UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA
Denkleiers • Leading Minds • Dikgopolo tša Dihlalefi

Department of Computer Science
Faculty of Engineering, Built Environment & IT
University of Pretoria

COS110 - Program Design: Introduction

Bonus Practical Specifications

Release Date: 04-11-2024 at 06:00

Due Date: 07-11-2024 at 23:59

Total Marks: 200

Contents

1 General Instructions

- ***Read the entire assignment thoroughly before you start coding.***
- This assignment should be completed individually, no group effort is allowed.
- **To prevent plagiarism, every submission will be inspected with the help of dedicated software.**
- Be ready to upload your assignment well before the deadline, as **no extension will be granted.**
- If your code does not compile, you will be awarded a mark of 0. The output of your program will be primarily considered for marks, although internal structure may also be tested (eg. the presence/absence of certain functions or classes).
- Failure of your program to successfully exit will result in a mark of 0.
- Note that plagiarism is considered a very serious offence. Plagiarism will not be tolerated, and disciplinary action will be taken against offending students. Please refer to the University of Pretoria's plagiarism page at `https://portal.cs.up.ac.za/files/departmental-guide/`.
- Unless otherwise stated, the usage of c++11 or additional libraries outside of those indicated in the assignment, will **not** be allowed. Some of the appropriate files that you have submit will be overwritten during marking to ensure compliance to these requirements. **Please ensure you use c++98**
- All functions should be implemented in the corresponding `cpp` file. No inline implementation in the header file apart from the provided functions.
- The usage of ChatGPT and other AI-Related software is strictly forbidden and will be considered as plagiarism.
- Note, UML notation for variable and function signatures are used in this assignment.
- Please note that there is a late deadline which is 1 hour after the initial deadline, but you will lose 20% of your mark.
- Note you may only use recursion for this practical. The use of iterative loops (for, while, do while) are strictly forbidden.

2 Overview

As part of your studies of COS110, you have been introduced to many different concepts. This bonus practical will combine all of these concepts into a singular bonus practical.

You have been introduced to two different dynamic linear data structures, namely dynamic arrays and linked lists. You will need to implement a selection of functions which include pure virtual,

friend and overloaded operators. Lastly, both of the data structures have been abstracted into a hierarchy and then templatised.

Safe programming skills are key for any programmer, so you will also need to include bound checking and throw an appropriate exception when specified.

Additionally, **you may only use recursive solutions for this practical**. You are allowed to create stand-alone helper functions in the relevant cpp file.

3 Your Task:

You are required to implement the following class diagram illustrated in Figure 1. Pay close attention to the function signatures as the `h` files will be overwritten, thus failure to comply with the UML, will result in a mark of 0.

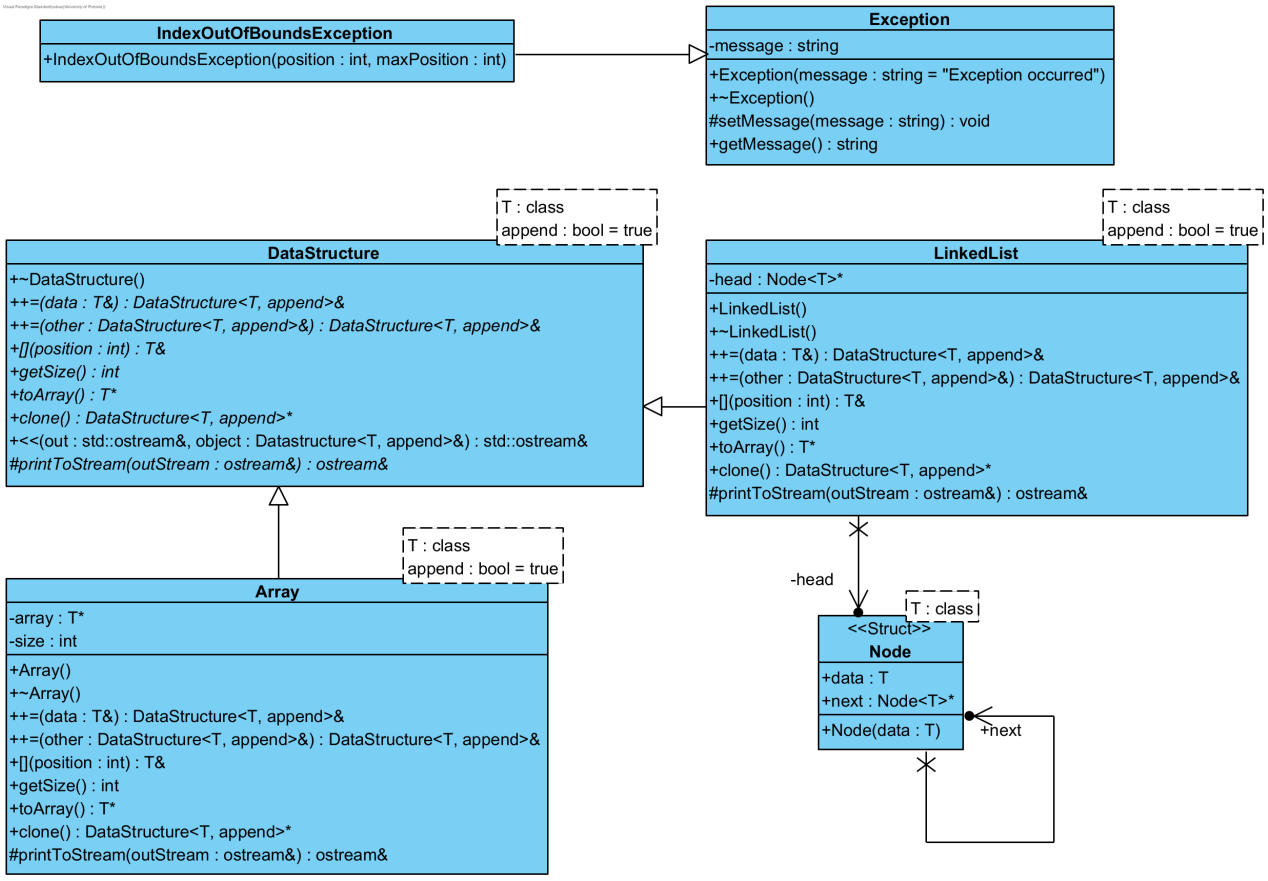

Figure 1: Class diagrams

Note, the importance of the arrows between the classes are not important for COS 110 and will be expanded on in COS 214.

3.1 Exception

This class is the base class for an exception hierarchy.

3.1.1 Members

- message: string
  - This is the error message associated with the exception.

3.1.2 Functions

- Exception(message: string = "Exception occurred")
  - This is the constructor for the Exception class.
  - The constructor should initialize the member variable with the passed in parameter.
- ~Exception()
  - This is the destructor for the exception class.
- setMessage(message: string): void
  - This function should set the message member variable with the passed-in parameter.
- getMessage(): string
  - This is the getter for the message member variable.
  - The function is a const function.

3.2 IndexOutOfBoundsException

This is a specialisation of the Exception class, which deals with incorrectly accessing elements in the data structure.

3.2.1 Functions

- IndexOutOfBoundsException(position: int, maxPosition: int)
  - This is the constructor for the IndexOutOfBoundsException class.
  - This constructor should formulate the following error message, and then store it in the inherited message member variable:

```
Tried to access <position> in bound <maxPosition>
```

  - Note the angle brackets should not be included in the final error message.
  - An example of an error message is:

```
Tried to access 10 in bound 2
```

  - There should be **no** newline at the end of the error message.

3.3 DataStructure

This is the base class for the array and linkedlist class. It takes in two template parameters. The first is the data type the class works on, and the second specifies what type of insert the class will utilise. If `append` is true new elements should be appended to the rear of the data structure, else elements should be prepended to the front of the data structure.

3.3.1 Functions

- ~DataStructure
  - This destructor should be empty.
- operator+=(data: T&): DataStructure<T, append>&
  - This is a pure virtual function.
  - This function will be used to insert the passed-in parameter to the data structure and is based on the append template parameter.
  - The function should return the dereferenced this.
- operator+=(other: DataStructure<T, append>&): DataStructure<T, append>&
  - This is a pure virtual function.
  - This function will be used to insert all of the data contained in the passed-in parameter to the data structure, based on the append template parameter.
  - The function should return the dereferenced `this`.
- operator[](position: int): T&
  - This is a pure virtual function.
  - This function should return the data at the passed in position. The first element of the data structure starts at position 0.
  - If the passed-in position is outside the bounds of the data structure, the function should throw a **new** `IndexOutOfBoundsException`.
  - *Hint: In your testing main, you should catch* `IndexOutOfBoundsException` *pointers.*
- getSize(): int
  - This is a pure virtual function.
  - This function will return the size of the data structure.
- toArray(): T*
  - This is a pure virtual function.
  - This function will return a perfectly sized dynamic array containing all the elements in the data structure.

  - The elements should be inserted in the same order as they appear in the data structure.
  - If the data structure is empty, an array of size 0 should be returned.

- clone(): DataStructure<T, append>*
  - This is a pure virtual function.
  - This function will return a deep copy of the data structure, populated with shallow copies of the elements in the data structure.

- operator«(out: const ostream&, object: Datastructure<T,append>&): ostream&
  - This is a friend function.
  - This function should return the result of the passed in parameter's `printToStream` function when called the passed-in ostream reference.

- printToStream(outStream: ostream&): ostream&
  - This is a pure virtual function.
  - This function should populate the passed-in parameter with the elements in the data structure as a comma-delimited list.
  - Example: If the data structure contains
    * 1
    * 2
    * 5
    * 3

    The result of using the operator« for printing the list should be:

```
1,2,5,3
```

  - There should be **no** newline at end of the comma-delimited list.

3.4 Array

This class is a wrapper class for a dynamic array of type T. The class publicly inherits from DataStructure. This section will highlight important information related to the array class for the functions described in Section 3.3. If the function is listed in the UML and `h` files but not in the description, it means that no Array specific information is needed and the description in Section 3.3 is sufficient.

3.4.1 Members

- array: T*
  - A dynamic 1D array of type T.

- size: int
  - The size of the array.

3.4.2 Functions

- Array()
  - This is the constructor for the Array class.
  - Initialize the `array` to NULL and `size` to 0.
- ~Array()
  - This is the destructor for the array class.
  - The function should deallocate any dynamic memory allocated to it.
- operator+=(data: T&): DataStructure<T, append>&
  - Along with the description in Section 3.3, the function should dynamically increase the array's size to accommodate the new element.
  - In other words, the array should **not** contain any gaps.
  - The appropriate member variable should also be updated accordingly.
- operator+=(other: DataStructure<T, append>&): DataStructure<T, append>&
  - Along with the description in Section 3.3, the function should dynamically increase the array's size to accommodate the new elements.
  - The appropriate member variable should also be updated accordingly.
- clone(): DataStructure<T, append>*
  - This function should return a deep copy of the current Array class object.
  - Only the array member variable should be deep copied. The elements in the array can be shallow copies.

3.5 Node

This struct is the nodes for the `LinkedList` class. The Node is templatized to take in any type.

3.5.1 Members

- data: T
  - Data stored in the node.
- next: Node<T>*
  - The next node in the LinkedList.

3.5.2 Functions

- Node(data: T)
  - The constructor for the Node class.
  - Initialize the data with the passed-in parameter and set `next` to NULL.

3.6 LinkedList

This class is the linked list class and publicly inherits from the DataStructure class. This section will highlight important information related to the `LinkedList` class for the functions described in Section 3.3. If the function is listed in the UML and `h` files but not in the description, it means that no LinkedList specific information is needed and the description in Section 3.3 is sufficient.

3.6.1 Members

- head: Node<T>*
  - This is the start of the linked list.

3.6.2 Functions

- LinkedList()
  - This is the constructor for the LinkedList class.
  - The function should initialize head to NULL.
- ~LinkedList()
  - This is the destructor for the LinkedList class.
  - The destructor should deallocate any dynamic memory allocated to the class.
- operator+=(data: T&): DataStructure<T, append>&
  - Along with the description in Section 3.3, the function should insert the new node into the list depending on the append template parameter.
- operator+=(other: DataStructure<T, append>&&): DataStructure<T, append>&
  - Along with the description in Section 3.3, the function should insert the new nodes into the list depending on the append template parameter.
- getSize(): int
  - The function should calculate the size of the list, and return it.
- toArray(): T*
  - This function should create a new array and recursively populate the array with the data contained in the list before returning the newly created array.

- clone(): DataStructure<T, append>*
  - This function should return a deep copy of the current LinkedList class object, populated with shallow copies of the data in the linked list.

4 Recursive Restrictions

Please note you may only use recursion in this practical. The usage of any iterative solutions like for loops, while loops, and do-while loops are strictly forbidden.

You will receive a mark of 0 if you use any iterative solutions in the required files. Only in your main.cpp may you use iterative solutions.

You may also add standalone helper functions which you can declare in the relevant cpp file. Do not modify the provided H files.

You are also advised to avoid using for, while and do in your function names, variable names, and comments. If you believe you have fasly been flagged for iterative loops please make a ticket on Discord.

5 Memory Management

As memory management is a core part of COS110 and C++, each task on FitchFork will allocate approximately 10% of the marks to memory management. The following command is used:

```
valgrind --leak-check=full ./main
```

Please ensure, at all times, that your code *correctly* de-allocates *all* the memory that was allocated.

6 Testing

As testing is a vital skill that all software developers need to know and be able to perform daily, 10% of the assignment marks will be allocated to your testing skills. To do this, you will need to submit a testing main (inside the `main.cpp` file) that will be used to test an Instructor Provided solution. You may add any helper functions to the main.cpp file to aid your testing. In order to determine the coverage of your testing the gcov [1] tool, specifically the following version *gcov (Debian 8.3.0-6) 8.3.0*, will be used. The following set of commands will be used to run gcov:

```
g++ --coverage *.cpp -o main
./main
gcov -f -m -r -j ${files}
```

This will generate output which we will use to determine your testing coverage. The following coverage ratio will be used:

$$\frac{\text{number of lines executed}}{\text{number of source code lines}}$$

[1] For more information on gcov please see `https://gcc.gnu.org/onlinedocs/gcc/Gcov.html`

We will scale this ration based on class size.

The mark you will receive for the testing coverage task is determined using Table 1:

Coverage ratio range	% of testing mark
0%-5%	0%
5%-20%	20%
20%-40%	40%
40%-60%	60%
60%-80%	80%
80%-100%	100%

Table 1: Mark assignment for testing

Note the top boundary for the Coverage ratio range is not inclusive except for 100%. Also, note that only the functions stipulated in this specification will be considered to determine your testing mark. Remember that your main will be testing the Instructor Provided code and as such, it can only be assumed that the functions outlined in this specification are defined and implemented.

As you will be receiving marks for your testing main, we will also be doing plagiarism checks on your testing main.

7 Implementation Details

- You must implement the functions in the header files exactly as stipulated in this specification. Failure to do so will result in compilation errors on FitchFork.
- You may only use **c++98**.
- You may only utilize the specified libraries. Failure to do so will result in compilation errors on FitchFork.
- Do not include using `namespace std` in any of the files.
- You may only use the following libraries:
  - string
  - sstream
  - iostream

8 Upload Checklist

The following c++ files should be in a zip archive named uXXXXXXXX.zip where XXXXXXXX is your student number:

- DataStructure.cpp
- Exceptions.cpp
- Array.cpp
- LinkedList.cpp
- `main.cpp`
- Any textfiles used by your `main.cpp`
- `testingFramework.h` and `testingFramework.cpp` if you used these files.

The files should be in the root directory of your zip file. In other words, when you open your zip file you should immediately see your files. They should not be inside another folder.

9 Submission

You need to submit your source files on the FitchFork website (`https://ff.cs.up.ac.za/`). All methods need to be implemented (or at least stubbed) before submission. Your code should be able to be compiled with the following command:

```
g++ *.cpp -o main
```

and run with the following command:

```
./main
```

Remember your `h` file will be overwritten, so ensure you do not alter the provided `h` files.
You have 10 submissions and your best mark will be your final mark. Upload your archive to the Bonus Practical slot on the FitchFork website. Submit your work before the deadline. **No late submissions will be accepted!**

10 Accessibility

Figure 1 contains a class diagram which is explained in the document.