COMP9024: Structs, Pointers to Structs, Self Referential Structures (Linked List)

Term: 20T0

[Credits: Lecture slides from COMP1511 (18s2)]

Decimal Representation

- Can interpret decimal number 4705 as:

$$4 \times 10^3 + 7 \times 10^2 + 0 \times 10^1 + 5 \times 10^0$$

- The *base* or *radix* is 10
  Digits 0 – 9
- Place values:

...	1000	100	10	1
...	10^3	10^2	10^1	10^0

- Write number as 4705_{10}
  - Note use of subscript to denote base

Binary Representation

- In a similar way, can interpret binary number 1011 as:

$$1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$$

- The *base* or *radix* is 2
  Digits 0 and 1
- Place values:

...	8	4	2	1
...	2^3	2^2	2^1	2^0

- Write number as 1011_2
  $(= 11_{10})$

Hexadecimal Representation

- Can interpret hexadecimal number 3AF1 as:

$$3 \times 16^3 + 10 \times 16^2 + 15 \times 16^1 + 1 \times 16^0$$

- The *base* or *radix* is 16
  Digits 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F
- Place values:

...	4096	256	16	1
...	16^3	16^2	16^1	16^0

- Write number as $3AF1_{16}$
  $(= 15089_{10})$

Binary to Hexadecimal

0	1	2	3	4	5	6	7
0000	0001	0010	0011	0100	0101	0110	0111
8	9	A	B	C	D	E	F
1000	1001	1010	1011	1100	1101	1110	1111

- *Idea:* Collect bits into groups of four starting from right to left
- "pad" out left-hand side with 0's if necessary
- Convert each group of four bits into its equivalent hexadecimal representation (given in table above)

Binary to Hexadecimal

- Example: Convert 1011111000101001_2 to Hex:

1011	1110	0010	1001_2
B	E	2	9_{16}

- Example: Convert 10111101011100_2 to Hex:

0010	1111	0101	1100
2	F	5	C_{16}

Hexadecimal to Binary

- Reverse the previous process
- Convert each hex digit into equivalent 4-bit binary representation
- Example: Convert $AD5_{16}$ to Binary:

A	D	5
1010	1101	0101_2

Memory Organisation

- During execution programs variables are stored in memory.
- Memory is effectively a gigantic array of bytes.
  COMP1521 will explain more
- Memory addresses are effectively an index to this array of bytes.
- These indexes can be very large
  up to $2^{32} - 1$ on a 32-bit platform
  up to $2^{64} - 1$ on a 64-bit platform
- Memory addresses usually printed in hexadecimal (base-16).

Memory Organisation

In order to fully understand how pointers are used to reference data in memory, here's a few basics on memory organisation.

0xFFFFFFFF	*High Memory*
0xFFFFFFFE	
⋮	
0x00000001	
0x00000000	*Low Memory*

Memory

- computer memory is a large array of *bytes*
- a variable will stored in 1 or more bytes
- on CSE machines a *char* occupies 1 byte, a an *int* 4 bytes, a *double* 8 bytes
- The & (address-of) operator returns a reference to a variable.
- Almost all C implementations implement pointer values using a variable's address in memory
- Hence for almost all C implementations & (address-of) operator returns a memory address.
- It is convenient to print memory addresses in Hexadecimal notation.

Variables in Memory

```
int k;
int m;

printf( "address of k is %p\n", &k );
// prints address of k is 0xbffffb80

printf( "address of m is %p\n", &m );
// prints address of k is 0xbffffb84
```

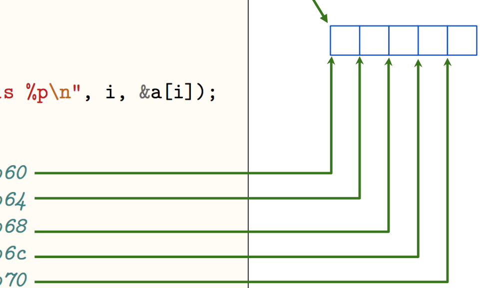
- `k` occupies the four bytes from `0xbffffb80` to `0xbffffb83`
- `m` occupies the four bytes from `0xbffffb84` to `0xbffffb87`

Arrays in Memory

Elements of the array will be stored in *consecutive* memory locations:

```
int a[5];

int i = 0;
while (i < 5) {
    printf("address of a[%d] is %p\n", i, &a[i]);
}
// prints:
// address of a[0] is 0xbffffb60
// address of a[1] is 0xbffffb64
// address of a[2] is 0xbffffb68
// address of a[3] is 0xbffffb6c
// address of a[4] is 0xbffffb70
```

a points to the start of the array (a[0])



Size of a Pointer

Just like any other variable of a certain type, a variable that is a pointer also occupies space in memory. The number of bytes depends on the computer's architecture.

- 32-bit platform: pointers likely to be 4 bytes
  e.g. CSE lab machines (about to change)
- 64-bit platform: pointers likely to be 8 bytes
  e.g. many student machines
- tiny embedded CPU: pointers could be 2 bytes
  e.g. your microwave

Pointers

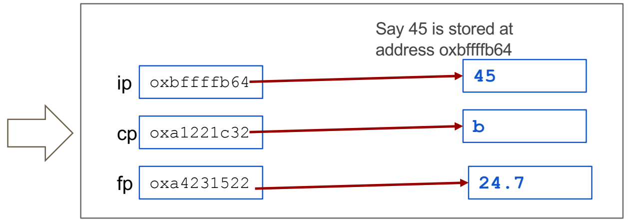
A pointer is a data type whose value is a reference to another variable.

```c
int *ip;    // pointer to int
char *cp;   // pointer to char
double *fp; // pointer to double
```

In most C implementations, pointers store the the memory address of the variable they refer to.

For example, *conceptually*

ip points to an ***int*** value,
cp points to a ***char*** and
fp points to a ***double*** value.

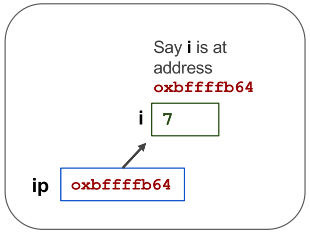


Pointers

- The & (address-of) operator returns a reference to a variable.
- The * (dereference) operator accesses the variable refered to by the pointer.

For example:

```c
int i = 7;
int *ip = &i;
printf("%d\n", *ip); // prints 7
*ip = *ip * 6;
printf("%d\n", i);   //prints 42
i = 24;
printf("%d\n", *ip); // prints 24
```



Pointers

- Like other variables, pointers need to be initialised before they are used .
- Like other variables, its best if novice programmers initialise pointers as soon as they are declared.
- The value NULL can be assigned to a pointer to indicate it does not refer to anything.
- NULL is a #define in stdio.h
- NULL and 0 interchangable (in modern C).
- Most programmers prefer NULL for readability.

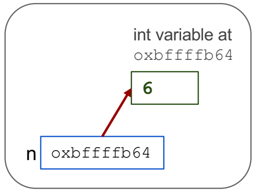
Pointer Arguments

We've seen that when primitive types are passed as arguments to functions, they are passed by value and any changes made to them are not reflected in the caller.

```
void increment(int n) {
    n = n + 1;
}
```

This attempt fails. But how does a function like `scanf` manage to update variables found in the caller? `scanf` takes pointers to those variables as arguments!

```
void increment(int *n) {
    *n = *n + 1;
}
```



Pointer Arguments

We use pointers to pass variables *by reference*! By passing the address rather than the value of a variable we can then change the value and have the change reflected in the caller.

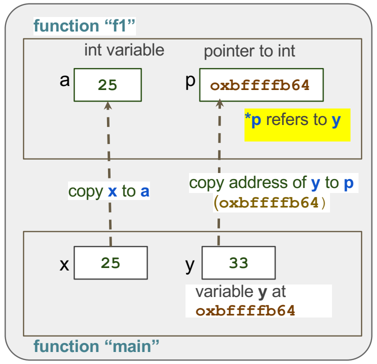
```
int i = 1;
increment(&i);
printf("%d\n", i);
//prints 2
```

In a sense, pointer arguments allow a function to 'return' more than one value. This greatly increases the versatility of functions. Take `scanf` for example, it is able to read multiple values and it uses its return value as error status.

Passing values by Reference

Simple example to illustrate

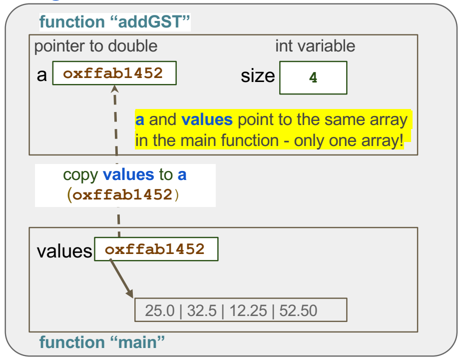
pass by value and reference




```
#include <stdio.h>

void f1(int a, int *p) {
    a = a + 5;
    *p = *p + 7;
}
int main(int argc, char *argv[]) {
    int x = 25;
    int y = 33;

    printf("Before calling f1: x=%d y=%d\n", x, y);

    f1(x, &y);

    printf("After calling f1:  x=%d y=%d\n", x, y);
    // x is unchanged, y changed

    return 0;
}
```


Array Reference

Simple example to illustrate how to **modify** an **array** passed as an **argument to a function**.




```
void addGST(double a[], int size) {
    int i = 0;
    while(i<size){
        a[i] = 1.1 * a[i];
        i++;
    }
}

void printArray(double a[], int size){

    int i = 0;
    while(i<size){
        printf("%10.2lf ", a[i]);
        i++;
    }
    printf("\n");
}

int main(int argc, char *argv[]) {
    double values[] = { 25.0, 32.5, 12.25, 52.50} ;

    printf("Before calling addGST: ");
    printArray(values, 4);

    addGST( values , 4 );

    printf("After calling addGST:  ");
    printArray(values, 4);

    return 0;
}
```

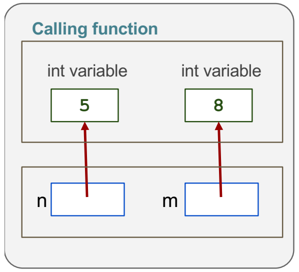
Pointer Arguments

Classic Example

Write a function that swaps the values of its two integer arguments.

Before we knew about pointer arguments this would have been impossible, but now it is straightforward.

```
void swap(int *n, int *m) {
    int tmp;

    tmp = *n;
    *n = *m;
    *m = tmp;
}
```



Pointer Return Value

You should not find it surprising that functions can return pointers.
However, you have to be extremely careful when returning pointers.
Returning a pointer to a local variable is illegal - that variable is destroyed when the function returns.
But you can return a pointer that was given as an argument:

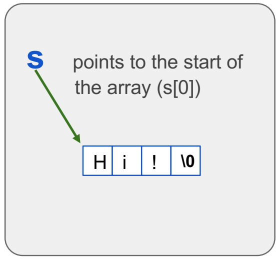
```
int *increment(int *n) {
    *n = *n + 1;
    return n;
}
```

Nested calling is now possible: `increment(increment(&i));`

Array Representation

A C array has a very simple underlying representation, it is stored in a contiguous (unbroken) memory block and a pointer is kept to the beginning of the block.

```
char s[] = "Hi!";
printf("s: %p *s: %c\n\n", s, *s);
printf("&s[0]: %p s[0]: %c\n", &s[0], s[0]);
printf("&s[1]: %p s[1]: %c\n", &s[1], s[1]);
printf("&s[2]: %p s[2]: %c\n", &s[2], s[2]);
printf("&s[3]: %p s[3]: %c\n", &s[3], s[3]);
// prints
// s: 0x7fff4b741060 *s: H
// &s[0]: 0x7fff4b741060 s[0]: H
// &s[1]: 0x7fff4b741061 s[1]: i
// &s[2]: 0x7fff4b741062 s[2]: !
// &s[3]: 0x7fff4b741063 s[3]:
```



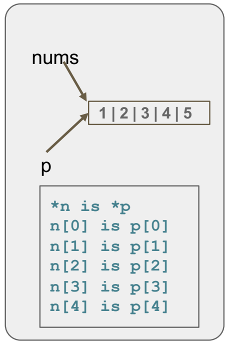
Array variables act as pointers to the beginning of the arrays!

Array Representation

Since array variables are pointers, it now should become clear why we pass arrays to `scanf` without the need for address-of (&) and why arrays are passed to functions by reference!
We can even use another pointer to act as the array name!

```
int nums[] = {1, 2, 3, 4, 5};
int *p = nums;

printf("%d\n", nums[2]);
printf("%d\n", p[2]);
// both print: 3
```

Since `nums` acts as a pointer we can directly assign its value to the pointer `p`!



Array Representation

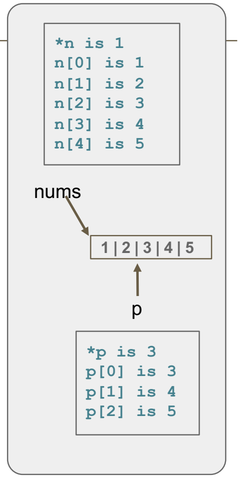
We can even make a pointer point to the middle of an array:

```
int nums[] = {1, 2, 3, 4, 5};
int *p = &nums[2];
printf("%d %d\n", *p, p[0]);
```

So is there a difference between an array variable and a pointer?

```
int i = 5;
p = &i; // this is OK
nums = &i; // this is an error
```

Unlike a regular pointer, an array variable is defined to point to the beginning of the array, it is constant and may not be modified.



Pointer Comparison

Pointers can be tested for equality or relative order.

```
double ff[] = {1.1, 1.2, 1.3, 1.4, 1.5, 1.6};
double *fp1 = ff;
double *fp2 = &ff[0];
double *fp3 = &ff[4];

printf("%d %d\n", (fp1 > fp3), (fp1 == fp2));
// prints: 0 1
```

Note that we are comparing the values of the pointers, i.e., memory addresses, not the values the pointers are pointing to!

Pointer Summary

Pointers:

- are a compound type
- usually implemented with memory addresses
- are manipulated using address-of(&) and dereference()
- should be initialised when declared
- can be initialised to NULL
- should not be dereferenced if invalid
- are used to pass arguments by reference
- are used to represent arrays
- should not be returned from functions if they point to local variables

typedef

We can use the keyword typedef to give a name to a type:

```
typedef double real;
```

This means variables can be declared as **numeric** but they will actually be of type **double**.
Do not overuse typedef - it can make programs harder to read, e.g.:

```
typedef int andrew;

andrew main(void) {
    andrew i,j;
    ....
```

Using typedef to make programs portable

Suppose have a program that does floating-point calculations.
If we use a typedef'ed name for all variable, e.g.:

```
typedef double real;

real matrix[1000][1000][1000];

real my_atanh(real x) {
   real u = (1.0 - x)/(1.0 + x);
   return -0.5 * log(u);
}
```

If we move to a platform with little RAM, we can save memory (and lose precision) by changing the typedef:

```
typedef float real;
```


structs

- We have seen simple types e.g. **int, char, double**
  - variables of these types hold single values
- We have seen a compound type: arrays
  - array variables hold multiple values
  - arrays are homogenous - every array element is the same type
  - array element selected using integer index
  - array size can be determined at runtime
- Another compound type: structs
  - structs hold multiple values (fields)
  - struct are heterogeneous - fields can be differenttype
  - struct field selected using name
  - struct fields fixed

structs - example

If we define a struct that holds COMP1511 student details:

```
#define MAX_NAME 64
#define N_LABS 10

struct student {
    int zid;
    char name[64];
    double lab_marks[N_LABS]
    double assignment1_mark;
    double assignment2_mark;
}
```

We can declare an arry to hold the details of all students:

```
struct student comp1511_students[900];
```


combining structs and typedef

Common to use typedef to give name to a struct type.

```
struct student {
    int zid;
    char name[64];
    double lab_marks[N_LABS]
    double assignment1_mark;
    double assignment2_mark;
}

typedef struct student student_t;

student_details_t comp1511_students[900];
```

Programmer often use convention to separate type names e.g. _t suffix.

Assigning structs

Unlike arrays, it is possible to copy all components of a structure in a single assignment:

```
struct student_details student1, student2;
...
student2 = student1;
```

It is *not* possible to compare all components with a single comparison:

```
if (student1 == student2) // NOT allowed!
```

If you want to compare two structures, you need to write a function to compare them component-by-component and decide whether they are "the same".

structs and functions

A structure can be passed as a parameter to a function:

```
void print_student(student_t student) {
   printf("%s z%d\n", d.name, d.zid);
}
```

Unlike arrays, a copy will be made of the entire structure, and only this copy will be passed to the function.
Unlike arrays, a function can return a struct:

```
student_t read_student_from_file(char filename[]) {
   ....
}
```


Pointers to structs

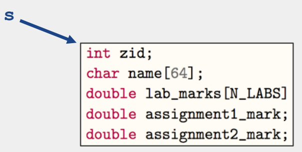
If a function needs to modify a structs field or if we want to avoid the inefficiency of copying the entire struct, we can instead pass a *pointer* to the struct as a parameter:

```
int scan_zid(student *s) {
return scanf("%d", &((*s).zid));
}
```

The "arrow" operator is more readable :

```
int scan_zid(student *s) {
return scanf("%d", &(s->zid));
}
```

If **s** is a pointer to a struct **s->field** is equivalent to **(*s).field**




```
(*s).zid = 1234567;
(*s).lab_marks[2]=7;
strcpy((*s).name, "John")
... ...
```

Alternatively,

```
s->zid = 1234567;
s->lab_marks[2]=7;
strcpy(s->name, "John")
```


Nested Structures

One structure can be nested inside another

```
typedef struct date     Date;
typedef struct time     Time;
typedef struct speeding Speeding;
```

```
struct date {
   int day, month, year;
};
struct time {
   int hour, minute;
};
struct speeding {
   Date   date;
   Time   time;
   double speed;
   char   plate[MAX_PLATE];
};
```

Dynamic memory allocation: malloc

- **malloc** allocates memory of a requested size (in bytes)
- Memory is allocated in "the heap", and it *lives **forever*** until we **free** it (or the program ends)
- *Important*: We MUST **free** memory allocated by **malloc**, should not rely on the operating system for cleanup.

```
malloc(number of bytes to allocate);
```

➔ returns a **pointer** to the block of allocated memory (i.e. the **address** of the memory, so we know how to find it!).

➔ **returns NULL** if insufficient memory available - you **must check** for this!

For example, let's assume we need a block of memory to hold 100,000 integers:

```
int *p = malloc( 100000 * sizeof(int) );
```


malloc : when it fails !

What happens if the allocation fails?

malloc returns NULL, and we need to check this:

```
int *p = malloc(1000 * sizeof(int));

if (p == NULL) {
    fprintf(stderr, "Error: couldn't allocate memory!\n");
    exit(1);
}
```


sizeof

- **sizeof** - C operator yields bytes needed for type or variable
- **sizeof (type)** or **sizeof variable**
- note unusual (badly designed) syntax - brackets indicate argument is a type
- use sizeof for every malloc call

```
printf("%ld", sizeof (char));    // 1
printf("%ld", sizeof (int));     // 4 commonly
printf("%ld", sizeof (double));  // 8 commonly
printf("%ld", sizeof (int[10])); // 40 commonly
printf("%ld", sizeof (int *));   // 4 or 8 commonly
printf("%ld", sizeof "hello");   // 6
```


free

- when we're done with the memory allocated by **malloc** function, we need to release that memory using **free** function.
- For example,

```
int *p = malloc(1000 * sizeof(int));

if (p == NULL) {
    fprintf(stderr, "Error: couldn't allocate memory!\n");
    exit(1);
}

// do some thing here with the memory allocation
//

// free up the memory that was used
free(p);
```

free

- `free()` indicates you've finished using the block of memory
- Continuing to use memory after `free()` results in very nasty bugs.
- `free()` memory block twice also cause bad bugs.
- if program keeps calling `malloc()` without corresponding `free()` calls program's memory will grow steadily larger called a **memory leak**.
- Memory leaks major issue for long running programs.
- Operating system recovers memory when program exists.

Scope and Lifetime

- the variables inside a function only exist as long as the function does
- once your function returns, the variables inside are "gone"

What if we need something to "stick around" for longer?

Two options:

- make it in a "parent" function
- dynamically allocate memory

Lifetimes

Make it in a "parent" function,
for example:

```
void changeA(int *b, int size){

    b[2] = 55;

}

void main(void) {

    int a[10] = {0};

    changeA( a , 10);

    printf("%d", a[2] ); // prints 55

}
```

Allocate in a "parent" function

pass a pointer

Lifetimes

Dynamically allocate memory in a function and return a pointer,
For example:

```
int *getA(void){

    int *b = malloc(10 * sizeof(int));
    b[2] = 55;

    return b;
}

void main(void) {

    int *a = getA();

    printf("%d", a[2] ); // prints 55

    free(a);

}
```

Dynamically allocate in a function

return a pointer

free

Self-Referential Structures

We can define a structure containing
a pointer to the same type of structure:

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

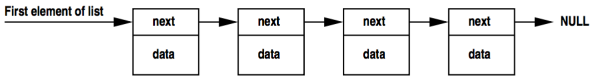
These "self-referential" pointers can be used to build larger "dynamic" data structures out of smaller building blocks.

Linked List

The most fundamental of these dynamic data structures is the *Linked List*:

- based on the idea of a sequence of data items or nodes
- linked lists are more flexible than arrays:
  - items don't have to be located next to each other in memory
  - items can easily be rearranged by altering pointers
  - the number of items can change dynamically
  - items can be added or removed in any order

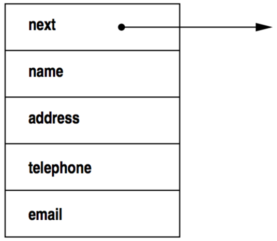
Linked List



- a *linked list* is a sequence of items
- each item contains data and a pointer to the next item
- need to separately store a pointer to the first item or "head" of the list
- the last item in the list is special
  it contains `NULL` in its `next` field instead of a pointer to an item

Example of List Item

Example of a list item used to store an address:



Example of List Item in C

```
struct address_node {
    struct address_node *next;
    char *telephone;
    char *email;
    char *address;
    char *telephone;
    char *email;
};
```


List Items

List items may hold large amount of data or many fields.
For simplicity, we'll assume each list item need store only a single int.

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


List Operations

Basic list operations:

- create a new item with specified data
- search for a item with particular data
- insert a new item to the list
- remove a item from the list

Many other operations are possible.

Creating a List Item

```
// Create a new struct node containing the specified dat
// and next fields, return a pointer to the new struct n

struct node *create_node(int data, struct node *next) {
    struct node *n;
    n = malloc(sizeof (struct node));
    if (n == NULL) {
        fprintf(stderr, "out of memory\n");
        exit(1);
    }
    n->data = data;
    n->next = next;
    return n;
}
```

Building a list

Building a list containing the 4 ints: 13, 17, 42, 5

```
struct node *head = create_node(5, NULL);
head = create_node(42, head);
head = create_node(17, head);
head = create_node(13, head);
```


Recap: Self-Referential Structures

We can define a structure containing
a pointer to the same type of structure:

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

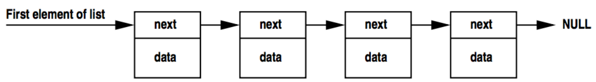
These "self-referential" pointers can be used to build larger "dynamic" data structures out of smaller building blocks.

Recap: Linked List

The most fundamental of these dynamic data structures is the *Linked List*:

- based on the idea of a sequence of data items or nodes
- linked lists are more flexible than arrays:
  - items don't have to be located next to each other in memory
  - items can easily be rearranged by altering pointers
  - the number of items can change dynamically
  - items can be added or removed in any order

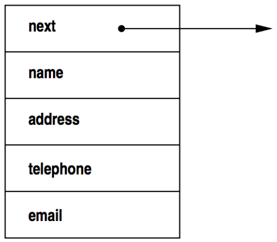
Recap: Linked List



- a *linked list* is a sequence of items
- each item contains data and a pointer to the next item
- need to separately store a pointer to the first item or "head" of the list
- the last item in the list is special
  it contains `NULL` in its `next` field instead of a pointer to an item

Recap: Example of List Item

Example of a list item used to store an address:



Recap: Example of List Item in C

```
struct address_node {
    struct address_node *next;
    char *telephone;
    char *email;
    char *address;
    char *telephone;
    char *email;
};
```


Recap: List Items

List items may hold large amount of data or many fields.
For simplicity, we'll assume each list item need store only a single int.

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


Recap: List Operations

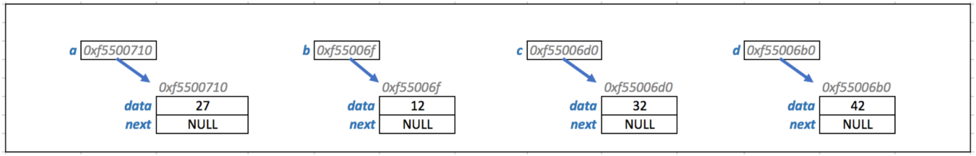
Basic list operations:

- create a new item with specified data
- search for a item with particular data
- insert a new item to the list
- remove a item from the list

Many other operations are possible.

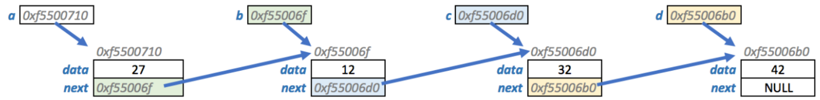
Creating a Node (List Item)

```
struct node *a = malloc(sizeof (struct node));
a->data = 27;
a->next = NULL;

struct node *b = malloc(sizeof (struct node));
b->data = 12;
b->next = NULL;

struct node *c = malloc(sizeof (struct node));
c->data = 32;
c->next = NULL;

struct node *d = malloc(sizeof (struct node));
d->data = 42;
d->next = NULL;
```

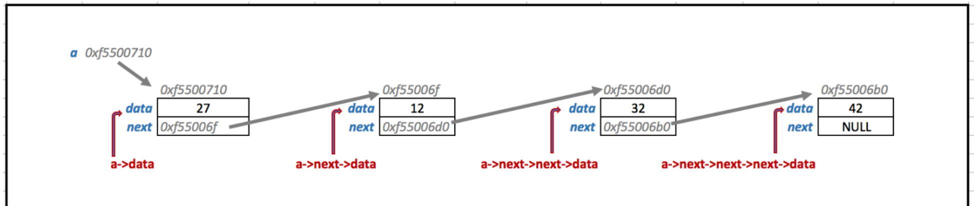


Link Nodes

```
a->next = b;
b->next = c;
c->next = d;
d->next= NULL;
```

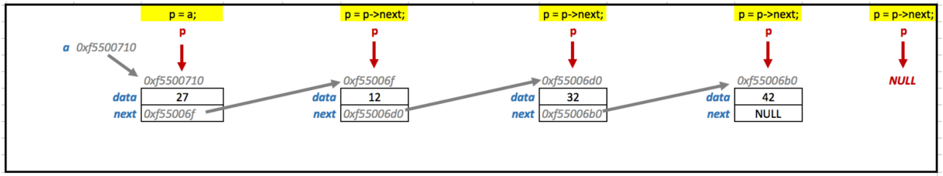


Link list




```
printf("---------------- ----------------\n\n");
printf("// After linking nodes: \n");
printf("//    a->next is same as b\n");
printf("//    a->next->next is same as c\n");
printf("//    a->next->next->next is same as d\n");
printf("   Node a->data: %d \n", a->data);
printf("   Node a->next->data: %d \n", a->next->data);
printf("   Node a->next->next->data: %d \n", a->next->next->data);
printf("   Node a->next->next->next->data: %d \n", a->next->next->next->data);
```


Link List - Traversal

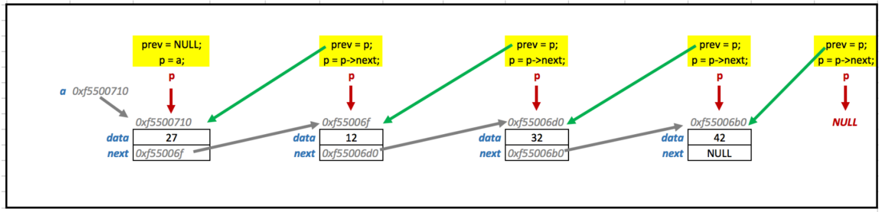



```
printf("---------------- ----------------\n\n");
printf("//  Below: process_list(a) ....... \n\n");
process_list(a);
```

```
void process_list(struct node *head) {
    struct node *p = head;

    while (p != NULL) {

        printf("   p->data=%d \n", p->data );

        p = p->next;
    }
}
```

Process node data here

Linked List: Previous pattern




```c
void prev_example(struct node *head) {
    struct node *prev = NULL;
    struct node *p = head;

    while (p != NULL) {

        printf("    p->data=%d \n", p->data );

        prev = p;
        p = p->next;
    }
}
```

Later we will see examples of this **previous pattern**, for example in **deleting** a node.

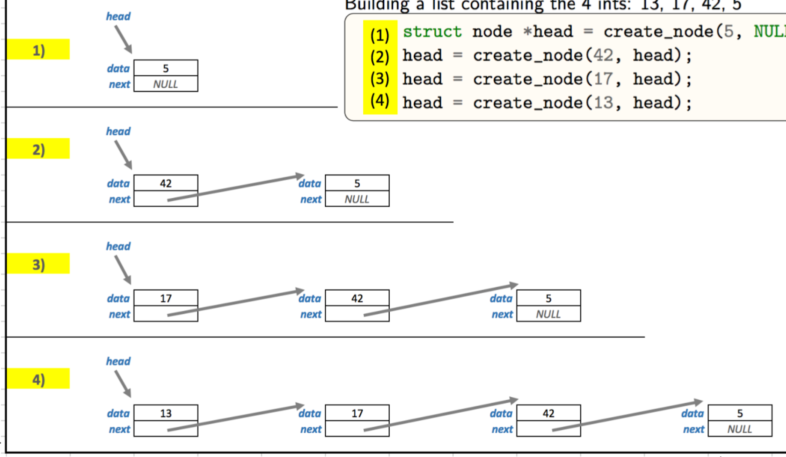
Creating a List Item/Node

```c
// Create a new struct node containing the specified dat
// and next fields, return a pointer to the new struct n

struct node *create_node(int data, struct node *next) {
   struct node *n;
   n = malloc(sizeof (struct node));
   if (n == NULL) {
      fprintf(stderr, "out of memory\n");
      exit(1);
   }
   n->data = data;
   n->next = next;
   return n;
}
```


Building a list

Building a list containing the 4 ints: 13, 17, 42, 5

```c
(1) struct node *head = create_node(5, NULL);
(2) head = create_node(42, head);
(3) head = create_node(17, head);
(4) head = create_node(13, head);
```



Summing a List

```c
// return sum of list data fields
int sum(struct node *head) {
   int sum = 0;
   struct node *n = head;
   // execute until end of list
   while (n != NULL) {
      sum += n->data;
      // make n point to next item
      n = n->next;
   }
   return sum;
}
```

Summing a List: For Loop

```
// return sum of list data fields: using for loop

int sum1(struct node *head) {
    int sum = 0;

    for (struct node *n = head; n != NULL; n = n->next) {

        sum += n->data;

    }

    return sum;
}
```


Finding an Item in a List

```
// return pointer to first node with specified data value
// return NULL if no such node

struct node *find_node(struct node *head, int data) {
    struct node *n = head;

    // search until end of list reached
    while (n != NULL) {
        // if matching item found return it
        if (n->data == data) {
            return n;
        }

        // make node point to next item
        n = n->next;
    }

    // item not in list
    return NULL;
}
```


Finding an Item in a List: For Loop

- Same function but using a for loop instead of a while loop.
- Compiler will produce same machine code as previous function.

```
// previous function written as for loop

struct node *find_node1(struct node *head, int data) {
    for (struct node *n = head; n != NULL; n = n->next) {
        if (n->data == data) {
            return n;
        }
    }
    return NULL;
}
```


Finding an Item in a List: Shorter While Loop

- Same function but using a more concise while loop.
- Shorter does not always mean more readable.
- Compiler will produce same machine code as previous functions.

```
struct node *find_node2(struct node *head, int data) {
    struct node *n = head;

    while (n != NULL && n->data != data) {

        n = n->next;
    }

    return n;
}
```

Printing a List - Python Syntax

For example,
[45, 67, 2, 43]

```c
// print contents of list in Python syntax

void print_list(struct node *head) {
    printf("[");
    for (struct node *n = head; n != NULL; n = n->next) {
        printf("%d", n->data);
        if (n->next != NULL) {
            printf(", ");
        }
    }
    printf("]");
}
```

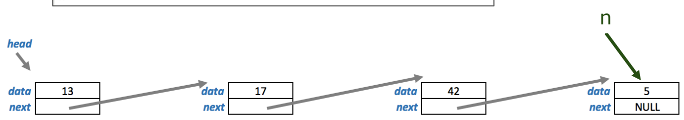
We print "," if there is a next node, otherwise skip printing ","

Finding Last Item in List

```c
// return pointer to last node in list
// NULL is returned if list is empty

struct node *last(struct node *head) {
    if (head == NULL) {
        return NULL;
    }

    struct node *n = head;
    while (n->next != NULL) {
        n = n->next;
    }
    return n;
}
```

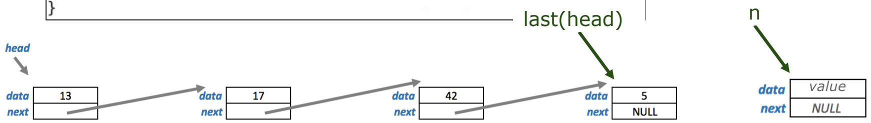
See the difference:
We are checking,
n->next != NULL
(in place of n != NULL)

The loop stops here because,
n->next == NULL



Appending to List

```c
// create a new list node containing value
// and append it to end of list

struct node *append(struct node *head, int value) {
    // new node will be last in list, so next field is NULL
    struct node *n = create_node(value, NULL);
    if (head == NULL) {
        // new node is now  head of the list
        return n;
    } else {
        // change next field of last list node
        // from NULL to new node
        last(head)->next = n;  /* append node to list */
        return head;
    }
}
```



Deleting all items from a List

```c
// Delete all the items from a linked list.

void delete_all(struct node *head) {
    struct node *n = head;
    struct node *tmp;

    while (n != NULL) {
        tmp = n;
        n = n->next;
        free(tmp);
    }
}
```

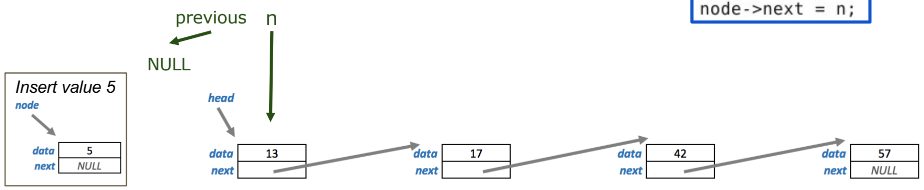
We **cannot** do the following in the body of while loop!

free(n);
n = n->next;

Insert a Node into an Ordered List

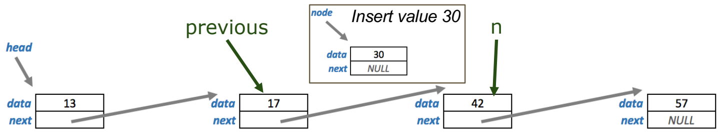
- Case-1: Insert a node at the **beginning**

```
head = node;
node->next = n;
```



- Case-2: Insert a node in the **middle** or at the **end**

```
previous->next = node;
node->next = n;
```



Insert a Node into an Ordered List

```
//Insert a Node into an Ordered List
//
struct node *insert_ordered(struct node *head, struct node *node) {
    struct node *previous;
    struct node *n = head;
    // find correct position
    while (n != NULL && node->data > n->data) {
        previous = n;
        n = n->next;
    }

    // link new node into list
    if (previous == NULL) {
        head = node;
        node->next = n;
    } else {
        previous->next = node;
        node->next = n;
    }

    return head;
}
```

Find correct position

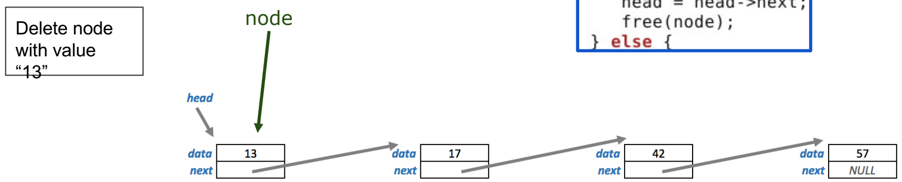
Case-1

Case-2

Delete a Node from a List

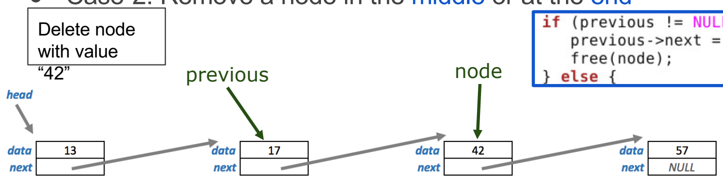
- Case-1: Remove **first** node

```
if (node == head) {
    head = head->next;
    free(node);
} else {
```



- Case-2: Remove a node in the **middle** or at the **end**

```
if (previous != NULL) { // node found in list
    previous->next = node->next;
    free(node);
} else {
```



Delete a Node from a List

```
// Delete a Node from a List
//
struct node *delete(struct node *head, struct node *node) {
    if (node == head) {
        head = head->next;          // remove first item
        free(node);
    } else {
        struct node *previous = head;
        while (previous != NULL && previous->next != node) {
            previous = previous->next;
        }
        if (previous != NULL) { // node found in list
            previous->next = node->next;
            free(node);
        } else {
            fprintf(stderr, "warning: node not in list\n");
        }
    }
    return head;
}
```

Case-1

Find correct position

Case-2

Recursion

- **Recursion** is a programming pattern where a function calls ***itself***

- For example, we define *factorial* as below,

  n! = 1*2*3* … *(n-1)*n

- We can ***recursively*** define *factorial* function as below,

  f(n) = 1 , if (n=0)

  f(n) = n * f(n-1) , for others

Pattern for a Recursive function

- Base case(s)
  - Situations when we **do not** call the same function (no recursive call), because the problem can be solved easily without a recursion.
  - All recursive calls eventually lead to one of the base cases.
- Recursive Case
  - We **call** the **same function** for a problem with ***smaller size***.
  - Decrease in a problem size eventually leads to one of the base cases.

```
// return sum of list data fields: using recursive call

int sum(struct node *head) {
    if (head == NULL) {
        return 0;
    }
    return head->data + sum(head->next)
}
```

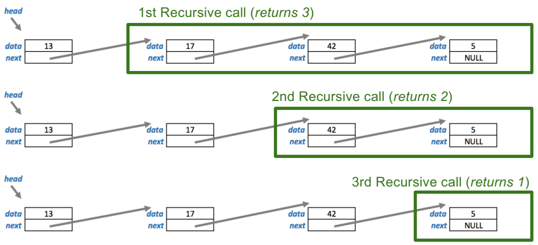
Base case

Recursive case,
Recursive call for a
smaller problem
(size-1)

Linked List with Recursion

```
// return count of nodes in list

int length(struct node *head) {
    if (head == NULL) {
        return 0;
    }
    return 1 + length(head->next);
}
```

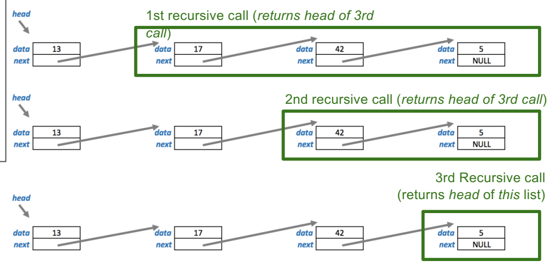
Recursive call



Base case
Head == NULL
returns 0

Last Node using Recursion

```
struct node *last(struct node *head) {
    // list is empty
    if(head == NULL) {
        return NULL;
    }
    // found the last node! return it.
    else if (head->next == NULL) {
        return head;
    }
    // return last node from the rest of the list
    // using a recursion
    else {
        return last(head->next);
    }
}
```



Base case
head->next == NULL
returns head of 3rd call

Find Node using Recursion

```c
// return pointer to first node with specified data value
// return NULL if no such node

struct node *find_node(struct node *head, int data) {
    // empty list, so return NULL
    if (head == NULL) {
        return NULL;
    }
    // Data at "head" is same as the "data" we are searching,
    // Found the node! so return head.
    else if (head->data == data) {
        return head;
    }
    // Find "data" in the rest of the list, using recursion,
    // return whatever answer we get from the recursion
    else {
        return find_node(head->next, data);
    }

}
```

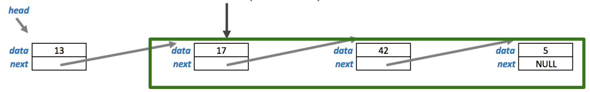
Recursive call

Delete From List using Recursion

```c
// Delete a Node from a List: Recursive
struct node *deleteR(struct node *head, struct node *node) {
   if (head == NULL) {
      fprintf(stderr, "warning: node not in list\n");
   }
   // Found the node!, remove this (first) node
   else if (node == head) {
      head = head->next;
      free(node);
   }
   // Delete node from the rest of the list, using recursion.
   // Assign "updated" rest of the list to head->next.
   else {
      head->next = deleteR(head->next, node)
   }
   return head;
}
```

Recursive call

node (to delete)



1st recursive call (node to delete is same as "head" of this call, *returns updated list, pointing to node with 42)*

Linked List with Recursion

```c
// Insert a Node into an Ordered List: recursive
struct node *insertR(struct node *head, struct node *node) {
   if (head == NULL || head->data >= node->data) {

      node->next = head;
      return node;
   }

   head->next = insertR(head->next, node);

   return head;
}
```

Recursive call

Print Python List using Recursion

```c
// print contents of list in Python syntax

void print_list(struct node *head) {
    printf("[");
    if (head != NULL) {
        print_list_items(head);
    }
    printf("]");
}

void print_list_items(struct node *head) {
    printf("%d", head->data);
    if (head->next != NULL) {
        printf(", ");
        print_list_items(head->next);
    }
}
```

Recursive function

Recursive call