Programming Exercise: Asynchronous task library

Introduction

The exercise has two components:

1. Write a C++ library that allows the user to manage an arbitrary number of asynchronous tasks. The library should allow users to schedule tasks to be run asynchronously and pause, resume and stop them. The library should also allow the user to query the status of a given task.
2. Write an example command line application that allows the user to launch and control tasks. The tasks run by the example can be hard-wired.

You must use standard C++ (C++11 or higher). You may also use boost or Qt libraries.

This exercise is meant to show various abilities we are interested in:

- Approach to design: The code should be easily usable both by a text-only application and by an hypothetical GUI application. It should be extensible, and usable with minimal coupling.
- Approach to testing.
- Approach to robustness.
- Care for software maintanibility, showed by attention to simplicity, testability and portability.

Development environment

You can use a development environment and operating system of your choice to develop the code. However, your code should build and run on a POSIX platform (for example: Linux, MacOS). Code should use standard C++. Code must be buildable from the command-line with either cmake, make or scons. It is not acceptable to require an IDE to build, although any IDE can be used for development.

Deliverables

A zip archive with source code, tests and build instructions and configuration to allow for simple building and testing.

1. Source code of library and example executable
2. Unit tests in C++, pick the test framework you prefer. If no preference, pick google test.
3. Version control history (.git folder or equivalent - git preferred)
4. Build scripts / config files (e.g. cmake, Make files etc.)
5. Readme file explaining how to build and use the code

Note: It is acceptable to list the requirements and leave it up to the user to install them using a package manager or otherwise. But the build configuration should make it easy to find the dependencies if installed.

Requirements

- It should be possible to safely pause, resume, stop a task.
- The example program should accept the following arguments:
  - `./program --help` prints help message and instructions
  - `./program` starts and waits for instructions
- Once the program is running, it should read instructions from the standard input. These instructions should have the following format:
  - `start` starts a task and prints its ID
  - `start <task_type_id>` (optional, nice to have) If more than one task type is supported, starts a task of a given type and and prints its ID
  - `pause <task_id>` pause the task with the given id and print a confirmation message
  - `resume <task_id>` resume task with the given id (if paused) and print a confirmation message
  - `stop <task_id>` stop the task with the given id (if not stopped) and print a confirmation message
  - `status` prints the id, the status (paused, running, stopped, completed) and an optional indicator of progress for each task. If the application supports multiple task types, prints the task type ID.
  - `status <task_id>` As above, but for a single task.
  - `quit` gracefully shut down

Definitions

- Task: a close-ended piece of work. Close ended means it must have a well-defined completion. It should not be, for example, a server or a daemon process. A task could be a simple function that prints a random number to a log file, or that reads a text file and counts the number of words.
- Task ID: a unique identifier for a task instance
- Task type ID: a unique identifier for a class of task

Task status values and state changes

- paused: the task has been paused. It can be stopped or resumed.
- running: the task is active. It can be paused or stopped. It can reach completed state.
- stopped: the task has been stopped. It cannot change state.
- completed: the task has completed its execution. It cannot change state.