**Lena Kemmelmeier**

CS 326 - Programming Languages, Concepts and Implementation

Homework 5
Submitted: April 10th, 2024

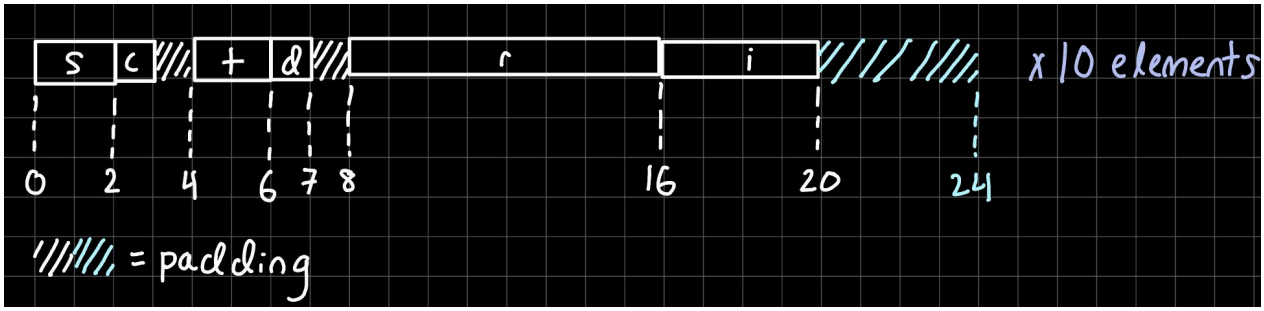
CS326 – Programming Languages, Concepts and Implementation

# Homework 5

## (Due April 11)

1. (25 pts) Suppose we are compiling for a machine with 1-byte characters, 2-byte shorts, 4-byte integers, and 8-byte reals, and with alignment rules that require the address of every primitive data element to be a multiple of the element's size. Suppose further that the compiler is not permitted to reorder fields. How much space will be consumed by the following array? Explain.

```
A : array [0..9] of record
                        s : short
                        c : char
                        t : short
                        d : char
                        r : real
                        i : integer
```




**There will be 240 bytes consumed (see illustration above). We need padding between c and t (and between d and r) because of the alignment rules that require the address of every primitive data member to be a multiple of the element's size (i.e. t could not have offset three because shorts need 2 bytes). Moreover, we need to pad out to 24 because 20 is not a multiple of 8, and we need to make sure that r is aligned. Altogether, we have 24 bytes for each array element, and the array has 10 elements, leading us back to our answer of 240 bytes.**

2. (25 pts) For the following code specify which of the variables a,b,c,d are type equivalent under (a) structural equivalence, (b) strict name equivalence, and (c) loose name equivalence.

```
Type T = array [1..10] of integer
         S = T
```

```
a : T
b : T
c : S
d : array [1..10] of integer
```

**(a) structural equivalence - a, b, c, and d are all compatible with one another.**
**(b) strict name equivalence - a and b would be compatible with one another (and every other combination would be incompatible).**
**(c) loose name equivalence - a, b, and c would be compatible with one another (and every other combination would be incompatible).**

3. (25 pts) We are trying to run the following C program:

```
typedef struct
{
   int a;
   char * b;
} Cell;


void AllocateCell (Cell * q)
{
   q = (Cell *) malloc ( sizeof(Cell) );
}
void main ()
{
   Cell * c;
   AllocateCell (c);
   c->a = 1;
   free(c);
}
```

The program produces a run-time error. Why? **AllocateCell is taking the pointer by value and not reference, meaning that the changes made inside of it don't affect what's going on with the pointer in main. In other words, when AllocateCell (in main) returns, c is still pointing to an undefined location in memory. This causes the line with c->a to produce a runtime error (free here is also an issue because you can't free that memory if c is uninitialized). To fix this, we need to be passing that pointer by reference instead (see changes below).**

Rewrite the functions `AllocateCell` and `main` so that the program runs correctly.

```
id AllocateCell (Cell ** q)
{
   *q = (Cell *) malloc ( sizeof(Cell) );
}
void main ()
{
   Cell * c;
   AllocateCell (&c);
   c->a = 1;
   free(c);
}
```

4. (25 pts) Consider the following C declaration, compiled on a 32-bit Pentium machine (with array **elements aligned at addresses multiple of 4 bytes**).

```
struct
{
   int n;
   char c;
} A[10][10];
```

If the address of `A[0][0]` is 1000 (decimal), what is the address of `A[3][7]`? Explain how this is computed.

1) **First, we need to figure out how much space we need for an element in the array. We need 4 bytes for the int and 1 byte for the char, but we also need to make sure elements are aligned at addresses multiple of 4 bytes, meaning we pad 3 bytes to offset c. Altogether, the size of one element in the array will be 8 bytes.**
2) **Splitting how we get there by rows and columns**
    a) **To find the distance of A[3][0] from A[0][0], we need to find the size of three rows: 3 rows * 8 bytes * 10 elements**
    b) **Now to get from A[3][0] to A[3][7], we need to move 7 elements over: 7 elements * 8 bytes**
3) **Now, we go from the address of A[0][0] and add the offsets. Our final answer is 1296:**
    a) **= 1000 + (3*8*10) + (7*8)**
    **= 1296**

5. (Extra Credit - 10 pts) Write a small fragment of code that shows how unions can be used in C to interpret the bits of a value of one type as if they represented a value of some other type (non-converting type cast).

```
union ExtraCredit{
    int integer;
    float floatNum;
};
int main(){
    union ExtraCredit ecUnion;
    ecUnion.float = 11.26f;
    printf("%f\n", ecUnion.float);
    printf("%d\n", ecUnion.int);

    return 0;
}
```