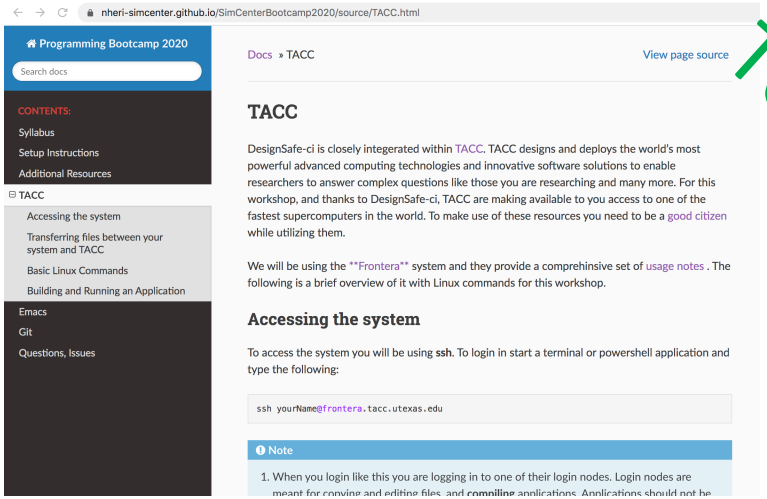
Reference Material

Website: https://nheri-simcenter.github.io/SimCenterBootcamp2020/index.html

TACC (login, linux, compile)

Emacs

Git



TODAY MIGHT NOT BE EASY!

Examples in your REPO: SimCenterBootcamp2020/code/c

CS50 2019 (Harvard)

C Intro: https://www.youtube.com/watch?v=e9Eds2Rc_x8
C Arrays: https://www.youtube.com/watch?v=8PrOp9t0PyQ
C Algorithms: https://www.youtube.com/watch?v=fykrlqbV9wM
C *MEMORY: https*://www.youtube.com/watch?v=cF6YkH-8vFk

SimCenter NHERI
Center for Computational Modeling and Simulation

2020 Programming Bootcamp

Introduction to C

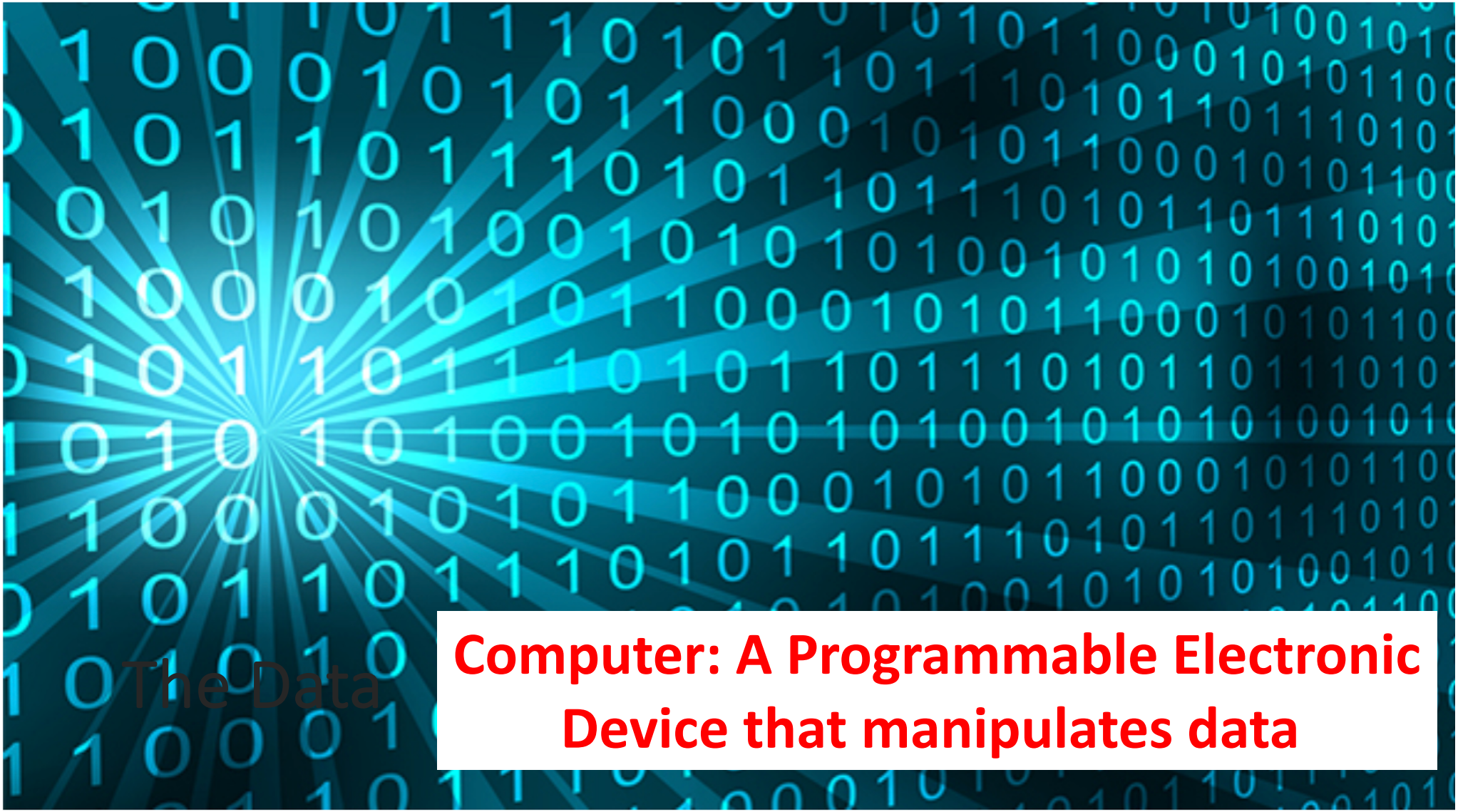
Frank McKenna
University of California at Berkeley

NSF award: CMMI 1612843

Overview

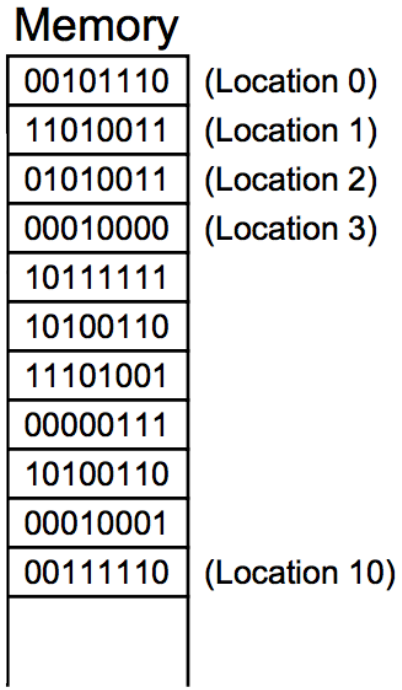
Computer Architecture & Programs

C


Computer: A Programmable Electronic Device that manipulates data
The Data

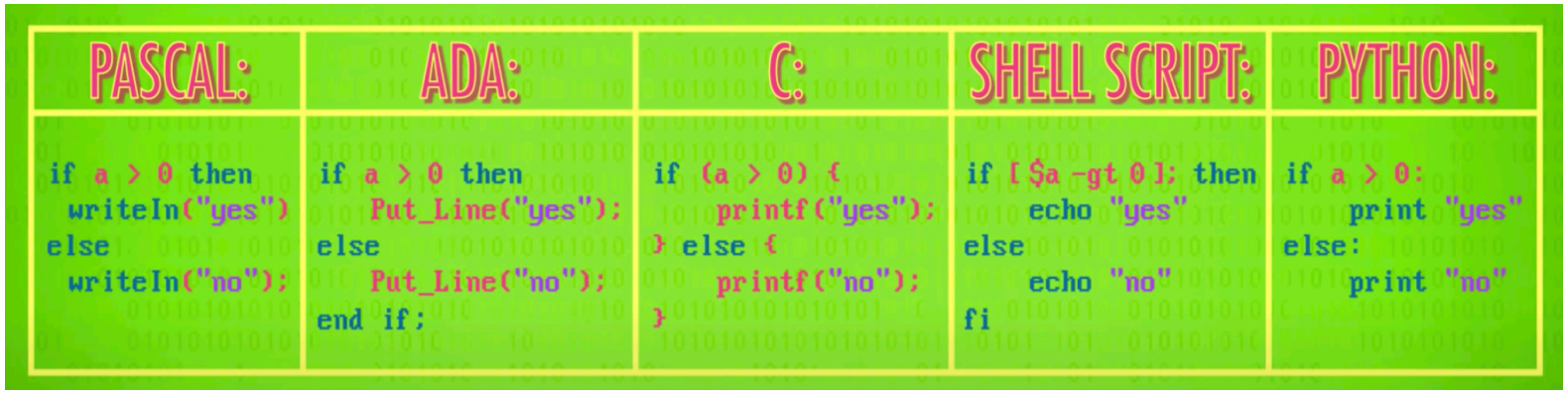
What is a Computer Program?

- A sequence of separate instructions one after another
- Each instruction tells CPU to do 1 small specific task
- When the instructions are completed the Computer has done something we wanted done.



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:

```
if a > 0 then
  writeln("yes")
else
  writeln("no");
```

ADA:

```
if a > 0 then
   Put_Line("yes");
else
   Put_Line("no");
end if;
```

C:

```
if (a > 0) {
   printf("yes");
} else {
   printf("no");
}
```

SHELL SCRIPT:

```
if [ $a -gt 0 ]; then
   echo "yes"
else
   echo "no"
fi
```

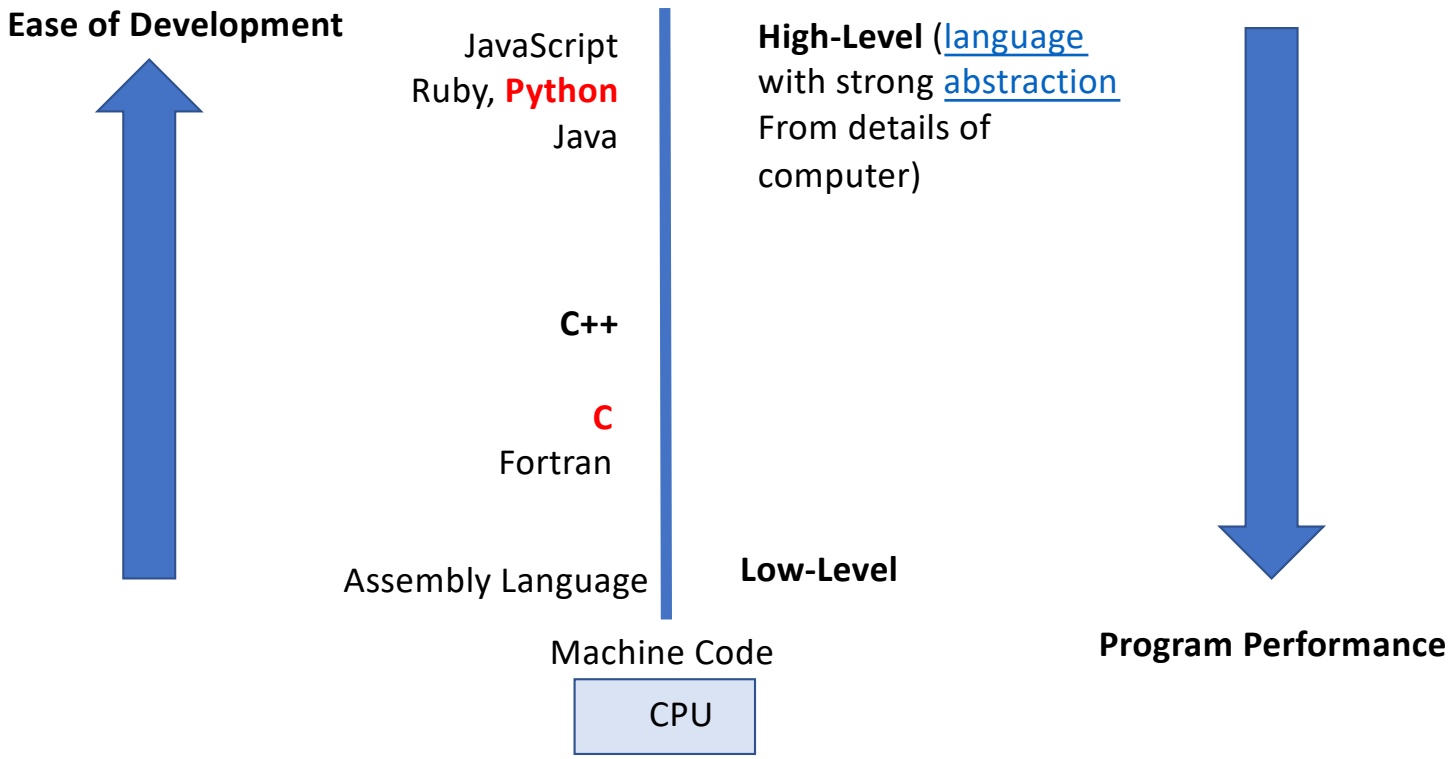
PYTHON:

```
if a > 0:
   print "yes"
else:
   print "no"
```

TCL

```
If ($a > 0} {
   puts “yes”
} else {
   puts “no”
}
```

Programming Language Hierarchy



Speed Comparison

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

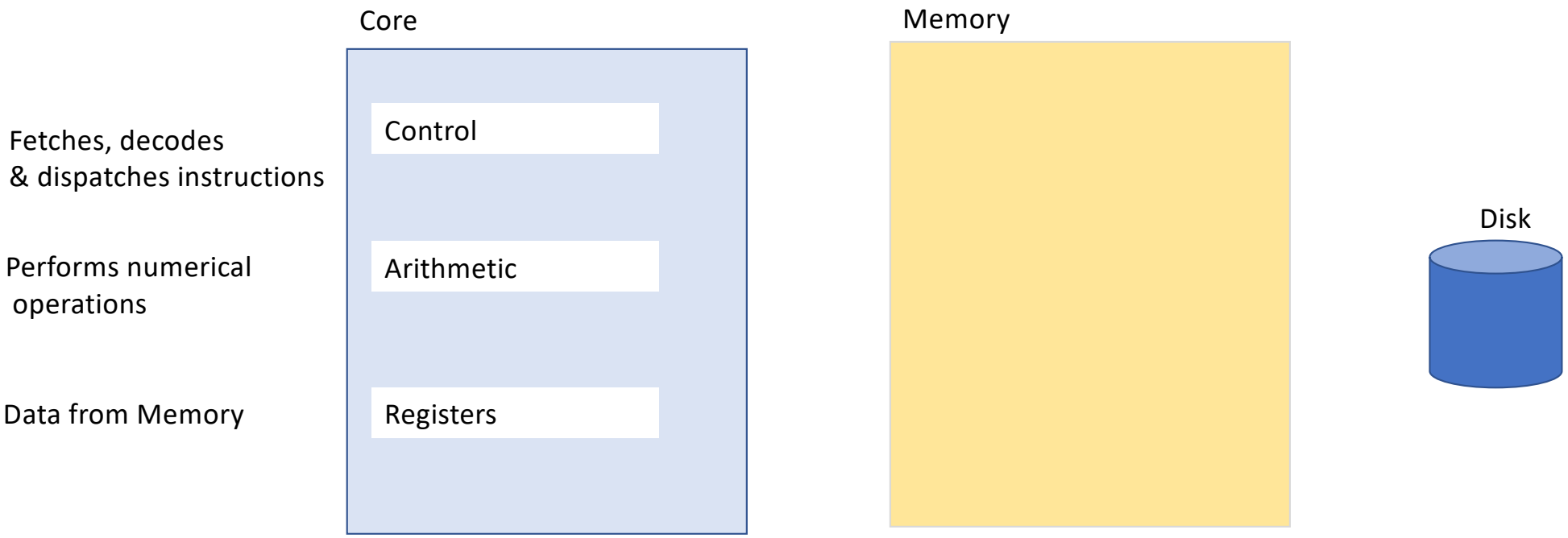
Charles E. Leiserson, Neil C. Thompson*, Joel S. Emer, Bradley C. Kuszmaul, Butler W. Lampson, Daniel Sanchez, Tao B. Schardl

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

Ultimate Performance Comes from writing code that takes advantage of HW

Single Processor Machine – Idealized Model



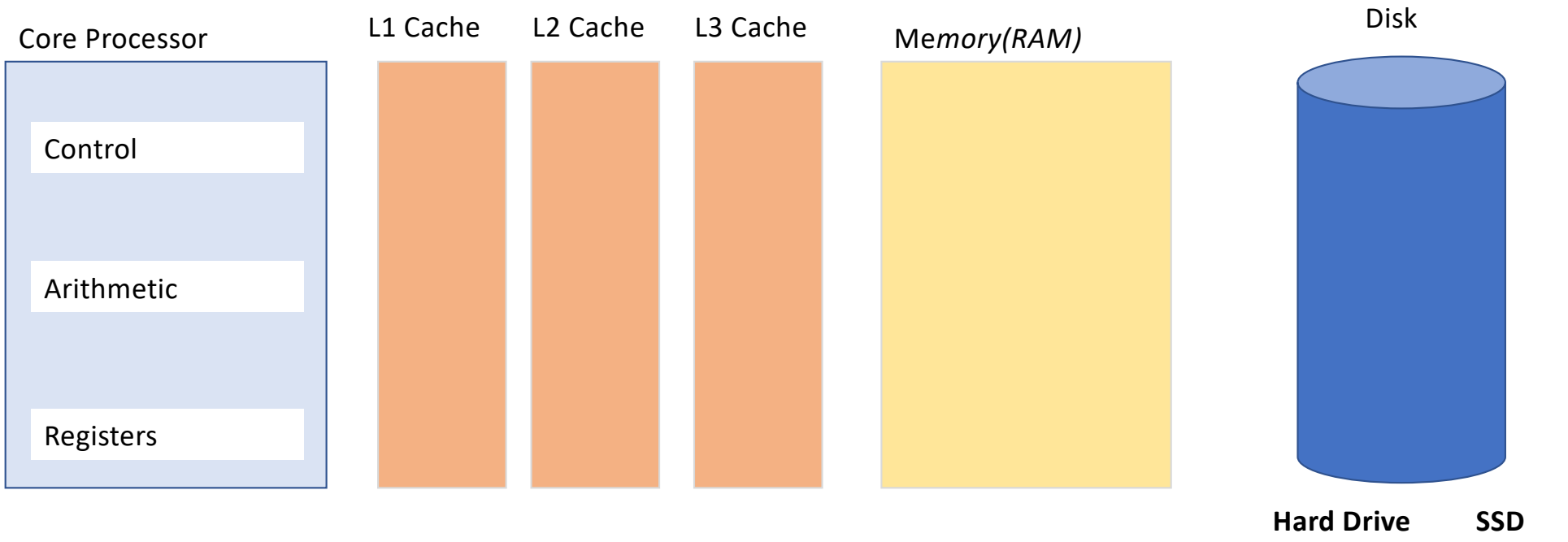
Cores only Work on Data in Registers

Which is Why Understanding Memory Hierarchy is Crucial

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

Memory Hierarchy



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	

What is Cache?

- Small, Fast Memory
- Placed Between Registers and Main Memory
- It keeps a copy of data in memory
- It is hidden from software (neither compiler or OS can say what gets loaded)

```
void main() {
   …
 load b into R2
   …
}
```

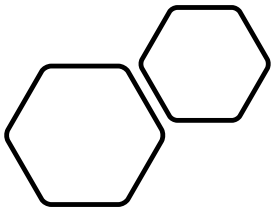
- Cache-hit: data in cache (b in cache)
- Cache-miss: data not in cache, have to go get from memory (b in memory)
- Cache-line-length: number of bytes of data loaded into cache with missing data (32 to 128bytes)

Why Do Caches Work?

- **Spatial Locality** – probability is high that if program is accessing some memory on 1 instruction, it is going to access a nearby one soon
- **Temporal Locality** – probability is high that if program is accessing some memory location it will access same location again soon.

```
int main() {
    …
   double dotProduct = 0
    for (int i=0, i<vectorSize; i++)
        dotProduct += x[i] * y[i];
    …
}
```

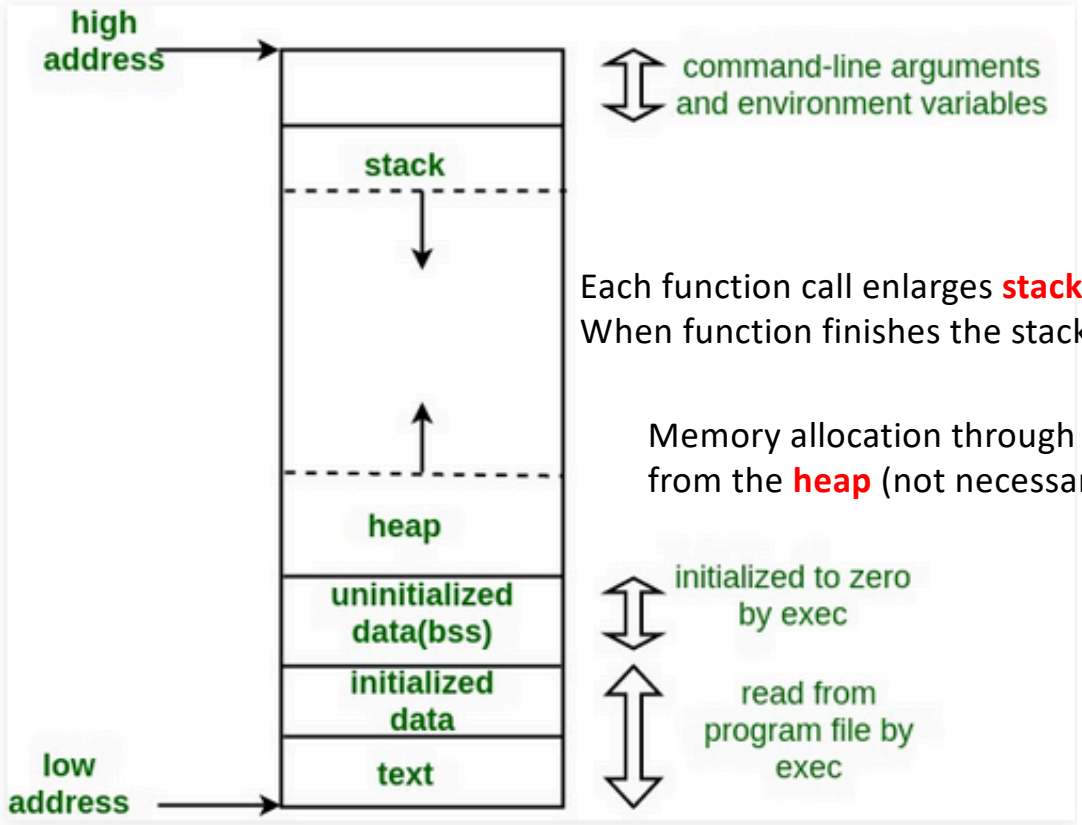
So Why Did I Bring Cache Up If You Have No Control Over It?



- Knowing caches exist, understanding how they work, allows you as a programmer to take advantage of them when you write the program, i.e. Allocate chunks of memory cache, think about where you store your variables, …

Program Memory – Main Memory Mismatch

64bit program – 128TB address range –> it does not quite fit in memory!

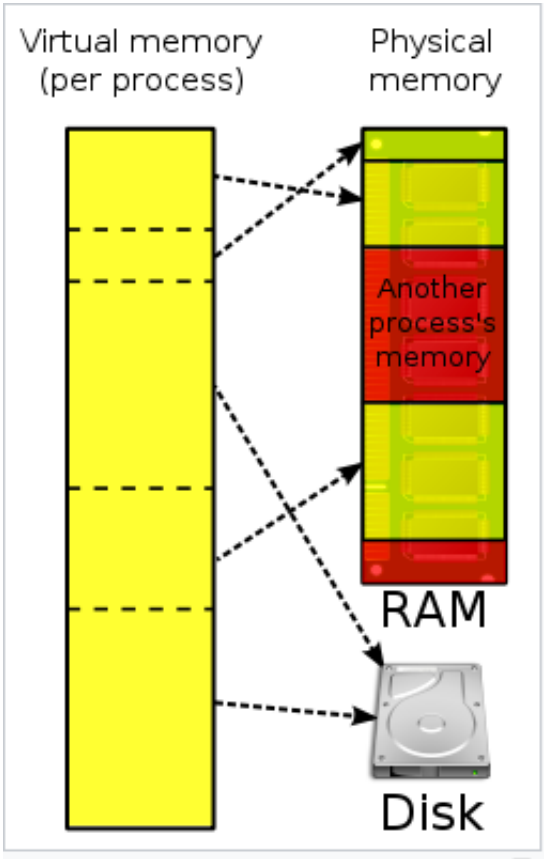


Memory(RAM)

Each function call enlarges **stack** with the function variables
When function finishes the stack is popped

Memory allocation through malloc() and new[] is allocated memory from the **heap** (not necessarily one chunk after another)

Virtual Memory



- Is a memory management technique that provides an "idealized abstraction of the storage resources that are actually available on a given machine" wikipedia.
- 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)
- 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.
- Page Table: table of pages in memory.
- Page Table Lookup – relativily expensive.
- Page Fault (page not in memory) very expensive as page must be brought from disk by OS
- Page Size: size of pages
- TLB Translation Look-Aside Buffer HW cache of virtual to physical mappings.
- Allows multiple programs to be running at once in memory.

Major page fault

- **Major => need to retrieve page from disk**
  1. CPU detects the situation (valid bit = 0)
     - It cannot remedy the situation on its own;
     - It doesn't communicate with disks (nor even knows that it should)
  2. CPU generates interrupt and transfers control to the OS
     - Invoking the OS page-fault handler
  3. OS regains control, realizes page is on disk, initiates I/O read ops
     - To read missing page from disk to DRAM
     - Possibly need to write victim page(s) to disk (if no room & dirty)
  4. OS suspends process & context switches to another process
     - It might take a few milliseconds for I/O ops to complete
  5. Upon read completion, OS makes suspended process runnable again
     - It'll soon be chosen for execution
  6. When process is resumed, faulting operation is re-executed
     - Now it will succeed because the page is there

Page Fault Takes a Boatload of Time

Overview

Computer Architecture & Programs

C

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)

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(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. It's implementation in the standard C library.
- The next statement **return 0;** terminates the main() function and returns the value 0.

Exercise: Compile & Run Hello World!

1. ssh yourname@frontera.tacc.utexas.edu
2. cd SimCenterBootcamp2020
3. mkdir myPrograms
4. cd myPrograms
5. mkdir hello
6. cd hello
7. emacs hello.c
8. <Control> x <Control> s
9. <Control> x <Control> c
10. icc hello.c
11. idev
12. ./a.out
13. exit
14. exit

```
#include <stdio.h>

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

A comment may also be Specified using a //. The compiler ignores all text from comment to EOL

Variables and Types

- Except in simplest of programs we need to keep track of data, e.g. current and max scores in a game, current sum in vector product calculation
- A **Variable** is a **name** a programmer can set aside for storing & accessing accessing a **memory location.**
- C is a **strongly typed language.** The programmer **must specify the data type associated with the variable.**

- Names are **made up of letters and digits;** they are **case sensitive;** names must **start with a character,** for variable names **'_' counts as a chracacter**
- **Certain keywords are reserved, i.e. cannot be used as variable names**

Reserved Keywords in C

C KEYWORDS OR RESERVED WORDS

auto	break	case	char
const	continue	default	do
int	long	register	return
short	signed	sizeof	static
struct	switch	typedef	union
unsigned	void	volatile	while
double	else	enum	extern
float	for	goto	if

Variable Example

var1.c

```
#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;
}
```

Uninitialized Variable

var2.c

```
#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;
}
```

Initialized Variable

Allowable Variable Types in C - I

char
int
float
double
void

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;
}
```

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

Arrays - I

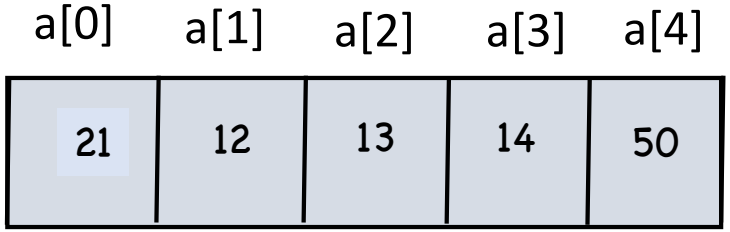
- A fixed size sequential collection of elements laid out in memory of the *same* type. We access using an index inside a square brackets, indexing start at 0
- to declare: type arrayName [size];

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

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 ;
 }
```

WARNING: indexing starts at 0

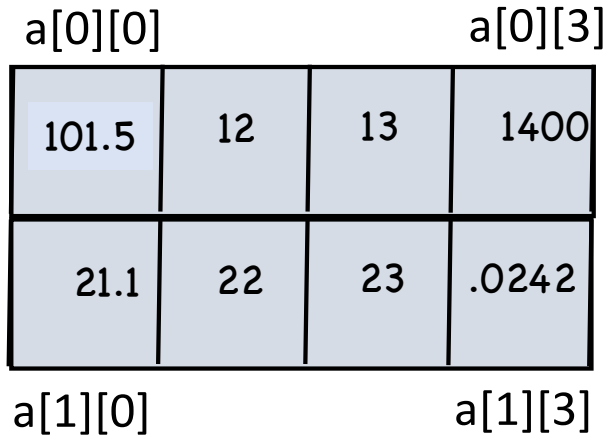


Multidimensional Arrays- I

- 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 var1.c
- to declare: type arrayName [l1][l2][l3]…;
  type arrayName [l1][l2][l3] = {l1*l2*… comma separated values

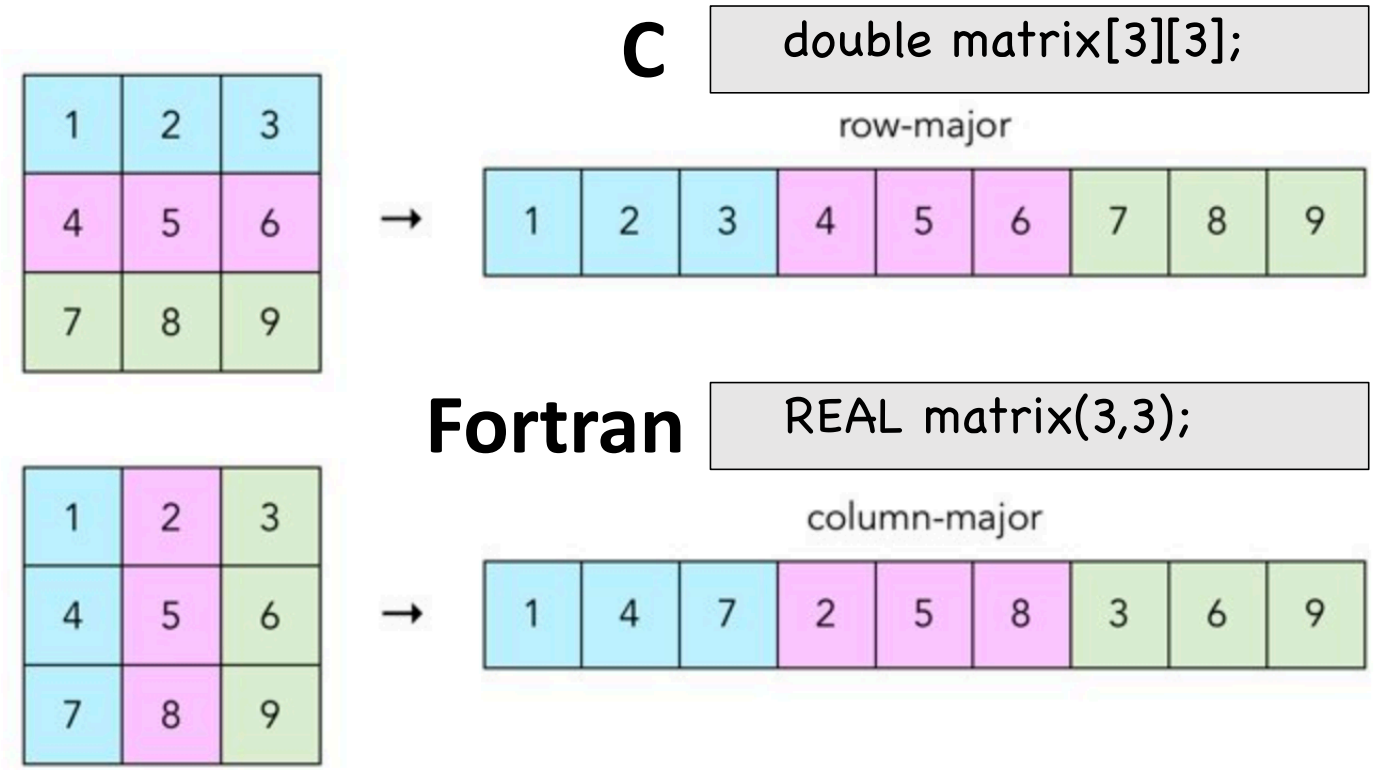
array2.c

```
#include <stdio.h>

int main(int argc, char **argv) {
   double dArray[2][4]= {{19.1, 12, 13, 14e2},
                         {21.1, 22, 23, 24.2e-3}};
  dArray[0][0] = 101.5;
  int first = dArray[0][0];
  int last = dArray[1][3];
  printf("First %f, last %f \n", first, last);
  return(0);
 }
```



NOT USED A LOT

Memory Layout of Arrays in C and Fortran



2 reasons to bring it up:

1. can use a 1 dimensional array
2. when calling fortran from c, you need to think about matrix (i.e. store in fortran format)

Operations

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

+, -, *, /, %

op1.c

```
#include <stdio.h>

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 –

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 >
```

C Operator Precedence Table

This page lists C operators in order of *precedence* (highest to lowest). Their *associativity* indicates in what order operators of equal precedence in an expression are applied.

Operator	Description	Associativity
( ) [ ] . -> ++ --	Parentheses (function call) (see Note 1) Brackets (array subscript) Member selection via object name Member selection via pointer Postfix increment/decrement (see Note 2)	left-to-right
++ -- + - ! ~ (*type*) * & sizeof	Prefix increment/decrement Unary plus/minus Logical negation/bitwise complement Cast (convert value to temporary value of *type*) Dereference Address (of operand) Determine size in bytes on this implementation	right-to-left
* / %	Multiplication/division/modulus	left-to-right
+ -	Addition/subtraction	left-to-right
<< >>	Bitwise shift left, Bitwise shift right	left-to-right
< <= > >=	Relational less than/less than or equal to Relational greater than/greater than or equal to	left-to-right
== !=	Relational is equal to/is not equal to	left-to-right
&	Bitwise AND	left-to-right
^	Bitwise exclusive OR	left-to-right
|	Bitwise inclusive OR	left-to-right
&&	Logical AND	left-to-right
||	Logical OR	left-to-right
? :	Ternary conditional	right-to-left
= += -= *= /= %= &= ^= |= <<= >>=	Assignment Addition/subtraction assignment Multiplication/division assignment Modulus/bitwise AND assignment Bitwise exclusive/inclusive OR assignment Bitwise shift left/right assignment	right-to-left
,	Comma (separate expressions)	left-to-right

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

if1.c

```
#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);
}
```

Conditional Operators

<
<=
>
>=
==
!=

If-else

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

if2.c

```
#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);
}
```

else-if

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

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

Logical 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);
}
```

&&

||

|

Conditional Code – switch statement

- Special multi-way decision maker that tests if an expression matches one of a number of **constant** values

```
switch(expression) {
  case constant-expression :
      statement(s);
      break;  /* optional */
 case constant-expression :
     statement(s);
     break; /* optional */
   .....
 default : /* Optional */
     statement(s);
}
```

```
#include <stdio.h>
int main(int argc, char **argv) {
  char c='Y';
  switch (c) {
    case 'Y':
    case 'y':
      c = 'y';
      break;
    default:
      printf("unknown character %c \n",c);
    }
  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,…

while1.c

```
#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++;   // 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
}
```

for1.c

```
#include <stdio.h>

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);
 }
```

```
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);
 }
```

Exercise: Code to count number of digits [0,9], white spaces (' ', '\n','\t') and other char in a file. Write info out.

```
#include <stdio.h>
int main(int argc, char **argv) {
  char c;
  int nDigit =0, nWhite =0, nOther = 0;
  while ((c = getchar()) != EOF) {
     // your code
  }
  // some more code here
}
```

1. emacs count.c
2. icc count.c
3. ./a.out < count.c

Exercise Advanced: There is a text file file elCentro.dat in SimCenterBootcamp2020/code. Write program to create another file with same information but reduced in size and you can't open file of type binary, i.e. no open file of type 'wb'

```
#include <stdio.h>
int main(int argc, char **argv) {
   // your code here
}
```

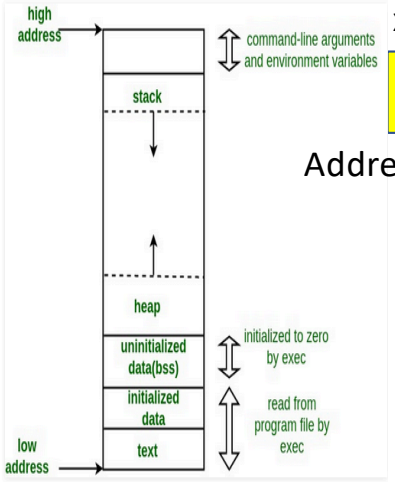
1. emacs count.c
2. icc count.c
3. ./a.out < count.c

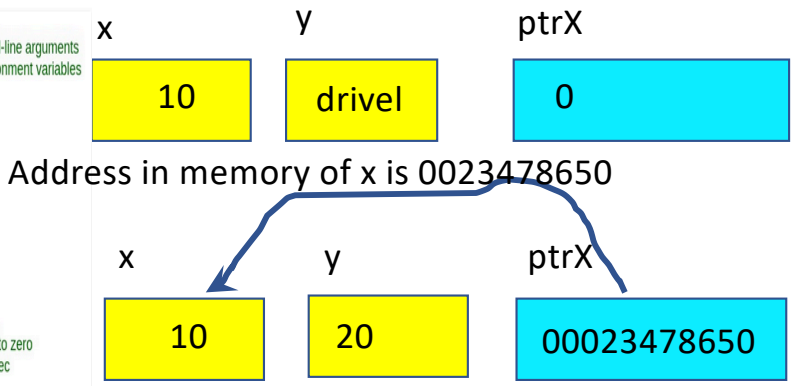
NOW COMES
THE HARD PART …….

Pointers & Addresses

- 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 also just setting aside a mem address and some name to refer to it.
- 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;
}
```






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

Functions

- **Art of Programming I**: *"To take a problem, and recursivily 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 the 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.

function1.c

```
#include <stdio.h>
    int *: data is a pointer to an int
// 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);
 }
```

function2.c

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

function3.c

```
#include <stdio.h>
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

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, matrix operations in another.
3. Put prototypes for all functions in another file.
4. If function large, put in separate file.
5. Get into a system of documenting inputs and outputs.

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;
}
```

Exercise: Write a function to sum two values

```
#include <stdio.h>
int sumInt(int a, int b);
int main() {
   int integer1, integer2, sum;
   printf("Enter first integer: ");
   scanf("%d", &integer1); // read input to integer 1
   printf("Enter second integer: ");
   scanf("%d", &integer2); // Read input into integer2
   sum = sumInt(integer1, integer2);
   printf("sum %d + %d = %d\n", integer1, integer2, sum);
   return(0);
   }
// your code here
int sumInt(int a, int b) {
    // your code here
}
```

&integer1: memory address of integer1

&integer2: memory address of integer2

1. emacs sumc
2. icc sum.c
3. ./a.out

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;
}
```

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*(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>
int main() {
double a = 34.0;
    double b = sqrt(a);
    print("%f + %f = %f \n", a, b)
    return 0;
 }
```

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

Scope of Variables

scope1.c

```
#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;
}
```

```
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 >
```

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?

recursion1.c

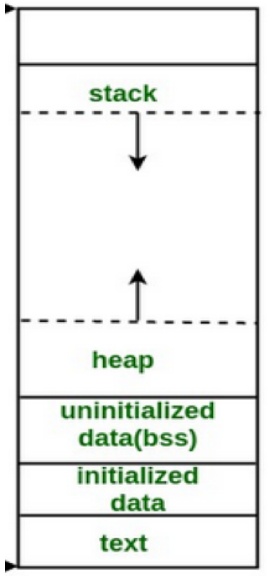
```
#include <stdio.h>
#include <stdlib.h>
int factorial(int n);
int main(int argc, char **argv) {
  if (argc < 2) {
    printf("Program needs an integer
argument\n");
    return(-1);
  }
  int n = atoi(argv[1]);
  int fact = factorial(n);
  printf("factorial(%d) is %d\n",n, fact);
  return 0;
}
int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

Arrays - II

- An array is 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}
- Works for arrays where we know the size at compile time. There are many times when we do not know the size of the array.
- Need to use **pointers** and functions **free()** and **malloc()**

```
type *thePointer = (type *)malloc(numElements*sizeof(type));
  ...
free(thePointer)
```



- Memory for the array using free() comes from the heap
- **Always remember to free() the memory** .. Otherwise can run out of memory.

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 >
```

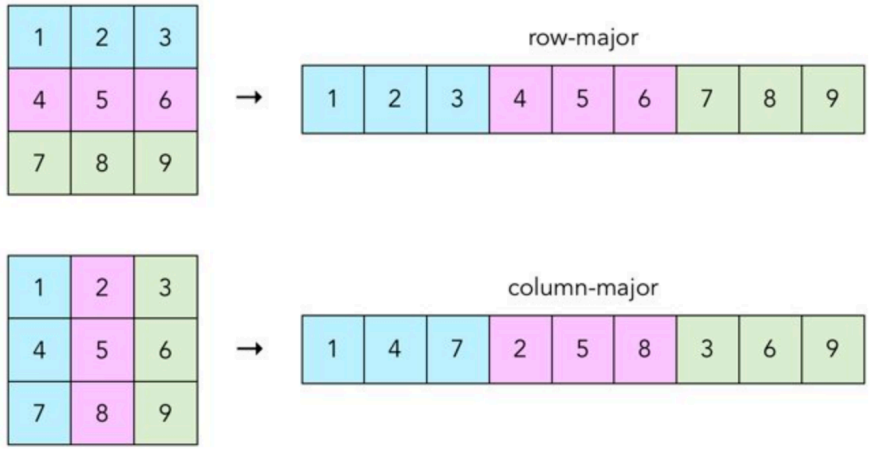
Pointers to pointers & multi-dimensional arrays

memory2.c

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

 int main(int argc, char **argv) {
  int n;
  double **matrix1 =0;

  printf("enter n: ");
  scanf("%d", &n);

  // allocate memory & set the data
  matrix1 = (double **)malloc(n*sizeof(double *));
  for (int i=0; i<n; i++) {
     matrix1[i] = (double *)malloc(n*sizeof(double));
     for (int j=0; j<n; j++)
        matrix1[i][j] = i;
  }
  for (int i=0; i<n; i++) {
     for (int j=0; j<n; j++)
        printf("(%d,%d) %.4f\n", i,j, matrix1[i][j]);
  }
  // free the data
  for (int i=0; i<n; i++)
      free(matrix1[i]);
  free(matrix1);
}
```

for Compatibility with many matrix libraries this is poor code:

```
double **matrix2 =0;
matrix2 = (double **)malloc(numRows*sizeof(double *));
for (int i=0; i<numRows; i++) {
    matrix2[i] = (double *)malloc(numCols*sizeof(double));
    for (int j=0; j<numCols; j++)
        matrix2[i][j] = i;
}
```



Because many prebuilt libraries work assuming continuous layout and Fortran column-major order:

```
double *matrix2 =0;
matrix2 = (double *)malloc(numRows*numCols*sizeof(double *));
for (int i=0; i<numRows; i++) {
    for (int j=0; j<numCols; j++)
        matrix2[I + j*numRows] =
}
```

```
double *matrix2 =0;
matrix2 = (double *)malloc(nu
for (int j=0; j<numCols; j++
    for (int i=0; i<numRows
        matrix2[I + j*numRo
}
```

```
double **matrix2 =0;
matrix2 = (double **)malloc(numRows*numCols*sizeof(double *));
double *dataPtr = matrix2;
for (int j=0; j<numCols; j++)
    for (int i=0; i<numRows; i++) {
        *dataPtr++ = i;
}
```

memory3.c

Special Problems: char * and Strings

- No string datatype, string in C is represented by type char *
- There are special functions for strings in <string.h>
  - strlen()
  - strcpy()
  - ….
- To use them requires a special character at end of string, namely **'\0'**
- This can cause no end of grief, e.g. if you use malloc, you need **size+1** and need to append '\0'

```
#include <string.h>
….
char greeting[] = "Hello";
int length = strlen(greeting);
printf("%s a string of length %d\n",greeting, length);

char *greetingCopy = (char *)malloc((length+1)*sizeof(char));
strcpy(greetingCopy, greeting);
```

WARNING

- Arrays and Pointers are the source of most bugs in C Code
  - You will have to use them if you program in C
  - Always initialize a pointer to 0
  - Be careful you do not go beyond the end of an array
    - Be thankful for segmentation faults
    - If you have a race condition (get different answers every time you run, probably a pointer issue)

What We Neglected

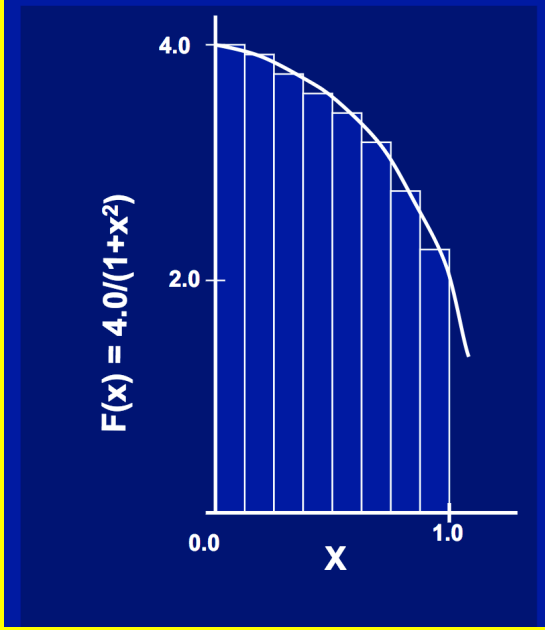
- File I/O
- Struct
- …. And some other stuff (not necessarily minor)
  - References
  - Operating on bits

Practice Exercises (1 hour): as many as you can

1. Write a program that when running prompts the user for two floating point numbers and returns their product.
   - i.e. ./a.out would prompt for 2 numbers a and b will output a * b = something
2. Write a program that takes a number of integer values from argc, stores them in an array, computes the sum of the array and outputs some nice message. Try using recursion to compute the sum. (hint start with recursion1.c and google function atoi(), copy from memory1.c)
   - i.e. ./a.out 3 4 5 6 will output 3 + *4 + 5 + 6 = 18*
3. *Taking the previous program. Modify it to output the number of unique numbers in the output.*
   - *i.e. ./a.out 3 1 2 1 will output 1*3 + 2*1 + 1* 2 = 7*
4. *Write a program that takes a number of input values and sorts them in ascending order.*
   - *I.e. ./a.out 2 7 4 5 9 will output* 2 4 5 7 9

Exercise: Compute PI

Numerical Integration



Mathematically, we know that:

$$\int_0^1 \frac{4.0}{(1+x^2)}\,dx = \pi$$

We can approximate the integral as a sum of rectangles:

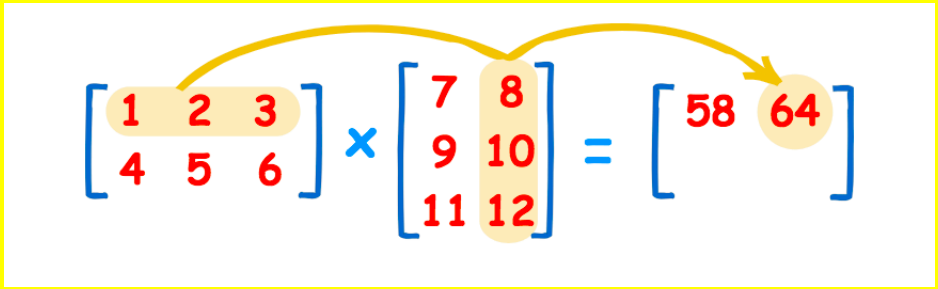
$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

Where each rectangle has width Δx and height $F(x_i)$ at the middle of interval i.

Source: UC Berkeley, Tim Mattson (Intel Corp), CS267 & elsewhere

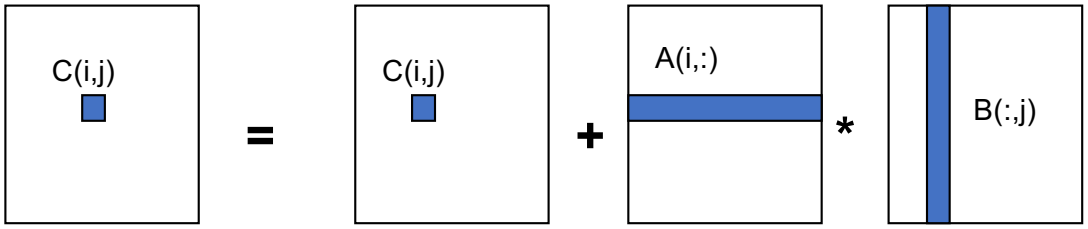
```
#include <stdio>
static int long numSteps = 100000;
int main() {
  double pi = 0; double time=0;
  // your code
  for (int i=0; i<numSteps; i++) {
    // your code
  }
   // your code
  printf("PI = %f, duration: %f ms\n",pi, time);
  return 0;
}
```

Exercise: Matrix-Matrix Multiply



Naïve Matrix Multiply

```
{implements C = C + A*B}
for i = 1 to n
  {read row i of A into fast memory}
   for j = 1 to n
       {read C(i,j) into fast memory}
       {read column j of B into fast memory}
       for k = 1 to n
           C(i,j) = C(i,j) + A(i,k) * B(k,j)
       {write C(i,j) back to slow memory}
```



Source: UC Berkeley, Jim Demmell, CS267

Blocked (Tiled) Matrix Multiply

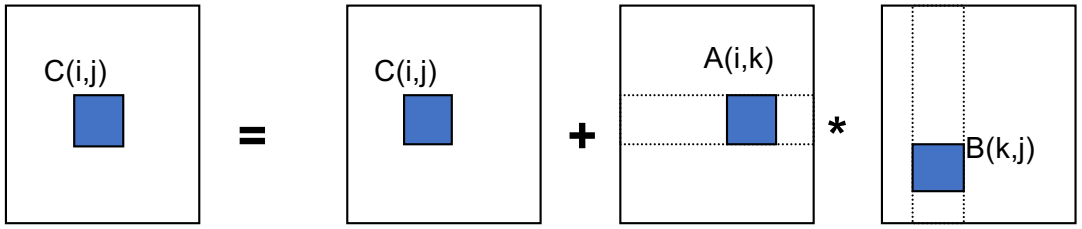
Consider A,B,C to be N-by-N matrices of b-by-b subblocks where b=n / N is block size

```
for i = 1 to N
  for j = 1 to N
    {read block C(i,j) into fast memory}
    for k = 1 to N
      {read block A(i,k) into fast memory}
      {read block B(k,j) into fast memory}
      C(i,j) = C(i,j) + A(i,k) * B(k,j) {do a matrix multiply on blocks}
    {write block C(i,j) back to slow memory}
```

cache does this automatically

3 nested loops inside

block size = loop bounds



Tiling for registers (managed by you/compiler) or caches (hardware)

Source: UC Berkeley, Jim Demmell, CS267