Function Pointers

Problem: Suppose I want to call a function in my program, but I won’t know which function the user wants to call until run-time.

Solution: Store the name of the function in a variable and use the variable (Function names are actually pointers!!)

```c
int (*funcPtr) (int i);   // funcPtr will hold the name
                         // of a function that has a
                         // prototype like
                         //   int func(int i)
...
funcPtr = doIt;         // doIt is an actual function
answer = funcPtr(num);  // really a call to doIt
                         // could also be written: answer = (*funcPtr)(num);
```

- Function pointers are most often used as parameters in function calls. (See the qsort function in stdlib)

- Pointers to functions
  - Contain address of function
  - Similar to how array name is address of first element
  - Function name is starting address of code that defines function

- Function pointers can be
  - Passed to functions
  - Returned from functions
  - Stored in arrays
  - Assigned to other function pointers

- Calling functions using pointers (see fig05_25.c)
  - Assume parameter:
    ```c
    bool ( *compare ) ( int, int )
    ```
  - Execute function with either
    ```c
    ( *compare ) ( int1, int2 )
    ```
    - Dereference pointer to function to execute
    OR
    ```c
    compare( int1, int2 )
    ```
    - Confusing - User may think compare name of actual function in program
Arrays of pointers to functions (see fig05_26.c)
  o Menu-driven systems
  o Pointers to each function stored in array of pointers to functions
    • All functions must have same return type and same parameter types
  o Menu choice → subscript into array of function pointers

Making ADTs more universal
• Data stored is of type void *
• Functions are passed as parameters to manipulate the data

// a linked list in C! -- this version includes a dummy
// header element (which exchanges some memory for ease of
// coding.)

#include "stdio.h"
#include <stdlib.h>
typedef struct node Link;
typedef Link List;    // [[ only definition that should be
                         // made public ]]
typedef struct node Node;

struct node
{
    int value;
    Link *next;
};
/*  
  * purpose: create a new linked list  
  * input: nothing!  
  * output: an empty linked list  
  */  
Link *list_create()  
{  
  Link *dummy = (Link) malloc(sizeof(Node));  
  
  if (dummy == NULL)  
  {  
    fprintf(stderr, "Out of memory :(
);  
    exit(-1);  
  }  
  
  dummy->value = -1;  
  dummy->next = NULL;  
  
  return dummy;  
}  

/*/  
* purpose: print a linked list  
* input: a description and the list to print!  
* output: nothing [[ (why it prints stuff???) ]]  
*/  
void list_print(char *description, List *head)  
{  
  printf("%s", description);  
  if (head == NULL)  
  {  
    printf("%s%s", "Warning list == NULL someone ",  
      "didn't call create!\n");  
  }  
  else  
  {  
    Link *tmp;  
    
    // [[ head->next skips the dummy header ]]  
    for(tmp = head->next; tmp != NULL; tmp = tmp->next)  
      printf("%d ", tmp->value);  
    
    printf("\n");  
  }  
}
/ purpose: delete an entry from a linked list
* input:  the head of the list and the value to delete
* output:  the updated list
*/
List *list_delete(Link *head, int value)
{
    Link *tmp;

    // this should not happen (the "empty" list still
    // contains the dummy header), but as a curtsey :)
    if (head == NULL)
    {
        return NULL;
    }

    // again we never delete the first node
    for(tmp = head; tmp->next != NULL; tmp = tmp->next)
    {
        if (tmp->next->value == value)
        {
            Link *rm = tmp->next;
            tmp->next = tmp->next->next;
            free(rm);
            return head;
        }
    }

    return head;
}

/*/ purpose: insert a value into a linked list
* input:  the head of the list and the value to insert!
* output:  the updated list
*/
List *list_insert(Link *head, int new_value)
{
    Link *new_node = (Link*) malloc(sizeof(Node));
    Link *tmp;

    if (new_node == NULL)
    {
        fprintf(stderr, "Out of memory :(
"");
        exit(-1);
    }
new_node->value = new_value;
new_node->next = NULL;

// head should not be null, but in case it is ....
if (head == NULL)
{
    head = list_create();
}

// we never insert before the dummy header, so only the
// 'insert middle and end' cases are needed!
for(tmp = head; tmp->next != NULL; tmp = tmp->next)
{
    if (new_value < tmp->next->value)
    {
        new_node->next = tmp->next;
        tmp->next = new_node;
        return head;
    }
}

tmp->next = new_node;
return head;

// a cheese-y test driver ... what cases are tested? which
// are missed (for new version of list)
ing main()
{
    List *l = list_create();

    list_print("empty list ", l);

    int a[] = {5, 9, 7};
l = list_insert(l, a, compare);
list_print("[5] ", l, output);

    l = list_insert(l, a+1, compare);
list_print("[5 9] ", l, output);

    l = list_insert(l, a+2, compare);
list_print("[5 7 9] ", l, output);

    l = list_delete(l, a+2, compare);
list_print("[5 7 ] ", l, output);
}