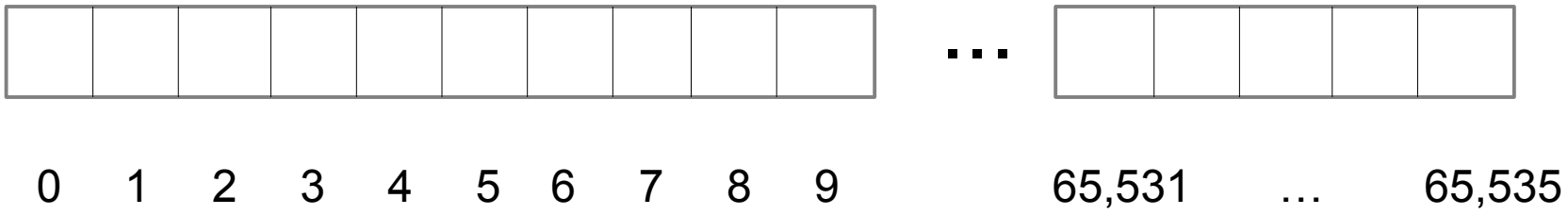
# CSC172
# Data Structures

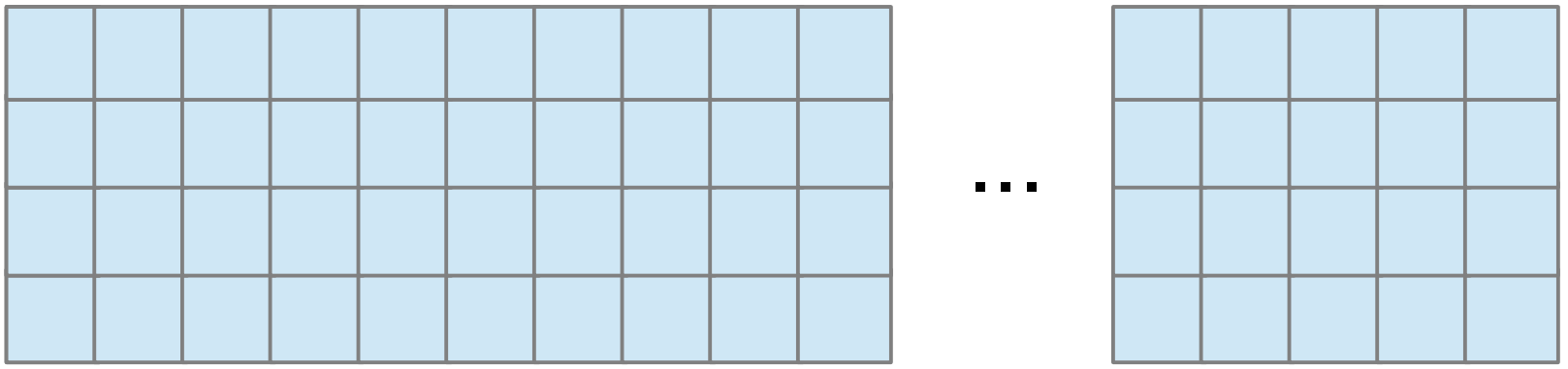
# Linked Lists
(C version)

# Virtual Memory Map (e.g., Unix)



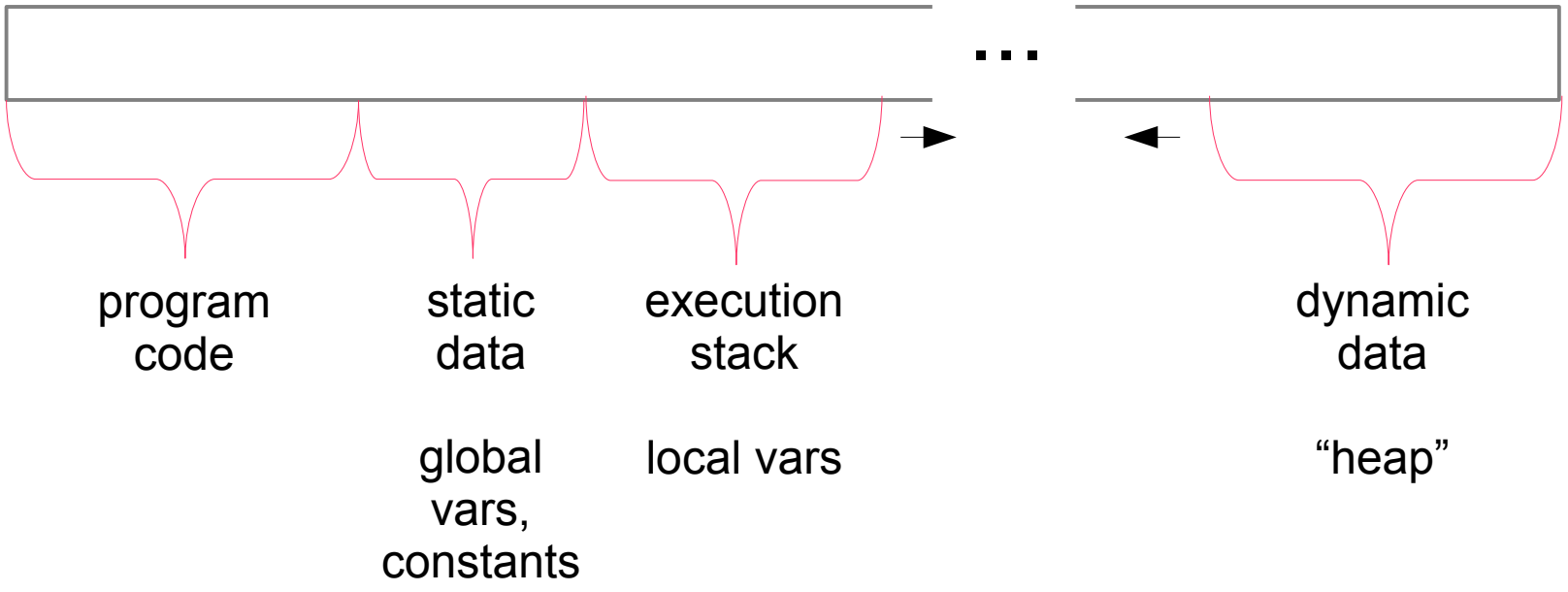

A contiguous sequence of bytes,
each one with a unique address

# Virtual Memory Map ii



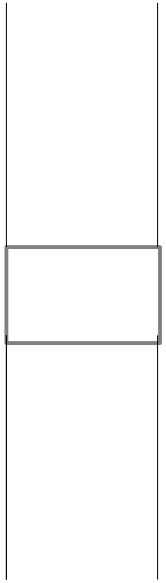

A contiguous sequence of words of k bytes each
Each word is large enough to hold an integer or address up to $2^{8k} - 1$

# Virtual Memory Map iii



# Static data

```
int K;

void main() {

}
```

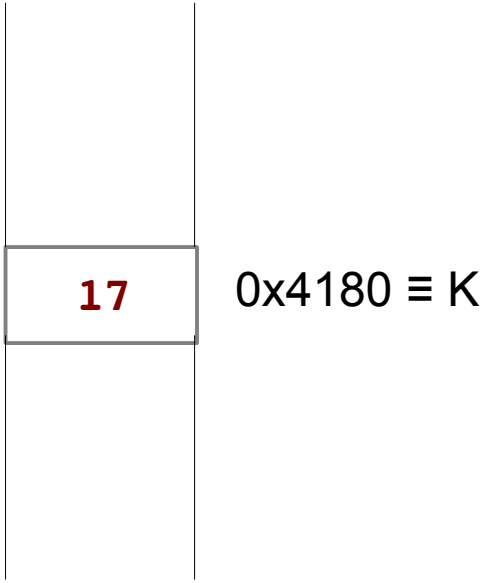


16,768 ≡ 0x4180 ≡ K

Static data is located in fixed positions during the entire run of the program, but positions may differ from one run to the next.

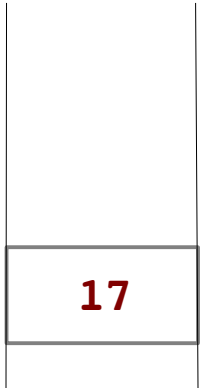
# Static data

```
int K;

void main() {
    K = 17;
}
```

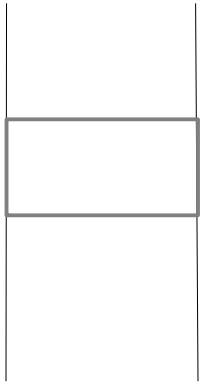


# Static data

```
int K;
…
int *P;

void main() {
    K = 17;
}
```



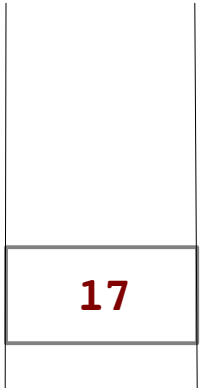

0x4180 ≡ K



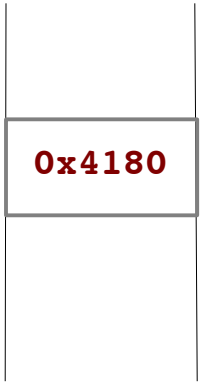
0x6820 ≡ P

# Static data

```
int K;
…
int *P;

void main() {
    K = 17;
    P = &K;
}
```




0x4180 ≡ K



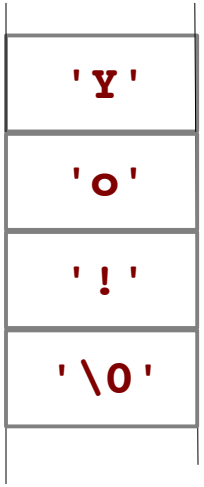

0x6820 ≡ P

A variable whose contents is the address of some memory location is called

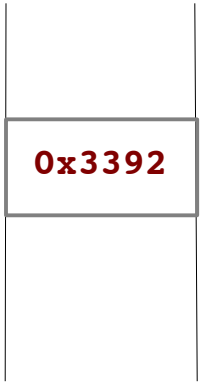
A POINTER

# Static data

```
char myCharData[] =
    { 'Y', 'o', '!', '\0' };
…
char *myString;

void main() {
    myString = myCharData;
    printf("%s\n", myString);
}
```

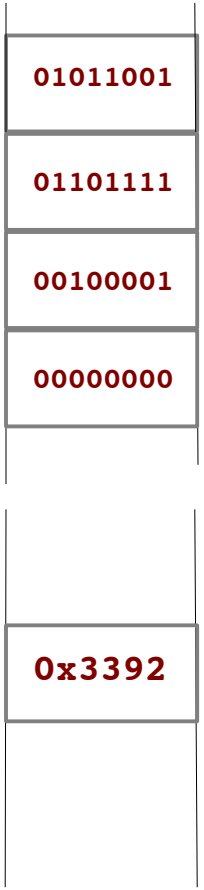



0x3392 ≡ myCharData




0x5620 ≡ myString

# Static data



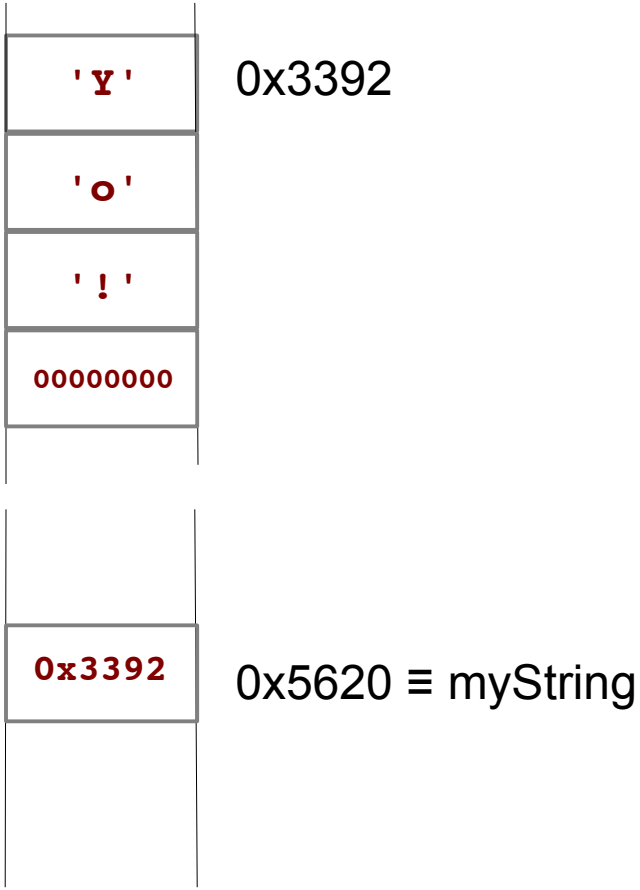

0x3392 ≡ myCharData

0x5620 ≡ myString

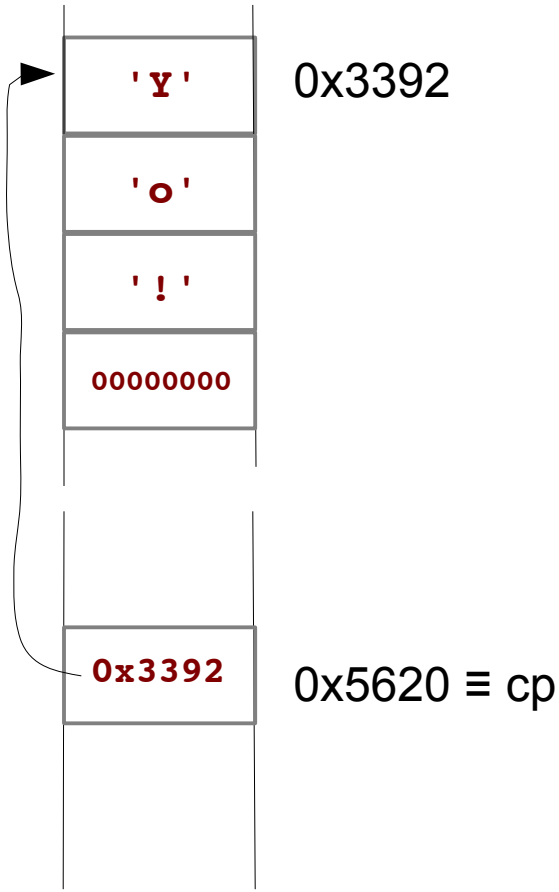
```
char myCharData[] =
    { 'Y', 'o', '!', '\0' };
…
char *myString;

void main() {
    myString = myCharData;
//  myString = &myCharData[0];
    printf("%s\n", myString);
}
```

# Static data

```
char *myString = “Yo!”;

void main() {
    printf(“%s\n”, myString);
}
```



# Static data




```
char *cp;

void main() {
    cp = "Yo!";
    while (*cp)
        printf("%c", *cp++);
    printf("\n");
}
```

*cp

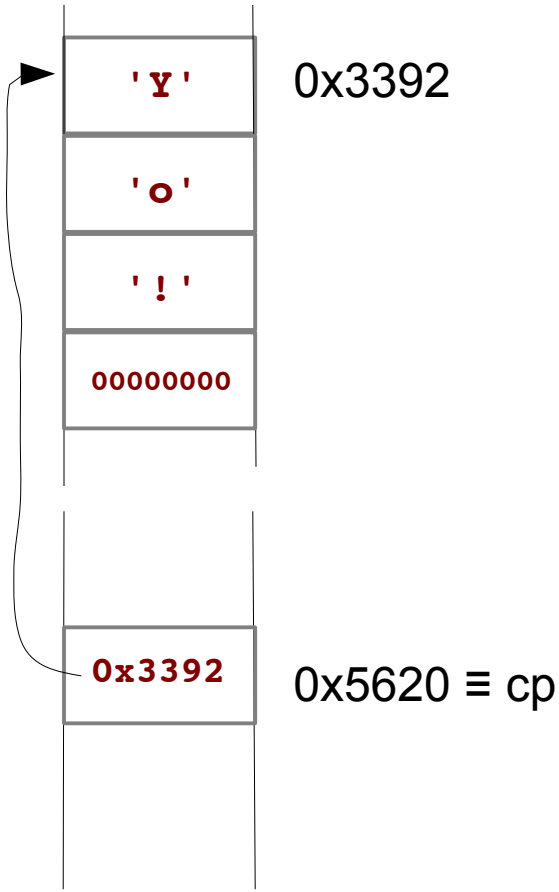
means

“fetch the value that cp points to”

*cp ≡ cp[0]

&cp[0] ≡ &*cp ≡ cp

# Static data




```
char *cp;

void main() {
    cp = "Yo!";
    while (*cp)
        printf("%c", *cp++);
    printf("\n");
}
```

*cp++

means

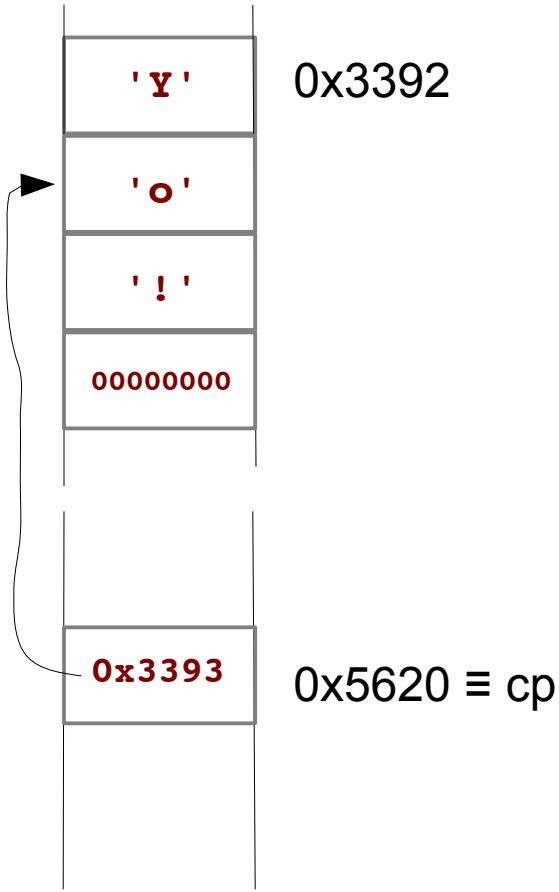
“fetch the value that cp points to, and then, as a side effect, change cp to point to the next location”

→ POINTER ARITHMETIC

*(cp+k) ≡ cp[k] , NOT ≡ *cp+k !!!

++*cp ≡ ? *++cp ≡ ?

# Static data




```
char *cp;

void main() {
    cp = "Yo!";
    while (*cp)
        printf("%c", *cp++);
    printf("\n");
}
```

*cp++

means

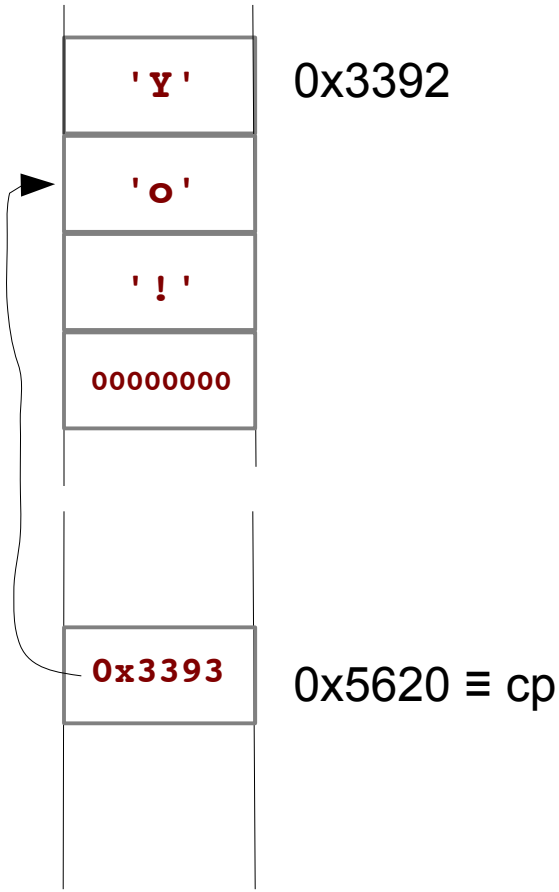
“fetch the value that cp points to, and then, as a side effect, change cp to point to the next location”

→ POINTER ARITHMETIC

*(cp+k) ≡ cp[k] , NOT ≡ *cp+k !!!

++*cp ≡ ? *++cp ≡ ?

# Static data




```
char *cp;

void main() {
    cp = "Yo!";
    while (*cp)
        printf("%c", *cp++);
    printf("\n");
}
```

*cp++

means

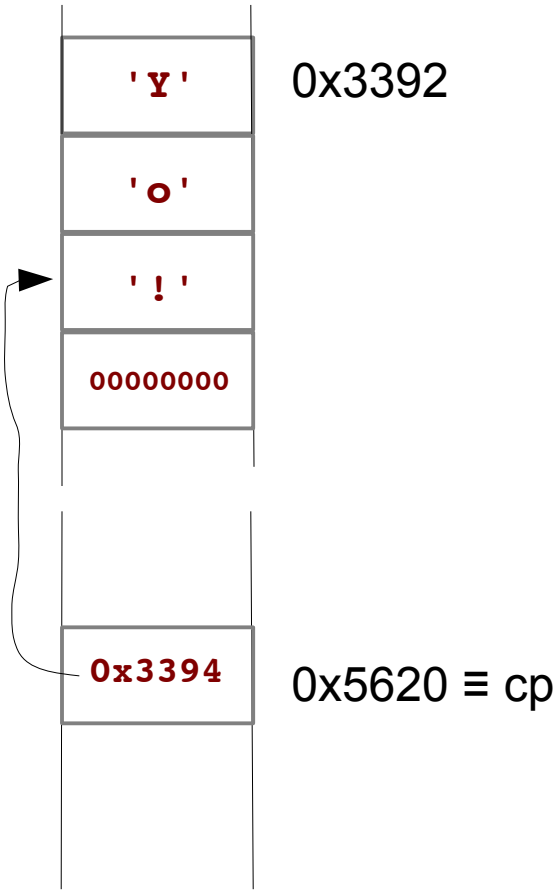
“fetch the value that cp points to, and then, as a side effect, change cp to point to the next location”

→ POINTER ARITHMETIC

*(cp+k) ≡ cp[k] , NOT ≡ *cp+k !!!

++*cp ≡ ? *++cp ≡ ?

# Static data




```
char *cp;

void main() {
    cp = "Yo!";
    while (*cp)
        printf("%c", *cp++);
    printf("\n");
}
```

*cp++

means

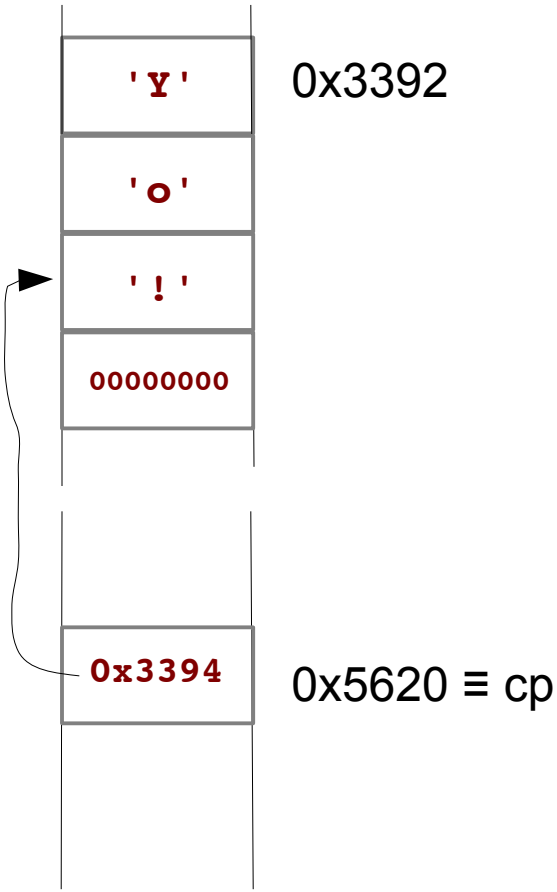
“fetch the value that cp points to, and then, as a side effect, change cp to point to the next location”

→ POINTER ARITHMETIC

*(cp+k) ≡ cp[k] , NOT ≡ *cp+k !!!

++*cp ≡ ? *++cp ≡ ?

# Static data




```
char *cp;

void main() {
    cp = "Yo!";
    while (*cp)
        printf("%c", *cp++);
    printf("\n");
}
```

*cp++

means

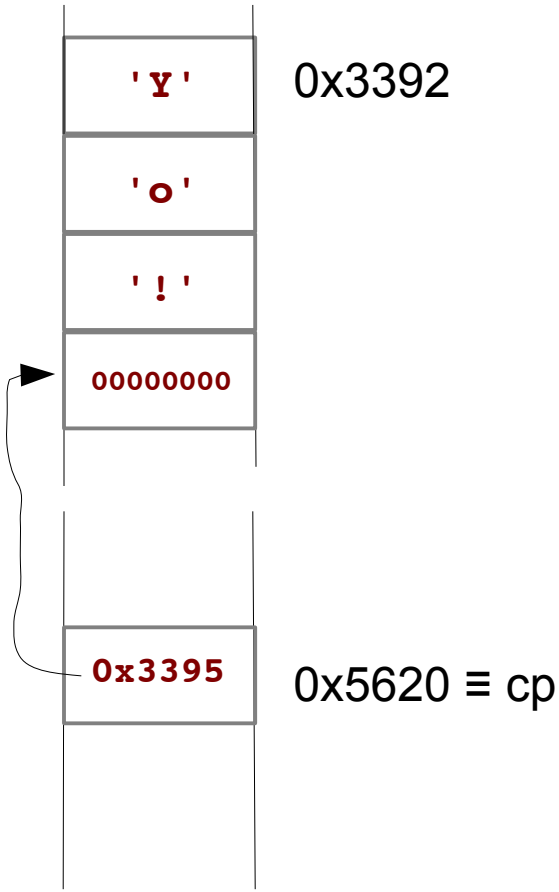
“fetch the value that cp points to, and then, as a side effect, change cp to point to the next location”

→ POINTER ARITHMETIC

*(cp+k) ≡ cp[k] , NOT ≡ *cp+k !!!

++*cp ≡ ? *++cp ≡ ?

# Static data




```
char *cp;

void main() {
    cp = "Yo!";
    while (*cp)
        printf("%c", *cp++);
    printf("\n");
}
```

*cp++

means

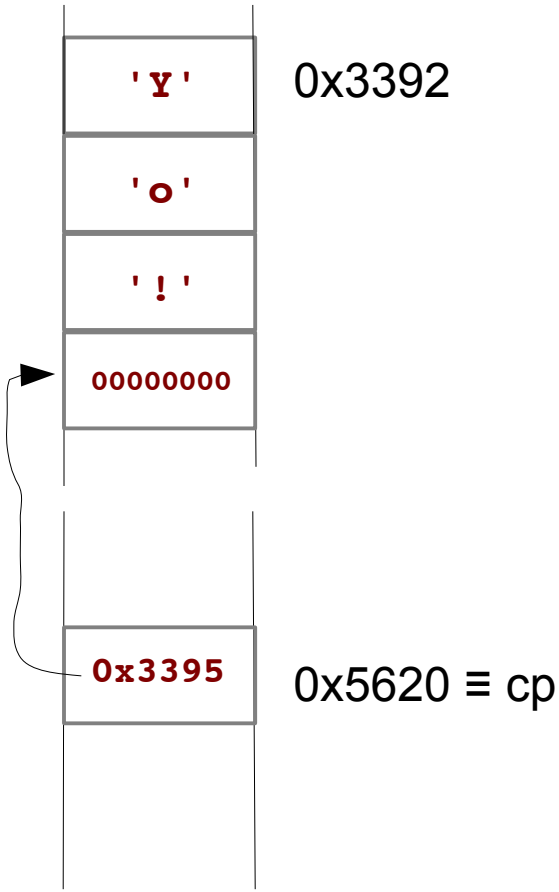
“fetch the value that cp points to, and then, as a side effect, change cp to point to the next location”

→ POINTER ARITHMETIC

*(cp+k) ≡ cp[k] , NOT ≡ *cp+k !!!

++*cp ≡ ? *++cp ≡ ?

# Static data




```
char *cp;

void main() {
    cp = "Yo!";
    while (*cp)
        printf("%c", *cp++);
    printf("\n");
}
```

*cp++

means

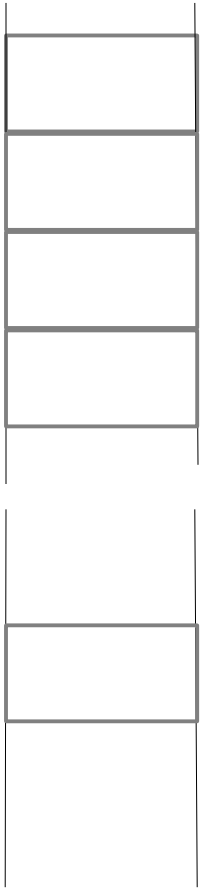
“fetch the value that cp points to, and then, as a side effect, change cp to point to the next location”

→ POINTER ARITHMETIC

*(cp+k) ≡ cp[k] , NOT ≡ *cp+k !!!

++*cp ≡ ? *++cp ≡ ?

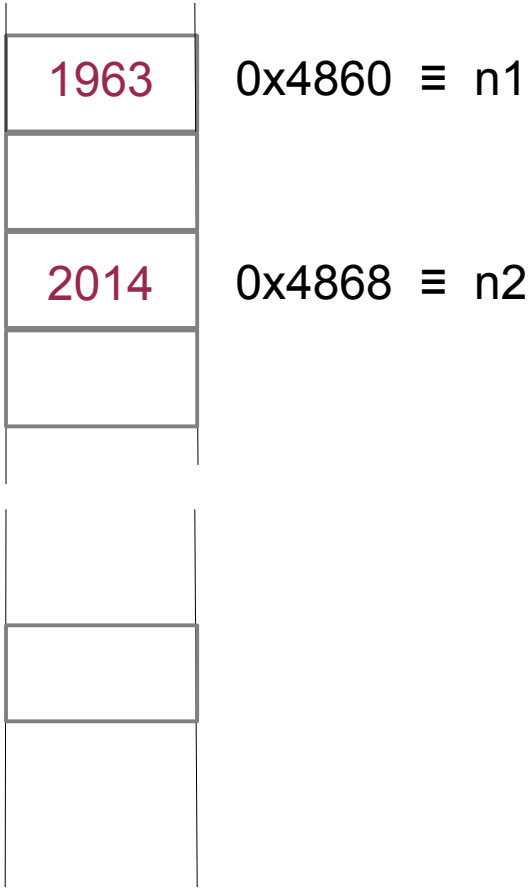
# Static data



0x4860

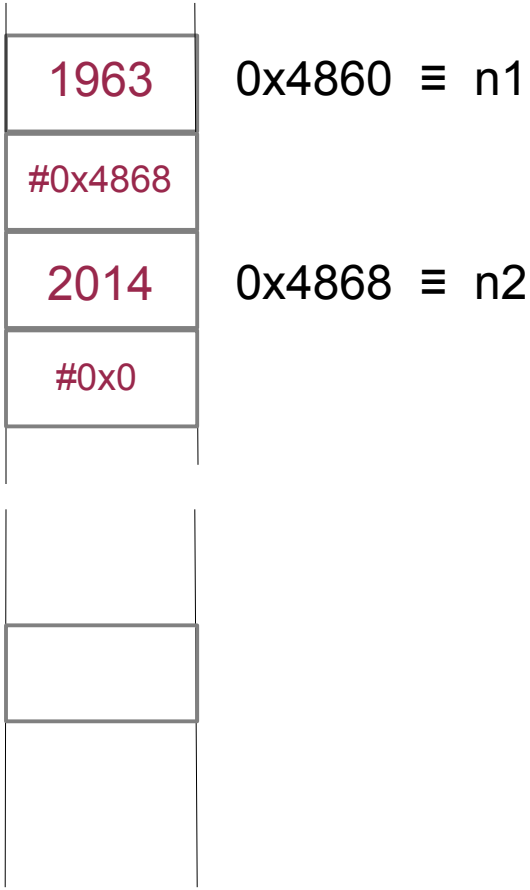
```
struct ListNode {
    int data;
    struct ListNode *next;
}

void main() {
}
```

# Static data



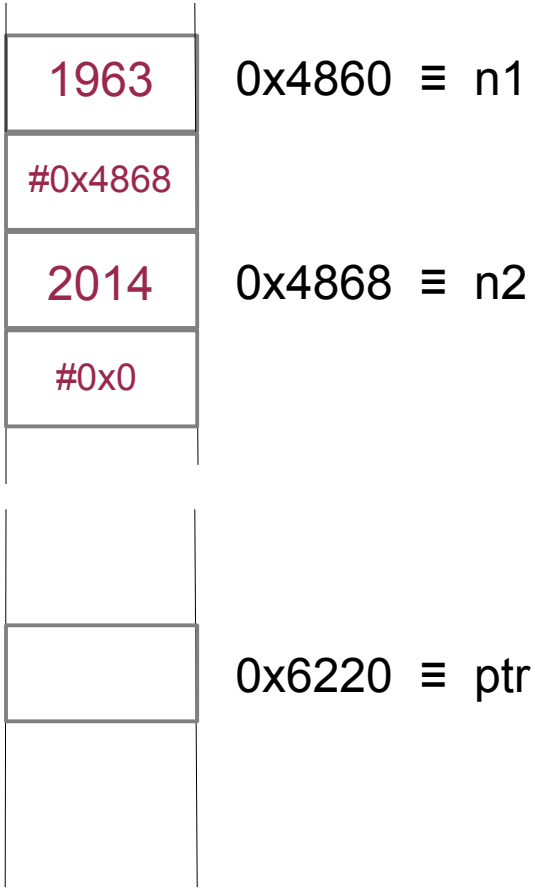

```
struct ListNode {
    int data;
    struct ListNode *next;
}

struct ListNode n1, n2;

void main() {
    n1.data = 1963;
    n2.data = 2014;
}
```

# Static data



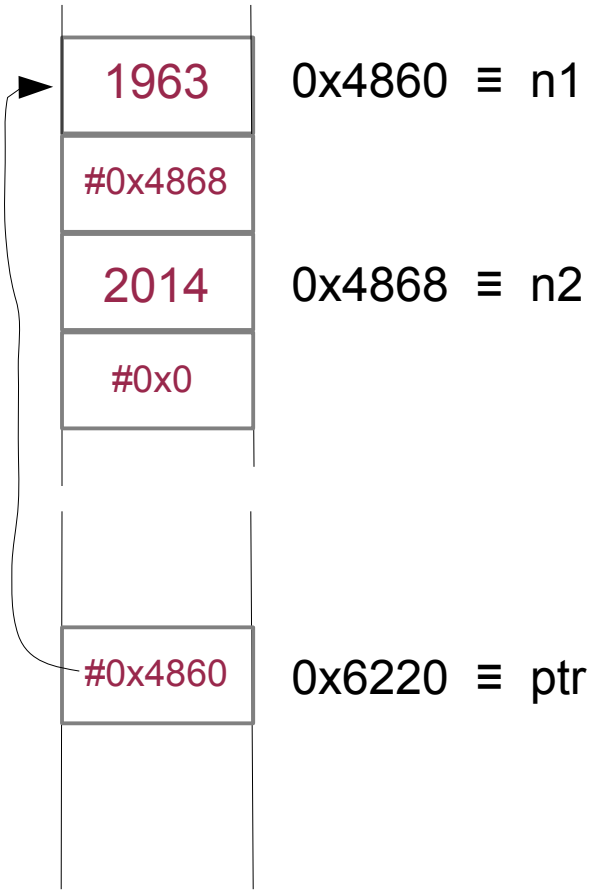

```
struct ListNode {
    int data;
    struct ListNode *next;
}

struct ListNode n1, n2;

void main() {
    n1.data = 1963;
    n2.data = 2014;

    n1.next = &n2;
    n2.next = null;
}
```

# Static data



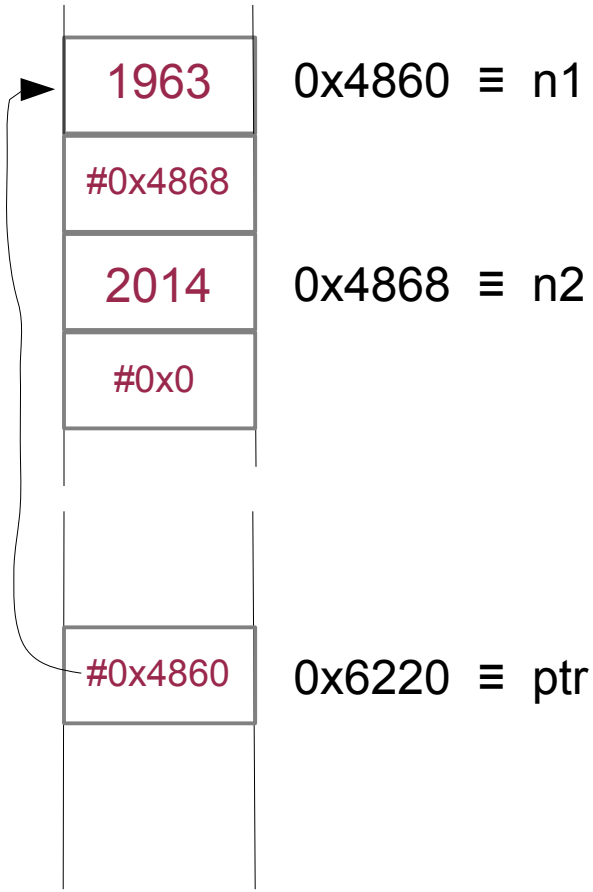

```
struct ListNode {
    int data;
    struct ListNode *next;
}

struct ListNode n1, n2;

struct ListNode *ptr;

void main() {
    n1.data = 1963;
    n1.next = &n2;

    n2.data = 2014;
    n2.next = null;
}
```

# Static data



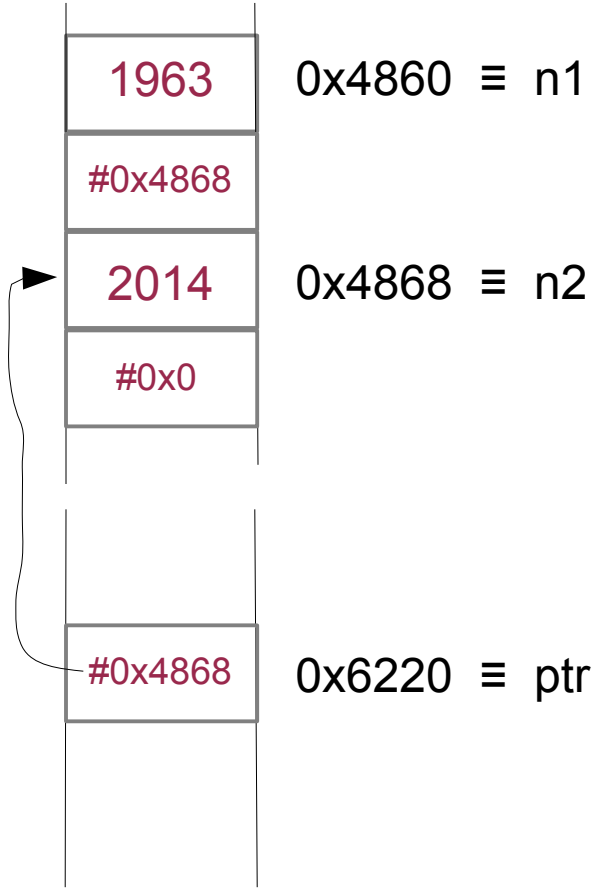

```
struct ListNode {
    int data;
    struct ListNode *next;
}

struct ListNode n1, n2;

struct ListNode *ptr;

void main() {
    n1.data = 1963;
    n1.next = &n2;

    n2.data = 2014;
    n2.next = null;

    ptr = &n1;
}
```

# Static data



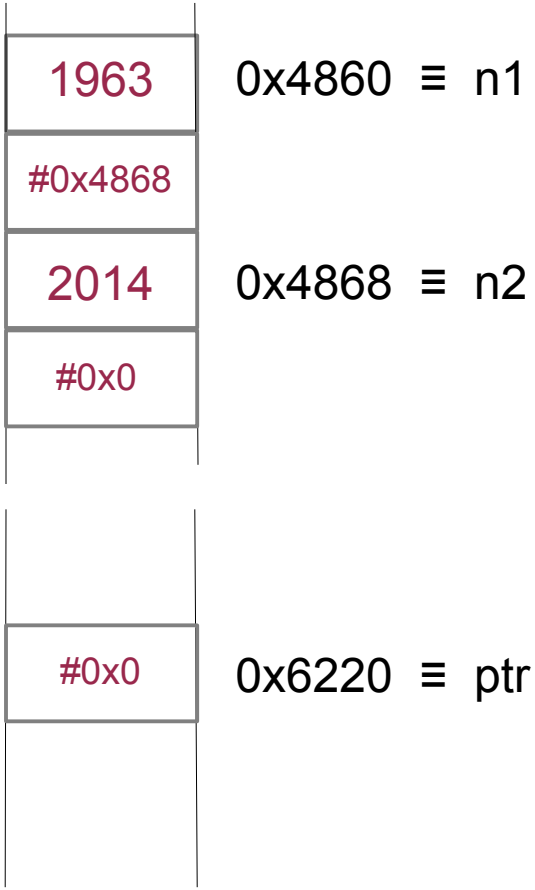

```
struct ListNode {
    int data;
    struct ListNode *next;
}

struct ListNode n1, n2;

struct ListNode *ptr;

void main() {
    n1.data = 1963;
    n1.next = &n2;

    n2.data = 2014;
    n2.next = null;

    ptr = &n1;

    while (ptr) {
        printf("%d\n",
                ptr->data);
        ptr = ptr->next;
    }
}
```

# Static data



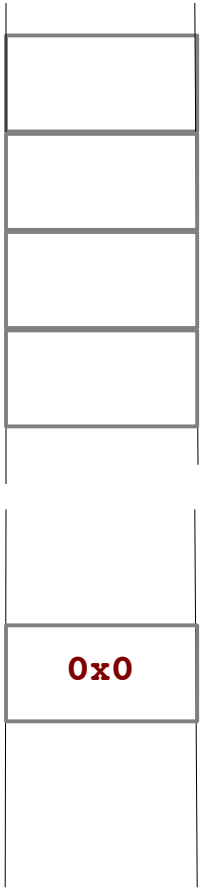

```
struct ListNode {
    int data;
    struct ListNode *next;
}

struct ListNode n1, n2;

struct ListNode *ptr;

void main() {
    n1.data = 1963;
    n1.next = &n2;

    n2.data = 2014;
    n2.next = null;

    ptr = &n1;

    while (ptr) {
        printf("%d\n",
                ptr->data);
        ptr = ptr->next;
    }
}
```

# Static data




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

struct ListNode n1, n2;

struct ListNode *ptr;

void main() {
    n1.data = 1963;
    n1.next = &n2;

    n2.data = 2014;
    n2.next = null;

    ptr = &n1;

    while (ptr) {
        printf("%d\n",
                ptr->data);
        ptr = ptr->next;
    }
}
```

# Static data




0x4880 ≡ n1

0x4888 ≡ n2

0x5330 ≡ ptr

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

struct ListNode n1, n2;

struct ListNode *ptr;

void main() {
}
```

```
ptr->data
```

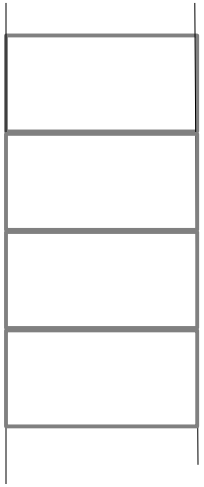
means

Go to the struct whose address is the contents of the ptr variable, and fetch the value of the 'data' component

Same as

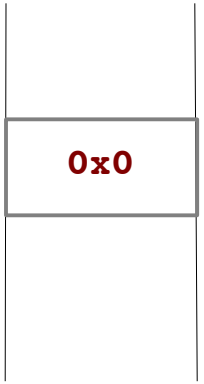
```
(*ptr).data
```

# Static data



0x4880 ≡ n1

0x4888 ≡ n2




0x5330 ≡ ptr

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

struct ListNode n1, n2;

struct ListNode *ptr;

void main() {
}
```

```
ptr = ptr->next
```

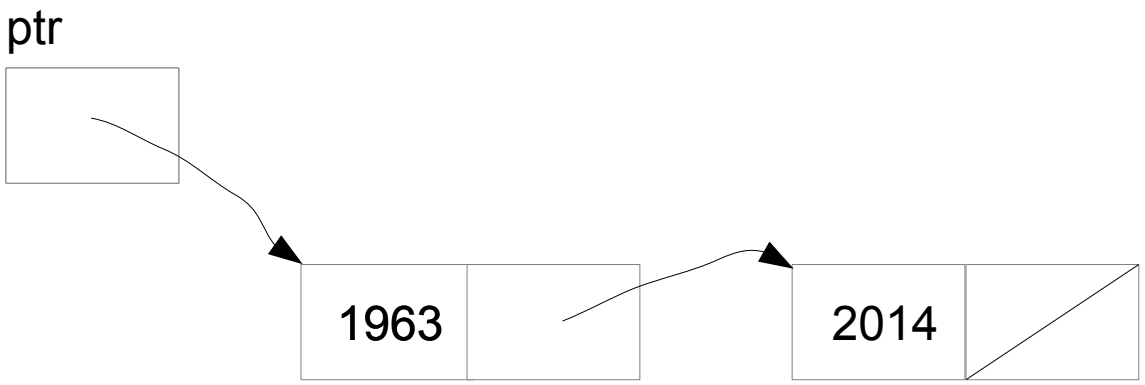
means

Go to the struct pointed to by the ptr variable, fetch the value of the 'next' component, and store that value in the ptr variable.

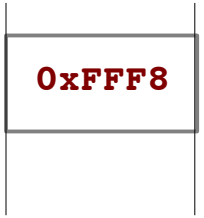
Same as

```
ptr = (*ptr).next
```

Normally we prefer a more abstract representation of a linked list:



# Dynamic data




0x1234 ≡ head

Static data


0xFFF8

Dynamic data
(aka “heap”)

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

struct ListNode *head;

void main() {
  int size = sizeof(struct ListNode);

  void *rawptr = malloc(size);

  head = (struct ListNode *) rawptr;

  head->data = 2015;
  head->next = null;

  …

  free(head);
}
```

# A Linked List abstraction in C

```
typedef int ListData;

typedef struct LLNode {
    ListData data;
    struct LLNode *next;
} ListNode;

typedef struct {
    ListNode *head;
} LinkedList;


LinkedList *newLinkedList()
{
    LinkedList *LL =
        (LinkedList *) malloc( sizeof( LinkedList ) );
    LL->head = null;
    return LL;
}
```

# A Linked List abstraction in C - insert()

```
typedef struct LLNode {
    ListData data;
    struct LLNode *next;
} ListNode;

typedef struct {
    ListNode *head;
} LinkedList;


ListNode *newNode( ListData data, ListNode *next )
{
    ListNode *nodePtr =
        (ListNode *) malloc( sizeof(ListNode) );
    nodePtr->data = data;
    nodePtr->next = next;
    return nodePtr;
}

void LL_insert( LinkedList *LL, ListData data )
{
    LL->head = newNode( data, LL->head );
}
```

# A Linked List abstraction in C - length()

```
typedef struct LLNode {
    ListData data;
    struct LLNode *next;
} ListNode;

typedef struct {
    ListNode *head;
} LinkedList;


int LL_length( LinkedList *LL )
{
    int cnt = 0;
    ListNode *nodePtr = LL->head;
    while ( nodePtr )
        cnt++, nodePtr = nodePtr->next;
    return cnt;
}
```

# A Linked List abstraction in C - empty()

```
typedef struct LLNode {
    ListData data;
    struct LLNode *next;
} ListNode;

typedef struct {
    ListNode *head;
} LinkedList;


void LL_empty( LinkedList *LL )
{
    ListNode *this, *next;
    for ( this = LL->head; this != null; this = next) {
        next = this->next;
        free(this);
    }
    LL->head = null;
}
```

# A Linked List abstraction in C - get()

```
typedef struct LLNode {
    ListData data;
    struct LLNode *next;
} ListNode;

typedef struct {
    ListNode *head;
} LinkedList;


ListData LL_get( LinkedList *LL, int index )
{
    ListNode *nodePtr = LL->head;
    while ( index-- > 0 && nodePtr )
        nodePtr = nodePtr->next;
    if( nodePtr )
        return nodePtr->data;
    else
        error( “out of bounds” );
}
```

# A Linked List abstraction in C - indexOf()

```
typedef struct LLNode {
    ListData data;
    struct LLNode *next;
} ListNode;

typedef struct {
    ListNode *head;
} LinkedList;


int LL_indexOf( LinkedList *LL, ListData data )
{
    int index = 0;
    ListNode *tail = LL->head;
    while ( tail && tail->data != data )
        index++, tail = tail->next;
    return tail ? index : -1;
}
```

# A Linked List abstraction in C - remove()

```
typedef struct LLNode {
    ListData data;
    struct LLNode *next;
} ListNode;
```

```
typedef struct {
    ListNode *head;
} LinkedList;
```

```
void LL_remove( LinkedList *LL, int index )
{
    LL->head = LN_remove( LL->head, index );
}

ListNode *LN_remove( ListNode *this, int index )
{
    ListNode *next;
    if ( this == null )
        return null;
    if ( index == 0 ) {
        next = this->next;
        free(this);
        return next;
    }
    this->next = LN_remove( this->next, index-1 );
    return this;
}
```

# A Linked List abstraction in C - append()

```
typedef struct LLNode {
    ListData data;
    struct LLNode *next;
} ListNode;

typedef struct {
    ListNode *head;
} LinkedList;


void LL_append( LinkedList *LL, ListData data )
{
    ListNode *this = LL->head, *tail;
    if ( this == null )
        LL_insert( LL, data );
    else {
        do {
            tail = this;
        }
        while ( this = this->next );
        tail->next = newNode( data, null );
}
```

# A Linked List abstraction in C – append() *fast*

```
typedef struct LLNode {
    ListData data;
    struct LLNode *next;
} ListNode;

typedef struct {
    ListNode *head, *tail;
} LinkedList;


void LL_append( LinkedList *LL, ListData data )
{
    if ( LL->head == null )
        LL_insert( LL, data );
    else {
        LL->tail = LL->tail->next = newNode( data, null );
}
```