

```
int compare_strings(const void *p, const void *q)
{
    return strcmp(p, q);
}
```

Now my program compiles, but `qsort` doesn't seem to sort the array. What am I doing wrong?

- A: First, you can't pass `strcmp` itself to `qsort`, since `qsort` requires a comparison function with two `const void *` parameters. Your `compare_strings` function doesn't work because it incorrectly assumes that `p` and `q` are strings (`char *` pointers). In fact, `p` and `q` point to array elements containing `char *` pointers. To fix `compare_strings`, we'll cast `p` and `q` to type `char **`, then use the `*` operator to remove one level of indirection:

```
int compare_strings(const void *p, const void *q)
{
    return strcmp(*(char **)p, *(char **)q);
}
```

Exercises

Section 17.1

1. Having to check the return value of `malloc` (or any other memory allocation function) each time we call it can be an annoyance. Write a function named `my_malloc` that serves as a "wrapper" for `malloc`. When we call `my_malloc` and ask it to allocate `n` bytes, it in turn calls `malloc`, tests to make sure that `malloc` doesn't return a null pointer, and then returns the pointer from `malloc`. Have `my_malloc` print an error message and terminate the program if `malloc` returns a null pointer.

Section 17.2

2. Write a function named `duplicate` that uses dynamic storage allocation to create a copy of a string. For example, the call


```
p = duplicate(str);
```

 would allocate space for a string of the same length as `str`, copy the contents of `str` into the new string, and return a pointer to it. Have `duplicate` return a null pointer if the memory allocation fails.

Section 17.3

3. Write the following function:


```
int *create_array(int n, int initial_value);
```

 The function should return a pointer to a dynamically allocated `int` array with `n` members, each of which is initialized to `initial_value`. The return value should be `NULL` if the array can't be allocated.

Section 17.5

4. Suppose that the following declarations are in effect:


```
struct point { int x, y; };
struct rectangle { struct point upper_left, lower_right; };
struct rectangle *p;
```

Assume that we want `p` to point to a `rectangle` structure whose upper left corner is at (10, 25) and whose lower right corner is at (20, 15). Write a series of statements that allocate such a structure and initialize it as indicated.

- W 5. Suppose that `f` and `p` are declared as follows:

```
struct {
    union {
        char a, b;
        int c;
    } d;
    int e[5];
} f, *p = &f;
```

Which of the following statements are legal?

- (a) `p->b = ' ';`
- (b) `p->e[3] = 10;`
- (c) `(*p).d.a = '*';`
- (d) `p->d->c = 20;`

6. Modify the `delete_from_list` function so that it uses only one pointer variable instead of two (`cur` and `prev`).

- W 7. The following loop is supposed to delete all nodes from a linked list and release the memory that they occupy. Unfortunately, the loop is incorrect. Explain what's wrong with it and show how to fix the bug.

```
for (p = first; p != NULL; p = p->next)
    free(p);
```

- W 8. Section 15.2 describes a file, `stack.c`, that provides functions for storing integers in a stack. In that section, the stack was implemented as an array. Modify `stack.c` so that a stack is now stored as a linked list. Replace the `contents` and `top` variables by a single variable that points to the first node in the list (the "top" of the stack). Write the functions in `stack.c` so that they use this pointer. Remove the `is_full` function, instead having `push` return either `true` (if memory was available to create a node) or `false` (if not).

9. True or false: If `x` is a structure and `a` is a member of that structure, then `(&x) -> a` is the same as `x.a`. Justify your answer.

10. Modify the `print_part` function of Section 16.2 so that its parameter is a *pointer* to a part structure. Use the `->` operator in your answer.

11. Write the following function:

```
int count_occurrences(struct node *list, int n);
```

The `list` parameter points to a linked list; the function should return the number of times that `n` appears in this list. Assume that the node structure is the one defined in Section 17.5.

12. Write the following function:

```
struct node *find_last(struct node *list, int n);
```

The `list` parameter points to a linked list. The function should return a pointer to the *last* node that contains `n`; it should return `NULL` if `n` doesn't appear in the list. Assume that the node structure is the one defined in Section 17.5.

13. The following function is supposed to insert a new node into its proper place in an ordered list, returning a pointer to the first node in the modified list. Unfortunately, the function

doesn't work correctly in all cases. Explain what's wrong with it and show how to fix it. Assume that the node structure is the one defined in Section 17.5.

```
struct node *insert_into_ordered_list(struct node *list,
                                     struct node *new_node)
{
    struct node *cur = list, *prev = NULL;
    while (cur->value <= new_node->value) {
        prev = cur;
        cur = cur->next;
    }
    prev->next = new_node;
    new_node->next = cur;
    return list;
}
```

Section 17.6

14. Modify the `delete_from_list` function (Section 17.5) so that its first parameter has type `struct node **` (a pointer to a pointer to the first node in a list) and its return type is `void`. `delete_from_list` must modify its first argument to point to the list after the desired node has been deleted.

Section 17.7

- W 15. Show the output of the following program and explain what it does.

```
#include <stdio.h>

int f1(int (*f)(int));
int f2(int i);

int main(void)
{
    printf("Answer: %d\n", f1(f2));
    return 0;
}

int f1(int (*f)(int))
{
    int n = 0;
    while ((*f)(n)) n++;
    return n;
}

int f2(int i)
{
    return i * i + i - 12;
}
```

16. Write the following function. The call `sum(g, i, j)` should return $g(i) + \dots + g(j)$.
`int sum(int (*f)(int), int start, int end);`
- W 17. Let `a` be an array of 100 integers. Write a call of `qsort` that sorts only the *last* 50 elements in `a`. (You don't need to write the comparison function).
18. Modify the `compare_parts` function so that parts are sorted with their numbers in *descending* order.
19. Write a function that, when given a string as its argument, searches the following array of structures for a matching command name, then calls the function associated with that name.