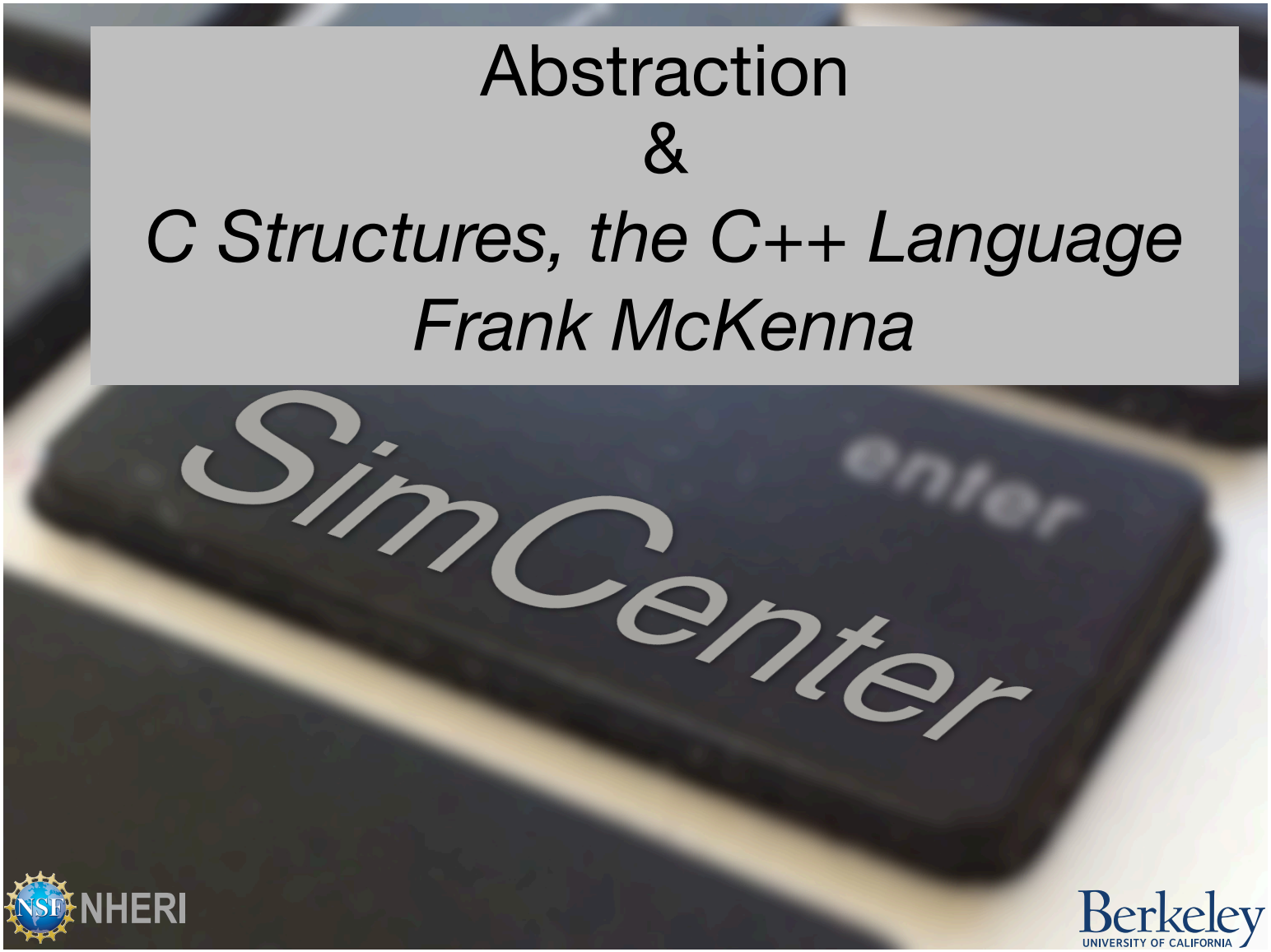
Time	Title	Presenter
Day 1	Programming & The C Language	Frank McKenna & You
Day 2	Parallel Programming: MPI and OpenMP	Peter Mackenzie-Helnwein
Day 3	Introduction	Peter Mackenzie-Helnwein
	Abstraction	Frank McKenna
12.00-1:00	C structures	
1:00-3:00	An Introduction to Programming & The C Programming Language (contd)	Frank McKenna
1:00-3:00	Hands-On	Frank McKenna
3:00-5:00	Exercises	You
Day 2	Debugging, Parallel Programming with MPI & OpenMP	
Day 3	Abstraction, More C & C++	
Day 4	User Interface Design & Qt	
Day 5	SimCenter & Cloud Computing	

SimCenter NHERI
Center for Computational Modeling and Simulation



Abstraction
&
C Structures, the C++ Language

Frank McKenna

NHERI

Berkeley
UNIVERSITY OF CALIFORNIA

Outline

- Abstraction
- C Programming Language –
  - Structures
  - Container Classes

  - Object-Oriented Programming
  - C++

Definition – 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)

Abstraction

- "*The process of removing physical, spatial, or temporal details*[2] *or attributes in the study of objects or systems in order to more closely attend to other details of interest*" [source: wikippedia]

We Work in Decimal

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

Computers in Binary

0,1

Computer Bit (on/off) (0,1)

We Combine Numbers

100 10 1

456

4*100 + 5*10 + 6

With 3 numbers we can represent any number 0 through 999

What can we represent on a computer with 3 bits

000
001
010
011
100
101
110
111

2^3 possibilities

We Combine Numbers

4 2 1

101

1*4 + 0*2 + 1*1

With 3 numbers we can represent any number 0 through 7

What can we represent on a computer with 3 bits

000	0	A
001	1	B
010	2	C
011	3	D
100	4	E
101	5	F
110	6	G
111	7	H

Computer groups bits into Bytes

1 Byte = 8 bits

2^8 = 256 possibilities

C Data Types

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

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

C Program to Print Character Set

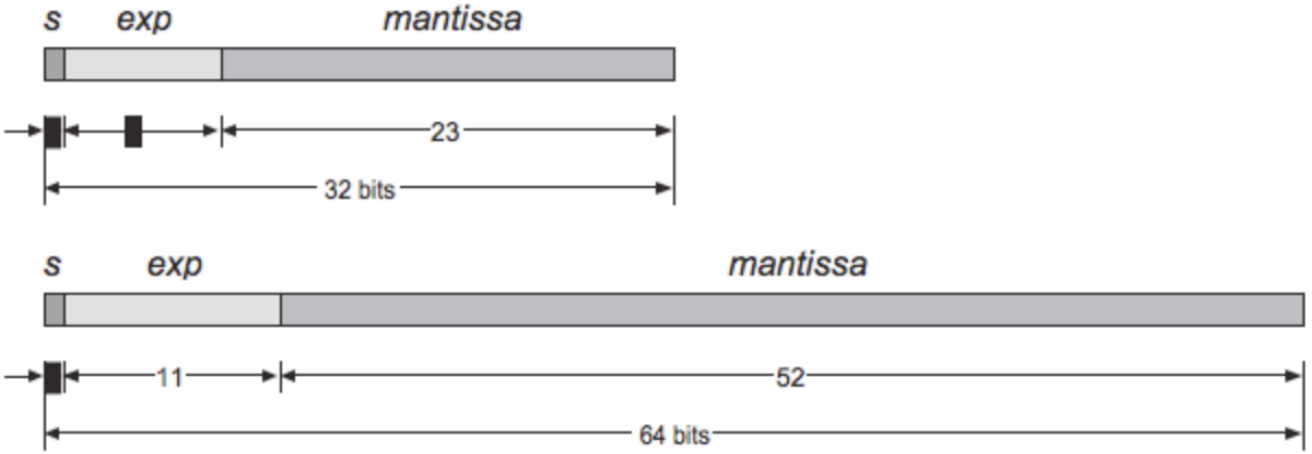
char	1 byte	-128 to 127 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127
int	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
long	4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

charset.c

```
#include<stdio.h>
#include<conio.h>
int main() {
    int i; clrscr();
    printf("ASCII ==> Character\n");
    for(i = -128; i <= 127; i++)
        printf("%d ==> %c\n", i, i);
     return 0;
}
```

Float and Double Point Numbers - IEEE 754 standard

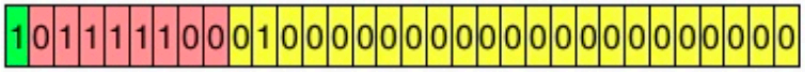
Single Precision



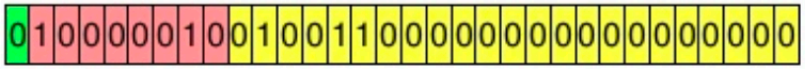
Double Precision

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

- What is the decimal value of this Single Precision float?



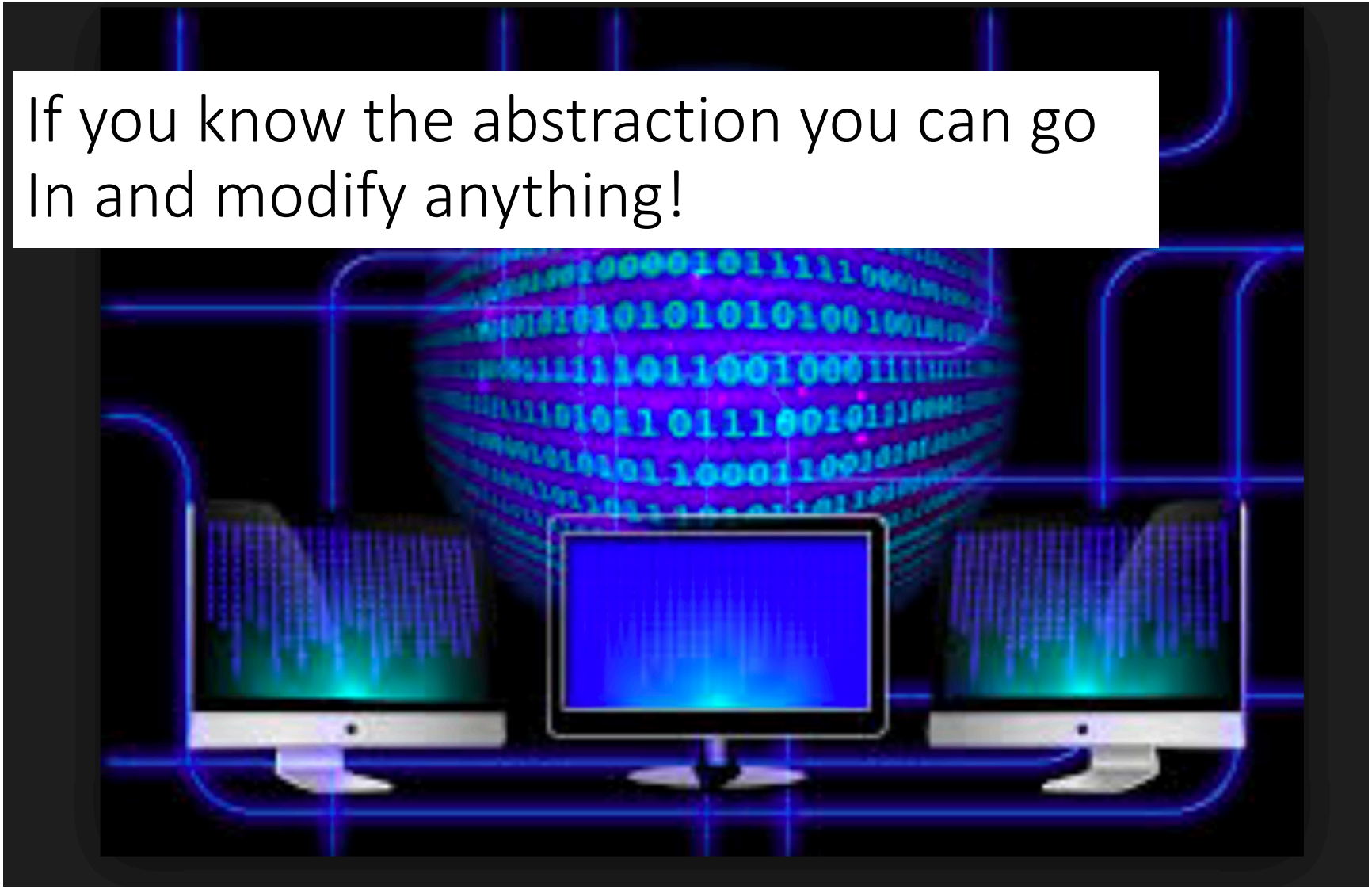
- Solution:
  - Sign = 1 is negative
  - Exponent = $(01111100)_2 = 124$, E – bias = 124 – 127 = –3
  - Significand = $(1.0100 \ldots 0)_2 = 1 + 2^{-2} = 1.25$ (1. is implicit)
  - Value in decimal = $-1.25 \times 2^{-3} = -0.15625$
- What is the decimal value of?



- Solution:
  - Value in decimal = $+(1.01001100 \ldots 0)_2 \times 2^{130-127} =$ $(1.01001100 \ldots 0)_2 \times 2^3 = (1010.01100 \ldots 0)_2 = 10.375$

implicit

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

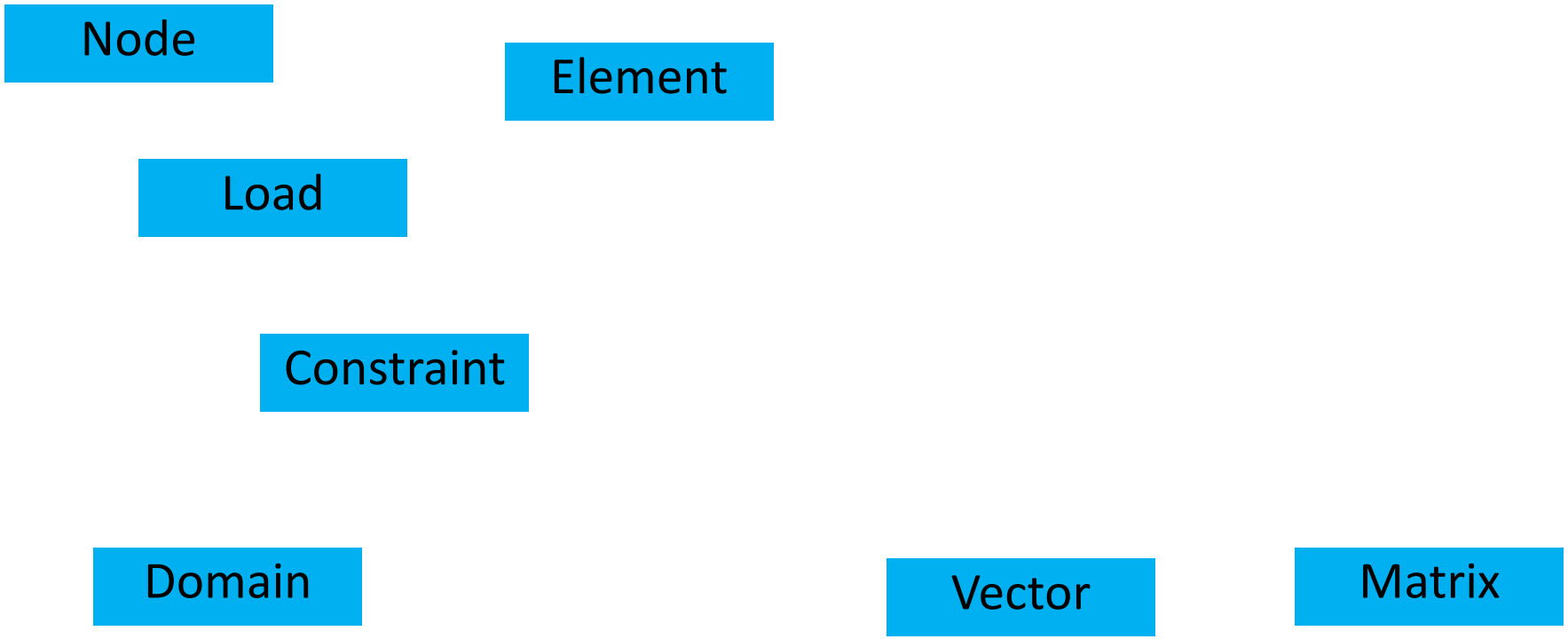


C Structures

- A powerful tool for developing your own data abstractions

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

What Abstractions for a Finite Element Method Application?



What is in a Node?

- 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 Large Project:

- Files for each data type and the functions
  - node.h, node.c, domain.h, domain.c, ….

```
#include “node.h”
#include “domain.h”
int main(int argc, const char **argv) {
    Domain theDomain;

    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

- Container class to store nodes, elements, loads, constraints,…
- What storage scheme for the different data types?
- What are the options:
  - Array
  - Linked List
  - Double Linked List
  - Tree
  - Hybrid approaches ……

```
#ifndef _DOMAIN
#define _DOMAIN

#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);
#endif
```

```
#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
```

```
void domainAddNode(Domain *theDomain, int tag, double crd1, double crd2) {
    Node *theNextNode = (Node *)malloc(sizeof(Node));
    nodeSetup(theNextNode, tag, crd1, crd2);

    if (theDomain->theNodes != NULL) {
         theNextNode->next = theDomain->theNodes;
    }
    theDomain->theNodes = theNextNode;
}
void domainPrintNodes(Domain *theDomain) {
    Node *theCurrentNode = theDomain->theNodes;
    while (theCurrentNode != NULL) {
        nodePrint(theCurrentNode);
        theCurrentNode = theCurrentNode->next;
    };
}
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;
}
```

Object Oriented Programming and C++

How do We Now Add Elements to the FEM code?

- Want 2d beam elements

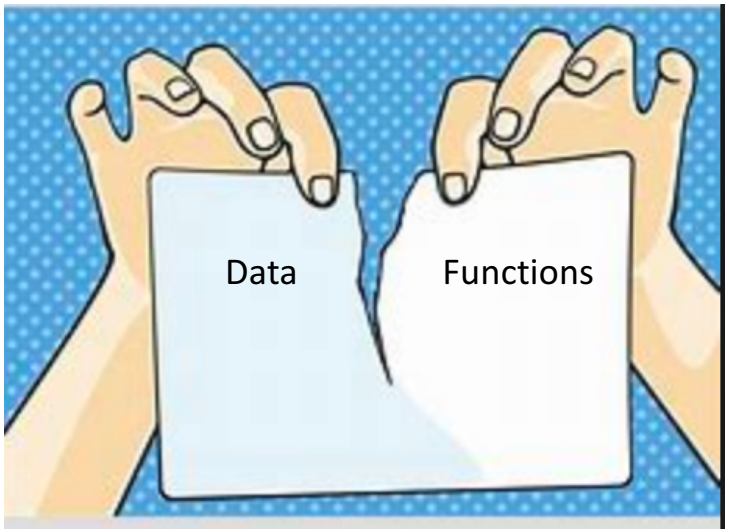
```
typedef struct struct_domain {
    Node *theNodes;
    Constraints *theConstraints;
    Beam *theBeams
}
```

And Trusses!

```
typedef struct struct_domain {
    Node *theNodes;
    Constraints *theConstraints;
    Beam*theBeams;
    Truss *theTrusses;
}
```

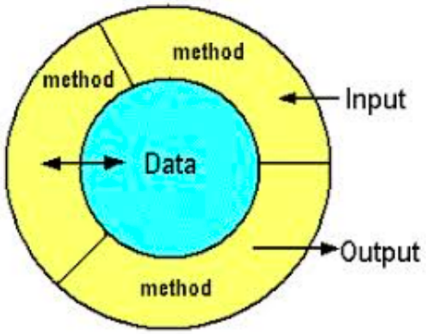
Why Not Just Elements .. That requires some functional pointers!

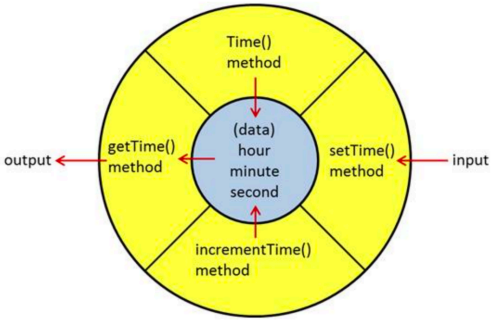
Problem With C is Certain Data & Functions Separate so need these function pointers



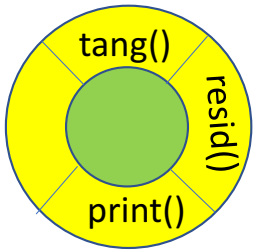
Object-Oriented Programming Offers a Solution

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.

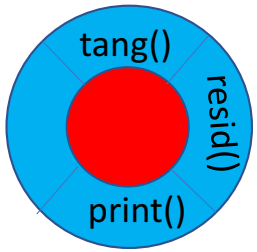




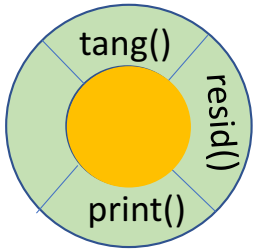
Object-Oriented Programs all provide the ability of one class to inherit the behaviour of a parent class (or even multiple parent classes). This allows the Beam and Trusses both to be treated just as elements. The are said to be polymorphic.



Beam



Truss



Shell

C++

- 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 Programmng easier!

C++ Program Structure

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

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

Hello World (of course C Hello World also works!)

Code/C++/hello1.cpp

```
#include <iostream>
using namespace std;

int main() {
   /* my first program in C++ */
   cout << "Hello World! \n";
   return 0;
}
```

- The first line of the program ***#include <iostream>*** is again a preprocessor directive, which tells a C++ compiler to include the iostreamfile before starting compilation.
- All the elements of the standard C++ library are declared within what is called a namespace. In this case the namespace with the name std. We put the **using namespace std** line in to declare that we will make use of the functionality offered in the namespace std.
- The next line ***int main()*** is the main function. Every program must have a main function as that is where the program execution will begin.
- The next line **/*...*/** will be ignored by the compiler. It is there for the programmer benefit. It is a comment.
- The next line is a statement to send the message "Hello, World!" to the output stream cout, casuing it to be displayed on the screen.
- The next statement **return 0;** terminates the main() function and returns the value 0.

pointer, new() and delete()

Code/C++/memory1.cpp

```
#include <iostream>
using namespace std;

int main(int argc, char **argv) {
  int n;
  double *array1=0, *array2=0, *array3=0;

 // get n
  cout << "enter n: ";
   cin >> n;
  if (n <=0) {printf ("You idiot\n"); return(0);}

  // allocate memory & set the data
  array1 = new double[n];
  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[];
  return(0);
}
```

You should not malloc something and delete it later or new something and free it later. The bahviour is undefined. "The program may continue normally, it may crash immediately, it may produce a well-defined error message and exit gracefully, it may start exhibiting random errors at some time after the actual undefined behavior event".

```
c >gcc memory1.c; ./a.out
enter n: 5
0.0000 0.0000 0.0000
0.5000 0.5000 0.5000
1.0000 1.0000 1.0000
1.5000 1.5000 1.5000
2.0000 2.0000 2.0000
c >./a.out
enter n: 3
0.0000 0.0000 0.0000
0.5000 0.5000 0.5000
1.0000 1.0000 1.0000
c >
```

strings

```
#include <iostream>
#include <string>
using namespace std;

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

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

  str += " says ";
  str = str + "HELLO World";
  cout << str << "\n";
  return 0;
}
```

Classes

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.

Programming Classes – Header Files

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

- keywod **class** defines this as a class, **Shape** is the name of the class
- Classes can have 3 sections:
  1. **Public**: objects of all other classes and program functions can invoke this method on the object
  2. **Protected**: only objects of subclasses of this class can invoke this method.
  3. **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 destrcutor.** This is method called when the object goes away either through a delete or falling out of scope.

```
class Shape {
 public:
   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(ostream &s);
  protected:
      // shared by subclasses
   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. Numrect is shared amongst all objects of type Rectangle.

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

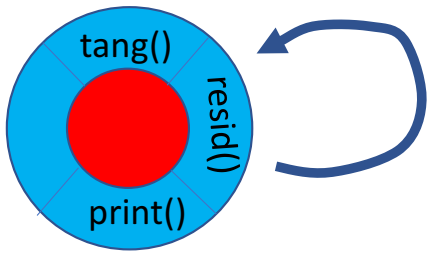
```
class Circle: public Shape {
  public:
    Circle(double d);
    ~Circle();
    double GetArea(void);
 private:
   double diameter;
   double GetPI(void);
};
```

- **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 objets of type Circle can invoke this method.

Programming Classes –Implementation Files

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

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



- 2 methods defined. The destructor ~Shape() and the PrintArea() method.
- The Destructor just sends a string to cout.
- The PrintArea methods 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.**

```
int Rectangle::numRect = 0;

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

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

double
Rectangle::GetArea(void) {
    return width*height;
}
```

- **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 constructor and then sets it's 2 private variables.
- The constructor also increments the static variable in **numRect++;** That variable is decremented in the **destructor.**

```
Circle::~Circle() {
    cout << "Shape 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!

A main.c

```
#include <iostream>
#include "shape1.h"


using namespace std;


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(cout);
  s2->PrintArea(cout);
  s3->PrintArea(cout);

  return 0;
}
```

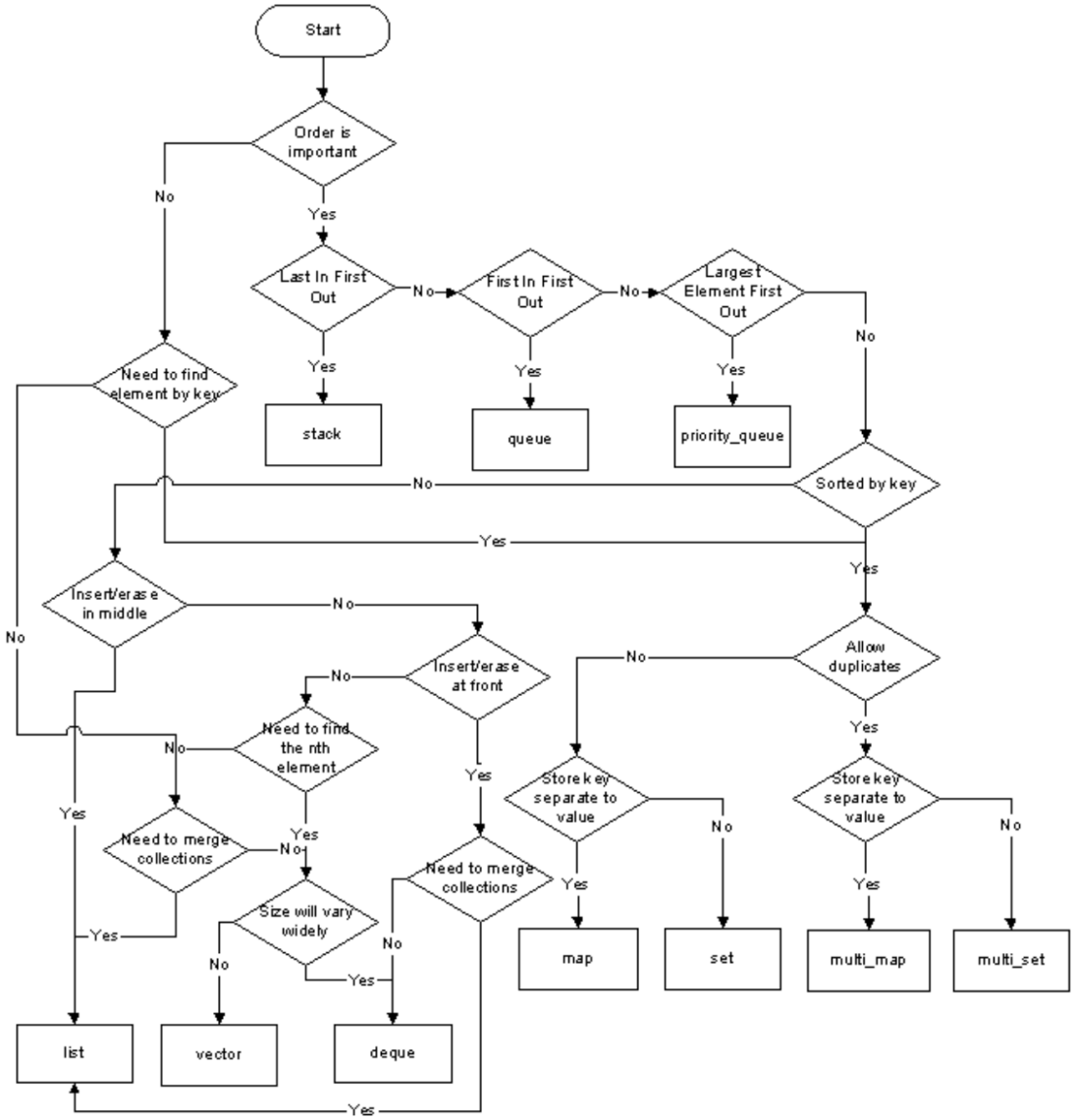
```
C++ >g++ shape1.cpp; ./a.out
UNKOWN area: 3.14159
Rectangle: 2 numRect: 2
Rectangle: 6 numRect: 2
Circle Destructor
Shape Destructor
C++ >
```

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 ->**

Containers


Start
Order is important
No
Yes
Last In First Out
No
First In First Out
No
Largest Element First Out
No
Yes
Yes
Yes
stack
queue
priority_queue
Need to find element by key
Sorted by key
No
Yes
Yes
Insert/erase in middle
No
No
Insert/erase at front
No
Allow duplicates
No
Yes
Need to find the nth element
No
Yes
Store key separate to value
No
Store key separate to value
No
Yes
Yes
Yes
Need to merge collections
No
Yes
Need to merge collections
Yes
Yes
Size will vary widely
No
No
Yes
No
map
set
multi_map
multi_set
list
vector
deque
Yes

Domain.h

```
#ifndef _DOMAIN
#define _DOMAIN

#include "Domain.h"
#include <map>
class Node;
using namespace std;

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

    Node *getNode(int tag);
    void Print(ostream &s);
    int AddNode(Node *theNode);
 private:
     map<int, Node *>theNodes;
};
#endif
```

- The #ifndef, #define, #endif are important to put in every header file to potentially stop compiler going into an infinite loop.

- To store the nodes we are using a built in STL container of type **map<int, Node *>;**

Domain.cpp

```
Node *
Domain::getNode(int tag){
  Node *res = NULL;

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

  while (it != theNodes.end()) {
     Node *theNode = it->second;
     if (theNode->GetTag() == tag) {
       res = theNode;
       break;
     }
     it++;
  }
  return res;
}
```

We create an iterator for our particular map. Then we simply iterate until we either find the node we want or we reach the end of the elements in the map.

Syntax is bloody awful, but they are very powerful.