Homework 3

Instructions

- The assignment is to be completed individually and cannot be solved in groups. You are allowed to receive help from your peers to better understand the material. Plagiarism, or any other form of copying will result in an automatic zero for all involved parties.
- Late assignments will be deducted 10-points per-day and will not be accepted after 5 days.
- You are to upload a single zipped file in form of `your-id.zip` that contains only the source code of your project. Do not upload visual studio's `sln` files.
- You are not allowed to include any third-party libraries or use any `cpp-addons` unless you receive explicit permission from the instructor.
- Points will be granted for both the correctness of the code as well as its readability and expressiveness.
- Bonuses will not be allotted to students who's overall assignment grade is less than 85%.

Background

The following assignment will be to simulate a filesystem that manages various file types on a disk. The questions are structured so that the whole system will be created incrementally. There are various types of files as well as different types of filesystems, each of which has a unique characteristic.

- You are allowed to add any helper methods or properties to the various classes. If additional classes are useful, feel free to create them.
- The encapsulation (`public/protected/private`) of the various properties and methods are up to your discretion but should be logical.
- The use of `const` as well as the `parameters/arguments` a method receives, are also up to your discretion.
- In the event of failure, the program should exit with a `status-code` of `55`.
- The cleanup of dynamically allocated memory is required, however, the choice of which class is responsible of doing so is also up to your discretion.
- You can utilize the `ADT` structures from the previous assignment or the built-in `std::string`, and `std::vector` classes.
- There should be no input or output (`std::cin` or `std::cout`) with the exception of the `main()` function, the `Menu` class, and the `operator<<()` or `operator>>()` methods.

Question 1 — *File* *(20 points)*

File	
A `File` is an `abstract class` that stores the various properties that are common among file types. You must make the decision of which method should be considered `pure virtual`.	
`name()`	The name of the file, including the file extension.
`extension()`	The file extension type, if one exists.
`size()`	The size of the file in bytes.
`created_at()`	The time and date the file was created.
`updated_at()`	The time and date the file was updated.
`update(...args)`	Updates the internal properties of the file *(the amount of arguments can be different depending on the concrete implementing class)*.

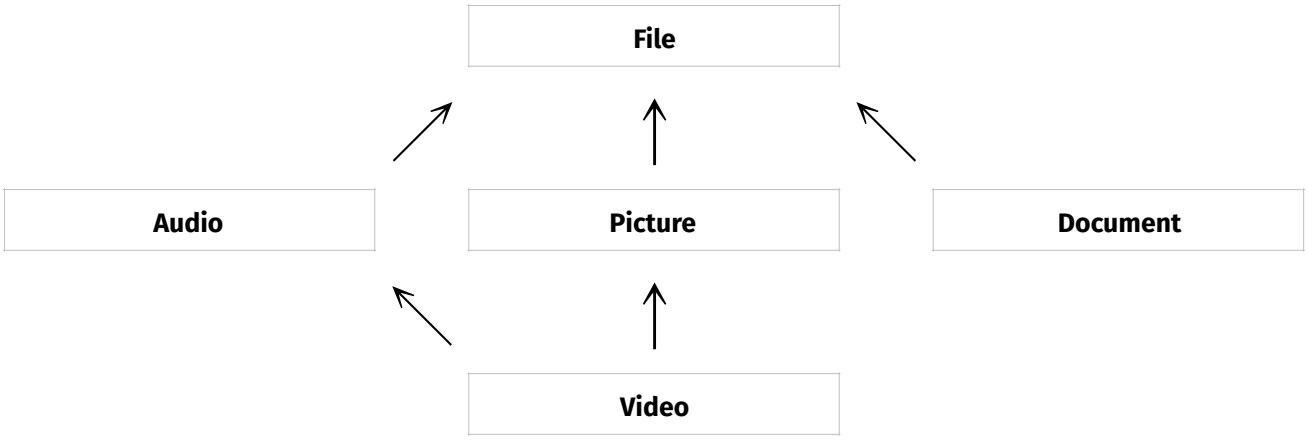
There can be different types of concrete files, each of which have their own properties in addition to their inherited (`File`) parent:

Audio	
`format()`	The music encoding *(i.e. mp3, wav, acc, etc.)*. This can be different from the file extension.
`bitRate()`	The quality of the file in bitrate *(i.e. 128kbps, 160kbps, etc.)*
`length()`	The length of the file in seconds.
`length(true)`	A human readable format of the file length *(i.e. 2:08:01)*.
`size()`	The size of the file based on its length and bitrate.

Photo	
`format()`	The photo encoding *(i.e. jpg, png, gif, etc.)*. This can be different from the file extension.
`dimension()`	The photo dimension in pixels *(i.e. 1920x1080)*.
`size()`	The size of the file in bytes based on its dimensions.

Video	
`format()`	The video and audio encoding *(i.e. mp4:acc, mkv:acc, m4p:wav, etc.)*. This can be different from the file extension.
`dimension()`	The video dimension in pixels *(i.e. 1920x1080)*.
`length()`	The length of the file in seconds.
`size()`	The size of the file based on its length, dimensions and audio bitrate.
`isHD()`	Whether or not the file is considered HD.

Document	
`application()`	The name of the application used to open the file.
`owner()`	The name of the file creator and owner.
`size()`	The size of the file based on its length and dimensions.
content()	Returns the content *(a* `string`*)* of the file.



Bonus *(3 points)*

Implement the `move-constructor` *(i.e.* `Audio(const Audio&& other)`*)* for each of the classes above.

Bonus *(3 points)*

Create a global `clone(const T& file)` function that uses templating and acts as a virtual copy-constructor for the classes above.

Question 2 — *FileSystem* *(35 points)*

FileSystem	
A filesystem is an `abstract class` that contains a collection of files of various types stored as a collection of the type `File`. Our filesystem is different than the one we are used to because it doesn't have any depth. Meaning, there are no folders that contain files, rather all the files exist in the root of the object. You must make the decision of which method should be considered `pure virtual`.	
`name()`	The name of the partition *(or disk)* that identifies the filesystem.
`capacity()`	The total capacity of the filesystem in bytes.
`capacity(true)`	The total capacity of the filesystem in human readable form.
`available()`	The amount available bytes on the filesystem.
`available(true)`	The human readable amount.
`all()`	Return a printable list of files on the system in ascending order.
`all(true)`	Return a printable list of files on the system in descending order.
`put(filename, file)`	Add a file to the filesystem, if there is enough space.
`remove(filename)`	Remove a file from the filesystem.
`exists(filename)`	Checks if a file exists on the system.
`touch(filename)`	Update the updated timestamps of a file.
`get(filename)`	Get a file by its name.

There can be different types of concrete filesystems, each of which have their own properties in addition to their inherited parent:

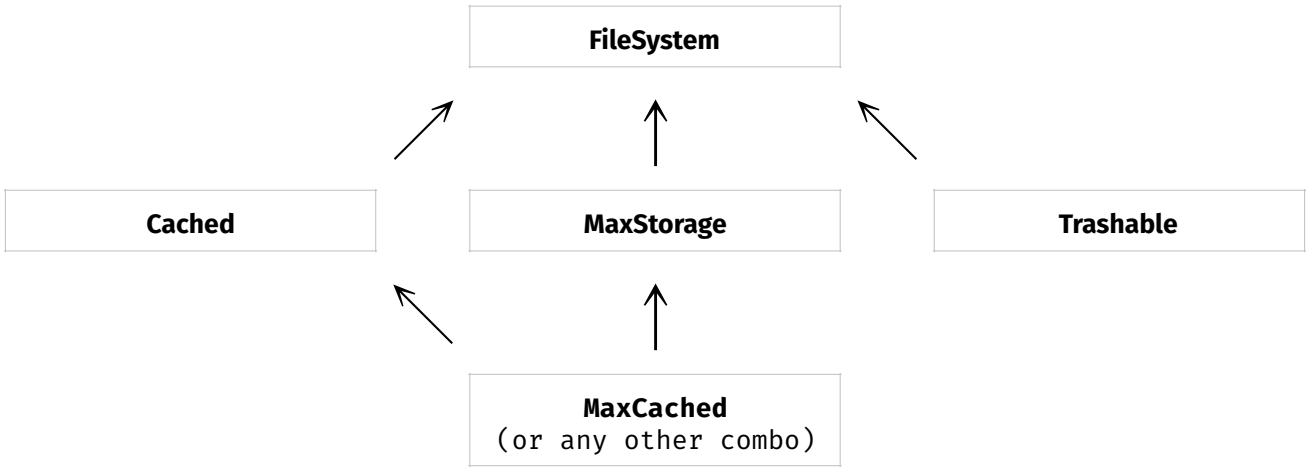
Encrypted	
The encrypted filesystem encrypts all file fields by changing each letter in any field to their representation in `ascii`. For example, if the file name is `"someFile.txt"`, then the encrypted version would be the string `"115 111 109 101 070 105 108 101 046 116 120 116"`.	
`all()`	Return an encrypted printable list of files on the system in ascending order.
`all(true)`	Return an encrypted printable list of files on the system in descending order.
`get(filename)`	Get a file by its actual name, and not the encrypted version of the name. The however, remains encrypted.
`decrypt(filename)`	Get a file by its actual name, and not the encrypted version. The file content is decrypted.
`remove(filename)`	Remove a file by its actual name, and not the encrypted version, from the filesystem.
`exists(filename)`	Checks if a file exists on the system using its actual name, and not the encrypted version..
`touch(filename)`	Update the updated timestamps of a file by its actual name, and not the encrypted version.

Cached	
The cached filesystem only stores files for a maximum amount of time *(in seconds)* from their last update. The `maximum` amount of time is provided to the object during its construction. Files that should be removed after the maximum time has elapsed are removed from the system when the user calls the next method *(i.e.* `touch()`, `put()`, `get()`, *etc.)* on the system.	
`clear()`	Remove items that have exceeded the maximum amount of time.

MaxStorage	
This filesystem only stores files that have a size less than a predefined maximum that is provided during the construction of the object.	
`reallocate(maximun)`	Can be provided a new maximum.

Trashable	
This filesystem places items in the trash instead of deleting them when `remove(filename)` is called.	
`empty()`	Removes all trashed files.
`trash()`	Return a printable list of files in the trash in ascending order.
`restore(filename)`	Return a file to the filesystem from the trash.

In addition to the filesystems above, you are to create a class that inherits *(diamond or multiple inheritance)* from two or more of the filesystems above. For example, `MaxCached` inherits from both `MaxStorage` as well as `Cached`.



Bonus *(3 points)*

Implement the creation, removal, searching, and updating of a file using operator overloading. For example, `get(filename)` should utilize `operator[](const char* filename)`, `put(filename, file)` should utilize `operator+=()` etc.

Question 3 — StorageManager *(20 points)*

StorageManager	
A computer can contain many partitions, each of which can have its own filesystem. You are to implement a `StorageManager` that can contain any number of `FileSystems`.	
`name(filesystem)`	Retrieves a filesystem by its name.
`capacity()`	The total capacity of all the filesystem in bytes.
`capacity(true)`	The total capacity of all the filesystem in human readable form.
`available()`	The amount available bytes on all the filesystem.
`available(true)`	The human readable amount.
`all()`	Return a printable list of filesystems in ascending order.
`all(true)`	Return a printable list of filesystems in descending order.
`destroy(filesystem)`	Remove a filesystem from the manger.
`exists(filename)`	Checks if a file exists on any of the filesystem.
`get(filesystem)`	Get a filesystem by its name.
`put(filesystem)`	Add a filesystem to the manager.

Bonus *(3 points)*

Save the various filesystems using the `std::map` object *(http://www.cplusplus.com/reference/map/map/map/)*. The `key` should be the name of the filesystem and the `T` should be the `FileSystem` object.

Question 4 — Menu *(25 points)*

You are to create a `Menu` object that will allow the user to manage files and filesystems. The menu is responsible for displaying the options and obtaining choices from the user to interact with all the objects described above. The menu must use `RTTI` to differentiate between the various files and filesystem types that have been created.

For example, when the program first begins to execute, the user will be prompted with the menu below:

```
*****************************************************************
*                Welcome to the StorageManger.                  *
*****************************************************************
What would you like to do (choose from the options below):
   1. Create a new filesystem
   2. Remove a filesystem
   3. Open a filesystem
   4. …
   5. …
   6. Quit and go back to the previous menu
```

Each menu should be a separate function within the menu class. You can structure the logic within the menu class as you please. **Remember**, the responsibility of obtaining user input is that of the menu, therefore, menu should obtain data from the user and then apply the data to the objects above using their methods methods. For example:

```
Menu::createFileSystem() {
   FileSystem* filesystem;
   std::cout << “Which filesystem would you like to create?”;
   // ...
   std::cin >> choice;
   // ...
   std::cout << “Enter a name for the filesystem;
   std::cin >> name;
   // ...
   filesystem = new Encrypted(name);
   // ...
   this→storageManager→put(filesystem)
}
```

Bonus *(3 points)*

Use the built-in `std::right`, `std::left`, `std::setw()`, `std::setfill()`, etc. from the `iomanip` *(https://en.cppreference.com/w/cpp/io/manip/)* library to properly layout and structure the printing of the menu.

Bonus *(6 points)*

Bonus 1 *(3 points)*

Debug your program using the `gdb` debugger *(https://www.gnu.org/software/gdb/documentation/)* and attach a screenshot of your process. Briefly explain your steps of debugging underneath the screenshot.

Bonus 2 *(3 points)*

For students using `UNIX`, add some color to your output using `ANSI` color codes *(https://en.wikipedia.org/wiki/ANSI_escape_code)*.

For example, `std::cout << "Not red" << ansi::red << "Red text" << ansi::end << std::endl;`