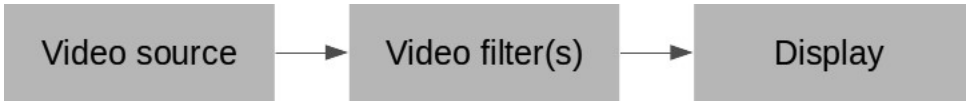
Question 3

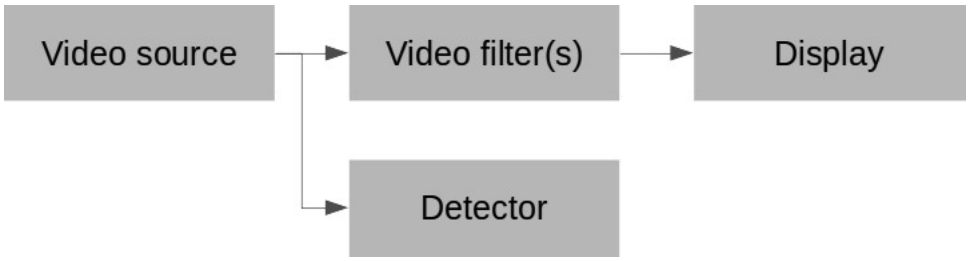
Introduction

The theme of this C++ coding test is to build a program emulating the behavior of a small video pipeline. A video pipeline can be seen as various elements linked together on which video frames will flow and be processed. Each element is usually dedicated to a specific processing task like: frame generation, video format conversion, image filtering, detection algorithm, etc.

A typical hierarchy can be as this:



A variant with some branching:



Scope of this coding test is to write the needed C++ objects that will emulate a basic video pipeline containing:

- A video source generating random video frame of two colors pixels.
- A detector that will find some pattern in the video frames.
- An output renderer to display the video frame (on stdout with one ascii character for each pixel color).

A few guidelines to follow during the test:

- Full C++17 at your disposal, feel free to use it
- Use C++ inheritance to factorize your code if relevant
- Use C++ containers if you need to hold references or store data
- Comment your code in English
- No need for a complex Makefile, a simple g++ or clang++ command is perfectly fine to test your code
  Ex: clang++ -Wall -Werror -std=c++17 *.cpp -o pipeline

Part 1: Base C++ components

Before implementing the pipeline elements, write some C++ object which will be your base components:

- An object that will represent video frames flowing into the pipeline. They will be generated by your first element and move across all downstream ones, possibly shared between multiple element branches. It should contain at least 3 members:
    a. Width in pixels
    b. Height in pixels
    c. Data container holding the pixels values
- Write a C++ interface that will be the base of your pipeline elements providing:
    a. A **link** function to associate the element to a downstream element in the pipeline. This function can be called multiple time on the same element to associate it with multiple downstream branches.
    b. A **process** virtual function to be implemented by inheritors. It receives a video frame and process an operation on it.
    c. A **processAndPushDownstream** function that will receive a video frame, call the **process** method of the element and then call **processAndPushDownstream** of each linked downstream elements.

Part 2: Source and output elements

On this part focus on providing the two minimum elements that will allow the pipeline to run: a source and an output.

- Write an element that will act as a fake video source. It should have a function to start to generate video frames. Width, height and rate at which the frame are produced should be configurable (sleep is fine for naive regulation of the frame rate). Fill the frame data with random values. Use two possible values: (0/1, true/false or space/dot ascii characters). Each generated frame should be pushed downstream with **processAndPushDownstream** of linked elements.
- Write an element that will act as your display. It can be a simple output on stdout of each pixel value if you associate it to a printable ascii character.
  On a Linux/Mac terminal you can clean and reset the cursor with the following escape sequence:
  `std::cout << "\033[2J" << "\033[1;1H";`
- Provide a minimal **main** function that instantiate the two elements, link them and run the pipeline by starting the video source.

Part 3: Detection element

- Write an element that will perform a simple pattern detection to detect people inside video frames. You can search for the following pattern by iterating the whole picture:

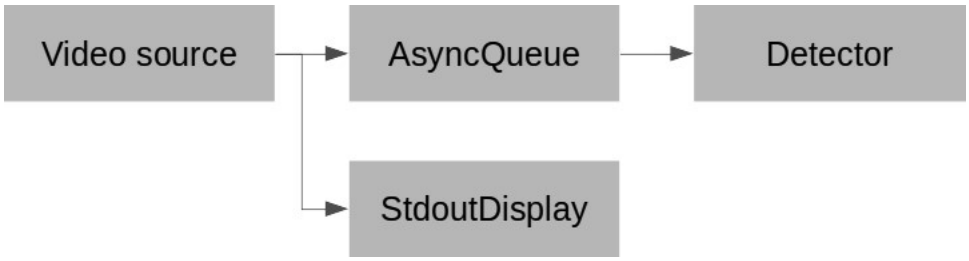
```
 x
xxx
 x
x x
```

- You can replace the pixel value with a specific one to highlight the detected pixels during the output.
- Add this element between the source and the display to run your pipeline with this detector.

Part 4: Asynchronous processing

In video pipeline implementations it is frequent to have element that will not be able to process a frame in real-time i.e. not fast enough before receiving a new frame. To circumvent this, it is often needed to introduce asynchronous handling of the video frame to not block the other elements of the pipeline and drop some of them to avoid filling the memory.

- Provide an “AsyncQueue” class that inherit your base element and can be linked in the pipeline before an element or a groupe of element to transform their processing to an asynchronous one:



  Use a container to hold received video frame references and process the downstream element in a dedicated thread to not block the upstream. A maximum number of held frame references should be configurable and the oldest frame should be dropped when the maximum is reached. Handle memory carefully to avoid leaking video frames.
- Add this AsyncQueue to your pipeline to demonstrate its utility.