ECE 3640 - Discrete-Time Signals and Systems

C: Basic Knowledge and Skills

Jake Gunther

Spring 2015

Department of Electrical & Computer Engineering

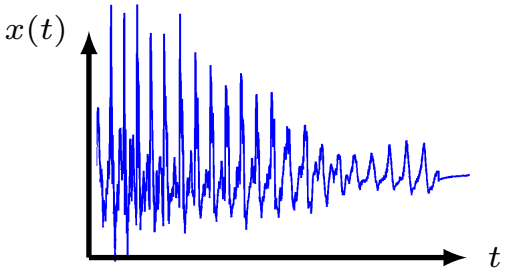
outline

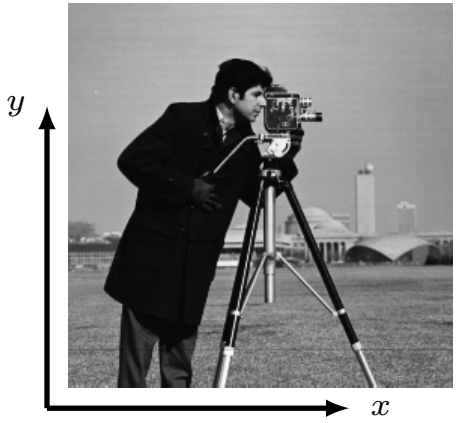
- different kinds of signals (dimension, number of channels)
- arrays in C and data types
- model of computer memory and process virtual memory
- allocation on the stack and heap
- indexing multidimensional arrays
- application: combine two monaural audio to stereo audio
- application: convert RGB color video to grayscale video

representing signals in computer memory

different kinds of signals

signal	dimension	number channels
audio (monaural)	1	1
audio (stereo)	1	2
image (grayscale)	2	1
image (RGB color)	2	3
video (grayscale)	3	1
video (RGB color)	3	3





multidimensional arrays in C

- signal dimension and array dimension are different
- array dimension = signal dimension + $\begin{cases} 0, & \text{number channels} = 1, \\ 1, & \text{number channels} > 1 \end{cases}$
- example: one dimensional array

```
// a 1D-1Ch signal (mono), 5 sec.,
// 8000 samples/sec., 16 bits/sample
short x[40000];
```

- example: two dimensional array

```
// a 1D-2Ch signal (stereo), 5 sec.,
// 8000 Samples/sec., 32 bits/sample
float x[40000][2];

// a 2D-1Ch signal = grayscale image 100 X 100 pixels,
// 8 bits/pixel
unsigned char x[100][100];
```

multidimensional arrays in C

- example: three dimensional array

```
// a 2D-3Ch signal = RGB color image
unsigned char x[100][100][3];

// a 2D-1Ch signal = grayscale video
unsigned char x[600][100][100];
```

- example: four dimensional array

```
// a 2D-3Ch signal = RGB color video
unsigned char x[600][100][100][3];
```

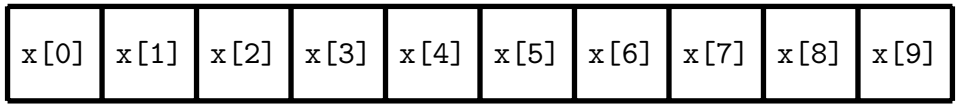
multidimensional arrays for signals

signal	signal dim.	number channels	array dim.
audio (monaural)	1	1	1
audio (stereo)	1	2	2
image (grayscale)	2	1	2
image (RGB color)	2	3	3
video (grayscale)	3	1	3
video (RGB color)	3	3	4

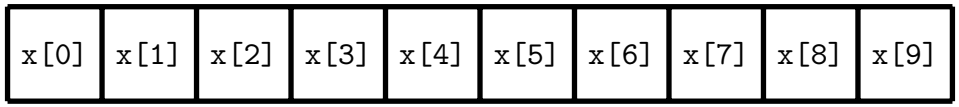
signal	MD-array indexing	array dimensions
audio (monaural)	`x[n]`	1
audio (stereo)	`x[n][chan]`	2
image (grayscale)	`x[row][col]`	2
image (RGB color)	`x[row][col][rgb]`	3
video (grayscale)	`x[frame][row][col]`	3
video (RGB color)	`x[frame][row][col][rgb]`	4

memory layout for 1D arrays in C

- signal is a list/sequence of samples



- array in memory is a linear arrangement of samples



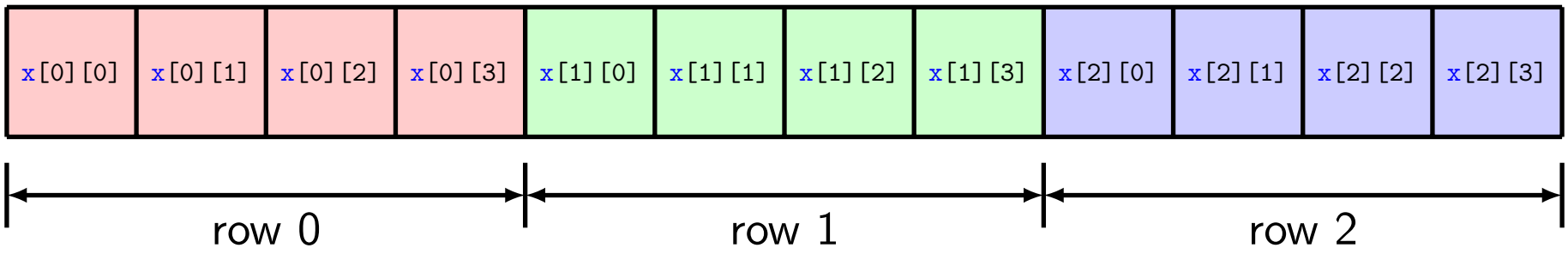
- to access the `n`-th sample in the array, use `x[n]`, for `n = 0, 1, 2, ...`

memory layout for 2D arrays in C

- signal is a 2D table of values

x[0][0]	x[0][1]	x[0][2]	x[0][3]
x[1][0]	x[1][1]	x[1][2]	x[1][3]
x[2][0]	x[2][1]	x[2][2]	x[2][3]

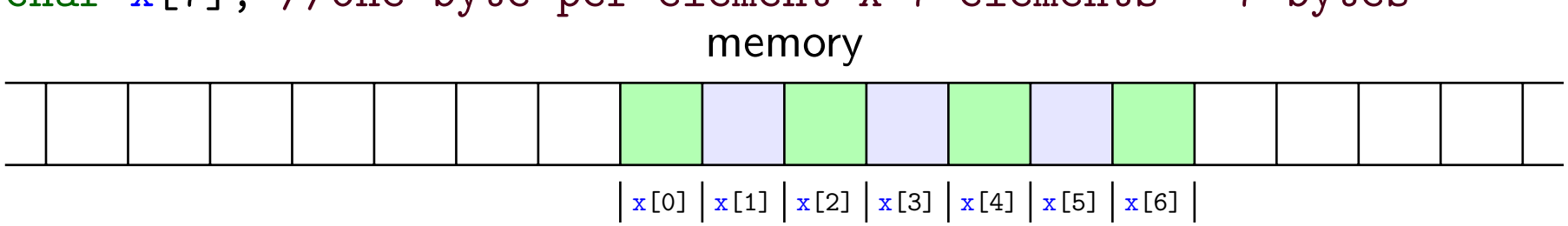
- array in memory is a linear arrangement of values



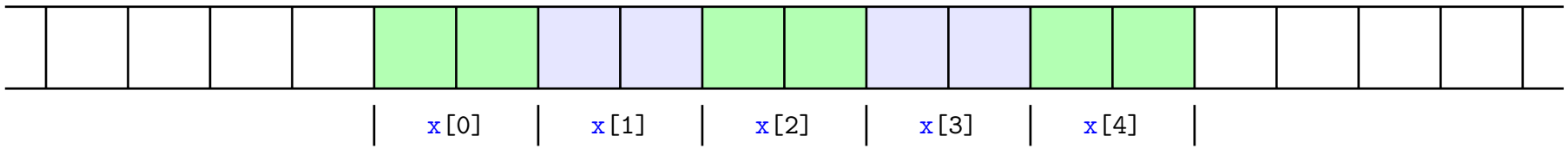
- to access the (m,n)-th value in the table, use `x[m][n]` or `x[m*NUMCOLS + n]`

data types in 1D arrays

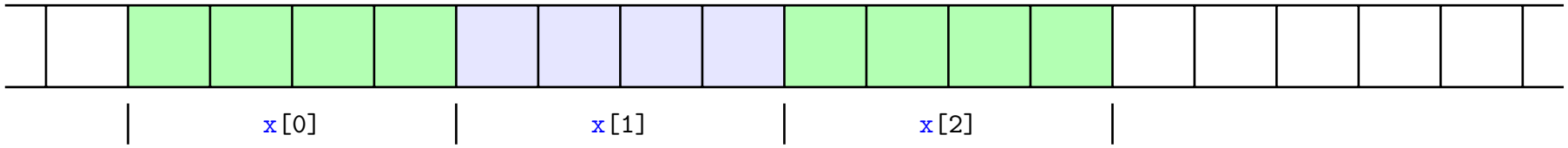
- `char x[7]; //one byte per element X 7 elements = 7 bytes`



- `short x[5]; //two bytes per element X 5 elements = 10 bytes`



- `float x[3]; //four bytes per element X 3 elements = 12 bytes`



data types and memory

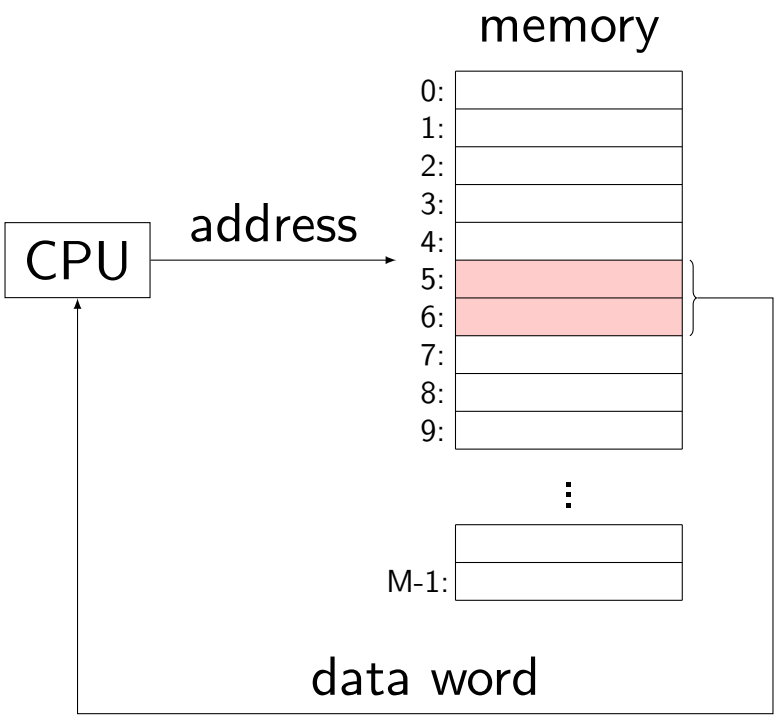
Type	Explanation
char	smallest addressable unit of the machine that can contain basic character set (8 bits). It is an integer type.
signed char	same size as char, but guaranteed to be signed.
unsigned char	same size as char, but guaranteed to be unsigned.
short short int signed short signed short int	short signed integer type ($\geq$ 16 bits). At least in the [-32767,+32767] range.
unsigned short unsigned short int	same as short, but unsigned
int signed int	basic signed integer type ($\geq$ 16 bits). At least in the [-32767,+32767] range.
unsigned unsigned int	same as int, but unsigned
long long int signed long signed long int	long signed integer type ($\geq$ 32 bits). At least in the [-2147483647,+2147483647] range.
unsigned long unsigned long int	same as long, but unsigned
long long long long int signed long long signed long long int	long long signed integer type ($\geq$ 64 bits). At least in the [-9223372036854775807,+9223372036854775807] range.
unsigned long long unsigned long long int	same as long long, but unsigned
float	IEEE 754 single-precision binary floating-point format (32 bits).
double	IEEE 754 double-precision binary floating-point format. (64 bits).
long double	It can be: 80-bit floating point, IEEE 754 quadruple-precision floating-point format, or the same as double.

Source: Wikipedia

memory model, stack, heap & indexing

model of computer memory

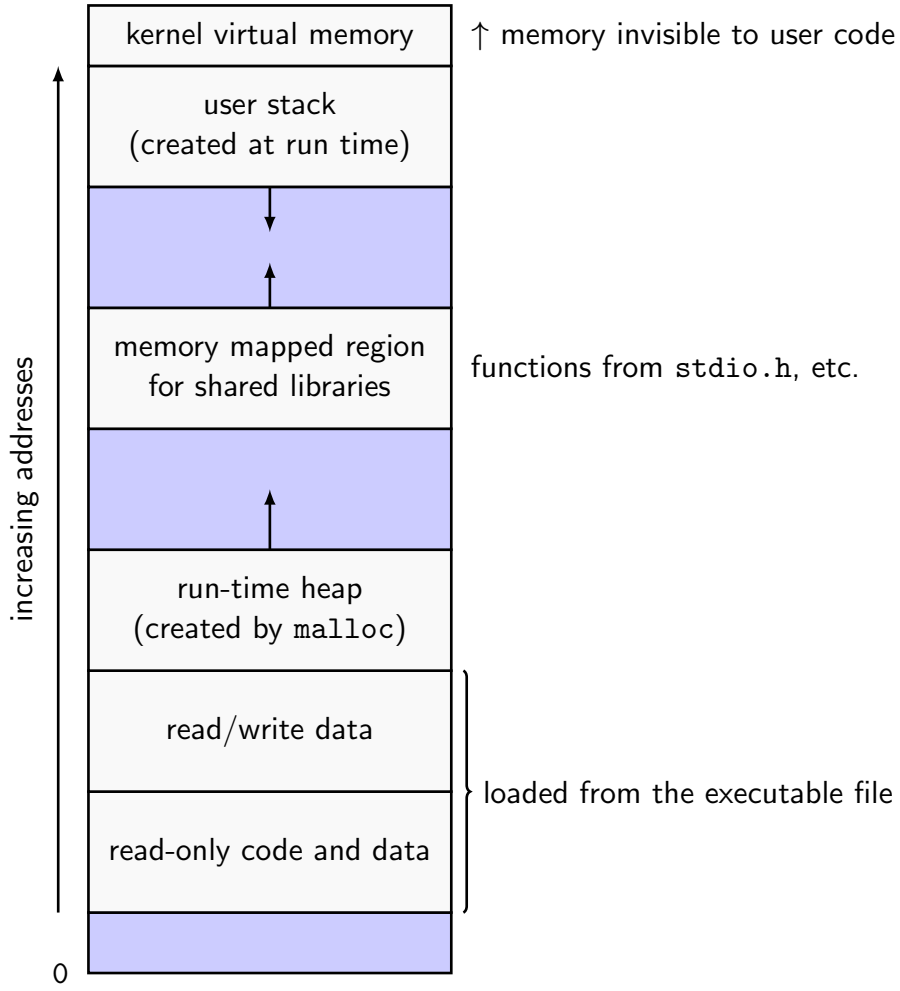
- ideal: memory is an semi-infinite linear array of bytes
- practical: memory is a large linear array of bytes
- CPU requests address and type (number of bytes), memory returns data word



virtual memory in Linux

- virtual memory = abstraction providing each process with the illusion that it has exclusive use of memory
- virtual address space gives each process a uniform view of memory

process virtual address space
source: Bryant, O'Hallaron, *Computer Systems: A Programmer's Perspective*, Prentice Hall, 2011.



1D and 2D arrays on the stack (static allocation)

- if the array size is known at compile time
- 1D array

```
#define LEN 40000
char x[LEN]; // declaration allocates space on stack
x[5] = x[4] + x[3]; // example of array access
```

- 2D array

```
#define NROWS 50
#define NCOLS 100
char x[NROWS][NCOLS]; // declaration allocates space on stack
x[5][10] = x[4][10] + x[3][10]; // example of array access
```

- 2D array as 1D array

```
#define NROWS 50
#define NCOLS 100
char x[NROWS*NCOLS]; // declaration allocates space on stack
x[5*NCOLS+10] = x[4*NCOLS+10] + x[3*NCOLS+10]; // same as above
```

- the rightmost subscript varies fastest as elements are accessed in memory

1D and 2D arrays on the heap (dynamic allocation)

- if the array size is known at run time
- 1D array

```
int len=40000;
// allocate space on heap
float *x = (float*)calloc(sizeof(float),len);
x[5] = x[4] + x[3]; // example of array access
*(x+5) = *(x+4) + *(x+3); // same thing
```

- 2D array as 1D array

```
int R=50;  /* number of rows    */
int C=100; /* number of columns */
// allocate space on heap
float *x = (float*)calloc(sizeof(float),R*C);
x[5*C+10] = x[4*C+10] + x[3*C+10]; // only way to do it
x[5][10] = x[4][10] + x[3][10]; // illegal
```

- the rightmost subscript varies fastest as elements are accessed in memory

3D arrays

- static allocation

```
#define NFRAMES 600
#define NROWS 100
#define NCOLS 100
float x[NFRAMES][NROWS][NCOLS];
x[t][m][n] = ...; \\ t-th frame, (m,n)-th pixel
```

- dynamic allocation

```
int nframes=600;
int nrows=100;
int ncols=100;
float *x = (float*)calloc(sizeof(float),nframes*nrows*ncols);
// x[t][m][n] = n-th frame, (m,n)-th pixel
x[t*nrows*ncols + m*ncols + n] = ...;
```

- the rightmost subscript varies fastest as elements are accessed in memory

4D arrays

- static allocation

```
#define NFRAMES 600
#define NROWS 100
#define NCOLS 100
#define NCHAN 3 /* number of color planes for RGB image*/
float x[NFRAMES][NROWS][NCOLS][NCHAN];
// t-th frame, (m,n)-th pixel, i-th RGB value
x[t][m][n][i] = 3.14159265358979323846;
```

- the rightmost subscript varies fastest as elements are accessed in memory

4D arrays

- dynamic allocation

```
int nframes=600;
int nrows=100;
int ncols=100;
int nchan=3; /* number of color planes for RGB image*/
float *x = ...
  (float*)calloc(sizeof(float),nframes*nrows*ncols*nchan);
// x[t][m][n][i] = n-th frame, (m,n)-th pixel, i-th RGB value
x[t*nrows*ncols*nchan + m*ncols*nchan + n*nchan + i] = ...;
x[((t*nrows + m)*ncols + n)*nchan + i] = ...;
```

- there are efficient indexing schemes when the array sizes are powers of 2

- the rightmost subscript varies fastest as elements are accessed in memory

reverse indexing in 2D arrays

- for linearized $M \times N$ array: convert linear index i to table index (m, n)

$$i = mN + n$$

```
m = i/N;
n = i-m*N;
```

- integer division in C truncates fractions, i.e. throws away remainders

reverse indexing in 3D arrays

- for linearized $M \times N \times P$ array: convert linear index i to table index (m, n, p)

$$i = mNP + nP + p = (mN + n)P + p$$

```
m = i/NP;
n = (i-m*NP)/P;
p = i-m*NP-n*P;
```

or

```
m = i/NP;
i -= m*NP;
n = i/P;
i -= n*P;
p = i;
```

reverse indexing in 4D arrays

- for linearized $M \times N \times P \times Q$ array: convert linear index i to table index (m, n, p, q)

$$i = mNPQ + nPQ + pQ + q = ((mN + n)P + p)Q + q$$

```
m = i/NPQ;
n = (i-m*NPQ)/PQ;
p = (i-m*NPQ-n*PQ)/Q;
q = i-m*NPQ-n*PQ-p*Q;
```

or

```
m = i/NPQ;
i -= m*NPQ;
n = i/PQ;
i -= n*PQ;
p = i/Q;
i -= p*Q;
q = i;
```

signal interface to C ... Matlab

signal processing in a computer

- to process signals in a C program
  1. transfer signal samples from the input interface (file, ADC, etc.) to computer memory
  2. format input samples (need compatibility with operations to be performed)
  3. signal processing - convert input samples to output samples
  4. format output samples (need compatibility with output interface)
  5. transfer signal samples from computer memory to output interface (file, DAC, etc.)

- ECE 3640 focuses on the signal processing (Step 3)

- use simple, fixed interface to Matlab to remove Steps 2 and 4

- use Matlab for visualization and interfacing to media file formats

processing, data types, file format

- in ECE 3640 processing will be done in C
- audio, image and video data files are commonly compressed (need C libraries to uncompress them)
- use Matlab for visualization and interfacing to formatted and compressed file types
- exchange raw floatoing point data between Matlab and C programs
- raw audio data often stored as signed 8-bit or 16-bit samples (e.g. .wav files), but we will use float in most of our C programs
- raw image data often stored as unsigned 8-bit (e.g. .pgm and .ppm files), but we will use float in most of our C programs
- raw video data often stored as unsigned 8-bit, but we will use float in most of our C programs

use simple file header

```
typedef struct
{
  int ndim;
  // signal (1)
  // image  (2)
  // video  (3)
  int nchan;
  // signal (1=mono, 2=stereo, etc.)
  // grayscale image/video (1)
  // color image/video (3)
  int dim0;
  // signal => length
  // image or video => number rows
  int dim1;
  // signal => if audio, then dim1 = sample rate
  // image or video => number columns
  int dim2;
  // signal => 0
  // image  => 0
  // video  => number_frames
} dsp_file_header;
```

reading and writing files in C

```
dsp_file_header h;

// file input and memory allocation
FILE *i = fopen("infile","rb");
fread(&h,sizeof(dsp_file_header),1,i);
int nsamples = nchan*d0*(d1==0?1:d1)*(d2==0?1:d2);
float *x = (float*)calloc(sizeof(float),nsamples);
fread(x,sizeof(float),nsamples,i);
fclose(i);

// file output and memory deallocation
FILE *o = fopen("outfile","wb");
fwrite(&h,sizeof(dsp_file_header),1,o);
fwrite(x,sizeof(float),nsamples,o);
fclose(o);
free(x);
```

- using `float` data type simplifies calculation and file interface

reading and writing files in Matlab

```
1 i = fopen('infile','rb');
2 ndim  = fread(i,1,'int');
3 nchan = fread(i,1,'int');
4 dim0  = fread(i,1,'int');
5 dim1  = fread(i,1,'int');
6 dim2  = fread(i,1,'int');
7 nsamples = nchan*dim0*((dim1==0)+dim1)*((dim2==0)+dim2);
8 x = fread(i,nsamples,'float');
9 if(ndim==1) % signal
10   x = reshape(x,nchan,dim0);
11   x = permute(x,[2 1]);
12 elseif(ndim==2) % image
13   x = reshape(x,nchan,dim1,dim0);
14   x = permute(x,[3 2 1]);
15 elseif(ndim==3) % video
16   x = reshape(x,nchan,dim1,dim0,dim3);
17   x = permute(x,[3 2 1 4]);
18 end
```

- probably don't want to load entire video into memory
- process video one frame (or a few frames) at a time

assignment

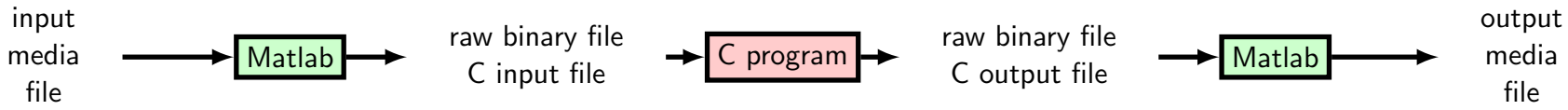
assignment

1. combine two monaural audio signals into a stereo signal (use `f1.wav` and `f2.wav`)

2. convert RGB color video to grayscale video (use `xylophone.mp4`)

3. for raw binary file formats see `ece3640_basic_matlab.pdf`

assignment: monaural audio $\times$ 2 $\longrightarrow$ stereo audio



1. Matlab: read in two monaural audio files (input media files) and write each out to a raw binary file (C input file)

2. C: read in the headers for input raw binary files

3. C: write out header for output raw binary file

4. C: statically allocate two buffers to hold input monaural audio data

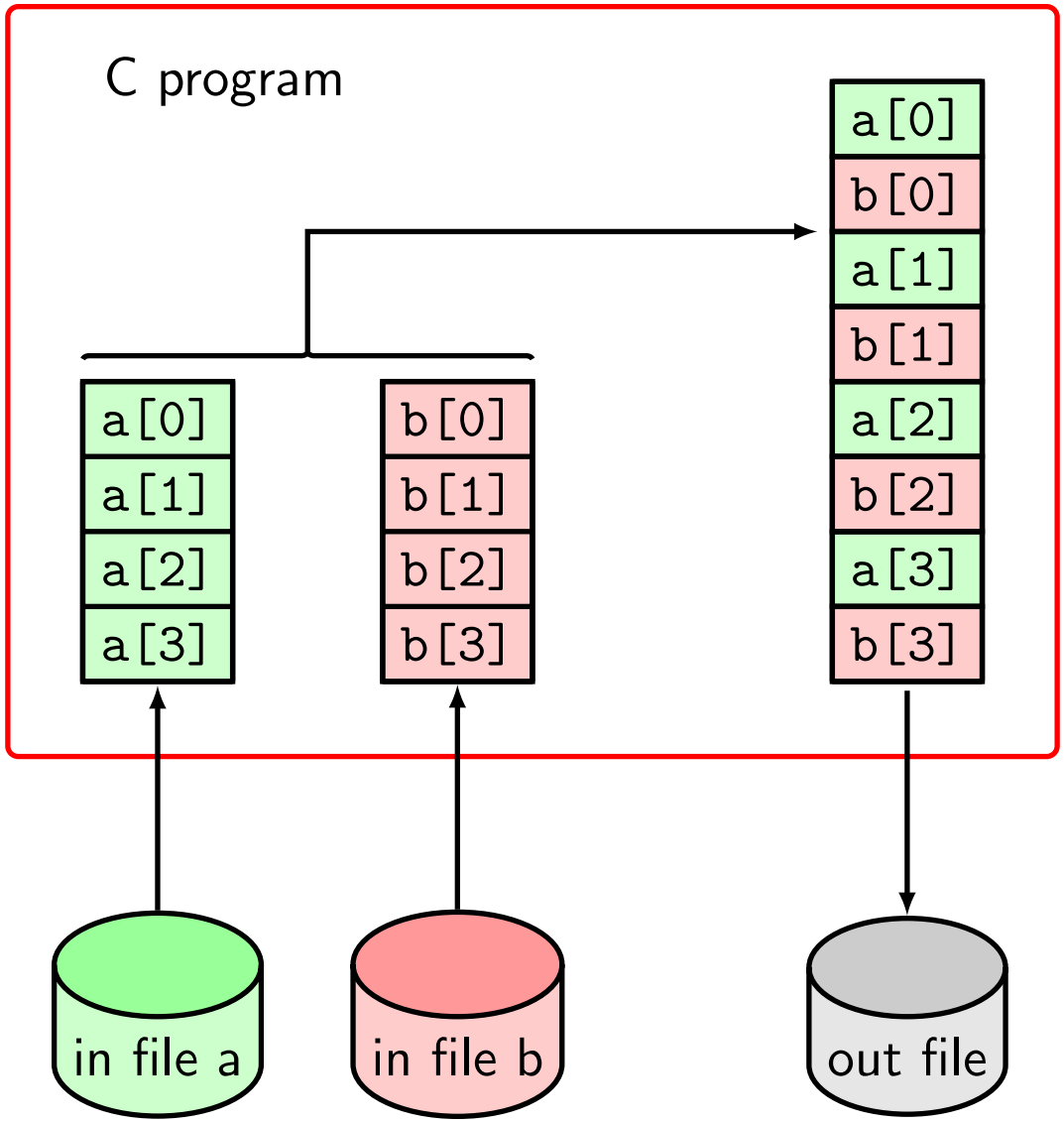
5. C: statically allocate one buffer to hold output stereo audio data

6. C: repeat to the end of the input files

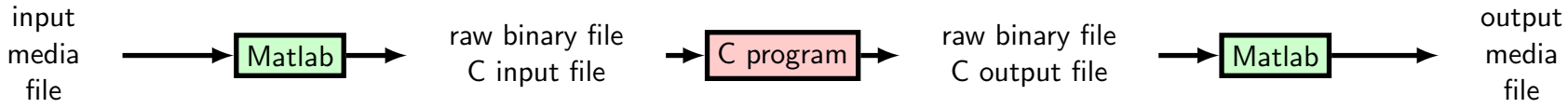
   (a) read in one buffer of audio from each input file
   (b) interleave samples in output buffer
   (c) write out the output buffer

7. C: close files

8. Matlab: read in the raw binary file (C output file) and write it out to output stereo audio (output media file)



assignment: convert RGB color video to grayscale video



1. Matlab: read in a color RGB video (input media file) one frame at a time and write it out to a raw binary file (C input file)

2. C: determine the number of rows and columns from the raw binary file header (C input file)

3. C: write out header for grayscale video (C output file)

4. C: dynamically allocate memory to hold one frame of RGB video

5. C: dynamically allocate memory to hold one frame of grayscale video

6. C: repeat to the end of the raw binary file

   (a) read in one frame of RGB video (C input file)

(b) convert to grayscale using the formula:

$$GRAY = (0.2989)R + (0.5870)G + (0.1140)B$$

(c) write out the grayscale frame (C output file)

7. C: close files

8. Matlab: read in the raw binary file (C output file) one frame at a time and write it out to a grayscale video (output media file)