Patrick Austin  
CS 326

Homework # 5

11/4/2015

1. Boolean types are represented on a byte even though one might expect a bit would suffice because of addressing limitations. A byte is the smallest possible address, so a Boolean is usually represented on a byte and not a bit.

2. Consider the following code fragment:

typedef union

{

int x;

float y;

} u ;

void main()

{

u test; //create a union object

test.x = 1065353216; //assign the int value of the union 1065353216

printf("%i\n", test.x); //prints int 1065353216, interpreting the bits using

//representation of an int

printf("%f\n", test.y); //prints float 1.000000, using the same bits

//but interpreting them using representation of float

}

This code snippet shows a union being used for a non-converting typecast in C. The union is first assigned an integer value, which is printed as 1065353216 to console.

Because of the properties of the union in C, attempting to then print the the float contained in the union performs a non-converting typecast. The bits that are used to represent the integer 1065353216 are interpreted using the representation of a float. In this case, the bits that represent integer 1065353216 represent the float 1.000000, which is printed to the console.

No conversion has occurred- only the bits underlying a value of one type have been interpreted as if they represented a value of another type.

3. a. Under structural equivalence, a b c and d are all equivalent.

b. Under strict name equivalence, only a and b are equivalent.

c. Under loose name equivalence, only a b and c are equivalent.

4.

This program produces an error during runtime due to an attempt to access a garbage memory location. That is to say: p is modified within the allocate\_node function to point to the allocated location, but this change is lost when leaving allocate\_node because p was passed by value. The program then attempts to access memory at the garbage value p had when initialized and produces a runtime error.

I opted to fix this issue by making allocate\_node take a pointer to a pointer, which is then dereferenced in order to allocate the memory, and passing allocate\_node the address of p within main. This sidesteps the pass by value issue- the “outer” pointer isn’t modified at all, but the “inner” one is, and that change will stick when leaving allocate\_node.

The following revised code enables the program to run correctly:

typedef struct

{

int x;

int y;

} Foo;

void allocate\_node (Foo \*\*f) //function now expects pointer to pointer

{

\*f = (Foo \*) malloc ( sizeof (Foo) ); //malloc at dereference of f, ie “inner” pointer

}

void main()

{

Foo \* p;

allocate\_node (&p); //pass address of p, so pointer to pointer is being passed

p->x = 2;

p->y = 3;

free(p);

}