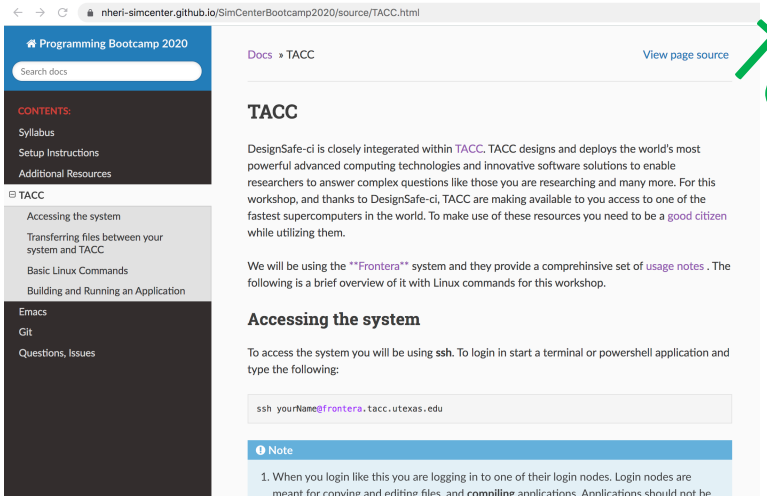
Reference Material

Website: https://nheri-simcenter.github.io/SimCenterBootcamp2020/index.html

TACC (login, linux, compile)

Emacs

Git



TODAY MIGHT NOT BE EASY!

Examples in your REPO: SimCenterBootcamp2020/code/c

CS50 2019 (Harvard)

C Intro: https://www.youtube.com/watch?v=e9Eds2Rc_x8
C Arrays: https://www.youtube.com/watch?v=8PrOp9t0PyQ
C Algorithms: https://www.youtube.com/watch?v=fykrlqbV9wM
C *MEMORY: https*://www.youtube.com/watch?v=cF6YkH-8vFk

SimCenter NHERI
Center for Computational Modeling and Simulation

Programming Bootcamp

C: Pointers Revisited

Frank McKenna
University of California at Berkeley

NSF award: CMMI 1612843

Pointer Variables

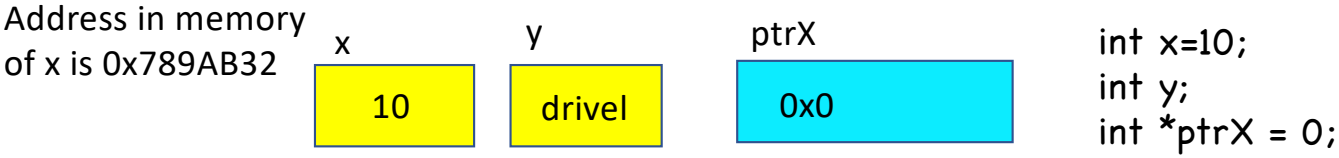
A **Pointer Variable** is a named memory location that can holds an address**.**

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.



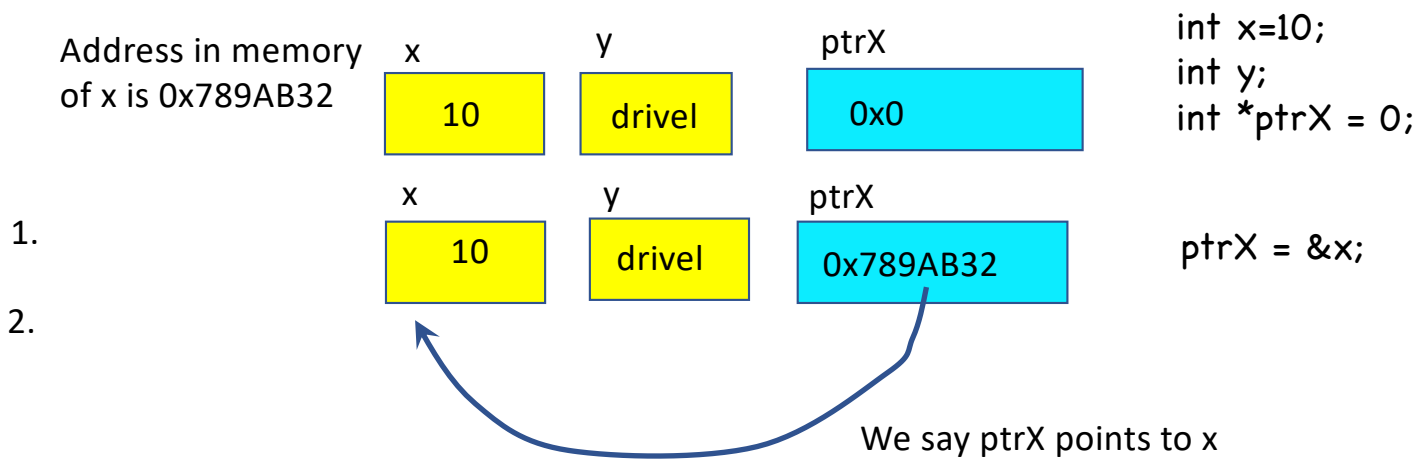
Pointer Variables

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

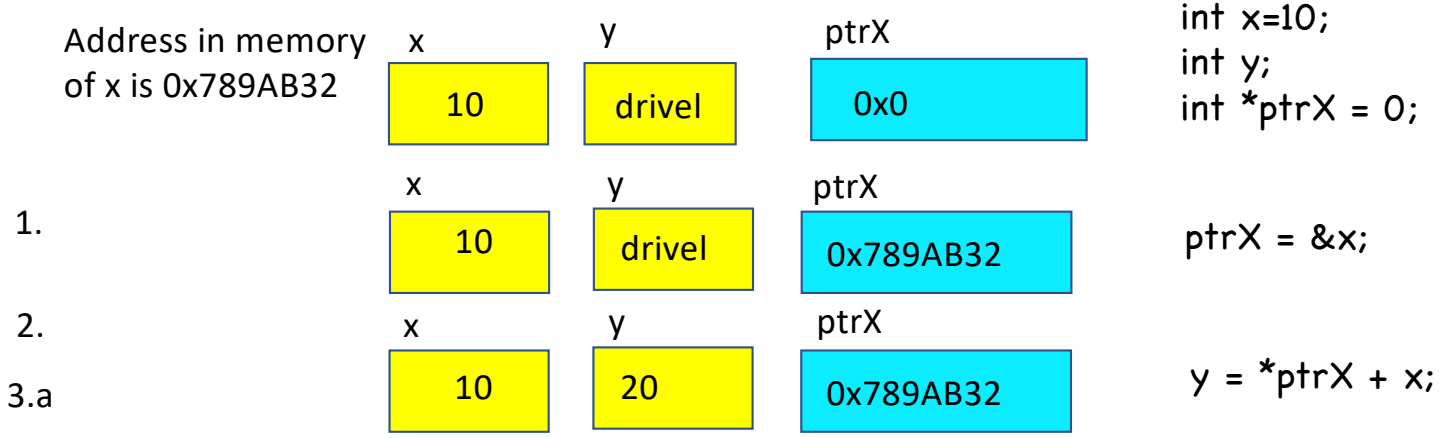
A **Pointer Variable** is a named memory location that can holds an address**.**

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



Pointer Variables

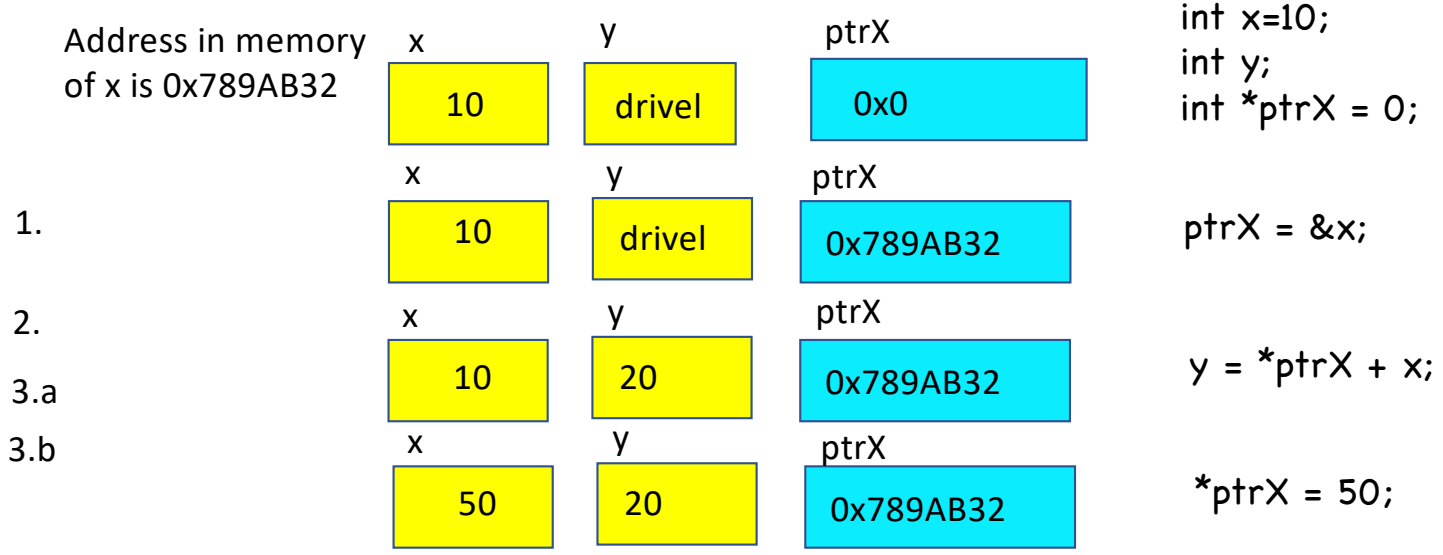
A **Pointer Variable** is a named memory location that can holds an address**.**

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;
}
```

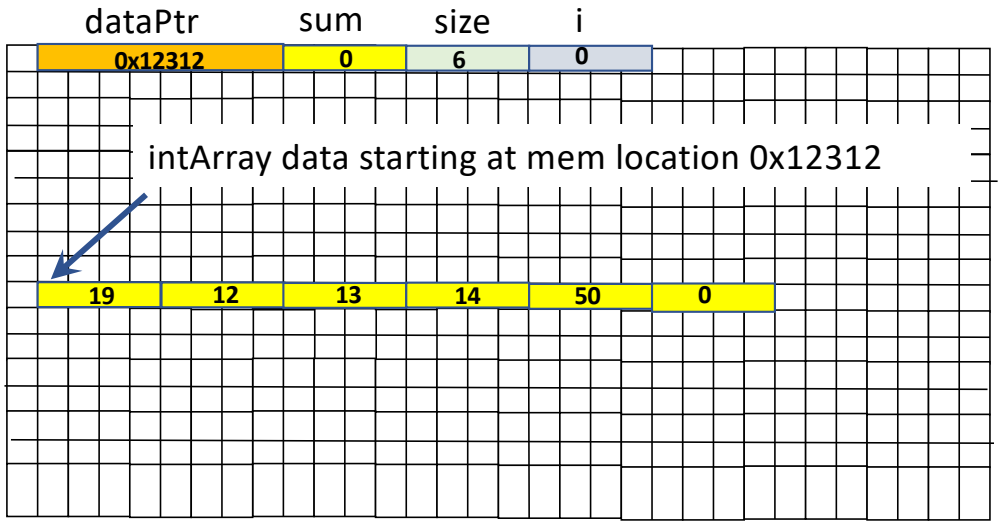
pointer1.c

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 and Memory Allocation

Frank McKenna
University of California at Berkeley

NSF award: CMMI 1612843

Arrays - II

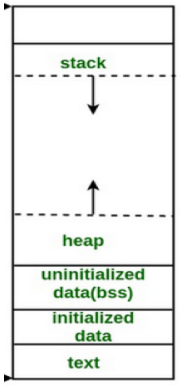
- An array is fixed size sequential collection of elements all of the same type. The are contiguously stored in memory. We access using an index inside a square brackets or by dereferencing pointers.
- to declare: type arrayName [size];

  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, array2, array3);
  }
  // free the array
  free(array1);
  return(0);
```

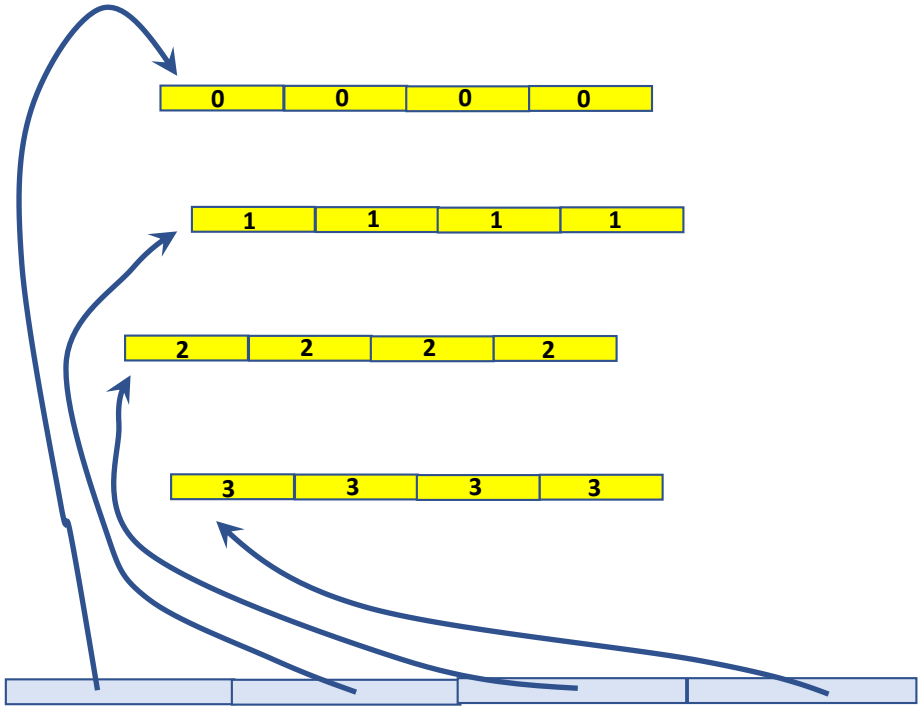
```
c >gcc memory2.c; ./a.out 4
0.0000 0.0000 0.0000 0x7f8dd84059a0 0x7f8dd84059a0 0x7f8dd84059a0
1.0000 1.0000 1.0000 0x7f8dd84059a8 0x7f8dd84059a8 0x7f8dd84059a8
2.0000 2.0000 2.0000 0x7f8dd84059b0 0x7f8dd84059b0 0x7f8dd84059b0
3.0000 3.0000 3.0000 0x7f8dd84059b8 0x7f8dd84059b8 0x7f8dd84059b8
```

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

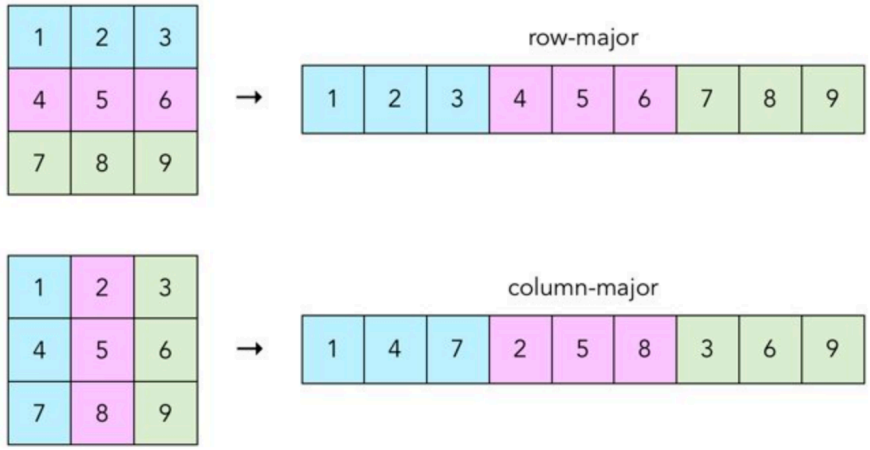


matrix1

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(nu
for (int j=0; j<numCols; j++
   for (int i=0; i<numRows
       matrix2[I + j*numRo
 }
```

```
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);
```

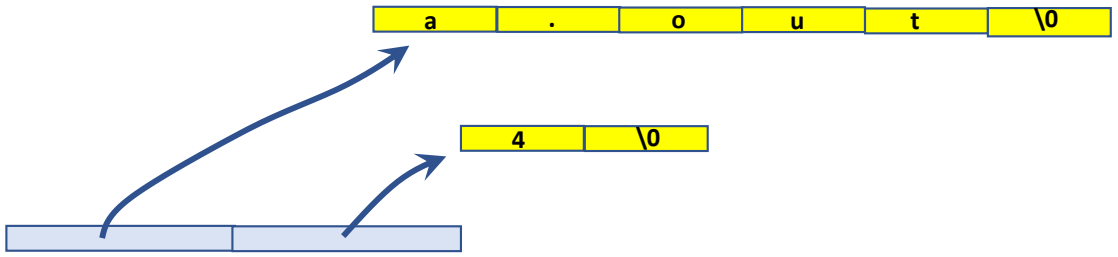
an implementation of strlen:
provided just to show how it will oook for `\0`

```
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;
}
```

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)

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, fileBinaryPtr
    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;
```

SimCenter NHERI
Center for Computational Modeling and Simulation

Programming Bootcamp

C: structs, Data Structures and Abstraction

Frank McKenna
University of California at Berkeley

NSF award: CMMI 1612843

Abstraction

The goal of "**abstracting**" data is to reduce complexity by removing unnecessary information. Think bigger, ignore the minutia.

float
integer
double
string

C Structures

```
struct structName {
    type name;
    ….
};
```

- A Powerful feature that allows us to put together **our own abstractions.**
- **A struct is a composite data type that we define that defines a physically grouped list of variables under one name in a block of memory.**
- We can compound as many different types as we want to form a new type

struct1.c

```
#include <stdio.h>
struct point {
   float x;
   float y;
};
```

Note the semi-colon after the struct definition

```
int main(int argc, char **argv) {
  struct point p1 = {1.0, 50};
  struct point p2;
  p2.x = 100 + p1.x;
  p2.y = 50;

  printf(" Point1: x %10f y%10f\n", p1.x, p1.y);
  printf(" Point2: x %10f y%10f\n", p2.x, p2.y);
  return 0;
}
```

typedef

typedef varType alias;

A way to create new type name. New names are an alias the compiler uses to make programmers life easier.

Something to utilize for making working with structs easier

```
typedef float numType;

int main(int argv, char **argc) {
  numType a = 1.0;
  return 0;
}
```

struct2.c

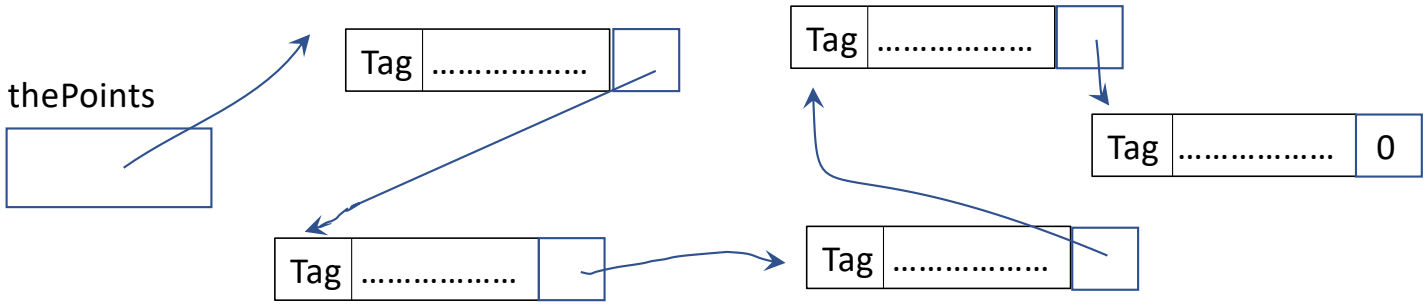
```
#include <stdio.h>

;

typedef struct point {
  float x;
  float y;
} Point;

int main(int argc, char **argv) {
  numType value = 20.;

  Point p1 = {1.0, 50};
  Point p2;
  p2.x = 100 + p1.x;
  p2.y = value;


  printf(" Point1: x %10f y%10f\n", p1.x, p1.y);
  printf(" Point2: x %10f y%10f\n", p2.x, p2.y);
  return;
}
```

Data Structures

"In computer science, a **data structure** is a **data** organization, management, and storage format that enables efficient access and modification to data" (Wikipedia). The C Programming language provides the ability to program many common data structures like *arrays, stacks, queues, linked list, tree, etc*. It is of course flexible. enough to allow you to come up with your own data structures.

Which data structures to use to store the objects depends on how the user intends to access the data

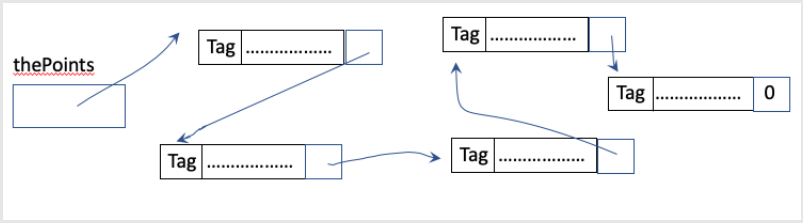
Example Linked List of Points



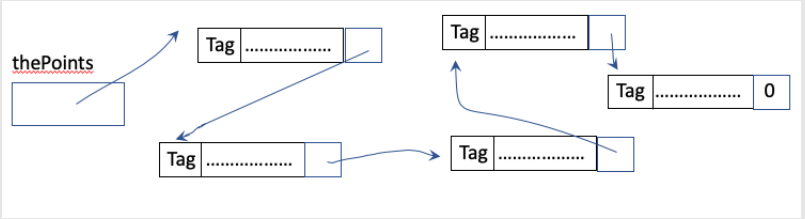
thePoints – pointer to a Point *, each Point has a pointer to another node

linkedList.c

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
typedef struct point {
  int     tag;
  float   x;
  float   y;
  struct point *next;
} Point;

int main(int argc, char **argv) {
  // pointer to hold the link to all points
  Point *thePoints = 0;
  // read in points
  int tag;
  float x,y;
  FILE *inputFile = fopen(argv[1],"r");
  while (fscanf(inputFile, "%d, %f, %f\n", &tag, &x, &y) != EOF) {
    Point *nextPoint = (Point *)malloc(sizeof(Point));
    nextPoint->tag = tag; nextPoint->x = x; nextPoint->y = y;
    nextPoint->next = thePoints;
    thePoints = nextPoint;
  }
  // do something with linked list
```



```
 // doing something with linked list
bool done = false;
 while (done == false) {
   int tagToFind;
   printf("Enter tag to find: ");
   scanf("%d",&tagToFind);
   int tagToFind;
   Point *currentPoint = thePoints;
    while (currentPoint != 0 && currentPoint->tag != tagToFind) {
      currentPoint = currentPoint->next;
    }

   if (currentPoint != 0) {
     printf("FOUND Point with tag %d at locaction: %f %f\n", tag, currentPoint->x, currentPoint->y);
    } else {
     printf("Could not find point with tag %d\nExiting\n", tagToFind);
     done = true;
   }
 }
 fclose(inputFile);
 return 0;
}
```

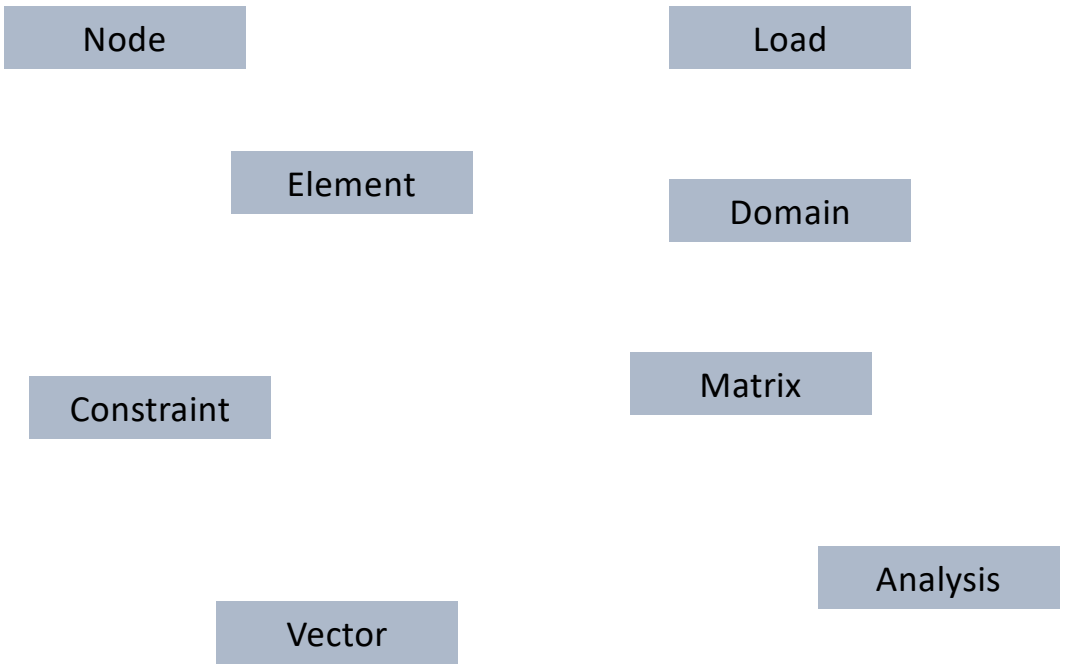


Structs, Pointers and Data Structures

allowed us to think of searching in terms of looking for a points in a file

Why not of course think in terms of other abstractions!

What about Abstractions for a Finite Element Application?



What Does A Node Have?

- Node number or tag
- Coordinates
- Displacements?
- Velocities and Accelerations??

2d or 3d?
How many dof?
Do We Store Velocities and Accel.

Depends on what the program needs of it

Say Requirement is 2dimensional, need to store the displacements (3dof)?

```
struct node {
   int tag;
   double xCrd;
   double yCrd;
   double displX;
   double dispY;
   double rotZ;
};
```

```
struct node {
   int tag;
   double coord[2];
   double displ[3];
};
```

I would lean towards the latter; easier to extend to 3d w/o changing 2d code, easy to write for loops .. But is there a cost associated with accesing arrays instead of variable directly .. Maybe compile some code and time it for intended system

```
#includede <stdio.h>
struct node {
    int tag;
    double coord[2];
    double disp[3];
};
void nodePrint(struct node *);

int main(int argc, const char **argv) {
    struct node n1;  // create variable named n1 of type node
    struct node n2;
    n1.tag = 1;        // to set n1's tag to 1 .. Notice the DOT notation
    n1.coord[0] = 0.0;
    n1.coord[0] = 1.0;
    n2.tag = 2;
    n2.coord[0] = n1.coord[0];
    n2.coord[0] = 2.0;
    nodePrint(&n1);
    nodePrint(&n2);
}
void nodePrint(struct node *theNode){
    printf("Node : %d ", theNode->tag);    // because the object is a pointer use -> ARROW to access
    printf("Crds: %f %f ", theNode->coord[0], theNode->coord[1]);
    printf("Disp: %f %f %f \n", theNode->disp[0], theNode->disp[1], theNode->disp[2]);
}
```

```
C >gcc node2.c; ./a.out
Node : 1 Crds: 0.000000 1.000000 Disp: 0.000000 0.000000 0.000000
Node : 2 Crds: 0.000000 2.000000 Disp: 0.000000 0.000000 0.000000
C >
```

```c
#includede <stdio.h>
typedef struct node {.
  int tag;
  double coord[2];
  double disp[3];
} Node;
void nodePrint(Node *);
void nodeSetup(Node *, int tag, double crd1, double crd2);
int main(int argc, const char **argv) {
    Node n1;
    Node n2;
    nodeSetup(&n1, 1, 0., 1.);
    nodeSetup(&n2, 2, 0., 2.);
    nodePrint(&n1);
    nodePrint(&n2);
}
void nodePrint(Node *theNode){
    printf("Node : %d ", theNode->tag);
    printf("Crds: %f %f ", theNode->coord[0], theNode->coord[1]);
    printf("Disp: %f %f %f \n", theNode->disp[0], theNode->disp[1], theNode->disp[2]);
}
void nodeSetup(Node *theNode, int tag, double crd1, double crd2) {
    theNode->tag = tag;
    theNode->coord[0] = crd1;
    theNode->coord[1] = crd2;
```

Using typedef to give you to give the new struct a name;
Instead of struct node now use Node

Also created a function to quickly initialize a node

```
C >gcc node2.c; ./a.out
Node : 1 Crds: 0.000000 1.000000 Disp: 0.000000 0.000000 0.000000
Node : 2 Crds: 0.000000 2.000000 Disp: 0.000000 0.000000 0.000000
C >
```

Clean This up for a large FEM Project

Files for each date type and their functions: node.h, node.c, domain.h, domain.c, …

fem/main1.c

```
#include "node.h"
#include "domain1.h"
int main(int argc, const char **argv) {
  Domain theDomain;
  theDomain.theNodes=0; theDomain.NumNodes=0; theDomain.maxNumNodes=0;
  domainAddNode(&theDomain, 1, 0.0, 0.0);
  domainAddNode(&theDomain, 2, 0.0, 2.0);
  domainAddNode(&theDomain, 3, 1.0, 1.0);
  domainPrint(&theDomain);
  // get and print singular node
  printf("\nsingular node:\n");
  Node *theNode = domainGetNode(&theDomain, 2);
  nodePrint(theNode);
}
```

Domain is some CONTAINER that holds the nodes and gives access to them to say the elements and analysis

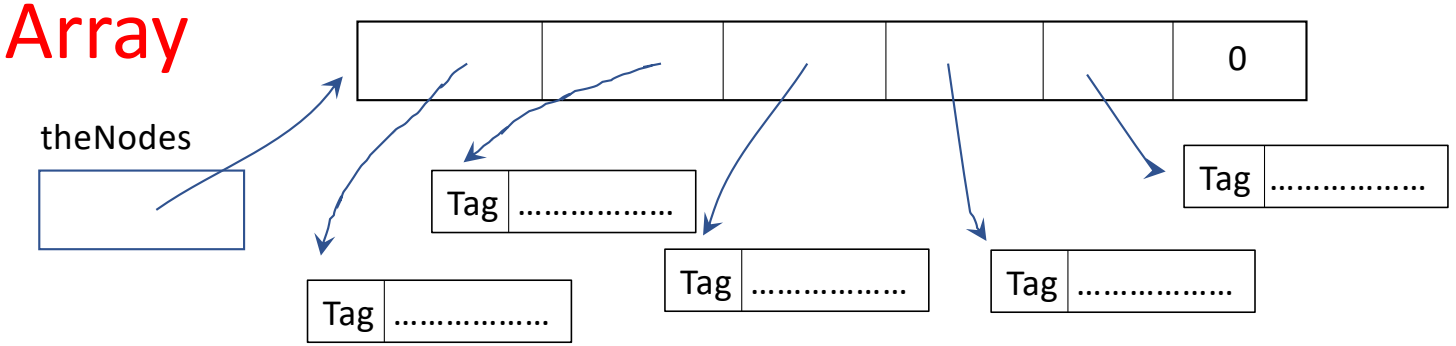
Domain

- Container to store nodes, elements, loads, constraints
- How do we store them
- In CS a number of common storage schemes:
  1. Array
  2. Linked List
  3. Double Linked List
  4. Tree
  5. Hybrid

Which to Use – Depends on Access Patterns, Memory, …

but all involve Pointers (2 examples)

Array

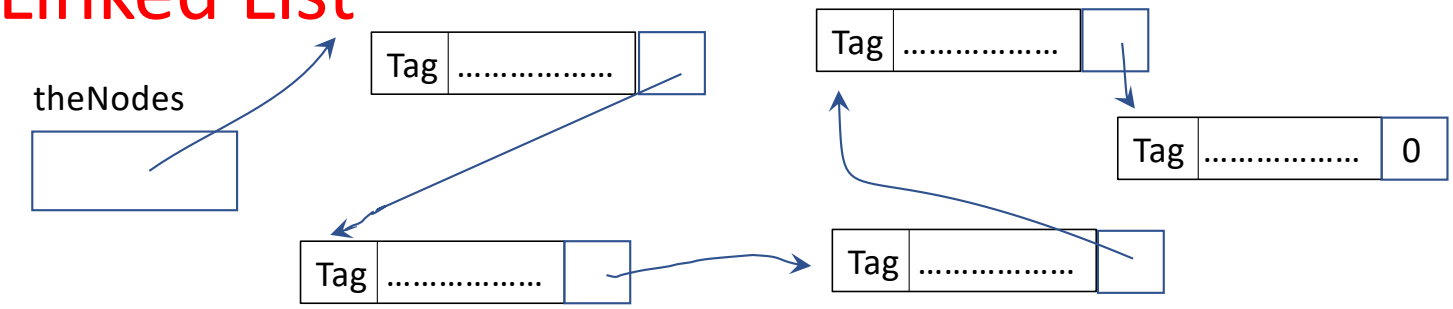


theNodes – pointer to an array of Node *, i.e. each component of array points to a Node.
Want a variable sized array (small and large problems), what happens if too many nodes Added – malloc an even bigger array, copy existing pointers (just address not objects)
=> need Node**, variable to hold current size, variable to hold max size

c/fem/domain1.h

```
#include "node.h"
typedef struct struct_domain {
  Node **theNodes;
  int numNodes;
  int maxNumNodes;
} Domain;

void domainPrint(Domain *theDomain);
void domainAddNode(Domain *theDomain, int tag, double crd1, double crd2);
void domainPrintNodes(Domain *theDomain);
Node *domainGetNode(Domain *, int nodeTag);
```

Linked List



theNodes – pointer to a Node *, each Node has a pointer to another node

c/fem/domain2.h

```
#include "node.h"
typedef struct struct_domain {
  Node *theNodes;
} Domain;

void domainPrint(Domain *theDomain);
void domainAddNode(Domain *theDomain, int tag, double crd1, double crd2);
void domainPrintNodes(Domain *theDomain);
Node *domainGetNode(Domain *, int nodeTag);
```

fem/domain1.c

```
Node *domainGetNode(Domain *theDomain, int nodeTag) {
 int numNodes = theDomain->numNodes;
 for (int i=0; i<numNodes; i++) {
  Node *theCurrentNode = theDomain->theNodes[i];
  if (theCurrentNode->tag == nodeTag) {
   return theCurrentNode;
  }
 }
 return NULL;
}
```

Array Search

fem/domain2.c

```
Node *domainGetNode(Domain *theDomain, int nodeTag) {
 Node *theCurrentNode = theDomain->theNodes;
 while (theCurrentNode != NULL) {
  if (theCurrentNode->tag == nodeTag) {
   return theCurrentNode;
  } else {
   theCurrentNode = theCurrentNode->next;
  }
 }
 return NULL;
}
```

List Search

c/fem/node.h

```
#ifndef _NODE
#define _NODE

#include <stdio.h>

typedef struct node {
  int tag;
  double coord[2];
  double disp[3];
  struct node *next;
} Node;

void nodePrint(Node *);
void nodeSetup(Node *, int tag, double crd1, double crd2);

#endif
```

What About Elements
Data & Function (tangent, resisting force)

We want a model that can handle many different element types and user defined types

Abacus element interface:

```
      SUBROUTINE UEL(RHS,AMATRX,SVARS,ENERGY,NDOFEL,NRHS,NSVARS,
     1 PROPS,NPROPS,COORDS,MCRD,NNODE,U,DU,V,A,JTYPE,TIME,DTIME,
     2 KSTEP,KINC,JELEM,PARAMS,NDLOAD,JDLTYP,ADLMAG,PREDEF,NPREDF,
     3 LFLAGS,MLVARX,DDLMAG,MDLOAD,PNEWDT,JPROPS,NJPROP,PERIOD)
```

For each element we have a function, for args to be same we need to pass element parameters and element state information (assuming nonlinear problem) in function call. We also need to manage for the element the state information (trial steps to converged step) in Newton iteration

Element?

```
#ifndef _ELEMENT
#define _ELEMENT

#include ”node.h”
#include <stdio.h>
typedef (int)(*elementStateFunc)(Domain *theDomain, double *k, double *P);

typedef struct element {
  int tag;
  int nProps, nHistory;
  int *nodeTags;
  double *props;
  double *history;
  elementStateFunc eleState;
  struct element *next;
} Element;

void elementPrint(Element *);
void elementComputeState(Element *theEle, double *k, double *R);

#endif
```

Creating Types is easy

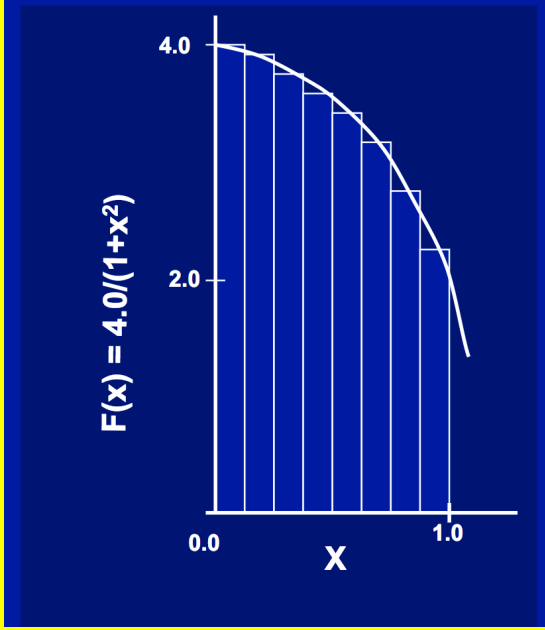
- Creating smart types where we need to keep data and functions that operate on the data for different possible types becomes tricky.

Practice Exercises (1 hour): as many as you can

1. Write a program that when running prompts the user for two floating point numbers and returns their product.
   - i.e. ./a.out would prompt for 2 numbers a and b will output a * b = something
2. Write a program that takes a number of integer values from argc, stores them in an array, computes the sum of the array and outputs some nice message. Try using recursion to compute the sum. (hint start with recursion1.c and google function atoi(), copy from memory1.c)
   - i.e. ./a.out 3 4 5 6 will output 3 + *4 + 5 + 6 = 18*
3. *Taking the previous program. Modify it to output the number of unique numbers in the output.*
   - *i.e. ./a.out 3 1 2 1 will output 1*3 + 2*1 + 1* 2 = 7*
4. *Write a program that takes a number of input values and sorts them in ascending order.*
   - *I.e. ./a.out 2 7 4 5 9 will output* 2 4 5 7 9

Exercise: Compute PI

Numerical Integration



Mathematically, we know that:

$$\int_0^1 \frac{4.0}{(1+x^2)}\,dx = \pi$$

We can approximate the integral as a sum of rectangles:

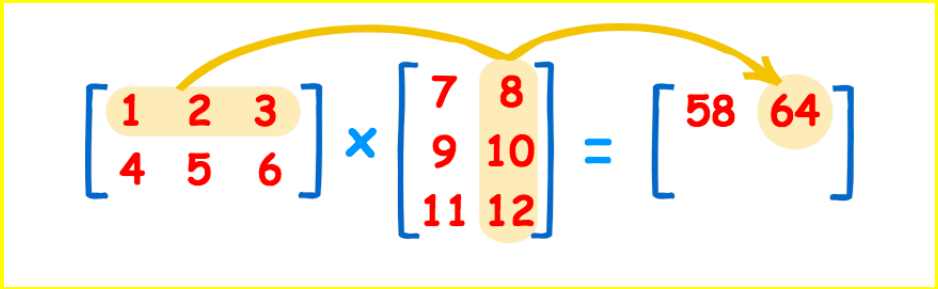
$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

Where each rectangle has width Δx and height $F(x_i)$ at the middle of interval i.

```
#include <stdio>
static int long numSteps = 100000;
int main() {
  double pi = 0; double time=0;
  // your code
  for (int i=0; i<numSteps; i++) {
    // your code
  }
   // your code
  printf("PI = %f, duration: %f ms\n",pi, time);
  return 0;
}
```

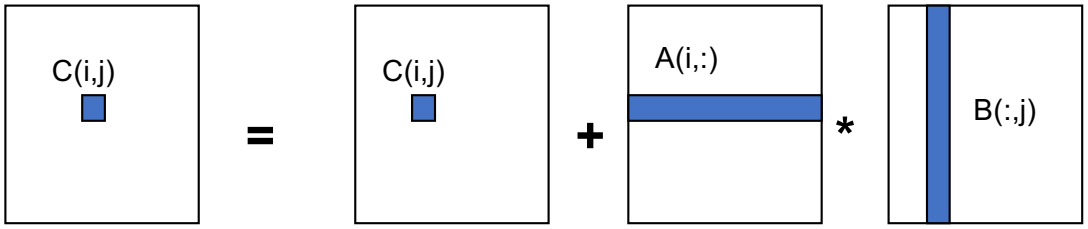
Source: UC Berkeley, Tim Mattson (Intel Corp), CS267 & elsewhere

Exercise: Matrix-Matrix Multiply



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

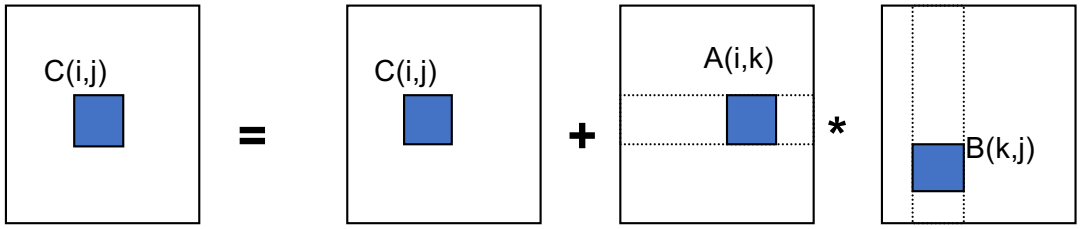
Consider A,B,C to be N-by-N matrices of b-by-b subblocks where b=n / N is block size

```
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