LECTURE 1

- C is evolving
    - C99 is the one that is most focused on, but C23 is the most recent version
- Half of the lecture is C, the other half is about systems
- Design principles
    - It is simple, easy to use/compile, low-level, and best for embedded controllers + OS
        - BUT it is also powerful and fast
- Paradigm of C
    - Purely procedural language
    - Object of interest is computations through procedures and functions
        - caller/callee system
        - Can abstract away the method that things are done
    - Programming in C = organizing processes as procedures and composing processes through procedure calls
        - The programmer is fully in charge of C programs
            - Can expose lower-level details easily and can help manage memory easily
            - Leads to more efficient programs than other languages
- Main function = the entry point of the function
    - Void = no args
    - Int before main means func returns an integer
- printf() is the print function
    - Takes string as argument
    - n/ is a newline char
- Returning from main shows how the program ran to the OS
    - Method of reporting back → 0 means it went smoothly
- #include imports a header file with the functions that need to be used in the program
    - Ex. printf
- Compile (gcc) process
    - Preprocesses the program
    - Compiles the program into another object
    - Links the object to linc
    - Writes the executable file a.out
- A.out is the default name of the executable
    - Can be changed through, and ./ shows that a.out is in the current directory
        - If not, the OS searcher directories in PATH

A SECOND PROGRAM

- Purpose
    - Reads an integer from the input n
    - Computes the sum of all integers between 1 and n
    - Prints the results of the calculation on the standard input
- New concept

- Standard input and output

LECTURE 2

- Expressions are inductively defined
  - Divided into constants, variables, and function calls
  - Combined through parentheses and operators
- Every C expression has a type
- Adding a semicolon to an expression turns it into a statement

- Basic data types
  - Int = integer
  - Char = single byte that stores a character is ASCII
    - Byte = 8 bits
  - Float = floating point numbers (decimals)
- Constants cannot be changed

Constants (of basic types)

```
// Constants cannot be changed
// char
'a', 'b', '\n'

// integer (note that compiler stores them in binary)
200, -34
0x7fffFFFF // hex
07112      // octal

// floating point numbers
3.1415, -0.34, 1.3E20
```

- Variables
  - Must be declared and initalized before use
    - Specifies they type and name
      - Helps the compiler allocate memory based on the type of variable
    - Names consist of *case sensitive* letters, digits, and ‘’_” but cannot *start* with a digit
    - Can be initialized when declared or can use separate assignments
- Operators
  - Conventional arithmetic, bitwise, and logic operators
  - Can be pre and post increment and decrement
  - Can be both simple and compound
- Precedence and associativity
  - Like the order of operations → determines which operation is done first

- Operators with the same precedence uses associativity (do it from either left ro right or right to left)
- Parentheses makes operations inside it first priority

Precedence and associativity

- Precedence determines which operation is done first
  - If operators have the same precedence, use associativity
  - Use parentheses

```
i + j * 10 - k / 20
(i + (j * 10)) - (k / 20)
```

Most ↓ Least

Operator precedence and associativity	
Operator	Associativity
() ++ (*postfix*) -- (*postfix*)	left to right
+ (*unary*) - (*unary*) ++ (*prefix*) -- (*prefix*)	right to left
* / %	left to right
+ -	left to right
= += -= *= /= etc.	right to left

7

- Assignment operators
  - LHS is the expression
    - Something that can be written to something else
  - LHS and the expression have compatible types
    - Value of expression is assigned to LHS and is the value of the assignment operation
- Compound Assignment Operators
  - var op= expr is the same as var = var op expr
- NOTE: Assignment *ARE NOT* Statements
  - They are expressions and the equals sign is an operator
    - You can change them or use them inside larger expressions
- Integer Data Types
  - The amount of bytes each type takes depends on the CPU and compiler

Integer Data Types

• Consider x86_64 (64-bit architecture) (what we all have)

size (in bits)	signed	unsigned
8	char -128 .. 127	unsigned char 0..255
16	short -32768..32767	unsigned short 0..65535
32	int -2^{31} .. $2^{31}-1$	unsigned int $0..2^{32}-1$
64	long -2^{63} .. $2^{63}-1$	unsigned long $0..2^{64}-1$
64	long long -2^{63} .. $2^{63}-1$	unsigned long long $0..2^{64}-1$

11

- Compared with 32-bit, the long and unsigned long would have 2 raised to 32 instead of 64

- Determining space
  - Operator sizeof gives the number of bytes needed for either a type or variable
    - Needed for dynamically allocating space
- Character Data Type
  - Each char has 8 bits (1 byte) of information
  - Encoded in ASCII
    - Each character is mapped to an integer between 0 and 127 inclusive
    - ASCII characters can be stored in char
  - Classes in ASCII
    - 0 to 31 inclusive is a control character (non-printable)
    - 48 to 57 inclusive is digits
    - 65 to 90 inclusive is uppercase letters
    - 97 to 122 inclusive is lowercase letters
  - Setting h1 = ‘H’ and h2 = 72 means that h1 and h2 have the same value

Non-printable characters?

- These are sometimes useful
  - Showing the constant (literal)

`'\n'`	newline
`'\r'`	carriage-return
`'\f'`	form-feed
`'\t'`	tabulation
`'\b'`	backspace
`'\x7'`	audible bell (x indicates hexadecimal)
`'\07'`	audible bell (0 indicates octal)

16

- Floating Points are 4 of 8 bytes, depending on the size that is used (32 or 64 bit)

size (in bits)	size (bytes)	Name & Range
32	4	**float** $1.17 * 10^{-38}$ to $3.40 * 10^{+38}$
64	8	**double** $2.22 * 10^{-308}$ to $1.79 * 10^{+308}$
80/128	16	**long double** $3.65 * 10^{-4951}$ to $1.18 * 10^{+4932}$

- Format specifiers tells the compiler about the type of data that needs to be printed or scanned

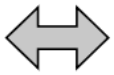
Format specifiers	Description
%d	int
%f	float
%c	char
%ld	long
%lf	double
%o	Octal representation
%x	Hexadecimal representation

- Automatic type conversion
  - Operands are automatically converted to a common type by the compiler if there are multiple different types of operands

        - Lower rank operands are converted to the type of the higher rank before any operations
        - char < short < int < long < long long < float < double < long double
            - Order for conversion
    - Automatic conversion can happen across assignments as well
        - Value of expression on the right-hand side can be widened or narrowed → narrowed means there could be some information loss
- Type casting
    - Helps convert an operand to another type before an operation
        - (<Type>)<expression> is the general format for this

Example: integer or double?

```
int x = 10;
int y = 3;

double z = x / y;
```



```
int x = 10;
int y = 3;

double z = (double)x / y;

// the following doesn't work
// z = (double)(x / y)
```

- In this case, setting double z to the double of the expression x/y is the same as having the solution of x/y to the double z

- What about booleans?
    - K&R and C89/C90 do not have a Boolean data type
        - 0 is FALSE, and anything else is TRUE
        - Common to use into or char to store Boolean values and define convenience macros
    - C99 introduced the _Bool variable, which is either 0 or 1
- Caution
    - Results may not be as expected

Examples

```
unsigned int x = 3;
unsigned int y = 7;
unsigned int z = x - y;
```

z holds the binary representation of -4, but reading it as an unsigned int yields a very different value!

```
_Bool b1;
char  b2, b3;
int   i = 256; // 0x100

b1 = i;
b2 = i;
b3 = i != 0;
```

b1 is 1 because i is not 0
b2 is 0 because lowest 8 bits in i are 0
b3 is 1 because i is not 0

Do you want b2 or b3?

22

- Examples: char consonants
    - Single quote marks is for only one character

```
'h'
'\n'
'\007'      // octal.
'\xAA'      // hex. 170 = 0xAA
'\''        // single quotation mark
'\\'        // back slash
'"'         // no need to escape double quotation mark here
```

- Examples: integer and floating-point constants

```
200
-300
0x7fffFFFFu  // hex. unsigned int. case insensitive
0123456      // octal. starting with 0!
0x12345678L  // hex. long int
123UL        // unsigned long
123LL        // long long
1234567890123456789ull // unsigned long long
3.14f          // float literals
3.14L          // long double literals
```

- Integer promotion
    - Integer types that are smaller than int (ex. char or short) are promoted to either into or unsigned int when an operation is performed on them

```
// If there is no integral promotion,
// c1 * c2 would not have 600 as result
char r, c1, c2, c3;
c1 = 100;
c2 = 6;
c3 = 8;
r = c1 * c2 / c3;
```

- Bitwise operators
    - All operators can be suffixed with an equals sign
        - a &= b is the same as a = a & b

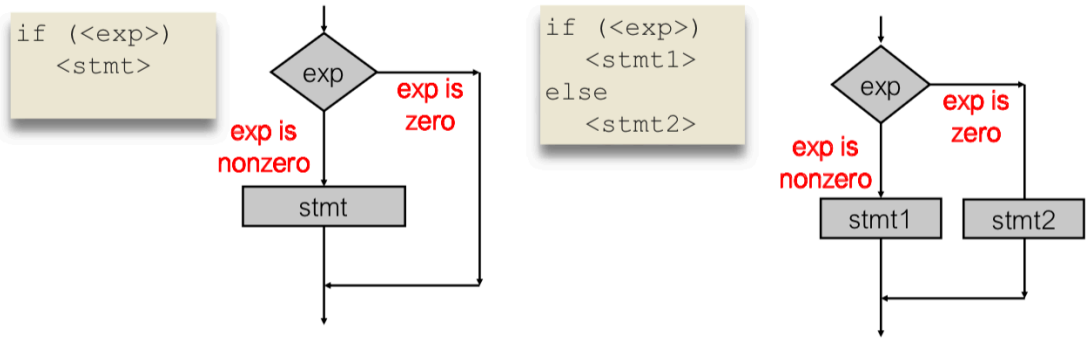
op.	Description	Example
&	bitwise AND	Set bits to 0. Mask out bits
|	bitwise OR	Set bits to 1
^	bitwise XOR	Flip some bits (using masks)
~	1's complement	Flip all bits
<<	Shift left	Move bits to left.
>>	Shift right	Move bits to right (pay attention to the sign)

LECTURE 3

- The output for the first printf is False
- Modulo in Python and C
    - 5 % 2 = 1 in Python and C
    - -5 % 2 = -1 in C and 1 in Python
    - 5 % -2 = -1 in C and Python
    - Difference is becaus

- Flow of Control
    - Let actions happen based on if certain conditions are met
        - If and If/Else are used for this
- Compound Statements/Blocks in C
    - List of statements in curly brackets
        - Single statement according to C
        - Can be empty or nested
    - Helpful for branching statements
    - Define variables at the beginning of blocks as needed
        - Mixing declarations and code is possible
- Comparison and logical operators
    - Comparison operators compare two expressions and provide True/False
        - ==, !=, >, <, >=, <=
    - Logical operators have a similar function
        - && || !
    - Result is 0 for False and 1 for True

- Branching: If and If/Else



  - "exp" is a comparison or logical expression most of the time, but can be ANY expression as needed
    - Can be compound statements as well
    - Nested ifs can be used as well
      - Make sure about avoiding dangling elses

Example: min

```
int  i, j, min;

if (i < j)
    min = i;
else
    min = j;
// Indentation is not required, above 4 lines are the same as
if (i < j) min = i; else min = j;
```

  - Program that prints out the minimum number between two variables
    - Note that indentation isnt required *but* aids in readability

Example: if-else statement with blocks

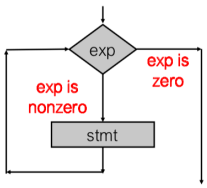
```
int  i, j, k;

if (i < j) {
    k = i;
    printf("i is selected.\n");
}     // no ; here
else {
    k = j;
    printf("j is selected.\n");
}
```


- If-else with blocks (shown in the curly brackets), so no need for the semicolons at the end of the line

- Ternary operator
  - exp1 ? exp2 : exp3
  - Has three expressions as operands, and evaluates *in order*
    - exp1 is evaluated first, and if it is true, exp2 is evaluated, and its value is used as the value of the *whole operation*
      - If exp1 is false, exp3 is evaluated instead, and its value is used as the value of the whole operation
  - min = i < j ? i : j
    - If i is less than j, i is the value of the operation. Else, j is the value of the operation

While Loop

while (<exp>)
  <stmt>

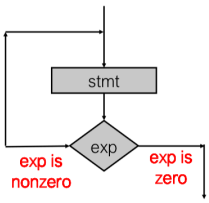


• Example: computing sum of 0..99

```
int  i = 0, sum = 0;
while (i < 100) {
    sum = sum + i;
    i++;
}
// Same as
while (i < 100)  sum += i++;
```

- While loops runs while a condition is met, and stops when it is invalid
  - Do-While loops check the condition *after* executing the loop body, so it is done at least once

```
do
   <stmt>
while (exp);
```

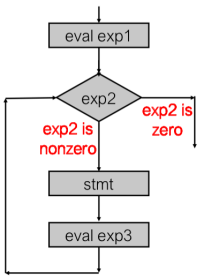


10

- For loops have an initialization, condition, and increment
    - Has a base condition, a condition for looping, and the increment of the condition is increased per loop

For Loop

```
for( <exp1>; <exp2>; <exp3>)
    <stmt>
```



- Sometimes called "counting" loop
  - More like swiss-army knife!
- Three expressions:
  - Initialization, condition, increment
- Equivalent to

```
exp1;
while (exp2) {
  <stmt>
   exp3;
}
```

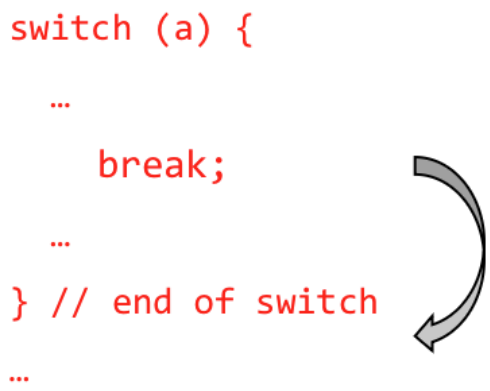
11

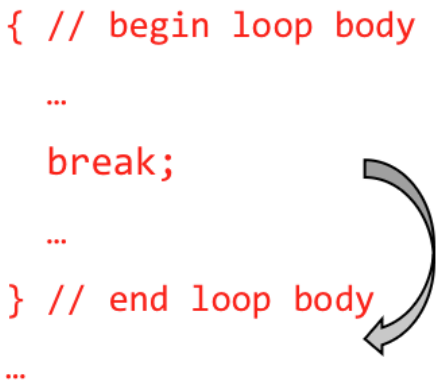
- Comma operator
    - exp1, exp2
    - Uses two expressions
        - exp1 is evaluated first, and then exp2
            - exp2 is the result of the *whole* expression
    - *HAS THE LOWEST PRECEDENCE*
    - Goes from left to right
        - Order affects final output, so keep in mind
- Multiple branching using else-if
    - Can be used to cover different cases and outcomes in one chunk of code
- Switch/Selection statement
    - Given an input, the output will be dependent on if it meets a certain case or not

Switch example

```
// Assume all variables are defined as int
switch (i) {
    case 0:
        n0 ++; break; // Note the break statement
    case 1: // No break for case 1. Will continue.
    case 2:
    {   // Can put a block here and define new variables
        int a = d + 10;
        n1 = a * 10; break;    }
    default:
        n_other ++;
}
```

- If the case meets case 0, it does what the associated action is and BREAKS the loop
    - If the case meet case 1, it goes to case 2 and does their action and BREAKS the loop
- Break statement
    - Commonly used in switch statements to prevent fall through to the successive case
    - Also works in for, while, and do-while loops
        - Loop termination happens immediately, and control resumes at the statement immediately after the loop

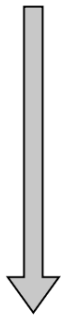




- Continue statement
    - Skips the rest of the current loop iteration and moves to the next one
        - Used in for, while, and do-while loops but alo can show in nested if/else statements as well
            - In nested loops, it applies to the innermost enclosing loop
            - For “for” loops, it goes to the evaluation of the increment expression

- Operators covered so far

Most



Least

Operator precedence and associativity	
Operators	Associativity
() ++ (*postfix*) -- (*postfix*)	left to right
+ (*unary*) - (*unary*) ++ (*prefix*) -- (*prefix*)	right to left
* / %	left to right
+ -	left to right
< <= > >=	left to right
== !=	left to right
&&	left to right
||	left to right
?:	right to left
= += -= *= /= etc	right to left
, (*comma operator*)	left to right

20

- Shortcut evaluation of logical expressions
  - Expressions that contain logical operator && and ||, the evaluation stops when the outcome true or false is known
  - Make is safe to write certan lines of code

```
exp1 && exp2
// Evaluate exp1. If false, exp2 is not evaluated.

exp1 || exp2
// Evaluate exp1. If true, exp2 is not evaluated.
```

- Makes it safe to write code like

```
if (a != 0 && c == b/a) … // no divide by 0 error
```

- Common pitfalls
  - Confusing assignments and tests for equality
  - Off by one errors in counting loops
  - Confusing logical and bitwise ops
  - Forgetting the “break” statements in a switch
  - Dangling else in nested if-then-else code blocks

- Dangling else cases

```
if (a) if (b) s1++; else s2++; // Avoid this

// which 'if' is 'else' associated with?

if (a) {if (b) s1++; else s2++;} // option 1

if (a) {if (b) s1++;} else s2++; // option 2

// C chooses option 1
```

- Comparing if-then in C and Python
  - Indentation has no information in C, and block structure is explicit
    - Uses {}
  - Condition must be in parentheses, unlike in Python

• C version

```
if (n==0) return 1; else return n*fact(n-1);
```

• Python version

```
if n==0:
   return 1
else:
   return n * fact(n-1)
```

```
// Easier to read
if (n == 0)
    return 1;
else
    return n * fact(n - 1);
```

- Declaring variables in for-loops (C99)
  - C99 allow variable declarations in init expression
    - The variable i lives inside the loop, and is undefined after the loop

```
int  sum = 0;
for (int i = 0; i < 100; i ++)
    sum += i++;
```

- Example: computing the sum of odd integers

```
int  i, sum;

sum = 0;
for (i = 1; i < 100; i++)
    if (i % 2) sum += i;

// (i % 2) is the same as (i % 2 != 0)
// the condition can also be (i & 1)
```

Lecture 4 - Functions

- Programming requires problem decomposition into manageable pieces
  - C programs are made up of functions
- Function definitions
  - No nesting functions within functions when defining them
  - Return type can be “void” → no return value is expected
    - Compiler assumes return type is int when given no explicit type
  - Return statements
    - Terminates execution and returns control to the caller
    - return expr; terminates and passes value of expr back to the caller
  - Execution also terminates when the end of the function is reached

- Function declarations
  - Can be defined in any order

Function declarations (prototypes)

- Functions can be defined in any order
  - Declare a function before first use if definition comes later
  - Function prototypes often placed in header files (and reused)

```
#include <stdio.h>

int fahrToCelsius(int);                          Declaration

int main() {
   for(int fahr=0; fahr <= 300; fahr += 10)
      printf("%d F is %d C degrees\n",fahr,fahrToCelsius(fahr));
   return 0;
}

int fahrToCelsius(int degF) {
   return 5 * (degF - 32) / 9;                   Definition
}
```

4

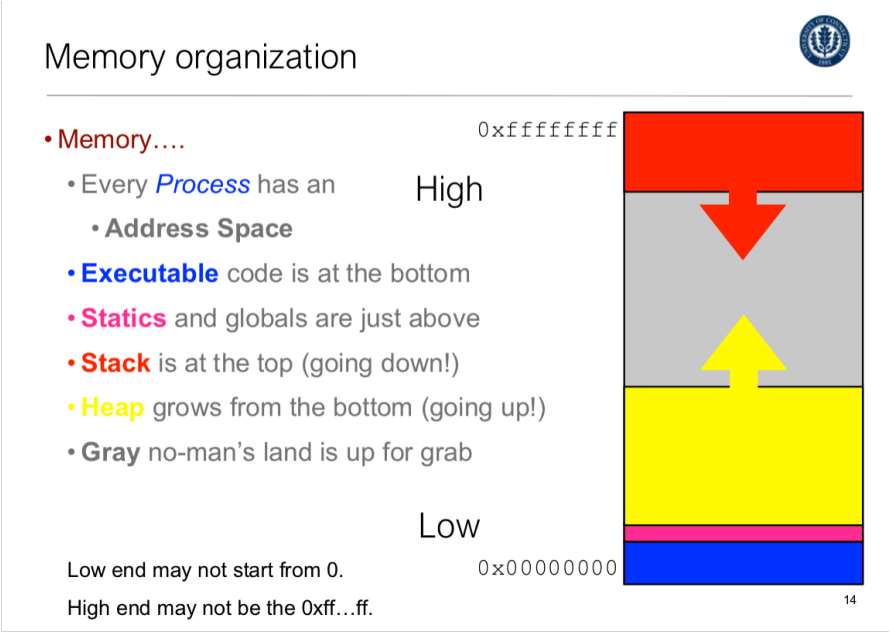
- Example: computing b^n
  - Power function
    - Returns an integer and has two parameters: base b and exponent n
      - b and n are both integers as well
  - Functions can declare local variables as wll
    - Parameters and local variables are only accessible within the function
    - They are classes as “auto”, which means they are discarded when function returns
  - Parameters are passed by value
    - n is changed by the power function, but i does not change

```
int power(int b, int n)
{
   int rv = 1;
   while (n>0) {
      rv *= b;
      n--;
   }
   return rv;
}

int foo ()
{
   int i = 10;
   return power(2, i);
}
```

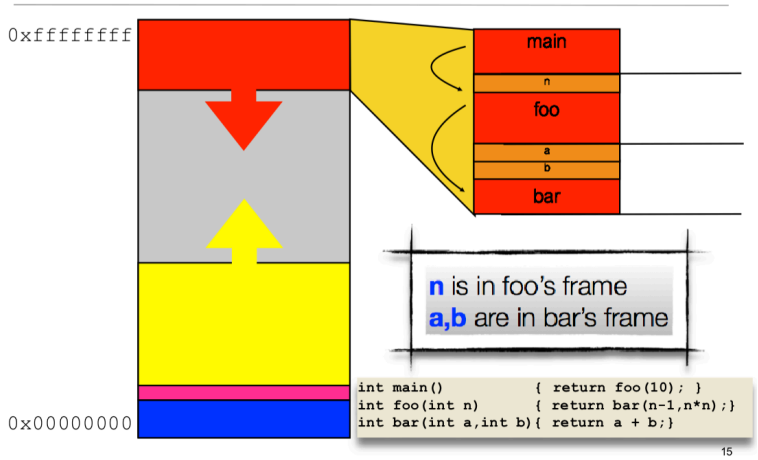
- Static and Global variables
  - Static local variables are not visible outside the function but retains value across function calls
  - Global variables
    - Declared outside functions and retained for the whole time it is running
    - Storage class of extern by default
      - It can be access from functions in other files
    - Static Global Variables are visible only in functions defined in the same file after variable declaration

- Caution with static and global variables
  - Nice functions only depend on their inputs
  - Static and global variables have side effects
    - Retains the values across function class
    - Changes the meaning of the function in each call
    - Cannot understand the function without all the code
  - Only use static or global variables if you know what to do

- Function call context
    - Copies of function arguments (call by value)
    - Automatic local variables
    - Return addresses
- Call contexts managed by the execution stack
    - Stack frames are created for each function call
        - Lasts for the duration of the call and discarded when the function terminates
        - Nothing in the frame survives the call



- Memory organization
    - Each process has an <u>address space</u>
        - Executable code is at the bottom
        - Static and global variables are above the executable code
        - The stack is at the top, going <u>down</u>
        - The heap starts on top of the statics and global layer and goes <u>up</u>



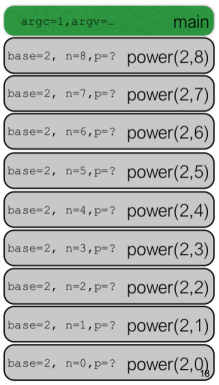
- Recursion
    - Recursive calls are supported
        - Induction on n, Base case returns 1, Inductive case does ____

```
int power(int base,int n) {
   int rv = 1;
   while (n>0) {
      rv *= base;
      n--;
   }
   return rv;
}
```

```
int power(int base,int n) {
   if (n==0)
      return 1;
   else {
      int p = power(base,n-1);
      return base * p;
   }
}
```

- Example
    - Program goes from bottom up until it hits the last call of the function

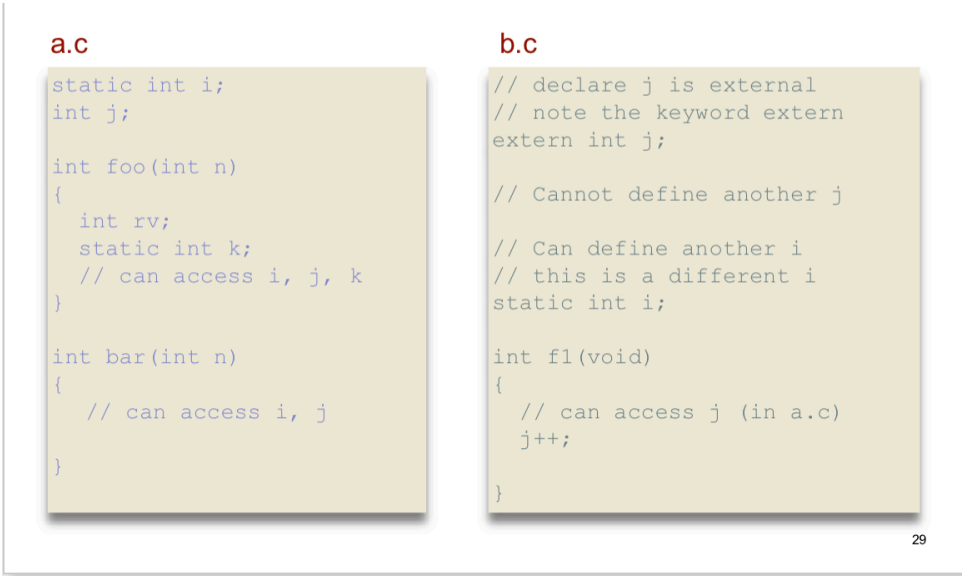
```
#include <stdio.h>

int power(int base,int n) {
   if (n==0)
      return 1;
   else {
      int p = power(base,n-1);
      return base * p;
   }
}

int main(int argc,char* argv[])
{
   printf("%d^%d = %d\n",2,8,power(2,8));
   return 0;
}
```



- In the image above, the call stack goes through the different calls of the power function until it hits power(2, 8)
    - When it reaches that, it prints out the value of the call
    - Done because the main function has it, so it prints the value of 2^8
        - Goes from the lowest possible n value (0) until the given value by increments of 1

- Variable scope - Example A
    - int j = a variable that can be used in future functions in the file
    - static int i = constant variable value for i, can be used in future functions in the file
    - int n = used in foo and bar, local variables that can have <u>different</u> values in each function
    - static int k = constant variable value for k, only accessible in the foo function since it was made there
- Variable scope - Example B
    - extern int j = external variable, so it can be accessed in other functions, but it <u>cannot</u> be defined in another function

- static int i = static variable value, but can be defined again in a different context
    - Can be accessed throughout the file



- Order of evaluating arguments
    - The order for evaluating arguments is important

- Macros as fast functions
    - Macros can take arguments, mainly for min/max uses

```
#define MIN(x,y) ((x) <  (y) ? (x) : (y))
#define MAX(x,y) ((x) >= (y) ? (x) : (y))

int main()
{
   int a = 10, b = 20;
   int x = MIN(a,b);
}
```

- Notice how
    - Arguments are always put in parentheses
    - Why?
- Arguments are put in parentheses in macros to avoid operator precedence issues

```c
#include <stdio.h>
#define MULG(x,y) ((x)*(y))
#define MULB(x,y) (x*y)

int main()
{
   int x = MULG(99+1,2);
   int y = MULB(99+1,2);
   printf("x is %d\n",x);
   printf("y is %d\n",y);
   return 0;
}
```

```
src (master) $ cc macros.c ; ./a.out
x is 200
y is 101
```

- MULG and MULB were already definined before main

- Variable numbers of args
    - Use functions in <stdarg.h> to access arguments

AI EXPLANATION:

The Key Components

1. **`va_list`**: A special type (think of it as a pointer) that holds the information needed to retrieve the additional arguments.
2. **`va_start(v, last_fixed)`**: Initializes the `va_list`. It needs the name of the last "fixed" argument (in your example, `n`) to know where the variable list begins.
3. **`va_arg(v, type)`**: This is the "grabber." It returns the next argument in the list. You must tell it the expected **type**(e.g., `int`, `double`).
4. **`va_end(v)`**: Cleans up the list. Always call this before the function returns to avoid issues.

- Example: static var in a function
    - Silly function only has a static declaration
        - Hidden is set to zero for only the first execution of silly
            - Value is retained for each call due to it being static
        - Function output is not dependent on only the inputs
        - If the hidden is OUTSIDE silly, it can be changed by the other functions, which can be bad

Lecture 5 - Arrays and Pointer Basics

- Arrays
  - A linear and contiguous collection of things
    - Each thing is the same type (ex. all char, int, array, etc.)
  - Accessing array elements is similar to lists in Python
    - ind starts at zero
    - can initialize with number of elements, but can leave that blank if all elements are listed

```
int x[5];   // define an array of 5 int's
// accessing array elements is similar to accessing list in Python
// the index starts from 0
x[0] = 1;   x[1] = 2;   x[2] = 3; x[3] = 4; x[4] = x[3] + 1;
// initialize array with a list
int y[5] = {1, 2, 3, 4, 5};
// Number of elements is optional if all elements are listed
int z[] = {1, 2, 3, 4, 5};
// Specify the value of first 2 elements. The rest are set to 0
int a[5] = {1, 2};
// C99. b will have 1, 2, 0, 0, 5.
int b[5] = {1, 2, [4] = 5};
```

- The last line works because [4] = 5 means that the fourth element will be 5 instead of automatically zero

- Array in memory
  - Index always starts at zero, and the last is one minus the length

- String initialization
  - Strings are char arrays that ends in a null char
  - Can be initialized with a list of characters or a string (double quoted)
- Arrays as automatic variables
  - Can declare arrays inside any function or block
    - Is destroyed when exiting from the function or block
  - Variable length arrays (VLAs) → the size of the array can depend on function arguments or other known values

```
int foo(int n,int k) {
    int x[n];  // The value of n is known at this time
               // Like other auto variables, x is kept on
               // the stack and is NOT initialized

for (int i = 0; i < n; i++)
        x[i] = 0;
    ……
    return -1;
}
```

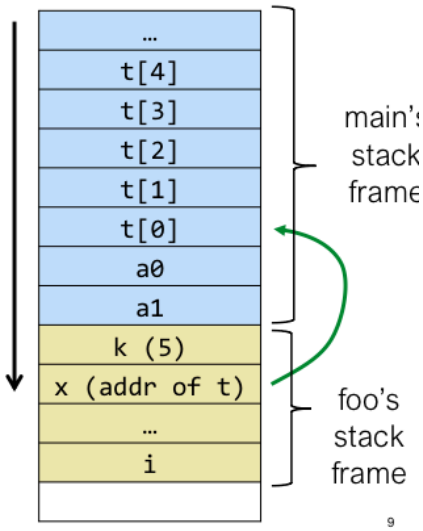
- Array assignment
    - Cannot assign a whole array at once to another array, even when the types match
        - Has to change the values in one array to another
- Arrays and functions
    - With one issue, arrays can be passed to functions
        - Calling convention! → its BY VALUE for all things <u>but arrays</u>
    - Arrays are always passed <u>by reference</u>
        - They use pointers
    - Functions cannot return arrays, so there is no easy assignments

- Array argument example
    - Address of t is passed to foo(), and the modifications to x are <u>visible</u> in main
    - The main stack frame is <u>above</u> the foo stack
        - Main holds the actual values for t, while x holds the data that <u>references</u> the things in main
        - When it gets to the point where x is dependent on t, it goes back up the stack frame to where it references t

---

- Address of t is passed to foo()
- Modifications to x are visible in main!

```
int foo(int x[], int k) {
    for (int i = 0; i < k; i++)
        x[i] = i;
    return x[0];
}

int main() {
    int t[5];
    int a0 = foo(t, 5);
    int a1 = t[4];
    printf("%d %d\n", a0, a1);
   // more code
}
```



- Multidimensional arrays

- Declaration involves listing the dimensions and the values in the array [width][length]
    - Its always row zero first, then row 1 + column zero first, then column 1 in memory

---

```
// declaration and initialization
int   h[2][3] = { {0, 1, 2}, {10, 11, 12} };
```

	0	1	2
0	0	1	2
1	10	11	12

	Address	Value
	1024	
h[1][2]	1020	12
h[1][1]	1016	11
h[1][0]	1012	10
h[0][2]	1008	2
h[0][1]	1004	1
h[0][0]	1000	0
	996	

Array layout in memory:
(assuming an int has four bytes)
- Row 0 first, then Row 1, ...
- In each row: column 0 first, then column 1, …

10

- Pointers
    - Is a value denoting the address of a memory cell
- Variables and memory
    - Memory is an array of bytes → the bytes are numbered with addresses
        - memories = unsigned integers
    - Every variable is associated with two numbers: address and the value stored at the address

- Implicit address use
    - Compilers use addresses implicitly to run code blocks
- Explciit use: pointers
    - Variable that holds the address of something
    - int *p
        - Value of p is an address of an int, with p itself having an address too

- Referencing and dereferencing
    - & is a referencing operator → it gets the address of something
    - * is a dereferencing operator → it uses the address given to it

Example:

```
int x = 10, y;

// px is a pointer to int, i.e., the address of an integer
int   *px;     // px itself has an address

px = &x;       // &x is the address of x. Save it to px

// *px: use px as an address to get the value at that location
y = *px;       // and save the value in y

// save 20 to location pointed to by px (use px as an address)
*px = 20;      // px has x's address, so x becomes 20
```

15

- Picturing memory
    - int x and y is actualized to address 1020 and 1016
        - pointer *px is also made, with its address being 1012
        - values in 1016 and 1012 is unknown, but the value at 1020 is defined as 10
    - pointer px's value is assigned to the address of x (&x = 1020), so the address of 1012 holds the value of 1020
    - var y is then given given the dereferenced value of px, which is 10
        - px has a value of 1020, so it goes to the value stored at the address of 1020
    - finally, the address of x is given the value of 20, but the address of the pointer remains the same

```
// assume 32 bits int and address

int x = 10, y;

int   *px;

px = &x;

y = *px;

*px = 20;
```

	Address	Value
	1007	
	1006	
x	1020	20
y	1016	10
px	1012	1020
	1008	
	1004	
	1000	

- Revisit the example
    - X is the starting address of an array, which is the same address as element zero

```
int foo(int x[], int k) {
    for (int i = 0; i < k; i++)
        x[i] = i;
    return x[0];
}

// x is the address of an int
int foo(int * x, int k) {
    for (int i = 0; i < k; i++)
        x[i] = i; // use the pointer as an array
    return x[0];
}
```

x is the starting address of an array, which is the same as the address of element 0.

20

- If all pointers do is store the address, why should they have a type?
    - Its because they also store the value, so they have to match to be able to work

- Automatic arrays summary
    - Local arrays
        - Allocated when entering the function, and deallocated when leaving (all automatic)
        - They are not initialized, and exist directly on the stack like other variables
    - Size
        - It can either be static (constant) or dynamic (an expression), but cannot be too big because it is on the stack
- VLA support in C
    - VLAs = Variable length arrays
        - Not desirable to put large amounts of data on a stack in some applications
            - You would have to use heap to keep the array

```
void      foo (int n)
{
   int    i;
   int    a[n];            // define an array of n integers
   int    b[2][n];         // 2 by n array
   // indexes start from 0. a[0], a[1], …, a[n-1].     Not a[n] !
   for (i = 0; i < n; i ++)// typical loop of n times
       a[i] = i;           // write to array a
  // copy a to row 0 of b. You cannot do b[0] = a
   for (i = 0; i < n; i ++)
       b[0][i] = a[i];
}
```

30

- Another array example

- int a[n] designs an array of n integers, but int b[2][n] makes a 2 x n array of integers
    - Indexes always start from zero to (n-1)
- The for loop can add in values for an array as long as the increment leads to a value one less than the length
- To copy a to a row of b, we have to define which row it goes to
    - It has the same length, but we need to say if it goes to b[0] or b[1]

- Pointer declarations
    - int* p, int * p, and int *p are all equivalent
        - All declare p to be a pointer to an integer
            - First makes the above statement clear, second is “non committing”, and the third says that what p points to is an integer

---

• Consider the following declarations:

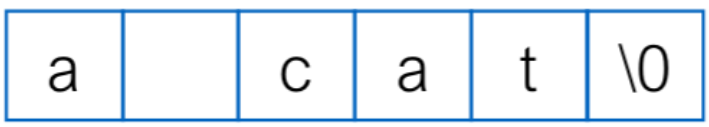
```
int  *a, b;
int* c, d;
int  e, *f;
int  *g, *h;
```

• What are the types of the variables?

- a, c, f, g, h are int *
- b, d, e are int

- Pitfalls
    - The declarations above show different things
        - a, c, f, g, and h are int*
            - Pointers that point to an integer
        - b, d, and e are just int
            - Not a pointer

“a cat” in memory



- Strings
    - Strings are just character arrays with a null (‘\0’) terminator
    - The compiler allocates memory space to contain all the characters (including spaces) plus the terminator
        - Cannot be changed by the program

Strings

• They are char arrays, but must end with '\0'

```
char line[100];
…
```

index must be range

Not the end of the string yet

```
for (i = 0; i < 100 && line[i]; i++) {
  if (isalpha(line[i])
     putchar(line[i]);  // print only one character
}
```

- For loops that go through strings need <u>two</u> checks to be error-free
    - Make sure that the increment is less than the number of characters in the array
    - Needs to check if the character it is on is <u>not</u> the line terminator (done with line[i])
- Refers to a string via a pointer to the <u>first</u> character
    - Also can be the <u>address of the first character</u>

```
// str is the address of a char,
// which is 'm', the first character in the string
// str[0] is 'm', str[1] is 'y', and so on
char *str = “my string”;

// s is a pointer to char
char *s;

// s will refer to the same string as str.
s = str;
```

- char *str stores the address of the character string “my string”
    - That is equal to the address of ‘m’ since its the first character
- char *s is a pointer to char, and s would refer to the same string as str

- Copying strings
    - strcpy() in string.h can help copy a string

```
// buf is a char array
char buf[100];
// str refers to the string literal
char *str = "a cat";
// now, buf has a string. strcpy copies '\0'!
strcpy(buf, str);
// s refers to the string in buf[100],
// which is different from str
char *s = buf;
```

- buf is defined as a character array of 100 elements
  - *str is a char pointer that refers to the literal string, “the cat”
- strcpy(buf, str) copies the string in “a cat” to buf
  - *s refers to the string in buf, but it si different than str

- Implementation of strcpy()

• Idea: Copy a character each time until '\0' is copied

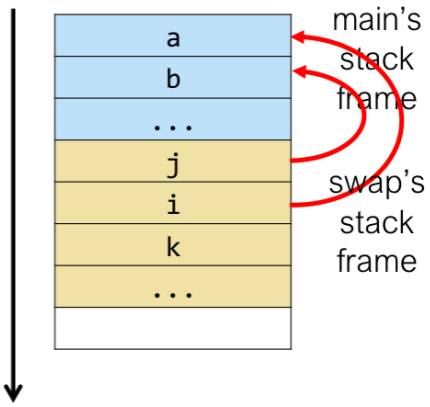
```
char * strcpy(char *dest, char *src)
{
    unsigned int i = 0;
    // copy any character that is not '\0'
    for (i = 0; src[i] != '\0'; i++)
        dest[i] = src[i];
    dest[i] = src[i]; // copy '\0'
    return dest;
}
```

- Example: pass argument by value
  - Since parameters are passed by value, the following would not work for swapping

```
void swap(int i, int j)
{
    int k = j;
    j = i;
    i = k;
}

int main()
{
   int a=1, b=2;
    swap(a, b);
   return 0;
}
```

- You need to pass pointers to the variables to do the swapping

```
void swap(int *i, int *j)
{
    int k = *j;
    *j = *i;
    *i = k;
}

int main()
{
   int a=1, b=2;
    swap(&a, &b);
   return 0;
}
```



- Example: scanf
    - We pass to scanf three different values
        - Name
        - Address of pears [&]
        - Address of apples [&]

```
#include <stdio.h>

int main()
{
   char name[128];
   int pears  = 0;
   int apples = 0;
   scanf("%s %d %d", name, &pears, &apples);
   printf("%s ate %d apples and %d pears.\n",name,apples,pears);
   return 0;
}
```

- Frame of main in picture
    - The name part of the stack has a size of 128 bytes
    - When doing scanf, the stack grows to incorporate the four argos that are called when running it in main
        - %s %d %d is done inside the format part of the scanf stack
        - a1 is assigned to the part between pears and apples, a2 to the part between apples and the name part of the stack, and a3 to the part between apples and a3

Lecture 6 - Dynamic Memory Allocation

- static/global memory pool
    - This is where
        - All constants are held (includes string literals)
        - Global variables
        - All variables declared as static

    - Allocated when the program starts
    - Deallocated when the program terminates
    - Has a fixed size → compiler needs to know the size to make reservations for the program

- Stack
    - This is where
        - Memory comes from for local variables in functions
    - Easier to manage since its automatic
        - Allocated when entering the function, and de-allocated when leaving
            - Scope of the memory is for the function
            - Shouldn't be used after the function returns
    - Default size is 2MB

- Heap
    - This is where
        - Memory comes from manual allocations
    - The programmer is in charge for allocating and de-allocating the heap
    - There can be a lifetime of memory blocks as longas they are not freed

- Requesting memory from heap
    - void* malloc(size_t size);
        - size_t is an unsigned int data type that is defined in <stdlib.h>
        - Represents the sizes of objects in bytes
    - A call to malloc(n) returns a generic pointer (void*)
        - Points to a memory block of n bytes in the heap\
    - NULL is returned if there is an error

- Generic pointers: void*
    - A pointer to a memory block whose content lacks a type
    - For raw memory operators or in generic functions
    - Is casted automatically when assigned to other pointer types
    - Needs casting before deferencing for read/write
    - NULL handles errors

- Use for raw memory operations or in generic functions
- Automatic casting when assigned to other pointer types

```
int * pox = malloc(6 * sizeof(int));
```

- Requires casting before dereferencing for read / write

```
*(int *)pv;  // use pv as an int *
```

- NULL, a special pointer value useful for initializations, error handling

```
#define NULL ((void*)0)
```

- Calls to malloc() may fail
  - Happens when you run out of memory
  - Will return a NULL value instead → can only report the error and terminate
- Need to tell malloc() the amount of bytes you need
  - Requesting space for an array
    - Number of elements
    - Amount of space per array element
      - Can use sizeof(T) to find the bytes needed for one element of the value

```
int* pox = malloc(6 * sizeof(int)); // request space for 6 ints
if (pox == NULL)
  report error and finish;
```

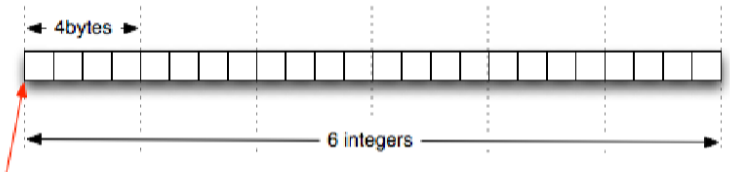
- Alternate method uses calloc()
  - void* calloc(size_t nmemb, size_t size)
    - Implemented in terms of malloc()
    - Intializes the content to zero!!

- Adjusting size
  - Can use realloc(void* ptr, size_t size)
    - Set the pointer to a value and original malloc, and then use realloc with the same pointer

- Deallocation
  - free(void *ptr)
    - Calls the function to freeup space allocated to the pointer
      - SET POINTER TO NULL AFTER
- Key rules
  - Everything you requested to eventually be freed
  - Only free what is allocated by malloc/calloc/realloc
- Consequences

- Memory leaks
    - Program will run out of memory over time
- Undefined behavior and crashes
    - Freeing unallocated memory or freed memory can lead to a memory error and program crash
        - Worse case, it can corrupt the heap and crash later
        - Worst case, it can still let the program run and corrupt your data and disk

- Pointers and arrays
    - pox = malloc(6 * sizeof(int))
        - Can use pox like it's an array



- Example: use pointer as array
    - pox = malloc(sizeof(int)*n)
        - requests the memory from the heap
    - *pox = 0
        - Sets the integer at the address of pox to be zero
    - pox[0] = 0
        - Same as the line before
    - pox[1] = 1
        - Int after pox[0]
    - Goes until all the ints are covered
    - free(pox)
        - Frees memory from pox

```
void doSomething(int n) {
    int * pox;
    pox = malloc(sizeof(int)*n); // request mem from heap
    *pox = 0;    // set the int at the address pox to 0
    pox[0] = 0;  // same thing
    pox[1] = 1;  // the int after pox[0]
    // more lines here ...
    free(pox);   // remember to free!
}
```

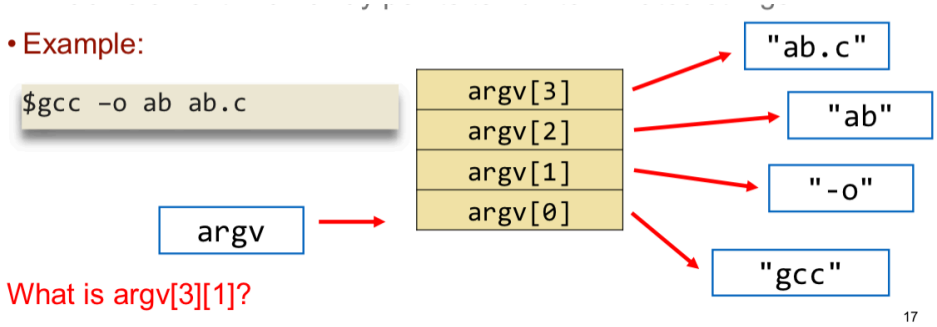
- Array of pointers
  - Pointers can also be placed into arrays

```
int a0;
int a1;
int a2;


char *p0 = malloc(10);
char *p1 = malloc(10);
char *p2 = malloc(10);
```

```
// array of int's
int a[3];

// array of pointers
char * p[3];

// Can also do with a loop
p[0] = malloc(10);
p[1] = malloc(10);
p[2] = malloc(10);
```

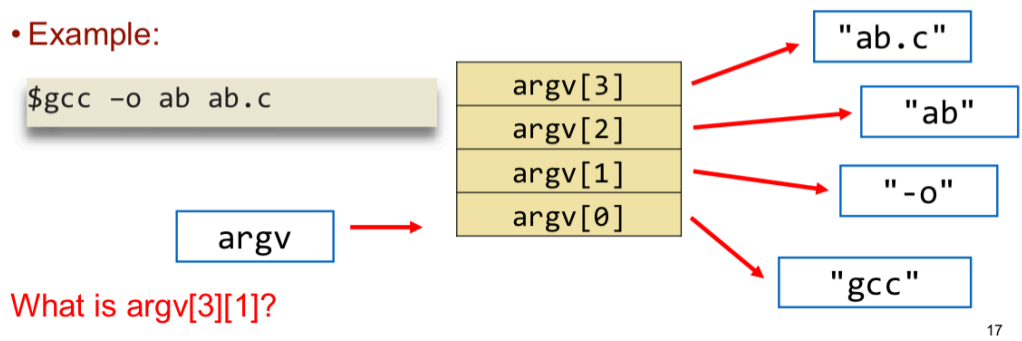
16

- Example: command line arguments
  - Int main(int argc, char, *argv[])
    - argc = number of arguments on the command line
    - argv[] = array of pointer to characters
      - Each element in the array points to null-terminated strings
  - arg[3][1] would be b
    - arg[3] refers the the fourth element in the array
      - arg[3][1] refers to the second element in the string



17

- Example: allocating 2d dynamical array



17

- Example: allocating 2d dynamical array (second option)
    - Request all the memory space needed with one malloc() call
        - More efficient than calling malloc() m times per row
    - Calculate rows[1], rows[2], … , rows[m-1]
        - Can calculate the address of each element too!
- Example: allocating 2d dynamical array (third option)
    - arr is a pointer to an array of n integers
    - malloc() is done for the amount of data stored
    - Eventually will free memory

```
void doSomething(int m, int n)
{ int (* arr)[n];    // arr is a pointer to an array of n int's
  arr = malloc(m * n * sizeof(int)); // one malloc() for data
  // to access elements
  arr[1][2] = 1; arr[2][3] = arr[1][2] + 10;
  ...
  free(arr); // free memory
}
```

n must be constant if Variable Length Array(VLA) is not supported

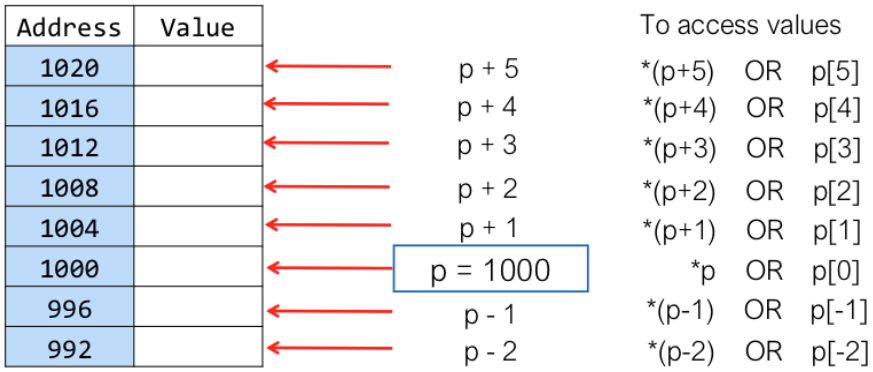
20

- Pointers with different addresses
    - A static?
        - Address wil never “go bad” → static lives as long as the program does
    - A stack (automatic) variable?
        - Address is as valid as the variable is
            - Address becomes bogus when the function returns a value
    - A heap variable?
        - Address is valid as long as the variable is
        - Variable disappears when explicitly de-allocated (or freed)

Lecture 7 - Pointer Arithmetic and Structures

- Pointers are addresses
    - Value of a pointer is a byte address
        - Unassigned integer used to number bytes in memory
        - Range is between:
            - 0x00000000 to 0xFFFFFFFF in 32-bit architecture
            - 0x0000000000000000 to 0xFFFFFFFFFFFFFFFF in 64-bit architecture
    - Corollary
        - If a pointer is an integer, you can do arithmetic operations to compute other addresses

- Pointer addition example
  - Suppose p is a pointer to an integer, and has a value of 1000
    - p+1 is not the next byte address, but is the address of the next item



- Adding a pointer and an integer
  - When adding a pointer and an integer, the result is a pointer of the same type
    - Different from regular integer addition
    - Integer is automatically scaled by the size of the type pointed to
  - If p is a pointer to type T and k is an integer
    - p + k and k + p are valid expressions that evaluate into a pointer of type T
    - Has a byte address equal to
      - (unassigned long)(address stored in p) + k * sizeof(T)
  - NOTE: C standard does not allow arithmetic on void *
    - gcc has an extension, treating sizeof(void) as 1

- Pointers subtraction
  - Subtracting one pointer from another must have the same type
  - The result is the number of data items between the two pointers
    - Is NOT the number of bytes
    - Can be thought of as “distance” between the two pointers

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

int main() {
  int *p = malloc(sizeof(int)*10);
  int *last = p + 9;
  int  dist = last - p; // both are int *
  printf("Distance is %d\n",dist);
  free(p);
  return 0;
}
```

Output

```
$ gcc ptrsub.c
$ ./a.out
Distance is 9
$
```

5

- Pointer comparisons
  - Can also compare pointers together with comparison signs
    - Meant to check boundary conditions and manually manage memory blocks

- Semantics
  - Only based on the memory layout and compares bits in pointers as unassigned integers

- Effects of casting types
  - If you cast a pointer type, any subsequent pointer arithmetic will use the type you choose
  - Cast a pointer to char * if you do not want scaling
    - sizeof(char) is one

```
int * t;
char * p = (char *) t + 8;   // 8 is not scaled
char * q = (char *) (t + 8); // 8 is scaled
```

- Arrays and pointers
  - Arrays and pointers can often be used interchangeably

```
int  a[10];
int *p = a;

// all of the following evaluate to the value of a[0]
*p              p[0]        *a          a[0]

// all of the following evaluate to the value of a[1]
*(p+1)          p[1]        *(a+1)      a[1]

// all of the following evaluate to the address of a[0]
// type is int *
p               &p[0]       a           &a[0]
```

- Example: arrays and pointers
  - There are multiple equivalent ways of initializing an array

```
int a[10], *p = a;  // not *p = a; it is int *p; p = a;

for(int i=0; i<10; i++) a[i] = i;  // array indexing

for(int i=0; i<10; i++) p[i] = i;  // indexing via pointer

for(int i=0; i<10; i++) *(p+i) = i; // explicit pointer arithmetic

for(i=0; i<10; i++) i[p] = i;   // obfuscated but valid C!

for(i=0; i<10; i++) *p++ = i;   // common pointer use idiom
```

```
int * tp = p;
p ++;
*tp = i;
```

- Arrays and pointers are NOT the same

```
int   a[10];
int   *p = a; // a is converted to int *

// a is still an array after &
&a  // pointer to array of 10 int's  int (*)[10]
&p   // pointer to a pointer to int   int **

// a is still an array after sizeof
sizeof(a) // 40 because a is an array of 10 int's
sizeof(p) // 8  because p is a pointer

p++;   // can increment p
a++;   // cannot increment a; this will not compile
       // Similar to n++ vs 2++
```

- *p = a turns the array a into an integer pointer
  - & doesn't change the array a
    - &a is a pointer to an array of 10 integers int(*)[10]
    - &p is a pointer to a pointer to an integer int **
  - sizeof doesnt change the array a
    - sizeof(a) is 40 bytes because it is an array of 10 integers (with the size of 4 bytes)
    - sizeof(p) is 8 because p is a pointer
  - p++ can increment p, but a++ does not increment a
    - like n++ vs 2++

- Example: arrays and pointers are NOT the same

```
#include <stdio.h>

void foo(int *x)
{
   printf("%lu\n", sizeof(x));
}

int main()
{
   int a[10], *p;

   p = a;  // a is converted to int *
   // a is still an array in sizeof
   printf("%lu %lu\n", sizeof(a), sizeof(p));
   foo(a); // a is converted to int *
}
```

Output

```
% gcc array.c
% ./a.out
40 8
8
```

- Code initializes pointer p and array a of 10 elements
  - a converted to int * when setting p to a
    - a is still an array in sizeof
  - foo(a) causes a to be converted to int *

- Structures
  - Mechanism to define new types
    - Called compound types and used to aggregate related variables of different types
  - Structures type declaration
    - Structures can have a type name
    - Can have “members” of any types
      - Basic types
      - Pointers
      - Arrays
      - Other structures
  - Structure variable definition
    - Specifies variable name

```
struct student_grade {
    char *name;
    int  id;
    char grade[3];
};
…
struct student_grade student1;
struct student_grade
    student2, student3;
struct student_grade all[200];
```

- Structure example

```
struct Person {
   int     age;
   char gender;
};

int main(){
   struct Person p;

   p.age = 44;
   p.gender = 'M';

   struct Person q = {44, 'M'};

   return 0;
}
```

Structure *type declaration*

Structure *variable definition*

Syntax for field access similar to Java and Python

Structure *variable definition* and *initialization*

- Example: array of structures
  - Member name is a char array
    - Certain caveats
      - Names cannot be longer than 31 characters long (since name is initialized with 32 spaces)
        - HAS TO INCLUDE THE NULL CHARACTER
      - Four people in the family (indexed from 0 to 3)
  - Array of structures for the family → nested initializers

```c
#include <stdlib.h>

struct Person {
   char   name[32];
   int    age;
   char   gender;
};

int main()
{
   struct Person family[4] = {
      {"Alice",34,'F'},
      {"Bob",40,'M'},
      {"Charles",15,'M'},
      {"David",13,'M'}
   };
   int juniorAge = family[3].age;
   return 0;
}
```

- Typedef
  - Structure names can be long → C can help define type abbreviations
  - typedef declarations
    - Gives existing types new type names
  - Helps make code more readable, with structure and typedef declarations often combined

```c
struct Person {
   char   name[32];
   int    age;
   char   gender;
};
typedef struct Person TPerson;
int main()
{
    TPerson family[4];
    ...
    return 0;
}
```

```c
typedef struct Person {
   char   name[32];
   int    age;
   char   gender;
} TPerson;
```

- Operations on struct
  - Assignment
    - All struct members copied
  - Can be passed to functions
    - Done through values
    - Still possible even if some members are arrays
  - Can be returned from a function
  - If passing by value, it cannot change members in functions
  - Passinf or reutnring large structures is costly → use pointers to structures

- Pass structure by reference

```
typedef struct Person {
   char   name[32];
   int    age;
   char   gender;
} TPerson;

TPerson * init_Person(TPerson *p, char * name, int age, char gender)
{
   strcpy(p->name, name);   // (*p).name
   p->age = age;            // (*p).age
   p->gender = gender;      // (*p).gender
   return p;
}
```

- Structure alignment
  - Structure members are aligned for the natural types

Alignment requirements on x64 architecture	
char	1
short	2
int	4
long	8
float	4
double	8

```
struct struct_random {
    char    x[5]; // bytes 0-4
    int     y;    // bytes 8-11
    double  z;    // bytes 16-23
    char    c;    // byte 24
};     // Total size 32
```

```
struct struct_sorted {
    double  z;    // bytes 0 - 7
    int     y;    // bytes 8 - 11
    char    x[5]; // bytes 12 - 16
    char    c;    // byte  17
};      // Total size 24
```

18

- Arrays and pointers
  - Copying arrays
    - Two different methods: array indexing and using pointers

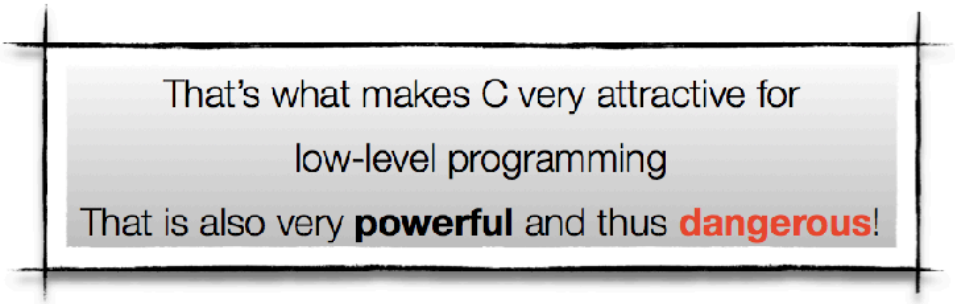
```
// using array indexing
void copy_array0(int source[], int target[], int n)
{
  for(int i = 0; i < n; i++)
    target[i] = source[i];
}

// using pointers
void copy_array1(int source[], int target[], int n)
{
  for(int i = 0; i < n; i++)
    *target++ = *source++;
}
```

```
int * tp = source;
source ++;
*target = *tp;
target ++;
```

- Array indexing method
  - Creates two arrays, source and target, as well as an integer n
    - Uses a for-loop for the length of n that copies elements on index i from the source to the target
    - n = length of array
- Pointers method
  - Still has the two arrays and the integer n
    - Similar for loop, but uses pointers for both of the arrays to make the copying of elements
      - Grey box has the corresponding code

- Typecasting pointers
  - C can cast pointers in any way I like → can forge pointers to point wherever you want

```
float f;
char p = * (char *) &f;     // 1st byte representing f
char ch = * (char *)1000;   // access any byte in memory
```



- Returning more than one value from functions
    - References
        - Caller prepares the storage
        - long int stroll (const char* stru, char** endptr, int base);
    - Arrays
        - Caller prepares the storage
        - int pipe(int pipefd[2]);
    - Returning a structure
        - Is costly if the structure is large
    - Returning a pointer
        - Can be to array or structure
            - Must be either static or dynamically allocated (read the manual!!)
        - char * strdup(const char *strl);
        - struct tm *localtime(const time_t *time);
    - Using global variables (DONT)
        - Leads to errorno

- Typedef
    - Think about how you would define a variable
    - typedef int BOOL;
    - typedef char name_t[100];
    - typedef char *Pointer

- Self-referential structures
    - Structures that uses itself in parts of its definition

```
struct Person {
   int    age;
   char   gender;
   char   name[32];
   struct Person * parents; // A pointer to this type of struct
} person1, person2;  // Can define variables here
```

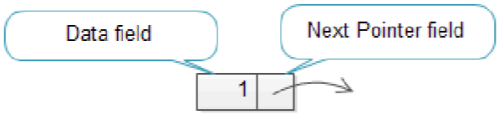
- Self-referential structures - example 2

```
struct student {
   char   name[128];
   // Can have a pointer to a struct defined later.
   // However, you cannot define an array of book here (e.g. books[8])
   struct book * books;
};
struct book {
   char   title[128];
   struct student * owner;
   struct book * next; // A pointer to this type of struct
};
```

Lecture 8 - Linked Lists, Enums, and Function Pointers

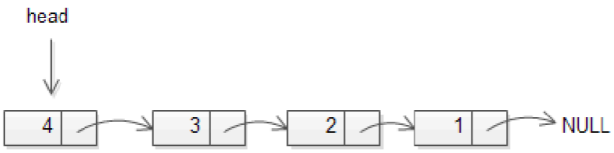
- Example: linked lists
  - Structure made up of chain of nodes
    - Starts from head, which is a node that has a reference to the next node

```
typedef struct node_tag {
   int    v;               // data
   struct node_tag * next;  // A pointer to this type of struct
} node;                     // Define a type. Easier to use.
```



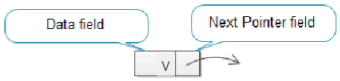
- Head
  - node * head; → head is a pointer now, <u>not a node</u>
  - head = NULL; → the head has a value of NULL in the beginning

After adding nodes into the list,



- Creating a node

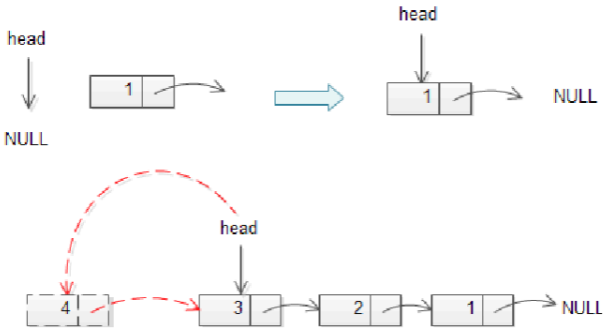
```
node * new_node(int v)     // create a node for value v
{
    node   * p = malloc(sizeof(node)); // Allocate memory
    assert(p != NULL);      // you can be nicer

    // Set the value in the node.
    p->v = v;        // you could do (*p).v
    p->next = NULL;
    return p;        // return
}
// is it similar to creating objects using "new"?
```



- node * new_node(int v); → creates a node with a value v
    - node * p = malloc(sizeof(node)); → pointer p that then allocates memory on heap based on the size/type of the node
    - assert(p != NULL); → makes sure that value of pointer p isn't NULL (crashes if so b/c there is no more memory left)
    - p->v = v;
        - -> means going from the address in p to v, and then it sets the value in that address to v
    - p->next = NULL;
        - The next node isn't made yet, so it is set to NULL to show it is the end of the list
    - return p; → returns the address back to p for future use

- Prepend

```
node * prepend(node * head, node * newnode)
{
    newnode->next = head;    // works even if the list is empty
    return newnode;          // head changed !!
}
```

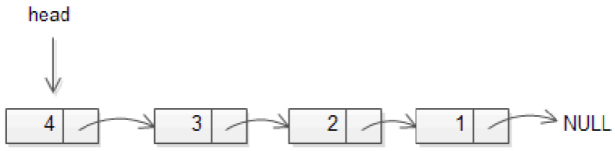


- node * prepend(node * head, node * newnode)
    - newnode->next = head;

        - Goes from the address in newnode to next, and stores the value of head into next
    - return newnode;
        - Changes the head address to newnode since it is the <u>new head</u>

- Finding the last node

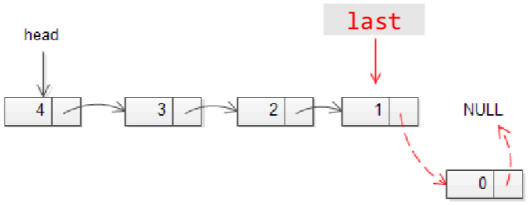
```
node * find_last(node * head)
{
    if (head != NULL) {        // only if the list is not empty
        while (head->next != NULL)
            head = head->next;
    }
    return head;
}
```



- node * find_last(node * head)
    - if(head != NULL) → checks if there is enough memory for the operations
        - while(head->next != NULL) → runs until the end of the linked list
            - head = head->next;
                - Goes from the address in head to next, and stores that value in head
    - return head;
        - Returns the head address of the <u>last</u> node

- Append

```
node * append(node * head, node * newnode) {
    node *last = find_last(head);  // find the last one
    if (last == NULL)    // if the list is empty, newnode is the head
        return newnode;
    last->next = newnode;
    newnode->next = NULL;
    return head;     // return the (unchanged) head
}
```



- node * append(node * head, node * newnode)
    - node *last = find_last(head);
        - Finds last node in the linked list
    - if(last == NULL) → newnode is the head if the list is empty
        - return newnode;

- last->next = newnode;
    - Takes the address of last and goes to next, putting the value of newnode into next
- newnode->next = NULL;
    - Node isn't made yet, so it is set to NULL to show it is the end of the linked list
- return head;
    - Returns the head address of the unchanged node

- Enumeration types
    - User-defined interger-like types
    - Names look like C identifiers
        - Are listed in the definition
        - Treated like integers
            - Add, subtract, compare
            - Cannot print as a symbol, but the debugger would normally

```
// enum start from 0 by default
enum week {Sun, Mon, Tue, Wed, Thur, Fri, Sat};
enum week dow = Mon;

// But can be initialized; Warning is 2, Error is 3, etc.
enum status {OK = 1, Warning, Error, Fatal};
```

- Type qualifier: const

```
// constant int
const int   a = 10;      // cannot change a
// a pointer to a constant int
const int  *pa = &a;  // can change pa, but not *pa
// a constant pointer to an int
int  * const pb = &b;    // can change *pb, but not pb
// a constant pointer to a constant int
const int  * const pc = &a; // cannot change *pc or pc

// cannot change the source string
char * strcpy(char * dest, const char * src);
```

- Function pointers

```
/* function returning integer */
int func();

/* function returning pointer to integer */
int * func();

/* pointer to function returning integer */
int (*func)();

/* pointer to function returning pointer to int */
int * (*func)();
```

- Pointer to function example

```
int     mymax(int a, int  b)
{
    return (a > b) ? a : b;
}
// a pointer to function
int (*pf)(int a, int b);

// assign a value to the pointer
pf = mymax;       // C99 style. Note that it is NOT mymax()
pf(3, 5);
pf = & mymax;
(*pf)(3,5);
```

- Use of function pointers
    - A call-back mechanism
        - Generic functions
        - pthread_create()
        - Dynamic signal handlers
        - etc.
    - Can store function pointers in arrays, and arrays can be stored in structures → like objects in OOL languages
        - Ex. Python

- Example: quicksort in C library
    - qsort() takes
        - base → address of the array as an untyped pointer
        - nel → number of elements in array
            - nel = “number elements)
        - width → size in bytes of one element in the array
        - compare → a pointer to a function that compares two values

- qsort() doesn't know the type of elements its comparing or how to compare them

```
void qsort(void * base,
           size_t nel,
           size_t width,
           int (*compare)(const void *, const void *));
```

- Example: sort array of strings
  - Elements are pointers to strings
    - We need to compare strings, not pointers
  - Element in the array words is (char *)
    - a is the address of element type char *, so the type of a is (char *)*

```
int compare_string(const void * a,const void * b)
{
   char *s1, *s2;
   s1 = *(char **)a;     s2 = *(char **)b;

   return strcmp(s1, s2);  // use library function to compare
}

// or on one line
int compare_string(const void * a,const void * b)
{
   return strcmp(*(char**)a,*(char**)b);
}
```

- Calling quicksort()
  - We're typecasting to char ** before referencing

Lecture 9 - I/O and Files

- errno
  - C functions could fail, and when it does, it reports a flag reporting the failure
    - Some can set a global variable, errno, to report the exact code
  - <errno.h> needs to be called to use errno
- Manual pages can help interpret code
- Print a more descriptive message with perror()
  - takes const chat and a pointer *str
- Avoid functions that puts errno in multithread, instead use threadsafe

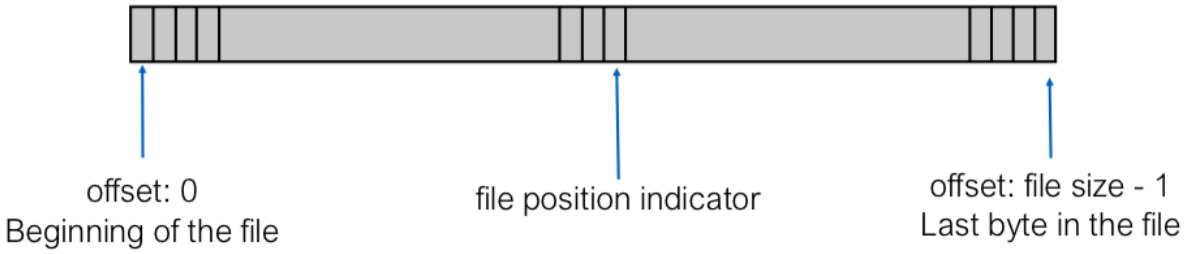
- Files and directories
  - Object that stores info, data, etc.
    - Organized in directories in Linux
      - Top directory is /
      - Can have subdirectories and files
  - Path specifies the location of a file or directory in the system
  - UNIX AND LINUX HAS EVERYTHING BE A FILE

- stdio library

- #include <stdio.h>
    - Declares the FILE type and function prototypes
        - FILE is opaque type (aka system dependent) for operating on files
            - Structure, but dont change directly
        - Uses library functions to access FILE objects through pointers
            - FILE *
    - Helps define “standard” streams stdin, sdout, stderr
        - Three are FILE *
        - Created automatically when program starts, and are also files
    - Library is linked automatically by compiler

- Files and I/O API
    - Files in C are sequential streams of bytes
    - “F” family of functions are C library functions to operate on files
        - All use FILE* abstractions to represent the target file
        - C library provides buffering
            - Reason why output of printf isn’t immediate

- Files as streams of bytes
    - Files must be opened before use
        - Sets position indicator for reading and writing
    - Each read/write starts from current position, and moves indicator
        - Writing after last byte increases file size
        - Indicator position can be moved with fseek
    - Open files are closed when program ends
        - Close explicitly when no longer needed



- Opening stream
    - FILE* fopen(const char *filename, const char *mode)
        - Opens file filename in mode as a stream of bytes
        - Returns a pointer to FILE (FILE *) or NULL (and the errno is set)
            - Different modes depending on need

Mode

- "r" : Reading mode
- "r+" : Read and write
- "w" : Writing mode, file is created or truncated to zero length
- "w+" : Read and write, but the file is created or truncated
- "a" : Append mode, the file is created if it does not exist
- "a+" : Read and append, the file is created if it does not exist. Reading starts at beginning, but writing done at the end

Check return values!

- Closing stream
  - int fclose(FILE * stream);
    - Closes the steam
    - Returns either 0 or EOF
      - 0 = works
      - EOF = problem happened; errno was set

- fgetc/fputc (one byte at a time)
  - int fgetc(FILE *stream);
    - Reads a character from the stream and returns the character just read in
      - EOF is returned when an error is reached or at the end of the file
  - int fputcc(int c, FILE *stream);
    - Writes the character received as an argument to the steam and returns the character that was written out
      - Returns EOF on error
  - Can read or write on ASCII character at a type → size of one byte
- getc/putc and ungetc
  - int getc(FILE *stream); and int putc(int c, FILE *stream);
    - Same as fgetc/fputc except they can be implemented as macros
    - Usually should use fgetc/fputc unless theres a strong reason to not to
  - int ungetc(int c, FILE *stream);
    - Pushes last read character back into the stream, and would be available for future read operations
    - Only one pushback is guaranteed
- getchar/putchar
  - int getchar(void)
    - Same as fgetc(stdin)
      - Reads character from stdin
      - Returns the character that was just read in, or EOF on the end-of-file or error
  - int putchar(int c)
    - Same as fputc(c, stdout)

        - Writes the character received as an argument on stdout
        - Returns the character that was written out, or EOF on errors
- More than one byte: get a line
    - char *fgets(char *buf, int size, FILE *in)
        - Reads the next line from in to the buffer buf
        - Halts at '\n' or after size-1 characters have been read
            - NUL is placed at the end
        - Returns the pointer ot buf if ok, or NULL otherwise
        - Do not use gets(char*)) → leads to buffer overflow
    - int fputs(const char *str, FILE *out)
        - Writes the string str to out, and stops at '\0'
        - Returns the number of characters written or EOF

- Formatted output
    - int fscanf(FILE *stream, const char *format, …);
    - int fprintf(FILE *stream, const char *format, …);
    - Formatted input from file and output to the file
    - Similar to scanf()/printf(), but is not from stdin or to stdout

- For binary data
    - size_t fread(void *ptr, size_t sz, size_t n, FILE *stream);
    - size_t fwrite(void *ptr, size_t sz size_t n, FILE *stream);
    - read/write sequence of bytes from/to a stream
    - Returns the number of items read or written
        - If the number is smaller than n, returns EOF or an error

Example:

```
int   A[10][20];
size_t   n = 10 * 20;
if (fwrite(A, sizeof(int), n, fp) != n) {
   // error
```

- Moving file position indicator
    - long ftell(FILE *stream);
        - Reads file position indicator, and returns -1 if an error occurs
    - int fseek(FILE *stram, off_t offset, int whence);
        - Sets the file position indicator
        - Return 0 on success and -1 on error

EXAM 1

- Coding Problems
  - Problem 1: common characters in strings
    - Purpose
      - Array mapping exercise → uses a fixed array of 26 integers (as booleans) to represent the alphabet
    - Strategy
      - Global record: starter code initializes a common[26] array to true → represents the assumption that all letters are common
      - Temporary record: inside main for loop that goes through every string, we need to make a temporary array
        - bool temp[26] = {false};
      - Making characters: iterates through the current string by character → each character c has their alphabetical index calculated through ASCII math index = c-'a'
        - Sets temp[index] to True
      - Intersecting records: iterates from 0 to 25 after the inner loop finishes scanning the current string → updates global common array through an AND operation (common[i] = common[i] && temp[i])
        - Disqualifies any character that wasn't found in the current string
      - Formatting output: iterates through the common array one last time → if common[i] is TRUE, converts index back to a character (char)(i + 'a') and prints it
        - Sets a found flag to be TRUE
        - If a found flag is false, the starter code prints 'None' for the character

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

#define MAX_LEN 100

void commonChars(char arr[][MAX_LEN], int n) {
    int common[26];

    // Initialize global record assuming all characters are common initially
    for (int i = 0; i < 26; i++) {
        common[i] = true;
    }

    // Process each string
    for (int i = 0; i < n; i++) {
        bool temp[26] = {false}; // Temporary record for the current string

        // Mark characters present in the current string
        for (int j = 0; arr[i][j] != '\0'; j++) {
            temp[arr[i][j] - 'a'] = true;
        }

        // Update the global record by taking the intersection
        for (int k = 0; k < 26; k++) {
```

```
            common[k] = common[k] && temp[k];
        }
    }

    printf("Common characters: ");
    bool found = false;

    // Print the common characters
    for (int i = 0; i < 26; i++) {
        if (common[i]) {
            printf("%c", i + 'a');
            found = true;
        }
    }

    if (!found) {
        printf("None");
    }
    printf("\n");
}
```

- Problem 2: zipped linked list
  - Purpose
    - Structural pointer manipulation problem → cannot traverse backwards because it is singly linked
      - Breaks the problem into three operations
  - Strategy
    - Need to have a pattern of 1st to kth to 2nd to (k-1)th
  - Helper functions
    - Find the middle of list
      - Initailize two pointers: slow and fast
        - Slow moves one node at a time (slow = slow->next), while fast moves two nodes at a time (fast = fast->next->next)
          - When the fast pointer reaches the end of the list/set to NULL, the slow node would be in the middle
    - Reverse the second half
      - Nodes from the middle to the end need to be reversed to traverse them sequentially during the merge
        - Take the second half of the list (from slow->next) and disconnect it from the first half by setting slow->next = NULL
        - Create a helper function to reverse the second half
          - Need three pointers (prev, curr, and next) to iterate through sublist and flix the ->next direction for each node
    - Merge the two halves
      - Set a pointer to p to the head of first half of list, and p2 to the head of the reverse second half
      - Iterate through both lists simultaneously
      - Temporarily save the next nodes (next1 = p1->next and next2 = p2->next)
      - Wire p2 to point to p2, and p2 to point to next1
      - Advance p1 and p2 to next1 and next2, and repeat until fully zipped

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

typedef struct Node {
    int data;
    struct Node *next;
} Node;

// Helper function to reverse a singly linked list
Node* reverseList(Node* head) {
    Node* prev = NULL;
    Node* curr = head;
    Node* next = NULL;

    while (curr != NULL) {
        next = curr->next;
        curr->next = prev;
        prev = curr;
        curr = next;
    }
    return prev;
}

void zipList(Node **headRef) {
    // Edge case: Empty list or a list with only one node
    if (headRef == NULL || *headRef == NULL || (*headRef)->next == NULL) {
        return;
    }

    // Step 1: Find the middle of the linked list using slow/fast pointers
    Node* slow = *headRef;
    Node* fast = *headRef;

    while (fast->next != NULL && fast->next->next != NULL) {
        slow = slow->next;
        fast = fast->next->next;
    }

    // Step 2: Reverse the second half of the list
    Node* secondHalf = slow->next;
    slow->next = NULL; // Disconnect the first half from the second half
    secondHalf = reverseList(secondHalf);

    // Step 3: Merge the two halves alternately
    Node* p1 = *headRef;
    Node* p2 = secondHalf;

    while (p2 != NULL) {
        Node* next1 = p1->next;
```

```
            Node* next2 = p2->next;

            p1->next = p2;
            p2->next = next1;

            p1 = next1;
            p2 = next2;
        }
    }
```

EXAM 1 - LABS

- Lab 1: bitwise operations and standard I/O
  - Kernighan's Algorithm in GDB
    - Understand the expression v = v & (v-1)
      - Subtracting 1 from a binary number flips its rightmost 1-bit to 0, and flips all trailing zero bits to 1s
      - Bitwise AND with the original v, it effectively clears the rightmost 1-bit back to a zero
        - Commands like break 8 to set a breakpoint, run to start execution, and p/t v to print the variable b in binary format
          - Helps observe but-flipping in real time
  - Method for completing
    - GDB and Parity
      - Compile with symbols
        - Compile parity
  - Robust I/O Loops
    - while(scanf('%lf", &x) ==1) loop is important for the running average program
      - %lf format is required to read a double
      - Because scanf returns the number of successfully parsed items, the check for if it equal to one handles termination
        - For invalid user inputs or an end of file signal

```c
#include <stdio.h>

int main(void) {
    double x;
    double total = 0.0;
    double count = 0.0;
    double average = 0.0;

    // The while loop continues as long as scanf() returns 1[cite: 220].
    while (scanf("%lf", &x) == 1) {
        total += x;
        count += 1.0;
        average = total / count;

        // Formatted to match the required output[cite: 222].
        printf("Total=%f Average=%f\n", total, average);
    }

    return 0;
}
```

- Lab 2: Build Automation and Integer Arithmetic
    - Makefiles
        - Modify the makefile to handle new executables
            - Target update: change the default all rule so it lists both ex-factorial and catalan as dependencies, which forced make to build them both
            - Clean rule: update clean rule to remove both ex-factoral and catalan executables as wll as any .o files
    - Robust factorial
        - Debugging: use GDB to step through ex-factorial.c and fix syntax or logical error
        - Input validation: check if the input integer is less than 0, need to print error message before calling factorial() function
    - Catalan numbers
        - Need to implement without floating-point numbers
            - Base case: sequence defines the first term to be 1 when n is zero
            - Recursive case:
                - Do (4*n -2) * catalan_number(n-1)/(n-1)

2. Robust Factorial Input Validation

Inside `ex-factorial.c`, you must ensure the program only computes the factorial for non-negative integers.

C

```c
// Inside your main() function in ex-factorial.c:
int n;
if (scanf("%d", &n) == 1) {
    if (n < 0) {
        // Terminates after printing the exact error string
        printf("Error: Please enter a non-negative integer.
        return 1;
    }
    // Proceed to call factorial(n) here
}
```

3. Catalan Numbers

The `catalan_number` function must use the recurrence relation $C_k = \frac{4k-2}{k+1}C_{k-1}$ without using any floating-point numbers or operations.

C

```c
unsigned long int catalan_number(int n) {
    // Base case: C_0 = 1 [cite: 247]
    if (n == 0) {
        return 1;
    }
    // Compute the numerator first to avoid integer truncation,
    // strictly avoiding floating-point math
    return (4 * n - 2) * catalan_number(n - 1) / (n + 1);
}
```

- Lab 3: Dynamic String Concatenation
    - Implementing argvcat
        - Implement char *my_strcat(chat *s1, char *s2) to dynamically allocate memory for concatenated strings
            - Allocation: need to calculate the length needed for the string
                - size_t len = strlen(s1) + strlen(s2) + 1
                    - The +1 is for the NULL
            - Concatenation: will not modify s1 or s2, but will use strcpy() to copy s1 into the allocated memory
                - Will then strcat() to append s1 to s2
                - Returns pointer after the concatenation
            - Preventing leaks: must capture the pointer returned from the concatenation step
                - Need to do free() to remove the memory used after it is not needed

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

// Pre-existing function you are required to call on failure [cite: 281]
extern void my_error(char *msg);

char *my_strcat(char *s1, char *s2) {
    // Calculate exact lengths.
    size_t len1 = (s1 != NULL) ? strlen(s1) : 0;
    size_t len2 = (s2 != NULL) ? strlen(s2) : 0;

    // Allocate space for both strings plus the null terminator [cite: 277]
    char *result = (char *)malloc(len1 + len2 + 1);

    // If malloc() fails, call my_error() to print an error message and exit [
    if (result == NULL) {
        my_error("Memory allocation failed");
    }

    // Initialize the new string to empty
    result[0] = '\0';

    // Concatenate without altering the original s1 or s2 strings [cite: 278,
    if (s1 != NULL) strcat(result, s1);
    if (s2 != NULL) strcat(result, s2);

    return result;
}
```

- Lab 4: Singly Linked Lists
    - Consists of two parts
        - Deleting node
            - Need to implement delete_node(node *head, int v) to remove the integer from the list and return the new head pointer
            - Head deletion
                - If head is NULL, call the error message error_message(ERR_NOTFOUND)
                - If target v is in the head node, store head->next in a memporary pointer, free the old head, and make the temporary pointer the new head
            - Traversal deletion
                - Deeper targets uses a prev pointer and curr pointer to traverse the list

        - When curr->data == v, bypass the node by setting prev->next = curr->next
            - Free the curr node after and return what the original head was
    - Not found
        - When the traversal loop finishes without finding v, calls error_message(ERR_NOTFOUND) and returns head
- List reversal
    - Need to implement reverse_head(node *head) to reverse list pointer <u>without</u> allocating new nodes
        - Setup pointers
            - Initialize three pointers:
                - prev = NULL
                - curr = head
                - next = NULL
        - Reversal loop
            - Will iterate while curr != NULL
                - Save the next node next = curr->next
                - Reverse the current node's pointer curr->next = prev
                - Advance prev: prev = curr
                - Advance curr: curr = next
        - Return new head
            - Once curr != NULL, prev would be pointing to the original tail node → return prev as the new head

Node Deletion Code

C

```c
node *delete_node(node *head, int v) {
    // Check if list is empty
    if (head == NULL) {
        error_message(ERR_NOTFOUND); // Print error if v is not on the list [c
        return NULL;
    }

    // Case 1: The node to delete is the head node
    if (head->data == v) {
        node *temp = head->next;
        free(head);
        return temp; // Return the new head [cite: 308]
    }

    // Case 2: The node to delete is further down the list
    node *curr = head->next;
    node *prev = head;

    while (curr != NULL) {
        if (curr->data == v) {
            prev->next = curr->next; // Bypass the target node
            free(curr);
            return head;
        }
        prev = curr;
        curr = curr->next;
    }

    // If the loop finishes without finding the value [cite: 309]
    error_message(ERR_NOTFOUND);
    return head;
}
```

Reverse List Code

C

```c
node *reverse_list(node *head) {
    node *prev = NULL;
    node *curr = head;
    node *next = NULL;

    // Change the next fields of each node so they are linked in reverse [cite: 3
    while (curr != NULL) {
        next = curr->next; // Temporarily store the next node
        curr->next = prev; // Reverse the pointer

        // Move tracking pointers forward
        prev = curr;
        curr = next;
    }

    // Return the address of the new head node (originally last) [cite: 372]
    return prev;
}
```

HW REVIEW

- HW 1a: Approximating pi
  - Topics covered
    - Standard I/O, for loops, floating-point arithmetic
  - How to solve
    - Initialize pi to 0.0 and a tracking variable for the 16^i denominator starting at 1.0
    - Use a for loop that iterates from i = 0 to n
      - Inside loop, calculate the formula for approximating pi
      - Multiply the results of the fractions by the current value of the 16^i tracking variable → then add to the value to pi
    - Divide the 16^i tracking variable by 16.0, which simulates 1/16^i without calling a power function

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

int main() {
    int n, i;
    printf("n = ");
    scanf("%d", &n);

    double pi = 0.0;
    double factor = 1.0; // Acts as 1 / 16^i

    for (i = 0; i <= n; i++) {
        double term = (4.0 / (8 * i + 1)) -
                      (2.0 / (8 * i + 4)) -
                      (1.0 / (8 * i + 5)) -
                      (1.0 / (8 * i + 6));

        pi += term * factor;
        factor /= 16.0; // Progress to the next power of 16 without pow()
    }

    printf("PI = %.10f\n", pi);
    return 0;
}
```

- HW 1b: Happy Numbers
  - Topics covered
    - While loops, modulo arithmetic, digit extraction
  - How to solve
    - Set a while(m != 1 && m != 4) loop
      - Initialize a sum 0 and temporary variable temp = m
      - Set an inner while(temp < 0) loop to extract digits
        - Digit = temp % 10
          - Square the digit and add to sum
          - Divide temp by 10 to move to next digit
        - Set m = sum and print
      - Once outer loop breaks, check if m == 1 to print either success or failure message

- HW 2a: Recursion and 2D Arrays
    - Topics Covered
        - Recursive backtracking, problem partitioning
    - How to Solve
        - Base cases
            - If count == 0 and value == 0, a valid combination was found
                - Return 1
            - If count == 0, value < 0 or bound <= 0, the path is invalid
                - Return 0
        - Recursive Step 1 (includes the bound)
            - Include the current odd bound in the sum
            - Recursively call oddSumHelp(count-1, bound-2, value-bound)
                - If this returns 1, print bound and return 1
        - Recursive Step 2 (excludes the bound)
            - Happens if step 1 returns 0 → current bound does not lead to a solution
            - Exclude the bound and try again oddSymHelp(count, bound-2, value)

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

int oddSumHelp(int count, int bound, int value) {
    // Base cases
    if (count == 0 && value == 0) return 1; // Valid combination found
    if (count == 0 || value < 0 || bound <= 0) return 0; // Invalid path

    // Recursive Step 1: Try including the current largest odd number (bour
    if (oddSumHelp(count - 1, bound - 2, value - bound)) {
        printf("%d ", bound);
        return 1;
    }

    // Recursive Step 2: If the current bound doesn't work, skip it
    return oddSumHelp(count, bound - 2, value);
}

void oddSum(int count, int bound, int value) {
    if (value <= 0 || count <= 0 || bound <= 0) return;
    if (bound % 2 == 0) bound -= 1; // Ensure bound is odd

    if (!oddSumHelp(count, bound, value))
        printf("No solutions.\n");
    else
        printf("\n");
}

int main(int argc, char *argv[]) {
    if (argc != 4) return -1;
    int count = atoi(argv[1]);
    int bound = atoi(argv[2]);
    int value = atoi(argv[3]);

    oddSum(count, bound, value);
    return 0;
}
```

- HW 2b: Bounded 2D Random Walk
    - Topics Covered
        - 2D array manipulation, random number generation rand(), Monte Carlo simulation
    - How to Solve
        - Create 2D boolean array of size (2n + 1) by (2n + 1) to track coordinates visited
        - Shift all coordinates by +n to avoid negative array indices
            - Origin goes from [0][0] to [n][n]
        - Use while loop that checks if x == -n || x == n || y == -n || y = n
            - Inside loop, rand() % 4 is used is used to update x and y
                - 0 = up, 1 = right, 2 = down, 3 = left
            - If visited[x+n][y+n] is false, mark as true and increment a unique_visited counter
        - When loop terminates, calculate unique_visited/total area of walls
            - Area = (2n - 1)*(2n - 1)

c

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

double two_d_random(int n) {
    // 2D Array mapping: shift coordinates by +n to avoid negative indices
    int size = 2 * n + 1;
    int visited[size][size];

    // Initialize visited array to 0
    for(int i = 0; i < size; i++) {
        for(int j = 0; j < size; j++) {
            visited[i][j] = 0;
        }
    }

    int x = 0, y = 0;
    visited[x + n][y + n] = 1;
    int unique_visited = 1;

    // Walk until a boundary is hit
    while (x > -n && x < n && y > -n && y < n) {
        int r = rand() % 4;
        if (r == 0) y++;      // Up
        else if (r == 1) x++; // Right
        else if (r == 2) y--; // Down
        else if (r == 3) x--; // Left

        // If it's a new coordinate inside the grid bounds
        if (x > -n && x < n && y > -n && y < n) {
            if (!visited[x + n][y + n]) {
                visited[x + n][y + n] = 1;
                unique_visited++;
            }
        }
    }

    // Total number of points strictly inside the bounds is (2n - 1) * (2n - 1)
    double total_inside_points = (2.0 * n - 1.0) * (2.0 * n - 1.0);
    return (double)unique_visited / total_inside_points;
}

// (The provided main() function would go here)
```

- HW 3a: 3D Diffusion
    - Topics Covered
        - malloc/free, 3D to 1D array mapping, Euclidian distance logic
    - How to Solve
        - Dynamically allocate a 1D array to represent a 3D grid
            - Size of grid = (2n+1)(2n+1)(2n+1)
        - Simulate each particle using a loop of n steps, with rand() % 6 for updating x, y, or z

- When particle finished moving, map its 3D coordinates back into a 1D index
    - (x+n) + (y+n)*(2n+1) + (z+n)*(2n+1)*(2n+1) for the transformation
    - Increment the value at the grid space
- Need to calculate the density of the space <u>without sqrt()</u>
    - Square both sides so its x^x + y^y + z^z is less than or equal to (r*n)(r*n)
    - Add particle counts of all valid coordinates and divide by m

<u>PHOTO IN GITHUB</u>

- HW 3b: Monopoly
    - Topics Covered
        - Arrays of structures, cyclic/module arithmetic, state machines
    - How to Solve
        - Create TPlayer and TProperty arrays as intialized
        - Loop up to max_rounds
            - Inside, loop through every player sequentially
        - Update the player's location using dice rolls
            - loc = (loc + die1 + die2) % n
            - If the new loc is smaller than the old loc, add n to their balance
                - Means they passed Go
        - Check the property at prop[loc]
            - If owner_id == -1, it is set to player.id
            - If owned by another player, deduct prop[loc].rent from balance and ad to owners balance
                - If the player balance drops below zero, break <u>all</u> loops entirely

<u>PHOTO IN GITHUB</u>

- HW4: Hash Table and Linked Lists
    - Topics Covered
        - Hash table, collision resolution, spatial boundaries
            - Chaining with linked lists and wrapping
    - How to Solve
        - Movement and wrap around
            - Use rand() % 4 for movement, and if the host moves out of bounds (x > k) wrap them to the <u>opposite side</u> with x = -k
                - Makes the grid into a torus
        - State updates
            - For every infected host I, increment their timer t
                - When t == T, change the state to recovered
        - Hash table population
            - Clear the has array p_arr for every round
            - Iterate through all the hosts
                - If a host is inflected (I), pass the coordinates into the idx and hash functions
                    - Use modulo N (% N) to find the valid array bucket, and insert a dynamically allocated linked list node with the host at the <u>front</u> of the bucket
        - Infection lookup
            - Iterate through susceptible hosts (S)
                - Calculate the S host's hash bucket to minimize time spent
                    - Traverse only the linked list in that bucket, andif a node in the list matches the S host's coordinates, change its state to I

        - Memory Cleanup
            - Must traverse every linked list in the hash tale and free() the associated nodes before the start of the next round
                - Will crash the memory if this doesnt happen
            - Continue simulation until no l hosts remain

CODE IS IN GITHUB

Exam 1 - Code for Thought Explanations

- sum_n_bytes.c
    - Topic
        - Pointer casting and byte level access
    - Trick
        - Memory was allocated for an unsigned integer, which is usually 4 ytes
            - When doing p[1], compiler jumps forward 4 bytes to get to the next integer, but we want to sum the first n bytes
    - Strategy for solving
        - We need to cast a pointer when needing to manipulate data with a different precision than the one allocated
            - unsigned char *byte_ptr = (unsigned char*)p
                - Unsigned char is guaranteed to be only one byte in C, so now evaluating byte_ptr[i] leads to it jumping forwards once instead of four times

```c
unsigned sum_n_bytes(unsigned *p, int n)
{
    unsigned char *byte_ptr = (unsigned char *)p;
    unsigned sum = 0;

    for (int i = 0; i < n; i++) {
        sum += byte_ptr[i];
    }

    return sum;
}
```

- sum-2.c
    - Topic
        - Integer overflow and memory limits
    - Trick
        - Standard int is 32 bits, which means its maximum value is 2^32 -1
            - Sum of numbers 1 to 1,000,000 is larger than the maximum value, which leads to an integer overflow
    - Strategy for solving
        - When needing to solve problems involving larger sums, factorials, or file sizes, default to using a 64-bit integer
            - Change int sum into long long sum
            - Need to also update the format identifier in printf
                - Change %d into %lld

```c
#include <stdio.h>

int main(void) {
    long long sum; // Changed to long long to prevent overflow
    int i, n;
    sum = 0;
    printf("Enter n:\n");
    scanf("%d", &n);
    i = 1;
    while (i <= n) {
        sum = sum + i;
        i = i + 1;
    }
    // Changed format specifier to %lld for long long
    printf("Sum from 1 to %d = %lld\n", n, sum);

    return 0;
}
```

- Swap-2.c
    - Topic
        - Pass-by-reference and double pointers
    - Trick
        - Swapping two integers needs you to pass pointers (int*)
            - Exam asks to swap <u>two pointers</u>, meaning you need to pass the pointer to the pointer instead of to a function
                - Pointing to a function makes a copy instead of a new one
    - Strategy for solving
        - Use double pointers when you need to change the address stored in the caller's scope
            - void swap_pointer_int(int **p1, int **p2)
            - Inside the function, dereferncing once (*p1) accesses the <u>original pointer</u>
            - int *temp = *p1 saves the address, and *p1 =*p2 overwrites the original pointer with new address

```c
// Above test_swap_pointer_int
void swap_pointer_int(int **p1, int **p2) {
    int *temp = *p1;
    *p1 = *p2;
    *p2 = temp;
}

// Inside test_swap_pointer_int
swap_pointer_int(&pa, &pb);

// Above test_swap_person
void swap_person(person_t *p1, person_t *p2) {
    person_t temp = *p1;
    *p1 = *p2;
    *p2 = temp;
}

// Inside test_swap_person
swap_person(&x, &y);
```

- taxicab-number.c
  - Topics
    - Algorithm bounds and early exits
      - Efficient loops
  - Trick
    - Iterating i and j up to N every time is inefficient and can lead to overflows
  - Strategy
    - Bound your loops mathematically and use break statement to exit as needed
      - Outer loop bounds i such that i*i*i < n
        - No reason to test an i whose cube is already larger than the target value n
      - Inner loop starts a j = i (to avoid same combinations) and stops when i^3 + j ^3 > n
      - When count hits 2, break out fo the loop to lock in the smallest n

```
for(n=1; n<=1000000; n++)
{
        int count = 0;
        for (i = 1; i * i * i < n; i++) {
            for (j = i; i * i * i + j * j * j <= n; j++) {
                if (i * i * i + j * j * j == n) {
                    count++;
                }
            }
        }
        if (count >= 2) {
            break; // Early exit once we find the smallest n
        }
}
```

- shortest-2.c
  - Topics
    - Structure pointers and dynamic memory
  - Trick
    - Function must return a pointer to a pair of points (t_pair *)
      - Declaring t_pair pair as a local variable and dereferencing pair will lead to a fault → local variables are destroyed when the function ends, leading to a dangling pointer
  - Strategy
    - Need to allocate the struct on the heap p_pair = malloc(sizeof(t_pair))
      - p_pair is a pointer to a struct, cannot use the dot operator (p_pointer.p1)
      - O(n^2) loops need to make sure that the pointer doesnt point to itself
        - for(i = 0; i< n; i++) and for(j = i + 1; j < n; j++)

```
double min_d = squared_dist(&(points[0]), &(points[1]));
p_pair->p1 = points[0];
p_pair->p2 = points[1];

for (i = 0; i < n; i++) {
    for (j = i + 1; j < n; j++) {
        double current_d = squared_dist(&(points[i]), &(points[j]));
        if (current_d < min_d) {
            min_d = current_d;
            p_pair->p1 = points[i];
            p_pair->p2 = points[j];
        }
    }
}
```

- leibniz.c

- Topics
  - Implicit type promotion and division
- Trap
  - The expression 8/(4*i + 1)/(4*i + 3) had every single literal and variable be an integer
    - Integer division in C truncates decimals, which can lead to failures when doing evaluations
- Strategy
  - Force the compile to promote the calculation to floating-point arithmetic
    - Chage the 8 to an 8.0 to see that the expression is a double/int
      - Forces the to be promoted to double, which preserves the decimals

```
        for(i=0; i<=n; i++)
        {
        // Changed 8 to 8.0 to force floating-point division
                sum += 8.0 / (4 * i + 1) / (4 * i + 3);
        }
```

- hex-2.c
  - Topic
    - In-place array reversal
  - Trap
    - Modulo arithmetic extracts the least significant digit first, which when appending to array, makes the final string backward
  - Strategy
    - Use a lookup table to map integers to characters safely
      - char digits[] = {‘0’, …, ‘F’}, hex[k] = digits[d % 16]
    - Append null terminator to make it a valid C string
    - Use standard swap loop to reverse array in place
      - Swap index i with index len-1-i, stops at halfway point (len/2)

```
void dec_hex(int d, char hex[])
{
        char digits[] ={'0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'A', 'E
        int k = 0;

    // Handle 0 explicitly
    if (d == 0) {
        hex[k++] = '0';
    } else {
        while (d > 0) {
            int remainder = d % 16;
            hex[k] = digits[remainder];
            k++;
            d = d / 16;
        }
    }
        hex[k] = '\0';

        // Reverse the array
        for (int i = 0; i < k / 2; i++) {
            char temp = hex[i];
            hex[i] = hex[k - 1 - i];
            hex[k - 1 - i] = temp;
        }
}
```

- dice.c

- Topics
    - Simulating probabilities and forcig floating-point evaluation during returns
- Trap
    - Function tracks number of successful trials with an int counter, total trials is a long number
        - Reutrning count/trials performs integer division, but because count is always less than trials, it would result in a truncated value of 0
- Strategy
    - Generate random bounds
        - rand() % 6 + 1 to get a range from 1 to 6 for each of the dice
    - Force type promotion on the return
        - return (double)count / trials; casts the num to a float, and forces the return of a probability ratio

```
double cum_prob(int k, long trials) {
    long count = 0;

    for (long i = 0; i < trials; i++) {
        int die1 = rand() % 6 + 1;
        int die2 = rand() % 6 + 1;
        int die3 = rand() % 6 + 1;

        if (die1 + die2 + die3 >= k) {
            count++;
        }
    }

    return (double)count / trials;
}
```