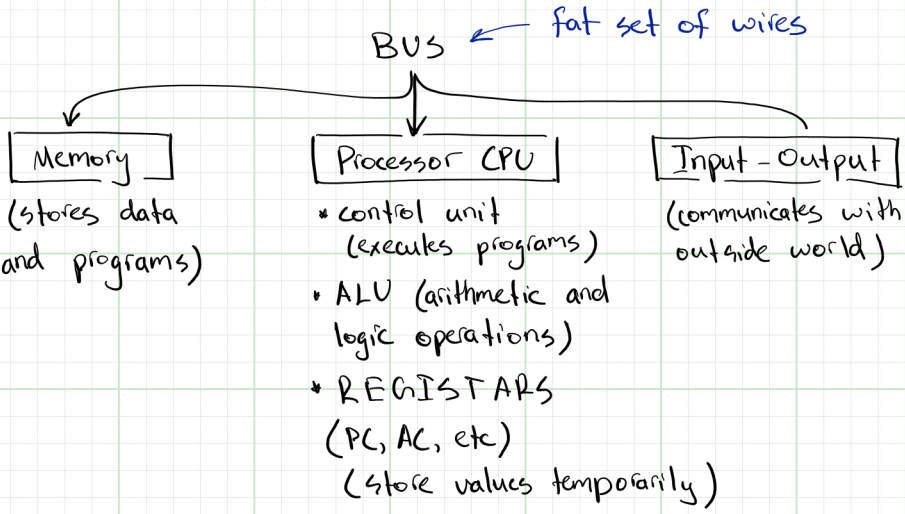
- C is a compiled language (not interpreted)
- Syntax errors → compile time
- logical errors → run time

Basic Computer System



- what is "word size"?
  - ↳ the size in bits that the processor uses to handle data
  - ↳ register size

Creating Executable

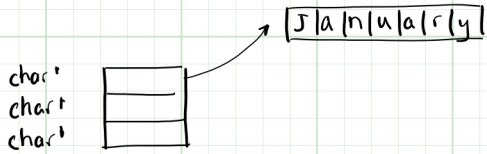
- pre-processor
- compiler
- assembler
- linkers

Array of Strings

- char array[][10] ← Yes can edit values
- char *months[12]; ← cannot modify values

month[0] = "January"

month[1] = "February"

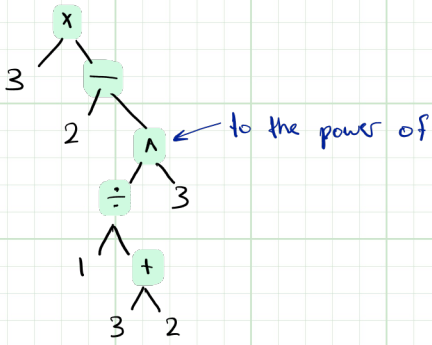


Recursion

- problems within a problem
- recursion continues until
  - some small indivisible problem is reached
  - a base case is reached (another stopping condition)

$3 \times (2 - (1 \div (3+2))^3) =$

base case



- when function calls itself.

→ Eg: Loop V.s Recursion

Recursive:

```
walk () {
    if (at wall)
        stop ;
    else
        walk one step;
        walk() ;
}
```

function calls itself again

```
cut (chunk) {
    if (chunk < 100)
        return
    else {
        cut chunk in half ;
        cut (right chunk)
        cut (left chunk)
}
```

→ Eg: $f(n) = \begin{cases} 2f(n-1)+1 & n>0 \\ 3 & n=0 \end{cases}$

make a recursive function

so... $f(3) = 2 * f(2) + 1$

$= 2 * (2 * f(1) + 1) + 1$ — base case reached

$= 2 * (2 * (2 * f(0) + 1) + 1) + 1$

$= 2 * (2 * (2 * 3 + 1) + 1) + 1$

$= 2 * (2 * (7) + 1) + 1$

$= 2 (15) + 1$

$= 31$

implementing in C:

```
int Function (int n) {
  int value;
    if (n == 0)
      value = 3;        // base case
    else
      value = 2 * Function (n-1) + 1;
    return value;
}
```

→ Eg: Factorial Function — base case

so... $3! = 3 \times 2 \times 1$

$5! = 5 * 4! = 5 * 4 * 3! \dots$

$n! = n(n-1)!$

```
int factorial (int n) {
  if (n == 0)
    return 1;
  return (n * factorial(n-1));
}
```

Recursion Tips

- identify base case
- find out how to get to base case by calling function itself

multiply (m, n)

$$f(m,n) = \begin{cases} m + m(n-1) & n > 1 \\ m & n = 1 \end{cases}$$

- **Recursion : GCF**

smaller − (big − small)

GCF (n, m)

base case: if (n == m), return n

else

result = GCF (smaller

```
int gcd(int n, int m) {
    if (m == n)
        return m;
    if (n < n)
        return gcd(n, m);
    return gcd(n, m-n);
}
```

Week 8: Lec 2

* Adding the first natural numbers

```
int myadd(int a) {
    int ans;
    if (a == 0 || a == 1)
        ans = a
    else
        ans = a + myadd(a-1);
    return ans;
}
```

$\mathbb{N} = \{1, 2, 3, 4 \cdots\}$

ans = 1 + 2 + 3 + 4 + ⋯

3 + 2 + 1 = 7

Winding Phase

myadd(3) ⟶ 3 ≠ 1, calls ans = 3 + myadd(2)

myadd(2) ⟶ 2 ≠ 1, calls ans = 2 + myadd(1)

myadd(1) = 1 = 1, so returns 1

2 + 1

1

* Multiplying 2 numbers a and b : add a to itself b times

eg: 6 × 3 = 6 + 6 + 6 (3 times)

```
int multiply(int a, int b)
    if (a =
```

* Calculating Power of

eg: $x^4 = x * x * x * x$

$x^6 = x * x^5 = x * x * x^4 = x * x * x * x^3 \ldots$

given $x^n = x * (x^{n-1})$

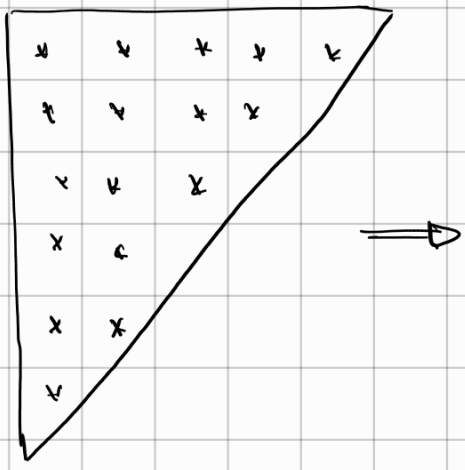
* Printing Star Patterns

* * * * * *

base case

```
void printstars (int n)
{  if (n == 1)
      printf("* \n");
   else {
      printf("* ");
      printstars(n-1);
   }
}
```

* Print triangle pattern




```
Void printtriangle (int n)
   if (n == 1)
      printrow(n);
   else {
      printrow(n);
      printtriangle(n-1);
   }
```

- Print inverted triangle

printT(5)

print triangle with (n-1) rows

print the row

```
void printT(int n)
  if (n>0) {
    printT(n-1);
    printrow(n);
  }
}
```

goes to base case

Recursion

Comparison	Recursion	Iteration
Basic	The statement in a body of function calls the function itself.	Allows the set of instructions to be repeatedly executed.
Format	In recursive a function, only termination condition (base case) is definitive.	Iteration includes initialization, condition, execution of statement within loop and update (increments and decrements) the control variable.
Termination	A conditional statement is included in the body of the function to force the function to return without recursion call being executed.	The iteration statement is repeatedly executed until a certain condition is reached.
Condition	If the function does not converge to some condition (base case), it leads to infinite recursion.	If the control condition in the iteration statement never become false, it leads to infinite iteration.
Infinite Repetition	Infinite recursion can crash the system.	Infinite loop uses CPU cycles repeatedly.

Bruno Korst, P.Eng. - Winter 2024 40

Comparison	Recursion	Iteration
Applied	Recursion is always applied to functions.	Iteration is applied to iteration statements or "loops".
Stack	The stack is used to store the set of new local variables and parameters each time the function is called.	Does not use stack.
Overhead	Recursion possesses the overhead of repeated function calls.	No overhead of repeated function call.
Speed	Slow in execution.	Fast in execution.
Size of Code	Recursion reduces the size of the code.	Iteration makes the code longer.

Week 8: Lec 3

* What is this doing?

```
int whatsup (int first, int last, int array[]) {
    if (first == last)
        return (array[first])
    else
        return (array[first] + whatsup(first+1, last, array))
}
```

  * sums up elements in array:

[1 8 3 2] ① 1 + (8 + (3 + (2))) = 14

* Eg 2

```
int wot(int x, int y) {
    return y==0 ? 1 : x * wot(x, y-1);
}
```

wot(2,4) → 2*(wot(2,3)) ... 2*2*2*2*1 = 2^4

  * Apply power using recursion

* String Recursion
  - search inner array
  - palindrome

* Palindrome Example
  - base case : char at centre

* Robot Walk Example
  - robot can take eigther 1-m or 2-m steps.

Intro to Data Structures

```
struct
{
    int itemNumber, quantity ;
    char size;
    double price;
} x, y;
```

} members / fields / components

↑ structure variables (contain all members / components)

→ You can use a tag to declare struct

```
struct Student {
    int student_number;
    char stdname[];
    int ranking ;
} student1, student2;
```

← declared struct named Student

↑ these are now 2 structs of the Student type

- members of struct are placed sequentially in RAM w/out spaces
- variable names declared inside struct are only in scope inside struct

→ Declaring and Defining

```
struct {
    int number;
    char name[NAME_LEN+1];
    int onhand;
} part1 = {528, "Disk drive", 2};
```

part1 :

528
Disk drive
2

→ To access member within a struct

↳ dot operator

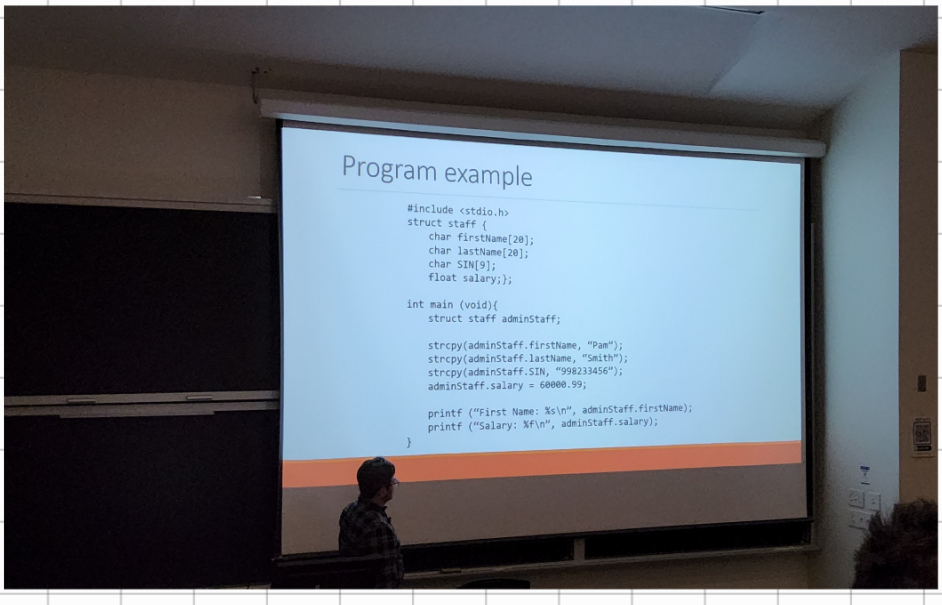
```
printf("%d\n", part1.number);
```

← prints out "528"

→ Can declare array of structs

```
struct staff allStaff[50]
```

← array containing 50 of struct "staff" named "allstaff"



→ TypeDef

```
typedef struct {
} Part;
```

⟶ declare easily now

```
Part part1, part2;
```

After declaring struct

```
Neutron type Myneuron;
Neutron type *pn;        ——declaring pointer to pn
pn = &Myneuron;
pn -> number = 50;       ← change member "number"
```

↖ used to reach into struct to access "number"

```
Neurontype *pn2;
pn2 = (Neurontype *) malloc (sizeof (Neurontype));
```

↙ dynamically allocate a new pn2.

→ Passing structs to funtions

(1) by value : `void function ( Neurontype) { }`

(2) by reference: `void function (Neurontype *pn) { }`

→ **Calloc**: allocates memory and also initializes it all to zero

Lists

* linked list: a chain of structures each with a node containing a pointer to the next node

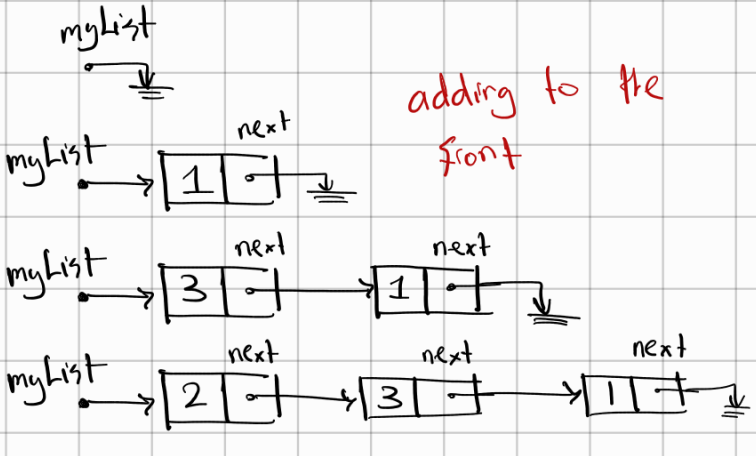
* operations → start new list
  → insert a new element arrays → fixed size
  → search for item
  → delete an item array → can't free the space

```
typedef struct node{
    int value  <- payload
    struct node *next  <- pointer to next node
} Node
```

①
```
Node *newNode;
Node *p = (Node *) malloc (sizeof (Node));
```

* pointer to first element ← head
* last element points to NULL ← tail


```
Node *myList = NULL;
myList = insFrList (myList, 1);
myList = insFrList (myList, 3);
myList = insFrList (myList, 2);
```



→ Printing all Elements

"start from top of list, print all values till you reach tail, or the NULL"

```
void printlist (Node *myList) {
    Node *current = myList;
    while (current != NULL) {
        printf("%d\n", current → value);
        current = current → next;
    }
}
```

print value of current

pass on the pointer to next node

→ Deleting Element

① reorganize pointers

② free the node

Creating a list (in words)

```
Node *myList = NULL; //beginning of the list

myList = newNode(1,NULL); //1 is the data passed
                          //to member "info" on the node
myList = newNode(3, myList);
```

- create a pointer to type **Node** (our typedef), make it point to NULL
- This "begins" the list – it's an empty list
- Insert a new node using **newNode** function, which
  - adds a value (**1**) to member **value**, and points to the next node: NULL
  - it also RETURNS A POINTER to type **Node**, which is assigned now to **myList**
- Can keep inserting nodes "at the top of" the list
- NEXT SLIDES: Substitute "link" by "next" and "info" by "value"

Lists: allocating a new node

- Using a function to do that:

-pass "data" and pointer to next node
-return pointer to newNode
-allocate space
-check if it is the end of the list
-use the parameters
    -store data in "value"
    -put link to next node in place;
-return the pointer to new node (created)

```
Node *newNode (int i, Node *np)
{
  Node *p = (Node *)malloc(sizeof(Node));
  if (p == NULL)
    printf("Error: out of mem!\n");
  else
  {
    p->value = i; //points member value
    p->next = np; //points member next
  }
  return p; //with the new values
assigned
}
```

Problem 1:

Write a function printDuplicates that (1) recieves pointer to int node of a linked list and (2) should find and print duplicates in the L.L.

```
typedef struct node {
    int info;
    struct node *link;
} Node
```

```
void printDuplicates (Node *Read) {
```

Problem 2

Write a function `splitSwap` that finds node with `info` equal to parameter `item`. The node is used to split the linked list into left and right.

↑ (containing nodes to left of node and including)

Function returns the modified linked list swapped L ↔ R

Problem 3

Complete the definition of non-recursive function copy so that it returns an exact copy of the linked list pointed to by variable head

```
typedef struct node {
    int info;
    struct node *link;
} Node, *NodePointer;

NodePointer newnode (int item, NodePointer, next) {
    NodePointer p = (NodePointer) malloc (sizeof(Node));
    if (p != NULL) {
        p -> info = item;
        p -> link = next;
    }
    return p;
}
```

Recursive Problem

- think about how to make problem smaller (to call recursively)
- After you get to the base case, you have to go back to the previous function calls and continue on with lines after where it stopped bc for "unfurling".

Strings

- 2 types: one with recursion and one without
- only use recursion if it says to
- string functions: strlen (excludes null char, returns # of chars)
  strcpy (copies everything till null char from one string into another string)

strncpy(s_1, s_2, 3)	strncpy	(copies "n" into dest from source)
strcat(s_1, s_2)	strcat	(chars in s_2 are added to end of s_1)
	strncat	(adds "n" numbes argument)
<0 ↳ strcmp("AB", "BA")	strcmp	(0 if same)
	strncmp	(compares "n" char)
strchr("Ab", "b")	strchr	(returns pointer to first occurence of char in a string)
	strstr	(returns pointer to first occurence of string in a string)

- stored like → `char s[] = "This is";`
  `char *s = "This is";`

String Functions

`puts(s)` : function prints string followed by newline character

↑ string identifier

`fgets(st, n, stdin)` : reads "n-1" char's and puts them in st

↑ array identifier

↑ reading and storing n-1 char's

↑ library call

`strlen(string)` : returns number of char's in string (not counting \0).

↑ string identifier

`char* strcpy(char* dest, char* src)` : copies src into destination including "\0"

↑ returns address of destination

`char* strncpy(char* dest, char* src, n)` : if n > src, it will copy everything including "\0" and just add "\0" for remaining n's.

↑ copies "n" characters

`char* strcat(char* dest, const char* src)` : replaces NULL of dest with first char of src

↑ address of dest

`char* strncat(char* dest, const char* src, n)` : concatenates "n" chars, does not add filler "\0"

`int strcmp(const char* s1, const char s2)`

↑ returns
- < 0 : s_1 befor s_2
- 0 : same string
- > 0 : s_1 after s_2

↓ similar idea for `strncmp()`

`char* strchr(const char* s, char c)` : returns null if not found

↖ returns pointer to the first occurence of "c" in string "s"

`char* strstr(const char* s1, const char* s2)` : returns null if not found

↖ returns pointer to the first occurence of s_2 in s_1