1 Submission Instructions

Submit to Brightspace on or before the due date a compressed file (.zip) that includes

1. Header and source files for all classes instructed below.
2. A working Makefile that compiles and links all code into a single executable. The Makefile should be specific to this assignment - do not use a generic Makefile.
3. A README file with your name, student number, a list of all files and a brief description of their purpose (where necessary), compilation and execution instructions. In addition, if you have done anything different from the specification, please detail it here. You may introduce functions, or change functions, as long as the tests still run correctly, and you do not use any libraries other than those permitted (`iostream, string, sstream, iomanip`)

2 Learning Outcomes

In this assignment you will make a basic CRUD (create, read, update, delete) application. You will learn the basics of classes, arrays, and static memory in C++. You will make a few simple classes, populate the members using constructors and write functions to process the data. You will learn how to provide a working Makefile, and implement a small amount of application logic.

3 Overview

You will make an app that manages reservations for a `Campground`. A `Campground` consists of `Campsites`. Each `Campsite` has a `Category`, which is an `enum` consisting of entries such as `tent` or `cabin`, and `Campsites` can be reserved by `Campers`. To manage the reservations, and ensure that we do not have `Campsites` reserved to more than one group on a given day, you will modify the `Date` class that we saw in class.

We will be implementing a basic back-end of the application, but not a user-facing interface. Instead of a user-facing interface you will be supplied with a test suite for your application.

4 Classes Overview

This application will consist of 4 classes, and one `Tester` class. In addition it will have header and source files `Category.h` and `Category.cc`, as well as `defs.h`. A brief description of each class and the given files follows.

1. `Date` - will store `Date` information and functions, modified to better manage `Camper` reservations.
2. `Camper` - an entity class containing information about someone who has reserved a `Campsite`.
3. `Campsite` - an entity class containing information about individual campsites, as well as a list of `Campers` staying at that site, organized by `Date`.
4. `Campground` - the main class. This class provides functions to add, remove, and print `Campsites` and `Campers`.
5. `Tester` - some functions used to test your code.
6. `Category.h`, `Category.cc` - compile these into `Category.o` and link to your executable. These contain a `Category` enum and a global function to convert a `Category` enum to an appropriate string, both in the `cat` namespace.
7. `defs.h` - this has a preprocessor constant that you may use to initialize your arrays to a consistent size.

All your classes should all be in separate header and source files that are compiled explicitly in your Makefile!!. In addition you should compile and link the `Tester` class into your executable.

5 Instructions

Download the starting code from Brightspace. All member variables are `private` unless otherwise noted. All member functions are `public` unless otherwise noted. Some return values are not explicitly given. You should use your best judgment (they will often be `void`, but not always). ALL CLASSES MUST HAVE A PRINT FUNCTION UNLESS OTHERWISE NOTED. This prints metadata about the class like names or dates but not data contained in a data structure like an array (unless explicitly specified).

All other objects should be passed by reference with one exception. You are allowed to pass `string` objects by value *for this assignment only*. Passing by `string&` does not work with string literals (and we will learn why in the lessons on **Encapsulation**). You may use `string` or `const string&`. For the purposes of this assignment they function the same, but the second version is more efficient, as we will learn.

Your finished code should compile into a single executable called `a1` using the command `make a1`, `make`, or `make all`, using a Makefile that you wrote yourself. Your submission should consist of a single zip file with a suitable name (e.g., `assignment1.zip`) that contains a folder containing all your files. This folder should also contain a README with your name, student number, a list of all files that are included. The main thing it should include is **any changes to the specification**. I don't mind when you do things differently - I would encourage it, as long as it is code you write yourself, and not outside libraries - but you *must* document these changes.

All data in the assignment must be statically allocated. That means no `malloc`, no `new`, and you will lose marks if there are pointers. Use references instead.

5.1 Notes on Automated Marking

Initially `main.cc` will not compile because it references classes and functions that do not exist. Feel free to comment out parts that are not working. Or you can write your classes with empty implementations that you complete as you go (which I would recommend, but it is up to you). You may also `#include` additional header files for classes other than `Campground`. You may make your own file with your own `main` function and write your own tests, and compile that separately. But the code provided in `main.cc` using the `Tester` class is what will be run to determine your **application requirements** mark.

Most of your bugs will be in your code. However, some of you will likely find bugs or inconsistencies in the provided test code as well. Please bring them to my attention.

When there is a conflict between this specification and the test code, the test code takes precedence. A rule of thumb:

> *Working applications are always better than applications that don't run, even if you break the rules. It is better to break a rule, or ignore the specification, and submit a working application than to submit something that doesn't run but follows all the instructions.*

The specification is just a guide. Your assignment is to produce a running program that passes the tests, while following the specification *as closely as possible*. Making it run correctly, and do the things it is supposed to do is the highest priority!

5.2 The defs.h file

This file defines some handy preprocessor constants so that these values are consistent across our application. In this case it defines `MAX_ARRAY` which you should use as the size to initialize your arrays to.

5.3 Category.h and Category.cc

These files are done for you. They contain a `Category` enum and a `categoryToString(Category c)` global function to convert a `Category` enum to an appropriate string (you will use this in your `print` functions). Both the enum and the function are in the `cat` namespace. You will need this in your `Campsite` class.

5.4 The Date Class

You are given a `Date` class. To that you will add the following (suggested) members.

1. Member functions:
   (a) A `bool equals(Date& d)` function. This should return `true` if the `Date` in question and `d` are equal in value and `false` otherwise.
   (b) A `bool lessThan(Date& d)` function. This should return `true` if the `Date` in question comes before `d` and `false` otherwise.

5.5 The Camper Class

The `Camper` class contains the information of someone who has reserved a `Campsite` including the `Dates` they have reserved.

1. Data members:
   (a) A `string`: name. For the purposes of this assignment we will assume that the name serves as a unique identifier, i.e., no two `Campers` have the same name. You do not have to check for this - you can assume each name given as input is unique.
   (b) A `string`: plate_number. This has the license plate of the `Camper`'s vehicle.
   (c) `int num_people`. The number of people staying at the `Campsite`. You should ensure this is a number is at least 1 and at most the `maxPeople` parameter of the `Campsite` they are staying at.
   (d) `Date check_in` and `Date check_out`. These are self-explanatory. Ensure that check-out is at least one day later than check-in.
2. Make a constructor that takes the information in the order and type given and initializes the member variables.
3. Make a no-argument constructor that initializes the member variables with dummy data.
4. Member functions:
   (a) A `print` function that prints out the `Camper` data, including the check-in and check-out dates.
   (b) Any other function you might need (you will almost certainly need more functions).

5.6 The Campsite Class

The `Campsite` class stores data about the campsite, as well as an array of `Campers` who have reserved the site.

1. Data members:
   (a) `int site_number;`
   (b) `Category category;`
   (c) `string description;`
   (d) `int maxPeople;`
   (e) `double price_per_day;`
2. Make a constructor with the parameters given above in the order given, and use these parameters to initialize your `Campsite` member variables. You should do routine bounds checking, i.e., make sure the `price` is not 0 or negative and make sure `maxPeople` is at least 1.

3. In addition you will need the following to keep track of `Campers`;
   (a) A statically allocated array of `Camper` objects with size `MAX_ARRAY`.
   (b) You should also make an `int` member that tracks the number of `Campers` in the array.
4. Member functions:
   (a) Make a `bool addCamper` function that takes a name, plate number, number of people, check-in date and check-out date as parameters, in that order. This should add a `Camper` to the `Camper` array according to the instructions given below. Return `true` if adding was successful and `false` otherwise.
   (b) Make a `removeCamper(string name)` function. This should remove a `Camper` from the array if the `name` parameter matches the camper name.
5. You will make 3 different `print` functions.
   (a) Make a `print()` function. This function should print all the `Campsite` data nicely formatted. For example:

```
Site Number: 5
Category: lodge
Description: Lake front
Max People: 5
Price per Day: $50.00
```

   (b) Make `printCampers()` function. This should print all `Campers` in the `Camper` array.
   (c) Make a `printCamper(Date& date)` function. This should print the `Camper c` for which `date` parameter is equal to or later than `c.check_in`, but before `c.check_out`. If no such `Camper` exists, print out an appropriate message.

   To help clarify the behaviour, if we think of `date`, `check_in` and `check_out` as integers, then print the `Camper c` for which `c.check_in <= date` and `date < c.check_out`.

Rules for storing `Campers`: the `Campers` should be stored in the order by `Date` that they are using the `Campsite`. There should never be two `Campers` using the `Campsite` on the same `Date`. The exception is that the `check_out Date` of one `Camper` can be equal to the `check_in Date` of the next `Camper` (one `Camper` moves on to the site on the same day as another `Camper` leaves the site - this is allowed).

In addition, all `Campers` should be stored in the array consecutively (there should be no gaps in the array).

5.7 The Campground Class

The `Campground` class will maintain an array of `Campsites`, as well as functions related to adding, removing, and displaying `Campsites` and `Campers`. A vanilla print function is not necessary - we will make multiple specialized print functions instead.

1. Member variables:
   (a) Make a *statically allocated array* of `Campsite` *objects*.
   (b) Make sure you track the number of elements in the above array.
2. Make a default (no-argument) constructor that initializes all member variables.
3. Member functions:
   (a) `addCampsite`: The parameters of this function are the same types in the same order as the `Campsite` constructor. If there is room in the `Campsite` array, and the `site_number` is unique, add the `Campsite` to the back of the array. Print out an appropriate message to `cout` whether successful or not.

(b) `removeCampsite(int site_number)`: If there is a `Campsite` with the given `site_number`, remove it from the array. In either case, print out an appropriate message to `cout`.

(c) `addCamper(int site_number, ...)`: Instead of `...`, the remaining parameters should be the same type and order as the `Camper` constructor. Find a `Campsite` with the given `site_number` and add a `Camper` with the given parameters to it. Print out a message to `cout` telling the user that the process succeeded, or else describe where it failed.

(d) `removeCamper(int site_number, string name)` - attempt to remove the `Camper` with the given name from the `Campsite` with the given number. Print out a message on `cout` describing that the operation succeeded, or else describe where and how it failed.

4. `print` functions for displaying data:

(a) `printCampsites()` - `print` all `Campsites` at this `Campground`.

(b) `printCampers(int site_number)` - print all `Campers` at the `Campsite` with the given `site_number`.

(c) `void printCampers(Date& date)` - print all `Campers` on any `Campsite` on the given `date`. The rules for this are the same as the rules described for `Campsite::printCamper(Date& date)`.

(d) `void printCampsitesByCategory(Category category)` - print all `Campsites` in the given `category`.

6 Grading

The marks are divided into three main categories. The first two categories, **Requirements** and **Constraints** are where you earn marks for making a working application that was done correctly. The third category, **Deductions** is where you are penalized marks.

6.1 Specification Requirements

These are marks for having a working application (even when not implemented according to the specification, within reason). These are the same marks shown in the test suite, repeated here for convenience. Marks are awarded for the application working as requested.

You are still responsible for, and may be penalized for, any errors the test suite does not catch, or any drastic departure from the specification (such as using outside libraries). We reserve the right to modify the mark given by the test script in these cases.

General Requirements

- All marking components must be called and execute successfully to earn marks.
- All data handled must be printed to the screen as specified to earn marks.

Application Requirements: 31 marks

1. (4 marks) Test print Campers
2. (3 marks) Test print Campsites
3. (7 marks) Test Campsite add, remove, and print Campers
4. (8 marks) Test add, remove, and print Campsites in Campground
5. (9 marks) Test add, remove, and print Campers in Campground

Requirements Total: 31 marks

6.2 Constraints

The previous section awards marks if your program works correctly. In this section marks are awarded if your program is written according to the specification and using proper object oriented programming techniques. This includes but is not limited to:

- Proper declaration of member variables (correct type, naming conventions, etc).
- Proper instantiation and initialization of member variables (statically or dynamically).
- Proper instantiation and initialization of objects (statically or dynamically).
- Proper constructor and function signatures.
- Proper constructor and function implementation.
- Proper use of arrays and data structures.
- Passing objects by *reference* or by *pointer*. Do not pass by value.
  - **Note:** For this assignment only, `strings` may be passed by value.
- Proper error checking - check array bounds, data in the correct range, etc.
- Reasonable documentation (remember the best documentation is expressive variable and function names, and clear purposes for each class).

6.2.1 Constraint marks:

1. 2 marks: Proper implementation of the `Date` class.
2. 2 marks: Proper implementation of the `Camper` class.
3. 2 marks: Proper implementation of the `Campsite` class.
4. 2 marks: Proper implementation of the `Campground` class.
5. 3 marks: Proper error messages in the `Campground` class.

Constraints Total: 11 marks

6.3 Deductions

The requirements listed here represent possible deductions from your assignment total. In addition to the constraints listed in the specification, these are global level constraints that you must observe. For example, you may only use approved libraries, and your programming environment must be properly configured to be compatible with the virtual machine. This is not a comprehensive list. Any requirement specified during class but not listed here must also be observed and may be penalized if done incorrectly.

6.3.1 Packaging and file errors:

1. 5%: Missing README
2. 10%: Missing Makefile (assuming this is a simple fix, otherwise see 4 or 5).
3. up to 10%: Failure to use proper file structure (separate header and source files for example), but your program still compiles and runs
4. up to 50%: Failure to use proper file structure (such as case-sensitive files and/or Makefile instructions) that results in program not compiling, but is fixable by a TA using reasonable effort.
5. up to 100%: Failure to use proper file structure or other problems that severely compromise the ability to compile and run your program.

As an example, submitting Windows C++ code and Makefile that is not compatible with the Linux VM would fall under 4 or 5 depending on whether a reasonable effort could get it running.

6.3.2 Incorrect object-oriented programming techniques:

- Up to 10%: Substituting C functions where C++ functions exist (e.g. don't use `printf`, do use `cout`).
- Up to 25%: Using smart pointers.
- Up to 25%: Using global functions or global variables other than the `main` function and those functions and variables expressly permitted or provided.

6.3.3 Unapproved libraries:

- Up to 100%: The code must compile and execute in the default course VM provided or Openstack. It must NOT require any additional libraries, packages, or software besides what is available in the standard VM or Openstack.
- Up to 100%: Your program must not use any classes, containers, or algorithms from the standard template library (STL) *unless expressly permitted.*

Assignment Total (Requirements and Constraints): 42