Abstraction
C Structures
& C++ Programming Language

Frank McKenna

Outline

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

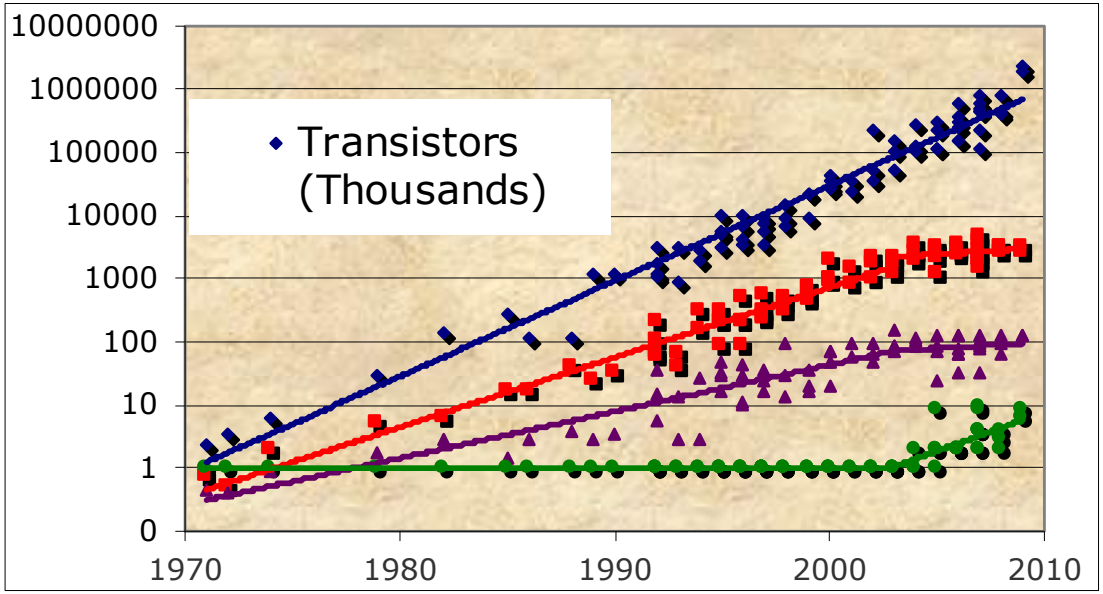
What has made computing pervasive?

Programmability

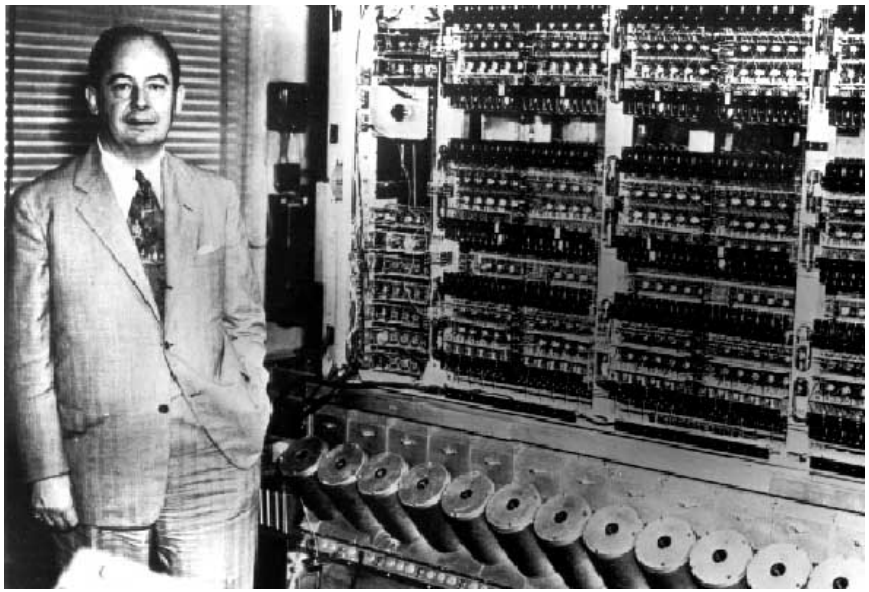
```
public class TcpClientSample
{
    public static void Main()
    {
        byte[] data = new byte[1024]; string input, stringData;
        TcpClient server;
        try{
            server = new TcpClient(" . . . . ", port);
        }catch (SocketException){
            Console.WriteLine("Unable to connect to server");
            return;
        }
        NetworkStream ns = server.GetStream();
        int recv = ns.Read(data, 0, data.Length);
        stringData  = Encoding.
          ASCII.GetString(data, 0, recv);
        Console.WriteLine(stringData);
        while(true){
            input = Console.ReadLine();
            if (input == "exit") break;
                newchild.Properties["ou"].Add
                ("Auditing Department");
                newchild.CommitChanges();
                newchild.Close();
```

Networking

Performance



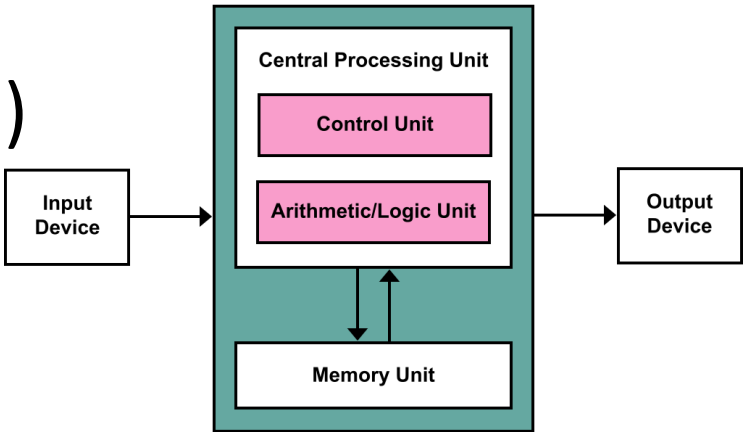
What makes computers programmable?



1. Common Computer Model
2. Abstraction

1. Von Neumann Architecture

- Components
  - Memory (RAM)
  - Central processing unit (CPU)
    - Control unit
    - Arithmetic logic unit (ALU)
  - Input/output system



- Memory stores program and data
- Program instructions execute sequentially

2. 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)**

Art of Program Design

- To take a problem, and continually break it down into a series of smaller ideally concurrent tasks until ultimately these tasks become a series of small specific individual instructions.
- Mindful of the architecture on which the program will run, identify those tasks which can be run concurrently and map those tasks onto the processing units of the target architecture.

What is Programming?

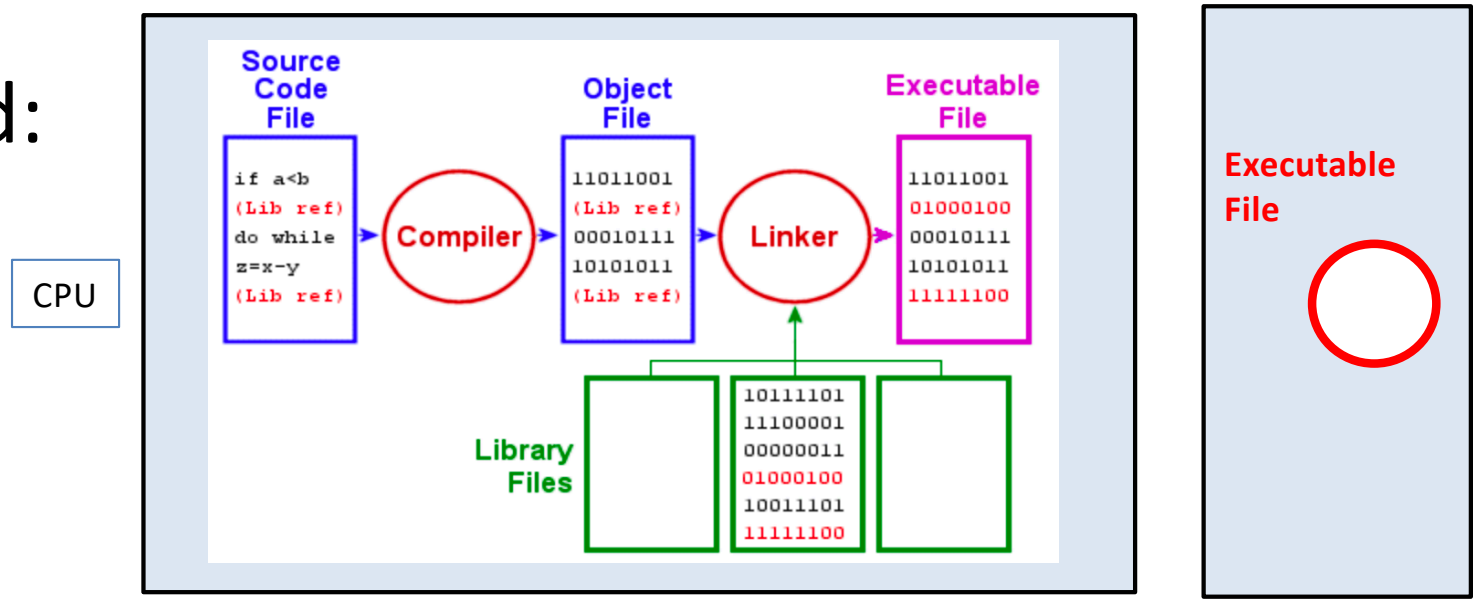
- Writing these instructions as a series of statements.
- A statement uses words, numbers and punctuation to detail the instruction. They are like properly formed sentences in English.
- A poorly formed statement -> compiler error
- Each programming language has a unique “syntax” that defines what constitutes correct statements in that language

What Programming Language?

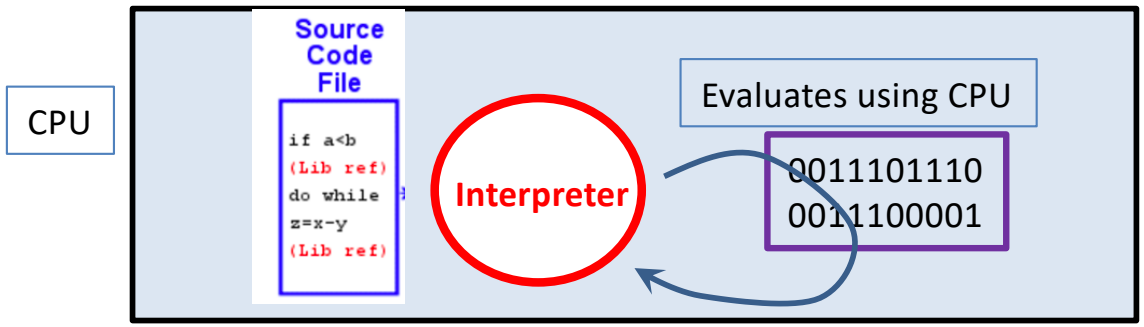
- Hundreds of languages ….
- Only a dozen or so are popular at any time
- We will be looking at C, C++ and Python

Types of Languages - Compiled/Interpreted

- Compiled:

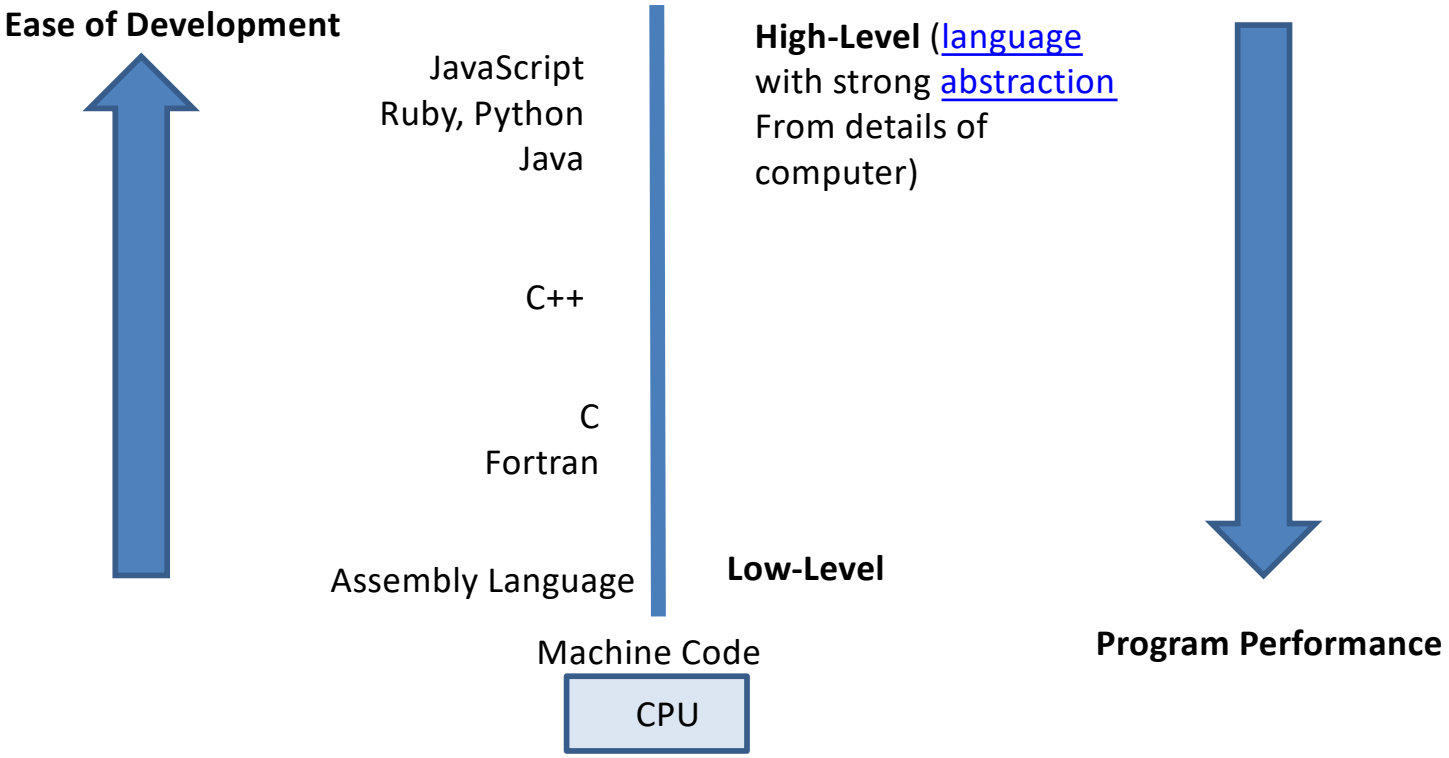


- Interpreted.



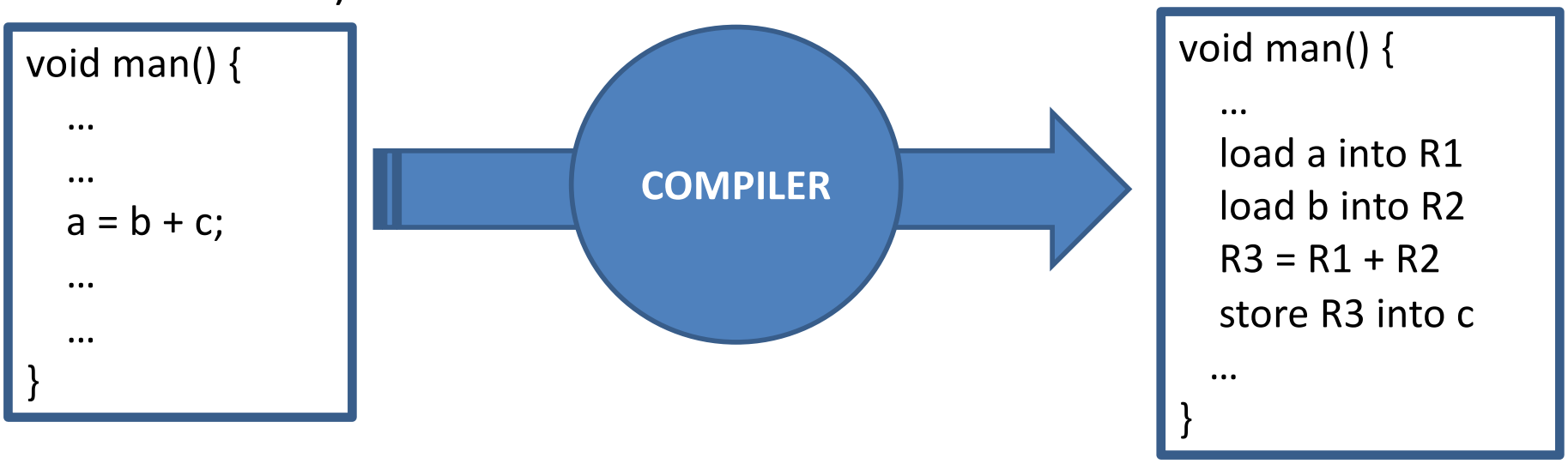
- Hybrid, e.g. Java. Compiler converts to another language, e.g. bytecode. Interpreter runs on machine and interprets this language, e.g. javaVM.

Programming Language Hierarchy

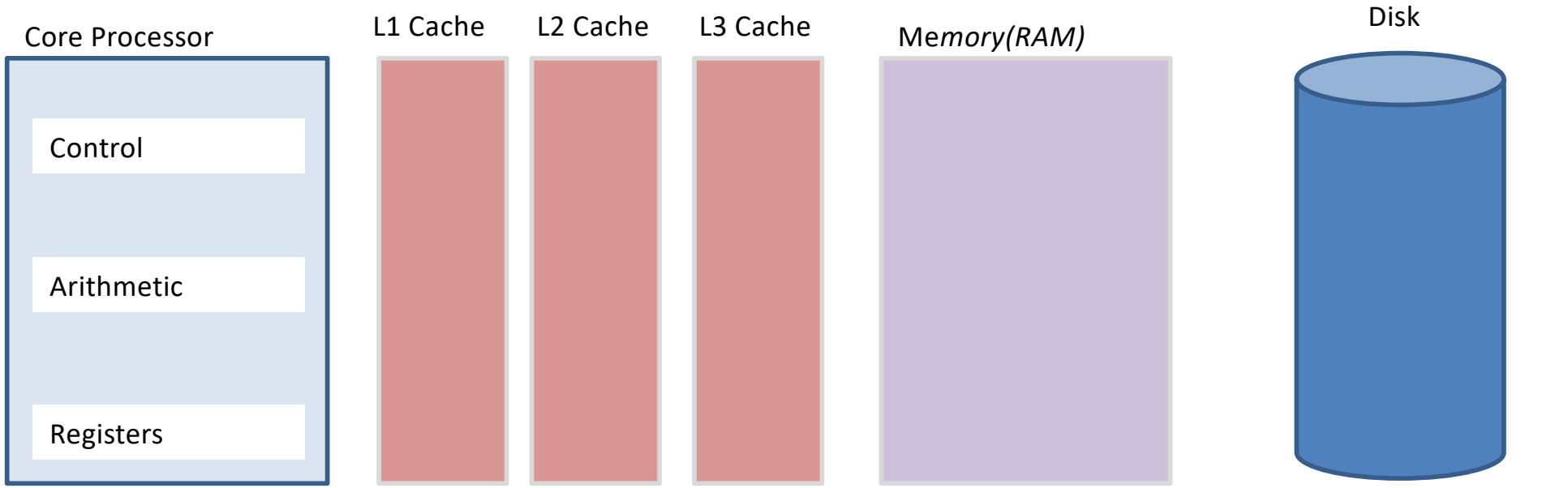


What is a Compiler?

- An application whose purpose is to:
  - Check a Program is legal (follows the syntax)
  - Translate the program into another language (assembly, machine instruction)

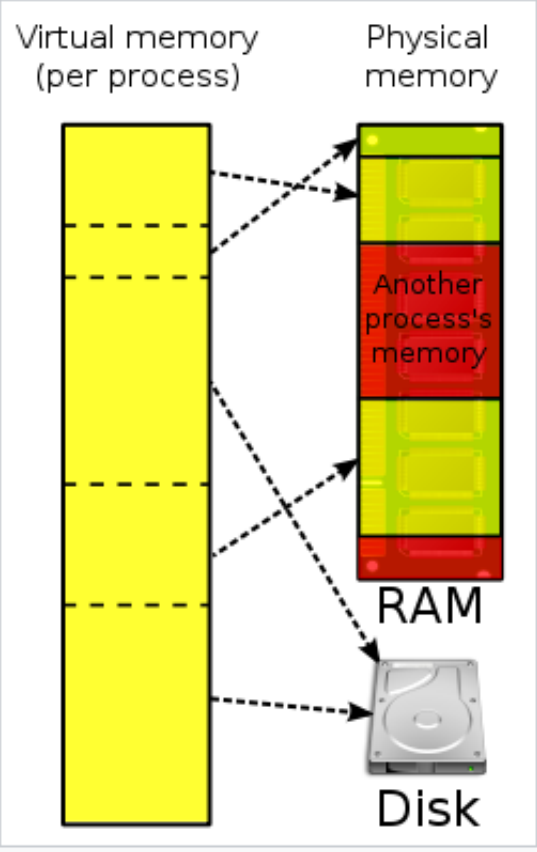


Memory Hierarchy



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

Need for Virtual Memory & Why We Page Fault



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

The C Programming Language

- Originally Developed by Dennis Ritchie at Bell Labs in 1969 to implement the Unix operating system.
- It is a **compiled** language
- It is a **structured** (PROCEDURAL) language
- It is a **strongly typed** language
- The most widely used languages of all time
- It's been #1 or #2 most popular language since mid 80's
  - It works with nearly all systems
  - As close to assembly as you can get
  - Small runtime (embedded devices)

C Program Structure

A C Program consists of the following parts:

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

Everyone's First C Program

no space between # and include

hello1.c

```
#include <stdio.h>

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

statements end with ;

Function that indicates they will return an integer, MUST return an integer

- The first line of the program ***#include <stdio.h>*** is a preprocessor command, which tells a C compiler to include the stdio.h file before starting compilation.
- 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 invoke the ***printf(...)*** function which causes the message "Hello, World!" to be displayed on the screen. The prototype for the function is in the stdio.h file. It's implementation in the standard C library.
- The next statement **return 0;** terminates the main() function and returns the value 0.

Allowable Variable Types in C

char
int
float
double
void
pointers

var3.c

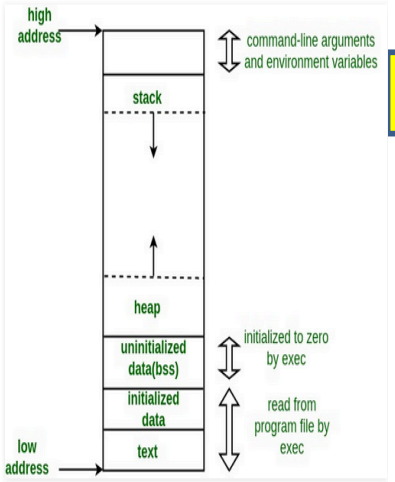
```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv) {
   int    i1 = 5;
   float  f1 = 1.2;
   double d1 = 1.0e6;
   char   c1 = 'A';
   printf("Integer %d, float %f, double %f, char %c \n", i1, f1, d1, c1);
   return 0;
}
```

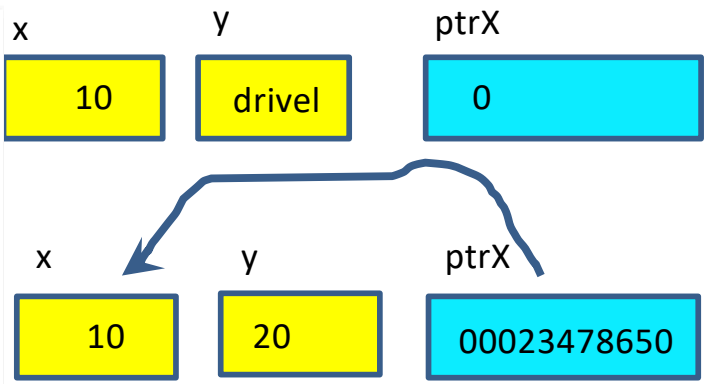
The Dreaded Pointers – not so bad!

- You will use pointers an awful lot if you write any meaningful C code.
- Remember when you declare variables you are telling compiler to set aside some memory to hold a specific type and you refer to that memory when you use the name, e.g. int x. When you specify a pointer, you are seeting aside a mem address.
- The unary **&** gives the "**address**" of an object in memory.
- The unary * in a declaration indicates that the object is a pointer to an object of a specific type
- The unary * elsewhere treats the operand as an address, and depending on which side of operand either sets the contents at that address or fetches the contents.

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

    ptrX = &x;
    y = *ptrX + x;
}
```

pointer1.c






```
void man() {
    …
    load ptrX into
R1
    load R1 into R2
    load x into R3
    R4 = R2 + R3
    store R4 into y
    …
}
```

Arrays - I

A fixed size sequential collection of elements laid out in memory of the *same* type. We access using an index inside a square brackets, indexing start at 0

to declare:

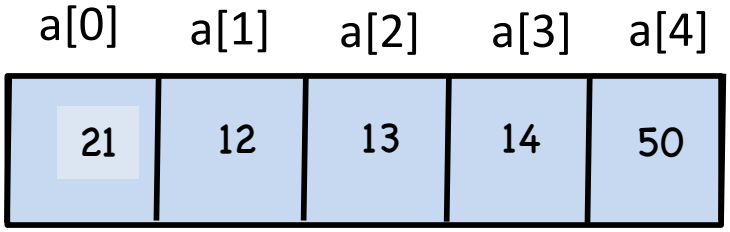
type arrayName [size];

type arrayName [size] = {size comma separated values}

WARNING: indexing starts at 0

array1.c

```
#include <stdio.h>

int main(int argc, char **argv) {
  int intArray[5] = {19, 12, 13, 14, 50};
  intArray[0] = 21;
  int first = intArray[0];
  int last = intArray[4];
  printf("First %d, last %d \n", first, last);
  return 0 ;
 }
```



Arrays - II

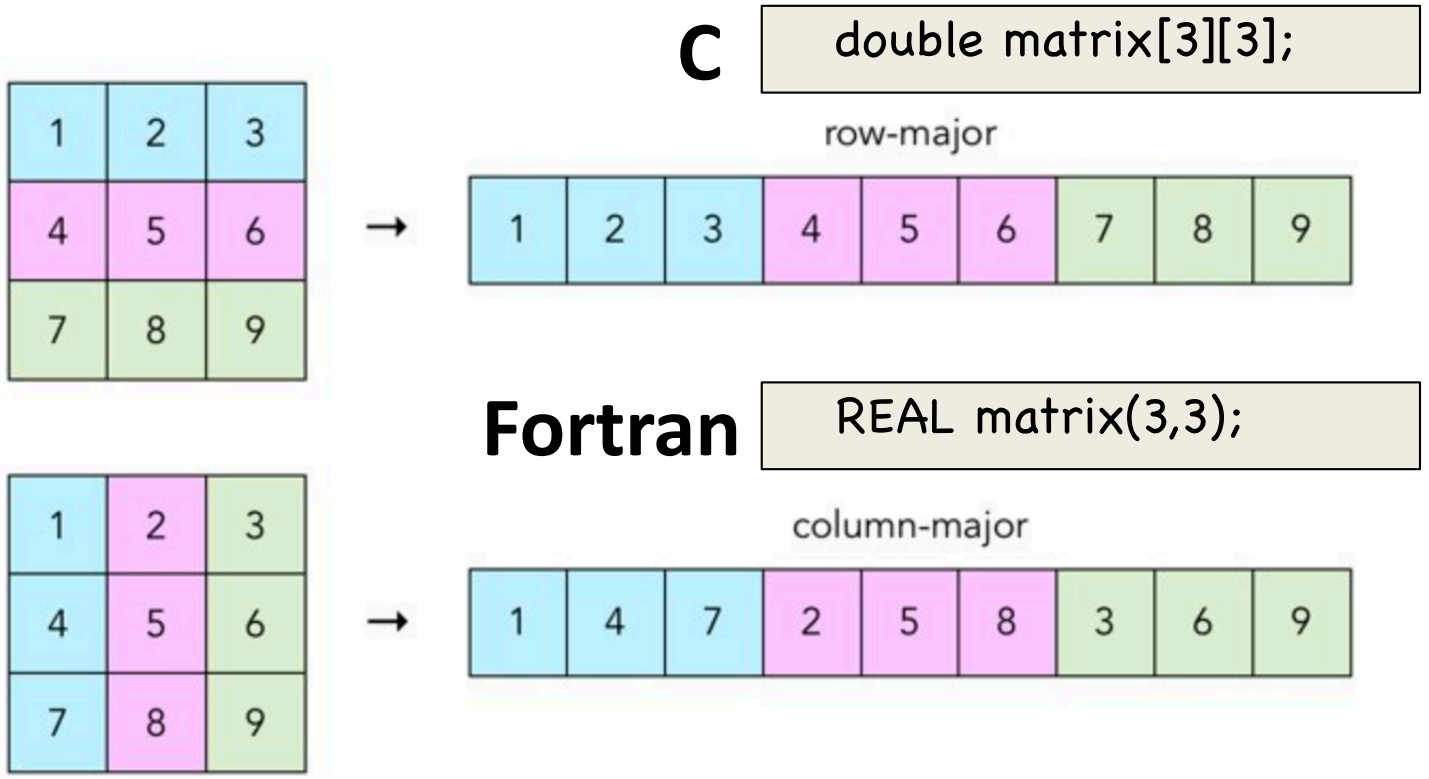
pointer, malloc() and free()

memory1.c

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

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

 // get n
  printf("enter n: ");
  scanf("%d", &n);
  if (n <=0) {printf ("You idiot\n"); return(0);}

  // allocate memory & set the data
  array1 = (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
  free(array1);
  return(0);
}
```

```
[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 >
```

Memory Layout of Arrays in C and Fortran



Operations –

op2.c

```
#include <stdio.h>
int main(int argc, char **argv) {
  int a = 5;
  int b = 2;
  int c = a + b * 2;
  printf("%d + %d * 2 is %d  \n",a,b,c);

  c = a * 2 + b * 2;
  printf("%d * 2 + %d * 2 is %d  \n",a,b,c);

  // use parentheses
  c = ((a * 2) + b ) * 2;
  printf("((%d * 2) + %d ) * 2; is %d  \n",a,b,c);
  return(0);
}
```

What is c? Operator precedence!

USE PARENTHESES

```
c >gcc oper3.c; ./a.out
5 + 2 * 2 is 9
5 * 2 + 2 * 2 is 14
((5 * 2) + 2 ) * 2; is 24
c >
```

If-else

```
if (condition) {
   // code block
} else if (condition) {
   // another code block
} else {
  // and another
}
```

while

```
while (condition) {
   // code block
}
```

if3.c

```
#include <stdio.h>
int main(int argc, char **argv) {
  int a = 15;
  if (a < 10) {
    printf("%d is less than 10 \n", a);
  } else if ( a == 10) {
       printf("%d is equal to 10 \n", a);
  } else {
     printf("%d is greater than 10 \n", a);
   }
  return(0);
}
```

Can have multiple else if in if statement

```
#include <stdio.h>

int main(int argc, char **argv) {
  int intArray[5] = {19, 12, 13, 14, 50};
  int sum = 0, count = 0;
  while (count < 5) {
    sum += intArray[count];
    count++;
  }
  printf("sum is: %d \n", sum);
 }
```

```
for (init; condition; increment) {
   // code block
}
```

for loop – multiple init & increment

for2.c

```
#include <stdio.h>

int main(int argc, char **argv) {
  int intArray[6] = {19, 12, 13, 14, 50, 0};
  int sum = 0;
  for (int i = 0, j=1; i < 5; i+=2, j+=2) {
    sum += intArray[i] + intArray[j];
  }
  printf("sum is: %d \n", sum);
}
```

C Function

```
returnType funcName (funcArgs) {
   codeBlock
}
```

- **returnType** <optional>: what data type the function will return, if no return is specified returnType is **int.** If want function to return nothing the return to specify is **void.**
- **funcName**: the name of the function, you use this name when “invoking” the function in your code.
- **funcArgs**: comma seperated list of args to the function.
- **codeBlock**: contains the statements to be executed when procedure runs. These are only ever run if procedure is called.

main.c

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

myVector.h

```
int sumArray(int *arrayData, int size);
int productArray(int *arayData, int size);
int normArray(int *arrayData, int size);
int dotProduct(int *array1, int *array2, int size);
```

myVector.c

```
// function to evaluate vector sum
//  inputs:
//    data: pointer to integer array
//    size: size of the array
// outputs:
//
// return:
//     integer sum of all values
int sumArray(int *data, int size) {
  int sum = 0;
  for (int i = 0; i < size; i++) {
    sum += data[i];
  }
  return sum;
}
```

Pass By Value, Pass by Reference

- C (unlike some languages) all args are passed by value

to change the function argument in the callers "memory" we can pass pointer to it, i.e it's address in memory.

This is Useful if you want multiple variables changed, or want to return an error code with the function.

function4.c

```
#include <stdio.h>
sumInt(int1, int2, int *sum);
int main() {
    int int1, int2, sum=0;
    printf("Enter first integer: ");
    scanf("%d", &int1);
    printf("Enter second integer: ");
    scanf("%d", &int2);
    sumInt(int1, int2, sum);
    print("%d + %d = %d \n", int1, int2, sum)
  }
void sumInt(int a, int b, int *sum) {
   *sum = a+b;
}
```

Multiple Cores!

What does it mean for Programmers

“The Free Lunch is Over” Herb Sutter

- Up until 2003 programmers had been relying on Hardware to make their programs go faster. No longer. They had to start programming again!
- **Performance now comes from Software**
- **To be fast and utilize the resources, Software must run in parallel, that is it must run on multiple cores at same time.**

Can All Programs Be Made to Run Faster?

- Suppose only part of an application can run in parallel
- Amdahl's law
  - let s be the fraction of work done sequentially, so (1-s) is fraction parallelizable
  - P = number of processors

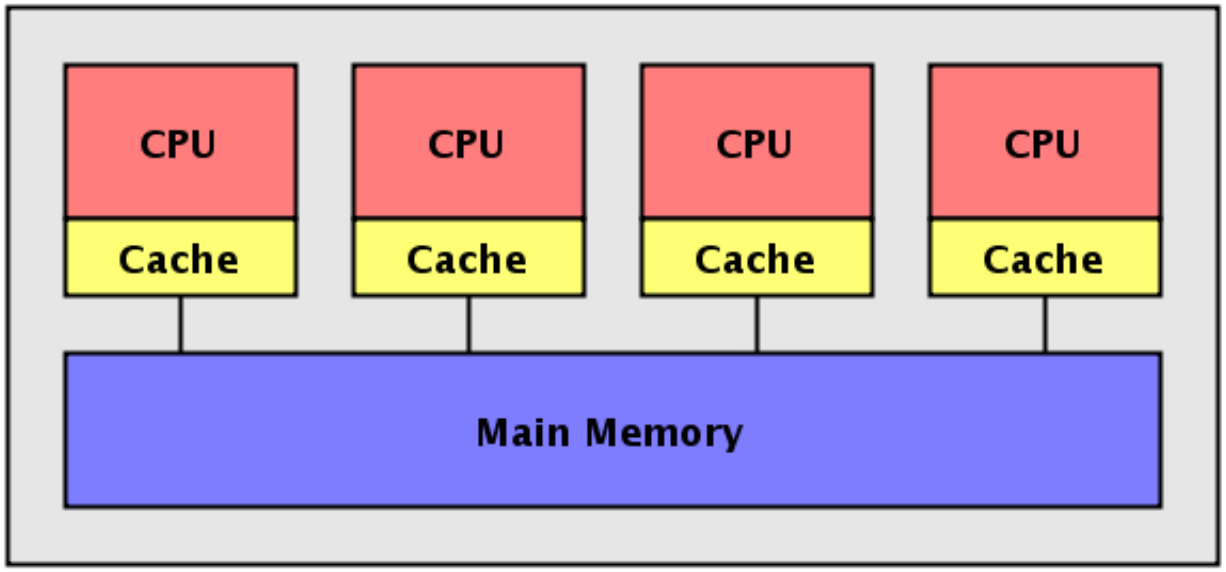
$$\text{Speedup}(P) = \text{Time}(1)/\text{Time}(P)$$

$$\leq 1/(s + (1-s)/P)$$

$$\leq 1/s$$

- **Even if the parallel part speeds up perfectly performance is limited by the sequential part**
- Top500 list: currently fastest machine has P~2.2M; Stampede2 has 367,000

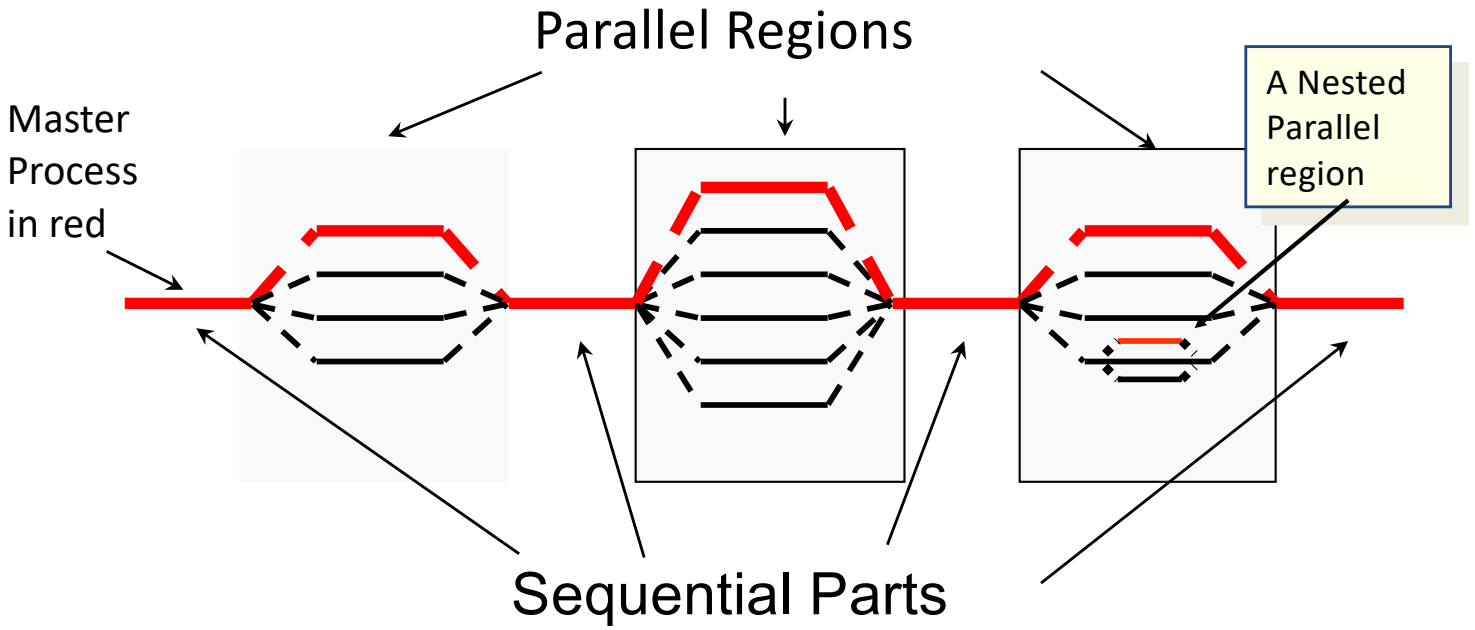
Simplified Parallel Machine Models



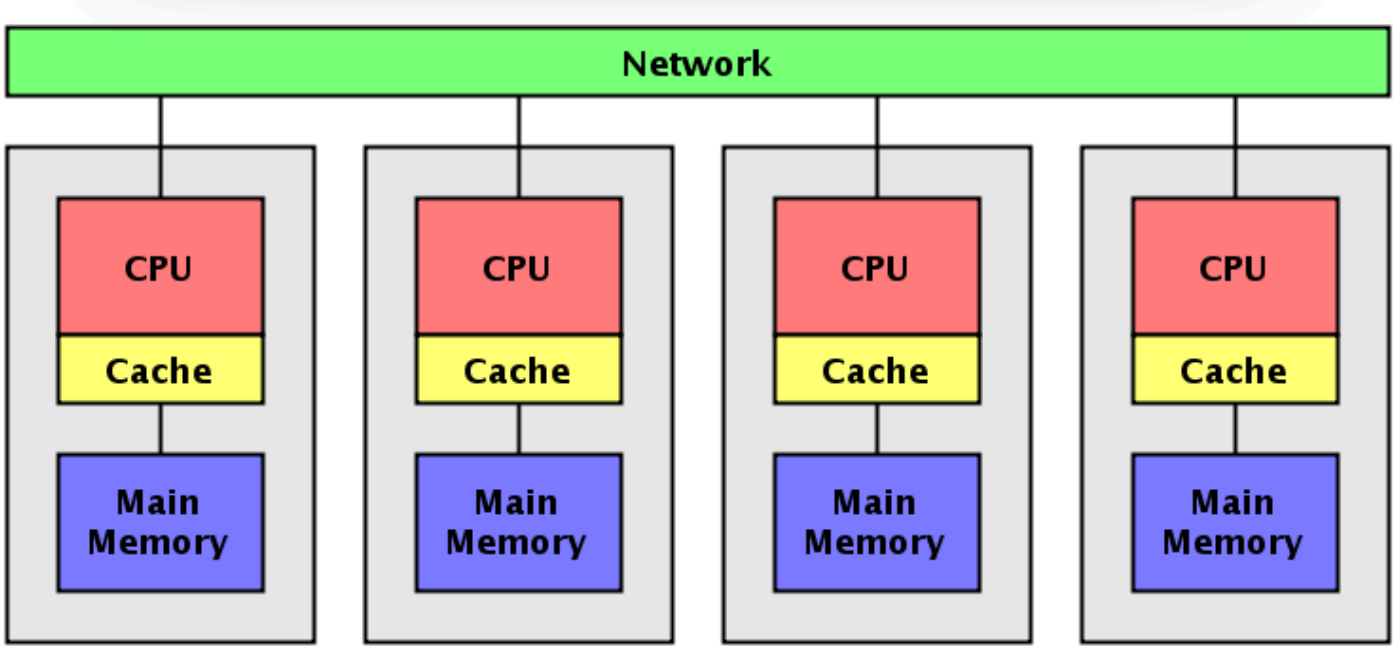
Shared Memory Model

Threads for Shared Memory Model (Posix Threads, OpenMP)

- Master Process spawns a team of threads as needed.
- Parallelism added incrementally until performance goals are met, i.e., the sequential program evolves into a parallel program.



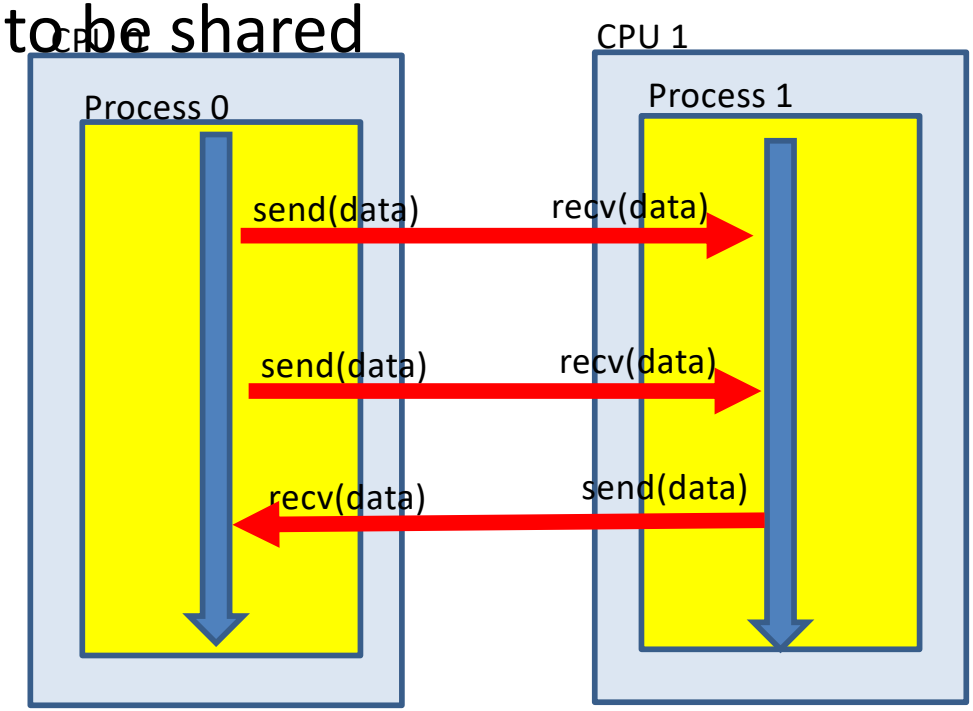
Simplified Parallel Machine Models



Distributed Memory Model

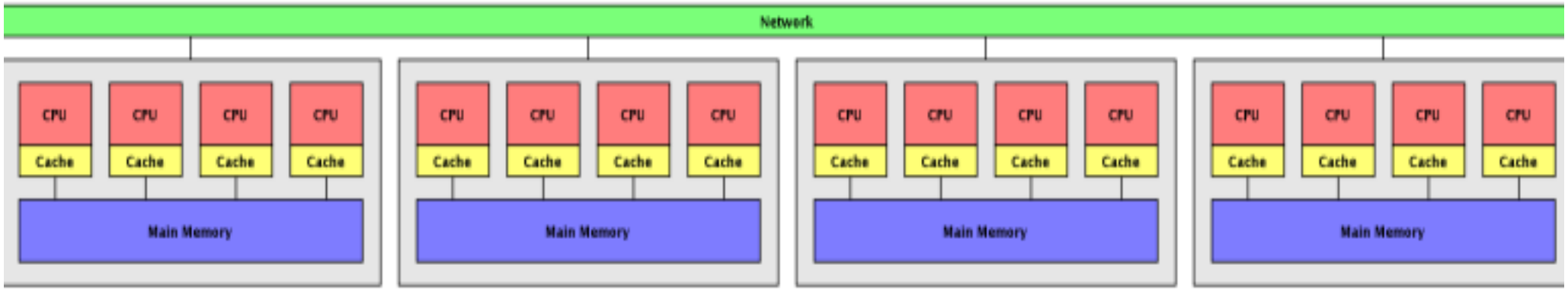
Message Passing for Distributed Memory (MPI API)

- Processes run independently in their own memory space and processes communicate with each other when data needs to be shared



- Basically you write sequential applications with additional function calls to send and recv data.

Simplified Parallel Machine Models



Hybrid Model

Outline

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

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

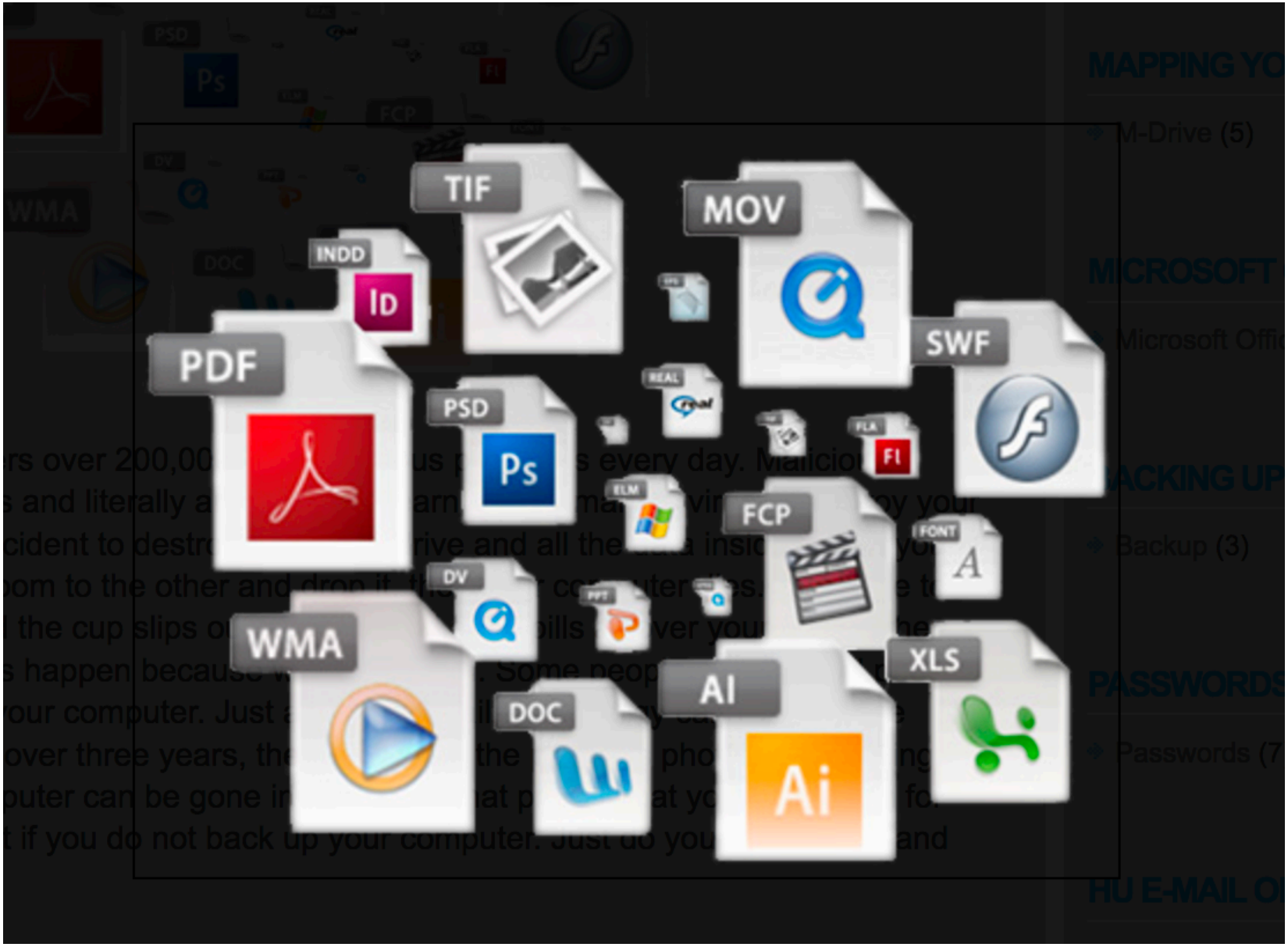
- **Abstraction simplifies many aspects of computing and makes it possible to build complex systems.**

- **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

Computer Bit (on/off) (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

What Might these 7 represent?

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

Allowable Variable Types in C – II qualifiers: unsigned, short, long

1. Integer 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

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

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

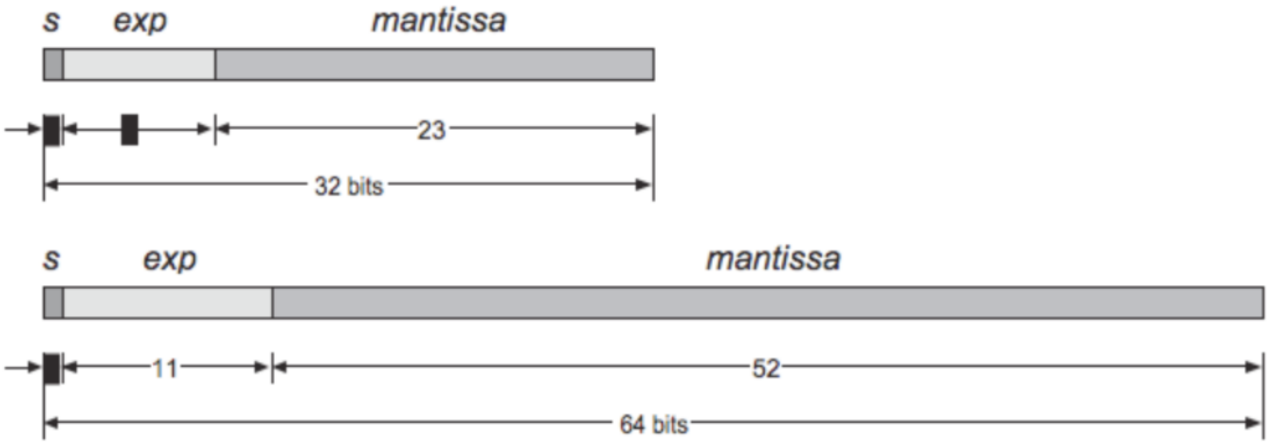
And Here is a Program To Print It

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);
}
```

Float and Double Point Numbers - IEEE 754 standard

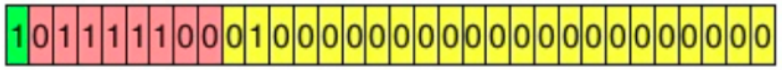
Single Precision



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?

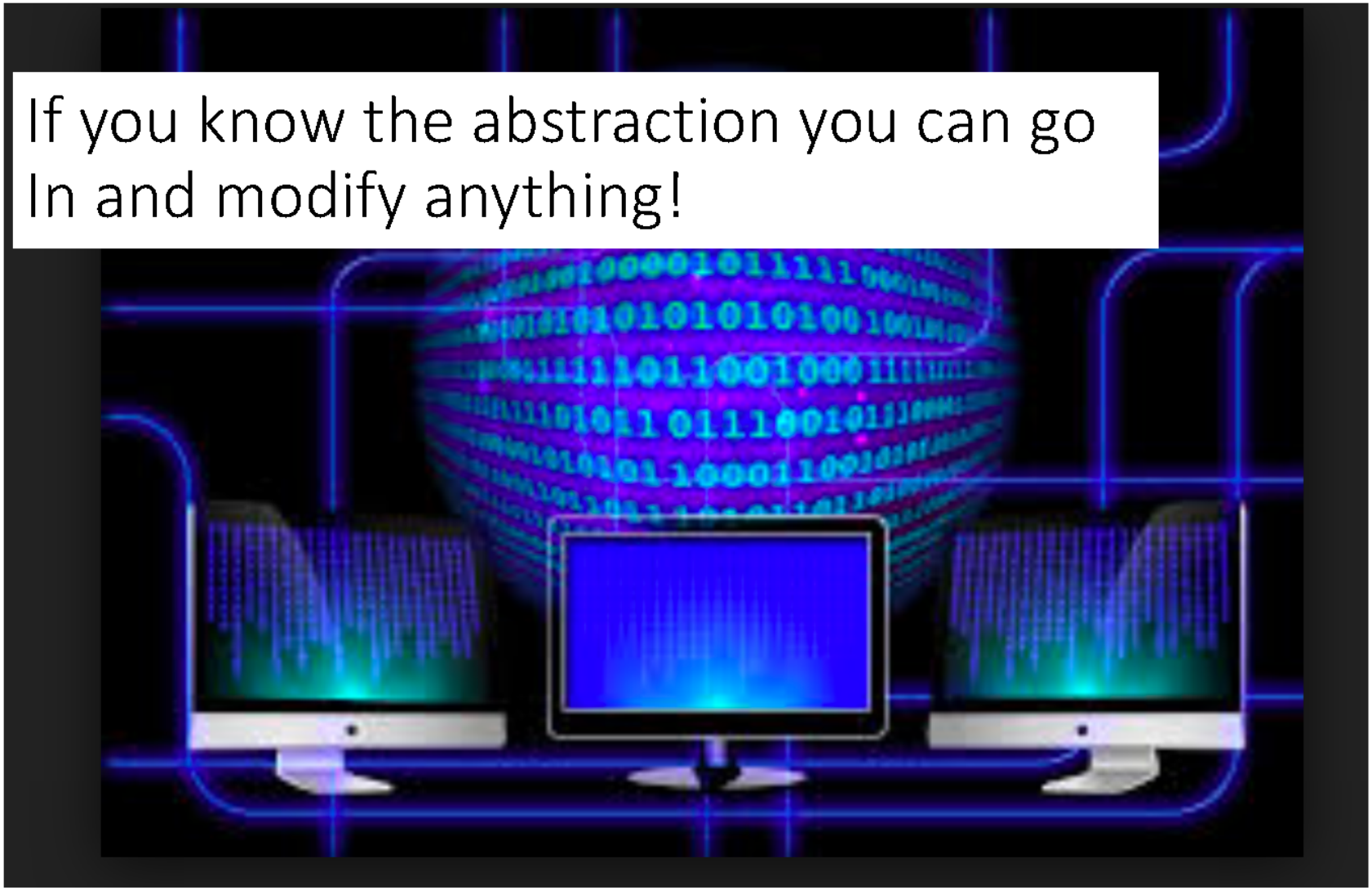


- Solution:
  - Sign = 1 is negative
  - Exponent = $(01111100)_2 = 124$, $E-$bias = $124 - 127 = -3$
  - Significand = $(1.0100 \ldots 0)_2 = 1 + 2^{-2} = 1.25$ (1. is implicit)
  - Value in decimal = $-1.25 \times 2^{-3} = -0.15625$
- What is the decimal value of?

- 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!



Outline

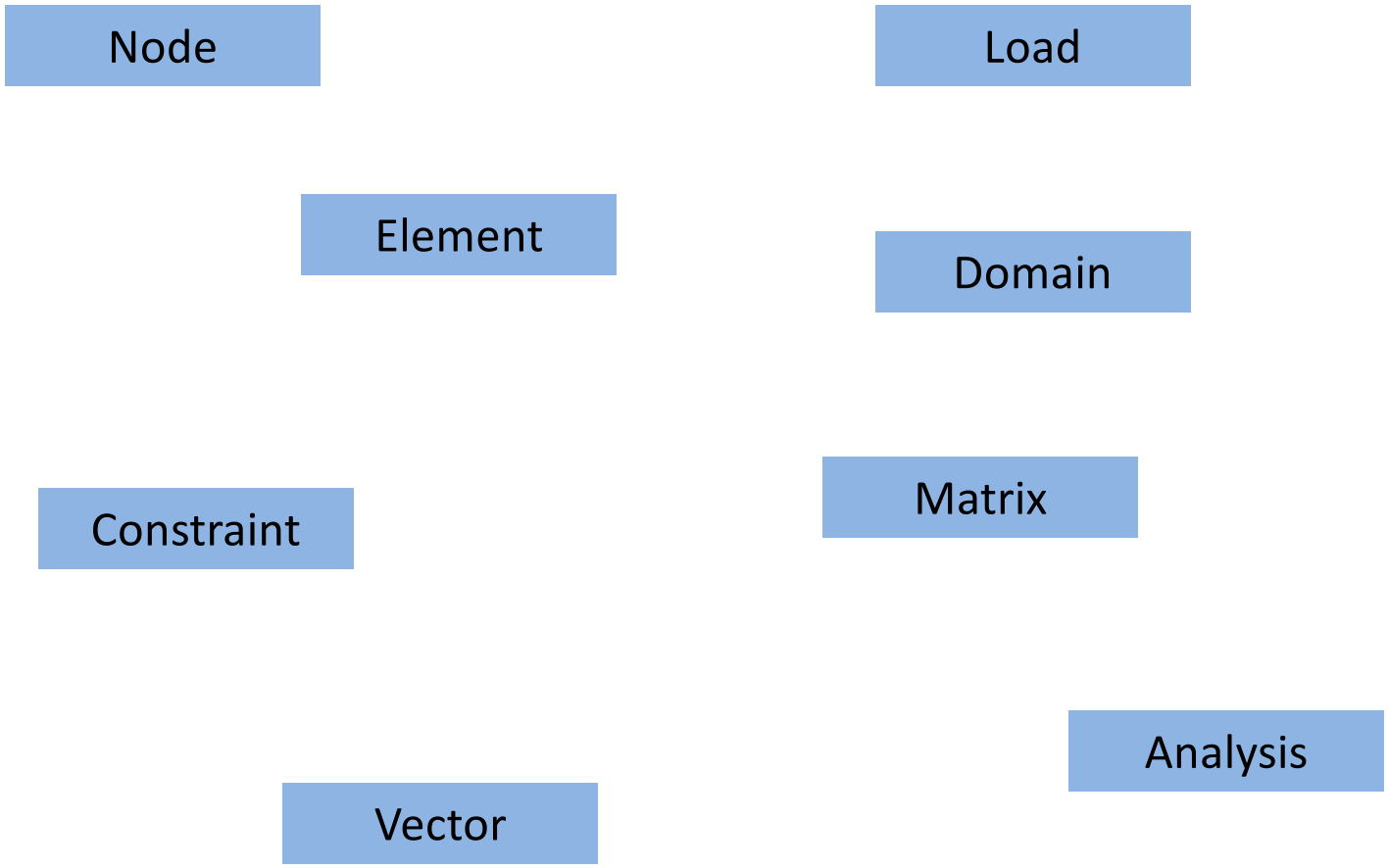
- **Review**
- **Abstraction**
- **C Programming Language Contd.**
  - Structures
  - Containers
- **Object Oriented Programming**
- **C++ Language**

C Structures

- A powerful tool for developing your own data 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 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);
}
```

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, …

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);
```

Array

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);
```

Linked List

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
```

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;
}
```

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;
}
```

Exercise: Add constraint to the fem example

1. Create constraint.h
2. Create constraint.c
3. Modify domain.c to handle constraints
4. Modify main.c to add nodes and constraints
5. Compile & Execute

Constraint:

Some tag, some node tag, for each degree-of-freedom some bool flag indicating whether free or constrained

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!

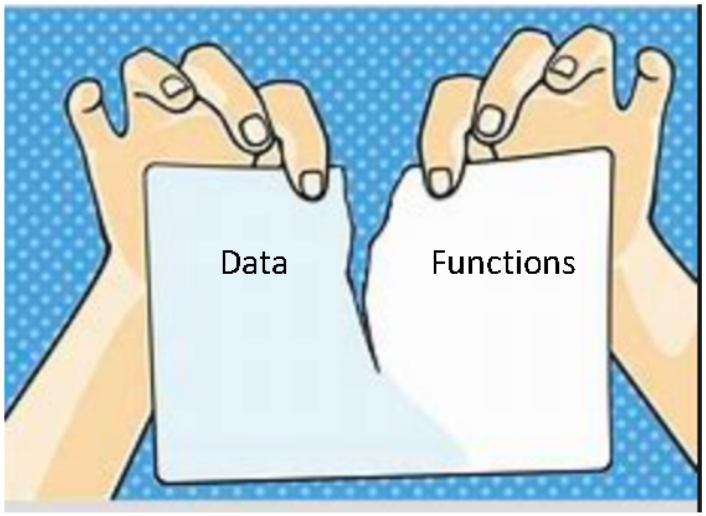
```
typedef struct struct_domain {
    Node *theNodes;
    Constraints *theConstraints;
    Beam*theBeams;
    Truss *theTrusses;
}
```

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

Outline

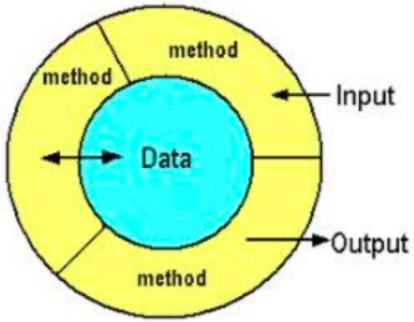
- **Review**
- **Abstraction**
- **C Programming Language Contd.**
  - Structures
  - Containers
- **Object Oriented Programming**
- **C++ Language**

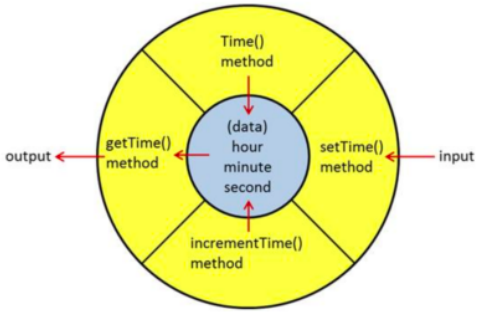
Problem With C is Certain Data & Functions Separate so need these function pointers



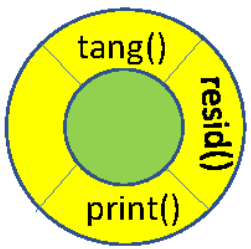
Object-Oriented Programming Offers a Solution

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.

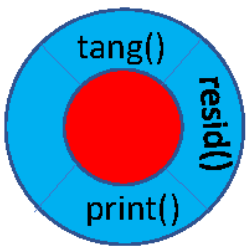




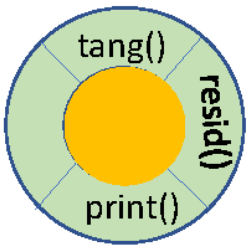
Object-Oriented Programs all provide the ability of one class to inherit the behaviour of a parent class (or even multiple parent classes). This allows the Beam and Trusses both to be treated just as elements. The are said to be polymorphic.



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

- **Review**
- **Abstraction**
- **C Programming Language Contd.**
  - Structures
  - Containers
- **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];
  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;
}
```

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;
}
```

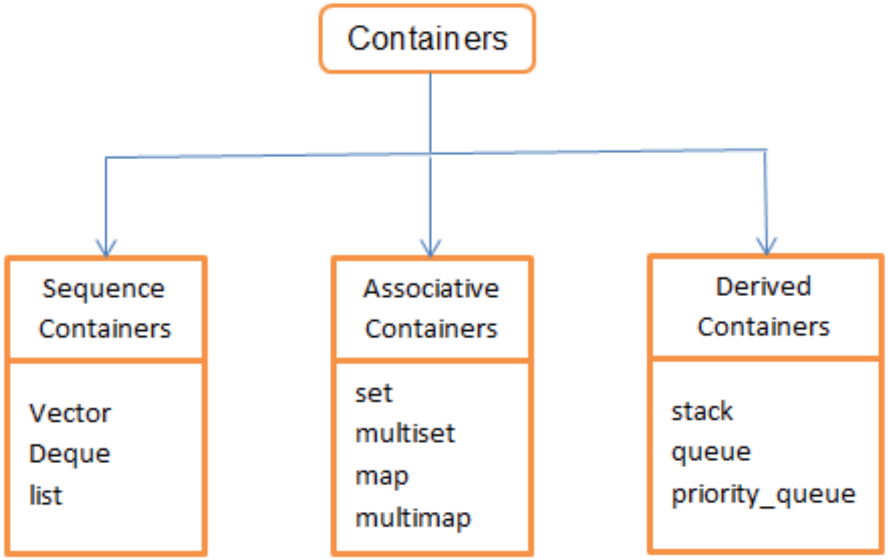
Pass by reference

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";
}
// c by value
void sum1(int a, int b, int *c) {
  *c = a+b;
}


// c by ref
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.

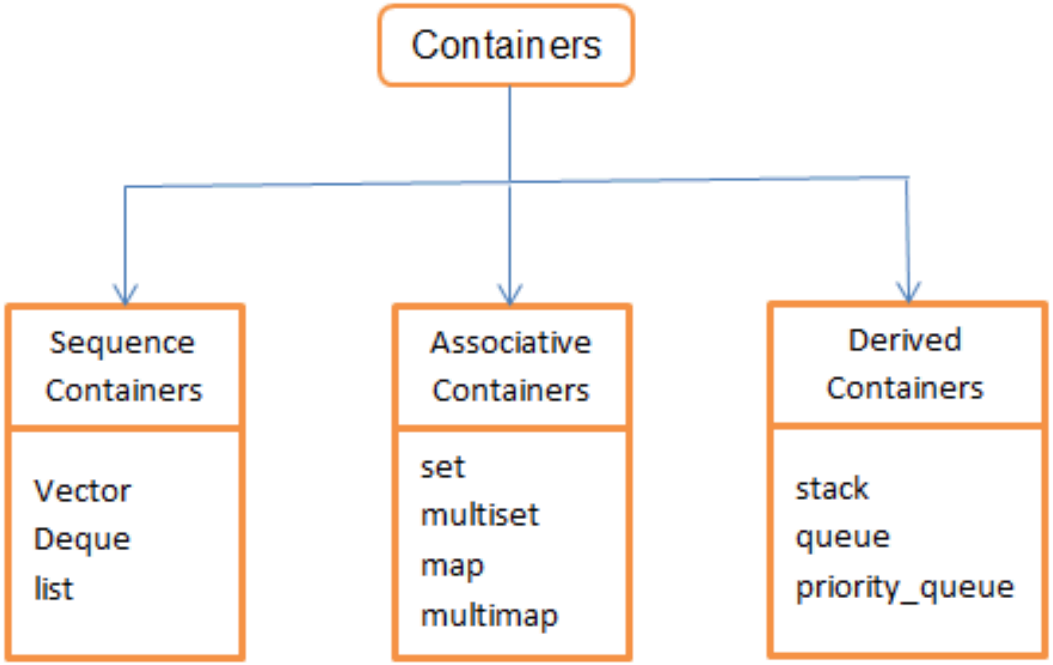
s1 is a variable of type Circle. To invoke methods on this object we use the **DOT .**

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

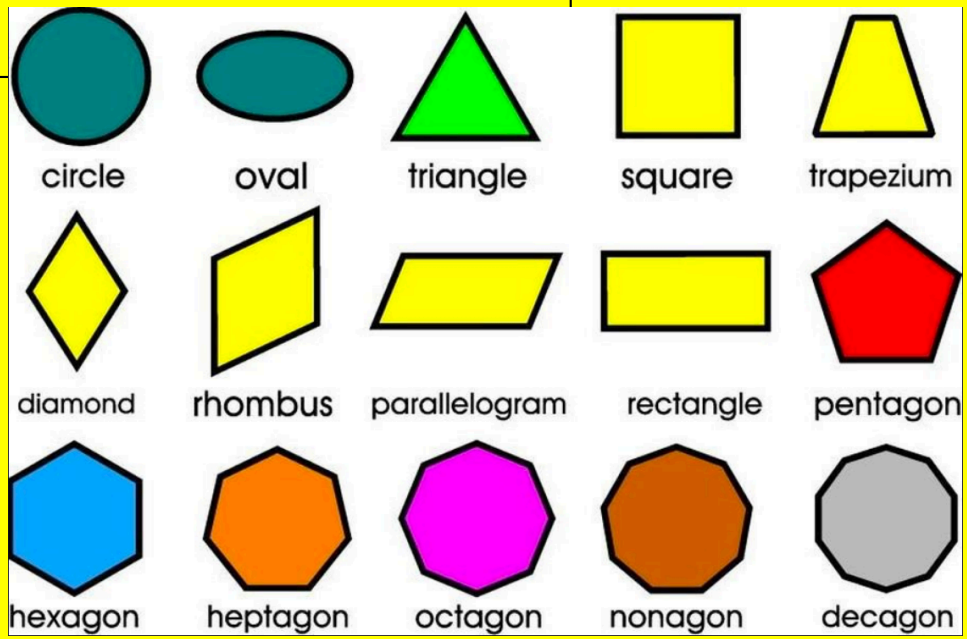
```
#include "Rectangle.h"
#include "Circle.h"
#include <list>
int main(int argc, char **argv) {
  std::list<Shape*> theShapes;

  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

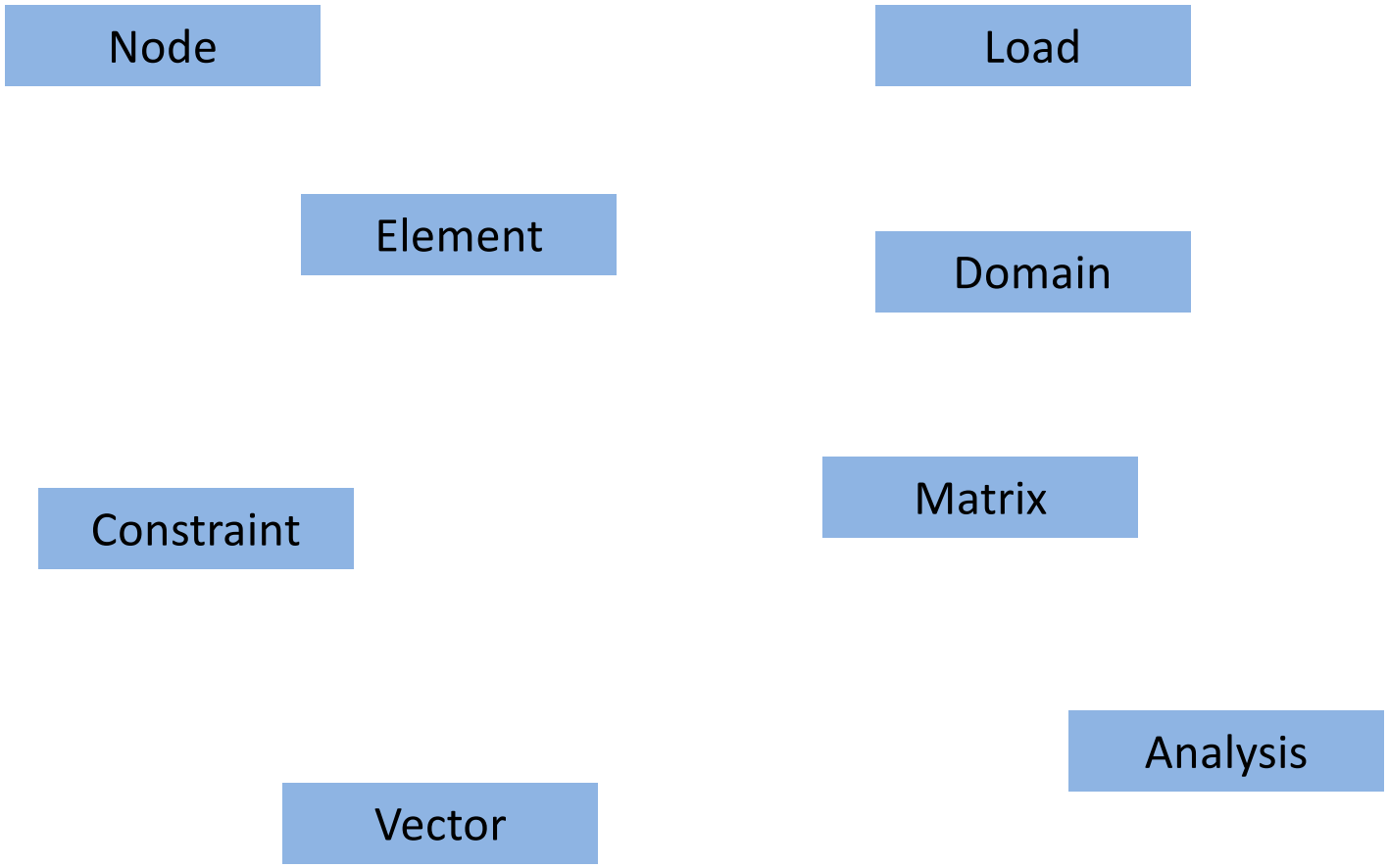
```
#include "Rectangle.h"
#include "Circle.h"
#include <list>
#include <vector>
typedef std::list<Shape*> Container;
//typedef std::vector<Shape*> Container;
typedef Container::iterator Iter;

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++;
    }
```

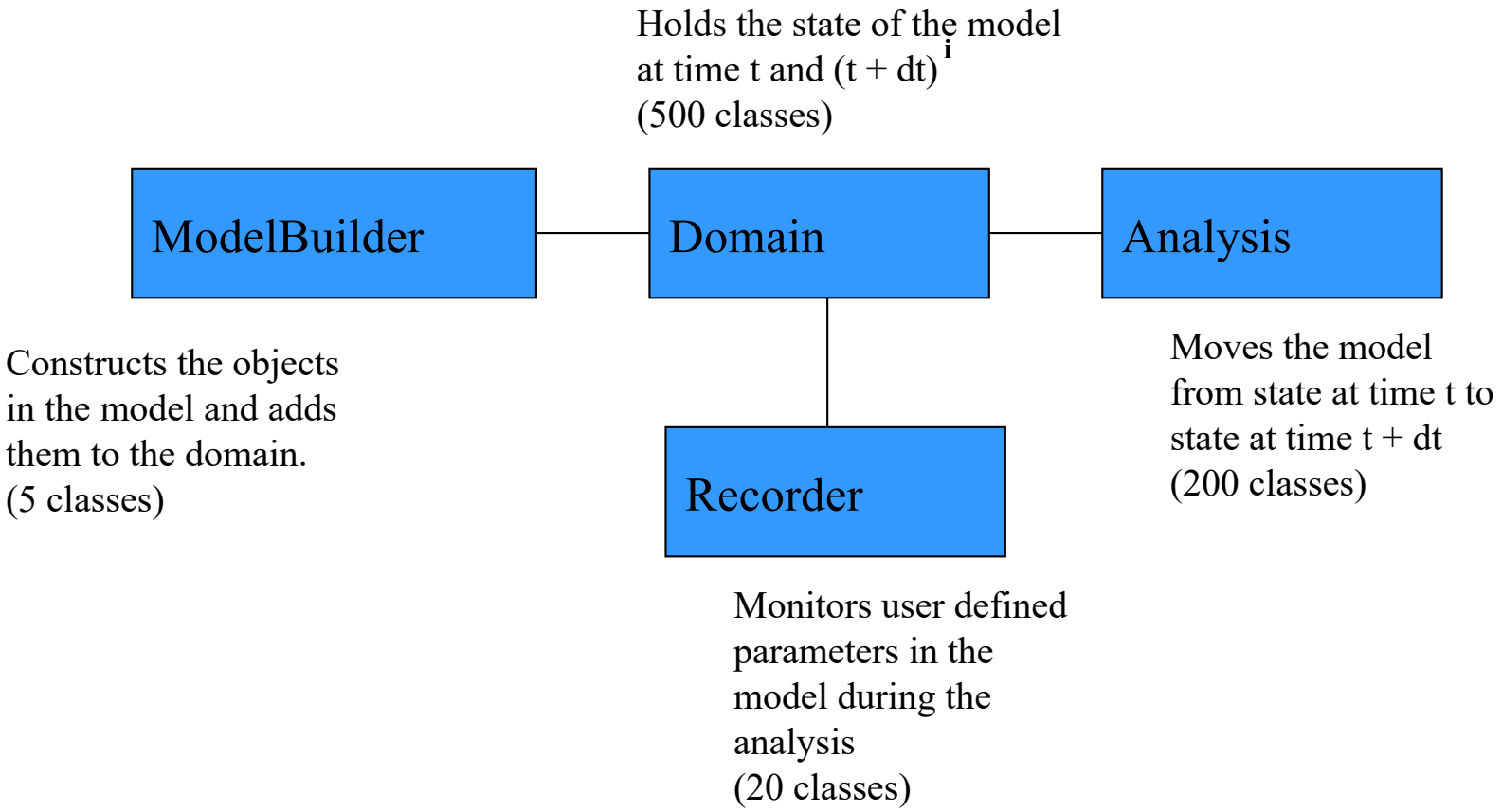
Exercise: Add constraint to the fem example

1. Create Constraint.h
2. Create Constraint.c
3. Modify Domain.c to handle constraints
4. Modify main.c to add nodes and constraints
5. Compile & Execute

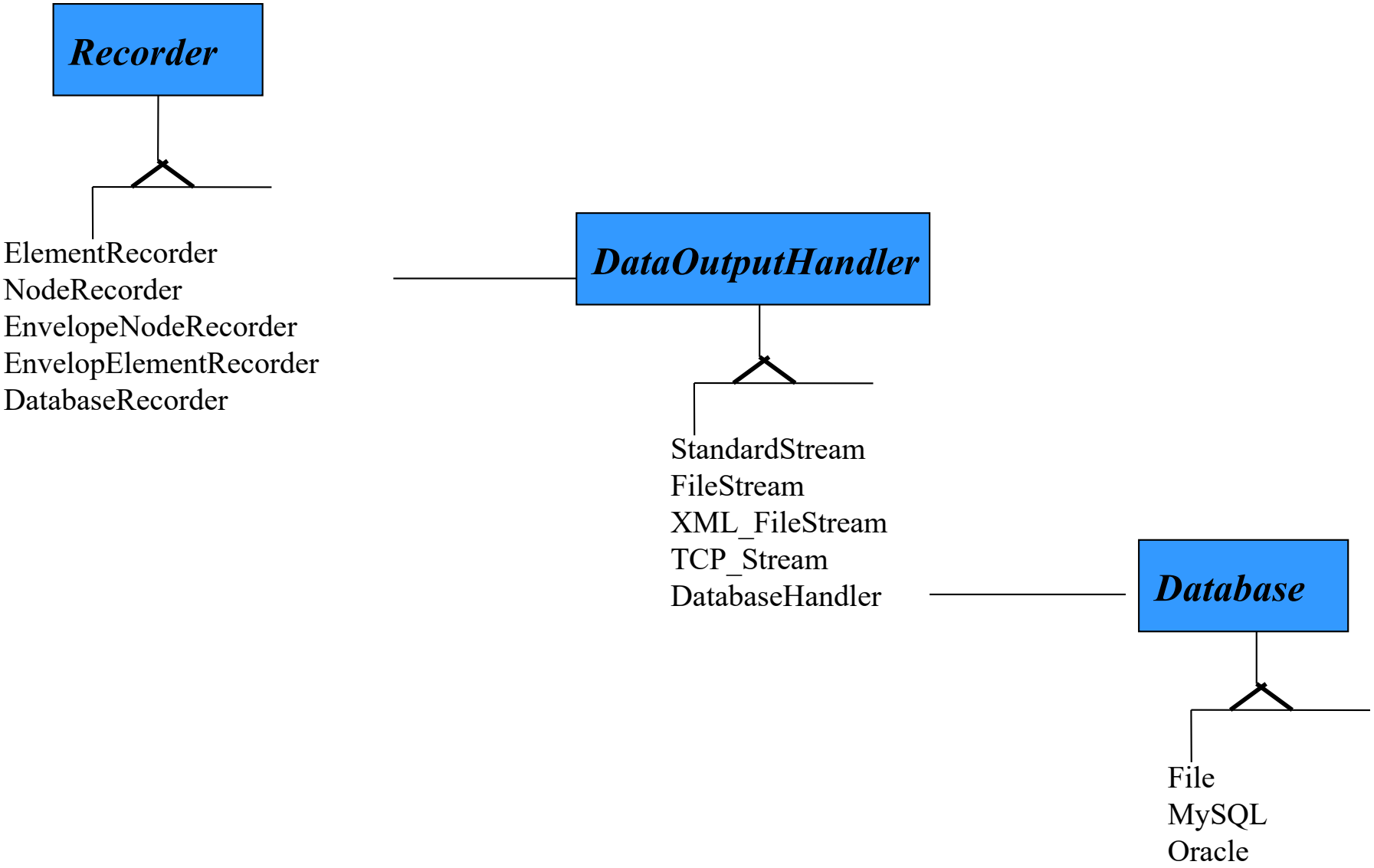
Constraint:

Some tag, some node tag, for each degree-of-freedom some bool flag indicating whether free or constrained

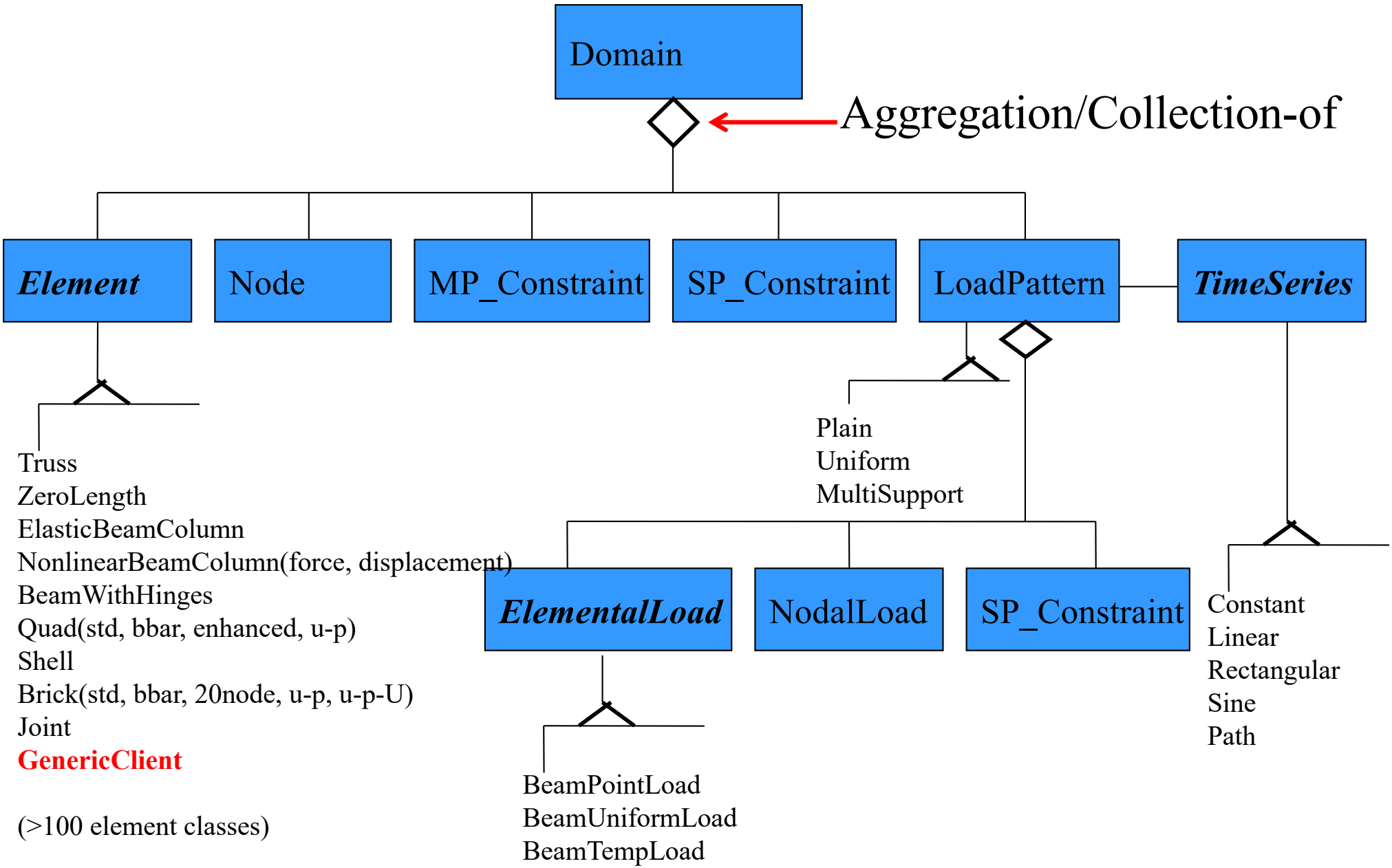
Main Abstractions in OpenSees Framework as an Example of OOP Design



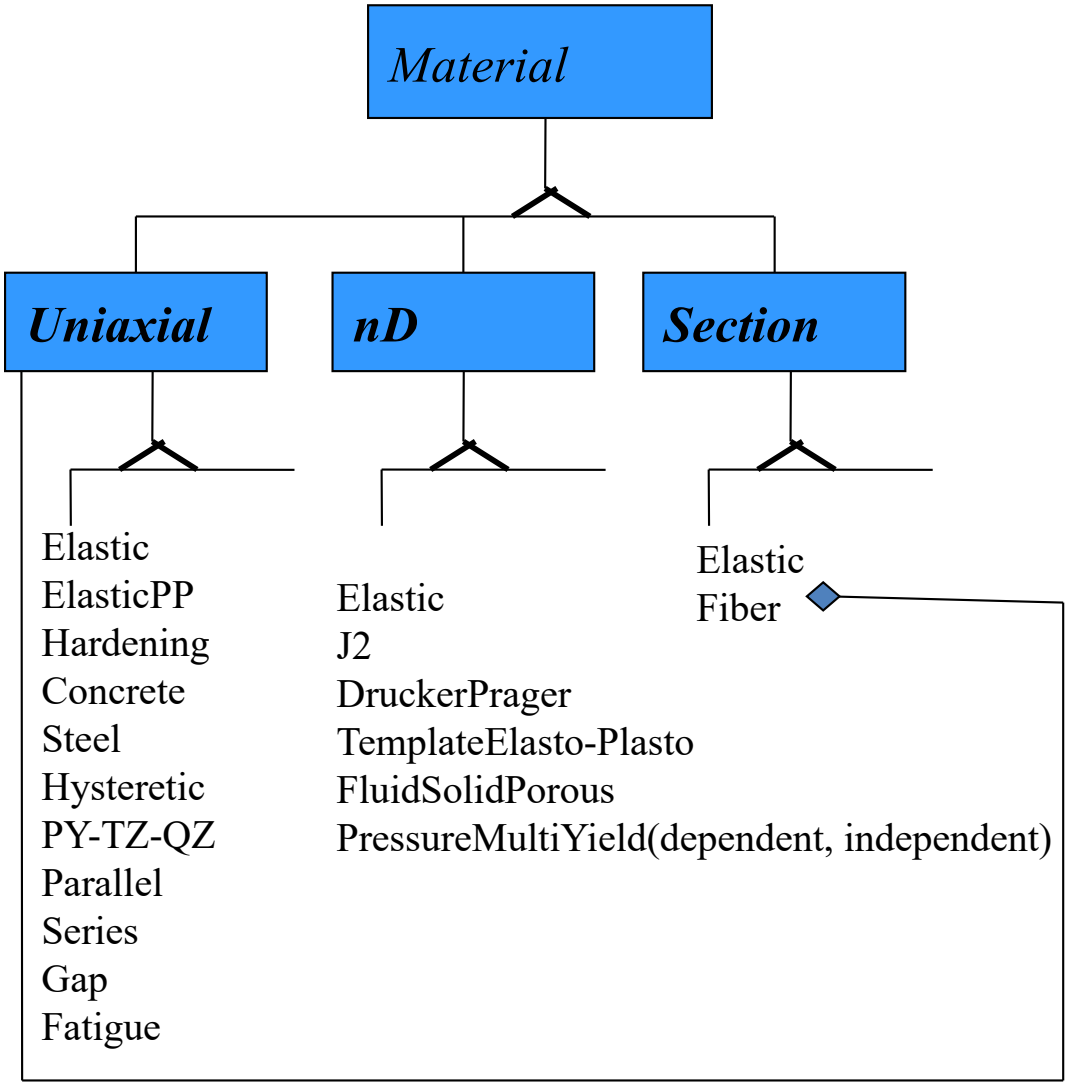
Recorder Options



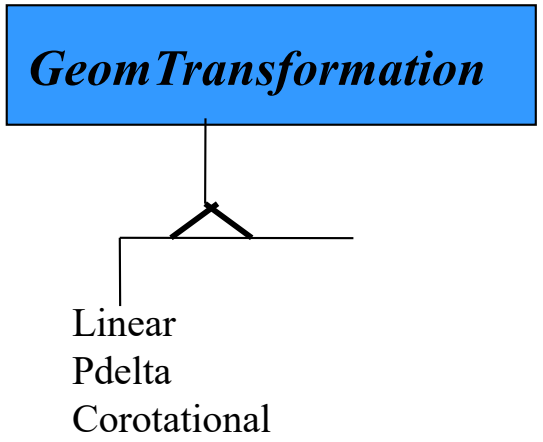
What is in a Domain?



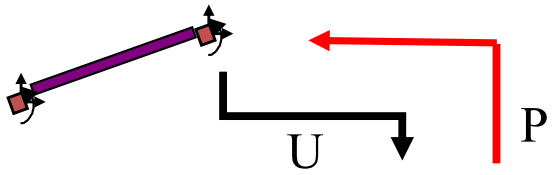
Some Other Classes associated with Elements:



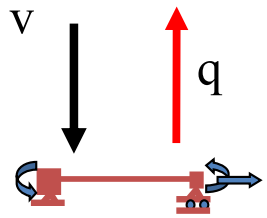
(over 250 material classes)



Element in Global System



Geometric Transformation



Element in Basic System

What is an Analysis?