Allocating memory for strings

```
/*
 * This doesn't solve our problem, but it does show how we use
 * malloc to allocate space for strings.
 */

char *string_duplicator(char *input) {
        char *copy;

        assert (input != NULL);
        copy = (char *)malloc(sizeof(char) * strlen(input) + 1);
        if (copy == NULL) {
                fprintf(stderr, "error in string_duplicator");
                exit(1);
        }

        strncpy(copy, input, strlen(input)+1);
        return copy;
}
```

we interrupt this broadcast...

- Consider this statement:
  - We must write our code to be flexible for as many situations as possible...
  - ... although this means we cannot make some assumptions about input sizes.
- Example:
  - For a file that processes text files, cannot make assumptions about the length of an input line
- Practical result:
  - Must (somehow) use malloc, realloc and possibly free appropriately

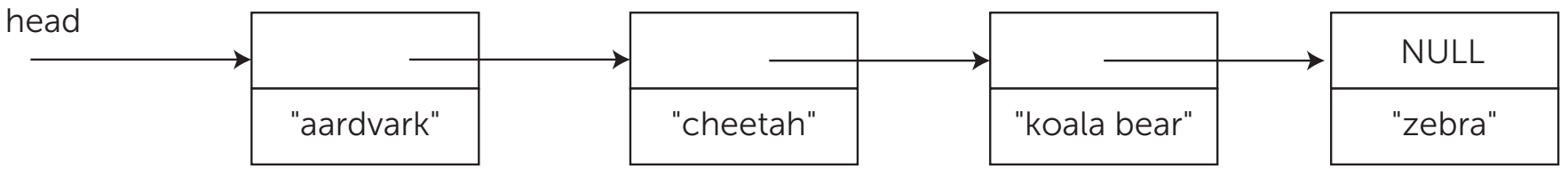
getline() solution

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

int main(void)
{
    FILE * fp;
    char * line = NULL;
    size_t len = 0;
    ssize_t read;

    fp = fopen("/etc/motd", "r");
    if (fp == NULL) {
        exit(1);
    }

    while ((read = getline(&line, &len, fp)) != -1) {
        printf("Retrieved line of length %zu :\n", read);
        printf("%s", line);
    }

    if (line) {
        free(line);
    }
    exit(0);
}
```

Lists

- While arrays are a convenient structure, they are not always the most suitable choice.
  - Arrays have a fixed size, yet a linked-list is exactly the size it needs to be to hold its contents.
  - Lists can be rearranged by changing a few pointers (which is cheaper than a block move like that performed by memmove in our implementation of delname)
  - When items are inserted or deleted from a list, the other items are not moved.
  - If we store pointers to the list elements in some other data structure, the list elements themselves won't be necessarily be invalidated by changes to the list.
- So:
  - If the set of items we want to maintain changes frequently…
  - … especially if the number of items is unpredictable…
  - then a list is the way to store them.

Singly-linked list



- Set of four items
  - Each item has data (in this case a string) along with a pointer to the next item.
  - Head of the list is a pointer to the first item
  - End of the list is denoted by a NULL pointer.
  - Handful of operations (add new item to front; find a specific items; add new item before or after a specific item; perhaps delete item)

Other languages

- Some languages have lists built into their core
  - Python does this
  - As does Lisp, Scheme, F#, etc.
- Other languages implement lists via a library
  - C++
  - Java
  - C#
- Most of these languages have a List type
  - However, the approach (or idiom) in C is to start with the element type.
  - That is, we are able to construct lists not via a list type but rather via a **node type**.

List node

- We'll revisit the same problem as described earlier (that of storing <name, value> pairs)
- The one addition to the Nameval struct is a "next" field
  - Its type is a pointer to the node type Nameval
  - This is the usual style in C of declaring types for self-referencing structures.
  - We'll see more recursive structures later…

```
typedef struct Nameval Nameval;
struct Nameval {
    char    *name;
    int      value;
    Nameval *next; /* in list */
};
```

Slight detour

- One of the tedious aspects of working with malloc is checking for success or failure
- We can accomplish this while still keeping our code clean by writing a small support function.
  - **emalloc**: a **wrapper function** that calls malloc; if allocation fails, it reports an error and exits the program.
  - Therefore we can use it as a memory allocator that never returns failure.

```
void *emalloc(size_t n)
{
    void *p;

    p = malloc(n);
    if (p == NULL) {
        fprintf(stderr, "malloc of %u bytes failed", n);
        exit(1);
    }
    return p;
}
```

Constructing an item

- Before "creating a list", let us write a function that constructs an item.
  - It will allocate memory from the heap…
  - … and then assign appropriate values to fields.
  - Note the use of "->" syntax
  - We assume here that some other function has allocated memory for the name

```
Nameval *newitem (char *name, int value)
{
    Nameval *newp;

    newp = (Nameval *) emalloc(sizeof(Nameval));
    newp->name  = name;
    newp->value = value;
    newp->next  = NULL;
    return newp;
}
```

Adding an item to the front

- This is the simplest way to assemble a list
  - Also the fastest.
- This function (and others we'll write) all return a pointer to the first element as their function value
  - Note that this even works if the list is empty (e.g., pointing to NULL)

```
Nameval *addfront(Nameval *listp, Nameval *newp)
{
    newp->next = listp;
    return newp;
}
```

```
/* typical usage */
Nameval *nvlist = NULL;
...
nvlist = addfront(nvlist,
      newitem(string_duplicator("zastre"), 5771));
```

Adding an item to the end

- With a singly-linked list this is an O(n) operation
  - Traverse list until we reach the last node
  - Adjust that node's pointer to indicate the new node.
  - Note that the next field of node created by newitem is already set to NULL.

```
Nameval *addend(Nameval *listp, Nameval *newp)
{
    Nameval *p;

    if (listp == NULL) {
        return newp;
    }
    for (p = listp; p->next != NULL; p = p->next)
        ;
    p->next = newp;
    return listp;
}
```

Find an item

- As with adding to the end, we have an operation that is O(n)
  - Unlike a sorted array, binary search does not work on list.
  - However, the code is uncomplicated and its main loop is similar to that in addend.
- The function returns the node even though it is searching on the name
  - If the function succeeds, the return value will be a memory location on the list which can be dereferenced.
  - Otherwise the return value is NULL (i.e., the lookup failed)
  - This is in keeping with the usual C idiom for success and failure of operations.

```
Nameval *lookup(Nameval *listp, char *name)
{
    for ( ; listp != NULL; listp = listp->next) {
        if (strcmp(name, listp->name) == 0) {
            return listp;
        }
    }
    return NULL;
}
```