CS 202 Fall 2022 - Assignment 4

Pointers and Dynamic Memory Allocation

Overview

Ah, baseball - as American as apple pie and... *baseball*. In this assignment we'll be keeping track of a baseball league with several teams and learning about dynamic allocation.

In this program, you will be asked to keep track of information about a baseball League, consisting of several Teams that are planned to play against each other. In order to be space efficient, all of the Teams in a League will be kept track of using pointers/references. Essentially, this means rather than keep an array of just Teams, we'll keep an array of references to Teams (So Team*s). Likewise, each Team will keep track of references to all Players on that team (Player*s). This is done so that we can avoid doing excessive allocations of objects. Recall that when declaring a variable, a constructor is always called, however if we want to delay the construction of an object, we can use a pointer and simply wait to make a new object until later with the *new* keyword. Below is an example.

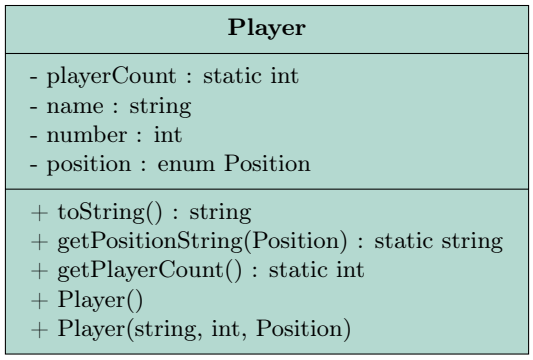
```
1 Player p; //Constructs a Player object automatically and stores it in p
2 Player* pPtr; //Just makes a pointer, no Player
3 pPtr = new Player(); //Explicitly makes a Player and stores a reference in pPtr
```

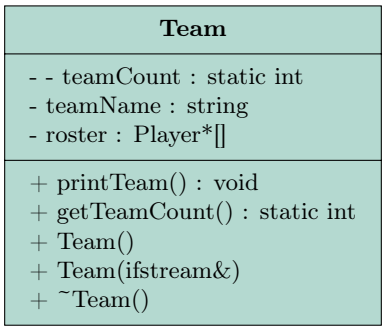
To implement our League, we'll also use classes for the Teams within the League and the Players on each Team. Each Team will contain exactly nine Players - the amount on a normal baseball team. When the amount of entries in an array is known beforehand, that array should be implemented statically - that is the way you may already be familiar with where the size never changes. This is appropriate for our baseball team since there will always be exactly nine Players. The League will also need an array to store some number of Teams. However, the number of Teams in the League may change, so in that case a static array is not appropriate. The League will need to dynamically allocate an array to store Teams into, and as more Teams are added to the League, allocate more space. This implementation will be done similar to that of

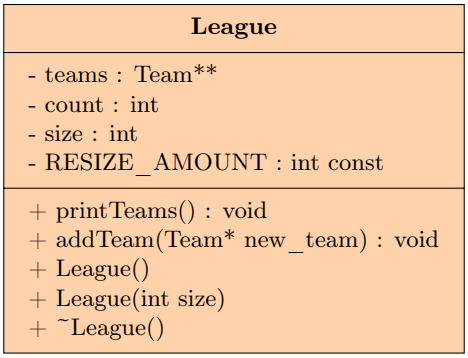
a dynamically allocated list - we will keep a count of how many things are in the array as well as the size of the array currently allocated. When the count reaches the size, a bigger array will need to be made to replace the current array of Teams. Note that an empty array here will be represented with *nullptr* Since an array with no space cannot fit any items in general, the array will also need allocated if the pointer to it is nullptr.

The provided main program will make a League and populate it with a predetermined number of Teams. For the sake of this assignment, main will contain three hard-coded example that can be chosen from for testing purposes.

UML Diagrams







Important Classes and Functions

Below is a list of functions and variables that are important for this assignment. ***Variables are in green***, **functions that you will need to write are in red**, and **functions implemented for you already are in blue**

Globals and Others

- ***const int TEAM_SIZE = 9*** - A constant for how many Players are on one baseball team.
- ***enum class Position*** - An enumeration for the positions in baseball (Catcher, Pitcher, etc.). Enumerations quickly assign integers to easier to read names, so here CATCHER = 0, PITCHER = 1, FIRST_BASE = 2, etc.
- ***PlayerRef*** - A preprocessor directive definition for Player* (a reference to a Player). Basically, you can use this name in place of Player* if it easier for you. Since our team will contain an array of Player*s, you may have an easier time conceptualizing it as an array of PlayerRefs. A reference here just means a pointer to some object that we will use, versus a pointer to a dynamically allocated array.
- ***TeamRef*** - Similar to PlayerRef, but for Team*s. You can use this with the League class if you would prefer to think of it's Team** array as a dynamic array of TeamRefs. PlayerRef and TeamRef are optional to use and your code will compile on CodeGrade regardless of whether you use them or not.

Player

A class representing an individual baseball Player. This will contain some basic information and functions.

- ***static int playerCount*** - Count of how many Players have been constructed. Used mostly to check the output at the end and should only be modified in the constructors.
- ***string name*** - The name of the Player
- ***int number*** - The Player's number on their jersey
- ***Position position*** - The Position that the Player plays
- **string toString()** - Returns the Player's information in a nicely formatted string for printing.
- **static string getPositionString(Position pos)** - Converts a position enum to an easier to read string. Used by the **toString** function for formatting players.
- **static int getPlayerCount()** - Accessor for the ***playerCount*** variable.
- **Player()** - Default constructor that just increments the count of Players made. This should go unused in the solution
- **Player(string name, int number, Position position)** - This constructor should initialize the Player's member variables to the passed parameters and increment the count of Players made. When assigning the variables here, recall how to distinguish between a parameter and a variable member with the same names.

Team

A class representing a team of 9 Players. The Players will be stored via references in this class' roster array.

- ***static int teamCount*** - Count of how many Teams have been constructed.
- ***string teamName*** -The name of the team
- ***Player* roster[]*** - A static array of references to Players. This array is static since it is always going to be of size 9 (i.e. how many Players are on a baseball team).

- **void printTeam()** - Prints the name of the team to cout and then all of the Players on the Team. Use the Player's toString function to get the Player's info as a formatted string.
- **static int getTeamCount()** - Accessor for the teamCount
- **Team()** - Default constructor that just increments the team count. Shouldn't be used in the solution.
- **Team(ifstream& file)** - Constructs a Team by reading it from the passed file. This should read first the Team's name, then each of the nine Players, being all on separate lines. Each Player line will consist of the Player's number and then their name. The positions of the Players is consistently in the order: CATCHER, PITCHER, 1B, 2B, SHORTSTOP, 3B, RF, CF, LF. That is, the first Player in the file is the catcher, the next player is the pitcher, etc.
- **~Team()** - Destructor for the Team. Since our array is statically allocated, we won't need to delete it. However, the Players inside the array are dynamically allocated and will need to be deleted before the array disappears with the Team. This function should go through the roster array of 9 Players and deallocate each one.

League

A League will represent a collection of some number of Teams that all play against each other. Unlike with the array in the Team class that always held 9 Players, the number of Teams that a League holds is variable. Because of this, we will not use a static array but instead will dynamically allocate an array to hold different amounts of Teams depending on how many we want to store. A League will be implemented like a dynamically sized list; we will keep a pointer to an array of Team*s. When adding a Team to the League, it will be inserted to the end of the array if there is space, otherwise we will make space to insert the Team before placing it into the array.

- ***Team** teams* - A 1D dynamically allocated array of references to Teams. This array holds references to all of the Teams that are in our League. Don't be tripped up by the double pointer, this array is only one-dimensional, with the contents of that array being pointers to Teams. IF you're using the TeamRef definition, you could say this is an array of TeamRefs. A type of TeamRef* for the array might also make sense, then.
- ***int count*** - How many Teams are stored in the ***teams*** array.
- ***int size*** - How big the ***teams*** array currently is. In other words, how many spaces have been allocated in the array that the ***teams*** pointer is pointing to.
- ***const int RESIZE_AMOUNT = 4*** - A constant for how much to resize the ***teams*** array by when it needs to be expanded. As an example, if the teams array held five Teams and the array could not hold any more Team*s, the size of the array after allocating new space and deep copying would be nine.
- **void printTeams()** - Prints all Teams in the ***teams*** array by calling their **printTeam** function.
- **void addTeam(Team* new_team)** - Adds the passed new_team to the end of the ***teams*** array. Remember that the size keeps track of how big the array and the count keeps track of how many Team*s are actually in the array, so update and check each accordingly. In the event that there is no space to fit the new_team into the array (either from all spaces being filled or the array not existing yet), a bigger array should be allocated first. When allocating a bigger array, make an array that is ***RESIZE_AMOUNT*** bigger than the size of the existing array, copy over all Team*s from the old array, then deallocate the old array and copy the pointer to of the new array to ***teams***.
- League() - Default constructor that makes an empty League. This will set the ***count*** and ***size*** to 0 and set the ***teams*** pointer to nullptr.
- League(int size) - This constructor will initialize the size of the League's ***teams*** array to the given size. It allocates the amount of space asked for and sets the ***size*** member.

- **~League()** - The destructor for the League class should deallocate not only the Teams that are in the League, but also the dynamically allocated array of ***teams***. Start by deallocating all of the Teams in the array and once they are all cleared out, deallocate the array itself. Remember that there is a difference in the syntax with *delete* when it is used on an object vs an array.

1 Compiling and Testing

This assignment provides a makefile. If you're unfamiliar with a makefile, it can be basically be seen as a series of commands describing how to compile and do important tasks. Here, our makefile basically just compiles the three cpp files and generates an executable named *baseball*. To use the makefile, ensure that it is in the same directory as your source files and then type **make**. Note that **make** is a bash command, so you can only use it on Mac or Linux, not Windows. If at any point it seems that your files may be corrupted and the makefile is not working properly, please try make clean before trying to build again.

This is also the first assignment in which we will be using dynamically allocated memory. There is a program called valgrind that will be used to check your program for memory leaks. You will see the result of it running on CodeGrade, but if you'd like to test your program locally, simply type **valgrind** before the name of the program to run. So here, **valgrind ./baseball**. Valgrind can only tell you when a memory leak occurs and what memory has been leaked, but it will not be able to tell you where the leak occurred exactly in your code. If you have 0 blocks still allocated by the end of the program, then everything is good. Otherwise, try double-checking your destructors, it is possible that memory was not deallocated properly. Also note that if a program crashes, valgrind may report memory leaks since some destructors may not have been called before the program crashed.

TO-DO

The provided main is split up to do three different tests to make debugging easier. It is recommended to start with the Player and Team classes and then check if they seem to work correctly with the first test. Then, when you are confident that those classes are working as intended, try moving on to the League class. The second test can be used to get a general sense if it is working and the third test makes sure that the dynamic resizing works.

Sample Run

Case 1

```
---------- All American Team ----------
C  18 :  Sleve McDichael
P  27 :  Onson sweemet
1B 90 :  Bobson Dugnutt
2B 4  :  Dean Wesrey
SH 17 :  Tim Sandaele
3B 20 :  Todd Bonzalez
RF 86 :  Mike Sernandez
CF 64 :  Dwigt Rortugal
LF 32 :  Kevin Nogilny
```

Case 2

```
---------- All American Team ----------
C  18 :  Sleve McDichael
P  27 :  Onson sweemet
1B 90 :  Bobson Dugnutt
2B 4  :  Dean Wesrey
```

```
SH 17 :  Tim Sandaele
3B 20 :  Todd Bonzalez
RF 86 :  Mike Sernandez
CF 64 :  Dwigt Rortugal
LF 32 :  Kevin Nogilny


---------- The Serpents ----------
C  12 :  Snake
P  2  :  Hyleg Ourobockle
1B 67 :  Kaa
2B 3  :  Rope Snake
SH 9  :  Steel Python
3B 42 :  John Cobra
RF 17 :  Anaconda Jones
CF 18 :  Black Mamba
LF 19 :  Bob Constrictor


---------- Jaguars ----------
C  12 :  John Doe
P  13 :  Jane Doe
1B 27 :  Jim Doe
2B 30 :  Janine Doe
SH 47 :  James Doe
3B 36 :  Julie Doe
RF 94 :  Joran Doe
CF 74 :  Jennifer Doe
LF 32 :  Julian


---------- Red Dragons ----------
C  8  :  Yogi Berra
P  32 :  Sandy Koufax
1B 1  :  Lou Gehrig
2B 42 :  Jackie Robinson
SH 8  :  Cal Ripken Jr.
3B 5  :  Albert Pujols
RF 2  :  Babe Ruth
CF 7  :  Mickey Mantle
LF 25 :  Barry Bonds
```

Case 3

```
---------- All American Team ----------
C  18 :  Sleve McDichael
P  27 :  Onson sweemet
1B 90 :  Bobson Dugnutt
2B 4  :  Dean Wesrey
SH 17 :  Tim Sandaele
3B 20 :  Todd Bonzalez
RF 86 :  Mike Sernandez
CF 64 :  Dwigt Rortugal
LF 32 :  Kevin Nogilny
```

---------- The Serpents ----------
C 12 : Snake
P 2 : Hyleg Ourobockle
1B 67 : Kaa
2B 3 : Rope Snake
SH 9 : Steel Python
3B 42 : John Cobra
RF 17 : Anaconda Jones
CF 18 : Black Mamba
LF 19 : Bob Constrictor

---------- Jaguars ----------
C 12 : John Doe
P 13 : Jane Doe
1B 27 : Jim Doe
2B 30 : Janine Doe
SH 47 : James Doe
3B 36 : Julie Doe
RF 94 : Joran Doe
CF 74 : Jennifer Doe
LF 32 : Julian

---------- Red Dragons ----------
C 8 : Yogi Berra
P 32 : Sandy Koufax
1B 1 : Lou Gehrig
2B 42 : Jackie Robinson
SH 8 : Cal Ripken Jr.
3B 5 : Albert Pujols
RF 2 : Babe Ruth
CF 7 : Mickey Mantle
LF 25 : Barry Bonds

---------- All American Team ----------
C 18 : Sleve McDichael
P 27 : Onson sweemet
1B 90 : Bobson Dugnutt
2B 4 : Dean Wesrey
SH 17 : Tim Sandaele
3B 20 : Todd Bonzalez
RF 86 : Mike Sernandez
CF 64 : Dwigt Rortugal
LF 32 : Kevin Nogilny