Return to C

- Function pointers
- Separate modules in C
- Dynamic memory (malloc, realloc, calloc)
- Arrays that grow in C
- Linked lists in C
- Trees in C

Function pointers

- In your travels you will see code that looks a bit like the following:

  foo = (*fp)(x, y)

  – The function call is actually performed to whatever function is stored at the address in variable "fp"
- Strictly speaking:
  – A function is not a variable…
  – … yet we can assign the address of functions into pointers, pass them to functions, return them from functions, etc.
- A function name used as a reference without an argument is just the function's address
- Example: qsort's use of a function pointer

Function pointers (example)

```c
/* Code here is computing compare_ints very literally --
 * can actually produce the needed result in one arithmetic
 * operation (assuming no overflows, that is...). */

int compare_ints(const void *a, const void *b) {
    int value_a = *(int *)a;
    int value_b = *(int *)b;

    if (value_a < value_b) {
        return -1;
    } else if (value_a > value_b) {
        return 1;
    } else {
        return 0;
    }
}
```

```c
void some_function(int count) {
     int numbers[count];
     /* .... read values and store them into array "numbers"... */
     qsort(numbers, count, sizeof(int), compare_ints);
     /* ... values in "numbers" now in sorted order ... */
}
```

Abstract Data Types

- So far, we have described basic data types, all the standard C statements, operators and expressions, functions, and function prototypes.
- We want to introduce the concept of modularization
- Before there were object-oriented languages like Java and C++, users of imperative languages used **abstract data types** (ADT):
  - an abstract data type is a set of operations which access a collection of stored data
  - in Java and C++ this idea is called **encapsulation**

Abstract Data Types

- Since ANSI compilers support separate compilation of source modules, we can use abstract data types and function prototypes to **simulate modules**:
  - this is simply for convenience
  - a C compiler does not force us to use separate files
  - allows us to implement the “one declaration – one definition” rule

Abstract Data Types (2)

- For module `"mod"` there are two files:
  - **Interface module**: named `"mod.h"` contains function prototypes, public type definitions, constants, and when necessary declarations for global variables. Interface modules are also called header files.
    - Interface modules are accessed using the `#include` C preprocessor directive
  - **Implementation module**: named `"mod.c"` contains the implementation of functions declared in the interface module.

Example: module bitstring

- **Module** `bitstring`
  - Interface module: `bitstring.h` contains the declarations for data structures and operations required to support bitstring manipulation. Contains items (i.e., types, functions) which **must** be visible to the user of the module.
  - Implementation module: `bitstring.c` contains implementation of bitstring operations

Interface Module: bitstring.h

```
#ifndef BITSTRING_H
#define BITSTRING_H

typedef unsigned int Uint;
typedef enum _bool { false = 0, true = 1 } bool;

#define BITSPERBYTE     8
#define ALLOCSIZE       (sizeof(Uint)* BITSPERBYTE)
#define BYTESPERAUNIT   (sizeof(Uint))

/* -- Bit Operations */

extern void clear_bits( Uint[], Uint );
extern void set_bit( Uint[], Uint );
extern void reset_bit( Uint[], Uint );
extern bool test_bit( Uint[], Uint );

#endif
```

Implementation Module: bitstring.c

```
 #include "bitstring.h"

/* Clear a bit string */
void clear_bits( Uint bstr[], Uint naunits ) {
    Uint i;

    for ( i = 0; i < naunits; i++ )
        bstr[i] = 0;
}

/* Set a bit in a bit string */
void set_bit( Uint bstr[], Uint bit ) {
    Uint b_index = ( bit - 1 ) / ALLOCSIZE;
    Uint b_offset = ( bit - 1 ) % ALLOCSIZE;

    bstr[b_index] |= ( 1 << b_offset );
}
```

Implementation Module (2)

```
/* Reset a bit in a bit string */
void reset_bit( Uint bstr[], Uint bit ) {
    Uint b_index = ( bit - 1 ) / ALLOCSIZE;
    Uint b_offset = ( bit - 1 ) % ALLOCSIZE;

    bstr[b_index] &= (~( 1 << b_offset));
}

/* Determine the state of a bit in a bitstring */
bool
test_bit( Uint bstr[], Uint bit ) {
    Uint b_index = ( bit - 1 ) / ALLOCSIZE;
    Uint b_offset = ( bit - 1 ) % ALLOCSIZE;

    return( (bstr[b_index] & (1 << b_offset)) ? true : false );
}
```

Using the Bitstring Module

```
#include "bitstring.h"

#define NUNITS 4

int main( int argc, char *argv[] ) {
    Uint set[NUNITS];

    clear_bits(set,NUNITS);
    set_bit(set,8);
    set_bit(set,12);

    if (test_bit(set,12) == true)
        reset_it(set,12);

    return 0;
 }
```

```
$ gcc module_user.c bitstring.c -o module_user
```

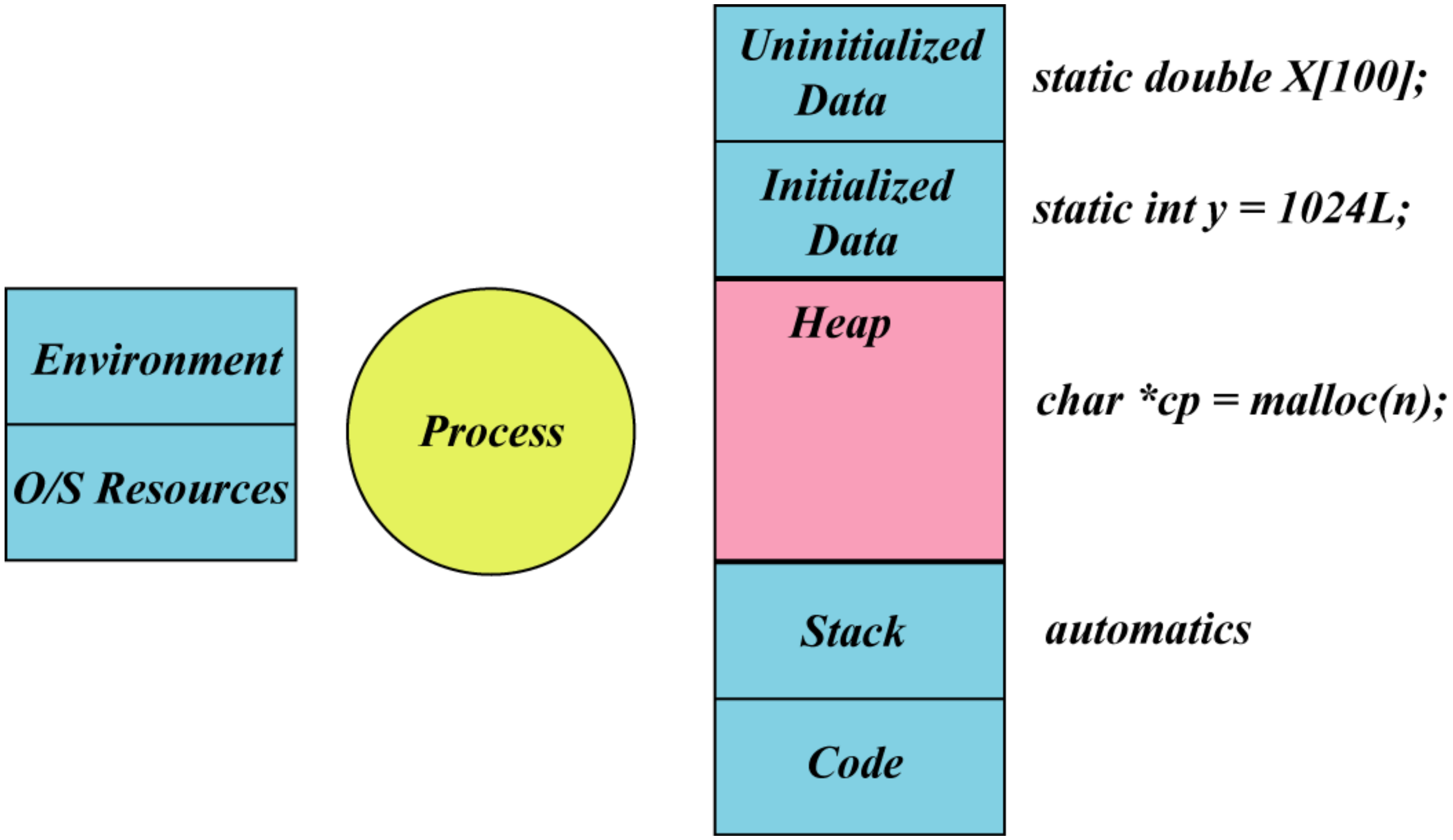
Dynamic memory in C

- The C memory model
- Managing the heap: `malloc()` and `free()`
- The void pointer and casting
- Memory leaks
- Applying the concepts:
  - **arrays that grow**
  - **linked lists**
  - **binary trees**
  - **hash maps**

C memory model

- Memory is divided into five segments: **uninitialized data**, **initialized data**, **heap**, **stack**, and **code**
- **Stack** is used to store **automatics** and activation records (stack frames) for functions
- Stack is located at the "bottom" of writable memory (low addresses)
- **Heap** stores explicitly requested memory which must be **dynamically allocated**
- Limits of heap and stack can bump into each other
- Initialized and uninitialized data located at at "top" of writable memory (high addresses)

C memory model



C memory model

- As the program executes, and function call-depth increases, the stack **grows upward**
- Similarly, as memory is explicitly requested for allocation, the heap **grows downward**
- Every time a function call is made, a new **stack frame** is created and memory allocated for variables to be used by the function
- Eventually, if stack and heap continue to grow, all memory available to the process will be exhausted and an **out of memory** condition will occur

Motivation for dynamic memory

- Memory must be allocated for storage before variables can be used, yet the amount might not be known at compile time
- Examples:
  - Reading records from a file in order to sort them, where file size is not known at time program is written
  - Constructing a list of "keywords" based on the content of some text file (whose size is unknown at compile time)
  - Representing a set as a linked list

Motivation for dynamic memory

- One solution: Write the program by hard-coding in the largest amount of memory that could possible be needed
- Problem with this approach: Possibly occupies large unused area of memory if program input sizes for program are almost always small
  - Note: virtual-memory systems mitigate the effects of this somewhat

Where to store what…

- Use **heap memory** for dynamic allocation when size is not known until run-time
- Use **stack memory** for parts where size is known at compile time
- Working with the stack is easy -- all variables are defined at compile time (no extra work for you)
- Working with the heap is a bit harder
  - In Java and Python, heap memory is automatically allocated to objects through use of the "new" keyword
  - Also in Java & Python, heap memory no longer used by a variable may be reclaimed for the system (**garbage collection**)

Heap memory in C

- Memory addresses in the heap are sometimes called **anonymous variables**
  - These variables do not have names like automatics and static variables
- `malloc()`: allocate dynamic memory
  - Takes a single parameter representing the **number of bytes of heap memory to be allocated**
  - **Returns a memory address** to the beginning of newly allocated block of memory
  - **If allocation fails**, `malloc()` returns NULL
- `#include <stdlib.h>` : contains function prototypes for malloc and related functions.

sizeof

- `sizeof()` is a macro that computes the number of bytes allocated for a specified type or variable (basic types, aggregate types)
- Use `sizeof()` to determine block size required by `malloc()`
- You must **always** check the value returned by `malloc()`!

```
int *a = malloc (sizeof(int));
if (a == NULL) { /* error */ }

struct datetype *dt = malloc (sizeof(struct datetype));
if (dt == NULL) { /* error */ }

char *buffer = malloc (sizeof(char) * 100);
if (buffer == NULL) { /* error */ }
```

malloc() + casting

- Function prototype for malloc() :
    - `extern void *malloc( size_t n );`
    - `typedef unsigned int size_t;`
- The pointer returned is a generic pointer
- To use the allocated memory, we must **typecast** the returned pointer
    - Denoted by (<sometype> *)
    - Casting is a hint to the compiler (applies different typechecking to the block of memory after the typecast)

```
double *f = (double *) malloc (sizeof(double));

char *buf = (char *) malloc(100);
```

Casting

- What if we do not typecast?

```
double *f = malloc(sizeof(double));
...
<from some compilers>
 warning: assignment makes pointer from
      double without a cast
```

- Always a good idea to cast
- Once heap memory is allocated:
  – It stays allocated for the duration of the program's execution…
  – ...unless it is **explicitly deallocated**.
  – All memory used in heap is returned to system when process/program terminates.

A family of functions

- There is more than just `malloc`:
  - `calloc`
  - `realloc`
  - `valloc`
- These serve slightly different purposes
  - One function both allocates and initializes the block of heap memory
  - One function adjusts heap structures to change size of a previously allocated block/chunk
  - Etc.
- As we need these extra functions we'll trot them out

free()

- We use `free()` to return heap memory no longer needed back to the heap pool
  - `extern void free( void * );`
- `free()` takes a pointer to the allocated block of memory

```
void very_polite_function( int n )
{
    int *array = (int *) malloc (sizeof(int) * n);

    /* Code using the array */

    free (array);
}
```

Issues with dynamic memory

- A **memory leak** occurs when heap memory is **constantly allocated** but is **not freed** when no longer needed
- Memory leaks are almost always unintentional
  - Allocation and deallocation code locations are often widely separated.
  - Can be hard to find the memory-leak bug as it often depends upon the program running for a long time.
- Systems with automatic garbage collection (almost) never have memory leaks
  - Redundant memory is returned to heap for re-use
  - Downside: garbage collection is not always under control of the programmer
  - Also: some garbage collectors cannot reclaim some kinds of redundant instances of data types.

Arrays that grow

- All of our C programs using arrays to date have been static in size
  - Assignment specifications state the largest input size.
  - Memory is allocated for these arrays from the C compiler and run time
  - We never need to manage this memory.
- Arrays are very handy structure
  - Easy to index and access (O(1) operations)
  - Contiguous block of memory can be exploited by other functions (qsort, memcpy).
- Therefore we would like to keep the convenience of arrays but also obtain the benefits of dynamic memory
  - … and do so without having to write more complex structures like lists, heaps, etc.

Nameval array

- Suppose we wish to maintain an array of <name, value> pairs
  - Name is a string
  - Value is an integer
- We want to add new items to our array as the arrive
- If there is not enough room in the array, we want to grow it.
- To support this we'll keep the array's size and items-to-date associated with the array via a struct.
  - Note use of "typedef"

```
typedef struct Nameval Nameval;
struct Nameval {
    char   *name;
    int     value;
};
```

```
struct Nvtab {
    int   nval;
    int   max;
    Nameval *nameval;
} nvtab;

enum { NVINIT = 1, NVGROW = 2 };
```

Creating a new nameval

```
Nameval *new_nameval(char *name, int value)
{
    Nameval *temp;

    temp = (Nameval *)malloc(sizeof(Nameval));
    if (temp == NULL) {
        fprintf(stderr, "Error mallocing a Nameval");
        exit(1);
    }

    /* temp->name === (*temp).name */
    temp->name = (char *)malloc((strlen(name)+1) * sizeof(char));
    if (temp->name == NULL) {
        fprintf(stderr, "Error mallocing a memory for string");
        exit(1);
    }
    strncpy(temp->name, name, strlen(name)+1);

    temp->value = value;

    return temp;
}
```

addname

```
int addname(Nameval newname)
{
    Nameval *nvp;

    if (nvtab.nameval == NULL) { /* first use of array */
        nvtab.nameval =
            (Nameval *) malloc(NVINIT * sizeof(Nameval));
        if (nvtab.nameval == NULL) { return -1; }
        nvtab.max = NVINIT;
        nvtab.nval = 0;
    } else if (nvtab.nval >= nvtab.max) {
        nvp = (Nameval *) realloc(nvtab.nameval,
            (NVGROW * nvtab.max) * sizeof(Nameval));
        if (nvp == NULL) { return = -1; }
        nvtab.max = NVGROW * nvtab.max;
        nvtab.nameval = nvp;
    }
    nvtab.nameval[nvtab.nval] = newname;
    return nvtab.nval++;
}
```

Deleting a name

- Arrays are contiguous…
  - Yet we may sometimes want to remove elements that are within the array
  - That is, neither at the start or end
- This can be tricky:
  - We need to decide what to do with the resulting gap in the array.
  - If element order doesn't matter: just swap last item in array with gap
  - If element order does matter (i.e., must be preserved), the we must move all the elements beyond the gap by one position

delname

```
int delname (char *name)
{
    int i;

    for (i = 0; i < nvtab.nval; i++) {
        if (strcmp(nvtab.nameval[i].name, name) == 0) {
            memmove(nvtab.nameval + i, nvtab.nameval + i + 1,
                (nvtab.nval-(i+1)) * sizeof(Nameval));
            nvtab.nval--;
            return 1;
        }
    }
    return 0;

    /* Note that no realloc is performed to resize the array.
     * Do you think this action is needed???
     */
}
```

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
  - Safe alternative: **getline()**

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);
}
```

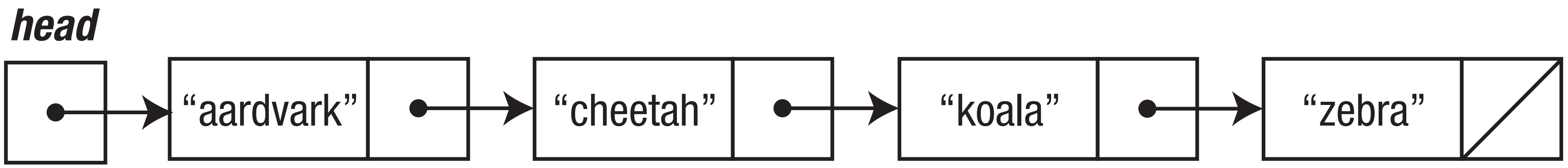
Lists

- While arrays are a convenient structure, they are not always the most suitable choice.
  - Arrays have a fixed size (albeit it may be resized with effort), 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 addresses to the **list elements** in some other data structure, the **list elements themselves** won't be necessarily be invalidated by changes to the **list**.

Lists

- 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.
- Typical usage of list for problem will dictate the kind of linked-list:
  - singly
  - singly, with head & tail pointers
  - doubly
  - circular
  - skip-list

Singly-linked list (no tail pointer)



- 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;  /* Is this exactly what we want??? */
    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;
...
Nameval *newnode = newitem(string_duplicator("Michael"), 50);
nvlist = addfront(nvlist, newnode);
```

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;
}
```

An observation about lists

- Many other operations on list have a similar structure
  - Traverse through the list...
  - ... and while doing so, compute some value / perform some comparison / etc.
  - After traversing the list, return some value
- One approach is to write many such functions with this structure.
- Another approach is to write a more general-purpose function…
  - which traverses through the list…
  - … and applies some function to each element in the list.
  - Let's call this function **apply**
  - It will take three arguments (the list; a function to be applied to each element on the list; and an argument for that function)

apply

```
/* apply: execute fn for each element of listp */

void apply(Nameval *listp, void (*fn)(Nameval*, void*), void *arg)
{
    for ( ; listp != NULL; listp = listp->next) {
        (*fn)(listp, arg);  /* call the function */
    }
}
```

```
void (*fn)(Nameval*, void*),

  Declare fn to be a pointer to a void-valued function
  (i.e., it is a variable that holds the address of a function
  that returns void).

  Such a function takes two arguments: an address to a Nameval
  (list element) and a void * (a generic point to an argument
  for the function being passed in).
```

example: printing out all elements

```
/* apply: execute fn for each element of listp */

void apply(Nameval *listp, void (*fn)(Nameval*, void*), void *arg)
{
    for ( ; listp != NULL; listp = listp->next) {
        (*fn)(listp, arg);  /* call the function */
    }
}
void printnv(Nameval *p, void *arg)
{
    char *fmt;

    fmt = (char *) arg;
    printf(fmt, p->name, p->value);
}
```

```
apply(nvlist, printnv, "%s: %x\n");
```

example: count of all elements

```
void inccounter(Nameval *p, void *arg)
{
    int *ip;

    /* p is not used -- all we care about is that this function
     * is called once per node.
     */
    ip = (int *)arg;
    (*ip)++;   /* Note the parentheses!!! */
}
```

```
int n;

n = 0;
apply(nvlist, inccounter, &n);
printf("%d elements in nvlist\n", n);
```

Deleting elements from the list

- We have yet to see the use of `free` in the management of our lists
- Let's take the simplest case first: deleting the whole list
  - Here we must be rather careful
  - We cannot free an element if we need to dereference that same element later.
  - Also: `free` may itself modify the newly deallocated memory
- Must make good use of temporary variables

freeing the list

```
void bad_freeall(Nameval *listp)
{
    for ( ; listp != NULL; listp = listp->next ) {
        /* What is the value of listp->next after the next
         * operation?
         */
        free(listp);
    }
}
```

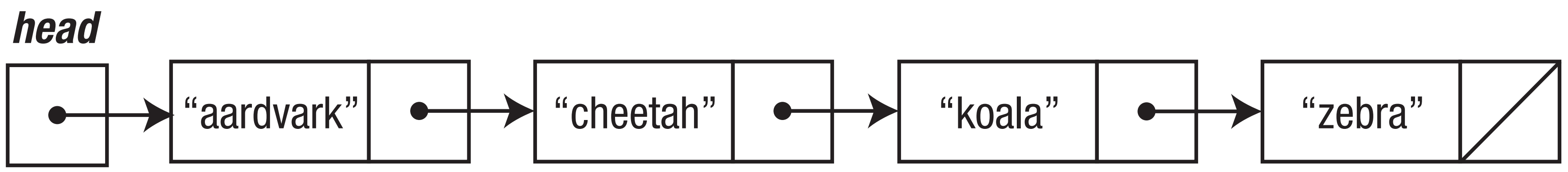
```
void freeall(Nameval *listp)
{
    Nameval *next;

    for ( ; listp != NULL; listp = next ) {
        next = listp->next;
        /* assume here the listp->name is freed someplace else */
        free(listp);
    }
}
```

Deleting elements from the list

- Deleting a single element requires more work than adding an element
  - Part of this is due to the consequences of using a singly-linked list.
  - It would be much easier with a doubly-linked list—but then again, such a list does require twice as many pointers to be maintained.
- This is the place where bugs are often introduced
  - Yet if we are careful—and correctly diagram what we intend to do—then we can get it right the first time.
  - Recall the two main cases: are we deleting the first element? or one past the first

Recall our list example…



Deleting a single element

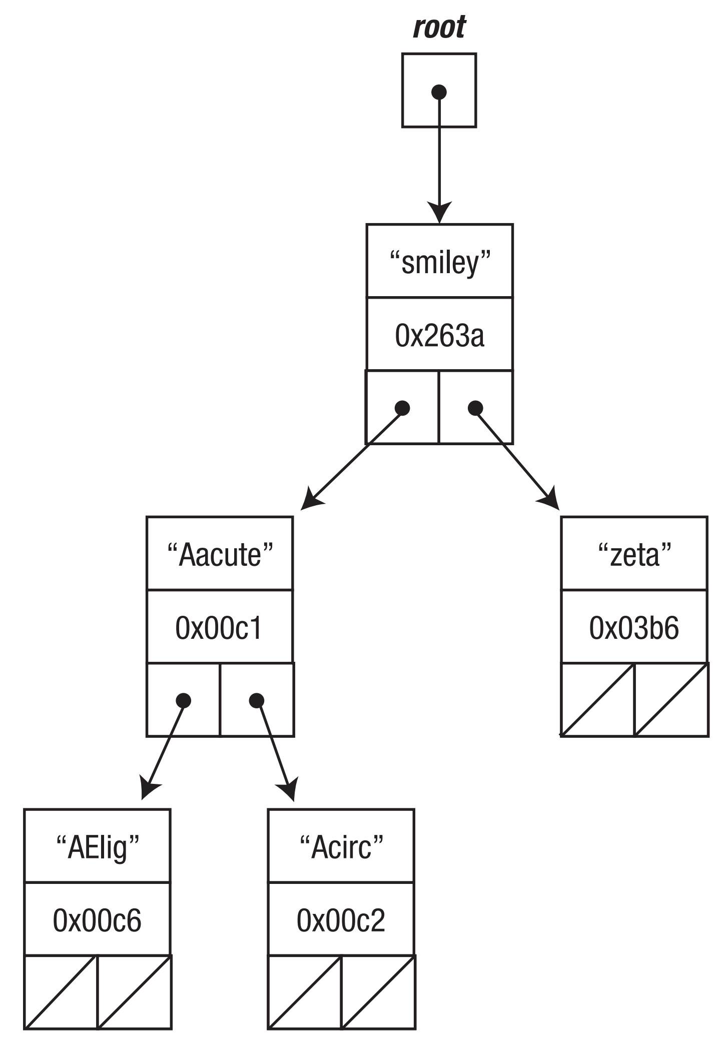
```
Nameval *delitem (Nameval *listp, char *name)
{
    Nameval *curr, *prev;

    prev = NULL;
    for (curr = listp; curr != NULL; curr = curr-> next) {
        if (strcmp(name, curr->name) == 0) {
            if (prev == NULL) {
                listp = curr->next;
            } else {
                prev->next = curr->next;
            }
            free(curr);
            return listp;
        }
        prev = curr;
    }
    /* Ungraceful error handling, but gets the point across. */
    fprintf(stderr, "delitem: %s not in list", name);
    exit(1);
}
```

Trees

- Hierarchical data structure
  - We use them implicitly when navigating through the Unix file system
  - Also used in compilers (e.g., parse trees)
  - We also construct trees programmatically to get O(lg n) behavior rather than O(n) for our algorithms (after some assumptions)
- **Binary search tree**
  - Simpler tree flavour, straightforward to implement
  - Node in a binary search tree has a **value** and two pointers, **left** and **right**
  - The pointers lead to the node's children
  - All children to the left of a particular node have lower values than the node.
  - All children to the right of a particular node have greater values than the node.

Binary search tree example



Tree node

- We'll again re-use the same problem as shown earlier (that of storing <name, value> pairs)
- We now need links to the left and right subtree
- Note that the "lesser" and "greater" comments are used to help the programmer
  - Must still ensure the semantics of our operations follow the meaning of the comments!
- Code to create such a node is left as an exercise.

```
typedef struct Nameval Nameval;
struct Nameval {
    char    *name;
    int      value;
    Nameval *left;  /* lesser */
    Nameval *right; /* greater */
};
```

Construction

- Constructing a tree means descending into the tree recursively.
  - At insertion time, each new node ends up as a leaf node.
  - As other items are inserted later, a leaf node above a new leaf node will become the new node's parent.
- The algorithm must choose the left or right branch until the right place to link is found
- As with the linked-list routines, the insertion algorithm returns the root of the tree as the result.

Inserting node into tree

```
/* Assume newp has been already initialized. */
Nameval *insert(Nameval *treep, Nameval *newp)
{
    int cmp;

    if (treep == NULL) {
        return newp;
    }
    cmp = strcmp(newp->name, treep->name);
    if (cmp == 0) {
        fprintf(stderr, "insert: ignoring duplicate entry %s\n",
            newp->name);
    } else if (cmp < 0) {
        treep->left = insert(treep->left, newp);
    } else {
        treep->right = insert(treep->right, newp);
    }
    return treep;
}
```

Some observations

- The tree routines just shown do not permit duplicate entries.
- The insertion routine does not try to keep the tree balanced
  - It is possible that a sequence of inserts could yield a linear list instead of a tree (i.e., inserting a sequence of items that are already sorted).
  - However, this means our routines are a lot simpler (although it is not an oppressive amount of work to implement an AVL tree; rather, it is just a bit complicated!)
- The code for a lookup is similar to that for insertion
  - Recursively search by choosing the left or right subtrees
  - Return the correct node if matching lookup criteria, NULL otherwise.

lookup

```
Nameval *lookup(Nameval *treep, char *name)
{
    int cmp;

    if (treep == NULL) {
        return NULL;
    }
    cmp = strcmp(name, treep->name);
    if (cmp == 0) {
        return treep;
    } else if (cmp < 0) {
        return lookup(treep->left, name);
    } else {
        return lookup(treep->right, name);
    }
}
```

Must be recursive?

- Both insert and lookup were recursive
  - The routines were defined in terms of themselves.
  - Base case: empty tree
  - Inductive step: left tree, then right tree
- However, not all recursive routines need be recursive
  - **Tail recursion**: when the recursive step (i.e., invocation of the recursive function) is the last step of the function
  - We can transform tail-recursive functions into iterative ones
  - All we require is some patching up of arguments (via assignments) and need a way to restart the body of the routine (via some loop)

Non-recursive lookup

```
Nameval *lookup(Nameval *treep,
    char *name)
{
    int cmp;

    if (treep == NULL) {
        return NULL;
    }
    cmp = strcmp(name, treep->name);
    if (cmp == 0) {
        return treep;
    } else if (cmp < 0) {
        return lookup(treep->left,
             name);
    } else {
        return lookup(treep->right,
             name);
    }
}
```

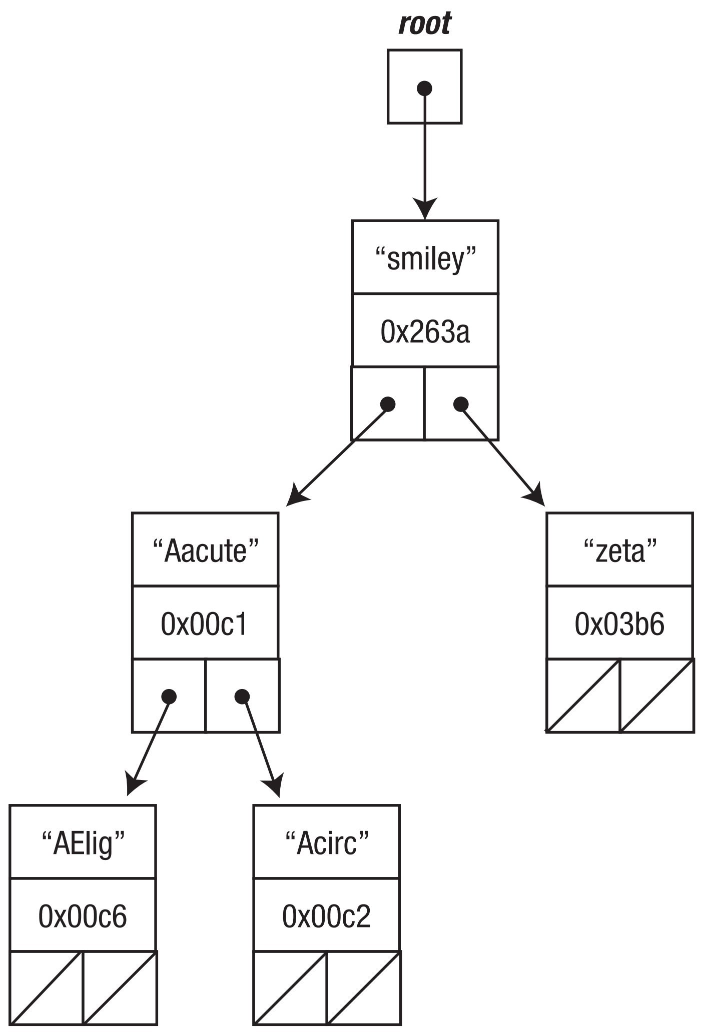
```
Nameval *nrlookup(Nameval *treep,
    char *name)
{
    int cmp;

    while (treep != NULL) {
        cmp = strcmp(name, treep->name);
        if (cmp == 0) {
            return treep;
        } else if (cmp < 0) {
            treep = treep -> left;
        } else {
            treep = treep->right;
        }
    }
    return NULL;
}
```

An observation about trees

- The same observations we made about operations on lists can also be made with respect to operations on trees
  - Traverse through the tree in some order
  - While doing so, compute some value / perform some comparison / etc.
  - After traversing the tree, return some value
- If we want to rewrite **apply** for a binary search tree, we must decide on some order
  - **inorder** traversal?
  - **pre-order** traversal?
  - **post-order** traversal?
- In effect we will have one **apply** function for each ordering, and each of these functions will take arguments similar to what we had for the list version of **apply**.

Binary search tree example



```
inorder:
    AElig
    Aacute
    Acirc
    smiley
    zeta

post-order:
    AElig
    Acirc
    Aacute
    aeta
    smiley

pre-order:
    ???
```

applyinorder

```
void applyinorder(Nameval *treep,
    void (*fn)(Nameval*, void*), void *arg)
{
    if (treep == NULL) {
        return;
    }
    applyinorder(treep->left, fn, arg);
    (*fn)(treep, arg);
    applyinorder(treep->right, fn arg);
}
```

```
/* We can even use some of the functions we passed as arguments
 * to the "list" version of apply!
 */

applyinorder(treep, printnv, "%s: %x\n");
```

```
/* Could you build a sort based on the tree routines +
 * a function (that you would write) given to applyinorder?
 */
```

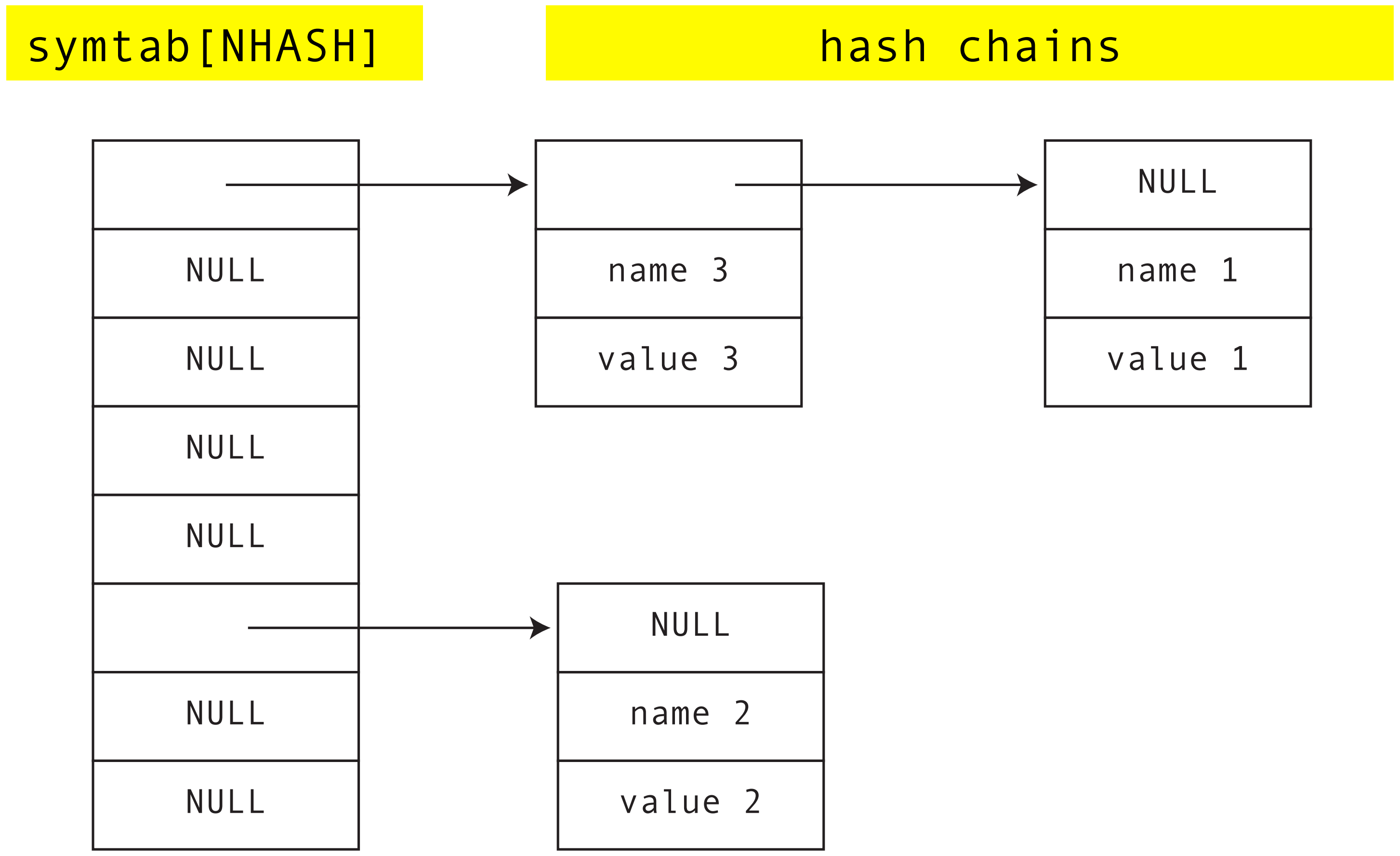
Hash tables

- These combine:
  - arrays
  - lists
  - some mathematics
- Efficient structure for storing and retrieving dynamic data
- Typical application for hash tables: symbol tables
  - Associates some value (the **data**)...
  - with each member of a dynamic set of strings (the **keys**)
- Lots of places where hash tables are used

The idea

- Hash tables work on the following principle
  - Pass a key to the **hash function**
  - The hash function produces a **hash value**
  - These values will be **evenly distributed** through a modest-sized integer range
- The hash value is then used as an array index
  - In C the usual style is to associate each hash value / array index with a list of items that share the hash
  - Each such list (sometimes called a **hash chain**) is known as a **bucket**

Example


symtab[NHASH]
hash chains
NULL
NULL
NULL
NULL
NULL
NULL
name 3
value 3
NULL
name 1
value 1
NULL
name 2
value 2

The practice

- Hash functions are pre-defined
- Array is sized appropriately (usually at compile time)
- Each element of the array is a list that **chains** together the items that share a hash value (i.e., hash chain)
- Equivalently:
  - A hash table of n items ...
  - ... is an array of lists whose average length is (n/array size)
- Retrieving an item is an O(1) operation provided the following two conditions hold:
  - we pick a good hash function
  - the lists do not grow too long

Element type

- A hash table is an array of lists...
  - ... therefore we can re-use the element type used for lists
- Maintaining individual hash chains is similar to maintaining individual lists
- Once we have a good hash function, the code falls out easily
  - just pick the hash bucket...
  - ... and walk along the list looking for a perfect match

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

/* symbol table */
Nameval *symtab[NHASH];
```

Lookup / insertion routine

- If item is found:
  - It is returned
- If item is not found and create flag is set:
  - add item to the table.

```
Nameval *lookup (char *name, int create, int value)
{
    int h
    Nameval *sym;

    h = hash(name);
    for (sym = symtab[h]; sym != NULL; sym = sym->next) {
        if (strcmp(name, sym->name) == 0) { return sym; }
    if (create) {
        sym = (Nameval *) emalloc(sizeof(Nameval));
        sym->name = name; /* assumed allocated elsewhere */
        sym->value = value;
        sym->next = symtab[h];
        symtab[h] = sym;
    }
    return sym;
}
```

Why combine lookup & insertion?

- This is a common combination
- Without it we often duplicate effort
- (Without it the hash function to be executed twice for the same item.)
- This is a stylistic point (but one which can save a bit of tedium and reduce possibility of buggy code)

```
/*
 * The code the might result if we
 * keep lookup and insertion separate.
 */

if (lookup("name") == NULL) {
    additem(newitem("name", value));
}
```

Two more questions

- How big should the array be?
  - In general: make it large enough that each hash chain will have at most a few elements
  - Example: A compiler might have an array size of a few thousand
- How is the hash function computed?
  - Must be deterministic (i.e., produce same value each time for same key)
  - Must be fast
  - Must distribute data uniformly through the array
  - Lots of research exists that investigates these properties of hash functions

Possible hash function

- Common hashing algorithm for strings:
  - Build a value by adding each byte of the string to a multiple of the hash so far
  - Multiplication spreads bits from the new byte through the value so far
- Empirically: the values 31 and 37 have proven to be good choices for ASCII strings

```
#define MULTIPLIER 31

/* hash: compute hash value of string */
unsigned int hash (char *str) {
    unsigned int h;
    unsigned char *p;

    h = 0;
    for (p = (unsigned char *) str; *p != '\0'; p++) {
        h = MULTIPLIER * h + *p;
    }
    return h % NHASH;
}
```

Summary

- malloc() is an important tool
  - but it can be tricky at first to use correctly
  - is usually paired with free()
- dynamically-allocated memory needed for implementing many kinds of data structures
- two big takeaways
  - arrays can be resized (and that's handy as arrays are easy to use)
  - lists are used in many data structures (so it is important to know how to write routines that add to, traverse through, and remove from lists)

Colophon

- Some code examples are from "The Practice of Programming" (Addison-Wesley) © 1999 Brian W. Kernighan and Rob Pike