SimCenter NHERI
Center for Computational Modeling and Simulation

2020 Programming Bootcamp

Abstractions & C Structures

Frank McKenna

University of California at Berkeley

NSF award: CMMI 1612843

Outline

- **Abstraction**
- **C Programming Language Contd.**
  - Structures
  - Containers

Abstraction

"*The process of removing physical, spatial, or temporal details or attributes in the study of objects or systems in order to more closely attend to other details of interest*" [source: wikippedia] .

Abstraction

- **Abstraction:** Focusing on the external properties of an entity to the extent of almost ignoring the details of the entity's internal composition

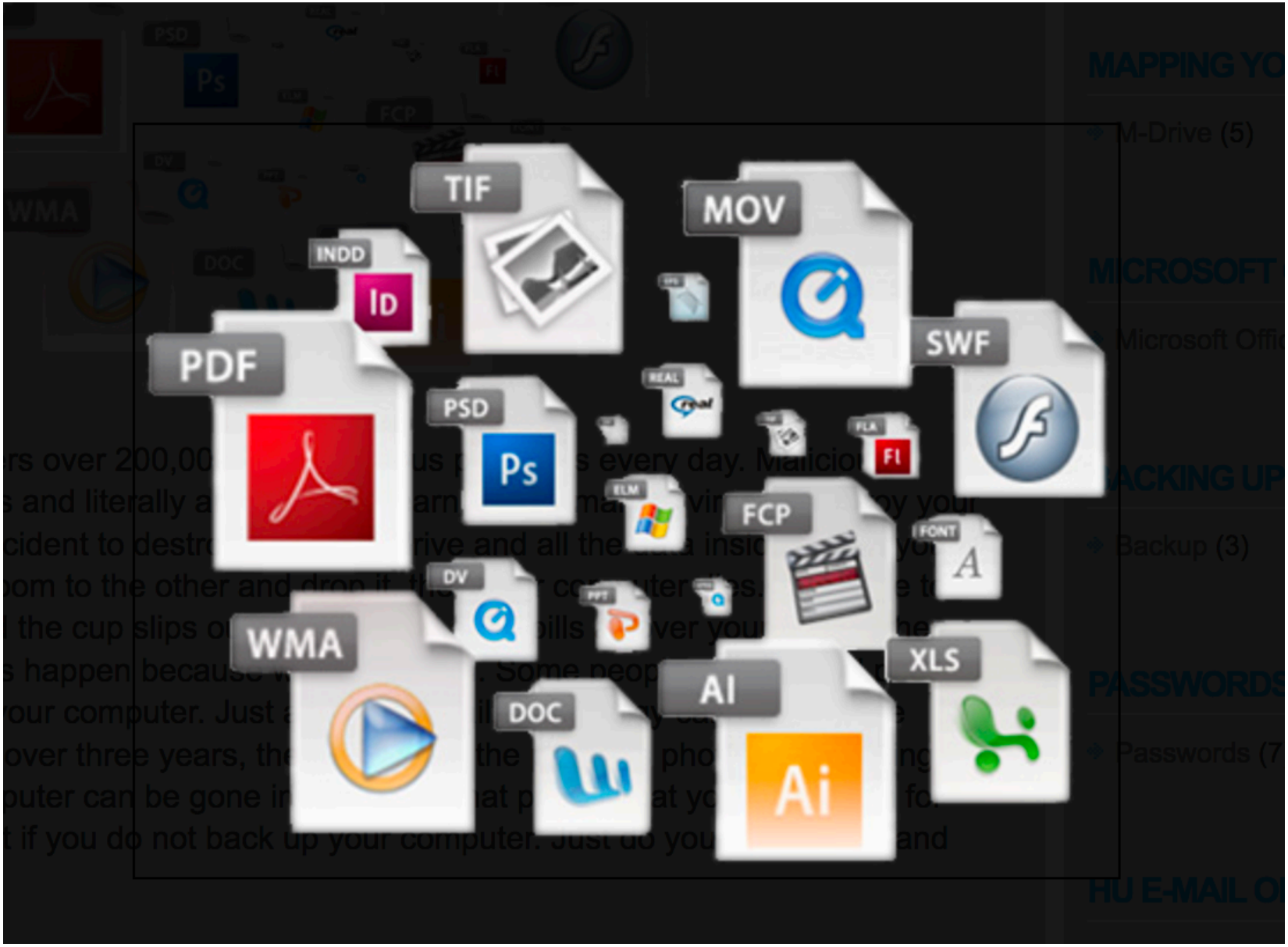
- **Abstraction simplifies many aspects of computing and makes it possible to build complex systems.**

- **Computing Languages Provide Programmers Ability to create abstractions. Higher Level Languages provide more abstraction capabilities (albeit at expense of performance)**

Digital Computer

*"**Digital computer**, any of a class of devices capable of solving problems by processing information in discrete form. It operates on data, including magnitudes, letters, and symbols, that are expressed in binary code —i.e., using only the two digits 0 and 1. By counting, comparing, and manipulating these digits or their combinations according to a set of instructions held in its memory, a digital computer can perform such tasks as to control industrial processes and regulate the operations of machines; analyze and organize vast amounts of business data; and simulate the behaviour of dynamic systems (e.g., global weather patterns and chemical reactions) in scientific research."* (source: enclyopedia Britannica)

Abstractions is What Makes Computers Usable



We Work in Decimal

0,1,2,3,4,5,6,7,8,9

Computers in Binary

0,1

We Combine Numbers

100	10	1
4	5	6

$$4*100 + 5*10 + 6$$

With 3 decimal digits we can represent any number 0 through 999

In Binary We Could Combine Numbers

4 2 1

101

1*4 + 0*2 + 1*1

With 3 binary digits we can represent any number 0 through 7

With 3 digits we have the following possibilities

000
001
010
011
100
101
110
111

2^3 possibilities

And these 8 could represent many things

000	0	A	Saudi Arabia
001	1	B	Iraq
010	2	C	Kuwait
011	3	D	Bahrain
100	4	E	Qatar
101	5	F	Oman
110	6	G	UAE
111	7	H	

Computer naturally groups bits into bytes

1 Byte = 8 bits

2^8 = 256 possibilities

We Have seen some standard abstractions

1. Integer

char	1 byte	-128 to 127 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127
int	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
long	4 bytes	-2,147,483,648 to 2,147,483,647
unsigned long	4 bytes	0 to 4,294,967,295

2. Floating Point Types

float	4 byte	1.2E-38 to 3.4E+38	6 decimal places
double	8 byte	2.3E-308 to 1.7E+308	15 decimal places
long double	10 byte	3.4E-4932 to 1.1E+4932	19 decimal places

3. Enumerated Types

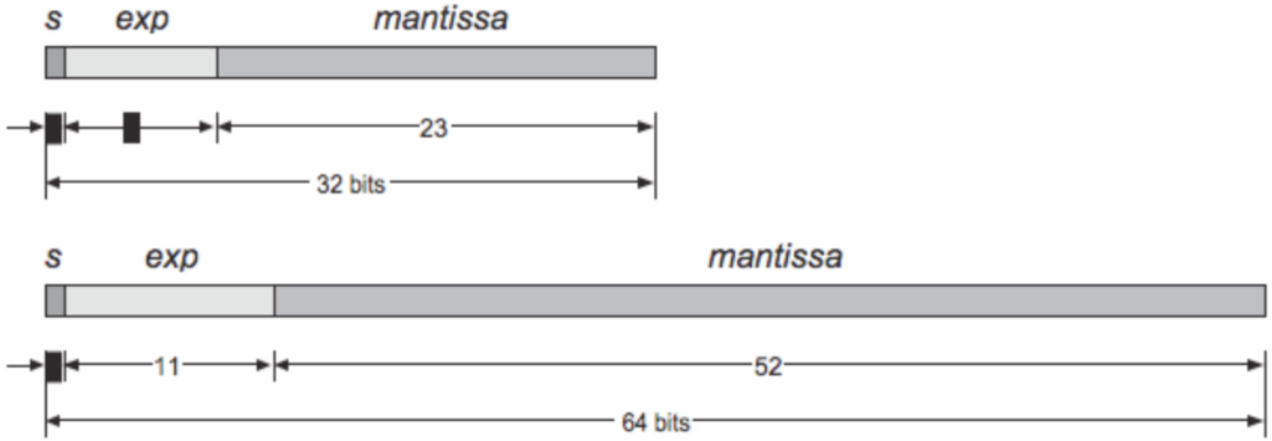
4. **void** Type

5. Derived Types

Structures,
Unions,
Arrays

Float and Double Point Numbers - IEEE 754 standard

Single Precision



Double Precision

float	4 byte	1.2E-38 to 3.4E+38	6 decimal places
double	8 byte	2.3E-308 to 1.7E+308	15 decimal places
long double	10 byte	3.4E-4932 to 1.1E+4932	19 decimal places

C Character Set

ASCII Value	Character	Meaning
0	NULL	null
1	SOH	Start of header
2	STX	start of text
3	ETX	end of text
4	EOT	end of transaction
5	ENQ	enquiry
6	ACK	acknowledgement
7	BEL	bell
8	BS	back Space
9	HT	Horizontal Tab
10	LF	Line Feed
11	VT	Vertical Tab
12	FF	Form Feed
13	CR	Carriage Return
14	SO	Shift Out
15	SI	Shift In
16	DLE	Data Link Escape
17	DC1	Device Control 1
18	DC2	Device Control 2
19	DC3	Device Control 3
20	DC4	Device Control 4
21	NAK	Negative Acknowledgement
22	SYN	Synchronous Idle
23	ETB	End of Trans Block
24	CAN	Cancel
25	EM	End of Mediium
26	SUB	Sunstitute
27	ESC	Escape
28	FS	File Separator
29	GS	Group Separator
30	RS	Record Separator
31	US	Unit Separator

ASCII Value	Character
32	Space
33	!
34	"
35	#
36	$
37	%
38	&
39	
40	(
41	)
42	*
43	+
44	,
45	-
46	.
47	/
48	0
49	1
50	2
51	3
52	4
53	5
54	6
55	7
56	8
57	9
58	:
59	;
60	<
61	=
62	>
63	?

ASCII Value	Character
64	@
65	A
66	B
67	C
68	D
69	E
70	F
71	G
72	H
73	I
74	J
75	K
76	L
77	M
78	N
79	O
80	P
81	Q
82	R
83	S
84	T
85	U
86	V
87	W
88	X
89	Y
90	Z
91	[
92	\
93	]
94	^
95	_

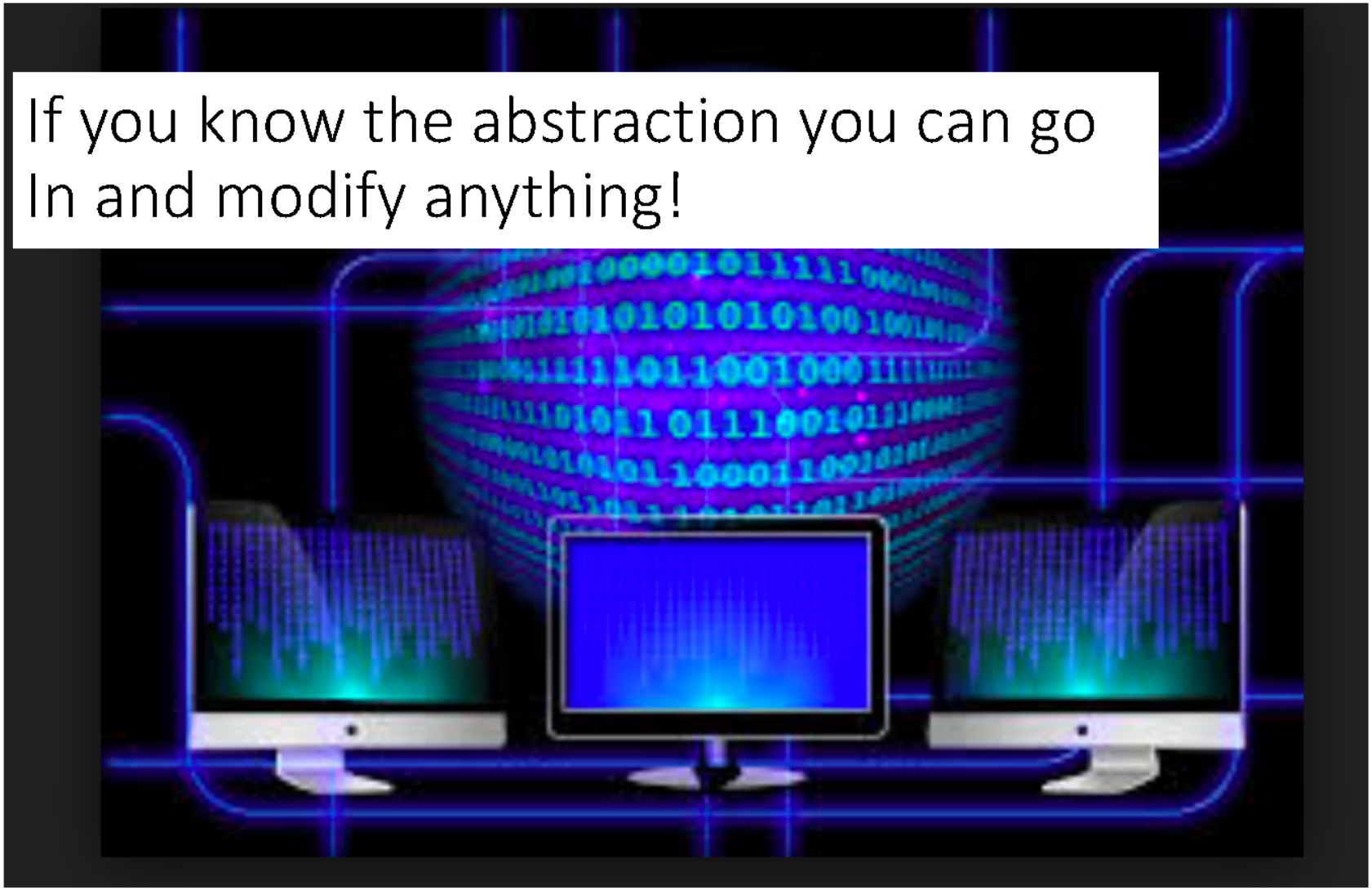
ASCII Value	Character
96	`
97	a
98	b
99	c
100	d
101	e
102	f
103	g
104	h
105	i
106	j
107	k
108	l
109	m
110	n
111	o
112	p
113	q
114	r
115	s
116	t
117	u
118	v
119	w
120	x
121	y
122	z
123	{
124	|
125	}
126	~
127	DEL

And Here is a Program To Print It

charset.c

```
#include <stdio.h>
int main(int argv, const char **argc) {
  for (int i=-127; i<127; i++)
    printf("%d -> %c \n",i,i);
}
```

If you know the abstraction you can go In and modify anything!



Outline

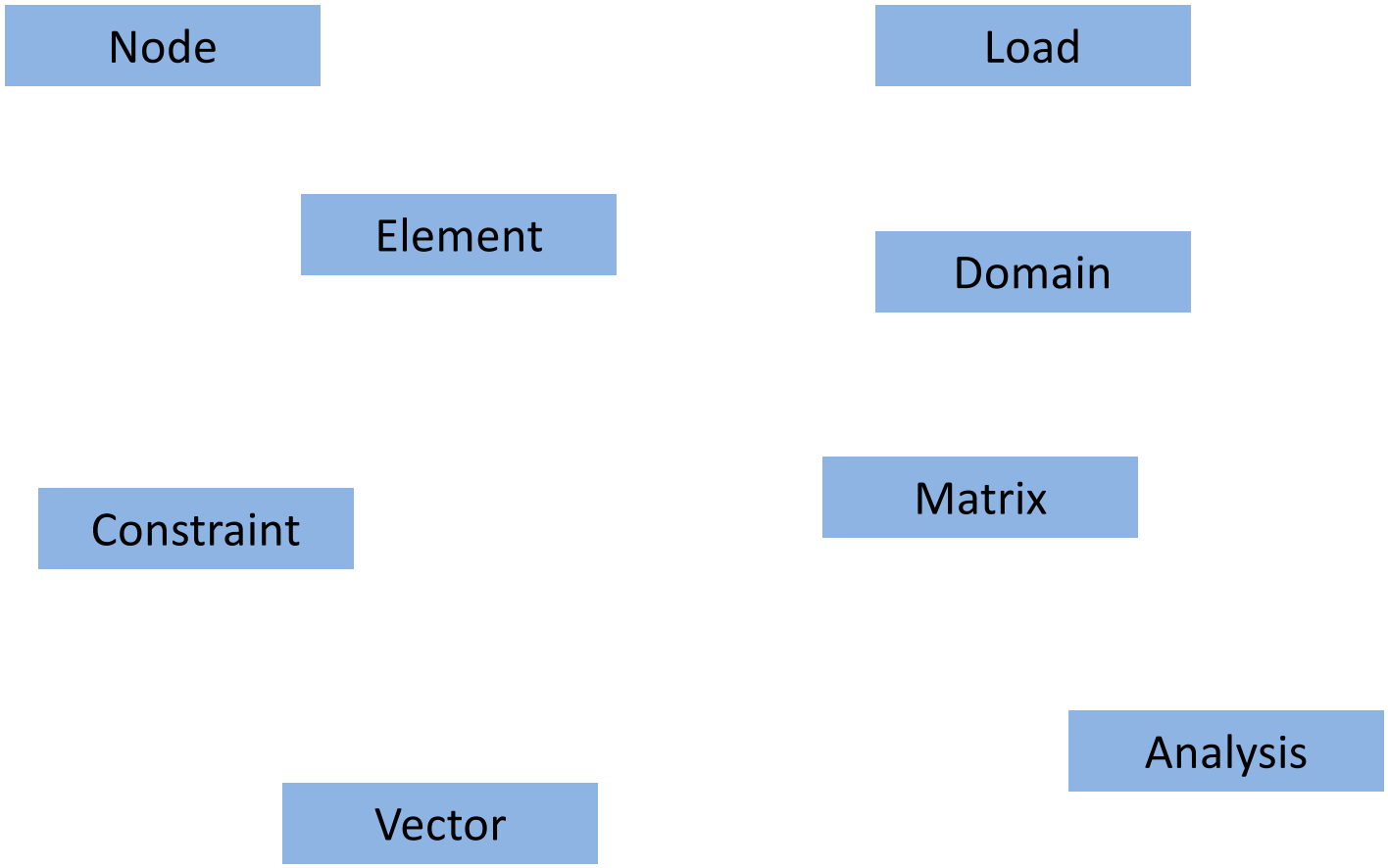
- **Abstraction**
- **C Programming Language Contd.**
  - Structures
  - Containers

C Structures

A Powerful feature that allows us to put together **our own abstractions**

```
struct structNameName {
   type name;
   ……
 };
```

What 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 >
```

```
#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

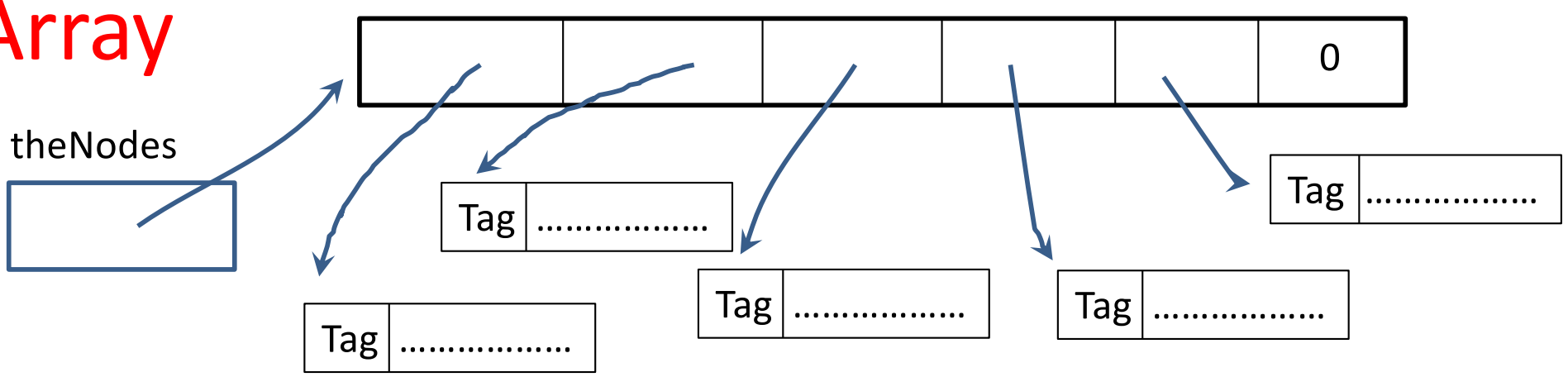
Domain

- 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

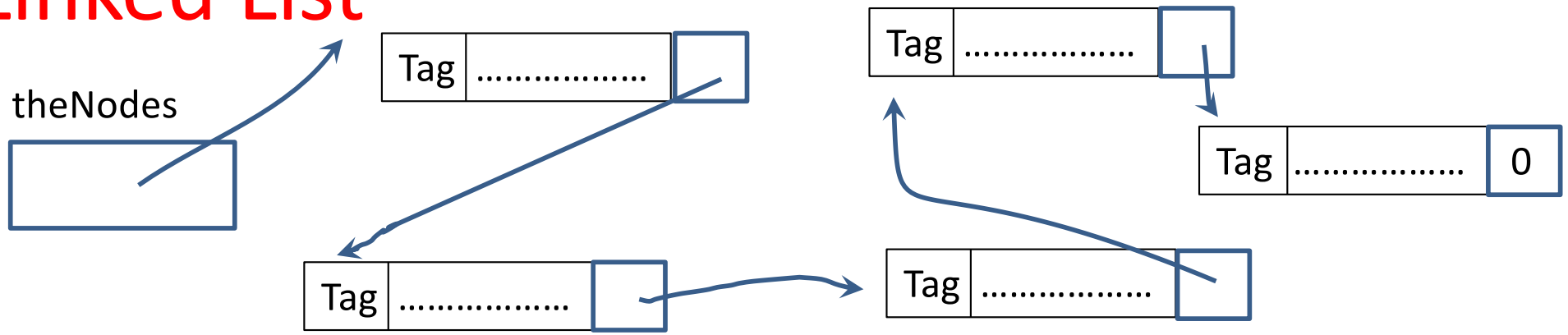


theNodes – pointer to an array of Node *, i.e. each component of array points to a Node.
Want a variable sized array (small and large problems), what happens if too many nodes Added – malloc an even bigger array, copy existing pointers (just address not objects)
=> need Node**, variable to hold current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

typedef struct element {
  int tag;
  int nProps, nHistory;
  int *nodeTags;
  double *props;
  double *history;
  int (*functionPtr)(int*, int (*functionPtr)(double *, double *, double *, double *, …..);
  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.

SimCenter NHERI
Center for Computational Modeling and Simulation

2020 Programming Bootcamp

Object Oriented Programming & C++

Frank McKenna

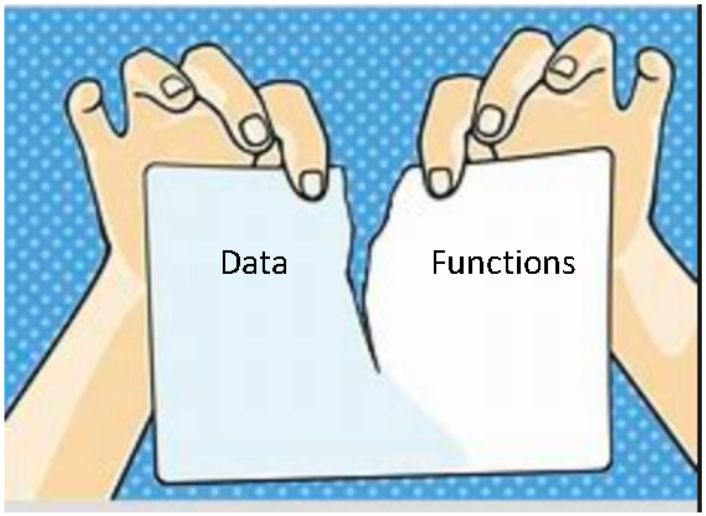
University of California at Berkeley

NSF award: CMMI 1612843

Outline

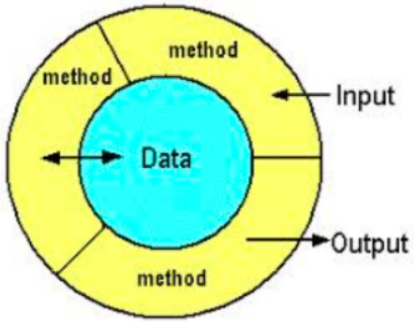
- **Review**
- **Abstraction**
- **C Programming Language Contd.**
  - Structures
  - Containers
- **Object Oriented Programming**
- **C++ Language**

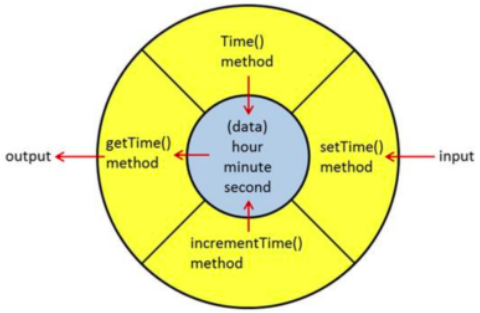
Problem With C is Certain Data & Functions Separate so need these function pointers



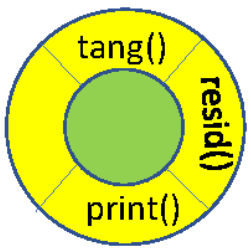
Object-Oriented Programming Offers a Solution

Object-Oriented Programming overcomes the problem by something called **encapsulation** .. The **data and functions(methods) are bundled together** into a class. The class presents an interface, hiding the data and implementation details. If written correctly only the class can modify the data. The functions or other classes in the program can only query the methods, the interface functions.

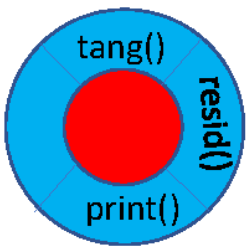




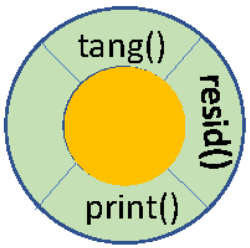
Object-Oriented Programs all provide the ability of one class to inherit the behaviour of a parent class (or even multiple parent classes). This allows the Beam and Trusses both to be treated just as elements. The are said to be polymorphic.



Beam



Truss



Shell

Approaches to Building Manageable Programs

PROCEDURAL DECOMPOSITION

Divides the problem into more easily handled subtasks, until the functional modules (procedures) can be coded

FOCUS ON: procedures

OBJECT-ORIENTED DESIGN

Identifies various objects composed of data and operations, that can be used together to solve the problem

FOCUS ON: data objects

Outline

- **Object Oriented Programming**
- **C++ Language**

The C++ Programming Language

- Developed by Bjourne Stroustroup working at Bell Labs (again) in 1979. Originally “C With Classes” it was renamed C++ in 1983.
- A general purpose programming language providing both functional and object-oriented features.
- As an incremental upgrade to C, it is both strongly typed and a compiled language.
- The updates include:
  - Object-Oriented Capabilities
  - Standard Template Libraries
  - Additional Features to make C Programming easier!

C Program Structure

A ~~C~~ C++ Program consists of the following parts:

- Preprocessor Commands
- Functions
- Variables
- Statements & Expressions
- Comments
- **Classes**

Hello World in C++

Code/C++/hell01.cpp

```
#include <iostream>
int main(int argc, char **argv) {
  // my first C++ program
  std::cout << "Hello World! \n";
}
```

Code/C/hell01.c

```
#include <stdio.h>
int main(int argc, char **argv) {
    // my first program in C
    printf("Hello World! \n");
    return 0;
}
```

pointers, new() and delete()

code/C++/memory1.cpp

```
#include <iostream>
int main(int argc, char **argv) {

  int n;
  double *array1, *array2, *array3;
  std::cout << "enter n: ";
  std::cin >> n;

  // allocate memory & set the data
  array1 = new double[n]; // replaces: (double*)malloc(n*sizeof(double);
  for (int i=0; i<n; i++) {
    array1[i] = 0.5*i;
  }
  array2 = array1;
  array3 = &array1[0];
  for (int i=0; i<n; i++, array3++) {
    double value1 = array1[i];
    double value2 = *array2++;
    double value3 = *array3;
    printf("%.4f %.4f %.4f\n", value1, value2, value3);
  }
  // free the array
  delete array1; // replaces: (array1) free
}
```

strings

code/C++/string1.cpp

```
#include <iostream>
#include <string>

int main(int argv, char **argc) {
  std::string pName = argc[0];
  std::string str;
  std::cout << "Enter Name: ";
  std::cin >> str;


  if (pName == "./a.out")
    str += " the lazy sod";

  str += " says ";
  str = str + "HELLO World";
  std::cout << str << "\n";

  return 0;
}
```

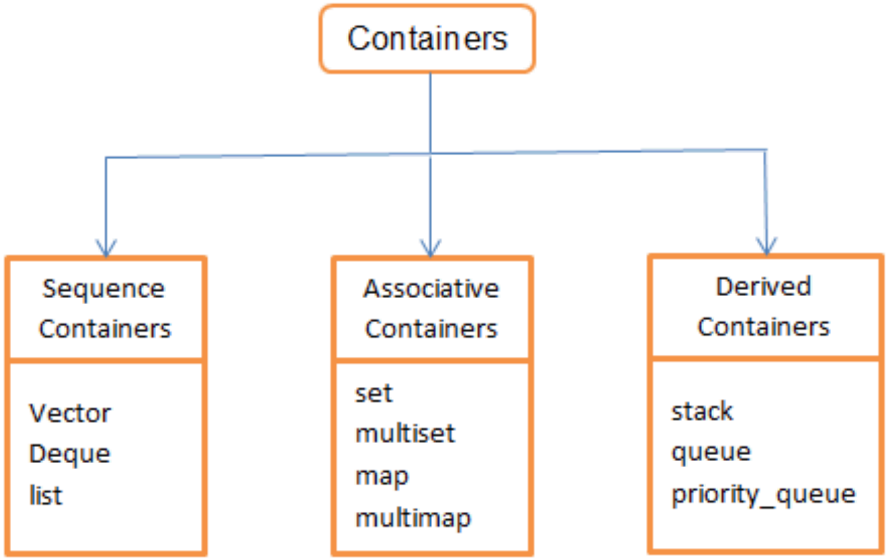
Pass by reference

Code/C++/ref1.cpp

```
#include <iostream>
void sum1(int a, int b, int *c);
void sum2(int a, int b, int &c);
int main(int argc, char **argv) {
  int x = 10;
  int y = 20;
  int z;
  sum1(x,y, &z);
  std::cout << x << " + " << y << " = " << z << "\n";
  x=20;
  sum2(x, y, z);
  std::cout << x << " + " << y << " = " << z << "\n";
}

void sum1(int a, int b, int *c) {
  *c = a+b;
}
void sum2(int a, int b, int &c) {
  c = a + b;
}
```

STL Library

The Standard Template Library (STL) is a set of C++ template classes to provide common programming data structures and functions such as lists, stacks, arrays, etc. It is a library of container classes, algorithms, and iterators.



Will hold off on an example for now

Class

A class in C++ is the programming code that defines the methods (defines the api) in the class interface and the code that implements the methods. For classes to be used by other classes and in other programs, these classes will have the interface in a .h file and the implementation in a .cpp (.cc, .".cxx", or ".c++") file

Will hold off on an example for now

Programming Classes – header file (Shape.h)

```
#ifndef _SHAPES
#defne _SHAPE

class Shape {
  public:
    Shape();
    virtual ~Shape();
    virtual double GetArea(void) =0;
    virtual void PrintArea(ostream &s);

  private:
};

#endif // _SHAPES
```

keyword **class** defines this as a class, **Shape** is the name of the class

Classes can have 3 sections:

- **Public:** objects of all other classes and program functions can invoke this method **on the object**
- **Protected:** only objects of subclasses of this class can invoke this method.
- **Private:** only objects of this specific class can invoke the method.

virtual double GetArea(void) = 0 , the =0;makes this an abstract class. (It cannot be instantiated.) It says the class does not provide code for this method. A subclass must provide the implementation.

virtual void PrintArea(ostream &s) the class provides an implementation of the method, the **virtual** a subclass may also provide an implementation.

virtual ~Shape() is the destccutor. This is method called when the object goes away either through a delete or falling out of scope.

Rectangle.h (in blue)

```
class Shape {
   public:
     Shape();
     virtual ~Shape();
     virtual double GetArea(void) =0;
     virtual void PrintArea(ostream &s);
};
```

```
class Rectangle: public Shape {
 public:
  Rectangle(double w, double h);
  ~Rectangle();
  double GetArea(void);
  void PrintArea(std::ostream &s);

 protected:

 private:
  double width, height;
  static int numRect;
};
```

- **class Rectangle: public Shape** defines this as a class, **Rectangle** which is a subclass of the class Shape.
- It has 3 sections, public, protected, and private.
- It has a constructor **Rectangle(double w, double h)** which states that class takes 2 args, w and h when creating an object of this type.
- It also provides the methods double GetArea(void) and void
- PrintArea(ostream &s); Neither are virtual which means no subclass can provide an implementation of these methods.
- In the private area, the class has 3 variables. Width and height are
- unique to each object and are not shared. Num rect is shared amongst all objects of type Rectangle.

Circle.h

```
class Shape {
   public:
     Shape();
     virtual ~Shape();
     virtual double GetArea(void) =0;
     virtual void PrintArea(ostream &s);
};
```

```
#ifndef _CiIRCLE
#define _CIRCLE
class Circle: public Shape {
 public:
  Circle(double d);
  ~Circle();
  double GetArea(void);


 private:
  double diameter;
  double GetPI(void);
};
#endif // _CIRCLE
```

- **class Circle: public Shape** defines this as a class **Circle** which is a subclass of the class Shape.
- It has 2 sections, public and private.
- It has a constructor **Circle(double d)** which states that class takes 1 arg d when creating an object of this type.
- It also provides the method **double GetArea(void).**
- **There is no PrintArea() method,meaning this class relies on the base class implementation.**
- In the private area, the class has 1 variable and defines a private method, GetPI(). Only objects of type Circle can invoke this method.

Programming Classes – source file (Shape.cpp)

```
#include <Shape.h>

Shape::Shape() {

}

Shape::~Shape() {
  std::cout << "Shape Destructor\n";
}



void
Shape::PrintArea(std::ostream &s) {
  s << "UNKOWN area: " <<
       this->GetArea() << "\n";
}
```

- Source file contains the implementation of the class.
- 3 methods provided. The constructor Shape(), the destructor ~Shape() and the PrintArea() method. A definition for each method defined in the header file.
- TheDestructor just sends a string to cout.
- The PrintAreamethods prints out the area. It obtains the area by invoking the **this** pointer.
- **This pointer is not defined in the .h file or .cpp file anywhere as a variable. It is a default pointer always available to the programmer. It is a pointer pointing to the object itself.**

Rectangle.cpp

```
int Rectangle::numRect = 0;

Rectangle::~Rectangle() {
  numRect--;
  std::cout << "Rectangle Destructor\n";
}

Rectangle::Rectangle(double w, double d)
  :Shape(), width(w), height(d)
{
  numRect++;
}

double
Rectangle::GetArea(void) {
  return width*height;
}
void
Rectangle::PrintArea(std::ostream &s) {
  s << "Rectangle: " << width * height <<
        " numRect: " << numRect << "\n";
}
```

- **int Rectangle::numRect = 0** creates the memory location for the classes static variable numRect.
- The **Rectangle::Rectangle(double w, double d) is the class constructor taking 2 args.**
- the line **:Shape(), width(w), height(d)** is the first code exe. It calls the base class constructo and then sets it's 2 private variables.
- The constructor also increments the static variable in **numRect++;** That variable is decremented in the **destructor.**
- **The GetArea()** method, which computes the area can access the private data variables height and width

Circle.cpp

```
#include <Circle.h>

Circle::~Circle() {
  std::cout << "Circle Destructor\n";
}

Circle::Circle(double d) {
  diameter = d;
}

double
Circle::GetArea(void) {
  return this->GetPI() * diameter *
diameter/4.0;
}

double
Circle::GetPI(void) {
  return 3.14159;
}
```

- Last but not least!

Main Program (main1.cpp)

```
# #include "Rectangle.h"
#include "Circle.h"

int main(int argc, char **argv) {
  Circle s1(2.0);
  Shape *s2 = new Rectangle(1.0, 2.0);
  Shape *s3 = new Rectangle(3.0,2.0);

  s1.PrintArea(std::cout);
  s2->PrintArea(std::cout);
  s3->PrintArea(std::cout);

  return 0;
}
```

When we run it, results should be as you expected. Notice the destructors for s2 and s3 objects not called. The **delete** was not invoked. Also notice order of destructor calls, base class **destructed** last.

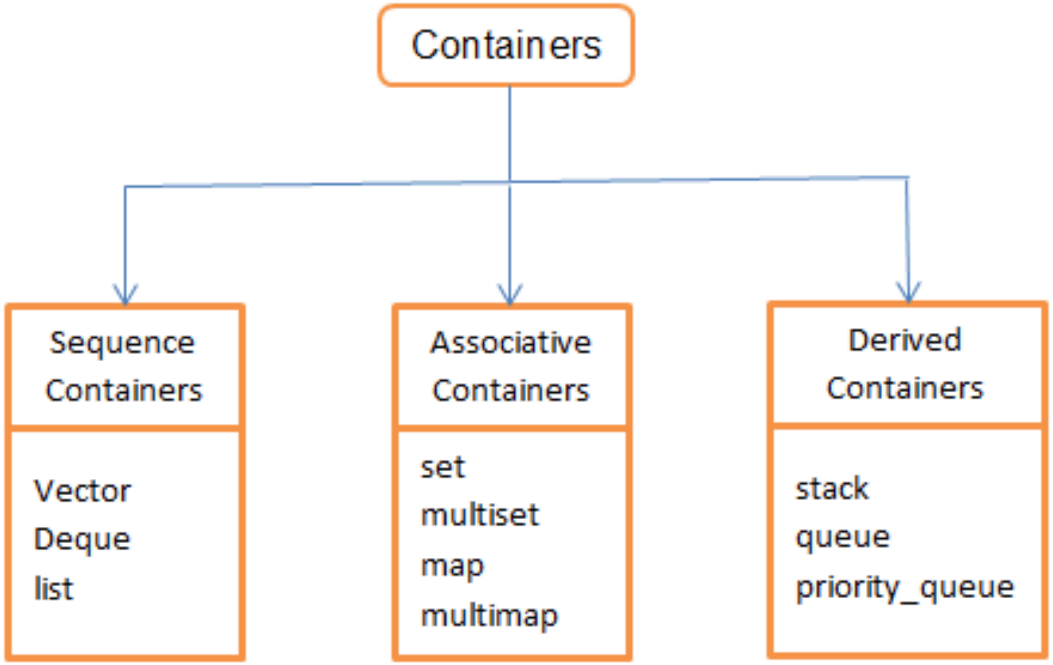
s1 is a variable of type Circle. To invoke methods on this object we use the **DOT .**

s2 and s3 are pointers to objects created with **new.** To invoke methods on these objects from our pointer variables we use the **ARROW ->**

```
shapes >g++ Circle.cpp -I ./ -c
shapes >g++ Rectangle.cpp -I ./ -c
shapes >g++ Shape.cpp -I ./ -c
shapes >g++ main1.cpp Rectangle.o Circle.o Shape.o -I ./ ; ./a.out
UNKOWN area: 3.14159
Rectangle: 2 numRect: 2
Rectangle: 6 numRect: 2
Circle Destructor
Shape Destructor
shapes >
```

STL Library

The Standard Template Library (STL) is a set of C++ template classes to provide common programming data structures and functions such as lists, stacks, arrays, etc. It is a library of **container classes**, algorithms, and **iterators**.



Main Program with STL Container

C++/shape/main2.cpp

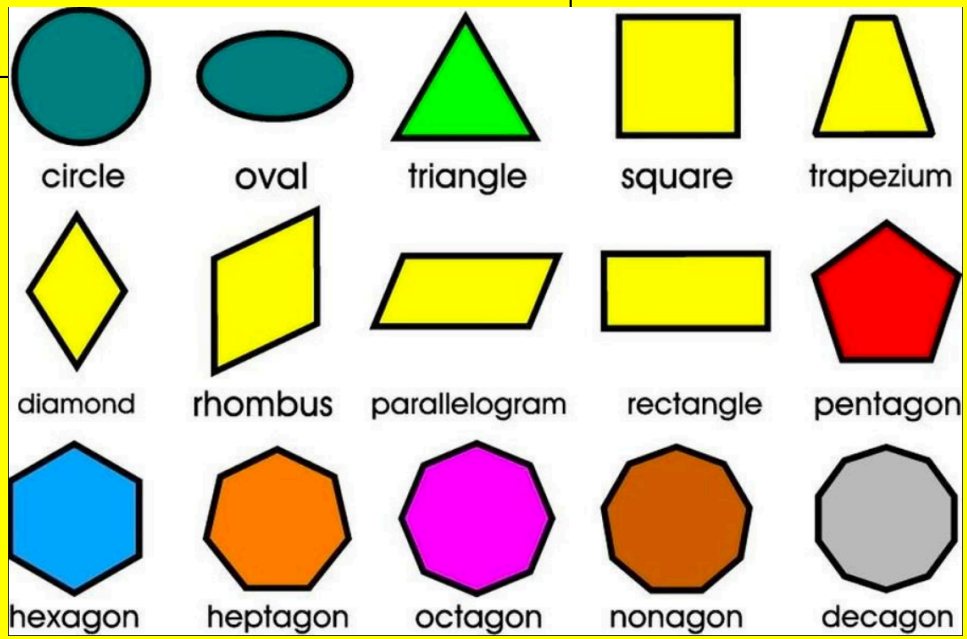
```
#include "Rectangle.h"
#include "Circle.h"
#include <list>
int main(int argc, char **argv) {
  std::list<Shape*> theShapes;

  Circle s1(2.0);
  Shape *s2 = new Rectangle(1.0, 2.0);
  Shape *s3 = new Rectangle(3.0,2.0);

  theShapes.push_front(&s1);
  theShapes.push_front(s2);
  theShapes.push_front(s3);

  std::list<Shape *>::iterator it;
  for (it = theShapes.begin();
    it != theShapes.end(); it++) {
    (*it)->PrintArea(std::cout);
  }
  return 0;
}
```

C++/shape/main3.cpp

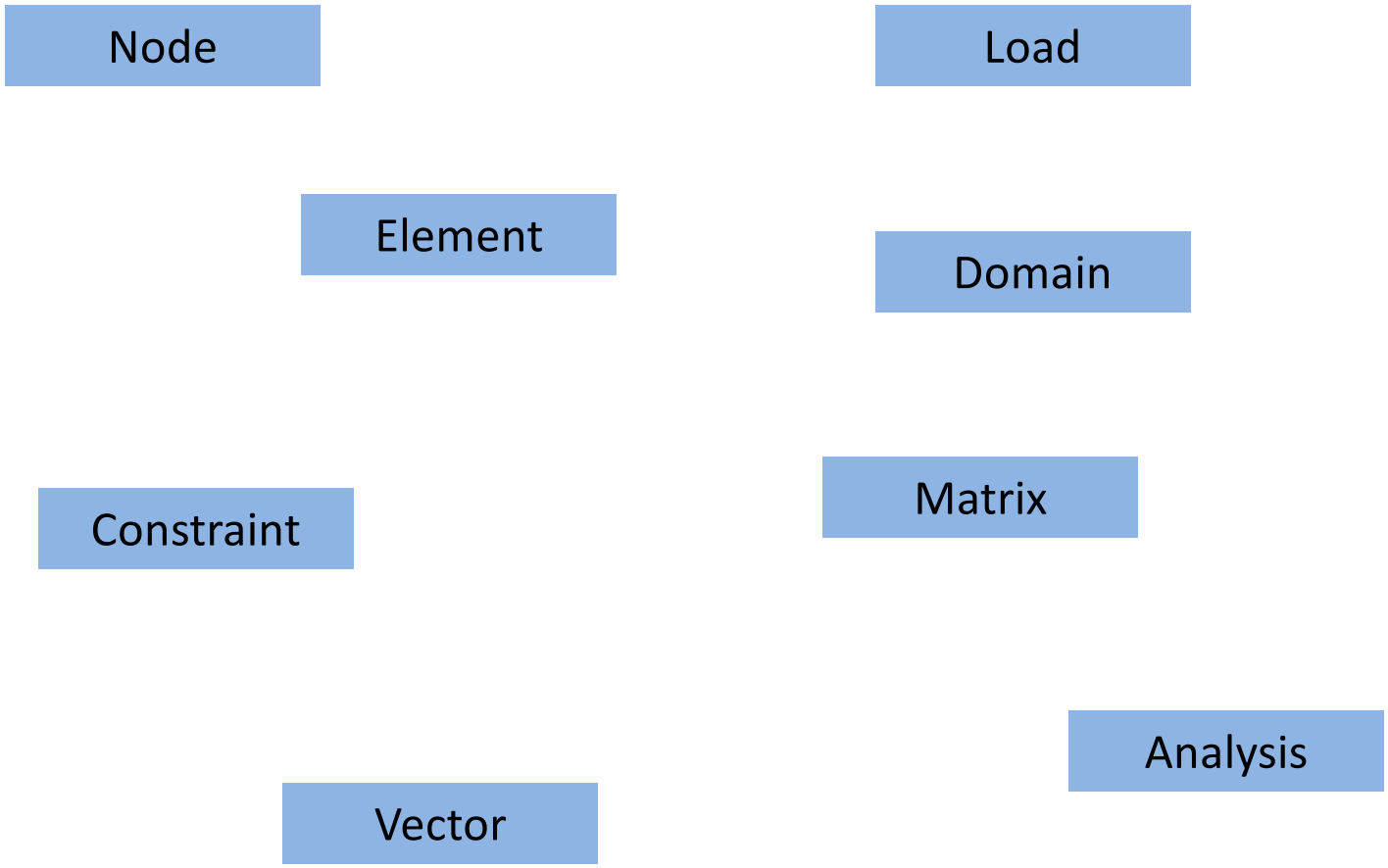
```
#include "Rectangle.h"
#include "Circle.h"
#include <list>
#include <vector>
typedef std::list<Shape*> Container;
//typedef std::vector<Shape*> Container;
typedef Container::iterator Iter;

int main(int argc, char **argv) {
  Container theShapes;

  Circle s1(2.0);
  Shape *s2 = new Rectangle(1.0, 2.0);
  Shape *s3 = new Rectangle(3.0,2.0);

  theShapes.push_front(&s1);
  theShapes.push_front(s2);
  theShapes.push_front(s3);

  Iter it;
  for (it = theShapes.begin(); it != theShapes.end(); it++) {
    (*it)->PrintArea(std::cout);
  }
  return 0;
}
```

Exercise: Add Some Other Shape

1. cp rectangle.h to ?.h
2. cp rectangle.c to ?.cpp
3. Edit both files, global replace …
4. Compile & Execute



C++ Finite Element Application?



Domain.h

C++/fem/domain.h

```
#ifndef _DOMAIN
#define _DOMAIN

#include "Domain.h"
#include <map>

class Node;

class Domain {
 public:
  Domain();
  ~Domain();

  Node *getNode(int tag);
  void Print(ostream &s);
  int AddNode(Node *theNode);

 private:
  std::map<int, Node *>theNodes;
};

#endif
```

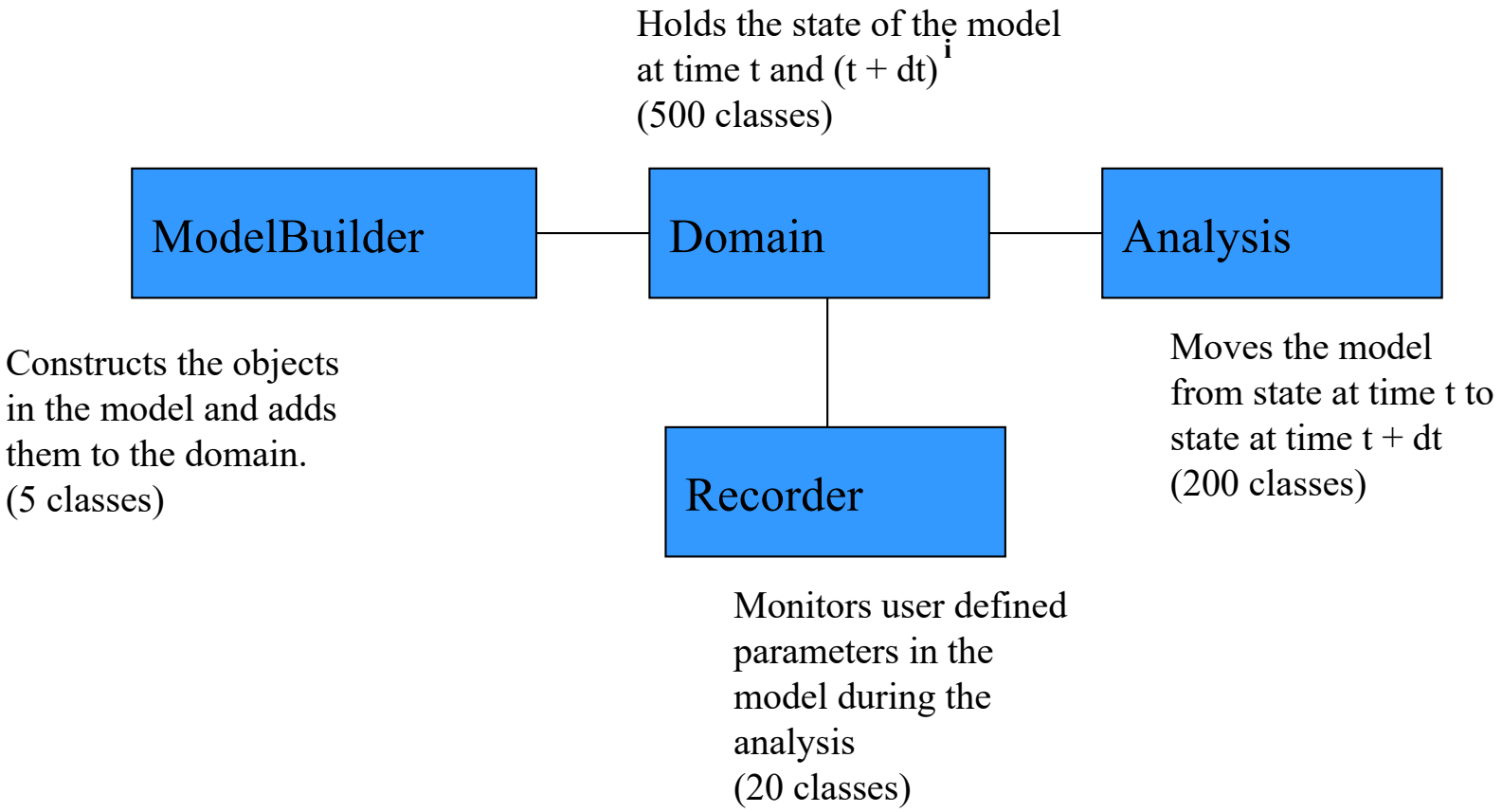
- The #ifndef, #define, #endif are important. You should put them in every header file

- Storing nodes in a map

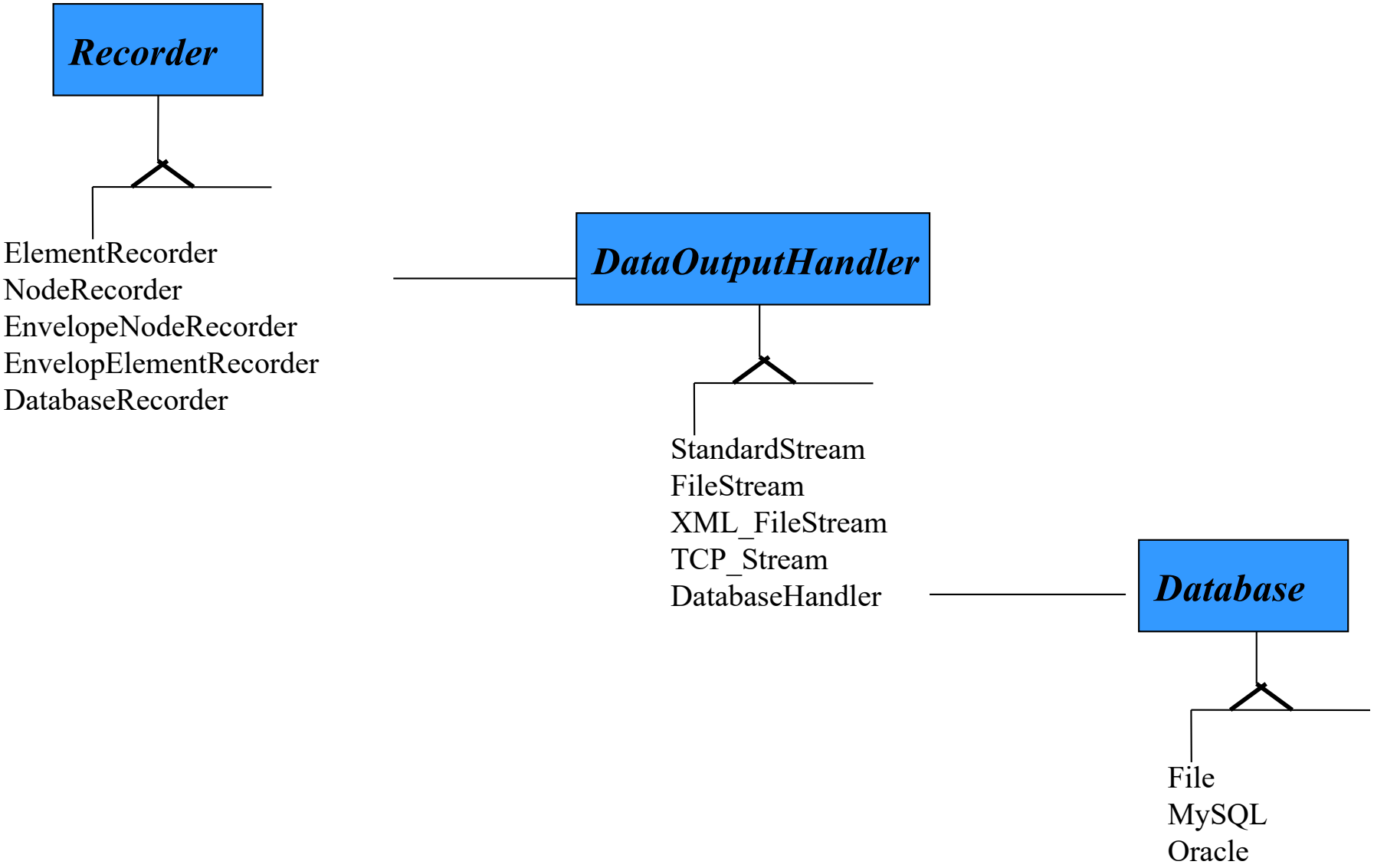
C++/fem/domain.cpp

```
Domain::Domain() {
  theNodes.empty();
}

Node *
Domain::getNode(int tag){
  Node *res = NULL;
  std::map<int, Node *>::iterator it = theNodes.find(tag);
  if (it != theNodes.end()) {
      Node *theNode = it->second;
      return theNode;
  }
  return res;
}

void
Domain::Print(ostream &s){
  // create iterator & iterate over all elements
  std::map<int, Node *>::iterator it = theNodes.begin();

  while (it != theNodes.end()) {
      Node *theNode = it->second;
      theNode->Print(s);
      it++;
    }
```

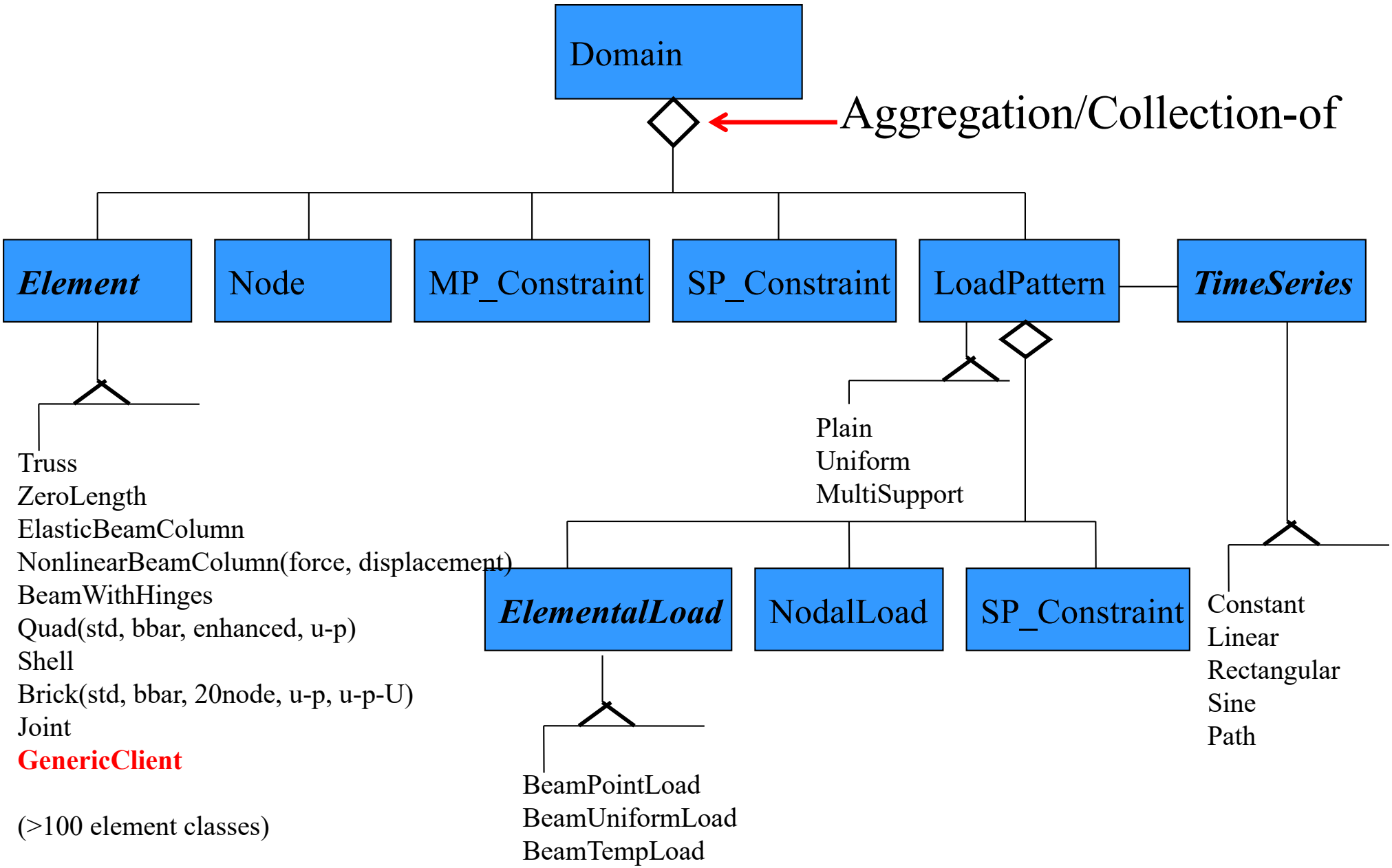
Main Abstractions in OpenSees Framework as an Example of OOP Design



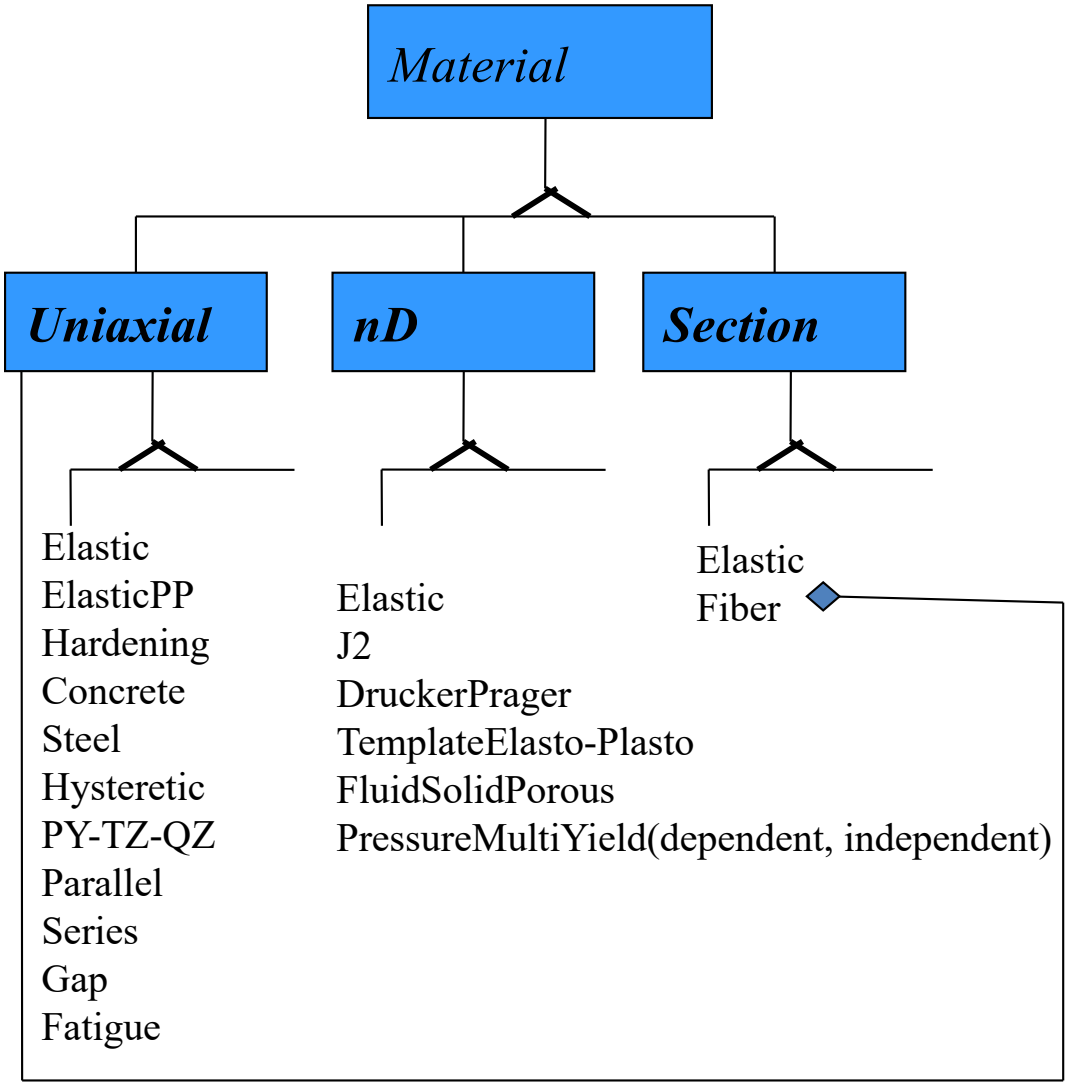
Recorder Options



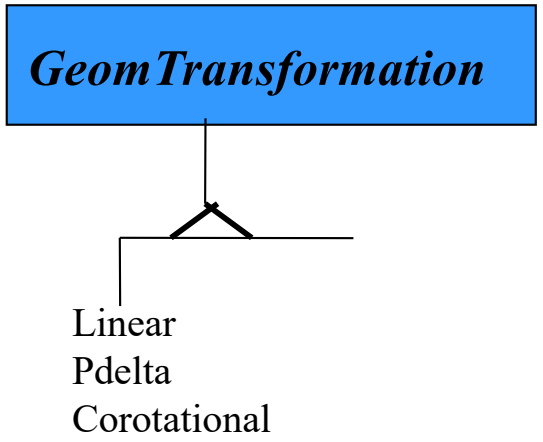
What is in a Domain?



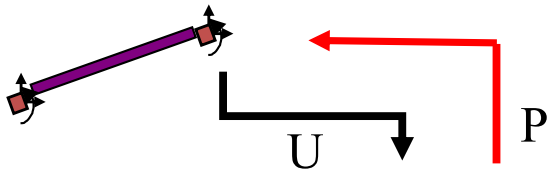
Some Other Classes associated with Elements:



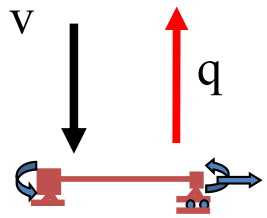
(over 250 material classes)



Element in Global System

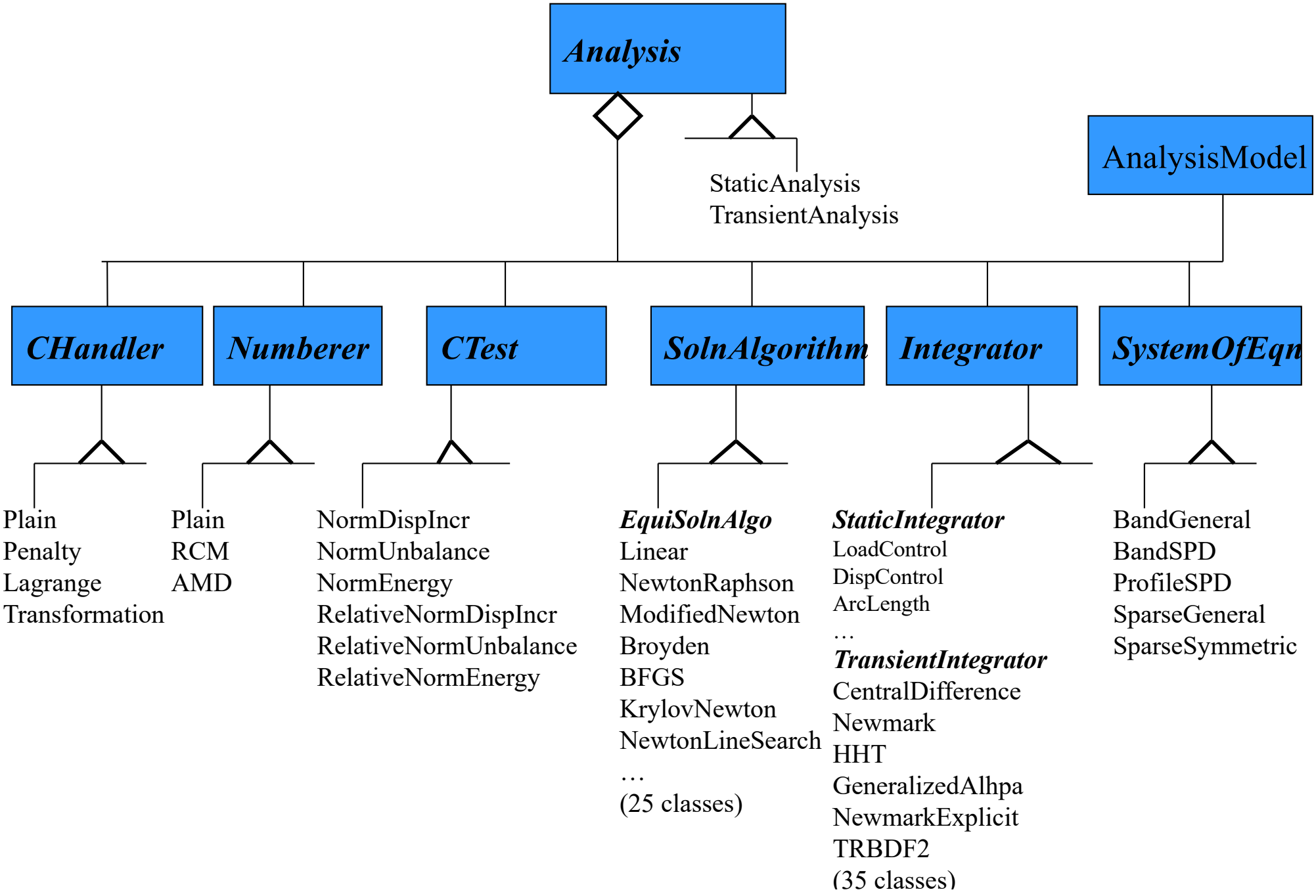


Geometric Transformation



Element in Basic System

What is an Analysis?



Exercise: Every *Good FEM program needs a Vector class*

1. Fetch and merge upstream SimCenterBootcamp2020
2. cd to code/c++/vector
3. There is a file Makefile there
4. Type make
5. It will fail .. Missing Vector.cpp .. Needs someone (YOU) to create one
6. There is a main.cpp, which when you have Vector.cpp correct will run w/o error
7. I will demo getting an initial Vector.cpp file and upload it
8. You get to finish it

```
COMPILER = icc
FLAGS = -D_DEBUG

all:
    $(COMPILER) $(FLAGS) Vector.cpp -c -o Vector.o
    $(COMPILER) $(FLAGS) Matrix.cpp -c -o Matrix.o
   $(COMPILER) $(FLAGS) main.cpp Vector.o Matrix.o

test: all
        ./a.out

clean:
        rm *~ *.o
```