C/C++ Programming Basics

OBJECTIVES

- Compile and execute C/C++ code in Ubuntu 12.04.4 using the Terasic DE2i-150 Development Kit.
- Learn about image convolution.
- Execute C applications that use loops, arrays, functions, structures, pointers, dynamic memory allocation.
- Learn how to read/write binary and text files in C.
- Execute C++ applications that use objects and functors.

FUNDAMENTALS

REPOSITORY EXAMPLES

- Refer to the Tutorial: Embedded Intel for the source files used in this Tutorial.

TERASIC DE2I-150 BOARD

- Refer to the board website or the Tutorial: Embedded Intel for User Manuals and Guides. We mention the information that is relevant for the Microprocessor system (these documents heavily focus on the FPGA system):
  - ✓ DE2i-150 Quick Start Guide: To quickly connect the power, mouse, keyboard, display for the Intel® Atom^{TM}.
  - ✓ DE2i-150 Getting Started Guide: Details on powering the Board.
  - ✓ DE2i-150 FPGA System User Manual: Installation of WiFi Module and Antenna on DE2i-150.
  - ✓ DE2i-150 Windows 7 User Manual: Boot DE2i-150 with a Bootable USB Flash Drive
  - ✓ Installing Ubuntu OS on the DE2i-150: Some tips for Ubuntu OS installation. * We use a USB flash drive with an image.

BOARD SETUP

- Connect the monitor (VGA or HDMI) as well as the keyboard and mouse.
  - ✓ Refer to the *DE2i-150 Quick Start Guide* (page 2) for a useful illustration.
- Install the Wi-Fi module and Antenna (see *DE2i-150 FPGA System User Manual*, page 108). * In the board you receive, this step will be already completed for you.

Powering up the DE2i-150 Board

- Connect the provided power cord to the power supply and plug the cord into a power outlet.
- Connect the supplied 12V DE2i-150 power adapter to the power connect (J1) on the DE2i-150 board. At this point, you should see the 12 V LED (D33) turn on.
  - ✓ Be careful not to plug the power adapter into the SATA power connector (see *DE2i-150 Getting Started Guide*, page 7).
- Click the **Power ON/OFF Button** (lower right corner) to boot the OS.
- The board should power on, emitting some beeps to indicate a successful load of the BIOS.
- Once Ubuntu screen loads, enter the login information: User: student, password: (sent to you via email, you can change it).

ACTIVITIES

FIRST ACTIVITY: 2D CONVOLUTION IN C

- This is a computation-intensive and popular application. The input image (I) of size SX×SY (SX columns, SY rows) is convolved with a kernel (K) of size KX×KY to generate and output image (O) of size (SX+KX-1)×(SY+KY-1).

$$O(m,n) = I(m,n)*K(m,n) \sum_{j=0}^{SX-1}\left(\sum_{i=0}^{SY-1} I(i,j)\times K(m-i,n-j)\right)$$

- $m = 0, \ldots, SY+KY-1$, $n = 0, \ldots, SX+KX-1$. When the indices of K are outside the bounds ($0, \ldots, KY-1, 0, \ldots, KX-1$), the product is ignored. Also, the restriction $i = 0, \ldots, SY-1$, $j = 0, \ldots, SX-1$, effectively zero-pads I.

- Fig. 1 illustrates these concepts. The convolution operation is like a sliding window: for every *O(m,n)*, the flipped kernel overlaps with I, where we only multiply-and-add the overlapping elements. In some cases, there are elements of the flipped kernel that do not overlap with the elements of I. Here, these operations do not occur (this is like I was zero-padded).
  - ✓ For the sake of simpler explanation, we use a matrix with integer elements as the "input image" I. In the Second Activity, we will use an actual image.

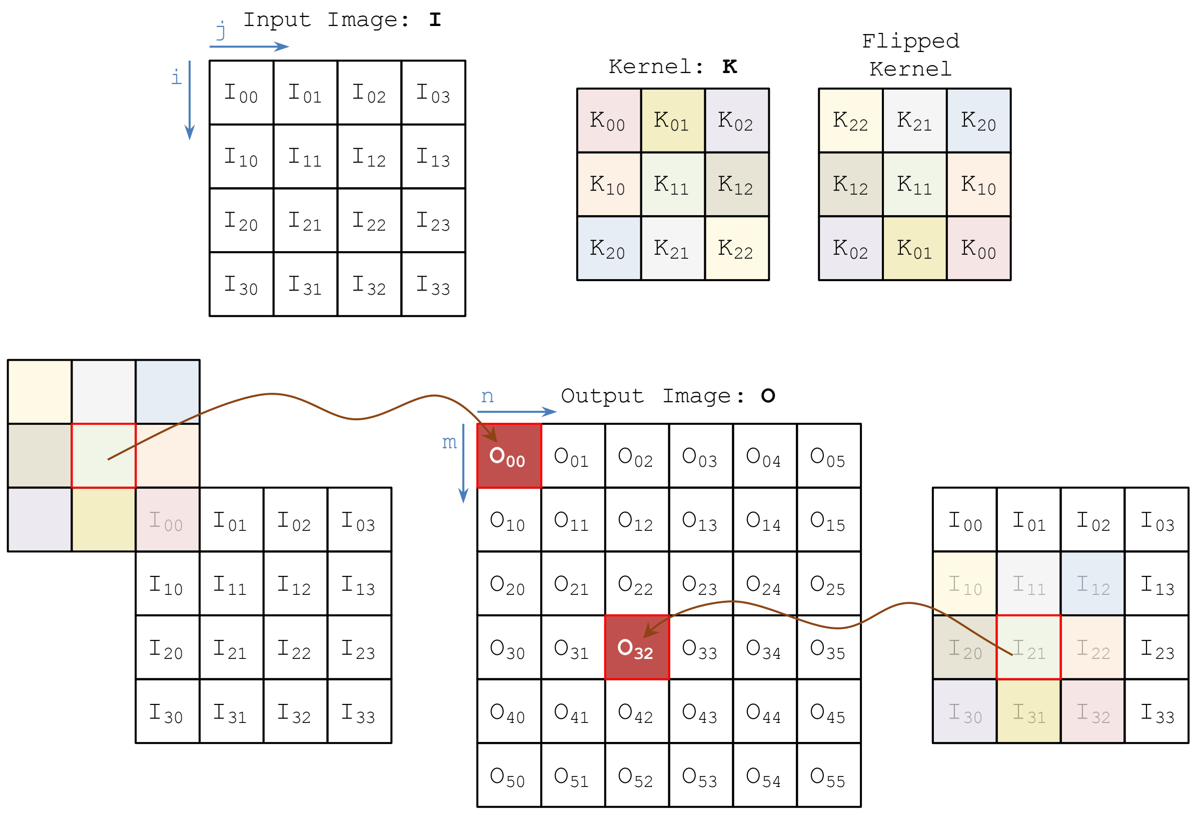


Figure 1. 2D convolution operation. Input Image: I. Kernel: K. Output Image: O

- Element Computation examples:
  - ✓ $O_{00} = I_{00}\times K_{00}$
  - ✓ $O_{32} = I_{10}\times K_{22} + I_{11}\times K_{21} + I_{12}\times K_{20} + I_{20}\times K_{12} + I_{21}\times K_{11} + I_{22}\times K_{10} + I_{30}\times K_{02} + I_{31}\times K_{01} + I_{32}\times K_{00}$

- A straightforward implementation involves nested loops where a dot product is computed for each element of the output matrix. There are (SX+KX-1)×(SY+KY-1) dot products in a 2D convolution, each dot product involving two KX×KY matrices.

- It is customary to consider only the central part of the convolution. If there is an odd number of rows or columns in the output matrix O, the "center" leaves one more at the beginning than the end. The software application tested here only considers the central part of the convolution. There are SX×SY dot products in a 2D convolution, where each dot product involves two KX×KY matrices. Each dot product adds up to KX×KY products.
  - ✓ Fig. 2 shows the example used in this activity. The output convolution size is SX×SY (central part of the convolution).

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix} * \begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix} = \begin{bmatrix} -2 & 0 & 2 & 9 \\ 9 & 6 & 7 & 17 \\ 17 & 10 & 11 & 25 \\ 42 & 32 & 34 & 53 \end{bmatrix}$$

Figure 2. 2D convolution example. SX=SY=4, KX=KY=3. Output size is the same as input size.

- Application files: `conv2.c, conv2_fun.h, conv2_fun.c, Makefile`
  - ✓ The application includes a main `.c` file, a header (`.h`) file for function declarations, and a `.c` file for function definitions.
  - ✓ This example uses loops, arrays, functions, structures, pointers, dynamic memory allocation, and read/write text files.
  - ✓ General Procedure:
    - ▫ The input matrix I and the kernel K are read from text files. We use SX=SY=4, KX=KY=3.
    - ▫ Convolution Computation
    - ▫ The resulting output matrix O is stored in a text file.
      - · Text files: Each data element (one line) is a 32-bit signed number. A matrix is represented in a raster-scan fashion.

- Compile this code: `make all` ↵
- Execute this application: `./conv2` ↵
  - ✓ Use SX=4, SY=4, KX=3, KY=3 when prompted.
  - ✓ You should get the same result as in Fig. 2.

SECOND ACTIVITY: IMAGE CONVOLUTION IN C

- The convolution procedure of the previous activity is now applied to a grayscale image. Fig. 3 depicts an input image that is converted to grayscale, so we can apply image processing algorithms.

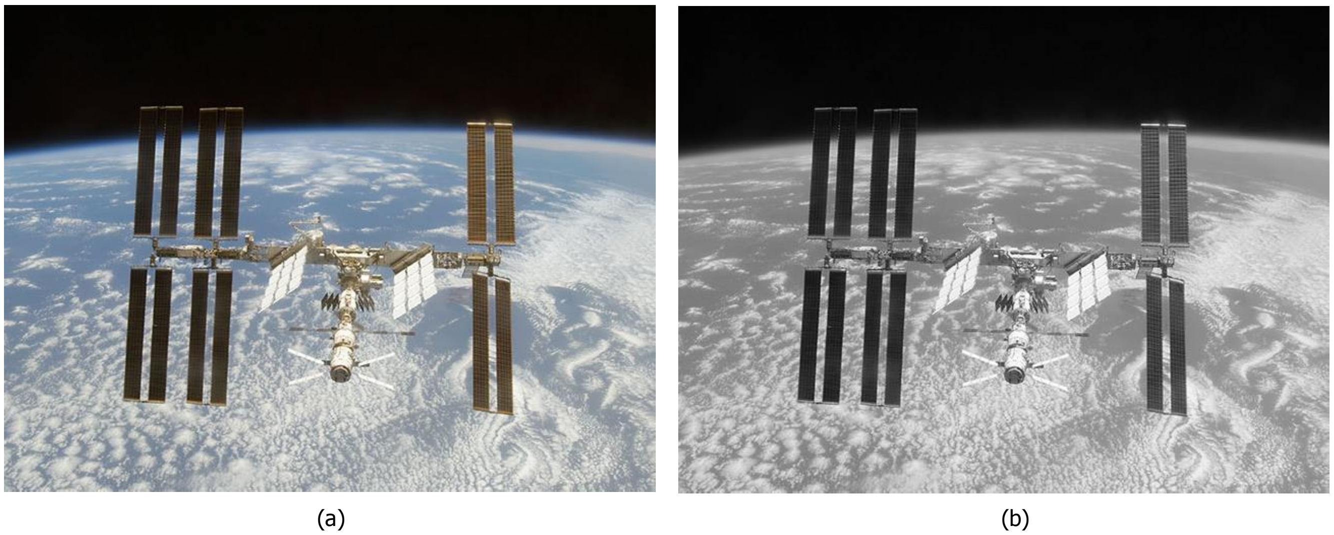
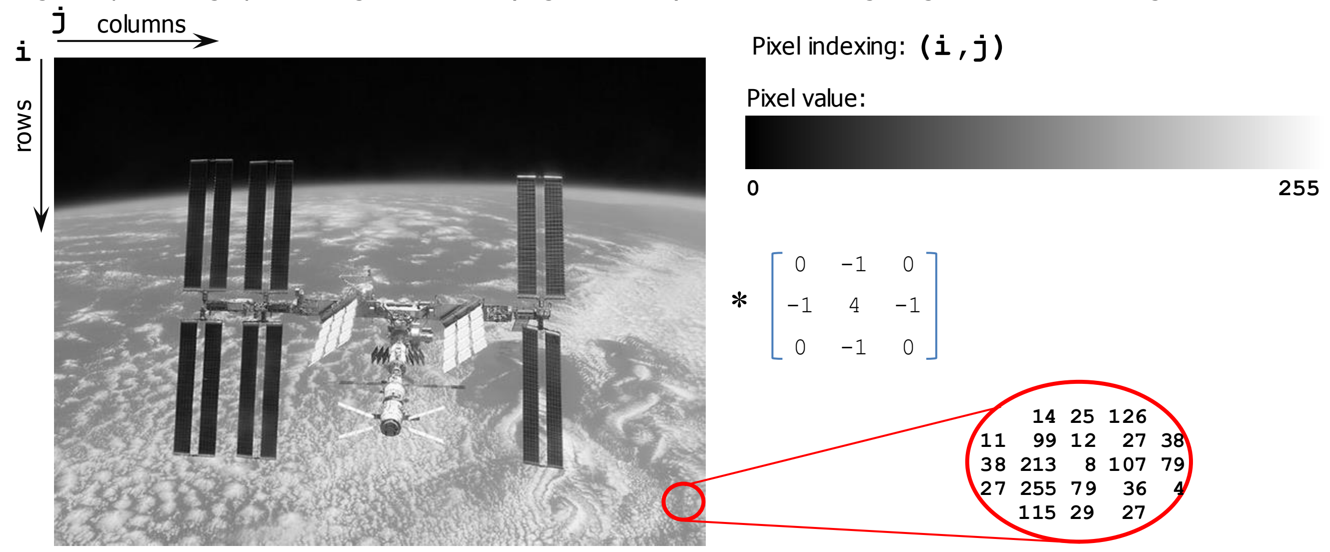
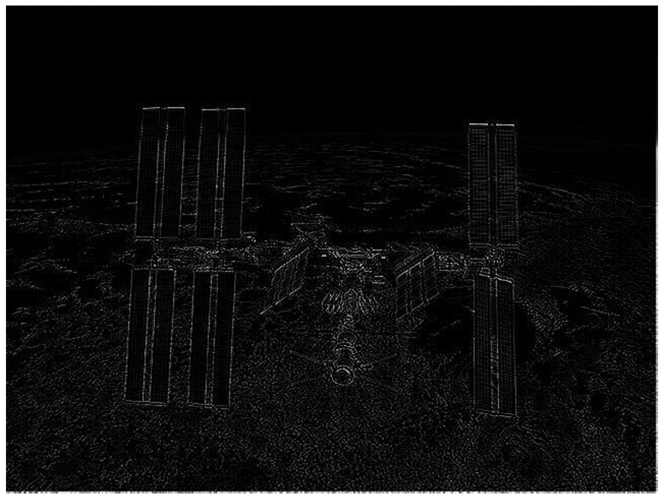


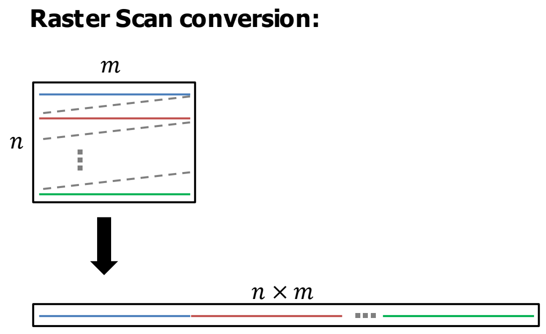
Figure 3. Image used in this example. (a) Color (RGB) image. (b) Grayscale version.

- Fig. 4 depicts the grayscale image, the kernel (edge detection), and the resulting image. It also shows image coordinates.



(a)





(b) (c)

Figure 4. (a) Grayscale image and the edge detection kernel. (b) Filtered image. (c) Raster scan procedure to turn a matrix into a linear array.

- From Fig. 4, note that:
  - ✓ Grayscale image: Represented as matrix of integers. The image coordinates are different than the cartesian coordinates.
  - ✓ Input grayscale image: each pixel value is represented by an unsigned 8-bit integer. The values are then bounded to [0,255], where 0 represents the darkest, and 255 the brightest.
  - ✓ A pixel is indexed as (i, j), where i is the row number, and j the column number.
  - ✓ Output image: the pixel values might fall outside the [0,255] bounds. When displaying, it is customary to restrict the pixel values to [0, 255].

PROCEDURE

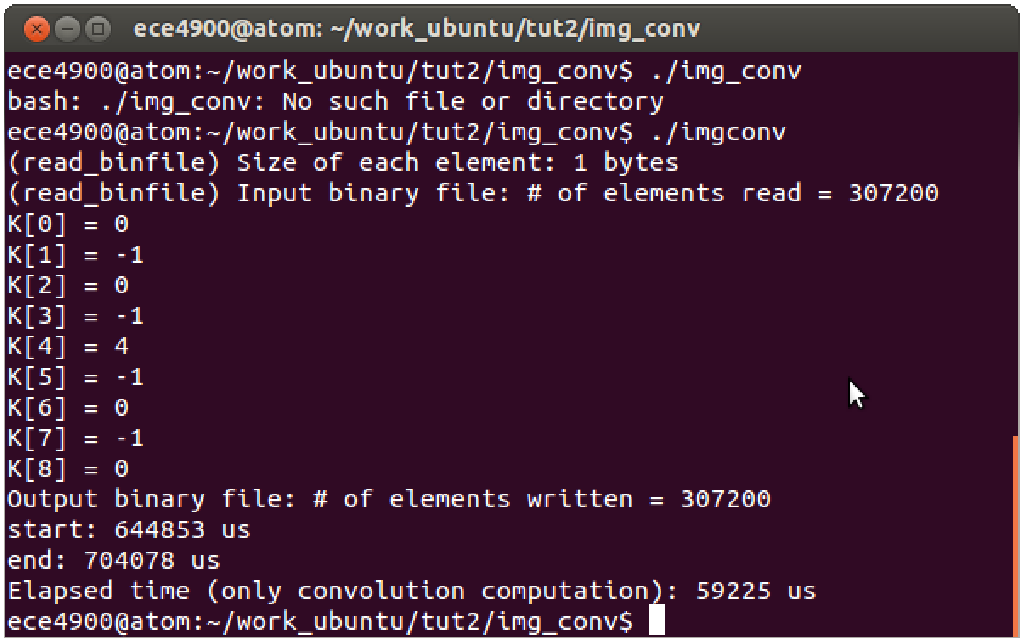
- The size of the input image is $m \times n$, with $m = 640$, $n = 480$. We prefer to store images as binary files, that will be read by the C application. The kernel is read from a text file (as in the 1st Activity). The resulting image is also stored as a binary file.
- For rapid generation of input binary files, for output data verification, and for displaying, we can use a MATLAB® script.
- The procedure is summarized in Table I.
  - ✓ To turn a matrix into a linear array, we use the common raster-scan approach. This is shown in Fig. 4(c).
  - ✓ Input binary file (grayscale image): In the C application, the input pixels are represented as unsigned 8-bit integers (`unsigned char`). The data is then interpreted as a linear array where each byte represents a pixel.
  - ✓ Output binary file (grayscale image): In the C application, the output pixels are stored as integers (`int`). The data is then interpreted as a linear array where each 4-byte group represents a pixel.
  - ✓ For the tasks executed in MATLAB®, we use a `.m` script. Select operator '1' for task 1-3, and operator '2' for task 8.
    - Files: `img_op.m, iss.jpg`

TABLE I. SUMMARY OF THE IMAGE CONVOLUTION PROCEDURE

Task	Platform
1. Read the color input image (.jpeg, .bmp, etc.)	MATLAB
2. Convert the input color image into a grayscale image.	
3. Store the grayscale image as a binary file (using a linear array).	
4. Read the binary image into a linear array.	Terasic Board (using C)
5. Read the kernel (from an input text file).	
6. Perform 2D convolution.	
7. Write resulting binary image on an output binary file (as a linear array).	
8. Read binary image and display it as a 2D grayscale image	MATLAB

- Application files: `img_conv.c, imgconv_fun.h, imgconv_fun.c, Makefile`
  - ✓ This example includes a main `.c` file, a header (`.h`) file for function declarations, and a `.c` file for function definitions.
  - ✓ Note that we measure the processing time (us) using `gettimeofday()`.

- Compile this code: `make all ↵`
- Execute this application: `./imgconv ↵`
  - ✓ Fig. 5 depicts execution on the Terasic DE2i-150 board.
  - ✓ You can use MATLAB® to verify that your result is correct.



```
ece4900@atom:~/work_ubuntu/tut2/img_conv$ ./img_conv
bash: ./img_conv: No such file or directory
ece4900@atom:~/work_ubuntu/tut2/img_conv$ ./imgconv
(read_binfile) Size of each element: 1 bytes
(read_binfile) Input binary file: # of elements read = 307200
K[0] = 0
K[1] = -1
K[2] = 0
K[3] = -1
K[4] = 4
K[5] = -1
K[6] = 0
K[7] = -1
K[8] = 0
Output binary file: # of elements written = 307200
start: 644853 us
end: 704078 us
Elapsed time (only convolution computation): 59225 us
ece4900@atom:~/work_ubuntu/tut2/img_conv$
```

Figure 5. Image convolution (640x480) execution on Terasic DE2i-150 FPGA Development Kit.

THIRD ACTIVITY: SIMPLE EXAMPLES IN C++

- The following are simple examples that illustrate the use of objects and functors.

FIRST EXAMPLE

- Basic declaration and usage of classes.

```
#include <iostream>
using namespace std;

class person {
public: // Access specifier
  string name;
  int id;

  void get_details () { // Member function
    cout << "Name: " << name << "\t" << "id: " << id << "\n"; }
};

class Circle {
private: // members only accessible by other functions in the class
  float radius;

public:
  void compute_area (float r) {   // Member function
     radius = r; // radius is modified by the function
     float area = 3.14*radius*radius;
     cout << "Area is: " << area << endl; }
};

int main() {
   person p1;     // Declaring an object of class 'person'
   Circle myobj;  // Declaring an object of class 'Circle'

   // Accessing public data members:
   p1.name = "Daniel"; p1.id = 35032;
   p1.get_details(); // Accessing member functions in 'p1'

   // Accessing private data member outside the class (indirectly) 'myobj'
   myobj.compute_area(1.5);
   return 0;
}
```

- Application file: `class_samples.cpp`
- Compile this code: `g++ class_samples.cpp -o class_sample ↵`
- Execute this application: `./class_sample ↵`
  - ✓ Program Output:

```
Name: Daniel     id: 35032
Area is: 7.065
```

SECOND EXAMPLE

- Declaration (and usage) of default and parameterized constructors.

```
#include <iostream>
using namespace std;

class Example {
public:
    int x, y, s;

    // Default constructor (no parameters). Called automatically when object is created
    Example(): x(0), y(0) {} // this means x = 0, y = 0.

    // Parameterized constructor
    Example (int xa, int ya): x(xa), y(ya) {} // this means x = xa, y = ya.

    int myoperate()  {
      s = x*x + y*y;
      return s; }
};

int main() {
  int result_1, result_2;
  // Declare an object of class 'Example'
  Example obj1; // default constructor called (x=y=0)
  result_1 = obj1.myoperate(); // operation performed.
  cout << "Result (obj1): " << result_1 << endl;

  Example obj2(10,5); // parameterized constructor called (x=10, y=5)
  result_2 = obj2.myoperate(); // operation performed.
  cout << "Result (obj2): " << result_2 << endl;

  return 0;
}
```

✓ Two constructors. When no parameters indicated, the default constructor is called. When parameters are indicated, the parameterized constructor is called.

- Application file: `basic_constrs.cpp`
- Compile this code: `g++ basic_constrs.cpp -o basic_constrs` ↵
- Execute this application: `./basic_constrs` ↵
  - ✓ Program Output:

```
Result (obj1): 0
Result (obj2): 125
```

THIRD EXAMPLE

- Declaration and use of functors.

```
#include <iostream>
using namespace std;

class MyFunctorClass {
public:
   // Parameterized constructor
   MyFunctorClass (int x): xt(x) {} // xt = x

   int operator() (int y) { // Member function
       return xt+ y; }

   private: // can only be accessed by functions inside the class
      int xt; // Data member (private)
};

class Distance {
private:
   int feet;
   int inches; // if a variable (e.g.: DI) is defined as 'Distance', it will have DI.feet, DI.inches

public:
   Distance () { feet = 0; inches = 0; } // default constructor (no arguments, no parameters)
   Distance (int f, int i) { feet = f; inches = i; } // parameterized constructor

   // overload function call: result is of type Distance: it has Distance.feet and Distance.inches
   Distance operator () (int a, int b, int c) {
      Distance D;
      D.feet = a + c + 10;
      D.inches = b + c + 100;
      return D; }

    void displayDistance () {  // method to display distance
      cout << "F: " << feet << " I: " << inches << endl; }
};

int main() {
   int b;
   MyFunctorClass example(5); // 'example' is an object where x=5 (using the constructor)
             // when objects of the class are called, it will return the result of adding x and y.
   b = example(6); // it will call operator() and return b = 6 + 5
   cout << "Result: " << b << "\n";

   Distance D1(11,10), D2; // initializing: D1.feet = 11, D1.inches = 10, D2.feet = 0, D2.inches = 0
   cout << "First Distance:";
   D1.displayDistance(); // for object D1, it will display D1.feet, D1.inches

   D2 = D1(10,10,10); // here, the operation is computed as specified in 'operator()'
   cout << "Second distance:";
   D2.displayDistance(); // for the class D2, it will display D2.feet, D2.inches
   return 0;
}
```

✓ `MyFunctorClass`: the parameterized constructor is called when declaring the object. After that, if we specify a parameter and return value of type `int`, an object call will be interpreted as a function call (the one defined in `operator()`).

- ✓ `Distance`: the parameterized or default constructor is called when declaring the object. After that, if we specify 3 parameters and a return value, an object call will be interpreted as a function call (the one defined in `operator()`).

- Application file: `basic_functors.cpp`
- Compile this code: `g++ basic_functors.cpp -o basic_functors ↵`
- Execute this application: `./basic_functors ↵`
  - ✓ Program Output:

```
Result: 11
First Distance: F: 11 I: 10
Second distance: F: 30 I: 120
```

FOURTH EXAMPLE

- Separate Header and Implementation Files. We use three different files and a Makefile:
  - ✓ simple.cpp

```
#include <iostream>
#include "simple_fun.h"
using namespace std;

int main () {
  MyClass Y; // default constructor called
  cout << Y.getnum() << endl;

  MyClass X(35); // parameterized constructor called
  cout << X.getnum() << endl;
  return 0;
}
```

  - ✓ simple_fun.cpp

```
#include "simple_fun.h"
// include: class implementation (constructors, functions), other functions implementation
MyClass:: MyClass(): num(0) {} // default constructor
MyClass:: MyClass(int n): num(n) {} //parameterized constructor

int MyClass::getnum() { // member function definition
  return num;
}
```

  - ✓ simple_fun.h

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

// include: class definitions (including functions inside class), other function prototypes
class MyClass {
  private:
    int num;
  public:
    MyClass(); // default constructor declaration
    MyClass(int n); // paramterized constructor declaration
    int getnum(); // member function declaration
};
```

  - ✓ Makefile

```
# Compiler/linker setup -----------------------------------------------------
# Linux-specific flags.  Comment these out if using Mac OS X.
PLATFORM = linux
CC       = g++
CFLAGS   = -O3 -Wall
OSLIBS   =
LDFLAGS  =

OBJS = simple
all: $(OBJS)

simple: simple.cpp simple_fun.o
  $(CC) $(CFLAGS) -o simple simple.cpp simple_fun.o

# library
simple_fun.o: simple_fun.cpp simple_fun.h
  $(CC) $(CFLAGS) -c simple_fun.cpp

# Maintenance and stuff -----------------------------------------------------
clean:
  rm -f $(OBJS) *.o core
```

- C++ classes (and often function prototypes) are normally split into two files. The header `.h` file contains class definitions and functions. The implementation of the class goes into the other `.cpp` file (`simple_fun.cpp`)

- Application file: `simple.cpp, simple_fun.cpp, simple_fun.h`
- Compile this code:
  - ✓ First method `g++ simple.cpp simple_fun.cpp -o simple ↵`
  - ✓ Second method: `make all ↵`

- Execute this application: `./simple ↵`
  - ✓ Program Output:

```
0
35
```

FOURTH ACTIVITY (NEURON IMPLEMENTATION)

- This example implements the computation of an artificial neuron via a class that includes constructors and functors.

NEURAL NETWORK

- A 3-layer neural network (also called a Fully Connected Layer) is depicted in Fig. 6(a). The input layer represents the input values to the network. Fig. 6(b) depicts the inputs and output of the first neuron (index '1') in layer 3.
- Fig. 6(c) depicts an artificial neuron model. The neuron output (action potential a_j^l) results from applying an activation function to the membrane potential (z_j^l). The indices correspond to the first neuron (index '1') in layer l.
- The membrane potential z_j^l is a dot product between the inputs and the associated weights, to which a bias is then added.

$$z_j^l = \sum_k w_{jk}^l a_k^{l-1} + b_j^l, l > 1$$

- The action potential of a neuron is denoted by a_j^l, is modeled as a scalar function of z_j^l:

$$a_j^l = \sigma(z_j^l) = \sigma\left(\sum_k w_{jk}^l a_k^{l-1} + b_j^l\right), l > 1$$

  - ✓ Common activation functions include:
    - ▫ Rectified Linear Unit (ReLU): $\sigma(z_j^l) = \max(0, z_j^l)$
    - ▫ Hyperbolic Tangent: $\sigma(z_j^l) = \tanh(z_j^l)$

- Vectorized notation for a layer output: (l is the layer the neuron is in, w^l is the weight matrix of the layer l, and $\vec{b}^l$ the bias vector of the layer l)

$$\vec{a}^l = \sigma(\vec{z}^l), \qquad \vec{z}^l = w^l \vec{a}^{l-1} + \vec{b}^l, l > 1$$

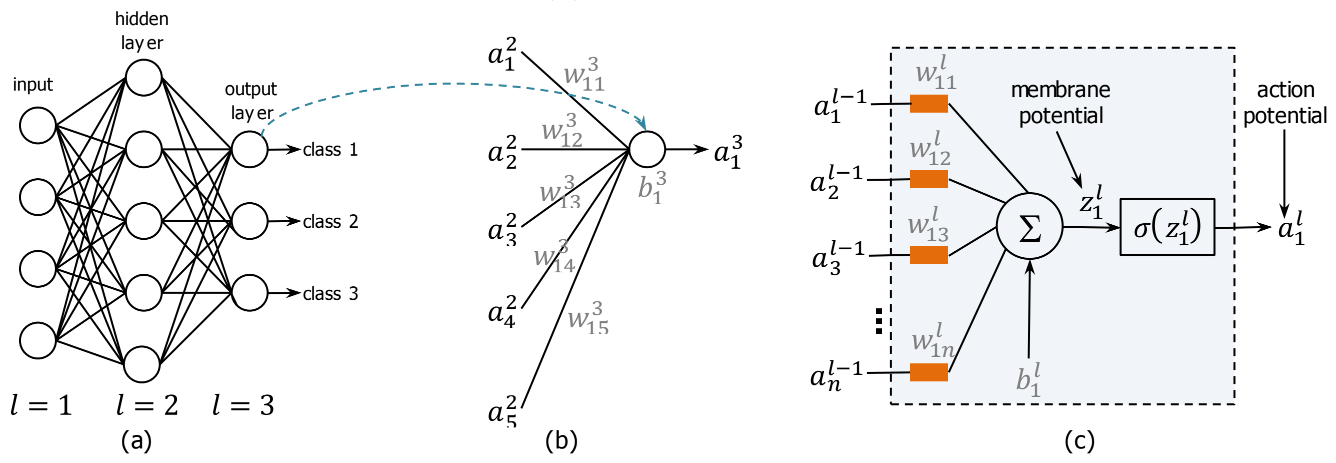


Figure 6. (a) 3-layer neural network. The input layer represents the input values to the network. (b) The first neuron (index '1') in the third layer. (c) Artificial neuron model. The membrane potential is a sum of products (input activations by weights) to which a bias term is added. The neuron shown belongs to a layer *l*. The input activations come from a previous layer (*l-1*). The neuron is the first neuron (index '1') in layer *l*.

Neuron Software Implementation

- To implement a neuron, we simplify the notation. We have a row vector w (for j fixed), a column vector a, and a scalar value b. The result is a scalar a_o, where the scalar z denotes the membrane potential.

$$a\_o = \sigma(z) = w \times a + b$$

- A serial implementation is listed below.

```
#include <iostream>
#include <stdio.h>      /* printf */
```

```
#include <stdlib.h>
#include <math.h>       /* tanh, log */
using namespace std;

class Neuron {
private: // data members can ONLY be accessed by functions inside the class
  double *a; // neuron input vector. This is the output vector from the neurons in previous layer
  double *w; // vector of weights (associated with the elements of a)
  double b;  // bias

public:
  size_t NI; // number of inputs
  size_t af; // activation function.. 1: ReLU, 2: tanh
  double z;  // membrane potential of neuron
  double a_o; // action potential of neuron

  Neuron () { // Default constructor
    NI = 10; cout << "Default constructor called. Setting NI=10 by default" << endl; }

  Neuron (size_t NI_i) { // Parameterized constructor
    cout << "Parameterized constructor called." << endl;
    NI = NI_i; }

  // overload function call: functor
  void operator() (double *a_i, double *w_i, double b_i, int af_i) {
    int i;
    a = a_i; w = w_i; b = b_i; af = af_i;

    z = 0;
    for (i = 0; i < NI; i++) z = z + a[i]*w[i];
    z = z + b; // membrane potential

    if (af == 1) { if (z >= 0) a_o = z; else a_o = 0; }
    else if (af == 2) { a_o = tanh(z); }
    else a_o = z; // invalid activation function, no processing
  }

  void display_results () {
    cout << "Membrane potential (z): " << z << endl;
    cout << "Action potential (a_o): " << a_o << endl; }
};

int main() {
  size_t i;
  double b, result;
  double *a, *w;
  size_t NI = 10; // number of inputs.

  Neuron AN(NI); // AN.NI = NI -> Parameterized constructor.

  // Defining the inputs, as well as weights and bias of the Neuron.
    a = (double *) calloc (NI, sizeof(double));
    w = (double *) calloc (NI, sizeof(double));
    for (i = 0; i < NI; i++) { a[i] = 0.75; w[i] = -0.5; }
    b = 1.25;

    AN(a,w,b,1); // af = 1 (ReLU), af = 2 (tanh)

    AN.display_results(); // display 'z' and 'a_o'

    // we can also get AN.z and AN.ao individually:
    cout << "AN.z: " << AN.z << endl;
    cout << "AN.a_o: " << AN.a_o << endl;
    free (a); free(w);
    return 0;
}
```

- Application file: `neuron_imp.cpp`
- Compile this code: `g++ neuron_imp.cpp -o neuron_imp ↵`
- Execute this application: `./neuron_imp ↵`
  - ✓ Program Output:

```
Membrane potential (z): -2.5
Action potential (a_o): 0
AN.z: -2.5
AN.a_o: 0
```