SimCenter NHERI
Center for Computational Modeling and Simulation

Programming Bootcamp

C: Pointers Revisited

Frank McKenna
University of California at Berkeley

NSF award: CMMI 1612843

Todays Outline

- 9-9:20 PI and stressTransform (20min)
- 9:20-9:40 Memory, Pointers & Arrays (20min)
- 9:40-10:10 Exercise: dgemm operation: C = C + A*B (30min)
- BREAK
- 10:15-10:30 Abstraction (15 min)
- 10:30-11:10 Exercise: stressTransform with structures (40min)
- BREAK
- 11:15-11:15 File I/O (15min)
- 11:15-12:00 Exercise: stressTransform – read and write (45min)

- Exercises: advanced options available.

SimCenter NHERI
Center for Computational Modeling and Simulation

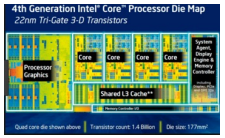
Programming Bootcamp

C: Pointers Revisited

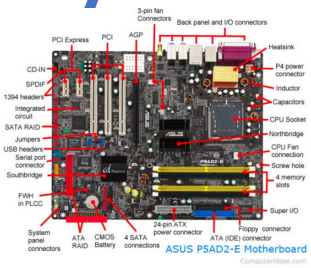
Frank McKenna
University of California at Berkeley

NSF award: CMMI 1612843

Computer Memory Hierarchy



Core Processor



Disk

Control

Arithmetic

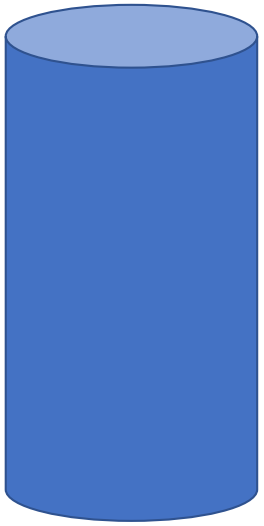
Registers

L1 Cache

L2 Cache

L3 Cache

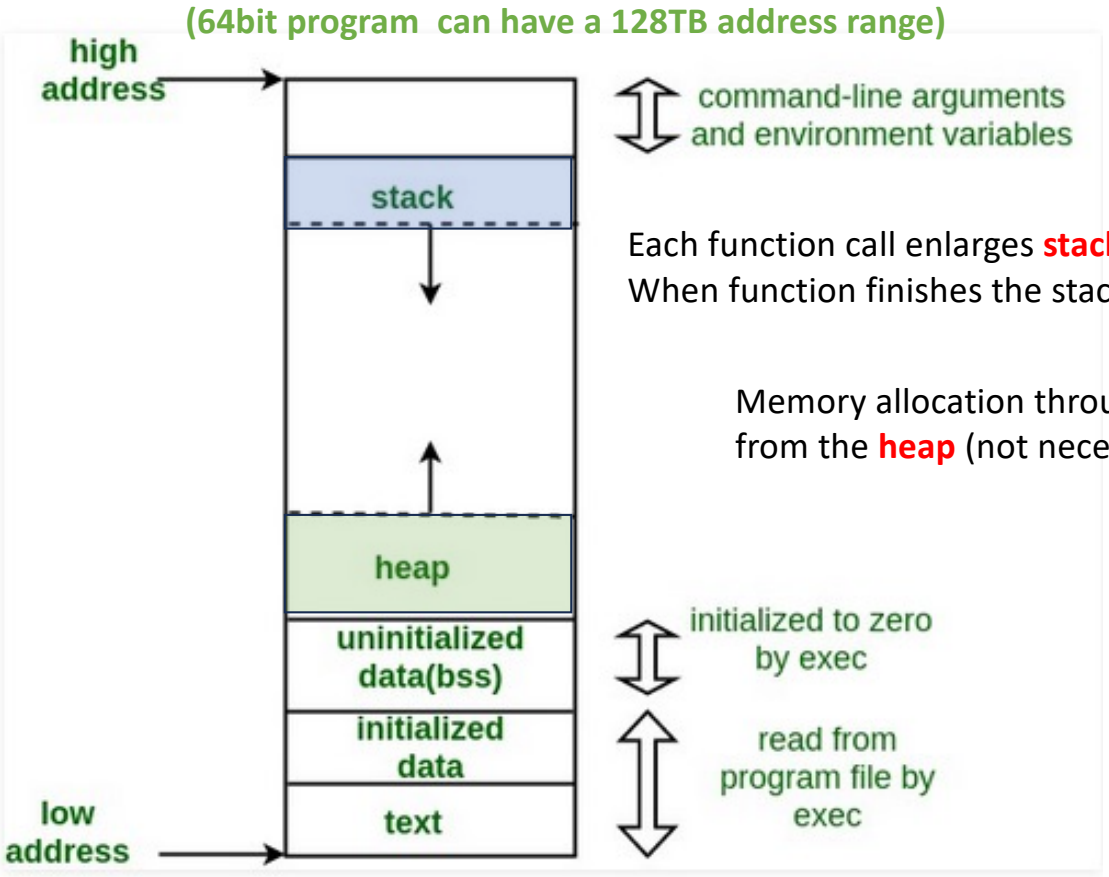
Me*mory(RAM)*



Hard Drive **SSD**

Size	1000 Bytes	64 KB	256 KB	2-4 MB	4-32 GB	4-16 TB	*.25-1TB*
Latency	0.3 ns	1 ns	3-10 ns	20-30 ns	50-100 ns	5-10e6 ns	25-50e3 ns
	Compiler	HW	HW	HW	Operating System	Operating System	

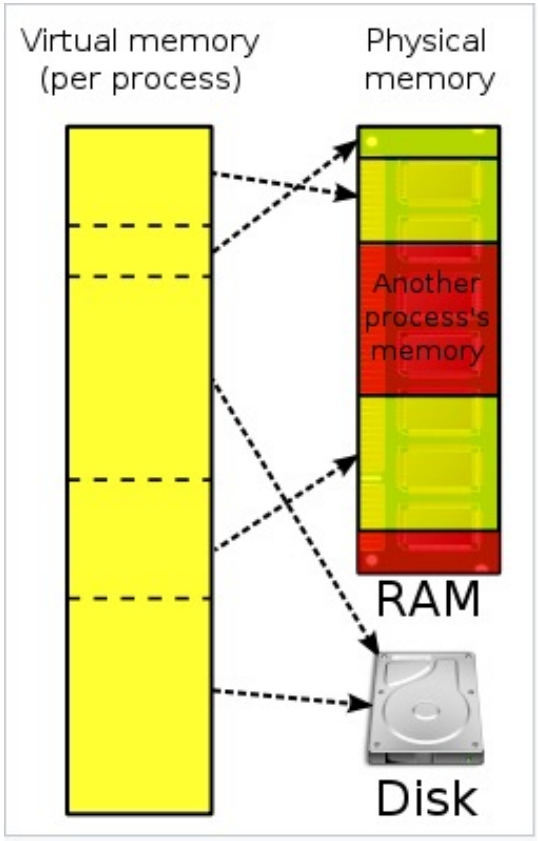
Memory Layout of a RUNNING Program



Each function call enlarges **stack** with the function variables
When function finishes the stack is popped

Memory allocation through malloc() and new[] is allocated memory from the **heap** (not necessarily one chunk after another)

Operating System & Virtual Memory



- Virtual Memory is a memory management technique that provides an "idealized abstraction of the storage resources that are actually available on a given machine" wikipedia.
- Program Memory is broken into a number of pages. Some of these are in memory, some on disk, some may not exist at all (segmentation fault)
- CPU issues virtual addresses (load b into R1) which are translated to physical addresses. If page in memory, HW determines the physical memory address. If not, page fault, OS must get page from Disk.
- Page Table: table of pages in memory.
- Page Table Lookup – relatively expensive.
- Page Fault (page not in memory) very expensive as page must be brought from disk by OS
- Page Size: size of pages
- TLB Translation Look-Aside Buffer HW cache of virtual to physical mappings.
- Allows multiple programs to be running at once in memory.

WARNING

- Arrays and Pointers are the **source of most bugs in C Code**
  - You will have to use them if you program in C
  - Always initialize a pointer to 0
  - Be careful you do not go beyond the end of an array
    - Be thankful for segmentation faults
    - If you have a race condition (get different answers every time you run, probably a pointer issue)

Pointer Variables

A **Pointer Variable** is a named memory location that can holds an address**.**

1. The unary * in a declaration indicates that the object is a pointer to an object of a specific type

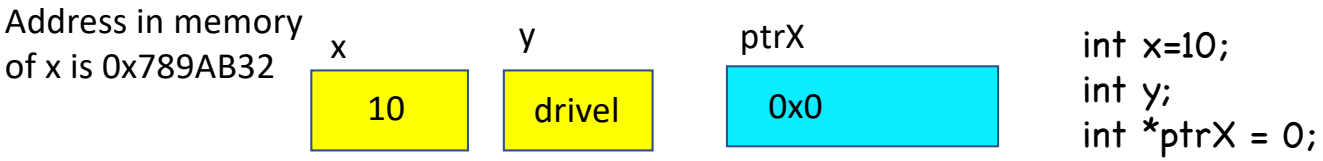
```
#include <stdio.h>
int main() {
    int x =10, y;
    int *ptrX =0;



}
```

pointer1.c

1.



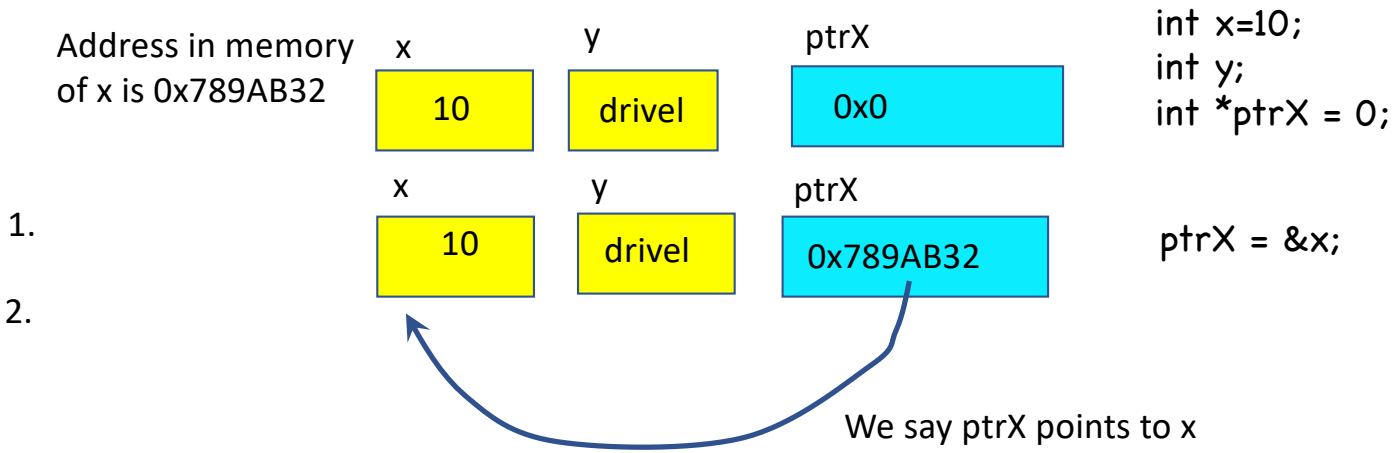
Pointer Variables

A **Pointer Variable** is a named memory location that can holds an address**.**

1. The unary * in a declaration indicates that the object is a pointer to an object of a specific type
2. The unary **&** gives the "**address**" of an object in memory, e.g. &x is location of variable x

```
#include <stdio.h>
int main() {
    int x =10, y;
    int *ptrX =0;
    ptrX = &x;
```

pointer1.c



Pointer Variables

A **Pointer Variable** is a named memory location that can holds an address**.**

1. The unary * in a declaration indicates that the object is a pointer to an object of a specific type
2. The unary **&** gives the "**address**" of an object in memory, e.g. &x is location of variable x
3. The unary * elsewhere treats the operand as an address and dereferences it, which depending on which side of operand is does one of two things:

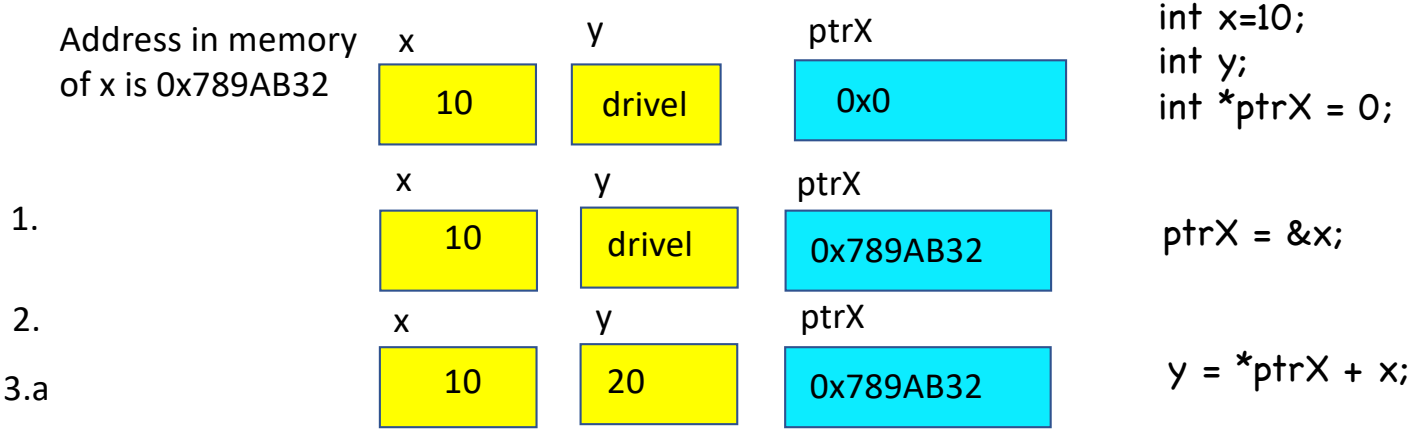
    a. fetches the contents for use in an expression

```
#include <stdio.h>
int main() {
    int x =10, y;
    int *ptrX =0;
    ptrX = &x;
    y = *ptrX + x;

}
```

pointer1.c

Address in memory of x is 0x789AB32



Pointer Variables

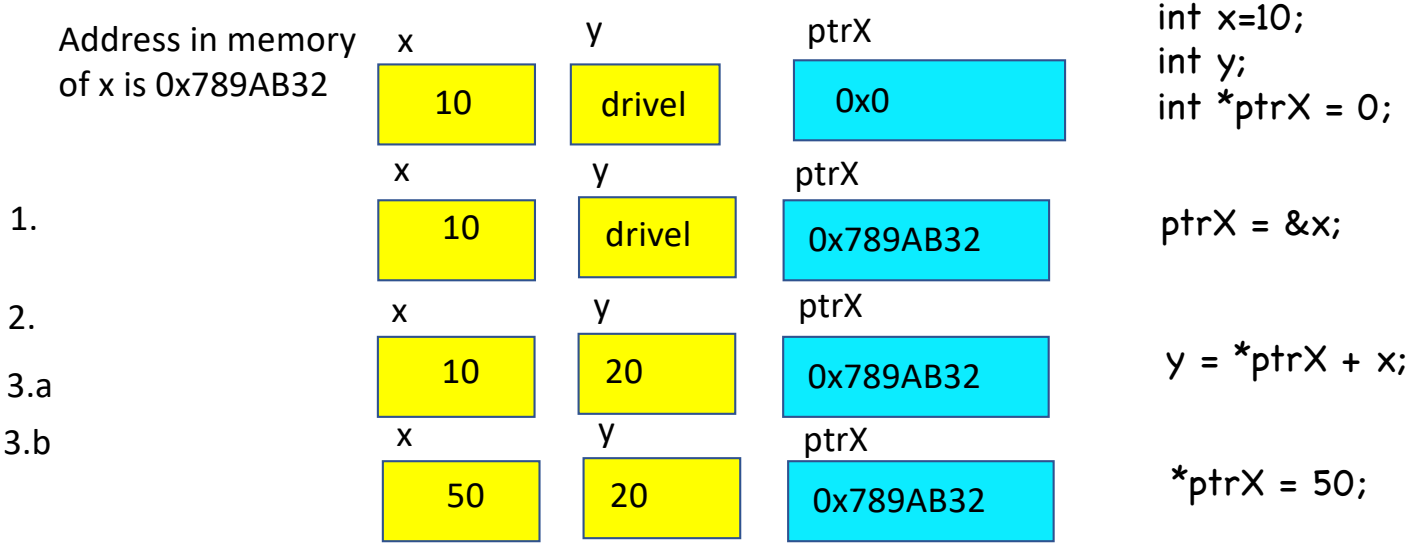
A **Pointer Variable** is a named memory location that can holds an address**.**

1. The unary * in a declaration indicates that the object is a pointer to an object of a specific type
2. The unary **&** gives the "**address**" of an object in memory, e.g. &x is location of variable x
3. The unary * elsewhere treats the operand as an address and dereferences it, which depending on which side of operand is does one of two things:
   a. fetches the contents for use in an expression
   b. Sets the memory location to which it points to some value.

```
#include <stdio.h>
int main() {
    int x =10, y;
    int *ptrX =0;
    ptrX = &x;
    y = *ptrX + x;
    *ptrX = 50;
}
```

pointer1.c

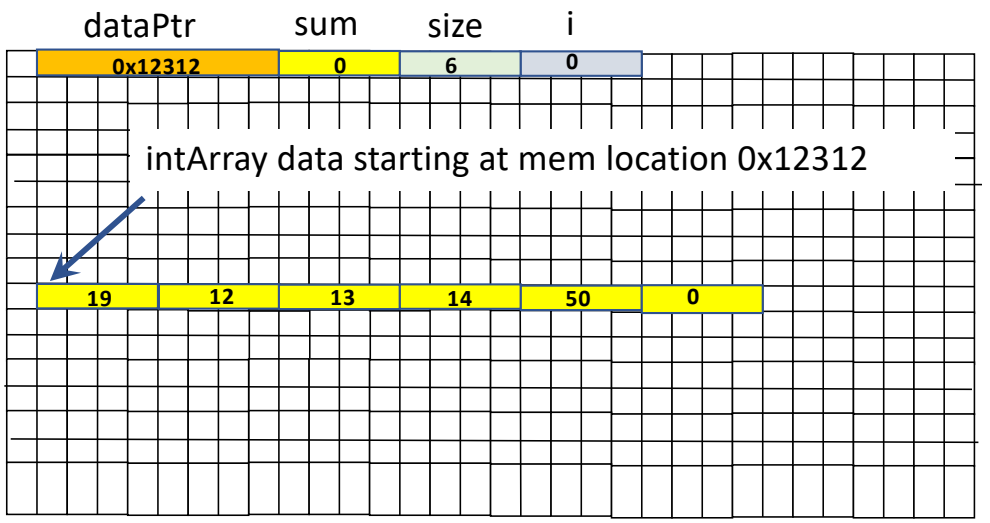
Address in memory of x is 0x789AB32



Iterating Through Arrays With Pointers

```
#include <stdio.h>
int sumArray(int *arrayData, int size);
int main(int argc, char **argv) {
  int intArray[6] = {19, 12, 13, 14, 50, 0};
  int sum1 = sumArray(intArray, 6);
  printf("sum: %d\n", sum1);
  return(0);
 }

// function to evaluate vector sum
int sumArray(int *dataPtr, int size) {
  int sum = 0;
  for (int i = 0; i < size; i++) {
    sum += *dataPtr;
    dataPtr++;
  }
  return sum;
}
```



I	dataPtr	*dataPtr	sum
0	0x12312	19	19
1	0X12316	12	31
2	Ox1231A	13	44
3	Ox1231E	14	58
4	Ox12322	50	108
5	Ox12326	0	108

0
1
2
3
4
5
6
7
8
9
A
B
C
D
E
F

SimCenter NHERI
Center for Computational Modeling and Simulation

Programming Bootcamp

C: Arrays, Pointers & Memory Allocation

Frank McKenna
University of California at Berkeley

NSF award: CMMI 1612843

Arrays - II

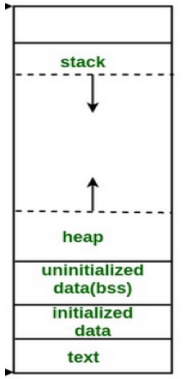
- An array is fixed size sequential collection of elements all of the same type. The are contiguously stored in memory. We access using an index inside a square brackets or by dereferencing pointers.
- to declare: type arrayName [size];

  type arrayName [size] = {size comma separated values}
- Works for arrays where we know the size at compile time. There are many times when we do not know the size of the array. There are other times

Were we want the array to persist after function has been popped from stack.

- Now we need to use **pointers** and the functions **free()** and **malloc()**

```
type *thePointer = (type*)malloc(numElements*sizeof(type));
  ...
free(thePointer)
```



- Memory for the array using free() comes from the heap
- **Always remember to free() the memory** .. Otherwise can run out of memory.

pointer, malloc() and free()

memory1.c

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

int main(int argc, char **argv) {
  if (argc != 2) {
      fprintf(stderr,"Need 3 args: appName n\n");
      return -1;
  }
  double *array1=0;
  int n = atoi(argv[1]);

  // allocate memory & set the data
  array1 = (double *)malloc(n*sizeof(double));
  for (int i=0; i<n; i++) {
    array1[i] = i;
  }
  for (int i=0; i<n; i++, array3++) {
    double value1 = array1[i];
    printf("%.4f %p\n", value1, &array1[i]);
  }
  // free the array
  free(array1);
  return(0);
}
```

```
c >./a.out 4
0.0000  0x7fab094059a0
1.0000  0x7fab094059a8
2.0000  0x7fab094059b0
3.0000  0x7fab094059b8
c >
```

pointer, malloc() and free()

memory2.c

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char **argv) {
  if (argc != 2) {
      fprintf(stderr,"Need 3 args: appName n\n");
      return -1;
  }
  double *array1=0, *array2=0, *array3=0;
  int n = atoi(argv[1]);

  // allocate memory & set the data
  array1 = (double *)malloc(n*sizeof(double));
  for (int i=0; i<n; i++) {
    array1[i] = i;
  }
  array2 = array1;
  array3 = &array1[0];

  for (int i=0; i<n; i++, array2++, array3++) {
     double value1 = array1[i];
    double value2 = *array2;
    double value3 = *array3;
    printf("%.4f %.4f %.4f %p %p %p\n",
      value1, value2, value3, &array1[i], array2, array3);
  }
  // free the array
  free(array1);
  return(0);
}
```

```
c >gcc memory2.c; ./a.out 4
0.0000 0.0000 0.0000 0x7f8dd84059a0 0x7f8dd84059a0 0x7f8dd84059a0
1.0000 1.0000 1.0000 0x7f8dd84059a8 0x7f8dd84059a8 0x7f8dd84059a8
2.0000 2.0000 2.0000 0x7f8dd84059b0 0x7f8dd84059b0 0x7f8dd84059b0
3.0000 3.0000 3.0000 0x7f8dd84059b8 0x7f8dd84059b8 0x7f8dd84059b8
```

Pointers to pointers & multi-dimensional arrays

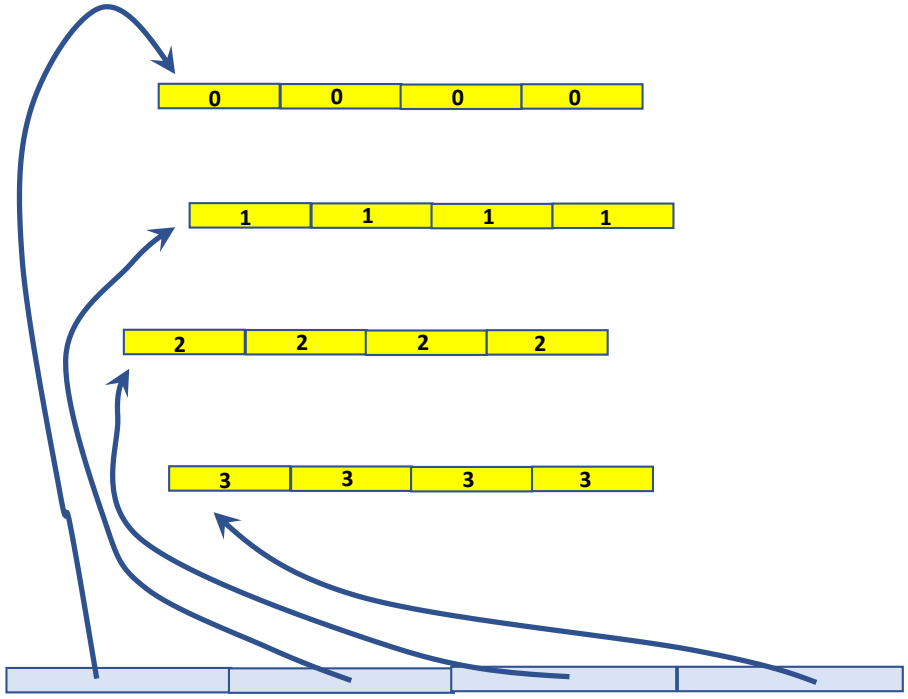
memory3.c

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

 int main(int argc, char **argv) {
    if (argc != 2) {
      fprintf(stderr,"Need 3 args: appName n\n");
      return -1;
  }
  int n = atoi(argv[1]);
  double **matrix1 =0;

  // allocate memory & set the data
  matrix1 = (double **)malloc(n*sizeof(double *));
  for (int i=0; i<n; i++) {
     matrix1[i] = (double *)malloc(n*sizeof(double));
     for (int j=0; j<n; j++)
        matrix1[i][j] = i;
  }
  for (int i=0; i<n; i++) {
     for (int j=0; j<n; j++)
        printf("(%d,%d) %.4f\n", i,j, matrix1[i][j]);
  }
  // free data
  for (int i=0; i<n; i++)
     free(matrix1[i]);
  free(matrix1);
}
```

Case for n = 4

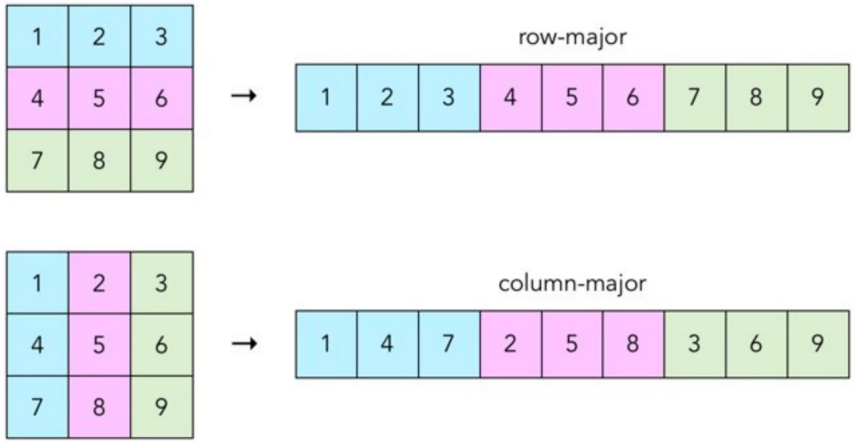


matrix1

matrix 1 is an array of pointers to double *, i.e. an each component of the matrix1 points to an array

for Compatibility with many matrix libraries this is poor code:

```
double **matrix2 =0;
matrix2 = (double  **)malloc(numRows*sizeof(double  *));
 for (int i=0;  i<numRows;  i++)  {
     matrix2[i]  =  (double  *)malloc(numCols*sizeof(double));
      for (int j=0;  j<numCols;  j++)
          matrix2[i][j]  =  i;
 }
```



Because many prebuilt libraries work assuming continuous layout and Fortran column-major order:

```
double  *matrix2 =0;
matrix2 = (double  *)malloc(numRows*numCols*sizeof(double  *));
 for  (int  i=0;  i<numRows;  i++)  {
     for  (int  j=0;  j<numCols;  j++)
         matrix2[I + j*numRows]  =
 }
```

```
double *matrix2 =0;
matrix2 = (double  *)malloc(
 for (int j=0;  j<numCols;  j
     for (int i=0;  i<numRow
         matrix2[I + j*num
 }
```

```
double **matrix2 =0;
matrix2 = (double  **)malloc(numRows*numCols*sizeof(double  *));
 double *dataPtr = matrix2;
 for (int j=0;  j<numCols;  j++)
     for (int i=0;  i<numRows;  i++)  {
         *dataPtr++  =  i;
```

memory3.c

Special Problems with char * and Strings

- No string datatype, string in C is represented by type char *
- There are special functions for strings in <string.h>
  - strlen()
  - strcpy()
  - ….
- To use them requires a special character at end of string, namely **'\0'**
- This can cause no end of grief, e.g. if you use malloc, you need **size+1** and need to append '\0'

```
#include <string.h>
….
char greeting[] = "Hello";
int length = strlen(greeting);
printf("%s a string of length %d\n",greeting, length);

char *greetingCopy = (char *)malloc((length+1)*sizeof(char));
strcpy(greetingCopy, greeting);
```

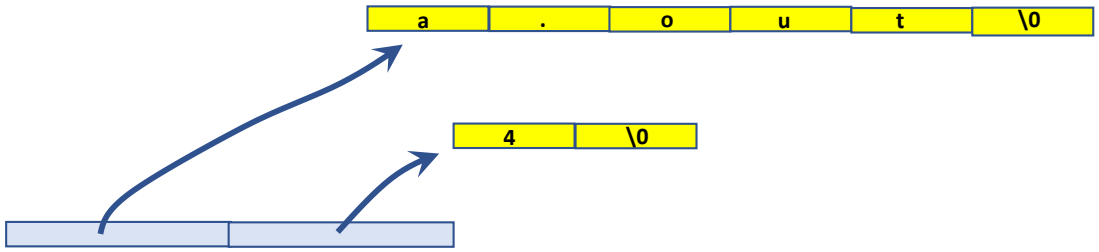
an implementation of strlen:
provided just to show how it will look for `\0`

```
int
strlen(char str[])
{
  int len = 0;
  while (str[len] != '\0')
    len++;
  return (len);
}
```

So Now you can understand char **argv in main!

argv is an array of pointers to char *, i.e. each component of the argv points to a string.

say program started with "./a.out 4"



argv1.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main(int argc, char **argv) {

  printf("Number of arguments: %d\n", argc);

  /* print out location, size and v
      alue of each argument */
  for (int i=0; i<argc; i++) {
    int length = strlen(argv[i]);
    printf("%d %d %s\n", i, length, argv[i]);
  }
  return 0;
}
```

How do you know if the pointer type is pointing to an array or a single value??

```
int x =10, y;
int *ptrX = &x;
```

```
double *array1 = (double *)malloc(n*sizeof(double));
double *dataPtr = array1;
for (int i=0; i<n; i++) {
   *dataPtr++ = i;
 }
```

What would compiler do with the following?

```
for (int i=0; i<n; i++) {
  *ptrX++ = i;
}
```

WARNING

- Arrays and Pointers are the source of most bugs in C Code
  - You will have to use them if you program in C
  - Always initialize a pointer to 0
  - Be careful you do not go beyond the end of an array
    - Be thankful for segmentation faults
    - If you have a race condition (get different answers every time you run, probably a pointer issue)

EXERCISE

CMake

- A widely used application for building cross platform applications and libraries which typically consist of many many different source files.
- Simple few commands to type from a shell window

```
> mkdir build
> cd build
> cmake ..
> cmake --build . --config Release
> cmake --install .
```

- If you install software, there will be a CMakeLists.txt file in source directory
- Still requires you to install dependencies
  - Conan is something that integrates with CMake

PROGRAMMING

DGEMM

$$c_{ij} = c_{ij} + a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj} = c_{ij} + \sum_{k=1}^{n} a_{ik}b_{kj}$$

Hands On – matMul

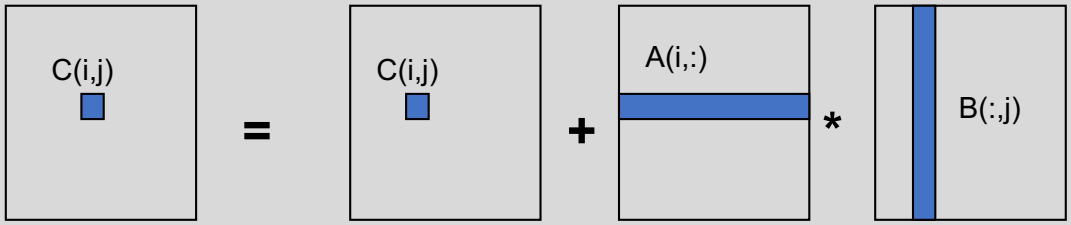
In assignments/C-Day2/matmul there are some files.
CMakeLists.txt will build 2 executables matMul & benchmark

You need to:
1. edit matMul.c (malloc & free functions)
2. edit myDGEMM.c (function needs filling in)
3. when done submit matMul.c

ADVANCED:
1. Run the benchmark exe. It shows GFLOP/s performance of your myDGEMM versus the BLAS dgemm.
2. Can you get at least 30% of the BLAS performance?? (see following 2 slides)

Naïve Matrix Multiply

```
{implements C = C + A*B}
for i = 1 to n
  {read row i of A into fast memory}
   for j = 1 to n
      {read C(i,j) into fast memory}
      {read column j of B into fast memory}
      for k = 1 to n
          C(i,j) = C(i,j) + A(i,k) * B(k,j)
      {write C(i,j) back to slow memory}
```



Source: UC Berkeley, Jim Demmell, CS267

Blocked (Tiled) Matrix Multiply

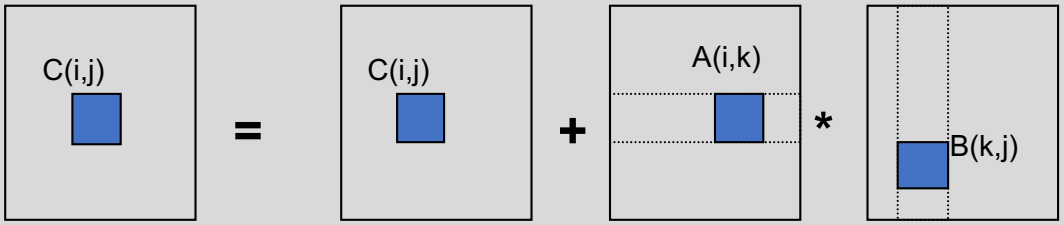
Consider A,B,C to be N-by-N matrices of b-by-b subblocks where b=n / N is block size

```
for i = 1 to N
  for j = 1 to N
    {read block C(i,j) into fast memory}
    for k = 1 to N
      {read block A(i,k) into fast memory}
      {read block B(k,j) into fast memory}
      C(i,j) = C(i,j) + A(i,k) * B(k,j) {do a matrix multiply on blocks}
    {write block C(i,j) back to slow memory}
```

cache does this automatically

3 nested loops inside

block size = loop bounds



Tiling for registers (managed by you/compiler) or caches (hardware)

Source: UC Berkeley, Jim Demmell, CS267

SimCenter NHERI
Center for Computational Modeling and Simulation

Programming Bootcamp

C: File I/O

Frank McKenna
University of California at Berkeley

NSF award: CMMI 1612843

File * - another pointer type, a pointer to a file

File I/O in C is done with the following built in functions:

- **fopen** and **fclose:** to open and close files
- **fwrite** and **fread:** to read and write chunks of data
- **fprintf** and **fscanf:** to read and write formatted blocks of data
- ~~**fgetc** and **fputc:** to read and write individual bytes(char)~~

fopen() and fclose()

```
FILE *fopen (const char *filename, const char *mode)
```

mode common options: (there are others)

"r" : Opens a file in read mode and sets pointer to the first character in the file. It returns null if file does not exist.
"w" Opens a file in write mode. It returns null if file could not be opened. If file exists, all data in existing file is lost.
"a" Opens a file in append mode, i.e. sets pointer to last character in file. It returns null if file couldn't be opened.

```
fclose (FILE *fPtr)
```

Like malloc() and free(), fopen() and fclose() should always be paired.

Hello World using a File

file1.c

```
#include <stdio.h>

int main(int argc, char **argv) {
  FILE *filePtr = fopen("file1.out","w");
  fprintf(filePtr, "Hello World\n");
  fclose(filePtr);
}
```

You have seen printf() before,
Only difference with fprintf()
Is that first argument which is a FILE *

If you open a file, be sure to close it!

More formatted output

file2.c

```
#include <stdlib.h>
#include <time.h>
int main(int argc, char **argv) {
  if (argc != 4) {
    fprintf(stdout, "ERRORusage appName n max filename \n");
    return -1;
  }
  int n = atoi(argv[1]);
  float maxVal = atof(argv[2]);
  FILE *filePtr = fopen(argv[3],"w");

  for (int i=0; i<n; i++) {
    float float1 = ((float)rand()/(float)RAND_MAX) * maxVal;
    float float2 = ((float)rand()/(float)RAND_MAX) * maxVal;
    fprintf(filePtr,"%d, %f, %f\n", i, float1, float2);
  }
  fclose(filePtr);
}
```

int fprintf(FILE *fp, const char *format, ...)

format is the C string that contains the text to be written to the file. It can optionally contain embedded format tags that are replaced by the values specified in subsequent additional arguments and formatted as requested. Format tags prototype is **%**

```
c >gcc file2.c -o file2
c >./file2 5 1 file2.out
c >cat file2.out
0, 0.153779, 0.560532
1, 0.865013, 0.276724
2, 0.895919, 0.704462
3, 0.886472, 0.929641
4, 0.469290, 0.350208
c >
```

Formatted Input

int fscanf(FILE *fp, const char *format, ...)

file3.c

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char **argv) {
  FILE *filePtr = fopen(argv[1],"r");
  int i = 0;     float float1, float2;
  int maxVectorSize = 100;
  double *vector1 = (double *)malloc(maxVectorSize*sizeof(double));
  double *vector2 = (double *)malloc(maxVectorSize*sizeof(double));
  int vectorSize = 0;
  while (fscanf(filePtr,"%d, %f, %f\n", &i, &float1, &float2) != EOF) {
    vector1[vectorSize] = float1;
    vector2[vectorSize] = float2;
    printf("%d, %f, %f\n",i, vector2[i], vector1[i]);
    vectorSize++;
    if (vectorSize == maxVectorSize) {
      // some code needed here .. programming exercise
    }
  }
  fclose(filePtr);
}
```

```
c >gcc file2.c -o file2
c >./file2 4 1 file2.out
c >cat file2.out
0, 0.153779, 0.560532
1, 0.865013, 0.276724
2, 0.895919, 0.704462
3, 0.886472, 0.929641
c >
c >gcc file3.c -o file3
c >./file3 file2.out
0, 0.560532, 0.153779
1, 0.276724, 0.865013
2, 0.704462, 0.895919
3, 0.929641, 0.886472
c >
```

BUT

int fscanf(FILE *fp, const char *format, ...)

file3.c

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char **argv) {
  FILE *filePtr = fopen(argv[1],"r");
  int i = 0;     float float1, float2;
  int maxVectorSize = 100;
  double *vector1 = (double *)malloc(maxVectorSize*sizeof(double));
  double *vector2 = (double *)malloc(maxVectorSize*sizeof(double));
  int vectorSize = 0;
  while (fscanf(filePtr,"%d, %f, %f\n", &i, &float1, &float2) != EOF) {
    vector1[vectorSize] = float1;
    vector2[vectorSize] = float2;
    printf("%d, %f, %f\n",i, vector2[i], vector1[i]);
    vectorSize++;
    if (vectorSize == maxVectorSize) {
      // some code needed here .. programming exercise
    }
  }
  fclose(filePtr);
}
```

```
c >./file2 1000 1 fileBIG.out
c >./file3 fileBIG.out
0, 0.560532, 0.153779
1, 0.276724, 0.865013
2, 0.704462, 0.895919
3, 0.929641, 0.886472
4, 0.350208, 0.469290
5, 0.096535, 0.941637
6, 0.346164, 0.457211
```

```
161, 0.533222, 0.600734
162, 0.073887, 0.854827
163, 0.808359, 0.811912
164, 0.884276, 0.084779
165, 0.301760, 0.022628
Segmentation fault: 11
```

Writing to Binary or ASCII file

```
size_t fwrite( const void * ptr, size_t size, size_t count,
               FILE * stream );
```

file4.c

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main(int argc, char **argv) {
  int n =  atoi(argv[1]);
  float maxVal = atof(argv[2]);
  float *theVector = (float *)malloc(n * sizeof(float));
  FILE *fileBinaryPtr = fopen("file3.out","wb");
  FILE *fileAsciiPtr = fopen("file3Ascii.out","w");
  for (int i=0; i<n; i++)
    theVector[i]= ((float)rand()/(float)RAND_MAX) * maxVal;

  for (int i=0; i<n; i++) {
    fprintf(fileAsciiPtr,"%f ",theVector[i]);
  }
  fprintf(fileAsciiPtr,"\n");
  fwrite(theVector, sizeof(float), n, fileBinaryPtr);
  fclose(fileBinaryPtr);
  fclose(fileAsciiPtr);
}
```

modes: “w” and “wb”

w = ascii text

wb = binary

Binary File:

- No data loss
- Smaller (half the size)
- BUT cannot read as not an ASCII file

```
c >gcc file4.c -o file4
c >./file4 5 5
c >cat file4Ascii.out
0.768894 2.802661 4.325066 1.383619 4.479593
c >
c >cat file4.out
>?D??^3@?f?@k???X?@c >
c >
c >ls -sal *4*.out
8 -rw-r--r--  1 fmckenna  staff  20 Jan  4 21:01 file4.out
8 -rw-r--r--  1 fmckenna  staff  46 Jan  4 21:01 file4Ascii.out
```

READING from Binary

```
size_t fread ( void * ptr, size_t size, size_t count, FILE * stream );
int main(int argc, char **argv) {
```

file5.c

```
#include <stdlib.h>
#include <time.h>
#include <stdbool.h>.
int main(int argc, char **argv) {
  FILE *fileBinaryPtr = fopen(argv[1],"rb");
  int vectorSize = 0;
  int maxVectorSize = 100;
  float *theVector = (float *)malloc(maxVectorSize*sizeof(float));
  // read multiple times until no more data, enlarging vector each time in maxVectorSize chunk
  int numValues = 0;
  long numRead = 0;
  bool allDone = false;
  while (allDone == false) {
    long numRead = fread(&theVector[vectorSize], sizeof(float), maxVectorSize, fileBinaryPt
    numValues += numRead;
    vectorSize += numRead;
    if (numRead == maxVectorSize) {
      // not done, enlarge for next time
      float *newVector = (float *)malloc((vectorSize + maxVectorSize)*sizeof(double));
      for (int i=0; i< vectorSize; i++)
          newVector[i] = theVector[i];
      free(theVector);
      theVector = newVector;
    } else
      allDone = true;
```

EXERCISE