Chapter 19
Data Structures

Based on slides © McGraw-Hill
Additional material © 2004/2005/2006 Lewis/Martin

Data Structures

Data structure: particular organization of data in memory

- Group related items together
- Organize so that convenient to program and efficient to execute

Example

- An array is one kind of data structure

We will look at. . .

`struct` – C's mechanism for building data structures
linked list – built from `struct`

Structures in C

The C `struct`

- Mechanism for grouping related data of different types
- (Array elements must be of same type)

Example

- Suppose we want to keep track of weather data for the past 100 days, and for each day, we want the following data

```
int highTemp;
int lowTemp;
double precip;
double windSpeed;
int windDirection;
```

We can use a `struct` to group these data

Declaring a `struct`

We name the `struct` and declare "fields" (or "members")

```
struct w_type {
  int highTemp;
  int lowTemp;
  double precip;
  double windSpeed;
  int windDirection;
};
```

This is declaration so no memory is actually allocated!

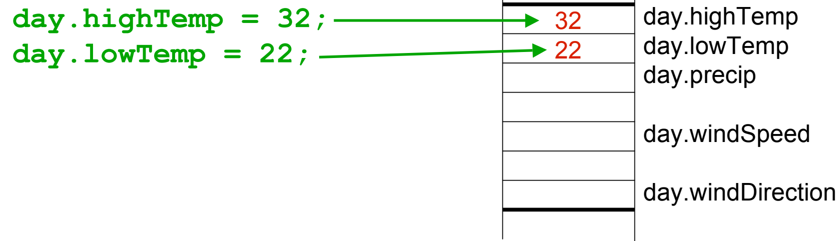
Defining and Using a `struct`

We define a variable using our new data type

```
struct w_type day;
```

Memory is allocated (on stack), and we can access individual fields of this variable

```
day.highTemp = 32;
day.lowTemp = 22;
```



`Struct` declaration determines layout in memory

Declaring and Defining at Once

You can both declare and define a `struct` at same time

```
struct w_type {
     int highTemp;
     int lowTemp;
     double precip;
     double windSpeed;
     int windDirection;
} day;
```

Can still use the `w_type` name to declare other `structs`

```
struct w_type day2;
```


`typedef`

C provides a way to define new data type names

Syntax

```
typedef <type> <name>;
```

Examples

```
typedef int Color;
typedef struct w_type WeatherData;
typedef struct ab_type {
  int a;
  double b;
} ABGroup;
```


Using `typedef`

Benefit

- **Code more readable because we have application-specific types**

```
Color pixels[128*124];
WeatherData day1, day2;
```

Common practice

- **Put `typedef`'s into a header file**
- **Use type names in program**
- **If the definition of `Color`/`WeatherData` changes, *might* not need to change the code in rest of program**

LC-3 Code for Structs

Consider the following code

```
. . .
int x;
WeatherData day;
int y;

day.highTemp = 12;
day.lowTemp = 1;
day.windDirection = 3;
...
```

offset		
offset = 3		x
4		day.highTemp
5		day.lowTemp
6		day.precip
7		
8		day.windSpeed
9		
10		day.windDirection
11		y

Code for Structs

```
; day.highTemp = 12;
    AND R1, R1, #0   ; R1 = 12
    ADD R1, R1, #12
    ADD R0, R6, #4   ; R0 has base addr of day
    ADD R0, R0, #0   ; add offset to highTemp
    STR R1, R0, #0   ; store value into day.highTemp
; day.lowTemp = 1;
    AND R1, R1, #0   ; R1 = 1
    ADD R1, R1, #1
    ADD R0, R6, #4   ; R0 has base addr of day
    ADD R0, R0, #1   ; add offset to lowTemp
    STR R1, R0, #0   ; store value into day.lowTemp
; day.windDirection = 3;
    AND R1, R1, #0   ; R1 = 3
    ADD R1, R1, #3
    ADD R0, R6, #4   ; R0 has base addr of day
    ADD R0, R0, #6   ; add offset to windDirection
    STR R1, R0, #0   ; store value into day.windDirection
```


Array of Structs

Can define an array of `structs`

```
WeatherData days[100];
```

Each array element is a `struct`

- 7 words, in this case

To access member of particular element

select field

```
days[34].highTemp = 97;
```

select element

Pointers to Struct

We can define and create a pointer to a `struct`

```
WeatherData *dayPtr;  // define ptr not WeatherData
dayPtr = &days[34];
```

To access a member of the `struct` addressed by `dayPtr`

```
(*dayPtr).highTemp = 97;
```

. operator has higher precedence than *, so this is NOT the same as

```
*dayPtr.highTemp = 97;      *(dayPtr.highTemp) = 97;
```

Special syntax for this common access pattern

```
dayPtr->highTemp = 97;
```

Passing Structs as Arguments

Unlike an array, `structs` passed by value

- `struct` members copied to function's activation record, and changes inside function are not reflected in the calling routine's copy

Most of the time, you'll want to pass a pointer to a `struct`

```
int InputDay(WeatherData *day)
{
  printf("High temp in deg F: ");
  scanf("%d", &day->highTemp);
  printf("Low temp in deg F: ");
  scanf("%d", &day->lowTemp);
  ...
}
```


Dynamic Allocation

Problem

- What if we don't know the number of days for our weather program?
- Can't allocate array, because don't know maximum number of days that might be required
- Even if we do know the maximum number, it might be wasteful to allocate that much memory because most of the time only a few days' worth of data is needed

Solution

- Allocate storage dynamically, as needed

`malloc`

C Library function for allocating memory at run-time

```
void *malloc(int numBytes);
```

Returns

- Generic pointer (`void*`) to contiguous region of memory of requested size (in bytes)

Bytes are allocated from memory region called heap

- Heap != stack
- Run-time system keeps track of chunks of memory in heap that have been allocated

Using `malloc`

Problem

- Data type sizes are implementation specific

Solution

- Use `sizeof` operator to get size of particular type

```
WeatherData *days;
days = malloc(n * sizeof(WeatherData));
```

Also need to change type of return value to proper kind of pointer

- Called "casting"

```
days = (WeatherData*)
         malloc(n* sizeof(WeatherData));
```

Example

```
int numberOfDays;
WeatherData *days;

printf("How many days of weather?");
scanf("%d", &numberOfDays);

days = (WeatherData*) malloc(sizeof(WeatherData)
                      * numberOfDays);
if (days == NULL) {
  printf("Error in allocating the data array.\n");
  ...
}
days[0].highTemp = ...
```

If allocation fails, `malloc` returns NULL (zero)

Note: Can use array notation or pointer notation

`free`

When program is done with `malloc`ed data

- It must/should be released for reuse
- Achieved via `free` function
- `free` is passed same pointer returned by `malloc`
  `void free(void*);`
- Other pointers (*e.g.,* to the stack) may not be `free`d

If allocated data is not freed

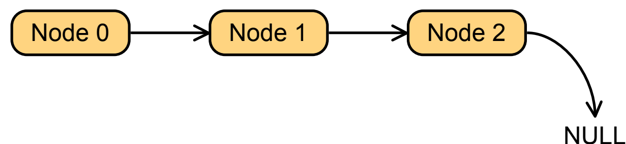
- Program may run out of heap memory and be unable to continue

Explicit memory management versus garbage collection

Example: The Linked List Data Structure

Linked list

- Ordered collection of nodes
- Each of which contains some data
- Connected using pointers
  - Each node contains address of next node in list
  - Last node points to NULL
- First node in list is called head
- Last node in list is called tail



Linked List vs. Array

Advantage of linked list

- Dynamic size (easy to add or remove nodes)

Advantage of array

- Easy/fast element access

Element access

- Linked list elements can only be accessed sequentially
  e.g., to find the 5th element, you must start from head and follow the links through four other nodes

Example: Car Lot

Goal

- Create inventory database for used car lot
- Support the following operations
  - Search database for a particular vehicle
  - Add new vehicle to database
  - Delete vehicle from database

Implementation

- Since we don't know how many cars might be on lot at one time, we choose a linked list representation
- In order to have "faster" search, the database must remain sorted by vehicle ID

CarNode

Each vehicle has the following characteristics

- Vehicle ID, make, model, year, mileage, cost
- (And… pointer to next node)

```
typedef struct car_node CarNode;

struct car_node {
  int vehicleID;
  char make[20];
  char model[20];
  int year;
  int mileage;
  double cost;
  CarNode *next; /* ptr to next car in list */
}
```


Scanning the List

Scanning

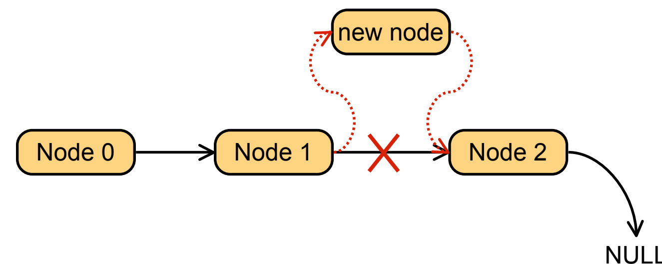
- Searching, adding, and deleting require us to find a particular node in the list
- Scan list until we find node whose ID is >= one we're looking for

```
CarNode *scan_list(CarNode *head, int searchID)
{
  CarNode *previous, *current;
  previous = head;
  current = head->next;
  /* Traverse until ID >= searchID */
  while ((current != NULL) &&
         (current->vehicleID < searchID)) {
    previous = current;
    current = current->next;
  }
  return previous;
}
```


Adding a Node

Steps

- Create new node with proper info (via `malloc`)
- Find node (if any) with a greater `vehicleID` (via `scan_list`)
- "Splice" the new node into the list (update `next` fields)



Excerpts from Code to Add a Node

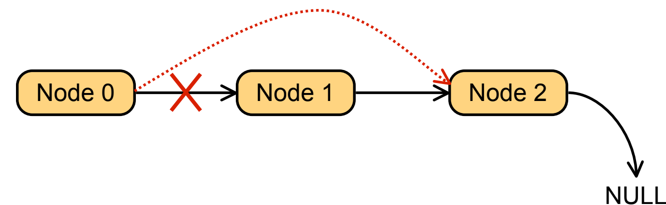
```
newNode = (CarNode*) malloc(sizeof(CarNode));
/* initialize node with new car info */
...
prevNode = scan_list(head, newNode->vehicleID);
nextNode = prevNode->next;

if ((nextNode != NULL) &&
    (nextNode->vehicleID == newNode->vehicleID))
  prevNode->next = newNode;
  newNode->next = nextNode;
} else {
  printf("Car already exists in database.");
  free(newNode);
}
```


Deleting a Node

Steps

- Find the node that points to the desired node (via `scan_list`)
- Redirect that node's pointer to the next node (or NULL)
- Free the deleted node's memory



Excerpts from Code to Delete a Node

```
printf("Enter vehicle ID of car to delete:\n");
scanf("%d", &vehicleID);

prevNode = scan_list(head, vehicleID);
delNode = prevNode->next;

if ((delNode != NULL) &&
    (delNode->vehicleID == vehicleID))
  prevNode->next = delNode->next;
  free(delNode);
}
else {
  printf("Vehicle not found in database.\n");
}
```


Beyond Linked Lists

The linked list is a fundamental data structure

- Dynamic
- Easy to add and delete nodes

Other data structures

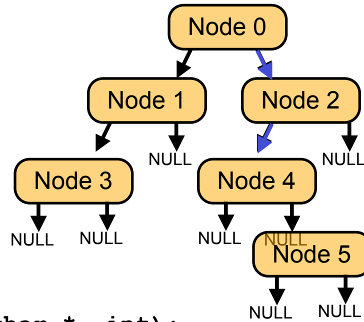
- Leverage same techniques as linked lists
- Trees
- Hash Tables
- Directed Acyclic Graphs
- . . .

Tree-Based Symbol Table

Big Picture

- Each node holds a symbol/address pair

```
struct sym_node_struct {
  char* symbol;
  int address;
  sym_node_t* left;
  sym_node_t* right;
};
```



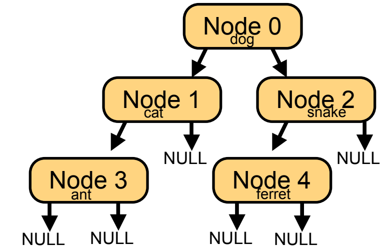
Operations

```
void sym_table_insert(char *, int);
int sym_table_lookup(char *);
```


Tree-Based Symbol Table (cont.)

Invariant

- At each node, all symbols in left children alphabetically precede symbol in node itself
- Also, all symbols in right children alphabetically follow node's symbol



Implications

- Faster lookup
- Slower insertion (can't just prepend)

Optimizations

- Balancing
- Redistributing

Multi-File Compilation

Problem

- What if your program is "too big" to fit in a single .c file?

Solution

- Distribute your code across multiple files

Building

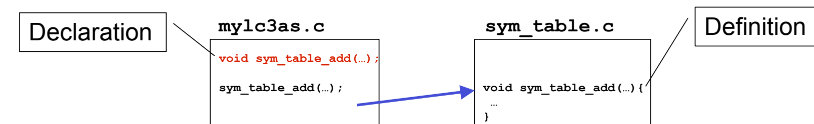
```
% gcc -o mylc3as mylc3as.c sym_table.c parser.c
...or...
% gcc -c mylc3as.c sym_table.c
% gcc -c parser.c
% ls
mylc3as.c      parser.c      sym_table.c
mylc3as.o      parser.o      sym_table.o
% gcc -o mylc3as mylc3as.o sym_table.o parser.o
```

Linking

Multi-File Compilation (cont.)

Problem

- How does one file share code/data with another?



Solution

- **Definitions versus declarations**
- **Definition:** actually allocates storage or creates function
- **Declaration:** states properties of variable or function

Multi-File Compilation (cont.)

What about variables?

- Use the `extern` keyword to distinguish declaration from definition

Example

```
extern int glob;
glob = 3;
```

What if declarations are wrong?

- Link-time error (if you are lucky)

Multi-File Compilation (cont.)

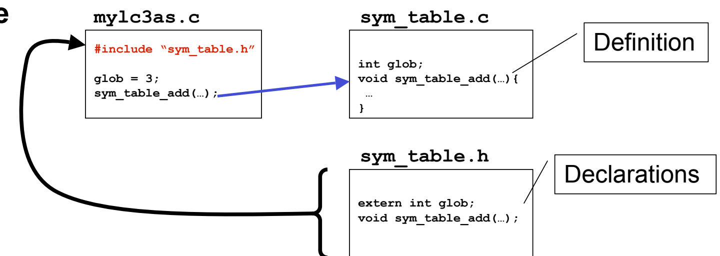
Problems

- We have to type in lots of declarations
- We have lots of duplicate declarations

Solution

- Type them in once and “include” them many times

Example



C Preprocessor (CPP)

CPP implements include

- Precedes actual C compilation process
- Allows programmer to “extend” the language

CPP supports macros

- `#define PI 3.14159`
- `#define max(a,b) (((a)>(b))?(a):(b))`

CPP supports conditional compilation

```
#ifdef PI
printf(“No PI…\n”);
#endif
```


Command Line Arguments

C programs may access command line arguments

```
% emacs mylc3sim.c
```

Q: How does a C program get these arguments?

A: Via function parameters to `main()`

No. of args +1

Array of strings

```
int main(int argc, char **argv) {
    if (argc > 1) {
        printf(”First arg: %s\n", argv[1]);
    }
}
```