challenged us to create a program that effectively manages students' data. We were given two header files defining two ADTs and their functionalities: student/studentInner, which holds a student's personal information such as their name, surname, student ID, year of enrollment, and a collection of the courses they've passed, and studman/studentManInner, which groups students. Our task was to implement these ADTs and their functionalities. In particular, the collection of passed courses needed to be dynamic to accommodate students taking new courses each semester, while the student manager system, though updated less frequently, was designed to provide quick access to specific student details.

Structures and Functions

As a general overview: *studman* are pointers to an inner structure *studManInner*, which saves the startingID, an array of *students*, as well as the size of the array and the current number of students stored inside it. Similarly, a *student* is a pointer to an inner structure *studentInner*, which saves the student's personal information, including a *List* passedCourses. The *List* is a pointer to the struct *ListNode* which saves a pointer to the name of a passed course (previously saved in the heap) and a pointer to the next node.

For both the collection of students and the collection of courses, we had to choose between using an array or a linked list. We chose to implement the collection of students using an array because the student manager is very static (the collection of students rarely changes), and requires quick and easy access to student's information. Both of these characteristics justify the use of an array, where adding students has a complexity of O(n) but accessing at a specific index is in constant time. The initial size was not specified, so we decided on 10, as we found it a good compromise between not needing to double very often, but also not taking up too much unnecessary memory. On the other hand, we implemented a linked list to store the passed courses for each student because linked lists can be easily modified. Assuming that the courses which the student passes changes frequently (around four times a year, at least), a linked list allows them to add these courses in constant time. In addition, unlike an array implementation, the linked list does not require the allocation of more space than is currently being used.

In the studentManager code, we implemented several functions to access students' information, including the getName function. This function retrieves a student's name based on their unique studentID. It first calls the studentIndex function, which checks that the studman isn't empty and then calculates the index of the student in the students array by subtracting the startingID from their studentID. This approach not only eliminated the need for a full search, ensuring constant time access, but also prevented unnecessary repetition of

code. If the index is valid, the function retrieves the student's name using the getStudentName function and returns it, otherwise it returns "NONE".

The getPassedCourses function returns an array of the courses that a student has passed, in the order they were completed. If the student is NULL or has not passed any courses, the function returns NULL. If this is not the case, the function dynamically allocates memory for an array to store the names of the courses. It then iterates through the passedCourses linked list, filling the array from the back to the front to ensure the correct order. This is necessary because the courses in the linked list are stored in reverse order, with the most recently passed courses at the front.

Testing

When we discussed how we would test our code, each of us thought of a different approach. Some suggested using makefiles, while others proposed adding a temporary main function to the implementation files. In the end, we decided to create our own tester file, which allowed us to debug more easily since we could call each function with print statements in between. We also employed other debugging techniques, such as Valgrind, in combination with drawing out the commands on paper, in order to identify the memory leaks we were experiencing. Aside from using the tests provided on CodeGrade to ensure that our program met the required functionality, we also looked at specific edge cases using our tester file.

These included passing an empty studman to the functions, handling non-existent student IDs, and trying to identify a student with a common surname. Additionally, we addressed what would happen if a student's name or surname was longer than 10 characters. We decided that the student would not be added and an error would be emitted (instead of truncating the name). We felt that this would be a better alternative in real life, as otherwise users could be unsuccessfully looking for a student they added because the name was altered with no warning.

Lastly, we intended to make getStudentPassedCourses return an array of pointers to copies of the strings so that the originals could not be edited, but this required the allocation of more memory and led to memory leaks, as the tester.c file did not free such memory.

Conclusion and Self-Reflection

This project helped us understand the importance of abstraction and separating declaration from implementation. While we had worked with header files before, this was our first time working with multiple header files in a single project. One of the challenges we faced was that, even though we included student.h in studentManager.c early on, we didn't acknowledge that studentManager.c could use the functions declared in student.h. This happened because, after discussing the data structures we wanted to use, we quickly split up the work, with two of us focusing on student.c and two on studentManager.c, without first considering how these files would interact. As a result, we had to revisit our code and integrate the existing functions later. Additionally, if we had more control over the project structure, we would have created a separate header and source file for the linked list to achieve further modularization. Overall, this project gave us the opportunity to apply our knowledge of Abstract Data Types and explore their practical use in real-world applications.