

# ECS 36A, May 9, 2024

# Strings to Numbers

- `int atoi(char *str)`, `long int atol(char * str)`:  
convert *str* to int or long, respectively
- `double atof(char * str)`: convert *str* to double
- No error checking
  - If any non-digit or non-floating character is found, these stop converting and return what they have converted

# Main Loop in *conv1.c*

```
while(fgets(buf, BUFFERSIZE, stdin) != NULL){  
    /* clobber the trailing newline */  
    /* to keep the output clean      */  
    if (buf[strlen(buf)-1] == '\\n')  
        buf[strlen(buf)-1] = '\\0';  
    /* convert to integer and print it */  
    i = atol(buf);  
    printf("\\'%s\\' is the integer %ld\\n", buf, i);  
    /* convert to double and print it */  
    f = atof(buf);  
    printf("\\'%s\\' is the double  %f\\n", buf, f);  
    printf("> ");  
}
```

# Examples

```
% conv1
How atol/atof work:
> 103
'103' is the integer 103
'103' is the double 103.000000
> 111.34
'111.34' is the integer 111
'111.34' is the double 111.340000
> 7a34
'7a34' is the integer 7
'7a34' is the double 7.000000
>
' ' is the integer 0
' ' is the double 0.000000
>
```

Type the command

The first input is "103", which is an integer

The second input is "111.34", which is a floating point number

Note atoi() discards the decimal fraction

The third input is "7a34" – it is *not* a number

Both atoi() and atof() read to the first non-number character (here, "a"), and do *not* give an error; they ignore the rest of the line

The fourth input is nothing – again, it is *not* a number

And the output is 0, which is incorrect

End of file (control-D) ends the program

# Strings to Numbers

- `long int strtol(char *str, char **eostr, int base):` convert *str* to long int in base *base*; return a pointer *eostr* to first char that is not converted
  - If *eostr* is **NULL**, the end pointer is not returned
  - If \**eostr* is '\0', there were no errors
  - If no digits, the value of *str* is put into \**eostr* and returns 0
- `float strtodf(char *str, char **eostr):` convert *str* to float
- `double strtod(char *str, char **eostr):` convert *str* to double

# Main Loop in *conv2.c*

```
while (fgets(buf, BUFFERSIZE, stdin) != NULL) {
    /* clobber the trailing newline to keep the output clean */
    if (buf[strlen(buf)-1] == '\n')
        buf[strlen(buf)-1] = '\0';
    /* convert to integer and print it */
    i = strtol(buf, &eol, 10);
    printf("\'%s\' is the integer %ld; rest of string:  \'%s'\n",
                                                buf, i, eol);
    f = strtod(buf, &eod);
    printf("\'%s\' is the double  %f; rest of string:  \'%s'\n",
                                                buf, f, eod);
    printf("> ");
}
```

# Examples

```
% conv2
How strtol/strtod work:
> 103
'103' is the integer 103; rest of string: ''
'103' is the double 103.000000; rest of string: ''
> 111.34
'111.34' is the integer 111; rest of string: '.34'
'111.34' is the double 111.340000 ; rest of string: ''
> 7a34
'7a34' is the integer 7; rest of string: 'a34'
'7a34' is the double 7.000000; rest of string 'a34'
>
'' is the integer 0; rest of string: ''
'' is the double 0.000000; rest of string: ''
>
```

Type the command

The first input is "103", which is an integer

The second input is "111.34", which is a floating point number

Note strtol() treats the decimal fraction like extraneous matter after the integer

The third input is "7a34" – it is *not* a number

Both strtol() and strtod() read to the first non-number character (here, "a"), and also returns the rest of the line

The fourth input is nothing – it is *not* a number

And the output is 0, which is incorrect

End of file (control-D) ends the program

# C and Files

- Files represented by a *file pointer*
  - Note: the actual representation in Linux is a *file descriptor*, which is a non-negative integer, but that is non-portable; the file pointer is
  - 3 predefined file pointers: *stdin*, *stdout*, *stderr*
- File pointer contains information:
  - Which file is being referenced (ie, the file descriptor)
  - Whether opened for reading, writing, or appending
  - Where in the file the next access is to occur
  - And lots of other information not relevant here



# Predefined File Pointers

- *stdin*: reads from the standard input
  - Usually a terminal or terminal emulator
  - Sometimes from a file, if input is redirected (use “<” in the shell)
  - Sometimes it is the output of another program (use “|”, like “ps | more”)
- *stdout*: writes to the standard output
  - Usually to a terminal or terminal emulator
  - Sometimes to a file, if output is redirected (use “>” in the shell)
  - Sometimes it is the input to another program (use “|”, like “ps | more”)
- *stderr*: writes to the error output
  - Usually to a terminal or terminal emulator, *even if output is redirected*
  - Sometimes to a file, if output is redirected (use “2>” in the bash shell)

# Accessing a file

- To open a file:

```
FILE *fp;  
if ((fp = fopen(filename, mode)) == NULL)  
    error handling
```

- Mode is one of the following:
  - "r" read
  - "w" write – erases file if it exists, creates it if it doesn't
  - "a" append – adds to end of file if it exists, creates it if it doesn't
- All references to the contents and the file itself (usually) use the file pointer
  - Underlying the file pointer is a file descriptor, more about which later

# When You're Done With the File

- Closing the file releases all resources associated with the process and the file
- To close a file:

```
result = fclose(fp)
```

- result is 0 if closed successfully, and **EOF** (-1) if not
- In either case, do *not* refer to that file pointer again!
  - If you do, the results are undefined
- Usual call:

```
(void) fclose(fp)
```

# Part of Main Loop of *printfile1.c*

```
/* open the file */
if ((fp = fopen(fname, "r")) == NULL) {
    fprintf(stderr, "Could not open file %s\n", fname);
    rv++; /* indicates an error */
    continue;
}
/* copy the file, putting line */
/* numbers in front of each line */
copyout(fp);
/* now close it */
(void) fclose(fp);
```

# Reading Text Files

- Read a line:

```
if (fgets(buf, n, fp) == NULL)
```

EOF and error handling

- On success, returns line (or maximum  $n-1$  characters) in *buf*
- On **EOF** or error, returns **NULL**

- Read a character:

```
if ((ch = fgetc(fp)) == EOF)
```

EOF and error handling

- On EOF or error, returns EOF
- On success, returns *ch* as an unsigned char cast to an int

# Reading Text Files

- Other ways to read characters:

- `getc(fp)`      same as `fgetc(fp)`
- `getchar()`    same as `getc(stdin)`

- Unread characters (really, push them back into the input stream):

```
if (ungetc(ch, fp) == EOF)
```

*handle error*

- Important: only 1 character of pushback is guaranteed
- If more than that are pushed back (bad idea), they will be read in reverse order of pushback

# Writing Text Files

- Write a line:

```
if (fputs(buf, fp) == EOF)
    handle error
```

- On success, returns a non-negative integer (it does *not* append a newline)
- On error, returns **EOF**

- Write a character:

```
if (fputc(ch, fp) == EOF)
    error handling
```

- On error, returns **EOF**
- On success, returns ch as an unsigned char cast to an int

# Writing Text Files

- Other ways to write lines:

```
if (puts(buf) == EOF)
    handle error
```

- On success, writes *buf*'s contents followed by a newline and returns a non-negative integer
  - On error, returns **EOF**
- Other ways to write characters:
    - `putc(ch, fp)`            same as `fputc(ch, fp)`
    - `putchar(ch)`            same as `putc(ch, stdout)`



# First *copyout* Routine

```
void copyout(FILE *fp)
{
    char buf[MAXLINELENGTH]; /* buffer to hold line */
    static int lineno = 1; /* current line number */

    /* read until done */
    while(fgets(buf, MAXLINELENGTH, fp) != NULL) {
        /* not done yet so print line number */
        printf("%6d\t", lineno++);
        fputs(buf, stdout); /* and the line */
    }
}
```

# First *copyout* Routine Notes

- `fgets()` reads a line of up to 1023 (`MAXLINELENGTH-1`) characters from the input
- It then prints a line number using `printf()`
  - Note there is no `'\n'` there so the next output is on the same line
- And it then uses `fputs()` to print the line
  - Note `fgets()` preserved the newline ending in `buf`, so `fputs()` prints the newline in `buf` also
- ***Bonus question***: when does this print two output lines for a single input line?
  - Suppose this is line 100 in the file; when might that same line be printed with line numbers 100 and 101?

# Example Run of *printfile1*

```
% cat XYZZY
```

```
Hello. This is an example  
file to demonstrate how  
the printfile programs work.
```

```
% printfile1
```

```
file name> XYZZY
```

```
1 Hello. This is an example  
2 file to demonstrate how  
3 the printfile programs work.
```

```
file name>
```

# Second *copyout* Routine

```
void copyout(FILE *fp)
{
    char buf[MAXLINELENGTH]; /* buffer to hold line */
    int i;                    /* counter in a for loop */
    static int lineno = 1; /* current line number */

    /* read until done
    while(fgets(buf, MAXLINELENGTH, fp) != NULL){
        /* not done yet so print line number */
        printf("%6d\t", lineno++);
        /* use a for loop to demonstrate sequential access */
        for(i = 0; buf[i] != '\0'; i++)
            putchar(buf[i]);
    }
}
```

# Second *copyout* Routine Notes

- `fgets()` reads a line of up to 1023 (`MAXLINELENGTH-1`) characters from the input
- It then prints a line number using `printf()`
  - Note there is no `'\n'` there so the next output is on the same line
- It then uses `putchar()` to write the line, character by character, on the standard output
- ***Bonus question***: Will this also print two output lines for a single input line, under the same conditions as `printf1.c`?

# Example Run of *printfile2*

```
% cat XYZZY
Hello. This is an example
file to demonstrate how
the printfile programs work.
% printfile1
file name> XYZZY
    1 Hello. This is an example
    2 file to demonstrate how
    3 the printfile programs work.
file name>
```

Notice the output is the same as for *printfile1*; only the way the output is generated is different (`putchar()` vs. `fputs()`)

# Part of Main Loop of *printfile3.c*

```
int main(int argc, char *argv)
{
    /* . . . omitting part of program . . . */
    for(i = 1; i < argc; i++){
        /* open the file */
        if ((fp = fopen(argv[i], "r")) == NULL){
            fprintf(stderr, "Could not open file %s\n", argv[i]);
            continue;
        }
        /* copy the file, putting line numbers in front of each line */
        copyout(fp);
        /* now close it */
        (void) fclose(fp);
    }
}
```

# *printfile3* Notes

- Here you can name the file to be printed on the command line
  - The program does not prompt for a file name
  - It goes down the argument list and prints each file named in that list
  - If no files named, argc is 1, and it skips the for loop
- Note that the `fprintf()` says the file could not be opened
  - But it does not say why the file could not be opened
- A better way to say what went wrong when the program tried to open the file follows . . .



# Example Run of *printfile3*

```
% cat XYZZY
```

```
Hello. This is an example  
file to demonstrate how  
the printfile programs work.
```

```
% printfile3 XYZZY XXX
```

```
1 Hello. This is an example  
2 file to demonstrate how  
3 the printfile programs work.
```

```
Could not open file XXX
```

Notice the output is the same as for *printfile1* and *printfile2* when the named file exists. If it does not exist, you get an error message.

# Error Handling

- Error codes are stored in a global variable *int errno*
  - Include the header file “errno.h”
- How to access the error code:

1. Print corresponding system error message

```
 perror(str)
```

2. Get a pointer to the corresponding system error message

```
char *strerror(errno)
```

# Error Handling

Getting a list of system error messages

- Get the number of system error messages

```
const int sys_nerr
```

- Get the list of the system error messages

```
const char * const sys_errlist[]
```

- **WARNING:** This does not work on the CSIF; those variables are undefined
  - Not sure if this is true on non-Ubuntu versions of Linux

# Part of Main Loop of *printfile4.c*

```
int main(int argc, char *argv)
{
    /* . . . omitting part of program . . . */
    for(i = 1; i < argc; i++){
        /* open the file */
        if ((fp = fopen(argv[i], "r")) == NULL){
            perror(argv[i]); ← perror() rather than fprintf()
            continue;
        }
        /* copy the file, putting line numbers in front of each line */
        copyout(fp);
        /* now close it */
        (void) fclose(fp);
    }
}
```

# Example Run of *printfile4*

```
% cat XYZZY
```

```
Hello. This is an example  
file to demonstrate how  
the printfile programs work.
```

```
% printfile4 XYZZY XXX
```

```
1 Hello. This is an example  
2 file to demonstrate how  
3 the printfile programs work.
```

```
XXX: No such file or directory
```

Notice the output is the same as for *printfile1* and *printfile2* when the named file exists. If it does not exist, you get an error message that is more informative than the previous one

# Part of Main Loop of *printfile5.c*

```
int main(int argc, char *argv)
{
    /* . . . omitting part of program . . . */
    for(i = 1; i < argc; i++){
        /* open the file */
        if ((fp = fopen(argv[i], "r")) == NULL){
            fprintf(stderr, "Error number %d: %s (file %s)\n", errno,
                strerror(errno), argv[i]);
            continue;
        }
        /* copy the file, putting line numbers in front of each line */
        copyout(fp);
        /* now close it */
        (void) fclose(fp);
    }
}
```

This is the file name

This is the error number

This is the the error string (what perror() prints)

# Example Run of *printfile5*

```
% cat XYZZY
```

```
Hello. This is an example  
file to demonstrate how  
the printfile programs work.
```

```
% printfile5 XYZZY XXX
```

```
1 Hello. This is an example  
2 file to demonstrate how  
3 the printfile programs work.
```

```
Error number 2: No such file or directory (file XXX)
```

The output is the same as for the other *printfiles* when the named file exists. If it does not exist, you get an even more informative error message than the previous ones

# *syslist.c*

- This was run on MacOS 14.1 (which runs a version of UNIX)

```
#include <stdio.h>
#include <errno.h>

int main(void)
{
    int i;          /* counter in a for loop */

    /* loop through the list, printing each message and error number */
    for(i = 0; i < sys_nerr; i++)
        printf("%3d. %s\n", i, sys_errlist[i]);

    return(0);
}
```



# Formatted Read

These read input or a string, and attempt to match the input with the format. If any match the desired input. the appropriate variable is set. It stops when it reads the first character that does not match the format.

- *scanf(format, variables)*
  - Same as *fscanf(stdin, format, variables)*
- *sscanf(string, format, variables)*
  - Reads from the given *string*
- *fscanf(fp, format, variables)*
  - Reads from the file with file pointer *fp*

# Formatted Write

These write to a file or the standard output as indicated by the format string.

- `printf(format, variables)`
  - Same as `fprintf(stdout, format, variables)`
- `sprintf(string, format, variables)`
  - Writes its output into *string*
- `fprintf(fp, format, variables)`
  - Writes to the file with file pointer *fp*

# Useful File Functions

```
int feof(FILE *fp)
```

- Nonzero if file pointer *fp* is at **EOF**; 0 if not

```
int ferror(FILE * fp)
```

- Nonzero if error indicator for *fp* is set; 0 if not

```
void clearerr(FILE * fp)
```

- Clear **EOF** and error indicators for *fp*

# Reading Binary Files

- Binary files contain non-ASCII values; for example, integers are written into the file as integers, not printed representations of integers

```
size_t fread(void *ptr, size_t sz, size_t num, FILE *fp)
```

- `size_t` is type used for sizes of things — it's a long unsigned int
- Reads *num* items of size *sz* from file *fp* and stores them beginning at address *ptr*
- *ptr* can point to any type (it's declared as `void *ptr`)
- Returns number of items actually read; on error or **EOF**, returns the number of items actually read (or 0 if none were)
  - Use `feof()` and `ferror()` to determine if it's an **EOF** or an error when it returns 0

# Writing Binary Files

- Writes out the data in the form it is stored in in memory.

```
size_t fwrite(void *ptr, size_t sz, size_t num, FILE *fp)
```

- Writes *num* items of size *sz* beginning from address *ptr* to file *fp*
- *ptr* can point to any type (it's declared as `void *ptr`)
- Returns number of items actually written; on error, returns the number of items actually written (or 0 if none were)

# Part of Main Routine of *sbcopy.c*

```
/* be sure there are 2 file names! */
if (argc != 3){
    fprintf(stderr, "Usage: scopy fromfile tofile\n");
    return(1);
}
/* open the input and output files */
if ((infp = fopen(argv[1], "r")) == NULL){
    perror(argv[1]);
    return(1);
}
if ((outfp = fopen(argv[2], "w")) == NULL){
    perror(argv[2]);
    return(1);
}
```

On Linux, it does not matter whether the files are binary data or text. On some other systems, like Windows, it might. On those systems, use "rb" and "wb" rather than "r" and "w", respectively.

# Part of Main Routine of *sbcopy.c*

```
while((nread = fread(buf, sizeof(char), BUFFERSIZE, infp)) != 0)
    if (fwrite(buf, sizeof(char), nread, outfp) != nread) {
        perror(argv[2]);
        return(1);
    }
if (ferror(infp)) {
    /* the read failed because of an error, not an EOF */
    fprintf(stderr, "%s: read error; quitting\n", argv[1]);
    return(1);
}
```

# Another Recursive Program: sort.c

- This sorts integers by finding the smallest number and putting it at the beginning
- Basic idea:
  - if number of elements in list is 1 or 0:
    - list is sorted – just return
  - find the smallest number in the list
  - swap it and the first number
  - sort the rest of the list



# Problem

- `sort.c` reads from an array of known length
- User must enter numbers into the program
- The compiler can compute the length (or the user can enter it)

So how do we get around this?

# Dynamic Memory Allocation

- Static memory allocation occurs when you declare a variable

```
int num;
```

- Compiler creates space for this variable
- There is also a pool of memory (the “heap”) that is available but initially unused
- Dynamic memory occurs when you obtain memory space this pool
  - Allocate: obtaining the space from the pool
  - Allocation: the amount of space you get
  - Deallocate, free: releasing memory that has been allocated; it goes back to the pool

# A Useful Operator

- To get the number of bytes in a data type, use `sizeof`
- Example: on a 32-bit machine:
  - `sizeof(char)` is 1
  - `sizeof(int)` is 4
  - `sizeof(float)` is 4
  - `sizeof(double)` is 8
- Works for variables, too
  - if  $a$  is an int, `sizeof(a)` is 4

# But Be Careful!

```
char a[100]
```

- Tempting to get the size of an array like this:

```
sizeof(a)
```

- Here, `a` is a pointer constant, so `sizeof` returns the number of bytes in that pointer, *not* the size of the array!
- To get the number of bytes in an array, use

```
sizeof(a[0]) * 100
```

where 100 is the number of elements in the array

- The `a[0]` is one element; works as all elements are of the same type

# Allocation Functions: *malloc()*

- Basic function

```
void *malloc(size_t space)
```

- Allocate *space* bytes of memory, returning its address; returns NULL if not available
  - Type `size_t` is same as unsigned int
- Declared `void *` so that it can be coerced into any type of pointer

```
char *p;
```

```
if ((p = (void *) malloc(100)) == NULL)
```

*error handling*

# Allocation Functions: *calloc()*

- Variant

```
void *calloc(size_t nelt, size_t space)
```

- Like malloc, but:
  - Gives you space in terms of elements and size of element, rather than a number of bytes
  - Memory is zeroed out; malloc() does not do so, and whatever is in that memory before call to malloc() is there once allocated

# Allocation Functions: *realloc()*

- Enlargening space already allocated (say `pmem` points to it):

```
void *realloc(void *pmem, size_t nbytes)
```

- This allocates *nbytes* of space, and the contents of `*pmem` are copied into the beginning of the new space
  - The new space may simply extend what `pmem` points to
  - Or, it may be completely new space, in which case what `pmem` points to is deallocated
  - If insufficient memory available, returns `NULL` and leaves the space `pmem` points to untouched, neither moved nor deallocated

# Allocation Functions: *realloc()*

- Common way to use this:

```
if ( (pmem = realloc (pmem, 1000) ) == NULL) . . .
```

- On success, pmem now points to a chunk of memory of size 1000 bytes
- On failure, pmem is now NULL — and you lose the address of the memory pmem used to point to

- Here's the right way:

```
tempptr = realloc (pmem, 1000);  
if (tempptr == NULL) error handling;  
else pmem = tempptr;
```



# Deallocation Function: *free()*

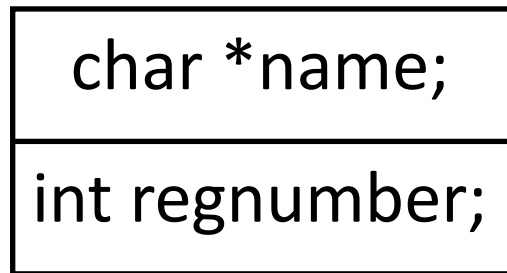
- To release memory allocated by one of the allocation functions, use:
- `void free(void *pmem)`
- If `pmem` is `NULL`, this does nothing
- Do *not* free memory that has already been freed!
  - Called a *double free error* and can often be a vulnerability
  - In all cases, the result is undefined

# Another Recursive Program: usort1.c

- Problem with earlier sort.c: numbers are embedded in program
- Better: have users enter the numbers
- Basic idea:
  - ask user how many numbers they want sorted
  - allocate the space
  - read in that many integers – if EOF entered, quit at once

# Structures

- Data structure used to group elements of a different type together
- Example: student registration number database
  - See element below



**type of structure**

**field**

```
struct student {  
    char *name; /* student name */  
    int regnumber; /* registration number */  
};
```

# Referring to a Structure

Here's how you declare a variable of the structure:

```
struct student xyzzy, *pxyzzy;
```

It's clumsy to write that, so you can define an alias for the type:

```
typedef struct student STUDENT;
```

The latter essentially produces a new type, `STUDENT`, that can be used wherever `struct student` can:

```
STUDENT xyzzy, *pxyzzy;
```

# Another Declarations

```
struct student {  
    char *name;      /* student name */  
    int  regnumber;  /* registration number */  
} xyzzy, *pxyzzy;
```

- Declares type `struct student` and 2 variables, `xyzzy` (an instance of `struct student`) and `pxyzzy` (a pointer to an instance of `struct student`)

# And Now, With a Typedef

```
typedef struct student {  
    char *name;        /* student name */  
    int  regnumber;    /* registration number */  
} STUDENT;  
  
STUDENT xyzzy, *pxyzzy;
```

This defines a new type, `STUDENT`, which is the same as the type `struct student`. Here `xyzzy` is a variable of type `STUDENT` and `pxyzzy` is a pointer to an instance of `STUDENT`.

# But Be Careful

- typedef defines an alias for a type
- #define does textual substitution

```
typedef int *PINT;
```

```
PINT a, b, c
```

- Now a, b, and c are all pointers to integers

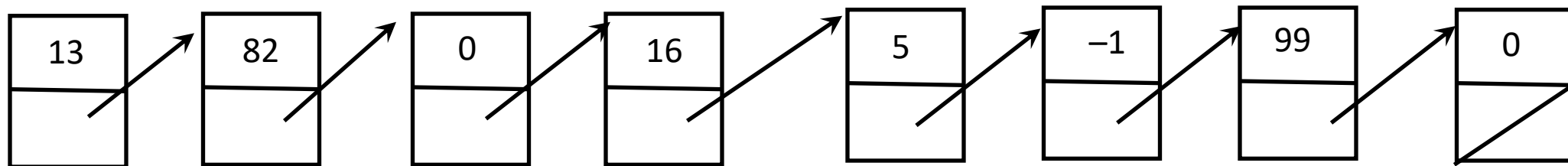
```
#define PINT int *
```

```
PINT a, b, c; /* becomes int * a, b, c; */
```

- Now a is a pointer to an integer, and b and c are integers

# Linked List

- A list composed of instantiations of structures
  - One element is whatever is to be sorted (int, for us)
  - Another element is a pointer to the next element; NULL if none






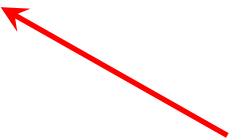
# Structure for This List

```
struct node {  
    int num;  
    struct node *next;  
};  
struct node *list;
```

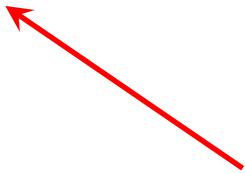
This holds the integer  
that you read in



This holds the pointer  
to the next element  
in the linked list; it's  
NULL if it's at the end



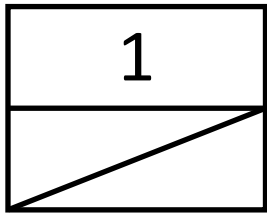
This points to the first  
element of the list



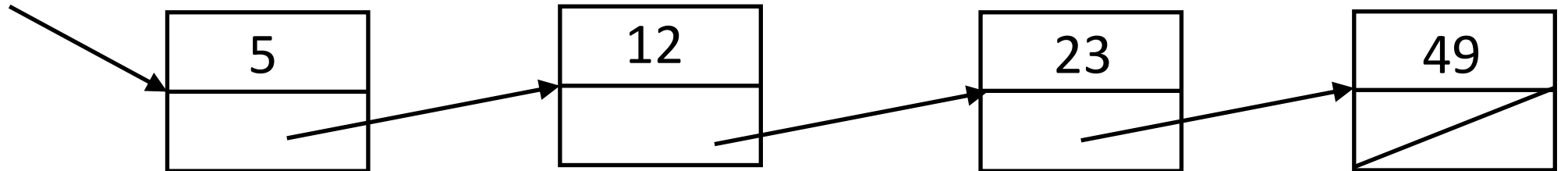
# Changing How Memory Is Allocated

- Now you can allocate memory one element (“node”) at a time
- Insertion at beginning is like this (see “linked.c”, ll. 72–76):
  - `new->next = first;`
  - `list = new;`
- Insertion in the middle between `prev` and `succ` is (see “linked.c”, ll. 78–97):
  - `new->next = succ;`
  - `prev->next = new;`
- Insertion at the end nomore of the list (same as above):
  - `nomore->next = new;`

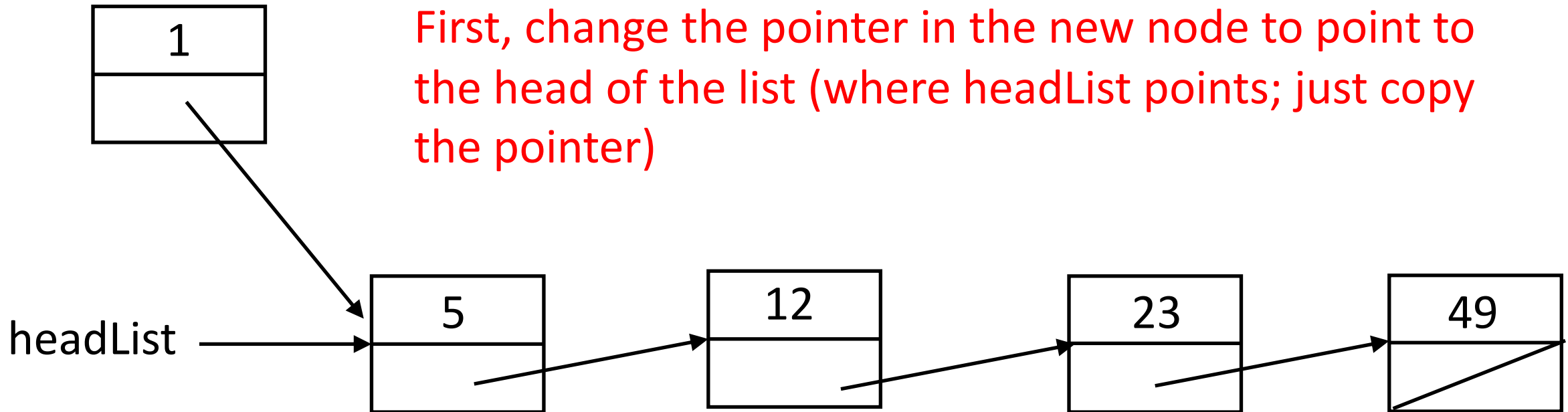
# Insertion



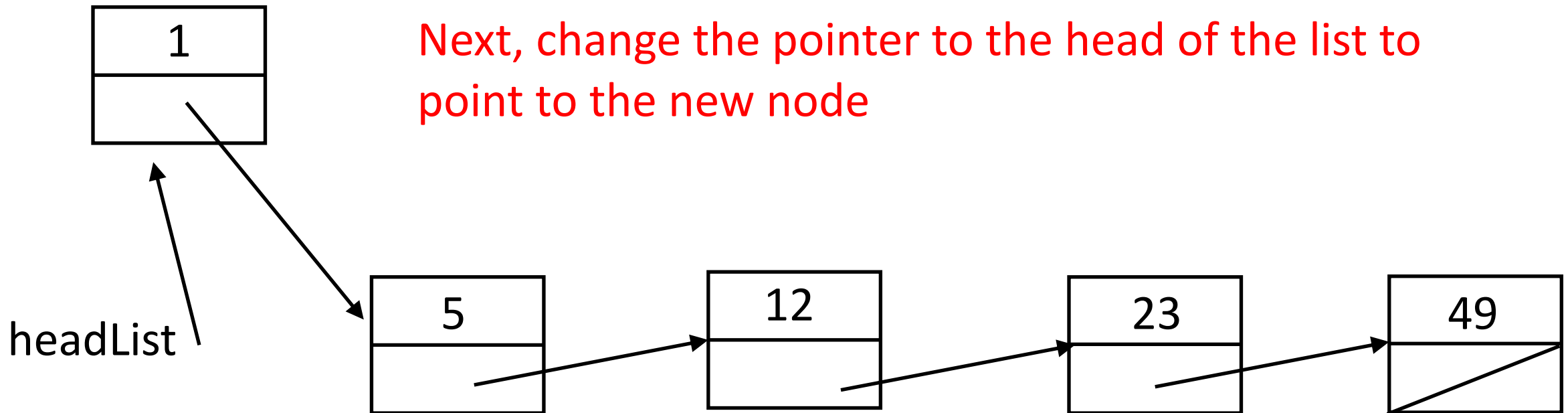
headList



# Insertion: At the Beginning of the List



# Insertion: At the Beginning of the List



# Code for This

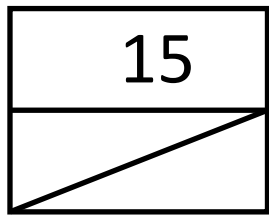
- `new` is a pointer to the new node, `headList` points to the head of the list
- First, make `new` point to the old head. of the list

```
new->next = headList;
```

- Next, make the pointer to the head of the list point to `new`

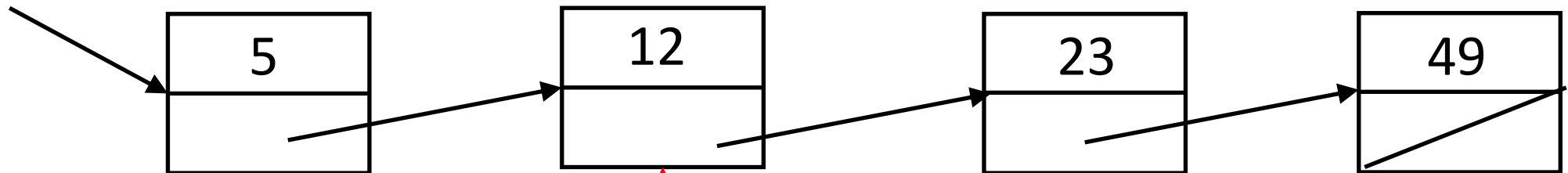
```
headList = new;
```

# Insertion: In the Middle of the List



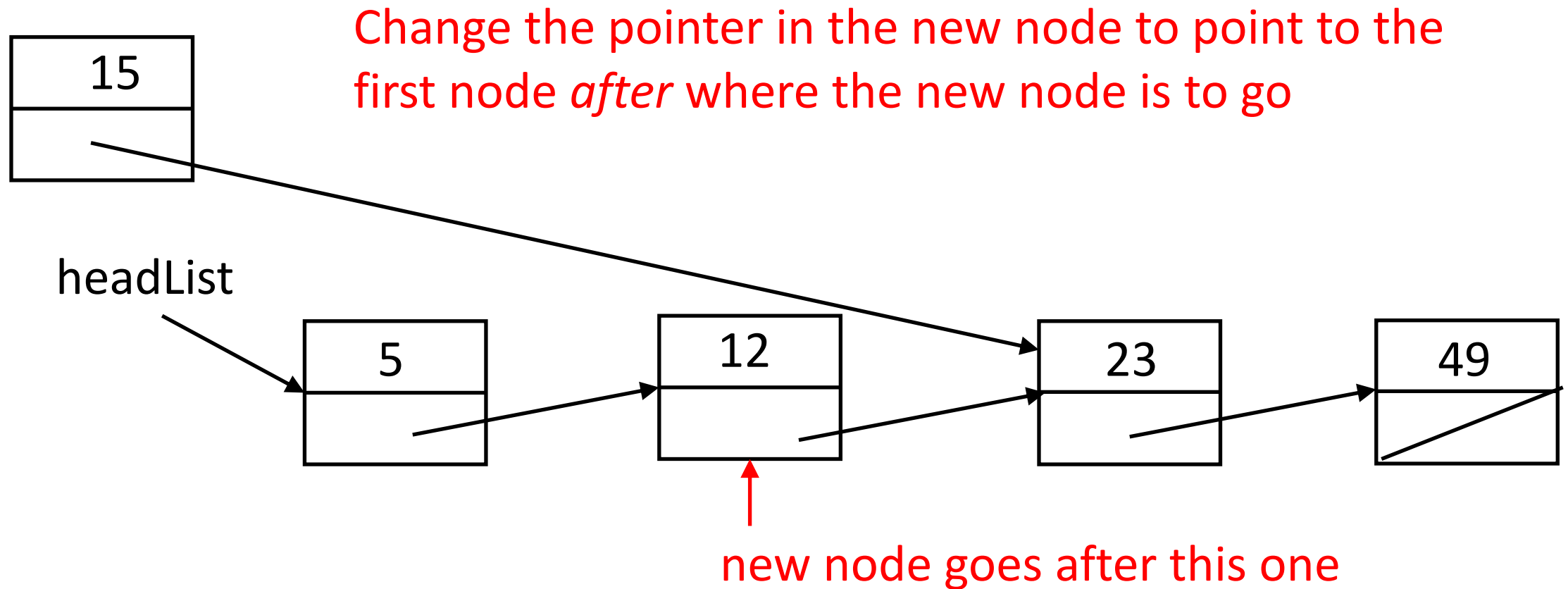
First, scan down the list until you reach the node before which the new node goes.

headList



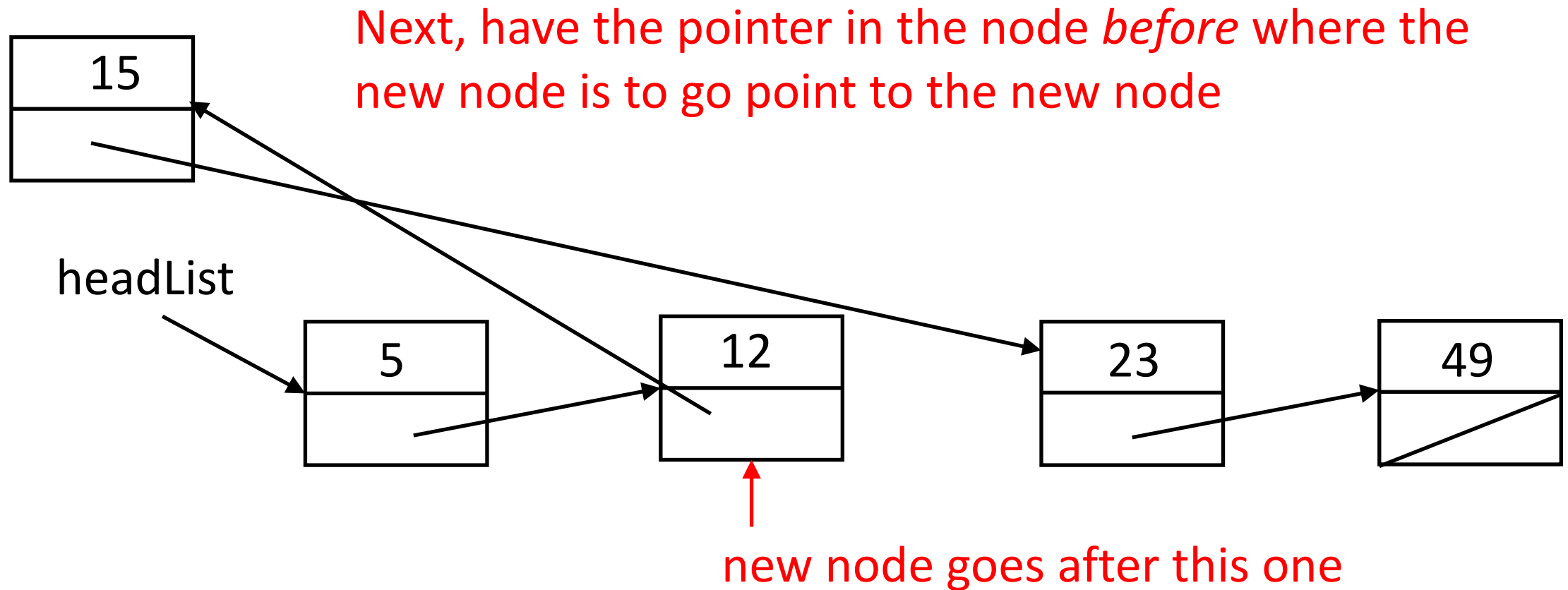
new node goes after this one

# Insertion: In the Middle of the List





# Insertion: In the Middle of the List



# Code for This

- `new` is a pointer to the new node, `headList` points to the head of the list, and `p` is a pointer to node
- First, find the node that `new` goes after

```
for (p = headList;  
     p != NULL && p->next < new->next;  
     p = p->next)  
    /* do nothing ;
```

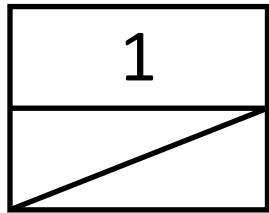
- Next, change the pointer in `new` to point to the node *after* where this one goes

```
new->next = p->next;
```

- Finally, make the node `p` points to point to `new`

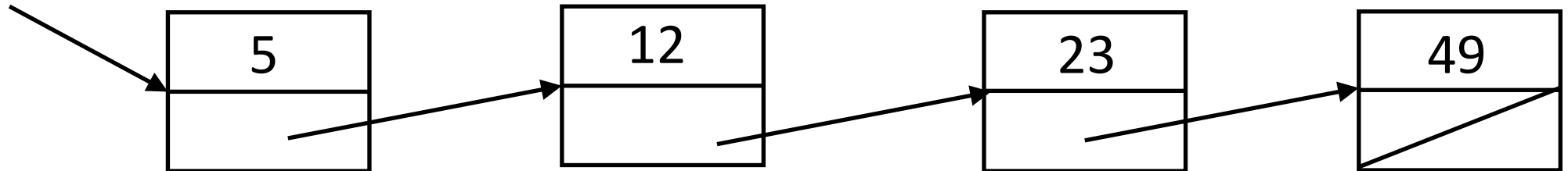
```
p->next = new;
```

# Insertion: At the End of the List



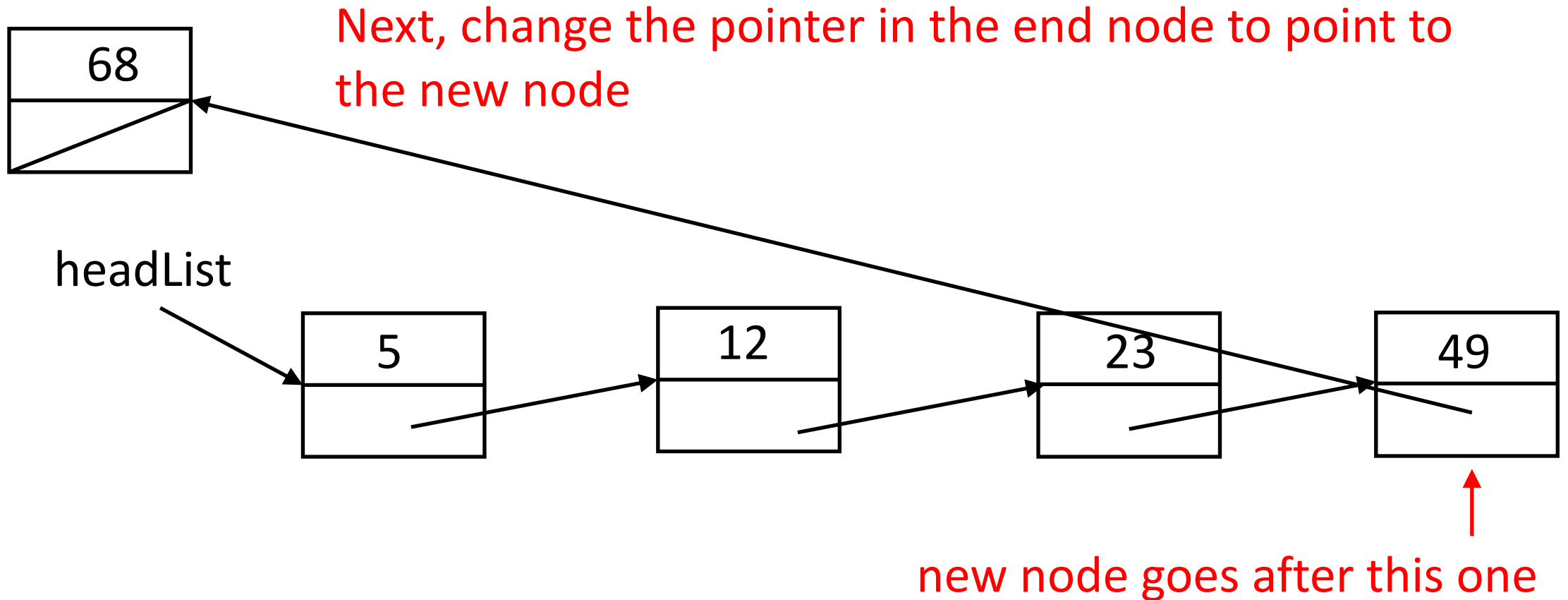
First, scan down the list until you reach the end node

headList



new node goes after this one

# Insertion: At the End of the List



# Code for This

- `new` is a pointer to the new node, `headList` points to the head of the list, and `p` is a pointer to node

- First, find the node at the end

```
for (p = headList;  
     p != NULL && p->next != NULL;  
     p = p->next)  
    /* do nothing */;
```

- Next, change the pointer in what `p` points to to point to `new`

```
p->next = new;
```

- This may be an excess, but make sure `new`'s pointer field is `NULL`

```
new->next = NULL;
```

# Multiple Arrays

- Need to store several data of different types about something
- Example: sort planets by their diameters
- Use 2 arrays
  - `char *names[9]`
  - `int diameters[9]`
- When sorting, need to keep both arrays aligned
  - So when swapping 2 elements of array diameter, the corresponding elements of array names must also be swapped
- Alternate approach: use structures!

# Same with Structures

- Instead of 2 arrays, combine into one structure for each element, and use an array of structures

```
struct celestial {  
    char *name;    /* pointer to name of planet */  
    int diameter; /* diameter of planet in km */  
} planets[9];
```

- This allocates space for 9 planets
- When you swap elements, you only need to swap one, not two, as in the parallel arrays case