ALGONQUIN COLLEGE

MODULE 7: ABSTRACT DATA TYPES AND FILE I/O

Professor : Dave Houtman

Office: T323

Office Hrs: Monday 15:30 – 16:00

Wednesday 15:30 – 16:00

Friday 15:30 – 16:00

Email: houtmad@algonquincollege.com

7.0 Abstract Data Types

- An ***abstract data type*** (or **ADT**) is a term used to mean a collection of data, combined with the functions needed to manipulate it. Such information could potentially be stored in a structure. In other words, an ADT is a primitive form of what you now know as a class containing both properties (variables) and methods (functions).

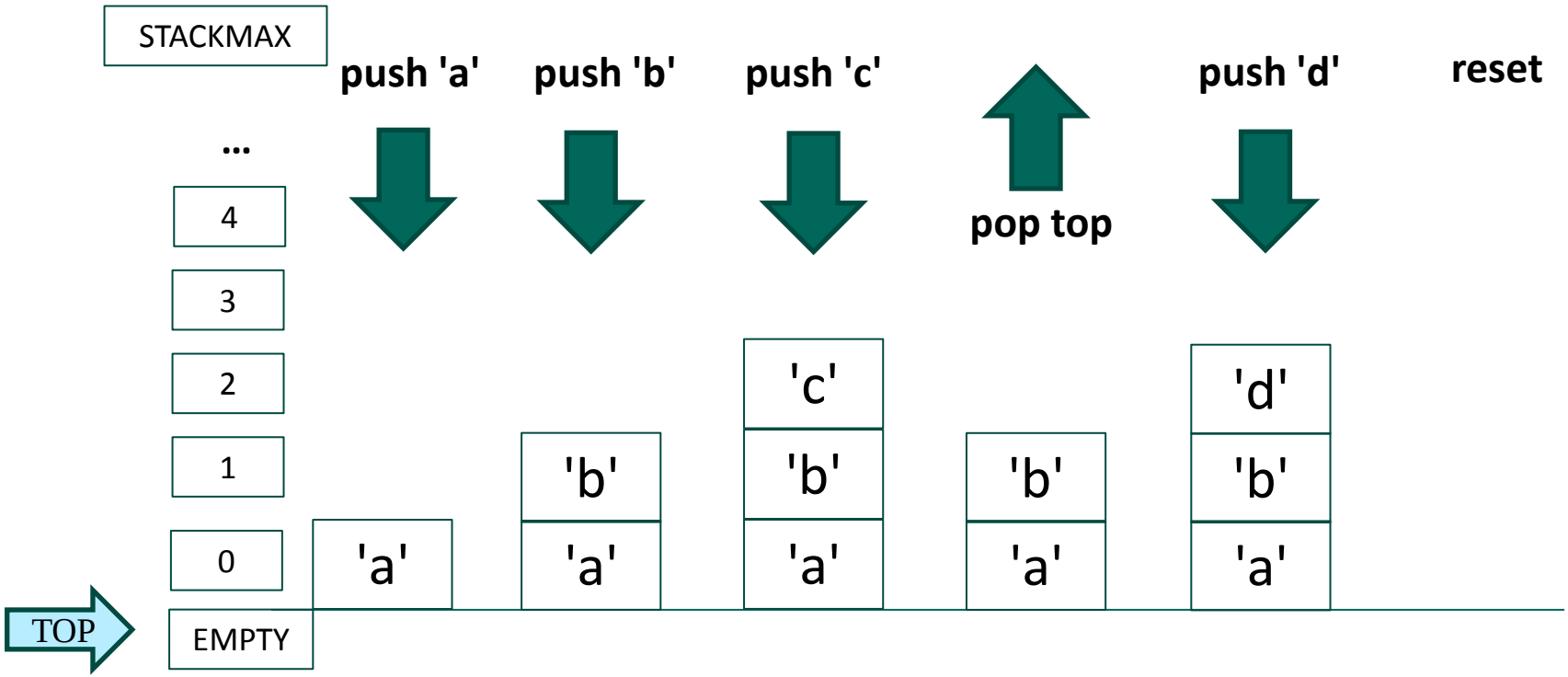
- A data structure is something simpler: "A data structure is a construct within a programming language that stores a collection of data"*--but does not include the functions that operate on that data.

- A data structure is often implemented as part of an ADT (since the data and its operations must be tightly bundled together). But while they are often related, an ADT is *not* the same thing as a data structure.

*From: Carrano F.M., Helman P., and Veroff, R. Data Abstraction and Problem Solving with C++: Walls and Mirrors, Addison-Wesley, 1998. pg. 109

7.1 ADTs – A Simple ADT Stack

- The following code implements the simplest type of data structure, a stack.
- To keep the implementation *very* simple, our stack will be stored as a predeclared array of individual characters, with the first characters entered on the bottom of the stack, the last characters on the top.



7.1 ADTs – A Simple ADT Stack

- Logically, a stack needs to know about two things: where its data is stored, and where the top of the stack is.

  In the following implementation, the top of the stack points to the *last* location at which data is stored.

  When the stack is empty, this means that the top of the stack is set to a negative value. Then, when the first value is added to the array, the top of the stack is first incremented by +1 (to 0), and the first element of the list added to the array at location [0].

7.1 ADTs – A Simple ADT Stack

- We begin by setting a few basic properties of the stack. Of these, the `stack` itself is the most important. Our stack is just an array of chars set to size `STACKMAX`. The integer `top` acts as an index to the location of the where last character was entered.

```
#define STACKMAX          200
#define EMPTY             -1
#define FULL              (STACKMAX - 1)

typedef enum boolean {false, true} boolean;

typedef struct stack {
   char      s[STACKMAX]; //stack stored here
   int       top;         //last index ptd to
} stack;
```

7.1 ADTs – A Simple ADT Stack

- Note that, with these variables, given a pointer to the structure `stack`,

```
stack *stk;
```

we can access its members using the –> notation, as follows

```
stk -> top  // integer value stored in top,
            // this is the index of the
            // last array location

stk -> s[0] // character value stored in
            // first stack element

stk -> s[FULL] // last element in the stack

stk -> s[stk -> top] // the element at the
            // current top of the stack
```

7.1 ADTs – A Simple ADT Stack

- To check to see if the array is empty, we need to know if the value of `top` is equal to `EMPTY` (= -1). If true, return our boolean `true`, otherwise boolean `false`:

```
boolean isEmpty (stack *stk){
    return((boolean) (stk -> top == EMPTY));
}
```

Similarly, if we want to check to see if our stack is full, we can return

```
boolean isFull (stack *stk){
    return((boolean) (stk -> top == FULL));
}
```

7.1 ADTs – A Simple ADT Stack

- Notice how the `boolean` cast works here. We declared the `boolean` enumeration type as:

```
typedef enum boolean {false, true} boolean;
```

So `false` and `true` are both `const int`s. Expressions like

`(stk -> top == EMPTY)` and `(stk -> top == FULL)`

return 0 or 1 depending on the result of the comparison. That value is cast as a `boolean false` or `boolean true` by an expression like

```
((boolean) (stk -> top == FULL))
```

7.1 ADTs – A Simple ADT Stack

- To return the value stored at value pointed to by `top`, we'll need a function that reads in a pointer to the stack, and returns…the `top` element of stack array.

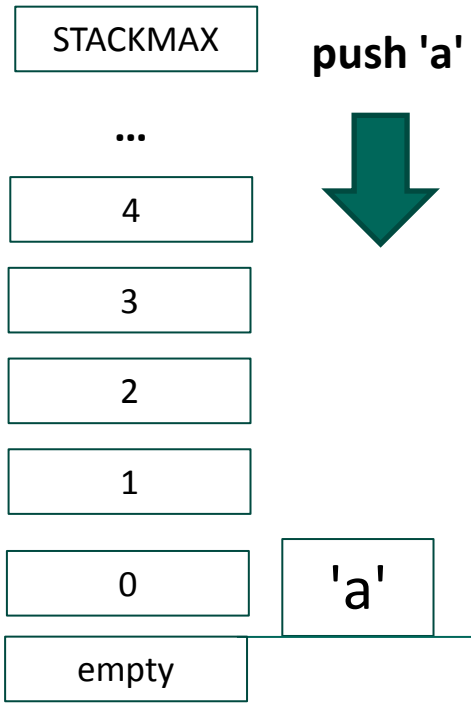
```
char top(stack *stk){
   return(stk -> s[stk -> top]);
}
```

And to reset the stack, simply assign the value of `top` to `EMPTY`:

```
void reset (stack *stk){
   stk -> top = EMPTY;  // where EMPTY = -1
}
```

7.1 ADTs – A Simple ADT Stack

- Assume we wish to add a character to our stack structure. For this we need a function that takes both the character to be added, and the stack itself. The character is added to the stack at the next location above the current `top` value. We'll assume the stack is in its reset condition, with `stack->top` set to `EMPTY` (i.e. -1). Therefore, our function looks like:

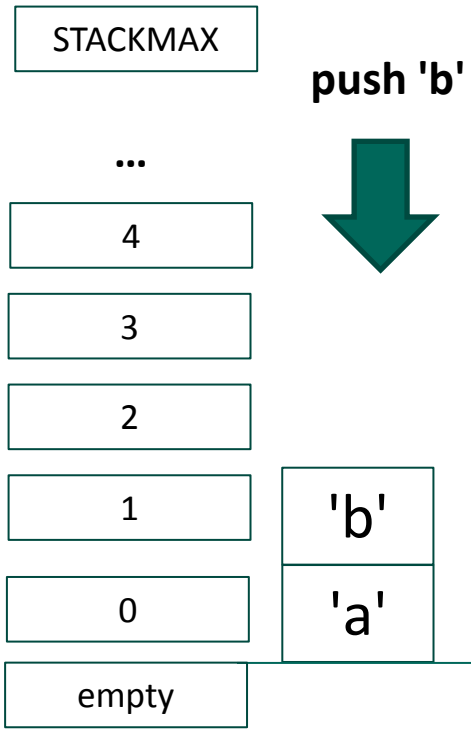



```
void push (char ch, stack *stk){
    stk -> top++; //inc from -1 to 0
    stk -> s[stk -> top] = ch;
}
```

The last line assigns the value of `ch`, the character being read in, to the stack array `s`, at the current `top` of the stack. Assuming the stack was empty initially, `stk -> top++;` increments top from -1 to 0; hence `stk->s[stk->top]` == `stk->s[0]`

7.1 ADTs – A Simple ADT Stack

- If we repeat the call to `push` a second time, the value of `top` is first incremented by one to point to index 1 of the array. Then, as before, the character input to `push` is added to this location.



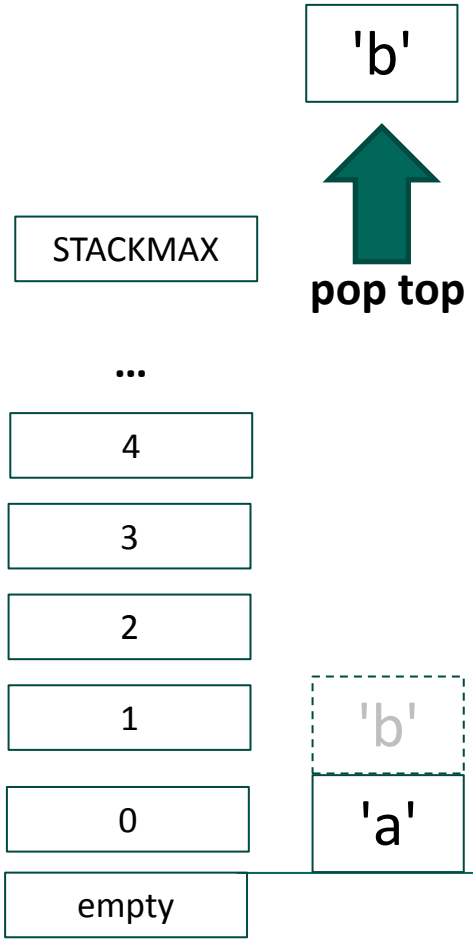

```
void push (char ch, stack *stk){
    stk -> top++; //inc from 0 to 1
    stk -> s[stk -> top] = ch;
}
```

After the first line, `stk->top` is set equal to 1, and so

```
stk->s[stk->top] == stk->s[1]
```

7.1 ADTs – A Simple ADT Stack

Similarly, we can pop the current value at the top of the stack off of the stack by returning the value currently pointed to and decrement top by 1.




```
char pop (stack *stk){
  return (stk -> s[stk -> top--]);
}
```

The last line returns the topmost value in `s`, and then decrements top, just as we'd expect

7.1 ADTs – A Simple ADT Stack

- We can use our stack like this:

```
#include <stdio.h>
#include "stack.h" // assume typedef listed earlier
int main(void){
    stack s;  //declare stack ADT, which includes array

    //string of chars to be pushed onto stack
    char str[] = "ADTs are such great fun in C!";

    reset(&s);// sets stack top to EMPTY
    // load the stack
    for (int i = 0; str[i] != '\0'; i++)   // load chars
        if (!isFull(&s)) push (str[i], &s);

    printf("The string\n %s\n backwards is \n",str);

    while (!isEmpty(&s)) // pop from top and print out
        printf("%c", pop(&s));
    printf("\n");
    return (0);
}
```

7.1 ADTs – A Simple ADT Stack

The output is:

```
The string
ADTs are such great fun in C!
backwards is
!C ni nuf taerg hcus era sTDA
```

Note that, while we've implemented this simple stack using an array of `char`s, we could instead use an array of structures in exactly the same fashion.

Alternately, we could declare an array of pointers to structures, and then `malloc()` new structures into existence, keeping track of them via a stack of pointers.

7.2 ADTs – A Linked List

- In our ADT stack example, we used a pre-defined array to hold our stack. In this example we create a linked list. Since each element in the list needs to be created as it is needed, we'll need to use an `_alloc()` function for each new list element.

7.2 ADTs – A Linked List

- When considering the components of an ADT, it's important to have a clear picture in mind of how the pieces fit together. A few questions help to focus the design of the code:

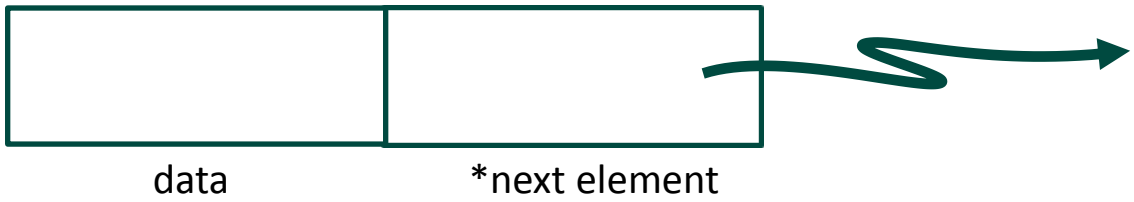
Q. What does a linked list do?

A. It connects, head-to-tail, a series of elements, like this:

7.2 ADTs – A Linked List

Q. What is the minimum amount of information that each element needs to store?

A. Some data, and a pointer to the address of the next element

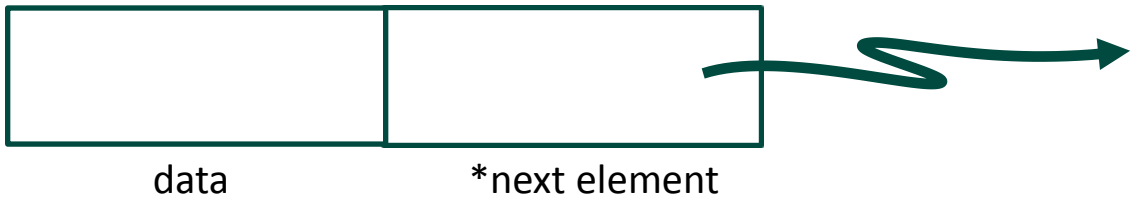


7.2 ADTs – A Linked List

Q. Given the function of the element, what will the structure look like (assuming character data)?

A.

```
struct linked_list{
  char data;      // use char for data stored
  struct link_list* next; // points to next
}
```



7.2 ADTs – A Linked List

Q. How will we connect one element to the next?

A. If we declare three such elements and initialize them, like this:

```
struct linked_list a, b, c;
a.data = b.data = c.data = 0;
a.next = b.next = c.next = NULL;
```

then we would expect to tie the elements together using:

```
a.next = &b;
b.Next = &c;
```

7.2 ADTs – A Linked List

Q. What operators will we need to make this data structure work?

A. Operators to:

a) Create a linked list based on an array of input values
b) Count the number of elements in the linked list
c) Insert an element
d) Delete an element
e) etc.

7.2 ADTs – A Linked List

- We start with a header file, `LList.h`, in which we place the following:

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

struct linked_list{
   char    data;     // the data actually stored
   struct linked_list* next; // points to next list
};
```

This is a bit wordy; we can shorten it using the following `typedef`:

```
typedef struct linked_list element;
```

7.2 ADTs – A Linked List

- So with this `typedef`

```
typedef struct linked_list element;
```

the word **`element`** becomes our linked list 'type'.

Since each `element` contains, as a member, a pointer to the next `element`, it is convenient to `typedef` a pointer to `element` as well.

```
typedef element* pElement;
```

As usual with `typedef`, we haven't done anything new here, just packaged the data in a way that helps clarify its usage.

7.2 ADTs – A Linked List

- The finished produce, inside `LList.h`, is:

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

struct linked_list{
   char    data;     // the data actually stored
   struct  linked_list* next; // points to next list
};

typedef struct linked_list element;
typedef element* pElement;
```

7.2 ADTs – A Linked List

- Note that this is the same as if we wrote

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

typedef struct linked_list{
   char    data;     // the data actually stored
   pElement next; // points to next list
} element;

typedef element* pElement;
```

except that this declaration triggers an error: **pElement** can only be defined after **element** has been declared, but the **pElement** type is a member of **element**, and so **element** cannot be resolved until **pElement** is resolved. Hence the need for the somewhat circular declaration shown on the previous slide—a consequence of C's single pass compiler.

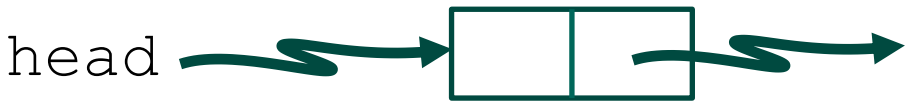
7.2 ADTs – A Linked List

- A linked list needs to start *someplace*. We'll need a 'handle' to the first element, which is just a pointer to the space in RAM where the code for the first element lives.

  We can then use `pElement` to create a 'handle' to the start of our list:

  ```
  pElement head;
  ```

  This says the `head` is a pointer to an element, an element which consists of a structure containing both data and a pointer to the next element. In other words:



  Currently, we aren't actually pointing to anywhere in memory. So we really should set initialize our pointer to NULL:

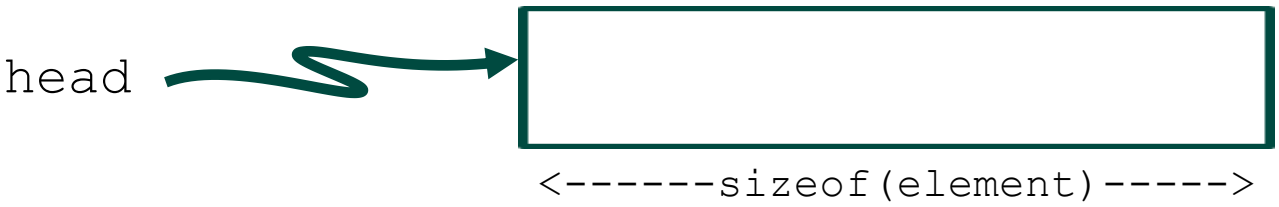
  ```
  pElement head = NULL;
  ```

7.2 ADTs – A Linked List

- To initialize `head` to an appropriately-sized location in memory, we set:

```
head = (pElement) malloc(sizeof(element));
```

`head` is a pointer that now contains the address of a block of memory (5 bytes in size, one char + one 4-byte pointer)



Thus far, we have a pointer to a structure, and memory allocated to store that structure. But the space being pointed to is empty so far.

7.2 ADTs – A Linked List

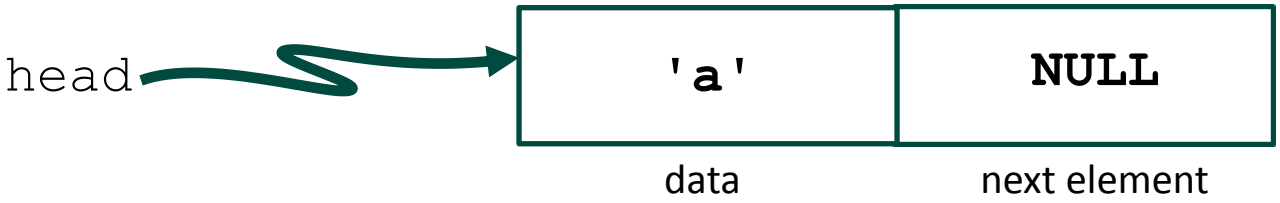
- We can initialize the value of data in the structure *that* `head` *points to* using the `->` notation:

```
head -> data = 'a';
```

And since our first structure doesn't point to anything yet, we should set the next pointer to NULL:

```
head -> next = NULL;
```

So we now have a linked list consisting of one element that looks like this:



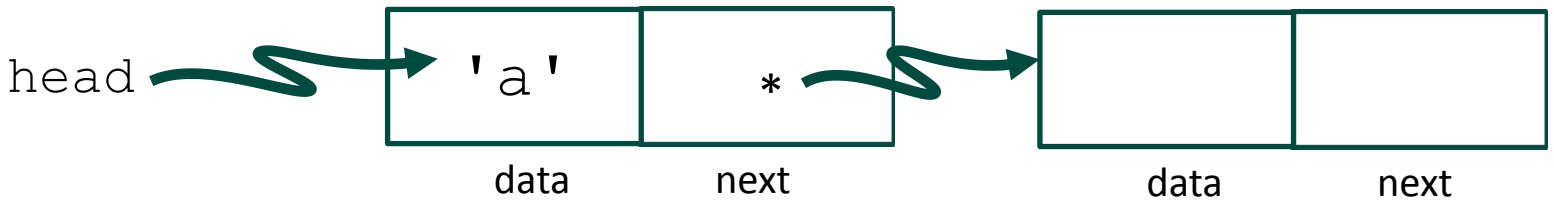
7.2 ADTs – A Linked List

- Following the example of the first element, each time we wish to create a new element to add to the list, we need to do the following:

  1. Allocate memory for the `next` element using:

```
pCurElement->next =(pElement) malloc(sizeof(element));
```

Where `pCurElement` is just a `pElement` type variable to temporarily hold the current element of the linked list. Note that the above command puts the address being pointed to into `next` at the same time.



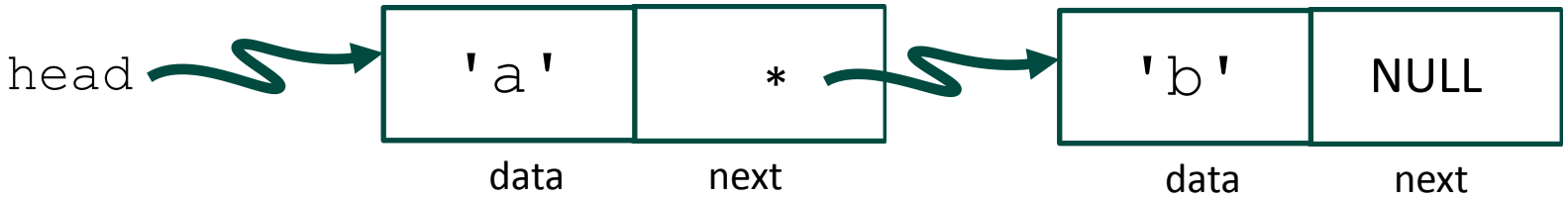
7.2 ADTs – A Linked List

2. At the place in memory pointed to, set the value of the actual data

```
pCurElement -> next -> data = 'b';
```

3. Set the address of the next pointer to NULL;

```
pCurElement -> next -> next = NULL;
```



7.2 ADTs – A Linked List

- Finally, since we'll be putting all this into a function, we'd like to be able to return the address of the new element itself:

```
return(pCurElement -> next);
```

What is the type of `next`? It is a pointer to an `element` structure, i.e. `pElement`. Hence our function should have the form:

```
pElement createNewElement (
        pElement pCurElement, char c){

  //...do something with pCurElement, etc.

  return(pCurElement -> next);
}
```

7.2 ADTs – A Linked List

Returning to our 'ToDo' list:

a) Create a list based on an array of input values, i.e. what we just covered
b) Count the number of elements in the list
c) Insert an element
d) Delete an element

7.2 ADTs – A Linked List

a) Create a list based on an array of input values. Assume an array of chars `char s[] = "abc.."` exists and we wish to link `a->b->c` etc.

```
pElement createNewElement (
            pElement pCurElement, char c){

   pCurElement->next=(pElement)malloc(sizeof(element));
   pCurElement->next->data = c;
   pCurElement->next->next = NULL;
   return(pCurElement -> next);
}
```

To call this function, use:

```
pElement link=(pElement) malloc (sizeof(element));
while (s[i] != '\0'){
   link = createNewElement(link, s[i++]);
}
```

7.2 ADTs – A Linked List

b) Count the number of elements in the list

This can be done recursively:

```
int count (pElement nextElement){
    if (nextElement == NULL)
        return(0);
    else
        return(1 + count(nextElement->next));
}
```

7.2 ADTs – A Linked List

Or using the ternary if:

```
int count (pElement nextElement){
    return ((nextElement == NULL)?
                0: (1 + count(nextElement->next)));
}
```

Rather than an iterative function, you could use a `for` loop:

```
int count (pElement nextElement) {
    int count = 0;
    for ( ; nextElement != NULL ;
                nextElement = nextElement -> next)
     count++;
    return count;
}
```

7.2 ADTs – A Linked List

c) Insert an element (insert element q between p1 and p2)

```
void insert (pElement p1, pElement p2, pElement q){
   assert(p1->next == p2); //ensure p1, p2 adjacent
   p1 -> next = q;  // set q to next after p1
   q -> next = p2;  // set p2 after q
}
```

7.2 ADTs – A Linked List

d) Delete elements

```
void deleteList(pElement link){
   if (link != NULL){
      deleteList(link->next); // go out to last
      free(link);  // release each element as
                   // you exit the recursion
   }
}
```

Note that this function deletes every element from the current `link` onward; the element *prior* to the deleted one is left pointing to freed memory, and needs to be `NULL`ed if a seg fault is to be which is impossible with a singly-linked list.

7.3 File/Device I/O – C's Input/Output Model

- C's I/O model follows the UNIX model for *file I/O.* Note that, ***in UNIX, everything is a file, even a device***. So in C, *all I/O is treated as file I/O*.
- Therefore, this model does not distinguish between different physical devices, which could include hard drives, tape drives, keyboards, terminals, monitors, DVD, network connections, pipes, FIFOs, USB sticks etc—since every device can be treated as a 'source' or 'sink' of information, just like a file.

See: https://www.usenix.org/legacy/events/bsdcon/full_papers/kamp/kamp_html/ for a detailed analysis of the problems associated in treating all devices as files.

7.3 File/Device I/O – C's Input/Output Model

- C allows two mechanisms for dealing with file I/O: ***file descriptors*** and ***streams***.
  - A *file descriptor* is just an integer that acts as a handle to a device driver, which is a specialized piece of software designed to handle the actual device I/O. File descriptors offer primitive, low-level, device-specific I/O operations. To talk to a device, you pass the file descriptor to the appropriate function for reading from, or writing to, that device. Because they are hardware-specific, file descriptors are less portable than streams.
  - A *stream* provides a high-level interface to a device. This is the most commonly-used method for file I/O, and the one we'll use throughout this section. As we'll see, each stream contains a file descriptor within its structure. Thus a stream is a high-level construct that *wraps* the low-level descriptor, making it easier to deal with. In this way, a stream allows you to treat most devices as if they were the same, regardless of the actual underlying hardware.

7.4 File I/O – Streams

- a stream is best conceptualized as a connection between your program and a physical device, whether input or output. It is an ADT in the form of a linear ***queue*** (plus support functions) that insert data in at one end, and ship it out at the other (i.e. like a FIFO).
- All file operations in C are abstracted as manipulations of *sequential* streams of data (hence the word, *stream*).
- C++ makes full use of streams. In C++, `cin` and `cout` are the rough equivalent of `scanf()` and `printf()`, and the >> and << symbols are used to indicate the 'flow' of the stream, e.g.

```
cout << "Enter your name" << endl;
cin >> Name
cout << "Hello, " << Name << endl;
```

7.4 File I/O – Streams

- Therefore, in C, a stream does *not* allow the user random access to any location inside the queue. You can push information on at one end and pop it off at the other, but you can't directly insert information into the middle of the stream.
- In practical terms, this means that in C you can't directly perform *random access* on data files temporarily stored in RAM while using a stream.
- Note that of the two general access types, *sequential access* is faster but limited in capabilities, while *random access* is slower and bulkier, but more powerful, since you can insert information directly into the structure at any point.
- Other languages, including C++ and Java, allow for random access.

7.4 File I/O – Streams

- Streams allow the programmer to handle all I/O through a uniform set of functions that can deal with data regardless of its source. Note that this is not unlike what happens when you use `Scanner` in Java. `Scanner` takes an input stream as its argument: it doesn't have to know anything about the underlying device. For example, you can write:

```
Scanner sysIn = new Scanner(System.in);
```

to handle input from the keyboard. Or you can write:

```
Scanner filRd = new Scanner (new FileReader("myFile"));
```

to handle file I/O. Scanner doesn't care about the hardware source; all it sees is a stream of bytes with a standard format.

7.4 File I/O – Streams

- Streams are closely related to the concept of buffering.
  - A ***buffer*** is a temporary storage space in RAM, used to collect input or output data before it is shipped to another device. Buffers are useful when the rate at which data is received or processed is different from the rate at which it can be retransmitted (and vice versa).
  - Buffers are used to assemble data when it arrives in pieces. A video buffer is a good example: the image needs to be fully constructed before it is displayed to the screen. File I/O is another example; information stored in RAM needs to be assembled completely before it is stored to disk, since multiple byte-wise moves to the drive are computationally costly. In some cases, you want instantaneous output—hence no buffer required—but in other cases it doesn't make sense to ship information until it's all assembled, and that's where buffers come in.

(Note that, like 'static', the term 'buffer' has a wide variety of meanings depending on its context; in C it is broadly employed to imply a space used for temporary storage.)

7.4 File I/O – Streams

- There are three types of buffering, depending on whether the programmer needs to ship information as quickly as possible (potentially slowing execution), or employing buffering to 'package' the information for shipment all at once:

  (1) *Unbuffered*—Minimum internal storage is used by `stdio` in an attempt to send or receive data as fast as possible.

  (2) *Line buffered*—Characters are processed on a line-by-line basis. This is commonly used in interactive environments, and internal buffers are flushed only when full or when a newline is processed.

  (3) *Fully buffered*—Internal buffers are only flushed when full.

Source: http://publications.gbdirect.co.uk/c_book/chapter9/input_and_output.html

7.4 File I/O – Streams

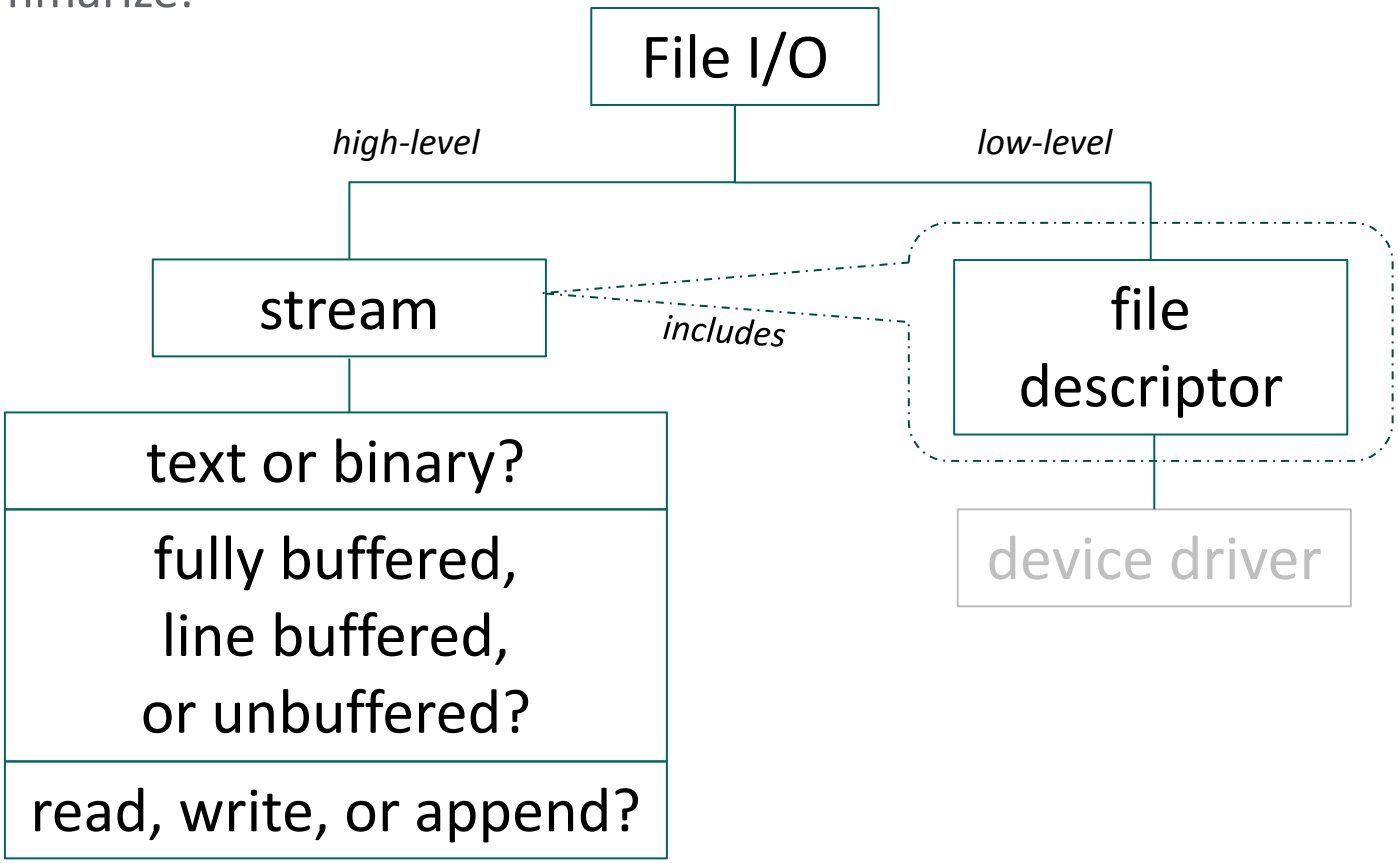
- C recognizes two file types: ***text files*** (which stores character data) and ***binary files*** (which stores byte data of different types, i.e. structures). These file types are manipulated, respectively, by *text streams* and *binary streams*. This distinction is a later addition to C; it does not exist in the UNIX environment: UNIX treats all streams as binary streams.

- Additionally, files may be opened for *read*, *write*, or *append*, where
  - *Read* – read data *from* the file buffer
  - *Write* – write data *to* the file buffer
  - *Append* – add data to the end of the buffer

  Note that the read/write terminology is from the 'system perspective', not the 'file perspective'. For example, writing a string *to* a file, from the system perspective, is the same thing as *reading* a string from a device (like the keyboard) from the file's perspective. It can be surprisingly easy to confuse reading from writing, unless you 'keep your sense of perspective.'

7.4 File I/O – Streams

- To summarize:



➡ This means that there are a huge number of functions available to handle all the different combinations of buffering, file type, and access

7.4 File I/O – Streams

- In C, a file I/O stream is represented by a structure called `FILE`, which is stored in RAM. This is declared in `stdio.h` as follows:

```
typedef struct{
    short level;
    unsigned flags;
    char fd;                        //file descriptor
    unsigned char hold;
    short bsize;                    //buffer size.
    unsigned char *buffer;          //start of  buffer
    unsigned char *curp;            //pointer into it
    unsigned istemp;
    short token;
} FILE;
```

In practice, we use a pointer to this structure, `FILE*`, as a handle to the actual contents of the file itself (more later).

7.4 File I/O – Streams

- Most of what is contained in the `FILE` structure isn't of particular interest to us. However, certain members deserve special attention:

```
char fd;  //char, but used as an int
```

  - This is the actual file descriptor, the low-level handle to the device itself.
  - This means that you can potentially use the (high-level) stream to access the file descriptor and perform low-level operations.

7.4 File I/O – Streams

```
unsigned char *buffer
```

is where the buffer itself is actually referenced by supporting functions (also located in `stdio.h`.) The actual size of the buffer is stored in the `FILE` structure member:

```
short bsize;
```

7.4 File I/O – Streams

- A ***file position indicator*** *(fpi)* is the offset from the start of the buffer. This is set to 0 when you create the file structure and increments as the buffer is read. The actual address being pointed to is represented in `FILE` by the member

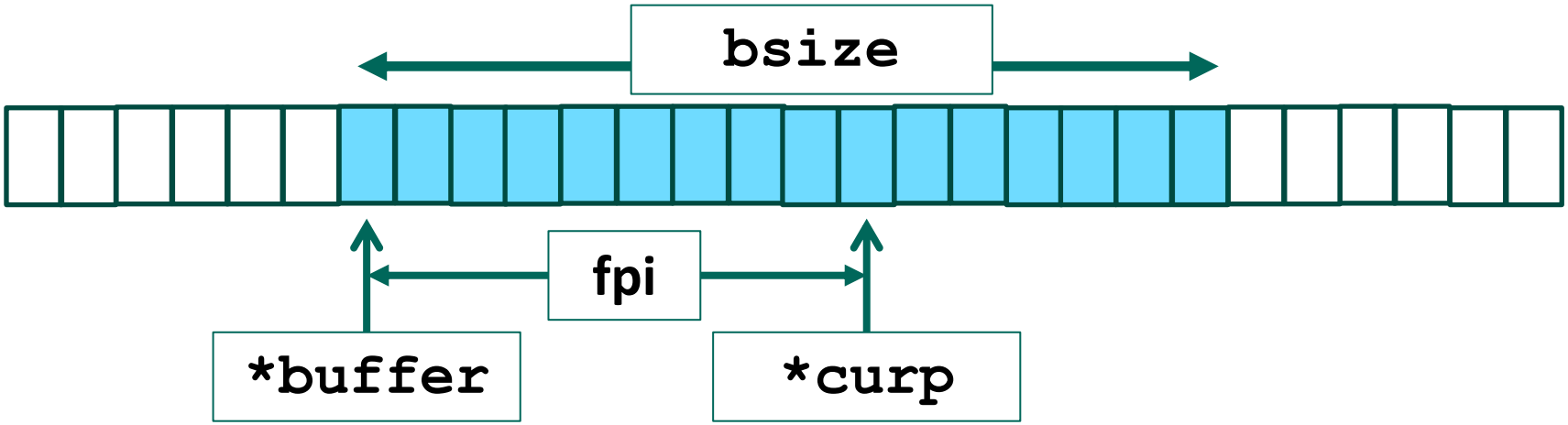
```
unsigned char *curp;
```

where `curp` stands for '*current active pointer*'—the physical address being pointed to in memory. So the relationship between `fpi`, the `buffer` start, and `curp` is thus:

```
curp = buffer + fpi
```

7.4 File I/O – Streams

- To summarize:



7.4 File I/O – Streams

- Additionally, `stdio.h` declares a useful set of constants, function, defines and macros. The following are of the most use to us:

Identifier	Stores
BUFSIZ	The size of the buffer used, which is typically greater than 256 bytes. This value can be overwritten by the `setbuf()` function.
EOF	A constant value that indicates that there is no more input left to read from the buffer. Note that `EOF = (const)(bsize==(curp-buffer+1))`
FILENAME_MAX	The maximum length a filename can have.
FOPEN_MAX	The minimum number of files that can be held open concurrently. Eight is the minimum number guaranteed by the system. However, this includes `stdin`, `stdout`, and `stderr` (discussed later).

7.4 File I/O – Streams – `fopen()`

- To open a file, use the `fopen()` function, which has the form:

```
FILE *fopen(const char *pathname, const char *mode)
```

where:

`pathname` points to a string that contains the path of the file, and

`mode` points to a string that specifies both the data access (read, write, or append) and the file type (binary or text), as discussed above.

The function returns a file stream, which can then be passed to any of the numerous file I/O functions found in `stdio.h`. When the function fails, `NULL` is returned.

7.4 File I/O – Streams – `fopen()`

- The second argument of `fopen()` is a pointer to one of the following strings shown below. The default type is the character type; to apply any of these operations to a binary file, append a "`b`" to the mode.

mode	Mean-ing	Explan-ation	If file exists	If file does not exist
"`r`"	read	Open file for reading	Read from start	Failure to open; return NULL
"`w`"	write	Open file for writing	Destroy contents	Create new
"`a`"	append	Append to a file	Write to end of file	Create new
"`r+`"	read extended	Open a file for read and write	Read from start	error
"`w+`"	write extended	Create a file for read and write	Destroy contents	Create new
"`a+`"	append extended	Open a file for read and write	Write to end of file	Create new

7.4 File I/O – Streams – fopen()

- So to read text from the existing file "path/filename.txt" you would use:

```
#include <stdio.h>

int main(void){
   FILE *inpfile;
   inpfile = fopen("/path/filename.txt", "r");

   //process inpfile here using stdio functions
}
```

- Note that, since a device is a file in UNIX/LINUX, you can 'talk' to a device using a command like:

```
pdev = fopen("/dev/prn", "w") //to OP to a printer
pcon = fopen("/dev/tty01", "r")//to IP from console
```

7.4 File I/O – Streams – `fopen()`

- To ensure that the file opened correctly, we should check for a `NULL` pointer, i.e.

```
if ((inpfile = fopen("/path/filename.txt", "r"))
                          == NULL)
    // File didn't open.  Bad path?.  Abort.
else
    // File opened.  Good to go.
```

Since `NULL` is just numerically equal to 0, an alternate shortcut version can be employed:

```
if (!(inpfile = fopen("/path/filename.txt", "r")))
    // File didn't open.  Bad path?.  Abort.
else
    // File opened.  Good to go.
```

7.4 File I/O – Streams – using `get()` functions

A small number of file I/O functions allow the programmer to add and retrieve information form files. To read characters or string in *from* a file use:

- **`int getc(FILE * stream)`** and
  `int fgetc(FILE * stream`)
  get the next single character from a file and advance the fpi. The character is returned by the function.

- **`char *gets(char *str, int n, FILE* stream)`** and
  `char *fgets(char *str, int n, FILE* stream)`
  reads in the next `n-1` characters from a file, or else stops when the newline character is found ('\n') or `EOF` is encountered. Note that a pointer to the string retrieved from the file can be passed as either **`str`**, or via the function return type.

7.4 File I/O – Streams – using `put()` functions

To output characters and strings *to* a file, a parallel systems exits using the **`put`** functions:

- **`int putc(int c, FILE* stream)`** and
  `int fputc(int c, FILE* stream`)
  puts the next single character **`c`** into a file and advances the fpi. The character input is returned by the function if successful, `EOF` otherwise.

- **`int puts(const char* str, FILE* stream)`** and
  `int fputs(const char* str, FILE* stream)`
  puts the `NUL`-terminated string pointed to by `str` into the file and returns a non-negative value if successful, `EOF` otherwise

7.4 File I/O – Streams – binary files

- Yet another system of functions exist for reading in a structure from a *binary* file. Binary file access differs from character file access in two fundamental ways:
  - character file access is sequential; this makes it fast, but you have to step your way through the file to get to what you're looking for. With binary files, you can access any structure within the file instantly.
  - You can change the contents of a structure within a file. Note that, because we are using streams, this does not mean you can insert a new structure into a file

So internally, a binary file looks *exactly* like an array of structures, except that it is not loaded into RAM, but onto disk.

7.4 File I/O – Streams – binary files

- Binary file I/O uses three basic operations: `read`, `write`, and `seek`. `read` corresponds to `get`, `write` to `put`, and `seek` has no equivalent, since character access is always sequential (you can't seek anything, just get and put to whereever `curp` is pointing to). Binary access has a small number of functions available for use; only the read operation is given here as an example.

  - **`size_t fread(void* sp, size_t size, size_t numStructs, FILE *fp);`**

    where **`sp`** is the address of the structure that receives the information read from the file, **`size`** is just the `sizeof` the structure pointed to, and **`numStructs`** is the number of structures to be read. The `size_t` value returned is the number of structures actually read.

  For additional information on the remaining functions see, for example, http://www.codingunit.com/c-tutorial-binary-file-io

7.4 File I/O – Streams – `fclose()`

- To close a file and save its contents back to disk (assuming it was written to or appended), use `fclose()`, which has the format

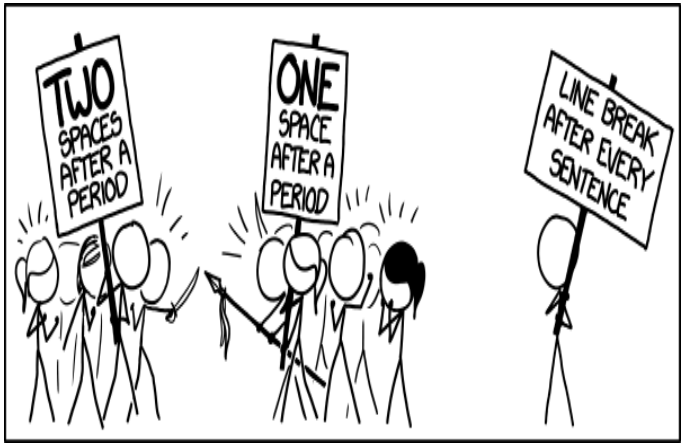
```
int fclose(FILE *stream);
```

where:

`stream` is the pointer to the `FILE` structure used to open the file

The integer returned is a 0 if the file closed properly, and `EOF` otherwise.

Example 17 : The Double Space File Function



Program Description:

This program opens one file, reads in each string until it finds a '\n', inserts an extra '\n', and outputs the results to another file. Hence each string in the output file is double-space relative to the strings in the input file.

Example 17 : The Double Space File Function

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

void double_space(FILE *, FILE *);   // ftn prototypes
void prnt_usage(char *);             // See Example 15

int main(int argc, char **argv){
   FILE *inp, *outp;
   if (argc != 3){      // if incorrect # of parameters
      prnt_usage(argv[0]); //…remind user of usage.
      exit(1);              // Exit and try again.
   }

   // else we have input and output file names…
```

An Example

```
    …

    inp = fopen(argv[1], "r");  // open 1st file for read
    outp = fopen(argv[2], "w"); // open 2nd file for write

    double_space(inp, outp);    // where all the real
                                // work gets done
    fclose(inp);                // tidy up
    fclose(outp);               // this saves buffer
                                // to file
}
```

An Example

The **`double_space()`** function reads from the first file, looks for `'\n'`s, and when it finds one, inserts a second copy into the output.

```
void double_space(FILE *I, FILE *O){
   int c;     // temp storage for the current character

   while ((c = getc(I)) != EOF){
      putc(c, O);
      if (c== '\n')
         putc('\n', O);
   }
}
```

From: A Book on C, A. Kelley, I. Pohl, Addison Wesley, 1998, pg. 507

7.5 stdin, stdout, and stderr

`stdio.h` automatically opens three streams each time a program runs, which are referred to as the ***standard streams***:

- `stdin`, `stdout`, and `stderr` are predefined streams of type (`FILE*`).
- These three structures are created automatically every time you execute a program: they are always there, whether you see them or not, and whether you use them or not. They are deallocated automatically each time you exit your program.

7.5 stdin, stdout, and stderr

- The standard input device, `stdin`, is set, by default, to your keyboard. In other words, `stdin` is a pointer to a structure of type `FILE`, and that structure contains (as we've just seen) both a buffer, which stores the current input, and a file descriptor, which handles the low-level I/O to the keyboard. Anything typed at the keyboard gets stored in this structure. (Remember, a device is a file, and a file's data can be stored in RAM as a `FILE` structure, whose address, in this case, is stored in the variable named `stdin`.)

7.5 stdin, stdout, and stderr

- The standard output device, `stdout`, is set to your monitor. Thus, anything that gets stored to the `stdout` stream gets transferred to the monitor via low-level calls involving that `FILE`'s file descriptor (`fd`) value.
- `stderr` is a stream designed to hold error messages. By default, it outputs to the monitor, but it can be overridden to output to, say, a file. (In fact, all three of these standard I/O types can be overridden to point to some other device.)

7.6 The `scanf()` family – `scanf()` 'bug'

- The best solution to solving the `scanf()` bug is to avoid using it for single character entry. There are a variety of solutions, of which one is to use `getc(stdin)`.

- `stdio.h` contains a special version of `getc()` designed to overcome the `scanf()` bug:

  `int getchar()` – reads in a single character from the keyboard. This function lacks `scanf()`'s problems with '\n'. The value returned is the character currently pointed to in the buffer, or `EOF` if we are at the end of the file. The pointer increments automatically, and ignores the '\n' that cause `scanf()` users such problems.

  `int getchar()` is identical to `int getc(stdin);`

see: http://en.wikibooks.org/wiki/C_Programming/File_IO

7.6 The `scanf()` family

- `scanf()` is one member of a family of input functions found in `stdio.h`. The '`f`' in `scanf()` stands for 'formatted'—`scanf()` 'expects' input string information to be properly formatted (and frequently fails when it isn't). Recall that `scanf`() has the form:

```
int scanf(const char *format, ...);
```

where the `format` is a string like "`%d`" or "`%c`". (Note that, as with all string literals, the value passed to format is the address of the first character in the string itself—a pointer to a `%` in both cases.)

7.6 The `scanf()` family

- In addition to `scanf()` itself, there are two other main functions in this family: `fscanf()` and `sscanf()`

```
int fscanf(FILE *stream, const char *format,...);
```

- Here the input is a file stream (the leading 'f' stands for 'file'). As with `scanf()`, you can specify a format of the data in the file stream.

```
int sscanf(const char *s, const char *format,...);
```

- In this case the input is a string (with a leading 's' in front), rather than a stream.

7.6 The `scanf()` family

- Here's an example of `sscanf()` in action

```
#include <stdio.h>
#include <stdlib.h>
int main() {
   int day, year;
   char weekday[20], month[20], dtm[100];
   strcpy(dtm, "Saturday March 25 1989" );
   sscanf(dtm, "%s%s%d%d", weekday, month,
       &day, &year);
   printf("%s %d, %d = %s\n", month, day, year,
       weekday);
   return(0);
}
```

The output is: `March 25, 1989 = Saturday`

Example from: http://www.tutorialspoint.com/c_standard_library/c_function_sscanf.htm

7.6 The scanf() family

Question: if

```
int fscanf(FILE *stream, const char *format,…);
```

gets its information from a stream, and

```
int sscanf(const char *s, const char *format,…);
```

gets its information from a string, then where does

```
int scanf(const char *format, ...);
```

get its information from?

7.6 The `scanf()` family – `scanf()` 'bug'

- Because `scanf()` expects its data to be well-formatted, it is highly susceptible to any non-standard formatting in the input stream found in `stdin`. And since this input comes from the user, it is typically the least formatted input, and therefore the most problematic.

  Recall the earlier example with `sscanf()`, and consider how `scanf()` actually works. `scanf()`

  (1) compares the contents in `stdin` with the format specifier found in the first parameter of the function (e.g. `"%d"`)

  (2) It reads the contents of `stdin` until it finds something that doesn't fit the pattern...and if there isn't another matching specifier or character in the format specifier, **`scanf()` skips over the remainder of the buffer**.

7.6 The `scanf()` family – `scanf()` 'bug'

In other words, `scanf()` is designed to work only until it fails to find a matching specifier in the `stdin` buffer. In gcc, `scanf()` ignores any remaining characters in the buffer after the point of failure.

7.6 The `scanf()` family – `scanf()` 'bug'

This rule has can have odd effects on your input. If your code says

```
printf("Enter a character");
scanf("%c", &myChar);
```

(where `myChar` is a declared as a char) and you enter

```
3<CR>
```

at the keyboard, the code reads `"3"` into your memory…but leaves the `'\n'` behind, since the program only looks for a single character each time. Thus '\n' becomes stuck in the buffer. If you simply enter another CR to get rid of the first, you've only added an additional '\n' to the buffer.

Note however that

```
scanf("%d", &myInt)
```

works, since "%d" does not treat '\n' as a formatting character. So the fact you want to treat the input as character rather than an number has odd results.

7.6 The `scanf()` family – `scanf()` 'bug'

If you call two `scanf()`'s one after the other

```
printf("enter a character");
scanf("%c", &myChar);
printf("enter another character");
scanf("%c", &myChar);
```

and enter

```
3<CR>
```

the first `scanf()` will 'eat' the 3, then the second `printf()` displays its line, and the second `scanf()` then 'eat's the `'\n'` still stored in the buffer. Your code will thus skip the second request for a character, since as far as `scanf()` is concerned, it has found a second character already in `stdin`—the `'\n'` associated with the '3' entered.

7.6 The `scanf()` family – `scanf()` 'bug'

But you can also get into trouble if you ask the user for a number, and then, using

```
scanf("%d", &myInt);
```

the user enters the word "three" instead of a value. Now `scanf()` skips over the characters—because you told it to expect a numerical value—and jumps straight to the next `printf()` statement, as before.

If there are more `scanf("%d", &myInt);` afterward, they are ignored: the buffer is full of characters that aren't recognized as integers, and so `scanf()` keeps looking for a number, and while it doesn't find one, ignores any requests for input.

7.6 The `scanf()` family – `scanf()` 'bug'

If instead you specify

```
scanf("%c\n", &myChar),
```

thinking that this will cause `scanf()` to look for a `'\n'`, then this too *will fail, and your input function will hang*.

Why? The `'\n'` part of the specifier string is treated as a *directive* to `scanf()`, an indication of what to look for, or in this case, what to ignore. Here, `'\n'` is interpreted to mean "ignore all white spaces in the string read from `stdin`". In practical terms, this means "ignore all white space characters—*along with the '\n' itself.*" So `scanf()` has no way to fulfill your complete request, since any subsequent carriage return gets ignored.

7.6 The `scanf()` family – `scanf()` 'bug'

So `scanf()` is problematic. It often works once in a program, since there's no problem reading the first character from the buffer, but not twice in a row, since the `'\n'` gets stuck in the buffer after the first read.

There are a number of prescribed solutions, none of which works perfectly:

(1) Use `getchar()` after `scanf()`. `getchar()` reads in a single character from the buffer, and so 'eats' the stuck `'\n'`.

(2) flush the buffer using `fflush(stdin)` after `scanf()`. This flushes all information from the buffer.

(3) Use special characters in `scanf()` to accommodate any unwanted characters after the desired input. For example `scanf(("%*[^\n]%*c, &a)` uses regular expressions ('regex') to specify processing of unwanted white space.

7.6 The `scanf()` family – `scanf()` 'bug'

(4) Use `sscanf("%s", &str)` to read *all* the input into a string (including the carriage return), and then parse the information stored in the variable `str`. This method will accommodate all possible input formats, but offloads responsibility for determining the format of the input to the user.

Notes

a. While `scanf()` is problematic and should generally be avoided, there are no similar exist with `fscanf()` and `sscanf()`. They work because they generally 'know' what kind of format to expect.

b. `Scanner` has similar problems (although it handles `'\n'` much better). When you write

```
input.nextInt
```

how do you know that the user will actually enter an integer?

7.6 The `scanf()` family – `scanf()` 'bug'

`scanf()` is flawed exactly because it tries to do too much, and because of the number of ways a user can enter input at a keyboard is unlimited. You can't anticipate every possible situation. (Note: Scanner() suffers the same problems.)

7.7 The EOF-'bug'

- `EOF` is declared as an integer, and this makes for problems when checking for the end of file while attempting to return a character. The following will generate an error, because it attempts to narrow-cast EOF to a single-byte `char`:

```
char c;
while ((c = getchar()) != EOF)
      putchar(c);
```

while the following will not:

```
int c;
while ((c = getchar()) != EOF)
      putchar(c);
```

Here, the character returned by `getchar()` is promoted to an `int`, thus EOF 'sees' the full four bytes of information, and not just 8 bits.

see: http://en.wikibooks.org/wiki/C_Programming/File_IO

7.8 Direct file manipulation

- Finally, note that while most of your manipulations on files are made via a file stream, `stdio.h` offers some standard function for operating on files *directly*. These include:

```
int remove (const char *filename)
```

which removes a file from the hard drive. A value of 0 returned indicates success, non-zero values indicate failure.

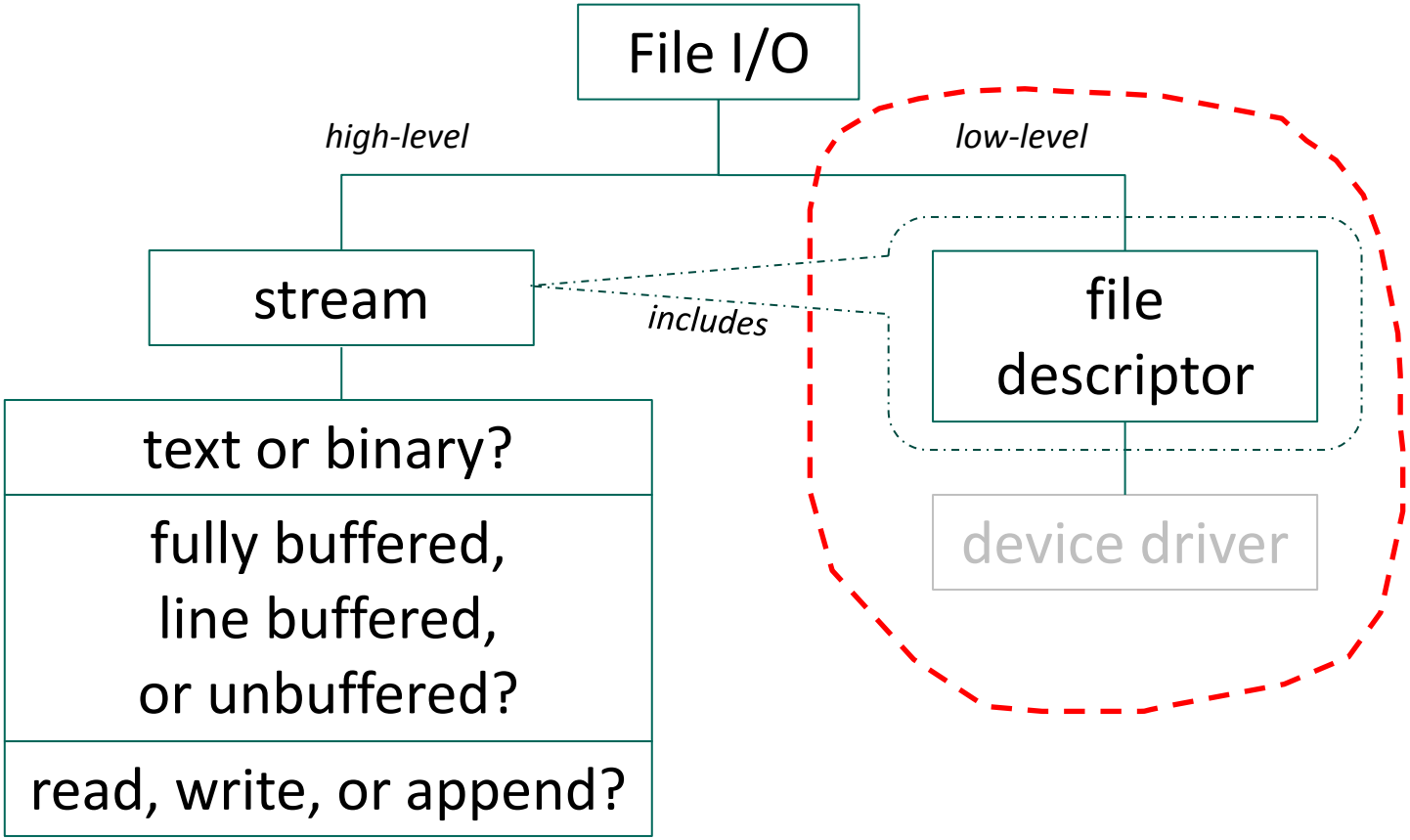
```
int rename(const char *old, const char *new);
```

renames the file from `old` to `new`. Again, 0 indicates success, non-zero failure.

7.9 File I/O – file descriptors

optional

- Recall that there are two mechanisms for handling file I/O: streams and file descriptors. We turn briefly to the latter of these two mechanisms.



7.9 File I/O – file descriptors

- While file streams are used most frequently, Linux/UNIX offer a standard set of six POSIX-compliant I/O system calls that allow you to operate on files using file descriptors directly, shown below

System Call	Function	Approx. C equiv.
`open()`	Open a file or device	`fopen()`
`close()`	Close a file or device	`fclose()`
`read()`	Read information from a file or device	`fread()`
`write()`	Write information to a file or device	`fwrite()`
`lseek()`	Move to a specific position in a file or device	`fseek()`
`ioctl()`	Input/Output control. Controls a device or the software used to access it	

Adapted from: *Internetworking with TCP/IP Vol 3: Client-Server Programming and Applications Linux/POSIX Sockets Version*, D. L. Comer and D.L. Stevens, Prentice-Hall, 2001.

7.9 File I/O – file descriptors

- The format of these functions differs from their 'C Language' equivalents. For example, the format for `open()` is not the same as `fopen()`.

```
int fd = open(char *fname, int access, int permis);
```

where:

`fd` is the file descriptor; set to -1 if the call fails;

`fname` is the name of the file or device;

`access` is the access mode; it uses special predefined constants such as `O_RDWR`, to signal, for example, that the file is to be opened for both read and write, or `O_TEXT` to indicate that the file is to be opened as a text file, rather than a binary;

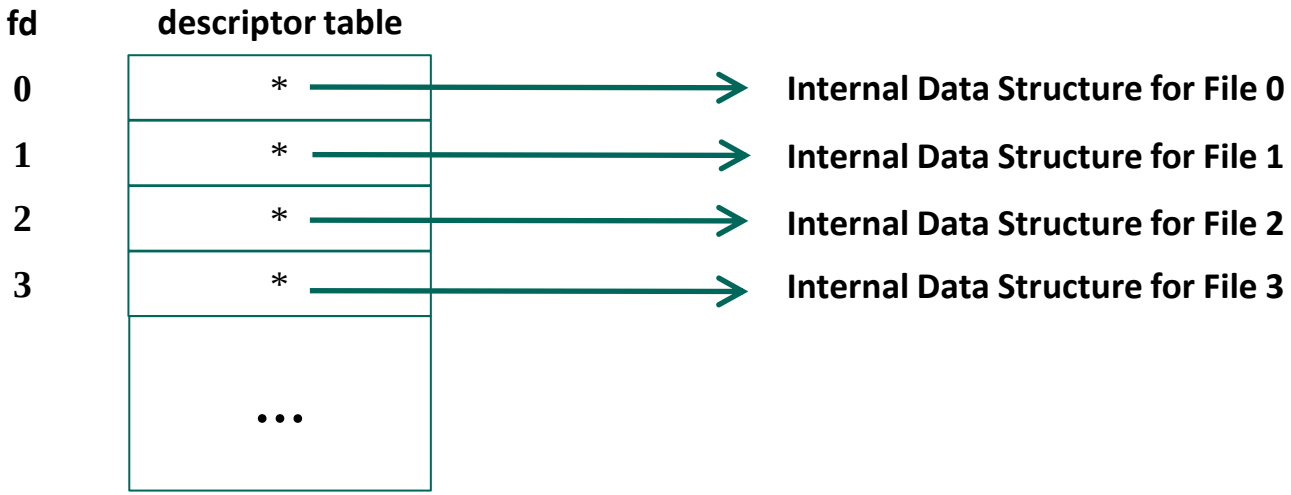
`permis` is used to set permissions; it is normally set to 0 for most applications.

See http://gd.tuwien.ac.at/languages/c/programming-bbrown/c_075.htm for the complete list of symbolic constants used in system-level file I/O

7.9 File I/O – file descriptors

optional

- The mechanics of the file descriptor are worth understanding, since the way in which file descriptors are allocated is exactly analogous to the way in which IP socket descriptors are allocated. A file descriptor is essentially an index to an array of pointers:



Each pointer in the descriptor table points to an internal file structure *maintained by the OS* (remember, this is `open()`, not `fopen()`)

Adapted from: *Internetworking with TCP/IP Vol 3: Client-Server Programming and Applications Linux/POSIX Sockets Version*, D. L. Comer and D.L. Stevens, Prentice-Hall, 2001

7.10 Miscellaneous

- Files are closed automatically when you call `fclose`, `exit`, or whenever you exit from `main()`. Any buffered data is flushed.
- When you work with `FILE*`, you are working on a copy of the file and NOT the file itself. To copy your information back to that file on the hard drive, you *must* save the data before you close the file. If the program shuts down before you have completed copying buffered data back to the hard drive, that information will not be saved.
- The word 'stream' is a bit misleading; think 'queue' whenever you hear it.

Another Example: from http://en.wikipedia.org/wiki/C_file_input/output

```
int main(void){
    char buffer[5] = {0};      /* initialized to zeroes */
    int i, rc;
    FILE *fp = fopen("myfile", "r");

    if (fp == NULL) {
        perror("Failed to open file \"myfile\"");
        return EXIT_FAILURE;
    }

   for (i = 0; i < 5; i++) {
        rc = getc(fp);
        if (rc == EOF) {
         fputs("An error occurred while reading the file", stderr);
            return EXIT_FAILURE;
        }
        buffer[i] = rc;
    }
    fclose(fp);
    printf("The bytes read were... %x %x %x %x %x\n", buffer[0],
           buffer[1], buffer[2], buffer[3], buffer[4]);
    return EXIT_SUCCESS;
}
```