Sessions Outline

- 1:30-2.00 File Example(20min)
- 2:00-2:15 C Strucres (15 min)
- 2:15-3:00 Exercise stressTransform with structs
- BREAK to 3:15
- 3:15-3:45 Data Structures
- 3:45-4:30 Exercise: stressTransform with linked list

- Exercises: advanced options available.

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

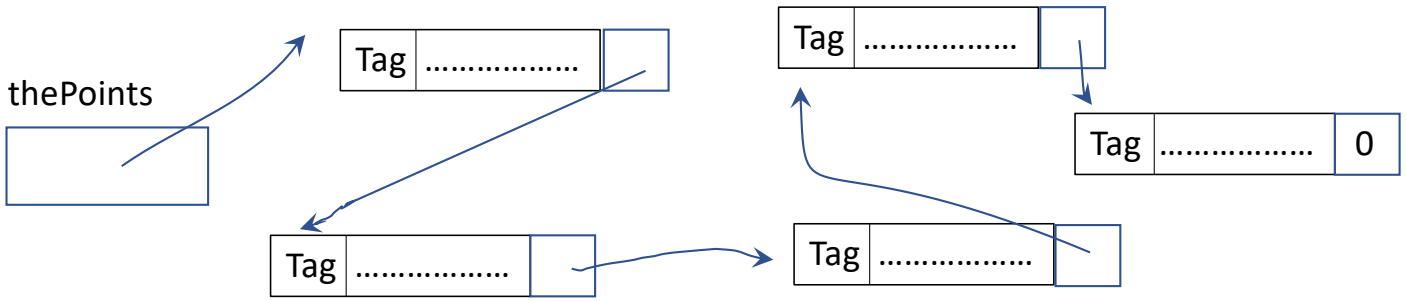
"In computer science, a **data structure** is a **data** organization, management, and storage format that enables efficient access and modification to data" (Wikipedia).

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

EXERCISE

Example Linked List of Points

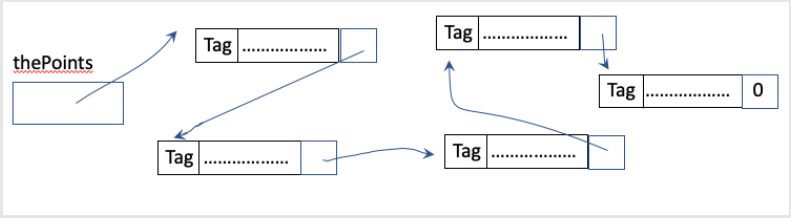


thePoints – pointer to a Point *, each Point has a pointer to another node

linkedList.c

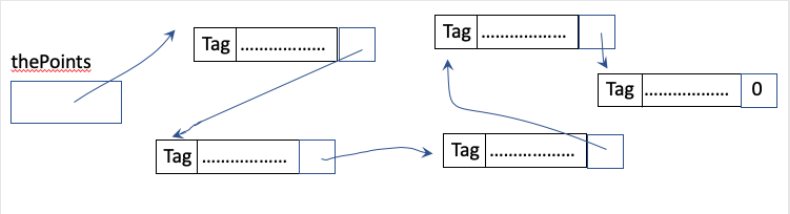
```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
typedef struct point {
  int      tag;
  float    x;
  float    y;
  struct point *next;
} Point;

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;
}
```

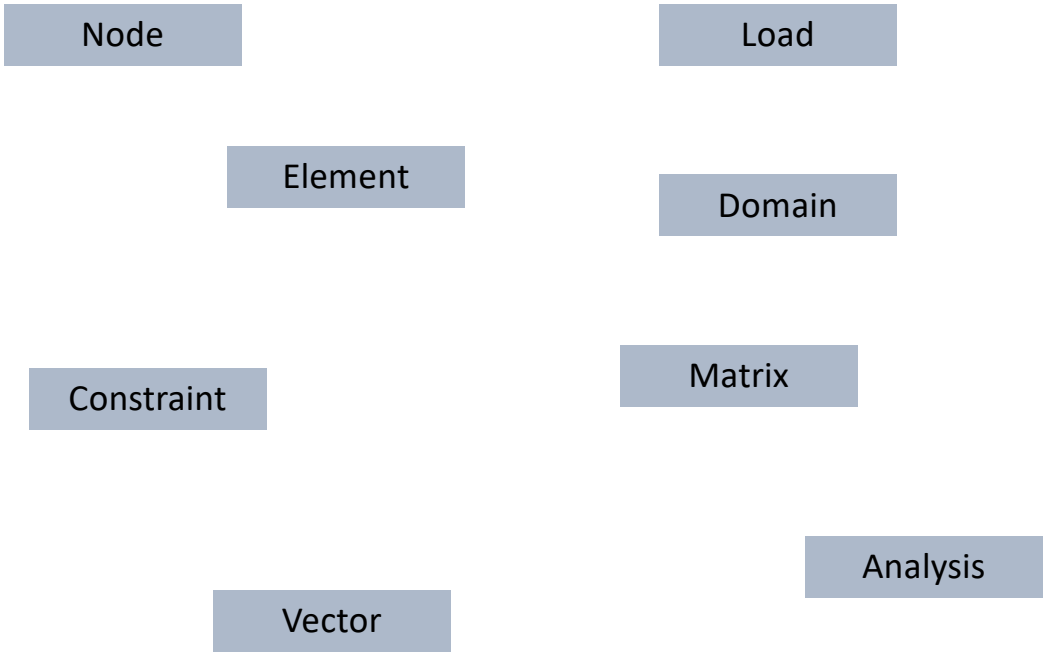


Structs, Pointers and Data Structures

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

```c
#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

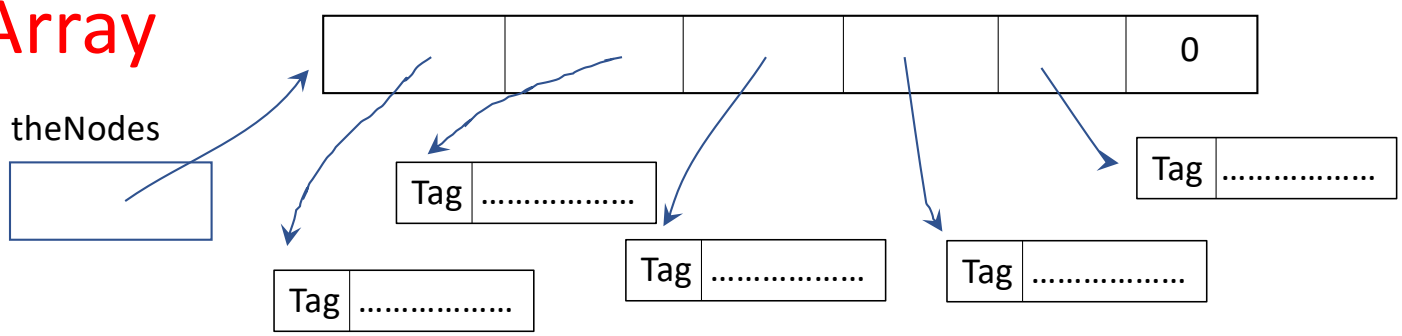
Domain

- 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



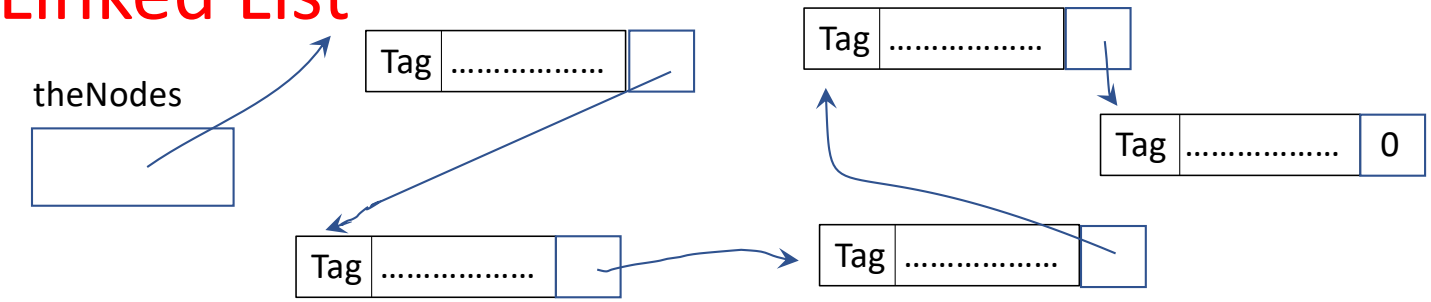
theNodes – pointer to an array of Node *, i.e. each component of array points to a Node.
Want a variable sized array (small and large problems), what happens if too many nodes Added – malloc an even bigger array, copy existing pointers (just address not objects)
=> need Node**, variable to hold current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

EXERCISE