



Center for Computational Modeling and Simulation

## Programming Bootcamp

# C/C++ & HPC Programming Bootcamp

Frank McKenna  
University of California at Berkeley



NSF award: CMMI 1612843

# Todays Outline

- Expectations
- Daily Task For You
- NHERI SimCenter
- C Day 1
  - Computers & Programs
  - Introduction to C
  - Variables
  - Operations
  - Exercise
  - Function
  - Exercise



## Programming Bootcamp

# C/C++/ HPC: Expectations

Frank McKenna  
University of California at Berkeley



NSF award: CMMI 1612843

- This is a breadth first approach  
we cover a lot of things in minor detail
- We follow reverse lecture style  
**Watch videos by yourself**  
Work with others on homework during live session
- To get most out of course **YOU NEED TO PROGRAM**  
Come to office hours if have problems (8.15 am (PDT))  
No question is stupid  
Mistakes are ok  
Contribute to discussions



## Programming Bootcamp

# C/C++/ HPC: Daily Task

Frank McKenna  
University of California at Berkeley



NSF award: CMMI 1612843

- We work with Git & GitHub
- I will be updating the “upstream” repo daily
- To update your repo, you need to do the following daily:

```
git fetch upstream  
git merge upstream/master
```



Center for Computational Modeling and Simulation

## Programming Bootcamp

# NHERI SimCenter

Frank McKenna  
University of California at Berkeley



NSF award: CMMI 1612843



Natural Hazards  
Engineering  
Research  
Infrastructure

## NSF's Facilities/Programs



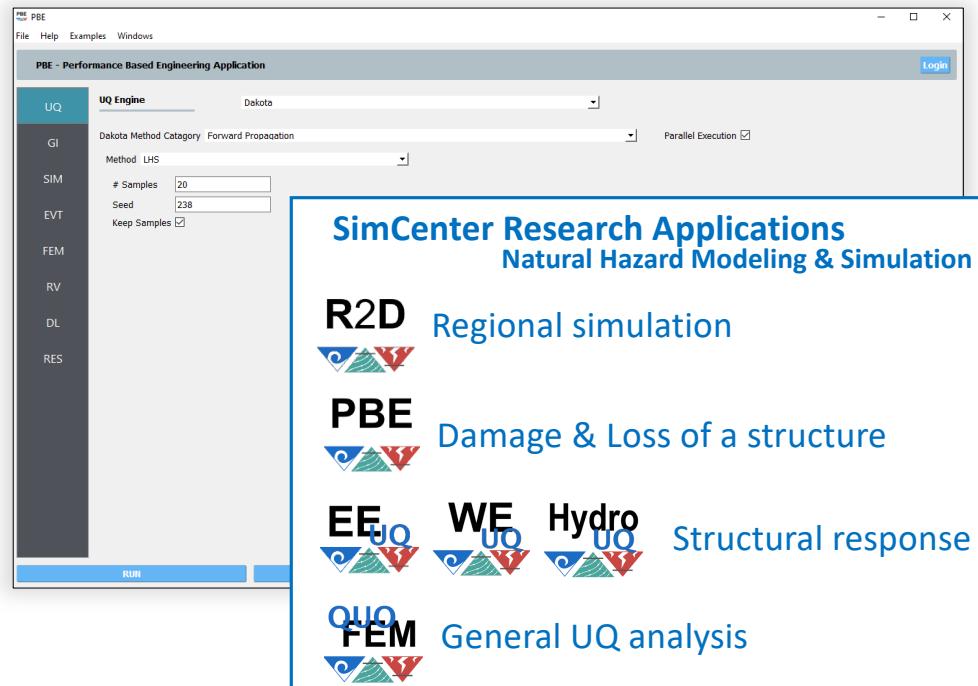
For more information, visit the  
NHERI DesignSafe website: [DesignSafe-ci.org](http://DesignSafe-ci.org)

# SimCenter

Develop software applications for researchers in **Natural hazard engineering**



Researchers  
Industry  
Government Agencies



The screenshot shows the PBE application window. The menu bar includes File, Help, Examples, Windows, and a Login button. The main title is "PBE - Performance Based Engineering Application". On the left is a vertical toolbar with buttons for UQ, GI, SIM, EVT, FEM, RV, DL, and RES. The central panel shows "UQ Engine" selected, with "Dakota" as the engine. Under "Dakota Method Category" is "Forward Propagation". Under "Method: LHS" are fields for "# Samples" (set to 20) and "Seed" (set to 238), with a checked "Keep Samples" option. A "Parallel Execution" checkbox is also present. A "RUN" button is at the bottom of the central panel. To the right of the application window is a box containing the following text:

**SimCenter Research Applications**  
Natural Hazard Modeling & Simulation

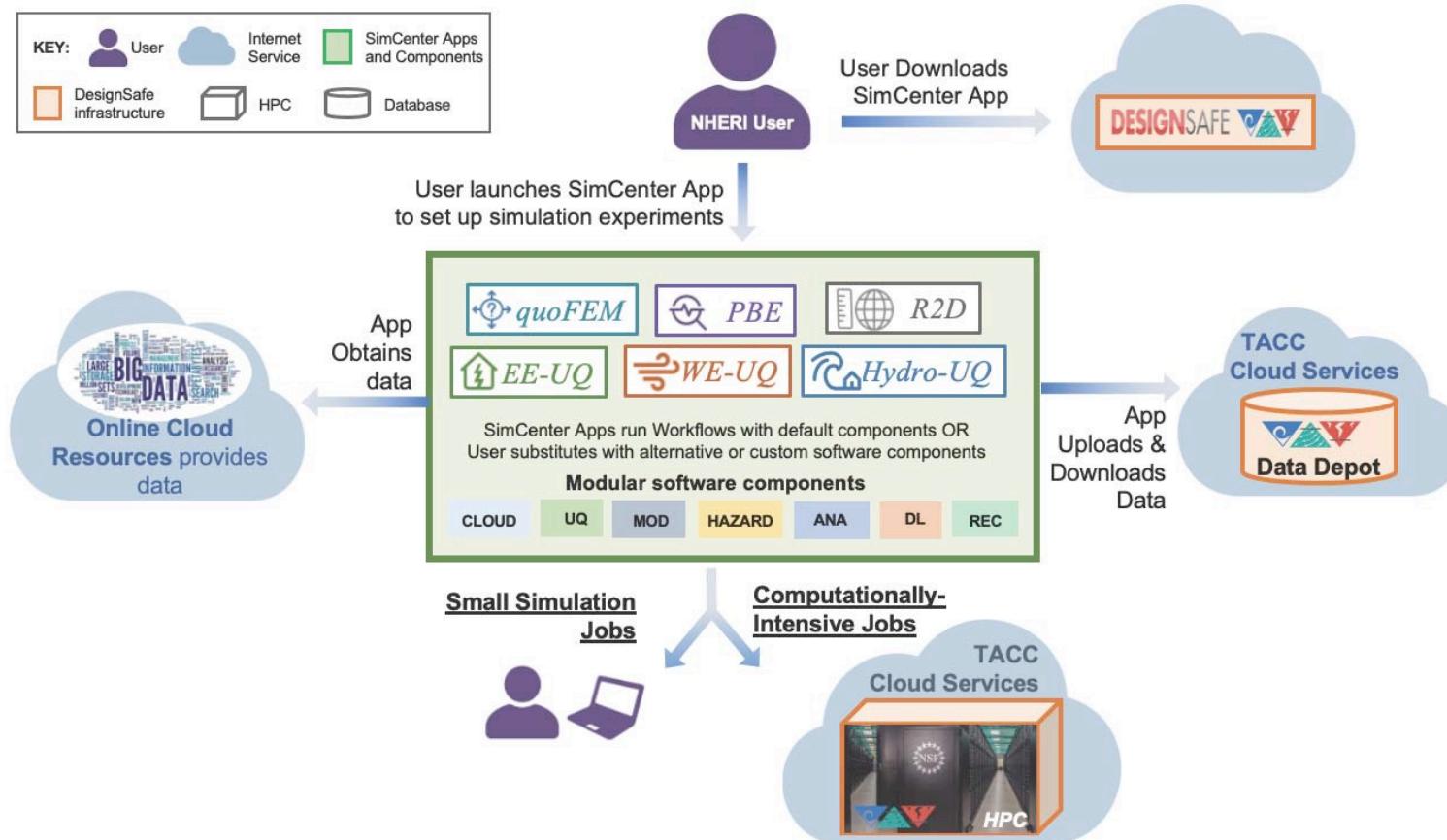
**R2D** Regional simulation  


**PBE** Damage & Loss of a structure  

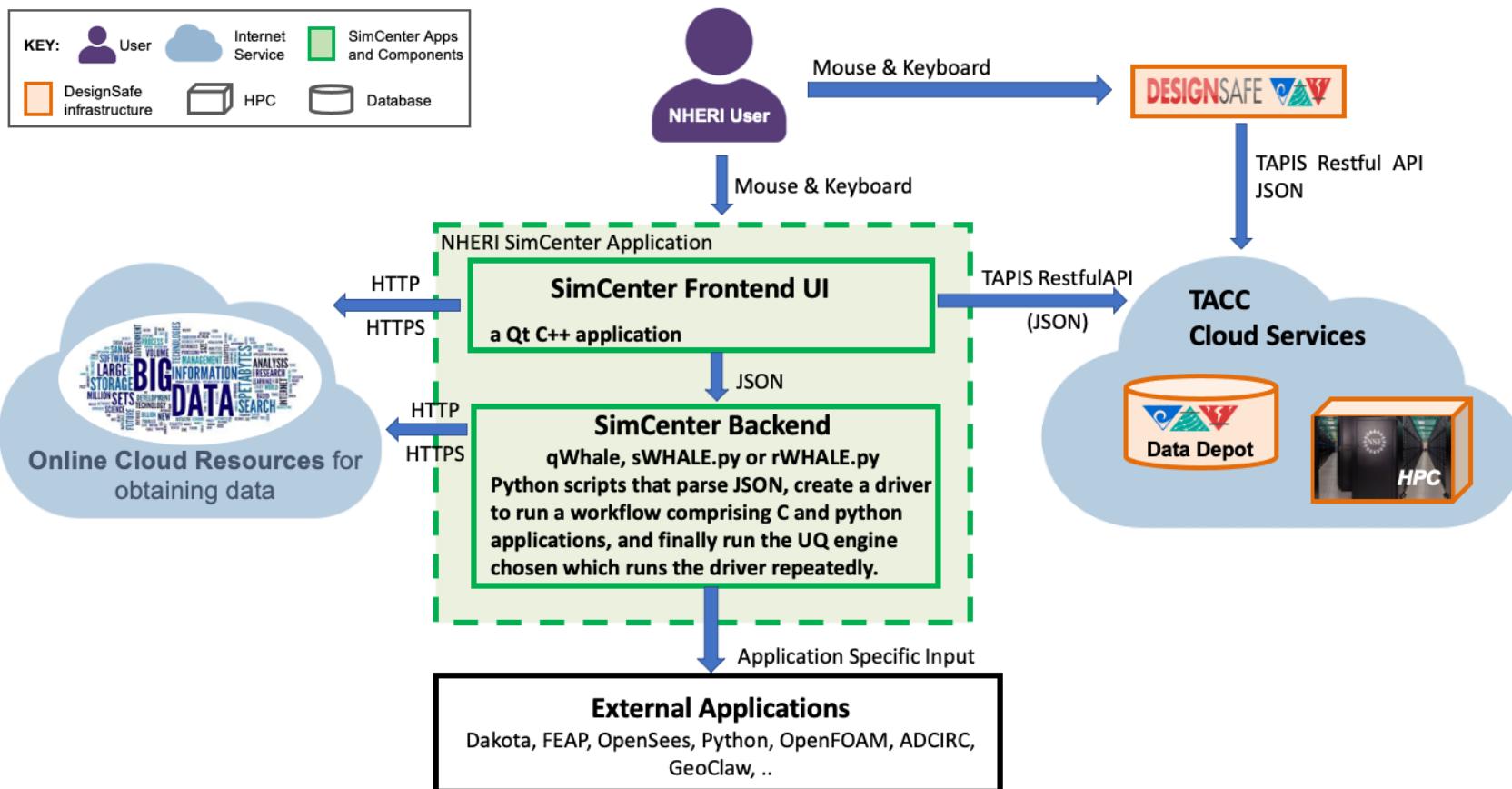

**EE**  **WE**  **Hydro**  Structural response

**QUO**  **FEM** General UQ analysis  

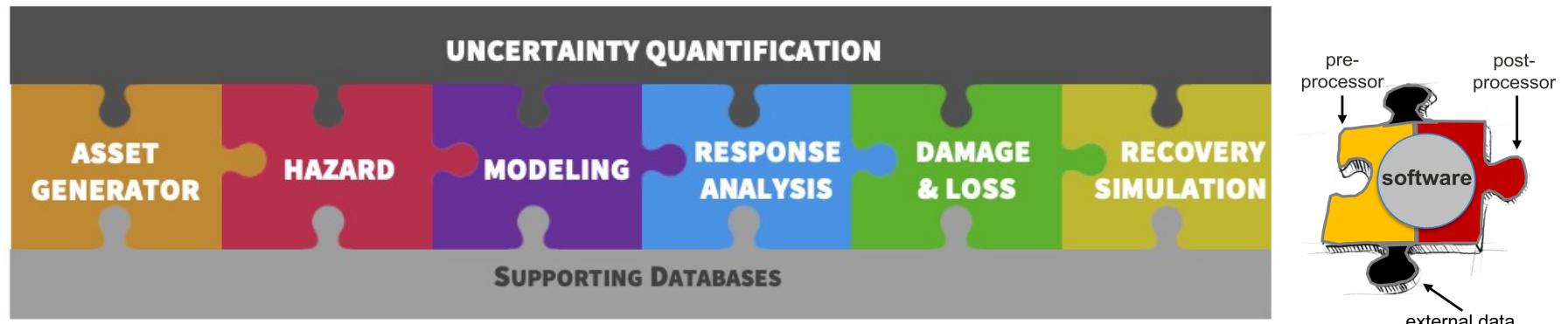

# Applications Available From and Utilize DesignSafe



# UI Launches Scientific Workflows

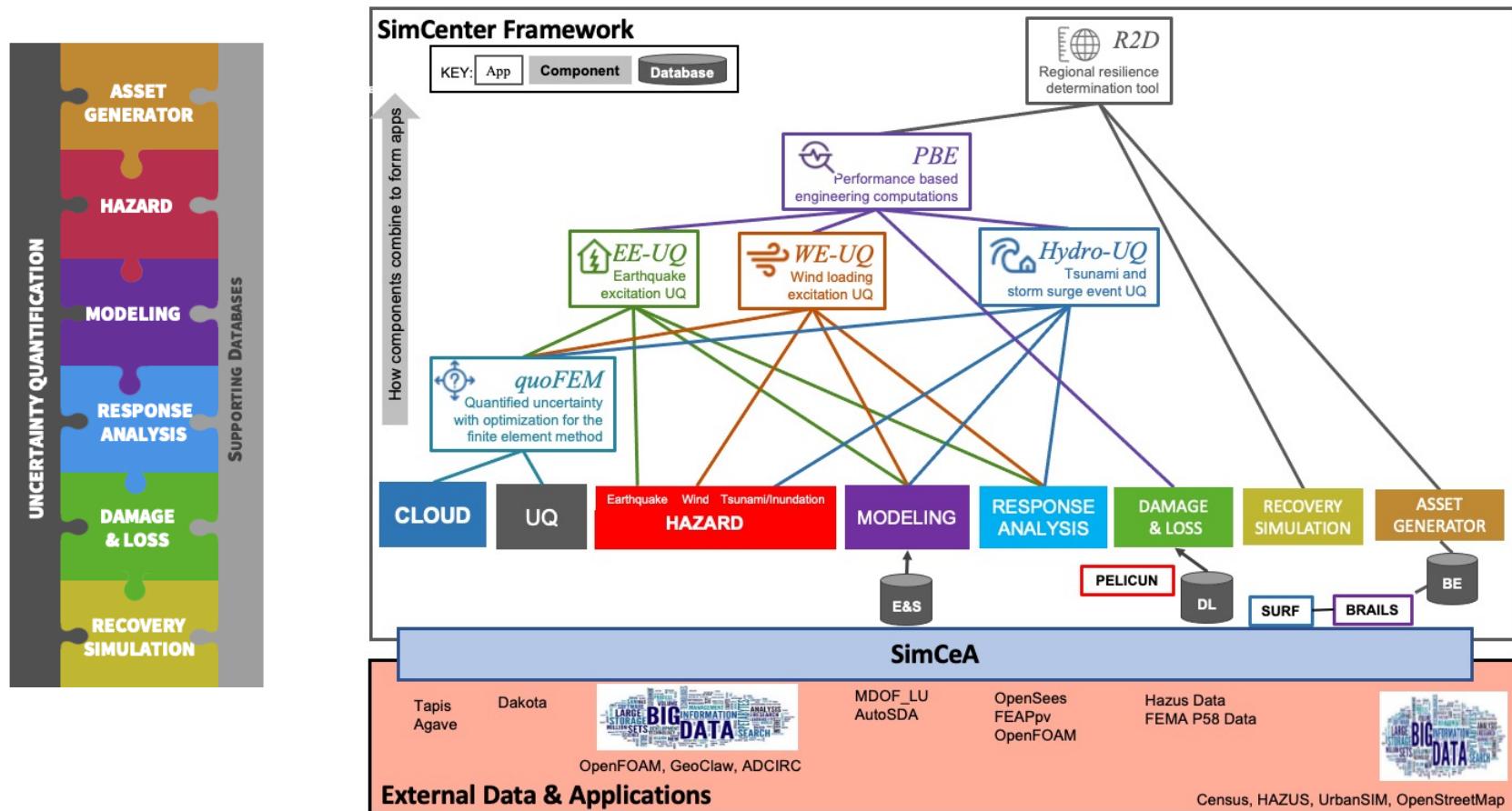


# Computational Framework for Building Workflows



**OpenSource :: Multi-Fidelity :: Multi-Hazard**

# SimCenter Applications Built with Framework





# Workflow Application

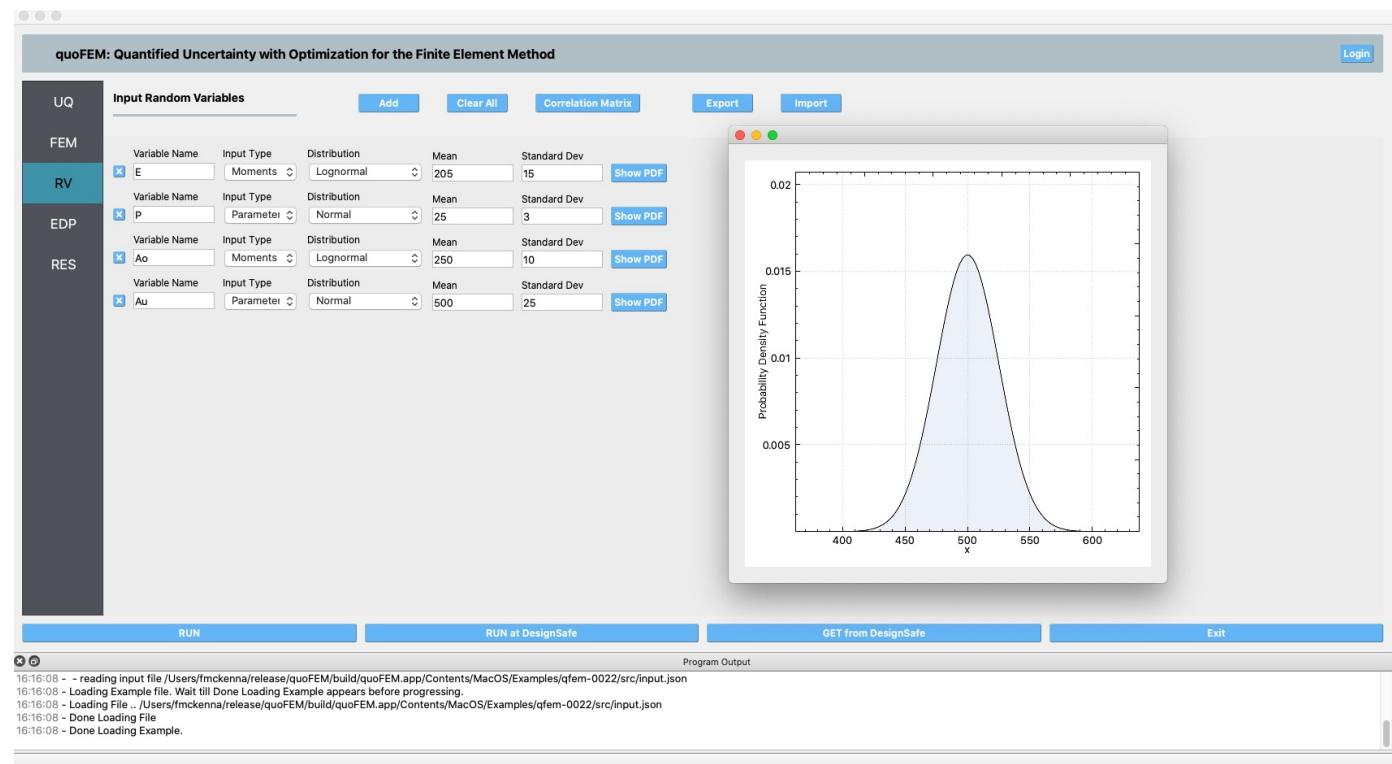


quoFEM (v3.0)

Integrates Simulation Applications with UQ Applications

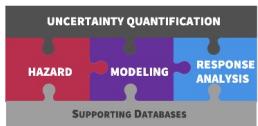
UQ Problem Types:

- Sampling**
- Sensitivity**
- Reliability**
- Calibration**
- Bayesian Calibration**
- Surrogate Modeling**





# EE-UQ Workflow Application



EE-UQ V3.0

Integrates UQ applications of quoFEM , Earthquake Loading Applications, Building Model Generators with analysis application to determining response of building to earthquake loading

## Hazard (Earthquake):

Stochastic Motions

PEER NGA Search with target spectrum

Site Response with Random Fields

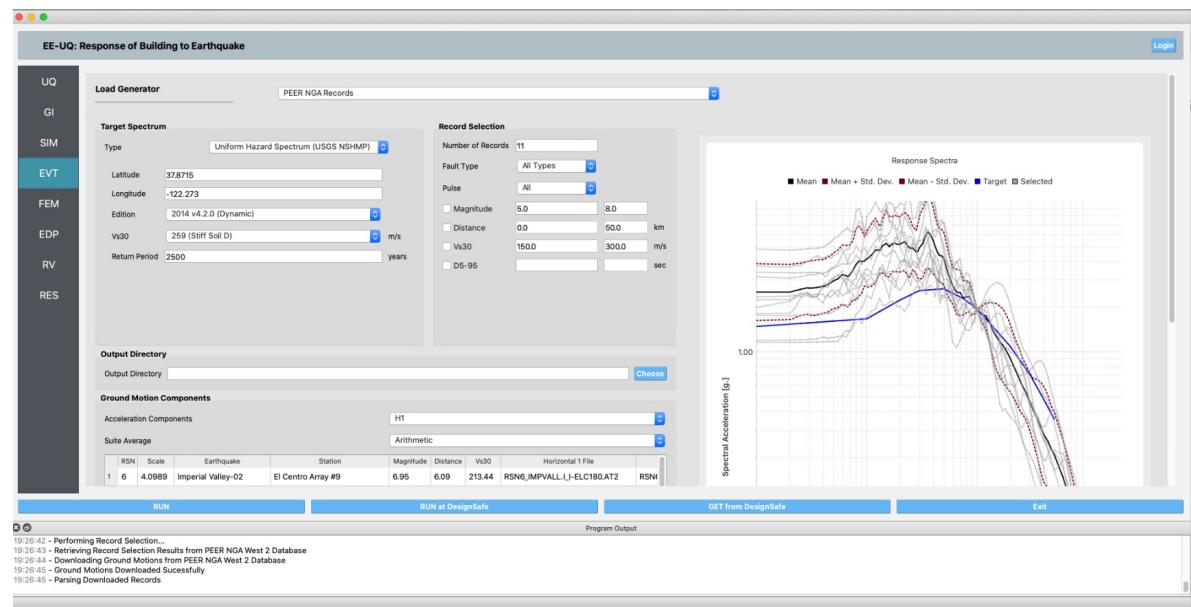
## Modeling (Building):

OpenSees

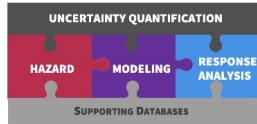
Nonlinear Shear Spring (MDOF)

Steel Building Design & Build (AutoSDA)

Concrete Building Design & Build



# WE-UQ Workflow Application



WE-UQ V2.2.0

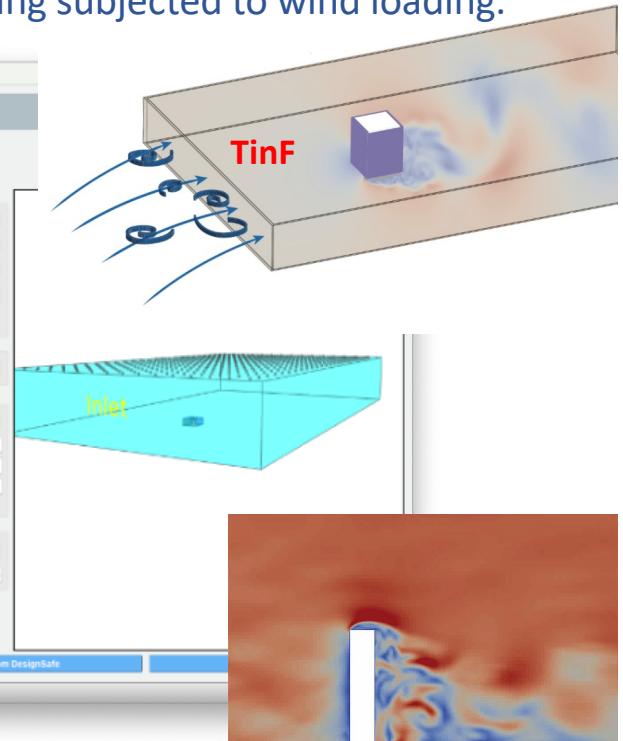
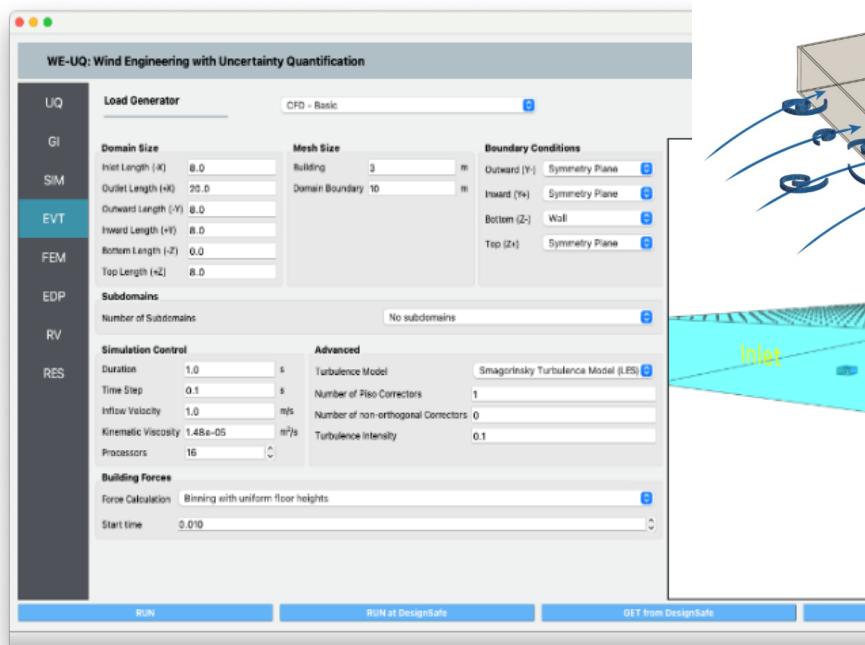
Integrates quoFEM UQ applications, Wind Loading Applications and Building Model Generators with analysis engine to determine response of building subjected to wind loading.

## Hazard (Wind):

Stochastic Wind  
Database-enabled utilizing Vortex-Winds TPU's low-rise wind tunnel datasets  
User-provided wind tunnel test data  
Uncoupled or **coupled** CFD simulations using OpenFOAM & incorporating TinF options for expert users

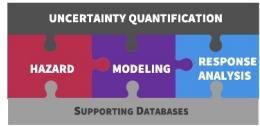
## Modeling (Building)

OpenSees  
Nonlinear Shear Spring (MDOF)





# Workflow Application



Hydro-UQ V1.0.0 –

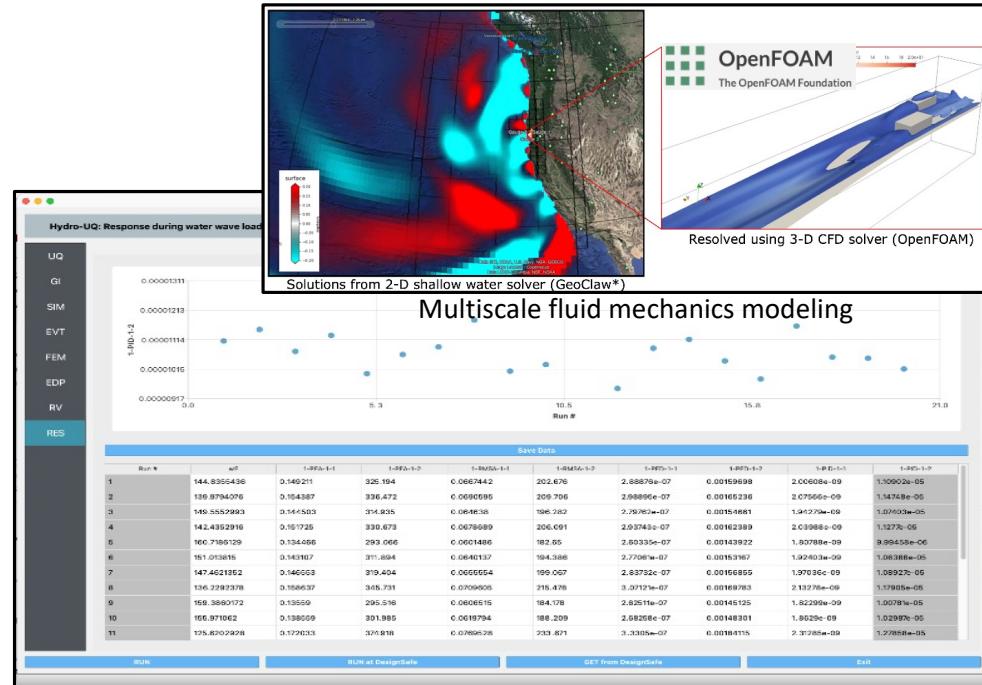
Integrates UQ applications, Tsunami & Storm Surge Loading Applications and Building Model Generators with Analysis engine to determine response of structure.

## Hazard (Tsunami and storm surge):

2D Shallow-water → 3D CFD

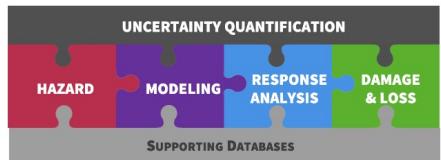
OSU wave flume digital twin

Easy to use turbulence models for studying  
broken & unbroken wave behavior



Probabilistic structural response

# PBE Workflow Application

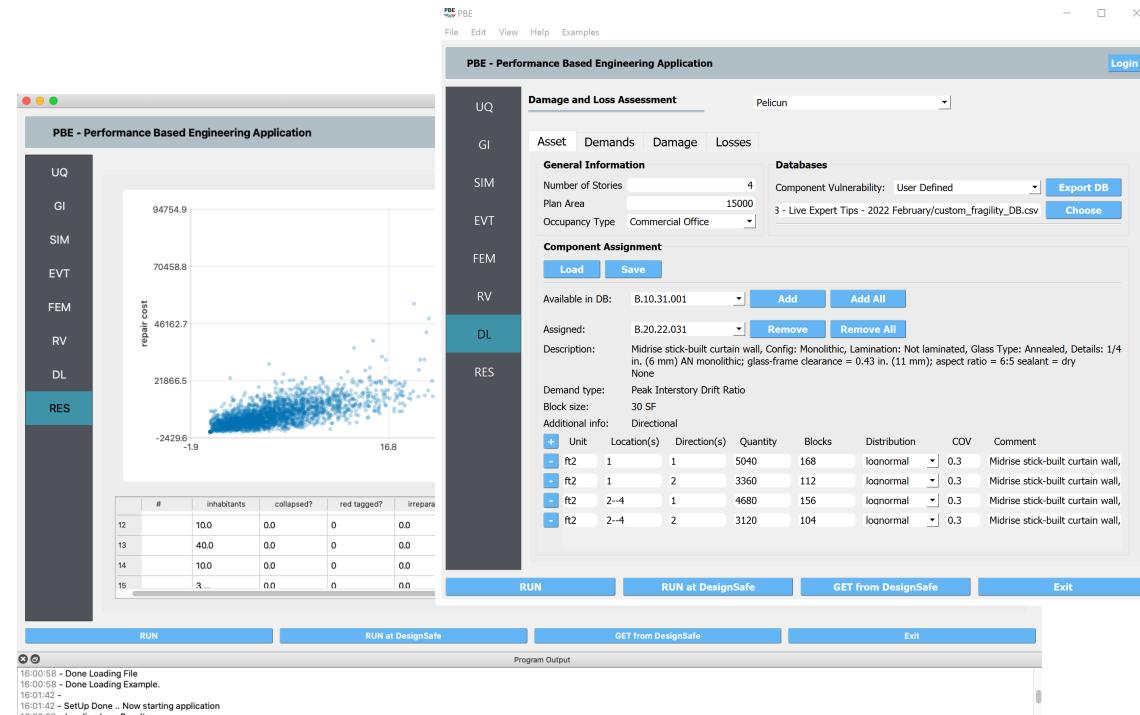
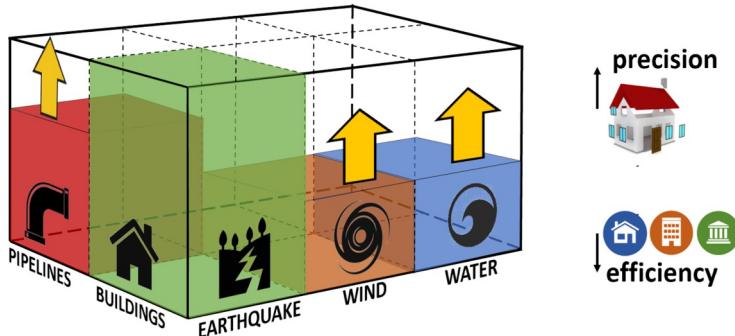


PBE v2.0 (3.0 coming this month!)

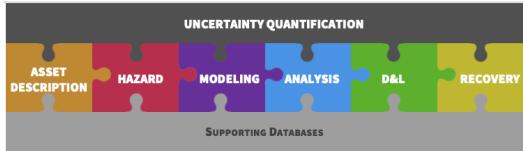
Integrates UQ applications of quoFEM, Building Model Generators, Earthquake Loading, analysis engine and our PELICUN tool for damage and loss assessment.

## Damage & Loss (using PELICUN):

- Building-level assessment (e.g., HAZUS)
- Component-level assessment (e.g., FEMA P58)
- Supports external response estimation
- Customizable fragility & consequence functions



# R2D Workflow Application



## Asset definition

csv files  
GIS files

## Hazard definition

**Regional Site Response**  
**User-supplied earthquake and hurricane grids**  
**Raster-defined earthquake, hurricane, and tsunami intensity fields**  
**Earthquake scenario simulation**  
**Hurricane wind field simulation**

## Damage and loss

**HAZUS**  
**User-provided fragility functions**

## R2D V2.1 –

Create **complex workflows for regional hazard simulation** to facilitate research in disaster risk management and recovery.

## Multiple Hazards



Earthquakes

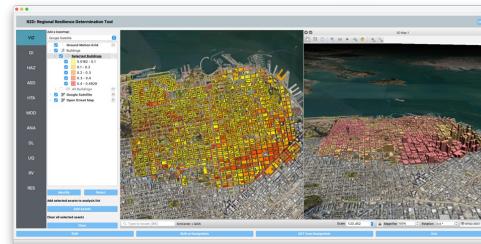


Hurricanes

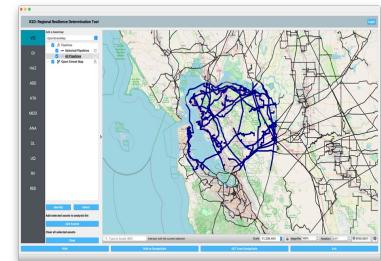


Tsunamis

## Multiple Assets



Buildings

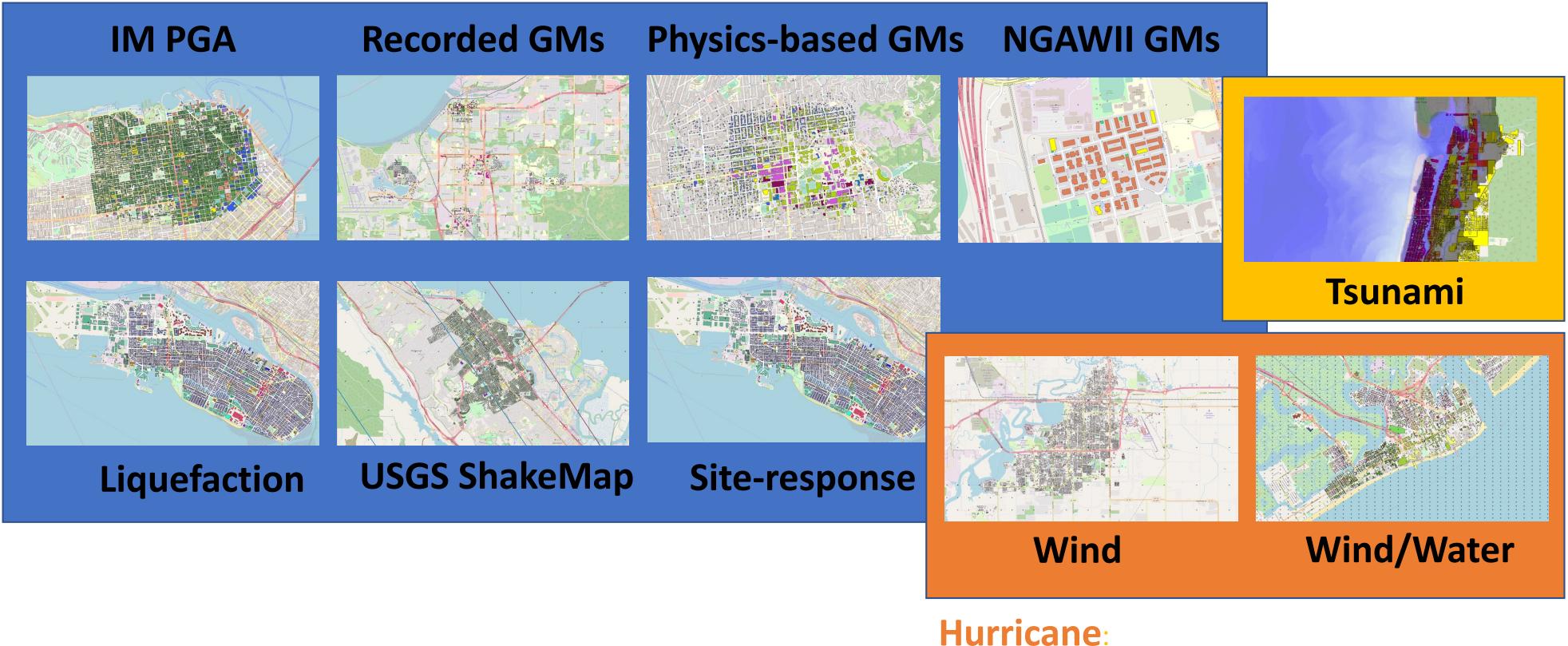


Lifelines



# Regional Testbeds & Examples Available

## Earthquake



# Todays Outline

- Expectations
- Daily Task For You
- NHERI SimCenter
- C Day 1
  - Computers & Programs
  - Introduction to C
  - Variables
  - Operations
  - Exercise
  - Function
  - Exercise



Center for Computational Modeling and Simulation

## Programming Bootcamp

# Computers & Computer Programs

Frank McKenna  
University of California at Berkeley



NSF award: CMMI 1612843



**A Computer is a programmable electronic device that manipulates data; that data being form of 0's and 1's; manipulation orchestrated by Programs running on the hardware**

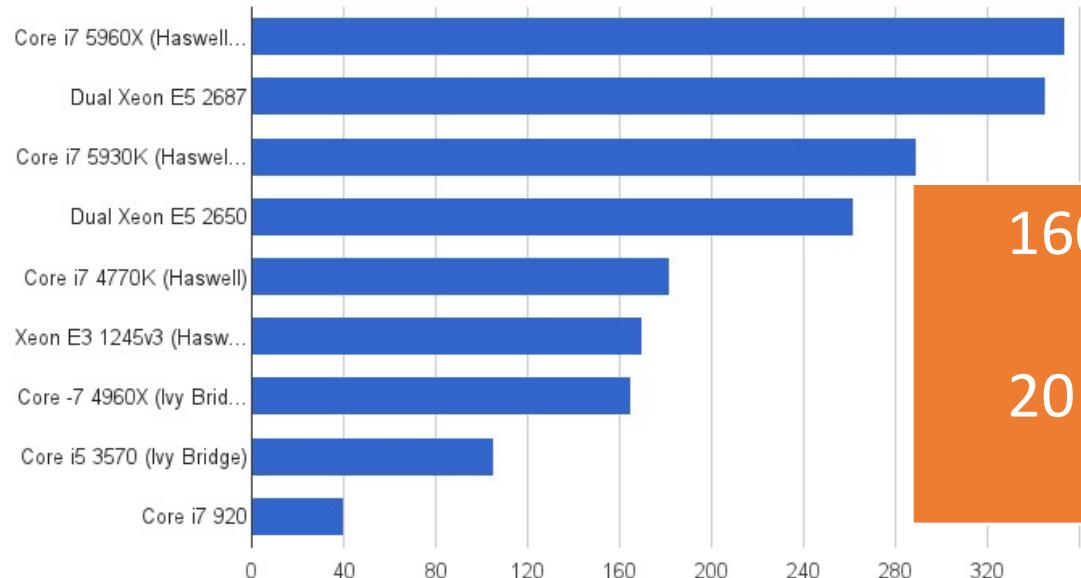
The Data

**Computers** Are Greatly Limited in What they can do!

They Just Do What They Can do Incredibly **Fast**

# How Fast?

Linpack Benchmark from Intel MKL

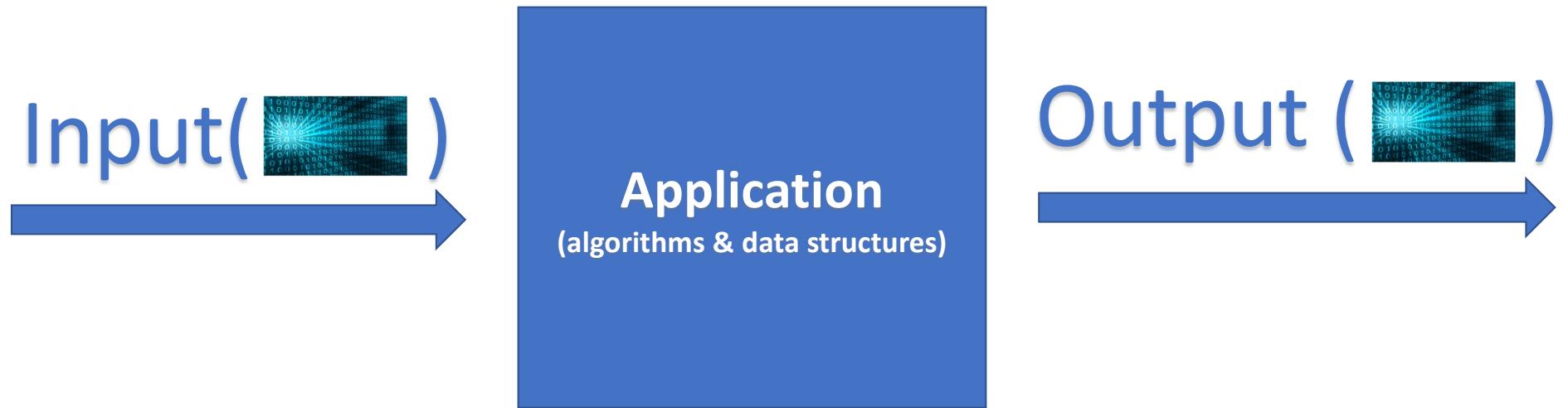


World population Dec 2020 = 7.8billion  
source: <https://www.worldometers.info/>

160 Billion FLOP/8 billion people  
=  
20 floating point operations per person per second!

GFLOPS = 1 billion floating point operations per second

# What is a Computer Application?

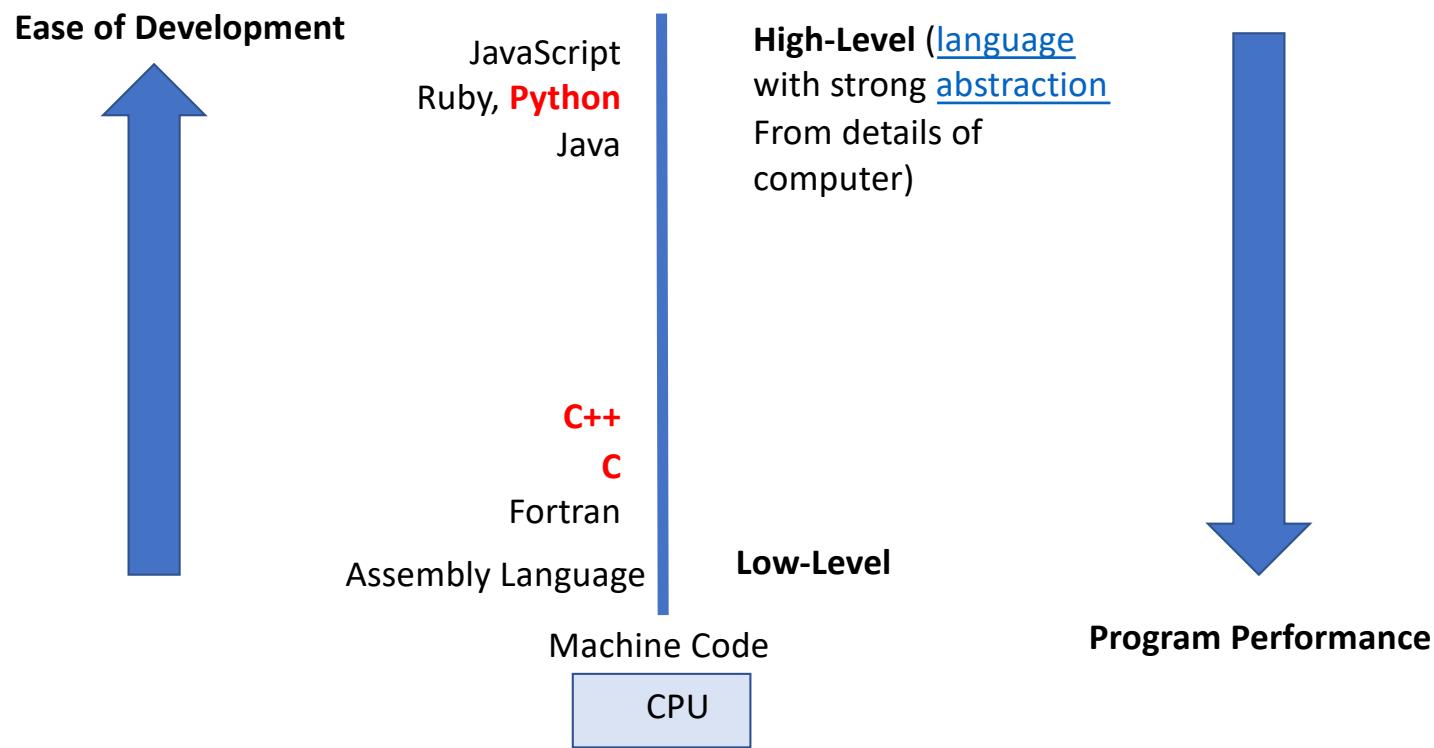


# What Programming Language?

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

PASCAL:	ADA:	C:	SHELL SCRIPT:	PYTHON:	TCL
<pre>if a &gt; 0 then   writeln("yes") else   writeln("no"); end if;</pre>	<pre>if a &gt; 0 then   Put_Line("yes"); else   Put_Line("no"); end if;</pre>	<pre>if (a &gt; 0) {   printf("yes"); } else {   printf("no"); }</pre>	<pre>if [\$a -gt 0]; then   echo "yes" else   echo "no" fi</pre>	<pre>if a &gt; 0:   print "yes" else:   print "no"</pre>	<pre>If (\$a &gt; 0) {   puts "yes" } else {   puts "no" }</pre>

# Programming Language Hierarchy



# Speed Comparison

## There's plenty of room at the Top: What will drive computer performance after Moore's law?

Charles E. Leiserson, Neil C. Thompson\*, Joel S. Emer, Bradley C. Kuszmaul, Butler W. Lampson, Daniel Sanchez, Tao B. Schardl

**Table 1. Speedups from performance engineering a program that multiplies two 4096-by-4096 matrices.** Each version represents a successive refinement of the original Python code. “Running time” is the running time of the version. “GFLOPS” is the billions of 64-bit floating-point operations per second that the version executes. “Absolute speedup” is time relative to Python, and “relative speedup,” which we show with an additional digit of precision, is time relative to the preceding line. “Fraction of peak” is GFLOPS relative to the computer’s peak 835 GFLOPS. See Methods for more details.

Version	Implementation	Running time (s)	GFLOPS	Absolute speedup	Relative speedup	Fraction of peak (%)
1	Python	25,552.48	0.005	1	—	0.00
2	Java	2,372.68	0.058	11	10.8	0.01
3	C	542.67	0.253	47	4.4	0.03
4	Parallel loops	69.80	1.969	366	7.8	0.24
5	Parallel divide and conquer	3.80	36.180	6,727	18.4	4.33
6	plus vectorization	1.10	124.914	23,224	3.5	14.96
7	plus AVX intrinsics	0.41	337.812	62,806	2.7	40.45

PYTHON: original  $160 \text{ GFLOP} / 60000 = 2.5 \text{ Million FLOPS}$  or each person in the 193rd most populous city, Lubumbashi Dr Congo (pop 2,478,262) doing 1 floating point operation a second!

# Language Types

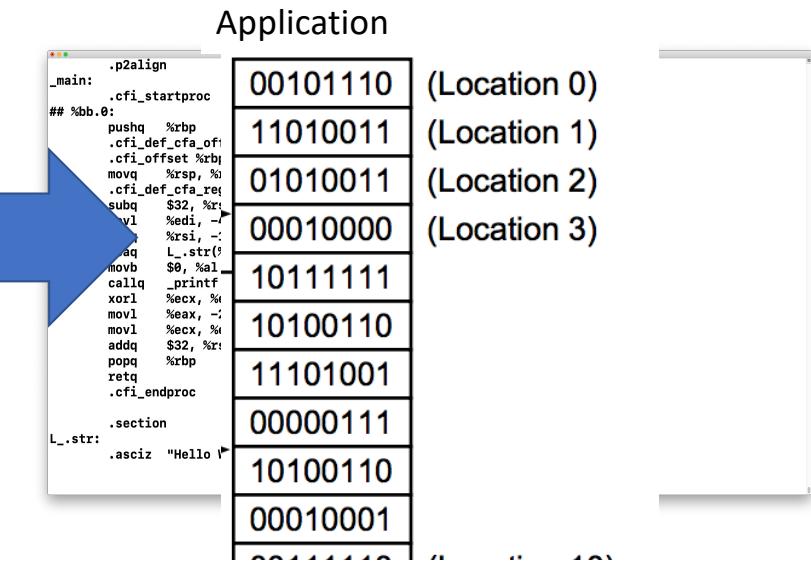
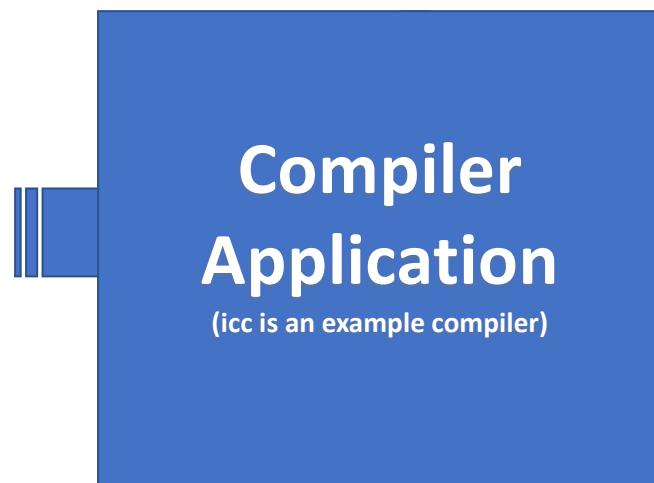
- C, C++ and Fortran are **compiled languages**. An application, the compiler takes the program and converts it into an application the user independently runs. A two step process to get results.
- Python is an **interpreted language**. The python interpreter directly runs the program when the user starts python and produces output. A one step process to get the results.

# Compiled Language:

- The purpose of the compiler:
  - Translate a program into lower level language (assembly, machine instruction)
  - Check a program is legal, i.e. follows the syntax
  - Optimize the output for performance, e.g. order of instructions,

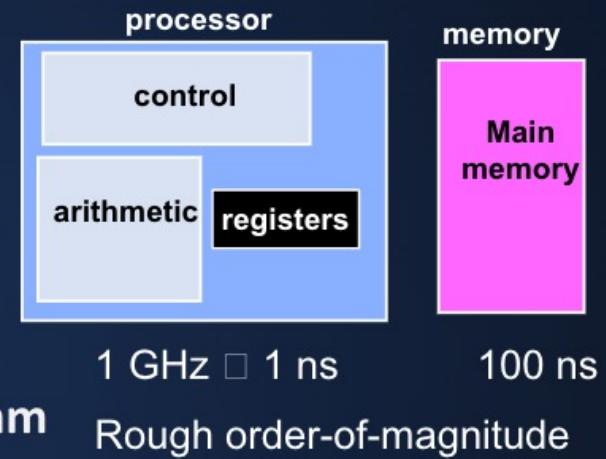
Typically ASCII Text File

```
void main() {  
    ...  
    ...  
    a = b + c;  
    ...  
    ...  
}
```

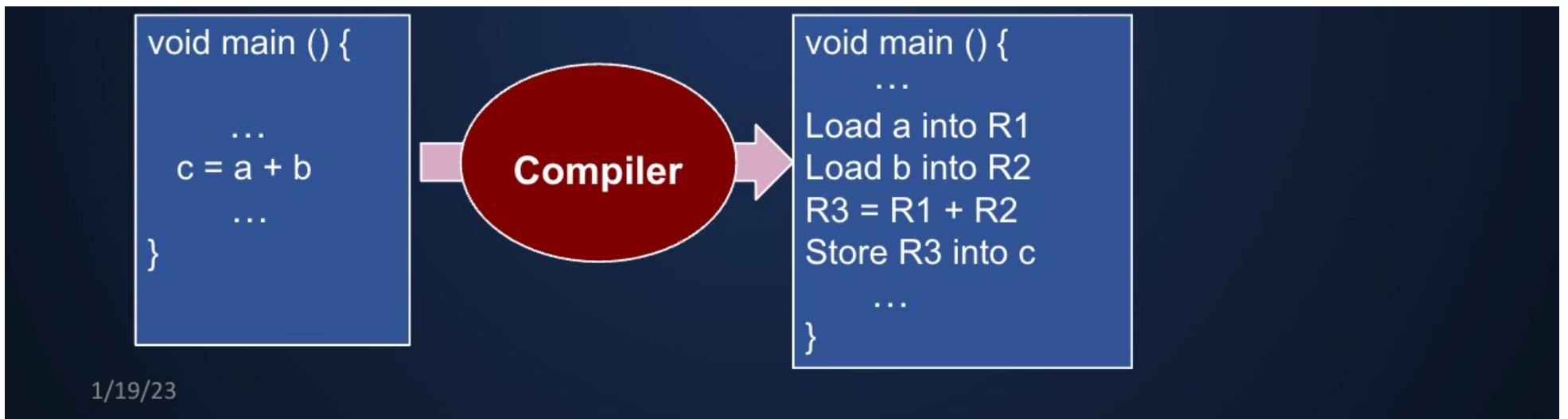
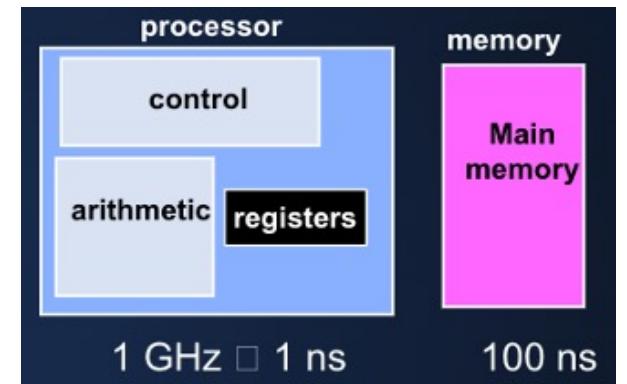


# Idealized Model of a Uniprocessor Machine

- **Processor names variables:**
  - Integers, floats, doubles, pointers, arrays, structures, etc
  - *really bytes, words, etc. in its address space*
- **Processor performs operations on those variables:**
  - Arithmetic operations, etc. only on values in registers
  - Load/Store variables between memory and registers
- **Processor controls the order, as specified by program**
  - Branches (if), loops, function calls, etc.
  - *Compiler translates into “obvious” lower level instructions*
  - *Hardware executes instructions in order specified by compiler*
  - *Read returns the most recently written data*
- ***Idealized Cost***
  - Each operation (+, \*, &, etc.) has roughly the same cost (on “free” registers)
  - Load/store is 100x the cost of +, \*, &, etc.



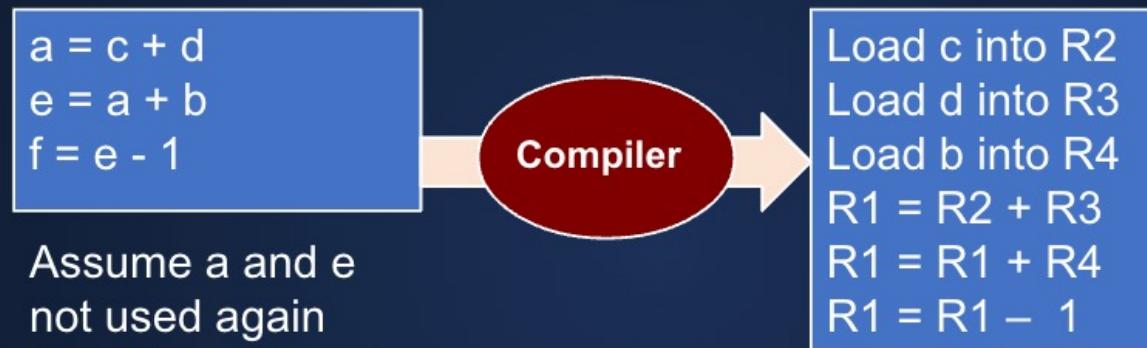
# Memory is the Bottleneck



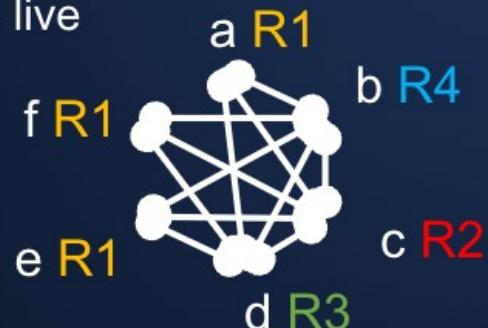
Source: Demmell, CS267 UC Berkeley, Spring 2023

# Why Compilers Like C/Fortran Outperform

- Compiler performs “register allocation” to decide when to load/store vs reuse



Assume a and e  
not used again  
(dead), b,c,d, and  
f are live



*Register allocation in  
first Fortran compiler in  
1950s, graph coloring  
in 1980. JITs may use  
something cheaper*

# Compiler Optimize Code

- Besides Register Allocation, compiler will:
  - Unroll loops
  - Fuse Loops (merges two together)
  - Interchanges Loops
  - Eliminates Dead Code
  - Reorder Instructions to improve reuse

BUT

- Sometimes it does the best thing possible
- But other times it does not ... i.e. it is up to you to help things.
- Compiler does not know about different algorithms that are better suited given the architecture



Center for Computational Modeling and Simulation

## Programming Bootcamp

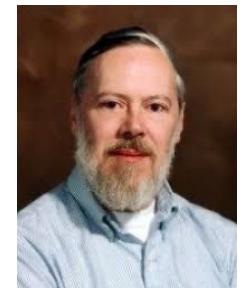
# Introduction to C & Compilation

Frank McKenna  
University of California at Berkeley



NSF award: CMMI 1612843

# 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. (Python is an interpreted language)
- It is a **structured** (PROCEDURAL) language. (Python is ‘object-oriented’ if you use classes, Fortran is procedural)
- It is a **strongly typed** language (Python is also strongly typed, but interpreter figures out current type!)
- The most widely used languages of all time (Python is language du jour)
- 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)
- It can produce **FAST** code by maximizing your hardware resources.

# Strongly Typed

- **Strongly typed language** refers to a programming language that typically requires the programmer to specify the type of a variable so that strict enforcement of restrictions on intermixing of values with differing data types can be performed.

# Structured Programming Language

- In this type of language, large programs are divided into smaller units called **functions**.
- Data is passed from one function to another through function arguments and function return types.
- The main focus in program design is on identifying functions and the defining the function interfaces, i.e. identifying the data types passed in the function arguments.

# C Program Structure

A C Program consists of the following parts:

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

# Everyone's First C Program

no space between # and include

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

hello1.c

```
int main(int argc, char **argv) {
    /* 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. Its implementation in the standard C library.
- The next statement **return 0;** terminates the main() function and returns the value 0.



Center for Computational Modeling and Simulation

## Programming Bootcamp

# C: Variable Types & Variable Names

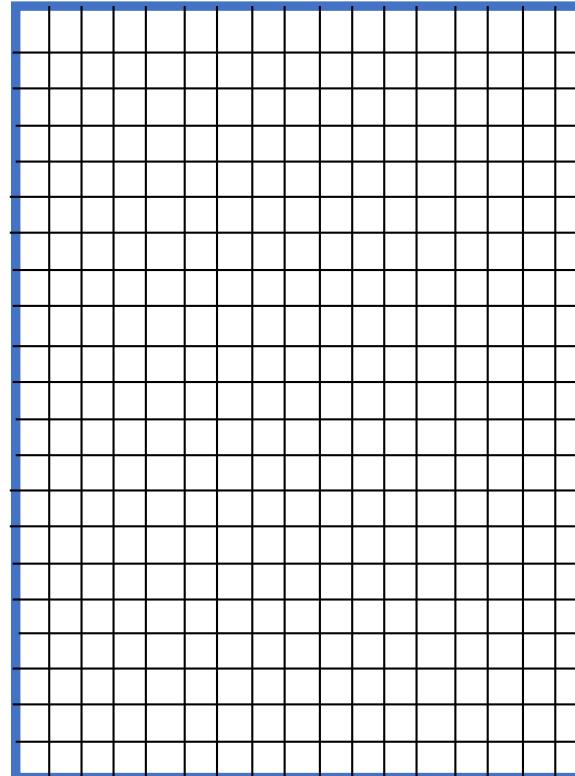
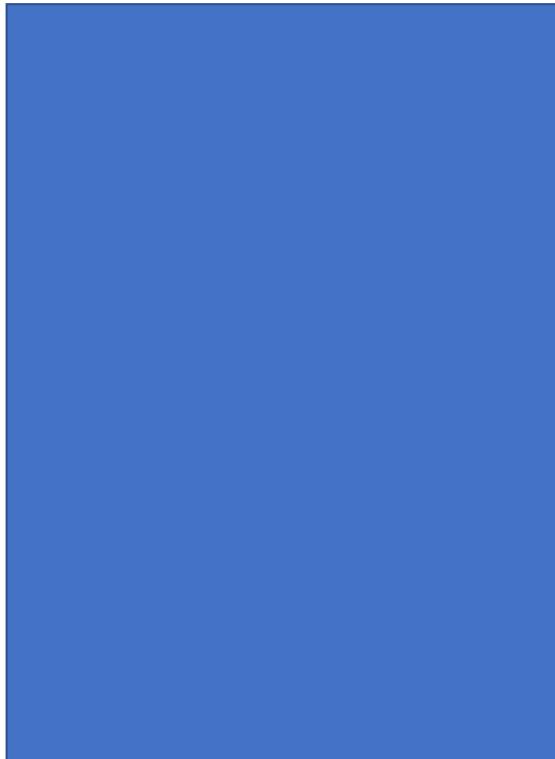
Frank McKenna  
University of California at Berkeley



NSF award: CMMI 1612843

All data in program memory is stored as 0's and 1's

C breaks program memory into byte sized units  
1 Byte = 8 bits



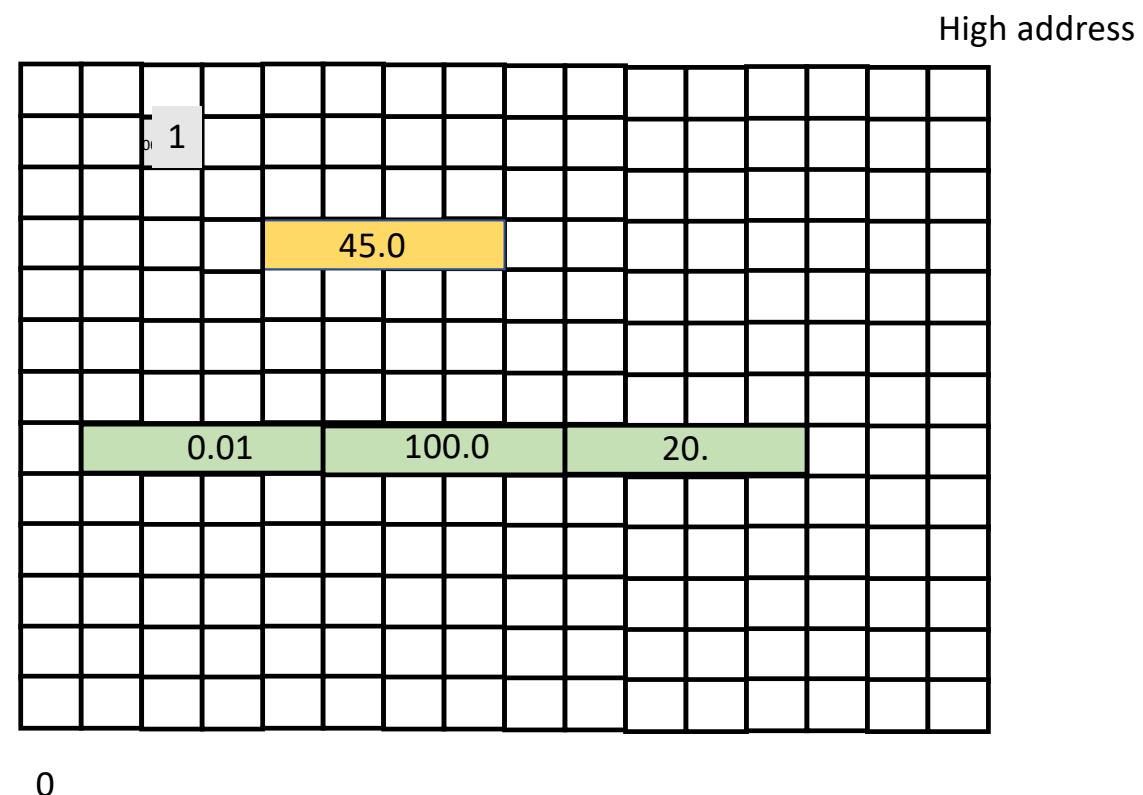
To a single Byte or Contiguous Array of Bytes the Programmer can associate the byte(s) with a variable name, and then use that name in the program to set and use the values stored in those bytes.

```
char a = '1';
```

```
float theta = 45.0;
```

```
float stress[3] = {0.01, 100. 20.};
```

NOTE: each of these byte has a memory location which can be obtained with the location varying from 0 to high address value..



# Variable Name

- A **Variable** is a named memory location that can hold various values.
- Because C is a **strongly typed language**, the programmer **must specify the data type associated with the variable**. The compiler will issue warnings if variable type is misused.
- The size of the memory associated with the variable depends on the data type.
- Names are **made up of letters and digits**; they are **case sensitive**; names must start with a character, for variable names ‘\_’ counts as a character
- **Certain keywords are reserved, i.e. cannot be used as variable names**

# char – size 1 byte ( $2^8 = 256$ (0-255) possibilities)

ASCII Character set (only uses 128 possibilities) – note '1' is 49 which corresponds to bit pattern 00110001

Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII
0	00000000	000	00	NUL	32	00100000	040	20	SP	64	01000000	100	40	@	96	01100000	140	60	'
1	00000001	001	01	SOH	33	00100001	041	21	!	65	01000001	101	41	A	97	01100001	141	61	a
2	00000010	002	02	STX	34	00100010	042	22	"	66	01000010	102	42	B	98	01100010	142	62	b
3	00000011	003	03	ETX	35	00100011	043	23	#	67	01000011	103	43	C	99	01100011	143	63	c
4	00000100	004	04	EOT	36	00100100	044	24	\$	68	01000100	104	44	D	100	01100100	144	64	d
5	00000101	005	05	ENQ	37	00100101	045	25	%	69	01000101	105	45	E	101	01100101	145	65	e
6	00000110	006	06	ACK	38	00100110	046	26	&	70	01000110	106	46	F	102	01100110	146	66	f
7	00000111	007	07	BEL	39	00100111	047	27	'	71	01000111	107	47	G	103	01100111	147	67	g
8	00001000	010	08	BS	40	00101000	050	28	(	72	01001000	110	48	H	104	01101000	150	68	h
9	00001001	011	09	HT	41	00101001	051	29	)	73	01001001	111	49	I	105	01101001	151	69	i
10	00001010	012	0A	LF	42	00101010	052	2A	*	74	01001010	112	4A	J	106	01101010	152	6A	j
11	00001011	013	0B	VT	43	00101011	053	2B	+	75	01001011	113	4B	K	107	01101011	153	6B	k
12	00001100	014	0C	FF	44	00101100	054	2C	,	76	01001100	114	4C	L	108	01101100	154	6C	l
13	00001101	015	0D	CR	45	00101101	055	2D	-	77	01001101	115	4D	M	109	01101101	155	6D	m
14	00001110	016	0E	SO	46	00101110	056	2E	.	78	01001110	116	4E	N	110	01101110	156	6E	n
15	00001111	017	0F	SI	47	00101111	057	2F	/	79	01001111	117	4F	O	111	01101111	157	6F	o
16	00010000	020	10	DLE	48	00110000	060	30	0	80	01010000	120	50	P	112	01100000	160	70	p
17	00010001	021	11	DC1	49	00110001	061	31	1	81	01010001	121	51	Q	113	01100001	161	71	q
18	00010010	022	12	DC2	50	00110010	062	32	2	82	01010010	122	52	R	114	01100010	162	72	r
19	00010011	023	13	DC3	51	00110011	063	33	3	83	01010011	123	53	S	115	01100011	163	73	s
20	00010100	024	14	DC4	52	00110100	064	34	4	84	01010100	124	54	T	116	01100100	164	74	t
21	00010101	025	15	NAK	53	00110101	065	35	5	85	01010101	125	55	U	117	01101011	165	75	u
22	00010110	026	16	SYN	54	00110110	066	36	6	86	01010111	126	56	V	char one = '1'; char yes = 'Y';				
23	00010111	027	17	ETB	55	00110111	067	37	7	87	01010111	127	57	W					
24	00011000	030	18	CAN	56	00111000	070	38	8	88	01011000	128	58	X					
25	00011001	031	19	EM	57	00111001	071	39	9	89	01011001	129	59	Y					
26	00011010	032	1A	SUB	58	00111010	072	3A	:	90	01011010	130	5A	Z					
27	00011011	033	1B	ESC	59	00111011	073	3B	;	91	01011011	131	5B	_					
28	00011100	034	1C	FS	60	00111100	074	3C	<	92	01011100	134	5C	\	124	01111100	174	7C	
29	00011101	035	1D	GS	61	00111101	075	3D	=	93	01011101	135	5D	]	125	01111101	175	7D	}
30	00011110	036	1E	RS	62	00111110	076	3E	>	94	01011110	136	5E	^	126	01111110	176	7E	~
31	00011111	037	1F	US	63	00111111	077	3F	?	95	01011111	137	5F	_	127	01111111	177	7F	DEL

**int** – size 4 Byte ( $2^{32}$  possibilities)

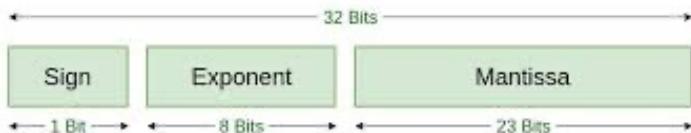
any int in range: -2,147,483,648 to 2,147,483,647

**unsigned int** – size 4 Byte ( $2^{32}$  possibilities)

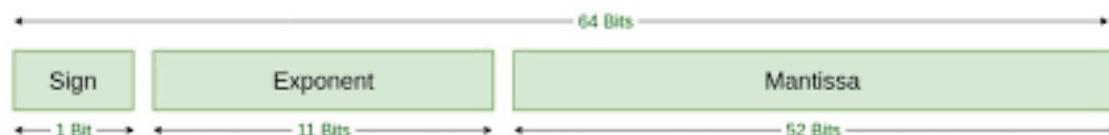
any non negative int in range: 0 to 4,294,967,295

```
int a = 1;  
unsigned int b = 2;
```

# float (4 Byte) and double (8 Byte) numbers



Single Precision  
IEEE 754 Floating-Point Standard



Double Precision  
IEEE 754 Floating-Point Standard

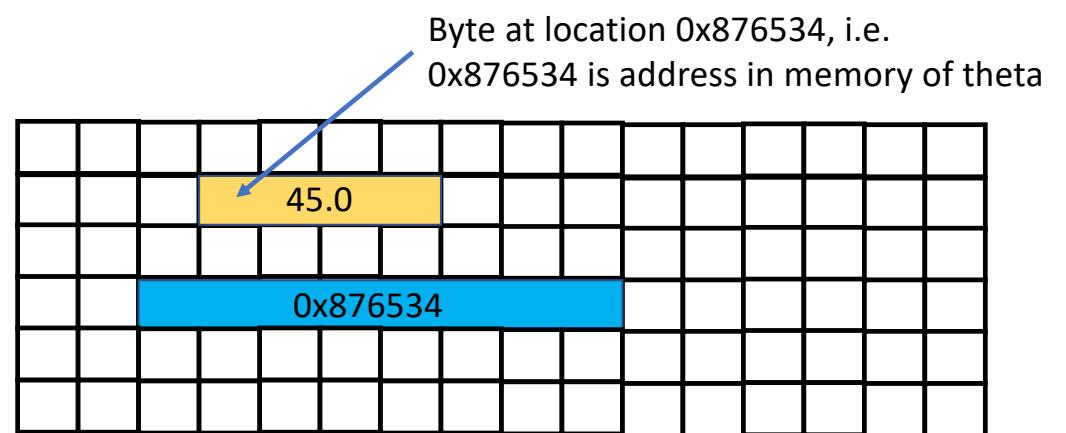
**float**    32 bits     $\pm 1.18 \times 10^{-38}$  to  $\pm 3.4 \times 10^{38}$     Approx. 7 decimal digits

**double**    64 bits     $\pm 2.23 \times 10^{-308}$  to  $\pm 1.80 \times 10^{308}$ .    Approx. 16 decimal digits

```
float theta = 45.0;  
double force = stress * Area;
```

**pointer** – size 8 Byte (x64 – 64bit application) or 4 Byte  
(x86 = 32 bit application)

```
float theta = 45.0;  
float *thetaPtr= &theta;
```



# Pointer Variables

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

1. The unary \* in a declaration indicates that the object is a pointer to an object of a specific type

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

pointer1.c

Address in memory  
of x is 0x789AB32

1.



```
int x=10;
int y;
int *ptrX = 0;
```

# Pointer Variables

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

1. The unary **\*** in a declaration indicates that the object is a pointer to an object of a specific type
2. The unary **&** gives the “**address**” of an object in memory, e.g. **&x** is location of variable x

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

pointer1.c

Address in memory  
of x is 0x789AB32

- 1.
- 2.



```
int x=10;
int y;
int *ptrX = 0;
```

```
ptrX = &x;
```

We say ptrX points to x

# Pointer Variables

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

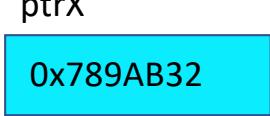
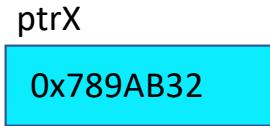
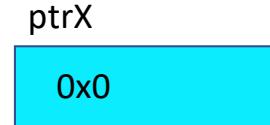
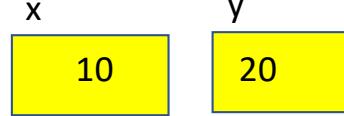
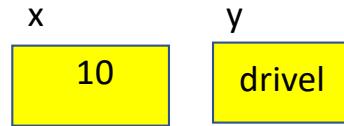
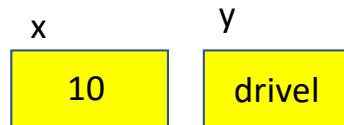
1. The unary **\*** in a declaration indicates that the object is a pointer to an object of a specific type
2. The unary **&** gives the “**address**” of an object in memory, e.g. **&x** is location of variable x
3. The unary **\*** elsewhere treats the operand as an address and dereferences it, which depending on which side of operand is does one of two things:
  - a. fetches the contents for use in an expression

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

pointer1.c

Address in memory  
of x is 0x789AB32

- 1.
- 2.
- 3.a



```
int x=10;
int y;
int *ptrX = 0;
```

```
ptrX = &x;
```

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

# Pointer Variables

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

1. The unary **\*** in a declaration indicates that the object is a pointer to an object of a specific type
2. The unary **&** gives the “**address**” of an object in memory, e.g. **&x** is location of variable x
3. The unary **\*** elsewhere treats the operand as an address and dereferences it, which depending on which side of operand is does one of two things:
  - a. fetches the contents for use in an expression
  - b. Sets the memory location to which it points to some value.

```
#include <stdio.h>
int main() {
    int x =10, y;
    int *ptrX =0;
    ptrX = &x;
    y = *ptrX + x;
    *ptrX = 50;
}
```

pointer1.c

Address in memory  
of x is 0x789AB32

	x	y	ptrX
1.	10	drivel	0x0
2.	10	drivel	0x789AB32
3.a	10	20	0x789AB32
3.b	50	20	0x789AB32

```
int x=10;
int y;
int *ptrX = 0;
```

ptrX = &x;

y = \*ptrX + x;

\*ptrX = 50;

# Variable Example

```
#include <stdio.h>
// define and then set variable
int main(int argc, char **argv) {
    int a;
    a = 1;
    printf("Value of a is %d \n", a);
    return 0;
}
```

var1.c

Uninitialized Variable

```
#include <stdio.h>
// define & set in 1 statement
int main(int argc, char **argv) {
    int a = 1;
    printf("Value of a is %d \n", a);
    return 0;
}
```

var2.c

Initialized Variable

# Some Common Variable Types in C

char  
int  
float  
double

```
#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("int %d, float %f, double %lf, char %c \n", i1, f1, d1, c1);
    printf("int %d, float %e, double %le, char %c \n", i1, f1, d1, c1);
    printf("int %3d, float %16.3e, double %23.8le, char %2c \n", i1, f1, d1,
return 0;
}
```

var3.c

## 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}`

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

array1.c

```
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 ;
}
```

**WARNING: indexing  
starts at 0**

a[0]	a[1]	a[2]	a[3]	a[4]
21	12	13	14	50

# 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. Pointers

### 6. Derived Types

Structures,  
Unions,  
**Arrays**



Center for Computational Modeling and Simulation

## Programming Bootcamp

# C: Operations, Conditionals & Loops

Frank McKenna  
University of California at Berkeley



NSF award: CMMI 1612843

# Operations

- We want to do stuff with the data, to operate on it
- Basic Arithmetic Operations

+, -, \*, /, %

```
#include <stdio.h> op1.c

int main(int argc, const char **argv) {
    int a = 1;
    int b = 2;
    int c = a+b;
    printf("Sum of %d and %d is %d \n",a,b,c);
    return(0);
}
```

# You Can String Operations Together –

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

op2.c

```
int main(int argc, char **argv) {
```

```
    int a = 5;
```

```
    int b = 2;
```

```
    int c = a + b * 2;
```

**What is c? Operator precedence!**

```
    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;
```

**USE PARENTHESES**

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

```
    return(0);
```

```
}
```

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

# Some Operations are so Common there are special operators

```
#include <stdio.h>  
  
int main() {  
    ...  
    a = a + 1;  
    ...  
}
```

**+ =**

```
a += 1;
```

**- =**

**\* =**

**/ =**

**++**

```
a ++;
```

**--**

# Conditional Code – if statement

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

- So far instruction sequence has been sequential, one instruction after the next .. Beyond simple programs we need to start doing something, if balance is less than 0 don't withdraw money

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

if1.c

Conditional Operators

<      <=      >      >=      ==      !=

# If-else

```
if (condition) {  
    // code block  
} else {  
    // other code  
}
```

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

if2.c

# else-if

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

```
#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);  
}
```

if3.c

Can have multiple else if in if statement

# Logical Operators: and/or/not

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

&&  
||  
!

# Iteration/loops - while

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

- Common task is to loop over a number of things, e.g. look at all files in a folder, loop over all values in an array,...

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

while1.c

```
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++; // If left out =>infinite loop ..  
                  // Something must happen in while to break out of loop  
    }  
    printf("sum is: %d \n", sum);  
}
```

If you do enough while loops you will recognize a pattern

- 1) Initialization of some variables,
- 2) condition,
- 3) increment of some value

Hence the for loop

# for loop

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

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

for1.c

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

# Math Functions in <math.h>, link with -lm

Pre-C99 functions [\[edit\]](#)

Name	Description
acos	inverse cosine
asin	inverse sine
atan	one-parameter inverse tangent
atan2	two-parameter inverse tangent
ceil	ceiling, the smallest integer not less than parameter
cos	cosine
cosh	hyperbolic cosine
exp	exponential function
fabs	absolute value (of a floating-point number)
floor	floor, the largest integer not greater than parameter
fmod	floating-point remainder: $x - y * (\text{int})(x/y)$
frexp	break floating-point number down into mantissa and exponent
ldexp	scale floating-point number by exponent (see article)
log	natural logarithm
log10	base-10 logarithm
modf(x,p)	returns fractional part of $x$ and stores integral part where pointer $p$ points to
pow(x,y)	raise $x$ to the power of $y$ , $x^y$
sin	sine
sinh	hyperbolic sine
sqrt	square root
tan	tangent
tanh	hyperbolic tangent]]

```
#include <stdio.h>
#include <math.h>
int main() {
    double a = 34.0;
    double b = sqrt(a);
    printf("%f + %f = %f \n", a, b);
    return 0;
}
```

```
c >gcc math1.c -lm; ./a.out
sqrt(34.000000) is 5.830952
c >
```

## EXERCISE: Exercise 1

Write a C program that takes three double inputs (a,b,c) and computes the solution to the quadratic equation:

$$ax^2 + bx + c = 0$$

\*Assume for simplicity NO complex roots.

Input: a,b,c

Output: x<sub>1</sub>,x<sub>2</sub> or x<sub>1</sub> if single root .. Specified to 6 significant digits.



Center for Computational Modeling and Simulation

## Programming Bootcamp

# C: Functions

Frank McKenna  
University of California at Berkeley



NSF award: CMMI 1612843

# Functions

- **Art of Programming:** “*To take a problem, and recursively break it down into a series of smaller tasks until ultimately these tasks become a series of small specific individual instructions.*”
- For large code projects we do not put all the code inside a single main block
- We break it up into logical/meaningful blocks of code. In object-oriented programming we call these blocks **classes**, in procedural programming we call these blocks **procedures or functions**.
- Functions make large programs manageable: easier to understand, allow for code re-use, allow it to be developed by teams of programmers,..

## 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.

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

function1.c

```
// function to evaluate vector sum
int sumArray(int *data, int size) {
    int sum = 0;
    for (int i = 0; i < size; i++) {
        sum += data[i];
    }
    return sum;
}
```

```
int main(int argc, char **argv) {
    int intArray[6] = {19, 12, 13, 14, 50, 0};
    int sum = sumArray(intArray, 6);
    printf("sum is: %d \n", sum);
    return(0);
}
```

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

function2.c

```
// function to evaluate vector sum
int sumArray(int *data, int size) {
    int sum = 0;
    for (int i = 0; i < size; i++) {
        sum += *data++;
    }
    return sum;
}

int main(int argc, char **argv) {
    int intArray1[6] = {19, 12, 13, 14, 50, 0};
    int intArray2[3] = {21, 22, 23};
    int sum1 = sumArray(intArray1, 6);
    int sum2 = sumArray(intArray2, 3);
    printf("sums: %d and %d\n", sum1, sum2);
    return(0);
}
```

# Function Prototype

```
#include <stdio.h>                                         function3.c
int sumArray(int *arrayData, int size);

int main(int argc, char **argv) {
    int intArray1[6] = {19, 12, 13, 14, 50, 0};
    int intArray2[3] = {21, 22, 23};
    int sum1 = sumArray(intArray1, 6);
    int sum2 = sumArray(intArray2, 3);
    printf("sums: %d and %d\n", sum1, sum2);
    return(0);
}

// function to evaluate vector sum
int sumArray(int *data, int size) {
    int sum = 0;
    for (int i = 0; i < size; i++) {
        sum += *data++;
    }
    return sum;
}
```

Good practice to give the args names. Names are not actually needed (but they remind you and show others the purpose of each arg)

## Good Practice:

1. For large programs it is a good idea to put functions into different files (many different people can be working on different parts of the code)
2. If not too large, put them in logical units, i.e. all functions dealing with vector operations in 1 file (*vector.c*), matrix operations in another (*matrix.c*).
3. Put prototypes for all functions in another file, a .h file, e.g. *vector.h*
4. Get into a system of documenting inputs and outputs.

# Example: application that requires use of vectors

```
#ifndef _MY_VECTOR_H
#define _MY_VECTOR_H

double *createMyVector(int size);
double normMyVector(double *data, int size);
double dotMyVector(double *data1, double *data2, int size);
void dotSumMyVector(double *data1, double *data2, double *result, int size);
void printMyVector(double *data, int size, char *name);

#endif
```

**myVector.h**

```
/* function to return component sum of two vectors */
void normMyVector(double *v1, double *v2, double *res , int size) {
    for (int i = 0; i < size; i++) {
        res[i] = vector1[i] + vector2[i];
    }
}
```

**myVector.c**

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include "myVector.h"

int main(int argc, char **argv) {
    if (argc != 3) {
        fprintf(stderr,"Need 3 args: appName n maxVal\n");
        return -1;
    }
    int n = atoi(argv[1]);
    double maxVal = atof(argv[2]);

    double *data1 = createMyVector(n);
    double *data2 = createMyVector(n);
    double *data3 = createMyVector(n);

    // fill in vectors with random values
    srand(100); //srand((unsigned int)time(0));
    for (int i=0; i<n; i++) {
        data1[i] = ((float)rand()/(float)RAND_MAX) * maxVal;
        data2[i] = ((float)rand()/(float)RAND_MAX) * maxVal;;
    }
    printMyVector(data1, n, "vector 1");
    printMyVector(data2, n, "vector 2");

    // dotSum the vectors & print result;
    dotSumMyVector(data1, data2, data3, n);
    printMyVector(data3, n, "dotSum");
    return 0;
}
```

**vector1.c**

# Scope of Variables

```
#include <stdio.h>
int sum(int, int);
int x = 20; // global variable
int main(int argc, char **argv) {
    printf("LINE 5: x = %d\n",x);

    int x = 5;
    printf("LINE 8: x = %d\n",x);

    if (2 > 1) {
        int x = 10;
        printf("LINE 12: x = %d\n",x);
    }
    printf("LINE 14: x = %d\n",x);

    x = sum(x,x);
    printf("LINE 17: x = %d\n",x);
}

int sum(int a, int b) {
    printf("LINE 21: x = %d\n",x);
    return a+b;
}
```

scope1.c

```
[c >gcc scope1.c; ./a.out
LINE 5: x = 20
LINE 8: x = 5
LINE 12: x = 10
LINE 14: x = 5
LINE 21: x = 20
LINE 17: x = 10
c >]
```

# 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.**

```
#include <stdio.h>                                         function4.c
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;
}
```

# Math Functions in <math.h>, link with -lm

Pre-C99 functions [\[edit\]](#)

Name	Description
acos	inverse cosine
asin	inverse sine
atan	one-parameter inverse tangent
atan2	two-parameter inverse tangent
ceil	ceiling, the smallest integer not less than parameter
cos	cosine
cosh	hyperbolic cosine
exp	exponential function
fabs	absolute value (of a floating-point number)
floor	floor, the largest integer not greater than parameter
fmod	floating-point remainder: $x - y * (\text{int})(x/y)$
frexp	break floating-point number down into mantissa and exponent
ldexp	scale floating-point number by exponent (see article)
log	natural logarithm
log10	base-10 logarithm
modf(x,p)	returns fractional part of $x$ and stores integral part where pointer $p$ points to
pow(x,y)	raise $x$ to the power of $y$ , $x^y$
sin	sine
sinh	hyperbolic sine
sqrt	square root
tan	tangent
tanh	hyperbolic tangent]]

```
#include <stdio.h>
#include <math.h>
int main() {
    double a = 34.0;
    double b = sqrt(a);
    printf("%f + %f = %f \n", a, b);
    return 0;
}
```

```
c >gcc math1.c -lm; ./a.out
sqrt(34.000000) is 5.830952
c >
```

# Recursion

- Recursion is a powerful programming technique commonly used in divide-and-conquer situations.

```
[c >gcc recursion.c -o factorial;
[c >./factorial 3
factorial(3) is 6
[c >./factorial 4
factorial(4) is 24
[c >./factorial 10
factorial(10) is 3628800
c >]
```

Can you think how this program can go into an infinite loop?

```
#include <stdio.h>
#include <stdlib.h>
int factorial(int n);
int main(int argc, char **argv) {
    if (argc < 2) {
        printf("Program needs an integer argument\n");
        return(-1);
    }
    int n = atoi(argv[1]);
    int fact = factorial(n);
    printf("factorial(%d) is %d\n",n, fact);
    return 0;
}
int factorial(int n) {
    if (n == 1)
        return 1;
    else
        return n*factorial(n-1);
}
```

recursion1.c

i