Concepts of C++ Programming
summer term 2022

Technical University of Munich
CIIP Group

Exercise sheet 6 2022-12-15

Due date: 2022-12-22 16:59

The goal of this exercise sheet is to get you used to inheritance in C++ , as well as other nuances of object-oriented programming.

To receive the point, you need to pass the CI pipeline, i.e. all tests must pass in the CI for both Clang and GCC.

We'll implement a very simple file management system, which supports 4 types of files:
Documents, Audio, Images and Videos.

Exercise 1:

Implement file content storage in `filecontent.cpp`. `FileContent` is meant to be read-only - so once created, file contents can't be changed any more. If one wants to change file contents, a new `FileContent` object has to be created to replace the old one.

Implement the `class FileContent` and extend it so it can store a `std::string`.

When a `FileContent` object is copied, the stored `std::string` must be shared among both instances. Likewise, when a `FileContent` object is moved into another `FileContent`, the `std::string` must not be copied, but moved into the other object.

Since this use-case is very common, there probably exists a very simple solution to this problem.

Exercise 2:

We now implement a hierarchy of `File` classes. The base class (`file.h`) defines the interface available for all files, but each specialized file sub-class can handle requests differently. Two things are common for all files: The file `name`, and the file content (`FileContent`) we implemented in the previous task.

We assign a string "type identifier" for each file sub-class, which is returned by `get_type`.

The "real" allocated/used space as occupied in memory of a file is returned by `get_size`.

Each file could support different modes of compression and storage. To calculate the "raw", uncompressed file size in bytes, a file only considers its metadata (resolution, sample rate, ...), and delivers this size via the `get_raw_size` method.

Because each file has unique properties stored in its member functions, each file has its own `update` function, which is used to replace the file content and file metadata by `moving` in new values.

We support at least four types of files:

- `Document` (`document.h`), whose type identifier is `"DOC"`. It has a member function `get_character_count` which returns the number of non-whitespace characters (so all except `' '`, `'\n'`, `'\t'`) in the file content. The `get_raw_size` equals the allocated content size.

- `Audio` (`audio.h`), which has type `"AUD"`. It stores its `duration` which determines the raw size: We assume 16 bit audio samples at a rate of 48000 Hz, for two channels, times the `duration`.
- `Image` (`image.h`), identified by `"IMG"`. It stores its `resolution`, which is used to calculate the raw size: we have, for each pixel, 4 color values storing 8 bit each (RGBA).
- and `Video` (`video.h`), denoted with `"VID"`. It also stores its `resolution` and the video `duration`. Both are used for the raw size calculation of an 8-bit RGB video at 30FPS: $3 \times \text{resolutionX} \times \text{resolutionY} \times \text{to\_int}(30 \times \text{duration})$
- If you want to implement more and play around, go ahead!

Exercise 3:

Define the `class Filesystem` which shall own and manage files. This allows us looking up, renaming and removing files.

In order to store a file of any sub-class type, we have to store each file as pointer.

Since a file should not vanish and deallocate while we might use it, we use `std::shared_ptr`. The `std::shared_ptr` works pretty much how Python manages all of its objects. For this exercise, it is a good fit, but your default choice of smart pointer should be `std::unique_ptr`.

All features of the file system are already declared in `filesystem.h`, you now need to implement them:

- Registering new files by a name
  - This transfers the ownership of a file to the `Filesystem`
- Deleting a file by name
- Renaming a file with source and destination name (`filesystem.rename(source, destination)`)
- Renaming a file with its handle directly (`file.rename(newname)`).
  - for this to work a file must ask the filesystem it was registered to to do the rename request
  - therefore you need to add a way for the file to reach its filesystem
  - a useful tool to get a pointer to the current object may be `std::enable_shared_from_this`
- counting the amount of files
- sum up the sizes of all files
- return a vector of all files that have a size in a given range

Some helpful resource to read for `enable_shared_from_this`:

- nextptr: enable_shared_from_this - overview, examples, and internals
- cppreference: enable_shared_from_this