# Outline

- Briefly review the last class
- Pointers and Structs
- Memory allocation
- Linked lists

# C Structures and Memory Allocation

- A struct is a data structure that comprises multiple types, each known as a member
  - Example:
    - A student may have members: name (char[ ]), age (int), GPA (float or double), sex (char), major (char[ ]), etc

```
struct student {
    char name[20];
    int age;
    …
};
```

- Memory allocation
  - If we want to create a structure that can vary in size, we will allocate the struct on demand and attach it to a previous struct through pointers

# Struct Examples

Recommended

```
struct point {
    int x;
    int y;
};

struct point p1, p2;
```

p1 and p2 are both points, containing an x and a y value

```
struct {
    int x;
    int y;
}  p1, p2;
```

p1 and p2 are both points containing an x and a y, but do not have a tag (struct type name)

```
struct  point {
    int x;
    int y;
}  p1, p2;
```

same as the other two versions, but united into one set of code, p1 and p2 have the tag point

```
union a {
  int x;
  struct  {
    int x;
    int y;
  };
}
```

unnamed struct, no instance

# Accessing structs

- A struct is much like an array
  - The structure stores multiple data
    - To access a particular member, you use the . operator

      as in student.firstName or p1.x and p1.y
      - we will see later that we will also use - > to reference a field if the struct is pointed to by a pointer
    - To access the struct itself, use the variable name

      Legal operations on the struct are assignment, taking its address with &, copying it, and passing it as a parameter
      - Point p1 = {5, 10}; // same as p1.x = 5; p1.y = 10; only works when declaring p1
      - p1 = p2; // same as p1.x = p2.x; p1.y = p2.y;

# structs as Parameters

- Passing as a parameter:
  - void foo(struct point x, struct point y) {…}

- Returning a struct:

```
struct point createPoint(int a, int b)
{
    struct point temp;
    temp.x = a;
    temp.y = b;
    return temp;
}
```

# Inputting a struct in a Function

- We will need to do multiple inputs for our struct
  - let's write a separate function to input all the values into our struct
    - The code to the right does this

```
#include <stdio.h>

struct point  {
        int x;
        int y;  };

void getStruct(struct point);
void output(struct point);
void main( )    {
        struct point y = {0, 0};
        getStruct(y);
        output(y);
}

void getStruct(struct point p)    {
        scanf("%d", &p.x);
        scanf("%d", &p.y);
        printf("%d, %d", p.x, p.y);
}

void output(struct point p)    {
        printf("%d, %d", p.x, p.y);
}
```

# This doesn't work

Results:

Input two numbers:

10 10

10, 10

0, 0

## Why? C uses pass by copy

- the struct is *copied* into the function so that p in the function is different from y in main
- after inputting the values into p, nothing is returned, so y remains {0, 0}

# One Solution For Input

- In our previous solution, we passed the struct into the function and manipulated it in the function, but it wasn’t returned
  - What was passed into the function was a copy
    - So structs differ from arrays!
- In our input function, we can instead create a temporary struct and return the struct rather than having a void function

```
void main( )
{
        struct point y = {0, 0};
        y = getStruct( );
        output(y);
}
```

```
struct point inputPoint( )
{
    struct point temp;
    scanf("%d", &temp.x);
    scanf("%d", &temp.y);
    return temp;
}
```

# Pointers to Structs

- The previous solution had two flaws:
  - It required twice as much memory
    - we needed 2 points, one in the input function, one in the function that called input
  - It required copying each member of temp back into the members of the original struct
    - with our point type, that's not a big deal because there were only two members, but this may be undesirable when we have a larger struct
  - So instead, we might choose to use a pointer to the struct
    - We see an example next, but first…

# Pointers to Structs

```
struct foo  {          // a global definition, the struct foo is known in all of
    int a, b, c;       // these functions
};
foo fooInstance={1,2,3};
foo* a=&fooInstance;
```

| | Memory | Address |
|---|---|---|
| fooInstance.a | 1 | 1084 |
| fooInstance.b | 2 | 1088 |
| fooInstance.c | 3 | 1092 |
| a | 1084 | 1096 |

- If a is a pointer to a struct, then to access the struct’s members, we use (*a).x
- Or use the - > operator as in a->x

# Pointer-based Example

```
#include <stdio.h>
struct foo  {                // a global definition, the struct foo is known in all of
    int a, b, c;             // these functions
};

//  function prototypes
void inp(struct foo *);                // both functions receive a pointer to a struct foo
void outp(struct foo);

void main( )   {
            struct foo x;              // declare x to be a foo
            inp(&x);                   // get its input, passing a pointer to foo
            outp(x);                   // send x to outp, this requires 2 copying actions
}

void inp(struct foo *x)
{                            // notice the notation here:  &ptr->member
            scanf("%d%d%d", &x->a, &x->b, &x->c);
}

void outp(struct foo x)                // same notation, but without the &
{
            printf("%d %d %d\n", x.a, x.b, x.c);
}
```

# Nested structs with pointers

- In order to provide modularity, it is common to use already-defined structs as members of additional structs
- Recall our point struct, now we want to create a rectangle struct
  - the rectangle is defined by its upper left and lower right points

```
struct point {
    int x;
    int y;
 }
struct rectangle {
    struct point pt1;
    struct point pt2;
}

struct rectangle r;
```

Then we can reference
r.pt1.x, r.pt1.y,
r.pt2.x and r.pt2.y

# Nested structs with pointers

- In order to provide modularity, it is common to use already-defined structs as members of additional structs
- Recall our point struct, now we want to create a rectangle struct
  - the rectangle is defined by its upper left and lower right points

```
struct point {
    int x;
    int y;
 }
struct rectangle {
    struct point pt1;
    struct point pt2;
}

struct rectangle r;
```

Then we can reference
r.pt1.x, r.pt1.y,
r.pt2.x and r.pt2.y

Now consider the following

```
struct rectangle r, *rp;
rp = &r;
```

Then the following are all equivalent

```
r.pt1.x
(*rp).pt1.x
rp->pt1.x
```

But not rp->pt1->x (since pt1 is not a pointer to a point)

# typedef

- typedef is used to define new types
  - The format is
    - typedef description name;
  - Where description is a current type or a structural description such as an array declaration or struct declaration
  - Examples:

```
typedef  int Length;           // Length is now equivalent to the type int

struct node {          // declares a node structure that contains
        int data;              // a data item and a pointer to a struct of type node
        struct node *next;
};
```

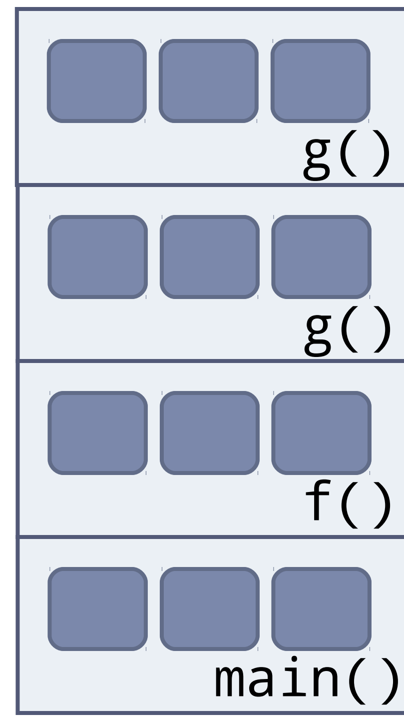
We can simplify our later uses of node by doing the following

typedef struct node aNode;     // this allows us to refer to *aNode* instead of *struct node*

# Overview of memory management

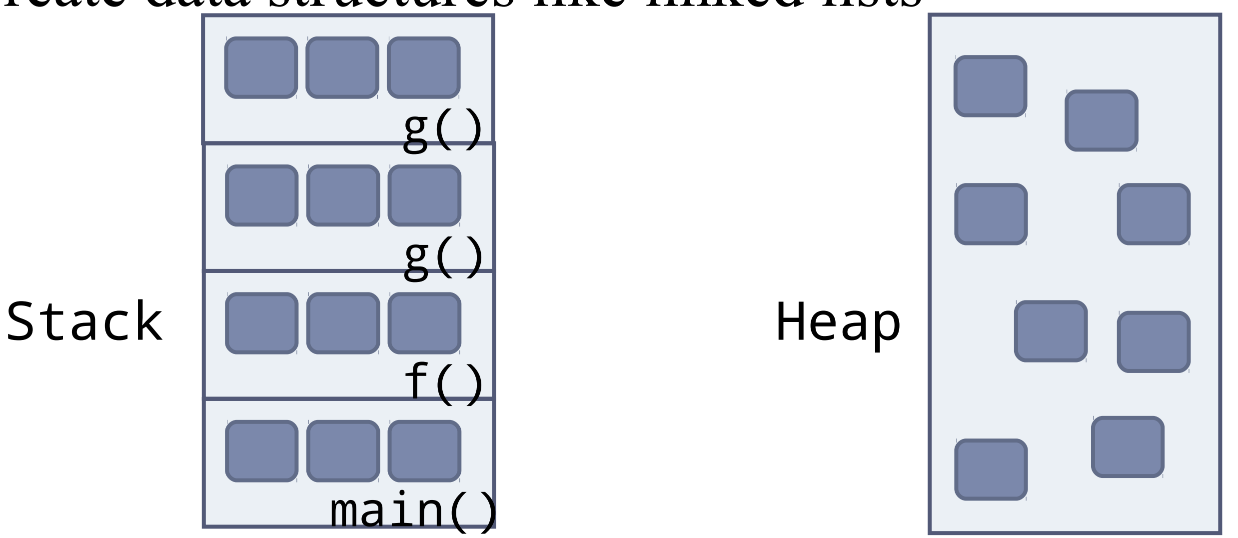
- To this point, we have been declaring pointers and having them point at already created variables/structures
- Stack-allocated memory
  - When a function is called, memory is allocated for all of its parameters and local variables.
  - Each active function call has memory on the stack (the current function call on top)
  - When a function call terminates, the memory is deallocated ("freed up")
- Ex: `main()` calls `f()`, `f()` calls `g()`

  `g()` recursively calls `g()`



# Heap-allocated memory (Dynamic allocation)

- However, the primary use of pointers is to create dynamic structures
  - Use heap allocation for persistant data
  - The pointer can point to a piece of memory that has just been created (allocated)
  - We will use this approach (memory allocation + pointers) to create data structures like linked lists



# malloc and calloc

- The two primary memory allocation operations in c are malloc and calloc
  - For most situations, we will use malloc:
    - pointer = (type *) malloc(sizeof(type));
    - This sets pointer to point at a newly allocated chunk of memory that is the type specified and the size needed for that type
      - NOTE: pointer will be NULL if there is no more memory to allocate
  - The cast may not be needed, but is good practice
  - calloc has the form:
    - pointer = (type *) calloc(n, sizeof(type)); // n is the size of the array
  - calloc is usually used for the creation of an array
  - Another C instruction is free, to free up the allocated memory when you no longer need it as in free(pointer);

# calloc example

```
#include <stdio.h>
#include <stdlib.h>                // needed for calloc

void main()
{
        int i;
        int *x, *y;                                 // two pointers to int arrays
        x = (int *) calloc(10, sizeof(int));        // x now points to an array of 10 ints
        for(i=0;i<10;i++) x[i] = i;                 // fill the array with values
        …                                           // oops, need more room than 10
        y = (int *) calloc(20, sizeof(int));        // create an array of 20, temporarily
                                                    //    pointed to by y
        for(i=0;i<10;i++) y[i] = x[i];              // copy old elements of x into y
        free(x);                                    // release memory of old array
        for(i=10;i<20;i++) y[i] = i;                // add the new elements
        x = y;                                      // reset x to point at new, bigger array
}
```

# Linked Structures

- Our last topic is in building linked structures
  - lists, trees, graphs
- These are dynamic structures, when you want to add a node, you allocate a new chunk of memory and attach it to the proper place in the structure via the pointers
  - In linked lists, the pointer is a next pointer to the next node in the list, in a tree, there are left and right children pointers
  - We will use malloc to allocated the node
  - We will need to traverse the structure to reach the proper place to insert a new node

# Declarations for Nodes

```
struct node {
   int data;
   struct node *next;
};

node *front=NULL;
```

front is a pointer to the first node in a linked list. It may initially be NULL. Traversing our linked list might use code like this:

```
temp = front;
while(temp!=NULL)
{
   // process temp
   temp=temp->next;
}
```

```
struct treeNode {
   int data;
   struct treeNode *left;
   struct treeNode *right;
};
```

Our root node will be declared as treeNode *root;

It is common in trees to have the root node point to the tree via the right pointer only with the left pointer remaining NULL

# Declarations for Nodes

```
struct node {
   int data;
   struct node *next;
};

node *front=NULL;
```

front is a pointer to the first node in a linked list. It may initially be NULL. Traversing our linked list might use code like this:

```
temp = front;
while(temp!=NULL)
{
    // process temp
    temp=temp->next;
}
```

```
struct treeNode {
   int data;
   struct treeNode *left;
   struct treeNode *right;
};
```

Our root node will be declared as treeNode *root;

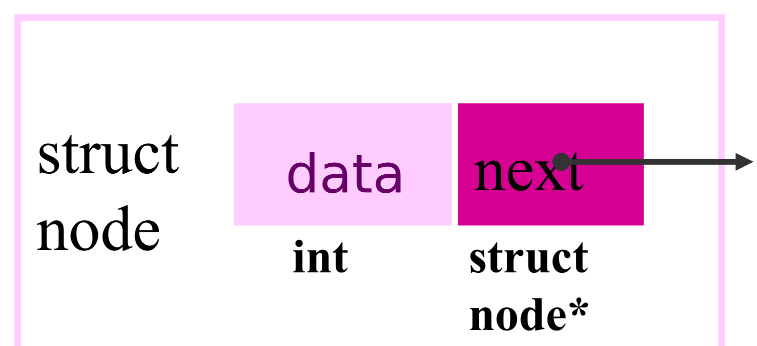
It is common in trees to have the root node point to the tree via the right pointer only with the left pointer remaining NULL

# Linked Lists

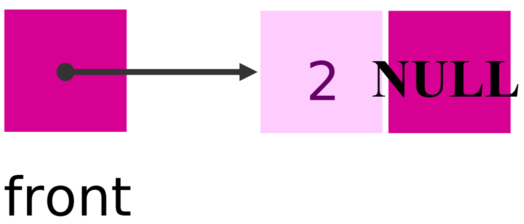

struct node* **front=NULL;**

- A *linked list* is a series of connected *nodes*
- Each node contains at least
  - A piece of data (any type)
  - Pointer to the next node in the list
- front: pointer to the first node
- The last node points to `NULL`

```
struct node {
   int data;
   struct node *next;
};
```



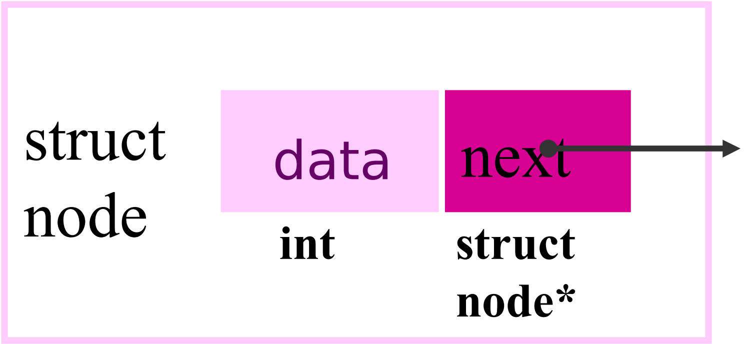
# Linked Lists



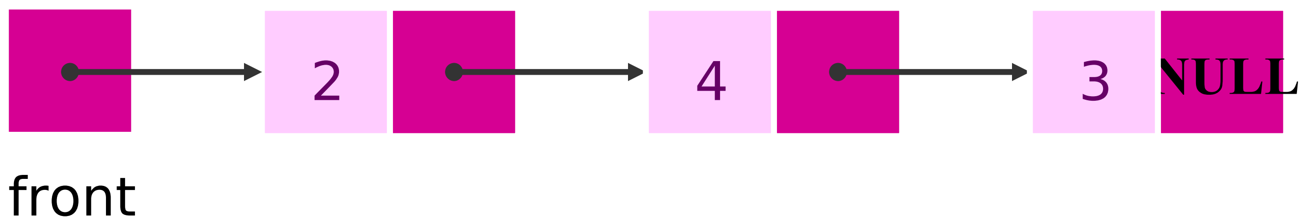

**front=(struct node)** malloc(sizeof(struct node));
front->data=2;
front->next=NULL;

- A *linked list* is a series of connected *nodes*
- Each node contains at least
  - A piece of data (any type)
  - Pointer to the next node in the list
- front: pointer to the first node
- The last node points to `NULL`

```
struct node {
  int data;
  struct node *next;
};
```

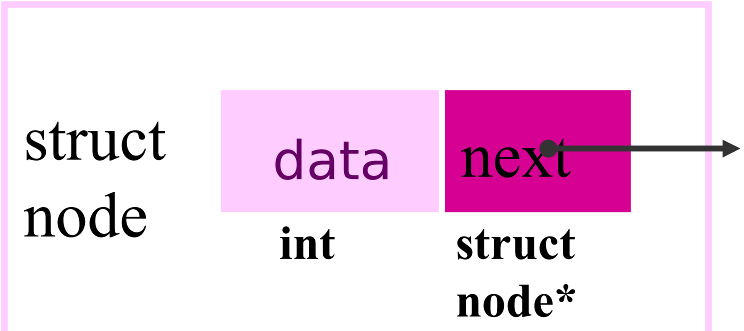


# Linked Lists



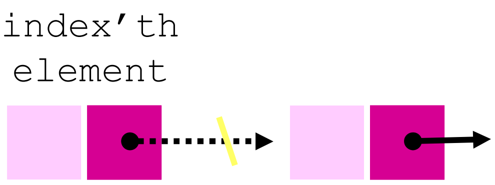

- A *linked list* is a series of connected *nodes*
- Each node contains at least
  - A piece of data (any type)
  - Pointer to the next node in the list
- front: pointer to the first node
- The last node points to `NULL`

```
struct node {
   int data;
   struct node *next;
};
```



# Inserting a new node

- `Node* InsertNode(int index, double x)`
- Steps
  1. Locate `index`'th element
  2. Allocate memory for the new node
  3. Point the new node to its successor
  4. Point the new node's predecessor to the new node



# Inserting a new node

- `Node* InsertNode(int index, double x)`
- Steps
  1. Locate `index`'th element
  2. Allocate memory for the new node
  3. Point the new node to its successor
  4. Point the new node's predecessor to the new node