## Agenda – Day 1

<table>
<thead>
<tr>
<th>Time</th>
<th>Title</th>
<th>Presenter</th>
</tr>
</thead>
<tbody>
<tr>
<td>9:00-9:30</td>
<td>Welcome</td>
<td>Frank McKenna &amp; You</td>
</tr>
<tr>
<td>9:30-11:30</td>
<td>The VM, Linux &amp; Git</td>
<td>Peter Mackenzie-Helnwein</td>
</tr>
<tr>
<td>11:30-12:00</td>
<td>An Introduction To Programming</td>
<td>Frank McKenna</td>
</tr>
<tr>
<td>12.00-1:00</td>
<td>LUNCH</td>
<td></td>
</tr>
<tr>
<td>1:00-3:00</td>
<td>The C Programming Language</td>
<td>Frank McKenna</td>
</tr>
<tr>
<td>3:00-5:00</td>
<td>Exercises</td>
<td>You</td>
</tr>
<tr>
<td>Day 2</td>
<td>Debugging, Parallel Programming with MPI &amp; OpenMP</td>
<td></td>
</tr>
<tr>
<td>Day 3</td>
<td>Abstraction, More C &amp; C++</td>
<td></td>
</tr>
<tr>
<td>Day 4</td>
<td>User Interface Design &amp; Qt</td>
<td></td>
</tr>
<tr>
<td>Day 5</td>
<td>SimCenter &amp; Cloud Computing</td>
<td></td>
</tr>
</tbody>
</table>
An Introduction to Programming
&
The C Programming Language
Frank McKenna
Outline

- A Computer Program and the Computer on Which it Runs
- C Programming Language
  - Variables
  - Operations
  - Program Control
  - Functions
  - Pointers & Arrays
  - Other Things
What is a Computer Program?

- A sequence of separate instructions one after another
- Each instruction tells CPU to do 1 small specific task
Art of Programming - I

- To take a problem, and continually break it down into a series of smaller tasks until ultimately these tasks become a series of small specific individual instructions.
What is Programming?

• Writing these instructions as a series of statements.
• A statement use 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”
What Programming Language?

• Hundreds of languages ....
• Only a dozen or so are popular at any time
• We will be looking at C and C++
Types of Languages – Compiled/Interpreted

- Compiled:
  - CPU
  - Source Code File
    - if a<b (Lib ref) do while z=x-y (Lib ref)
  - Compiler
  - Object File
    - 11011001 (Lib ref) 00010011 10101011 (Lib ref)
  - Linker
  - Executable File
    - 11011001 01000100 00010111 10101011 11111100
  - Library Files
    - 10111101 11100001 00000011 01000100 10011101 11111100

- Interpreted:
  - CPU
  - Source Code File
    - if a<b (Lib ref) do while z=x-y (Lib ref)
  - Interpreter
  - Evaluates using CPU
    - 0011101110 0011100001

- 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

Ease of Development

High-Level

JavaScript
Ruby, Python
Java
C++

Low-Level

C
Fortran

Assembly Language

Machine Code

CPU

Program Performance
Single Processor Machine – Idealized Model

CPU
- Fetches, decodes & dispatches instructions
- Performs numerical operations
- Data from Memory

Memory

Disk

- Control
- Arithmetic
- Registers
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)

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

void man() {
  ...
  load a into R1
  load b into R2
  R3 = R1 + R2
  store R3 into c
  ...
  }

void man() {
  ...
  ...
  a = b + c;
  ...
  ...
}
CPU only Works on Data in Registers!
Memory Hierarchy

Processor
- Control
- Arithmetic
- Registers

L1 Cache
- Size: 64 KB
- Latency: 1 ns
- HW

L2 Cache
- Size: 256 KB
- Latency: 3-10 ns
- HW

L3 Cache
- Size: 2-4 MB
- Latency: 20-30 ns
- HW

Memory (RAM)
- Size: 4-16 GB
- Latency: 50-100 ns
- Operating System

Disk
- Size: 4-16 TB
- Latency: 5-10^6 ns
- Operating System

Compiler
- Size: 1000 Bytes
- Latency: 0.3 ns


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)

```c
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.

```c
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 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
Program Memory – Main Memory Mismatch

Memory(RAM)
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
The C Programming Language

- Originally Developed by Dennis Ritchie at Bell Labs in 1969 to implement a UNîx 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 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

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. With a text editor create the file hello.c in a terminal window type: `gedit hello.c`
2. Compile it in a terminal window type: `gcc hello.c`
3. Run it in a terminal window type: `./a.out`

```c
#include <stdio.h>

int main() {
    // 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 character

• Certain keywords are reserved, i.e. cannot be used as variable names
Reserved Keywords in C

<table>
<thead>
<tr>
<th>C Keywords or Reserved Words</th>
</tr>
</thead>
<tbody>
<tr>
<td>auto</td>
</tr>
<tr>
<td>const</td>
</tr>
<tr>
<td>int</td>
</tr>
<tr>
<td>short</td>
</tr>
<tr>
<td>struct</td>
</tr>
<tr>
<td>unsigned</td>
</tr>
<tr>
<td>double</td>
</tr>
<tr>
<td>float</td>
</tr>
</tbody>
</table>
Variable Example

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

Uninitialized Variable

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

Initialized Variable

var1.c

var2.c
Allowable Variable Types in C - I

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

int main(int argc, const 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

<table>
<thead>
<tr>
<th>Type</th>
<th>Size</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1 byte</td>
<td>-128 to 127 or 0 to 255</td>
</tr>
<tr>
<td>unsigned char</td>
<td>1 byte</td>
<td>0 to 255</td>
</tr>
<tr>
<td>signed char</td>
<td>1 byte</td>
<td>-128 to 127</td>
</tr>
<tr>
<td>int</td>
<td>2 or 4 bytes</td>
<td>-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647</td>
</tr>
<tr>
<td>unsigned int</td>
<td>2 or 4 bytes</td>
<td>0 to 65,535 or 0 to 4,294,967,295</td>
</tr>
<tr>
<td>short</td>
<td>2 bytes</td>
<td>-32,768 to 32,767</td>
</tr>
<tr>
<td>unsigned short</td>
<td>2 bytes</td>
<td>0 to 65,535</td>
</tr>
<tr>
<td>long</td>
<td>4 bytes</td>
<td>-2,147,483,648 to 2,147,483,648</td>
</tr>
<tr>
<td>unsigned long</td>
<td>4 bytes</td>
<td>0 to 4,294,967,295</td>
</tr>
</tbody>
</table>

2. Floating Point Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Size</th>
<th>Range</th>
<th>Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>4 byte</td>
<td>1.2E-38 to 3.4E+38</td>
<td>6 decimal places</td>
</tr>
<tr>
<td>double</td>
<td>8 byte</td>
<td>2.3E-308 to 1.7E+308</td>
<td>15 decimal places</td>
</tr>
<tr>
<td>long double</td>
<td>10 byte</td>
<td>3.4E-4932 to 1.1E+4932</td>
<td>19 decimal places</td>
</tr>
</tbody>
</table>

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}
  ```

```c
#include <stdio.h>

int main(int argc, const 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

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>50</td>
</tr>
</tbody>
</table>
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

- to declare: `type arrayName [l1][l2][l3]...;

  `type arrayName [l1][l2][l3] = {l1*l2* ... comma separated values`
Memory Layout of Arrays in C and Fortran

C

double matrix[3][3];

row-major

Fortran

REAL matrix(3,3);

column-major
Operations

• We want to do stuff with the data, to operate on it

• Basic Arithmetic Operations
  +, -, *, /, %

```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);
}
```

```c
op1.c
```
You Can String Operations Together –

```c
#include <stdio.h>
int main(int argc, const 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 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.

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

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

+=  
-=  
*=  
/=  
++  
--  

```c
a += 1;
```
Conditional Code – if statement

• 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

```c
#include <stdio.h>
int main(int argc, const 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 (condition) {
    // code block
}

If-else

```c
#include <stdio.h>

int main(int argc, const 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

```c
#include <stdio.h>

int main(int argc, const 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

```c
#include <stdio.h>

int main(int argc, const char **argv) {
    int a = 15;
    if (condition) {
        // code block
    } else if (condition) {
        // another code block
    } else {
        // and another
    }
}
```
Logical and/or/not

```c
#include <stdio.h>
int main(int argc, const 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

```c
#include <stdio.h>
int main(int argc, const char **argv) {
    char c='Y';
    switch (c) {
        case 'Y':
        case 'y':
            c = 'y';
            break;
        default:
            printf("unknown character %c \n",c);
            break;
    }
    return(0);
}
```

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

• 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,...

```c
#include <stdio.h>

int main(int argc, const 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

```c
#include <stdio.h>

int main(int argc, const 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 loop – multiple init & increment

```c
#include <stdio.h>

int main(int argc, const 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, white spaces (' ', '
', '	') and other char in a file. Write info out.

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

1. gedit count.c
2. gcc hello.c
3. ./a.out << count.c
Pointers & Addresses  
(before I start using them in examples)

- 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 setting 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.

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

```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, const 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);
}
```
```c
#include <stdio.h>

int sumArray(int *arrayData, int size) {
    int sum = 0;
    for (int i = 0; i < size; i++) {
        sum += *arrayData++;
    }
    return sum;
}

int main(int argc, const 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);
}
```

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

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;
}

myVector.h
int sumArray(int *arrayData, int size);
int productArray(int *arrayData, int size);
int normArray(int *arrayData, int size);
int dotProduct(int *array1, int *array2, int size);
Exercise: Write a function to sum two values

```c
#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);
}

1. gedit sumc
2. gcc sum.c
3. ./a.out

\&integer1: memory address of integer1
\&integer2: memory address of integer2
```
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.

```c
#include <stdio.h>
sumInt(int1, int2, &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);
    printf("%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

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>acos</td>
<td>inverse cosine</td>
</tr>
<tr>
<td>asin</td>
<td>inverse sine</td>
</tr>
<tr>
<td>atan</td>
<td>one-parameter inverse tangent</td>
</tr>
<tr>
<td>atan2</td>
<td>two-parameter inverse tangent</td>
</tr>
<tr>
<td>ceil</td>
<td>ceiling, the smallest integer not less than parameter</td>
</tr>
<tr>
<td>cos</td>
<td>cosine</td>
</tr>
<tr>
<td>cabs</td>
<td>absolute value (of a floating-point number)</td>
</tr>
<tr>
<td>floor</td>
<td>floor, the largest integer not greater than parameter</td>
</tr>
<tr>
<td>fmod</td>
<td>floating-point remainder: ( x - y \times \lfloor x/y \rfloor )</td>
</tr>
<tr>
<td>frexp</td>
<td>break floating-point number down into mantissa and exponent</td>
</tr>
<tr>
<td>ldexp</td>
<td>scale floating-point number by exponent (see article)</td>
</tr>
<tr>
<td>log</td>
<td>natural logarithm</td>
</tr>
<tr>
<td>log10</td>
<td>base-10 logarithm</td>
</tr>
<tr>
<td>modf(x,p)</td>
<td>returns fractional part of x and stores integral part where pointer p points to</td>
</tr>
<tr>
<td>pow(x,y)</td>
<td>raise x to the power of y, ( x^y )</td>
</tr>
<tr>
<td>sin</td>
<td>sine</td>
</tr>
<tr>
<td>sinh</td>
<td>hyperbolic sine</td>
</tr>
<tr>
<td>sqrt</td>
<td>square root</td>
</tr>
<tr>
<td>tan</td>
<td>tangent</td>
</tr>
<tr>
<td>tanh</td>
<td>hyperbolic tangent</td>
</tr>
</tbody>
</table>

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

```
gcc math1.c -lm; ./a.out
sqrt(34.0000000) is 5.830952
```

```shell
c >
```
Scope of Variables

```c
#include <stdio.h>
int sum(int, int);
int x = 20; // global variable
int main(int argc, const 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
#include <stdio.h>
#include <stdlib.h>
int factorial(int n);
int main(int argc, const 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);
}
```

- Try it! 
  - gcc recursion.c -o factorial
  - ./factorial 3
    factorial(3) is 6
  - ./factorial 4
    factorial(4) is 24
  - ./factorial 10
    factorial(10) is 3628800
  - ./factorial 20
    factorial(20) is 2432902008176640000

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];`
  
```c
  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()`

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

int main(int argc, const 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);
}
```

```
Pointers to pointers & multi-dimensional arrays

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

int main(int argc, const 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
", 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:

```c
double **matrix2 = 0;
matrix2 = (double **)malloc(numRows*numCols*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:

```c
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] = i;
}
```
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’

```c
#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 atof(), copy from memory1.c)
   • i.e. ./a.out 3 4 5.5 6 will output 3 + 4 + 5.5 + 6 = 18.5

3. Taking the previous program. Modify it to output the number of unique numbers in the output.
   • i.e. ./a.out 3 1.1 2.0 1.1 will output 3 + 2*1.1 +2.0 = 7.7

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

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 \( \Delta x \) 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

\[
\begin{bmatrix}
1 & 2 & 3 \\
4 & 5 & 6
\end{bmatrix}
\times
\begin{bmatrix}
7 & 8 \\
9 & 10 \\
11 & 12
\end{bmatrix}
=
\begin{bmatrix}
58 & 64
\end{bmatrix}
\]
Naïve Matrix Multiply

\{\text{implements } C = C + A \times B\}\n
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 \(n\)
\(C(i,j) = C(i,j) + A(i,k) \times B(k,j)\)
\{write \(C(i,j)\) back to slow memory\}

Source: UC Berkeley, Jim Demmell, CS267
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}
            $C(i,j) = C(i,j) + A(i,k) \times B(k,j)$ \{do a matrix multiply on blocks\}
        {write block $C(i,j)$ back to slow memory}

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

Source: UC Berkeley, Jim Demmell, CS267
Recursive Matrix Multiplication (RMM) (1/2)

\[ C = \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix} = A \cdot B = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \]

\[ = \begin{pmatrix} A_{11} \cdot B_{11} + A_{12} \cdot B_{21} & A_{11} \cdot B_{12} + A_{12} \cdot B_{22} \\ A_{21} \cdot B_{11} + A_{22} \cdot B_{21} & A_{21} \cdot B_{12} + A_{22} \cdot B_{22} \end{pmatrix} \]

- True when each block is a 1x1 or \( n/2 \times n/2 \)
- For simplicity: square matrices with \( n = 2^m \)
  - Extends to general rectangular case

Source: UC Berkeley, Jim Demmel, CS267
Recursive Matrix Multiplication (2/2)

```python
func C = RMM (A, B, n)
    if n=1, C = A * B, else
    {  C_{11} = RMM (A_{11}, B_{11}, n/2) + RMM (A_{12}, B_{21}, n/2)
      C_{12} = RMM (A_{11}, B_{12}, n/2) + RMM (A_{12}, B_{22}, n/2)
      C_{21} = RMM (A_{21}, B_{11}, n/2) + RMM (A_{22}, B_{21}, n/2)
      C_{22} = RMM (A_{21}, B_{12}, n/2) + RMM (A_{22}, B_{22}, n/2)  }
    return
```

A(n) = # arithmetic operations in RMM( . , . , n)
= 8 · A(n/2) + 4(n/2)^2 if n > 1, else 1
= 2n^3 ... same operations as usual, in different order

W(n) = # words moved between fast, slow memory by RMM( . , . , n)
= 8 · W(n/2) + 4 · 3(n/2)^2 if 3n^2 > M_{fast} , else 3n^2
= O( n^3 / (M_{fast})^{1/2} + n^2 ) ... same as blocked matmul

Don’t need to know M_{fast} for this to work!

Source: UC Berkeley, Jim Demmel, CS267