Homework Assignment 5 (Extra Credit)

Name: **Asrar Syed**

Due by: **December 01, 2024**

Course Section: **CSC 3320-034**

1. (1 point) In the following declarations, the x and y structures have members named x and y:

struct { int x, y; } x;

struct { int x, y;} y;

Are these declarations legal on an individual basis? Could both declarations appear as shown in a program? Justify your answer.

On a individual basis they are both legal declarations, and are examples of anonymous structures. The variables **x** and **y** are declared as instances of these structures. No, both declarations **cannot appear in the same program** as written because they both declare **x** and **y** as identifiers at the global scope.

The main issue that arises is a naming conflict, with the variable **x** declared by the first structure conflicts with the variable **x** in the second structure declaration. Same for the variable **y**.

2. (1 point)

struct {

double a;

union {

char b[4];

double c;

int d;

} e;

char f[4];

} s;

If char values occupy one byte, **int** values occupy four bytes, and **double** values occupy eight bytes, how much space will a C compiler allocate for s? Justify. (Assume that the compiler leaves no "holes" between members.) **Hint:** consider the memory alignment requirement for structure/union.

The first **double a** occupies **8 bytes** and **starts at offset 0**. The **union e** occupies **8 bytes (size of the largest member, double c)** and **starts at offset 8**. Next, the **char f[4]** occupies **4 bytes** and **starts at offset 16** immediately after **union e**.

Lastly we must consider the padding for the struct alignment, since the total size of the struct must be a multiple of the largest alignment requirement (8 bytes). The total size so far is **8 + 8 + 4 = 20 bytes**.

To align the struct to 8 bytes, we need **4 bytes of padding**.

Total memory space for **struct s** is **24 bytes**. The total size of the struct is **20 bytes** (ignoring padding).

3. (1 point)

union {

double a;

struct {

char b[4];

double c;

int d;

} e;

char f[4];

} u;

If char values occupy one byte, **int** values occupy four bytes, and **double** values occupy eight bytes, how much space will a C compiler allocate for u? Justify. (Assume that the compiler leaves no "holes" between members.) **Hint:** consider the memory alignment requirement for structure/union.

Since all members of a union share the same memory space, the **union u** will allocate space based on the largest member. The largest member is **struct e**, which occupies **16 bytes** due to alignment requirements.

4. (1 point) Suppose that **b** and **i** are declared as follows:

enum {FALSE, TRUE} b ;

int i;

Identify which of the following statements are legal and/or “safe’ (always yield a meaningful result). Justify.

(a) b = FALSE;

**Is it Legal & Safe? (Answer: Yes & Yes)** This is legal cause **FALSE** is a valid value. Since **FALSE** is essentially corresponds to **zero**, then assigning it to **b** is also meaningful.

(b) b = i;

**Is it Legal & Safe? (Answer: Yes & No)** Assigning an integer (**ascii value of i is 10**5) to an enum variable is valid. No it is not safe because if **i** is any value besides **zero or one** then it could take on an undefined or meaningless value that is not part of the enumeration.

(c) b++;

**Is it Legal & Safe? (Answer: Yes & No)** Incrementing an enum variable is syntactically correct and is similar to incrementing integers in C. No it is not safe because, incrementing b could result in a value outside the defined range of the enumeration (FALSE and TRUE) i.e. undefined or meaningless value.

(d) i = b;

**Is it Legal & Safe? (Answer: Yes & Yes)** It is legal because assigning an enum variable to an int is allowed. It is also safe because the value of **b** is either FALSE = 0 or TRUE = 1 and this will be assigned to **i**, which is meaningful as an integer.

(e) i = 2 \* b + 1;

**Is it Legal & Safe? (Answer: Yes & Yes)** Because arithmetic operations involving enum variables are allowed it is legal. This is also safe, **b** will always be 0 (FALSE) or 1 (TRUE), and the expression will always yield a predictable result (either 1 or 3).

5. (1 point) Suppose that f and p are declared as follows:

struct {

union {

char a b; **// Asrar: This is invalid syntax in C**

int c;

} d;

int e[5];

} f, \* p = & f;

Which of the following statements are legal? Why? If not legal, correct them.

If we assume that this is intending to be:

char a, b; // Correct

--------------------------

char a;

char b; // Correct

1. p->b = ‘ ’;

**Legal? Yes:** Since **b** is a valid member of the **union** **d**. The expression **p->b** accesses **f.d.b**, so assigning **“ ”** to it is valid.

1. p->e [3] = 10;

**Legal? Yes:** The **array e** is a member of the **structure f** so it is also valid, and p->e[3] accesses its 4th element (index 3).

1. (\*p).d.a = ‘\*’;

**Legal? Yes:** This is also valid because the expression **(\*p).d** dereferences the **pointer p** to access the **union d**, and **.a** accesses the **char a** member of the union.

1. p->d->c = 20;

**Legal? No:** This is not valid as **d** is a union, not a pointer itself, so using arrow operator on **d** is illegal. The correct method is to access the integer variable c inside d using the following:

**p->d.c = 20;**

6. (1 point) 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);

There is a memory access issue within the loop after **free(p);** where the loop frees the **node p** without first storing **p->next**. Once **p** is freed, its memory is deallocated, and accessing **p->next** results in undefined behavior. This will cause the loop to crash or corrupt the memory due to it accessing the freed memory.

The proper method is to save the next pointer of the current node within a temporary pointer. This will allow us to to safely move to the next node after the current one is deallocated.

struct Node\* temp; // Temporary pointer to store the next node

for (p = first; p != NULL;) {

temp = p -> next; // Store the next node

free(p); // Free the current node

p = temp; // Move to the next node

}

7. (1 point) 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.

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;

}

The problem with the function is that it assumes the **new\_node** will always be inserted somewhere after the **first node**, which is not true. Specifically the function fails in the while loop conditional at new\_node -> value where the node is less than or equal to the value of first node. The reason is:

**prev** remains **NULL** and the line **prev->next = new\_node** will result in a **segmentation fault** since prev is not initialized to point to anything.

The case where **new\_node** needs to be inserted at the beginning of the list is not handled.

8. (1 point) Consider the following definition of node:

struct node {

int value;

struct node \* next;

};

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. Write a complete program to check your function. You **must** submit your complete solution code (.c file).

Code File attached.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Question: | [1](#_bookmark0) | [2](#_bookmark1) | [3](#_bookmark2) | [4](#_bookmark3) | [5](#_bookmark4) | [6](#_bookmark5) | [7](#_bookmark6) | [8](#_bookmark7) | Total |
| Points: | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 8 |
| Bonus Points: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Score: |  |  |  |  |  |  |  |  |  |