Introduction to C++ and OpenCV

UNIK 4690

Welcome to the computer vision course! In this course we will learn how machines can perceive and understand their environments using images and video. We will do this through a combination of on-line prerecorded lectures and practical programming exercises where we each week solve a computer vision problem.

For the programming exercises we are going to use the programming language C++[1]. In particular we are going to use the software libraries OpenCV for image processing, Eigen[2] for linear algebra and Sophus[3] for Lie algebra.

It is assumed that the reader is familiar with programming in general, so the purpose of this document is to point out and explain peculiarities of C++ and the libraries we are utilizing.

1 C++

C++ is a popular high level programming language which is an extension of the language C. The popularity comes from the mentioned fact that it is built upon C and that it is implemented with compilers for most operating systems. Compared to C, C++ is to a greater extent a high level language. It has support for object oriented programming (OOP), generic programming (the use of so called *templates*) and offers lots of general containers, iterators, algorithms and I/O-functionality. Memory management is usually taken care of under the hood of the data types utilized, and one could say that it is easier to write both safe and fast code. If you are a complete beginner to C++, you are encouraged to read more at[1] and to browse around at the Internet for more thorough introductions.

Ragnar Smestad
Forsvarets forskningsinstitutt, e-mail: ragnar.smestad@ffi.no

1.1 Hello, world

We will now start by looking at a "hello, world"[4] example.

```
#include <iostream>

/* This program prints "Hello, world!"
 * to the command line
 */
int main(int argc, char* argv[])
{
    std::cout << "Hello, world!" << std::endl;
    return 0; // Everything is ok!
}
```

Output:

Hello, world!

include

In C++, the *standard library*[5] provides many, if not most, of the basic building blocks we need to create any kind of program. We get access to these building blocks by using `#include <...>` statements. You can think of the include statement as a copy/paste operation, where the compiler replaces the include-line with the actual contents of the included file. Two variants of include exist:

```
#include "abc.h"
#include <iostream>
```

iostream

The difference of the two versions is where the compiler starts to look for the file that is to be included. For filenames surrounded by <> it will search at "implementation-defined places". This typically means system directories, so <> is mostly used to include files from the standard library. When quotation marks `""` are used, the search will begin in the directory of the current file and then move on to the other directories in the so called "include path". (Details about "include path" will not be discussed here). What is somewhat special about the files from the standard library is that they have no file extension, but they are none the less just ordinary human readable files containing code. (The file `iostream` is included in the example). `iostream` (short for "input and output stream") contains functions for printing text to the command line, reading user input and so on.

> Streams are serial interfaces to storage, buffers, files, or any other storage medium. The difference between storage media is intentionally hidden by the interface; you may not even know what kind of storage you're working with but the interface is exactly the same.[6]

stream operators

cout

When we are working with "streams", we prefer to use the streaming operators. We have the insert operator << for putting objects into streams and the extract operator >> for loading objects out of streams. `std::cout` is the **standard output stream**, which is what we are usually using for printing text to the command line. In the above example we are first sending "Hello, world!" to `std::cout`, followed by std::endl. That last statement writes a newline character ('\n') before "flushing" the stream. To flush causes the text to be written immediately.

An important thing to notice is that the stream cout is within the *namespace* std. You get access to a function, class or variable from a given namespace by adding the name of the namespace followed by :: in front of the entity, just as we saw here. Namespaces are useful for avoiding naming conflicts when two libraries define anything with the same name. (An example could be the functions std::min from the standard library and cv::min from OpenCV. The compiler wouldn't know which of them to use if both of them were named only min without a namespace).

namespace

An executable program in C++ is required to have a function called main. The "main" function takes two arguments, **int** argc and **char*** argv[] which contains respectively the number of arguments passed to the program from the command line and a pointer to an array containing these arguments. If our program is called **hello_world** and we start it from a Linux terminal by typing

main

argc, argv

```
./hello_world arg1 arg2
```

then argc will contain the number **3** and argv will be a pointer to an array containing **"/path/to/hello_world"**, **"arg1"**, **"arg2"**. These values are not used in our example, and then it is in fact allowed to declare main with empty parentheses:

```
int main()
{/* ... */}
```

The return value of main should indicate how the program was terminated. If everything turned out fine the return value should be **0**, and if anything went wrong some number other than zero should be returned. There is no standard for how these non-zero values are interpreted, so whether you return 1 or 100 means "error" none the less. If you are a fan of readable code (you should be) and you want to express without a doubt what is happening, you can choose to return the values EXIT_SUCCESS and EXIT_FAILURE, defined in <cstdlib>. (In C++ it is actually allowed to omit the return statement from the main function. The default value **0** will then automatically be returned).

return

In our example, two types of comments are used . /* */ are comments that may span several lines, while // ends at line breaks.

comments

1.2 Classes

C++ has as mentioned support for object oriented programming, and we can easily create our own classes[7]. It is assumed that the concept of classes and object oriented programming is known.

It is common practice to split the declaration of a class (called its *interface*) and the definition (called implementation) into separate units. The interface is placed in a so called header file, which only contains the name of the class and a listing of its methods and variables. What happens within the methods is specified in the implementation, which lies in a separate source file or eventually in an already compiled object file. Let's look at a trivial example:

numberkeeper.h

```
#ifndef NUMBERKEEPER_H
#define NUMBERKEEPER_H

/// This useless class does nothing else
/// than to store a number.
class NumberKeeper
{
public:
  /// Constructor which sets the number to be stored
  NumberKeeper(int the_number);

  /// Returns the stored number
  int getTheNumber();

private:
  int the_number_;
};
#endif // NUMBERKEEPER_H
```

This header file tells us the following things:

- A class named `NumberKeeper` exists.
- The class expects an integer number in its constructor.
- The class has a *public* method `getTheNumber`. We may use it since it is public.
- The class has a *private* variable called `the_number_`. It is not available to us users as it is private.

This should be enough information for us to use the class. The triple slash comments are put there in order to facilitate auto generation of code documentation, but it is also relevant to point out that mostly the same information is provided through the use of meaningful and explanatory names of both the class and its methods and variable. This is called "self documenting code" and is something you should strive for. That names of private variables end with an underscore is a pretty common convention which we also use in this course.

header guard

The lines in the header file starting with `#ifndef` , `#define` and `#endif` creates what is called a *header guard*. The header guard ensures that this file can be included only once, and thus preventing the compiler of complaining about "multiple definitions of ...". You can't declare two things with the same name, and header guards prevent this from happening. (Know that you can replace the header guards with a simple command `#pragma once`[8] at the top of your header file, which is easier to use but is in fact not standard C++. We stick to the standard in this course and use header guards).

How the class resolves its mission is for us completely unknown, unless we have access to the implementation. In this case we have:

numberkeeper.cpp

```
#include "numberkepper.h"

NumberKeeper::NumberKeeper(int the_number)
  : the_number_{the_number}
{}

NumberKeeper::getTheNumber()
{
  return the_number_;
}
```

The implementation shows the following:

- The scope operator that we recognize from namespaces is also used when implementing a method from a class. We can see from the example that we must prepend `NumberKeeper::` to both the constructor and `getTheNumber`.
- Before the curly braces in the constructor, we have used a *member initializer list*[9]. The syntax for this is the colon character, followed by a comma-separated list of initializations of class members. (We have only one member and therefore no commas). The alternative is to write `the_number_ = the_number;` between the curly braces.
- The method `getTheNumber` returns only the value of the private variable `the_number_` (like we could probably have guessed).
- The cpp-file (the implementation) includes the header file.

2 OpenCV

OpenCV is a poular software library for computer vision and machine learning. In most of our labs throughout the course we are using different parts of this library in order to test the theory in practice. Some more general information from the OpenCV website follows:

> OpenCV (Open Source Computer Vision Library) is an open source computer vision and machine learning software library. OpenCV was built to provide a common infrastructure for computer vision applications and to accelerate the use of machine perception in the commercial products. Being a BSD-licensed product, OpenCV makes it easy for businesses to utilize and modify the code.
>
> The library has more than 2500 optimized algorithms, which includes a comprehensive set of both classic and state-of-the-art computer vision and machine learning algorithms. These algorithms can be used to detect and recognize faces, identify objects, classify human actions in videos, track camera movements, track moving objects, extract 3D models of objects, produce 3D point clouds from stereo cameras, stitch images together to produce a high resolution image of an entire scene, find similar images from an image database, remove red eyes from images taken using flash, follow eye movements, recognize scenery and establish markers to overlay it with augmented reality, etc. OpenCV has more than 47 thousand people of user community and estimated number of downloads exceeding 14 million. The library is used extensively in companies, research groups and by governmental bodies. (...)
>
> It has C++, C, Python, Java and MATLAB interfaces and supports Windows, Linux, Android and Mac OS. OpenCV leans mostly towards real-time vision applications and takes advantage of MMX and SSE instructions when available. A full-featured CUDA and OpenCL interfaces are being actively developed right now. There are over 500 algorithms and about 10 times as many functions that compose or support those algorithms. OpenCV is written natively in C++ and has a templated interface that works seamlessly with STL containers. [10]

As you see, with OpenCV we are well prepared to learn and have fun with computer vision! This chapter will go through some selected parts and components of the library that you should know about in order to get started with the labs. There will be many references to the online documentation, so please go there, take a look and explore: https://docs.opencv.org/. . In it's current state, this booklet refers to version 3.3.1. The documentation contains an introductory chapter which goes throug the most common API-concepts of OpenCV. In stead of us pasting it all in this booklet, you are encouraged to read that introduction[1]. .

documentations

introduction

2.1 cv::Mat

Maybe the most common datatype of OpenCV is the `cv::Mat`, a class that represents an n-dimensional one- or multichannel dense array. In other words, this is the obvious class for representing images. You'll find it in the documentation under

[1] https://docs.opencv.org/3.3.1/d1/dfb/intro.html

Main modules: > Core functionality > Basic structures > cv::Mat[2]. The documentation actually contains a dedicated introduction to `cv::Mat` under *OpenCV Tutorials > The Core Functionality (core module) > Mat - The Basic Image Container*[3]. Take a look.

`cv::Mat`

An important property of the `cv::Mat` is that it mostly acts as a "smart pointer" to some memory where the image data are located, and that the class is responsible for safe memory management of that memory. The rest of the class (referred to as the *matrix header*) contains information about the image size, type and other meta information. The reason why it is implemented this way is to avoid unnecessary copying of data when passing matrices between functions. If you call a function that takes a `cv::Mat` as an argument, only the header is copied into the function. The data pointed to by that header is the same as pointed to by the original `cv::Mat`.

```
// Create a new matrix; memory is allocated under the hood.
cv::Mat a(/* constructor arguments */);

cv::Mat b(a);            // use "copy constructor"
cv::Mat c = a;           // use "assignment operator"
```

All the matrices, both `a`, `b` and `c` does now point to the same single data chunk in memory. The headers are different, but if you alter the pixel values in one of the matrices, the other two would be equally affected.

If you explicitly want to make a copy, you have two options:

```
cv::Mat a(/* ... */); // create new. Copy it by doing
// either
cv::Mat b = a.clone();
// or
cv::Mat b;
a.copyTo(b);
```

So far we haven't mentioned the constructor parameters. To create a new `cv::Mat`, we see from the documentation that we have several options:

```
Mat (int rows, int cols, int type)
Mat (Size size, int type)
Mat (int rows, int cols, int type, const Scalar &s)
Mat (Size size, int type, const Scalar &s)
```

Both `Size` and `Scalar` are datatypes from OpenCV that you may easily find in the docs. while the value **`int`** `type` is a little more peculiar and deserves some words of it own here. It is a macro composed of the format

type

```
CV_<bit-depth>{U|S|F}C(<number_of_channels>)
```

, so for example `CV_8UC1` would be a one channel image containing **`unsigned char`**s, `CV_32FC3` would be a tree channel image with 32-bit **`float`**s and so on. Some

[2] https://docs.opencv.org/3.3.1/d3/d63/classcv_1_1Mat.html

[3] https://docs.opencv.org/3.3.1/d6/d6d/tutorial_mat_the_basic_image_container.html

image processing functions in OpenCV requires the input image to be on a certain format, or returns a result that is of a different type than what you put in.

2.2 *Read images from file or from camera*

imread

We use the function `cv::imread`[4]to read images from disk. It takes two arguments: filename and "flags". By specifying different flags you can choose how the image should be loaded. The standard flag is `cv::IMREAD_COLOR`, which means that OpenCV will convert a one-channel gray level image to a tree-channel color image when you load it! To avoid surprises, it is therefore more convenient to call the function like this:

```
cv::Mat image = cv::imread("filnavn", cv::IMREAD_UNCHANGED);
```

VideoCapture

Fetching images from a camera is done through the use of the class `VideoCapture`[5]. As well as reading from cameras, this class can read video files, urls or even sequences of images from disk. (If you specify the string `img_\%02d.jpg` as a filename, it is interpreted as the sequence img_00.jpg, img_01.jpg, img_02.jpg, . . .). In order to read from a camera, you only need to to specify the "camera index". This is typically 0 for the first camera connected to the computer (the integrated webcam), and then 1, 2, ... for every new camera connected. Connecting a camera could go like this:

```
cv::VideoCapture cap(0);

if (!cap.isOpened())
{
  std::cerr << "ERROR! Unable to open camera\n";
  return EXIT_FAILURE;
}
```

2.3 *Display an image*

imshow

It is very common that we want to display an image on the screen, and for that we have the function `cv::imshow`[6]. The documentation has an important comment regarding to the use of `cv::imshow`:

waitKey

> This function should be followed by `cv::waitKey` function which displays the image for specified milliseconds. Otherwise, it won't display the image. For example, `waitKey(0)`

[4] https://docs.opencv.org/3.3.1/d4/da8/group__imgcodecs.html#ga288b8b3da0892bd651fce07b3bbd3a56

[5] https://docs.opencv.org/3.3.1/d8/dfe/classcv_1_1VideoCapture.html

[6] https://docs.opencv.org/3.3.1/d7/dfc/group__highgui.html#ga453d42fe4cb60e5723281a89973ee563

will display the window infinitely until any keypress (it is suitable for image display). `waitKey(25)` will display a frame for 25 ms, after which display will be automatically closed. (If you put it in a loop to read videos, it will display the video frame-by-frame)

If you press any key while `cv::waitKey` is waiting, it will return immediately with a number corresponding to the key that was pressed. If no key is pressed before the time expires, the function will return -1.

In addition to the `cv::Mat` that we want to display, the function `imshow` takes a window name as an argument. This makes it possible to display images in the same window over and over, or more importantly, it makes it possible to display separate images in separate windows. To create a new named window, we call the function `cv::namedWindow`[7] . Let's sum up with a complete example of how to fetch images from a camera and display it on screen.

namedWindow

```
#include "opencv2/highgui.hpp"
#include <iostream>

int main()
{
  cv::VideoCapture input_stream(0);

  if (!input_stream.isOpened())
  {
    std::cout << "could not open camera" << std::endl;
    return EXIT_FAILURE;
  }

  const std::string window_title = "window 0";
  cv::namedWindow(window_title);

  cv::Mat frame;
  while (true)
  {
    input_stream >> frame;
    if (frame.empty())
    { break; }

    cv::imshow(window_title, frame);
    if (cv::waitKey(15) >= 0)
    { break; }
  }

  return EXIT_SUCCESS;
}
```

[7] https://docs.opencv.org/3.3.1/d7/dfc/group__highgui.html#ga5afdf8410934fd099df85c75b2e0888b

2.4 Processing images

This section gives an introduction to image processing. We'll start by iterating over an image and at the same time modifying the value of singular pixels. We start by creating a matrix with zeros, without reading from any file or camera. For this we are using the method `cv::Mat::zeros` which gives us an image with all pixels set to 0. After that, we run through the image and update the pixel values according to a formula that depends on each pixel's coordinates.

zeros

```
#include "opencv2/highgui.hpp"
#include "opencv2/imgproc.hpp"
#include <iostream>

int main()
{
  cv::Mat frame = cv::Mat::zeros(16, 16, CV_8UC1);

  for (int row = 0; row < frame.rows; ++row)
  {
    for (int col = 0; col < frame.cols; ++col)
    {
      const uchar pixel_value = (row+1) * (col+1) - 1;
      frame.at<uchar>(row,col) = pixel_value;
    }
  }

  cv::resize(frame, frame, cv::Size(256, 256), 0, 0,
             cv::INTER_LINEAR);

  cv::imshow("gradient", frame);
  cv::waitKey(0);

  return EXIT_SUCCESS;
}
```

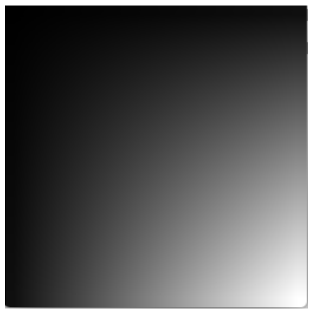

Fig. 1: Gradient image

The result is displayed in fig. 1. Notice that we did also use the function `cv::resize`[8] in order to change the resolution of our image (which was only 16 x 16 pixels) to a size that was more suitable for display. To get a nice and smooth gradient, the interpolation method `cv::INTER_LINEAR` was selected, that is of course linear interpolation.

resize

interpolasjon

In the code we see examples of meta data that the `cv::Mat` keeps for its matrix. In the test expressions of the for loops the image height and width are obtained by calls to `frame.rows` and `frame.cols`. It would have been possible also to get a `cv::Size` by calling `frame.size()`, and then get the height and width via `frame.size().height()` and `frame.size().width()`.

Size

A thing that is somewhat odd, is that we have to tell the function `cv::Mat::at` what kind of data type the image contains. In the example, `frame.at<uchar>` indicates that the image contains **`unsigned char`** (shortened to uchar), just as we specified with `CV_8UC1` when the matrix was created. Read more about accessing pixels of other types and channels in the tutorial *Operations with images*[9].

Other useful operations on images are those of converting between data types or changing of color space. To change the data type of pixels, we use the method `cv::Mat::convertTo`[10]. In addition to changing type, it can also scale the pixel values. In the following example we are going from integer values $\{0, 1, \ldots, 255\}$ stored as **`unsigned char`** to **`float`**s in the interval $[0, 1]$.

convertTo

```
cv::Mat uchar_image(16, 16, CV_8UC1);
cv::Mat float_image;
uchar_image.convertTo(float_image, CV_32FC1, 1/255.0f);
```

In order to convert a color image to a grayscale image, we use `cv::cvtColor`[11] with the parameter `cv::COLOR_BGR2GRAY`. The same function can convert images to and from colorspaces like HSV, L*a*b* and many more by just changing the parameter value.

cvtColor

```
cv::Mat color_image = imread("color_image.jpg");
cv::Mat gray_image;
cv::cvtColor(color_image, gray_image, cv::COLOR_BGR2GRAY);
cv::Mat hsv_image;
cv::cvtColor(color_image, hsv_image, cv::COLOR_BGR2HSV);
```

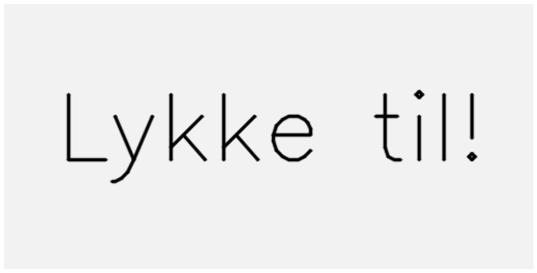

Fig. 2: Lykke til!

2.5 *Good luck!*

```cpp
#include "opencv2/imgcodecs.hpp"
#include "opencv2/imgproc.hpp"

int main()
{
  cv::Mat frame = cv::Mat::zeros(256, 512, CV_8UC1);
  frame += 242;

  const auto font = cv::FONT_HERSHEY_SIMPLEX;
  cv::putText(frame, "Lykke til!", {50,150}, font, 3,
              {0,0,0}, 2, cv::LINE_AA);
  cv::imwrite("tnx.png", frame);

  return EXIT_SUCCESS;
}
```

[8] https://docs.opencv.org/3.3.1/da/d54/group__imgproc__transform.html#ga47a974309e9102f5f08231edc7e7529d
[9] https://docs.opencv.org/3.3.1/d5/d98/tutorial_mat_operations.html
[10] https://docs.opencv.org/3.3.1/d3/d63/classcv_1_1Mat.html#adf88c60c5b4980e05bb556080916978b
[11] https://docs.opencv.org/3.3.1/d7/d1b/group__imgproc__misc.html#ga397ae87e1288a81d2363b61574eb8cab

References

[1] Wikipedia. *C++*. URL: https://no.wikipedia.org/w/index.php?title=C%2B%2B&oldid=17629715 (visited on 01/18/2018).

[2] Gaël Guennebaud, Benoît Jacob, et al. *Eigen v3*. http://eigen.tuxfamily.org. 2010.

[3] Daniel Stonier. *Sophus*. URL: https://github.com/stonier/sophus (visited on 01/18/2018).

[4] Wikipedia. *Hello, world*. URL: https://no.wikipedia.org/w/index.php?title=Hello,_world&oldid=18113685 (visited on 01/18/2018).

[5] Wikipedia. *C++ Standard Library*. URL: https://en.wikipedia.org/w/index.php?title=C%2B%2B_Standard_Library&oldid=817321592 (visited on 01/18/2018).

[6] Manasij Mukherjee — cprogramming.com. *A Gentle Introduction to C++ IO Streams*. URL: https://www.cprogramming.com/tutorial/c++-iostreams.html (visited on 01/18/2018).

[7] Wikipedia. *C++ classes*. URL: https://en.wikipedia.org/w/index.php?title=C%2B%2B_classes&oldid=816783232 (visited on 01/18/2018).

[8] Wikipedia. *Pragma once*. URL: https://en.wikipedia.org/w/index.php?title=Pragma_once&oldid=818631485 (visited on 01/18/2018).

[9] cppreference.com. *Constructors and member initializer lists*. URL: http://en.cppreference.com/w/cpp/language/initializer_list (visited on 01/18/2018).

[10] OpenCV team. *Open Source Computer Vision Library*. URL: https://opencv.org/ (visited on 01/18/2018).