Bryan Kline

CS326

Homework 5

LaTex (TeXstudio)

11/10/2016

1. The Boolean type is typically represented on a byte in memory, although it can have only two possible values. Why not represent the Boolean type on a single bit?

Addresses refer to whole bytes in memory and so eight bits is in a sense the smallest addressable length of memory in most computer architectures, and while it would in principle be possible to address individual bits, doing so would take up more space in terms of the number of addresses necessary to keep, eight times as many. It's more efficient to simply give Boolean type variables an entire byte to save from having to address to the bit level. There is also the question of type compatibility; in languages like C where non-Boolean types can be treated like Booleans and where Booleans can have some arithmetic operations carried out on them, it's important that they be a byte to facilitate this.

2. 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
{
    int x;
    float y;
} example;

//the number 23 in IEEE 754 floating point representation
example.x = 0b01000001101110000000000000000000;

printf("%f\n", example.y);
printf("%d\n", example.x);

example.x = example.x / 2;
example.y = example.y + 0.2;

printf("\n");
printf("%f\n", example.y);
printf("%d\n", example.x);
```

The above outputs the following:

```
23.000000
1102577664

0.200000
1045220557
```

The union creates space for the largest element in the union, in this case they are the same size, but space for only is allocated and in the first line where `example.x` is set to the bit string corresponding to the IEEE 754 floating point representation of the number 23, the value of `example.x` is 1102577664 and the value of `example.y` is 23.000000. When the int is divided by two, and the float has 0.2 added to it their values both change in ways that correspond to adding 0.2 to the IEEE representation. This demonstrates that the union creates space for only one variable and then any changes to either name in the union hold for the other.

3. Consider the following:

```
typedef struct
{
    int x;
    char y;
} Rec1;

typedef Rec1 Rec2;

typedef struct
{
    int x;
    char y;
} Rec3;

Rec1 a, b;
Rec2 c;
Rec3 d;
```

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.

a)

The variables `a,b` and `d` are structurally equivalent because `a` and `b` are both of type `Rec1` and `d` is of type `Rec3` and `Rec1` and `Rec3` are structurally the same, i.e. have the same fields in the same order.

b)

The variables `a` and `b` are strictly name equivalent because they are both of type `Rec1`, and not name equivalent with `c` because **`typedef`** `Rec1 Rec2;` is a declaration and a definition where `c` is a different type.

c)

The variables `a,b` and `c` are loosely name equivalent because they are both of type `Rec1`, the line **`typedef`** `Rec1 Rec2;` makes `Rec2` and alias of `Rec1`, just a different name for the same type.

4. Consider the following program, written in C:

```
typedef struct
{
    int x;
    int y;
} Foo;

void allocate_node (Foo * f)
{
    f = (Foo *) malloc ( sizeof(Foo) );
}

void main()
{
    Foo * p;
```

```
    allocate_node (p);
    p->x = 2;
    p->y = 3;
    free(p);
}
```

Although the program compiles, it produces a run-time error. Why? Rewrite the two functions allocate_node and main so that the program runs correctly.

The reason that it produces a run-time error is that when p is dereferenced and dotted to assign the fields x and y the pointer p doesn't point to anything and that causes a segmentation fault. The reason that p doesn't point to anything is that it's not changed in the function. In order to change p a pointer to it must be passed in, or a pointer to a Foo pointer. The code below, which has a print statement added to verify that the struct fields were changed, demonstrates this:

```
typedef struct
{
    int x;
    int y;
} Foo;

void allocateNode(Foo** f)
{
    *f = (Foo*) malloc(sizeof(Foo));
}

int main()
{
    Foo **p;
    Foo* z;

    allocateNode(p);

    z = *p;
    z->x = 2;
    z->y = 3;

    printf("x is %d and y is %d\n", z->x, z->y);

    free(*p);
    p = NULL;
    z = NULL;

    return 0;
}
```

The above code outputs:

```
x is 2 and y is 3
```

So the struct fields were successfully changed because a pointer to a Foo pointer was passed into the function wherein memory was allocated for a Foo struct so that when it's dereferenced it doesn't seg fault.

5. (**Extra Credit**) A compiler for a language with static scoping typically uses a LeBlanc-Cook symbol table to track the bindings of various names encountered in a program. Describe the mechanism that should be used by a compiler to ensure that the names of record fields used inside a with statement are bound to the correct objects. Specify what should be placed in (a) the symbol table, (b) the scope stack, and (c) the run-time program stack.