CS 61 Scribe Notes
Computer Arithmetic II, Structured Data, Memory Layout (Th 9/13)

Renzo Lucioni, Weishen Mead, Kat Zhou

Unsigned Computer Arithmetic II

Consider the following:

$$x \ \& \ -x = \ ?$$

We would like to find the value of the right-hand side of this equation. We know that

$$(\sim x + 1) = -x$$

Before forming a hypothesis, we work with small examples and look for patterns.

$x = 0$:

$$0000_2 \ \& \ 0000_2 = 0000_2 = 0$$

$x = 1$:

$$0001_2 \ \& \ 1111_2 = 0001_2 = 1$$

$x = 2$:

$$0010_2 \ \& \ 1110_2 = 0010_2 = 2$$

We now have enough information to form a reasonable hypothesis:

$$x \ \& \ -x = x$$

Let's test our hypothesis with a few more examples, to see if it holds.

$x = 5$:

$$0101_2 \ \& \ 1011_2 = 0001_2 = 1$$

$x = 8$:

$$1000_2 \ \& \ 1000_2 = 1000_2 = 8$$

As we can now tell from the above,

$$x \ \& \ -x \neq x$$

Let's try a new, revised hypothesis:

> "When x is even, the answer is x, and when x is odd, the answer is 1."

To test this, let's try giving x an even value:

$x = 6$:

$$0110_2 \,\&\, 1010_2 = 0010_2 = 2$$

Unfortunately, this disproves our revised hypothesis.

So what does $x \& -x$ really equal? It *returns the least significant 1 bit of* x*'s binary representation* (least significant meaning smallest, or the one furthest to the right).

Here's the proof:

Assume $-x = (\sim x + 1)$. Say we have

$x = [\alpha]1[0...0]$, where there are k zeroes

Then,

$\sim x = [\sim \alpha]0[1...1]$, where there are k ones

Finally,

$\sim x + 1 = [\sim \alpha]0[1...1] + 1 = [\sim \alpha]1[0...0]$, where there are k zeroes

As can be seen, adding the one causes a sort of "ripple effect" through the k ones, turning them into zeroes and causing the one to "bubble" up.

If we now take $x \& (\sim x + 1) = x \& -x$, the α cancels with ~α to make all zeroes. Anding zeroes and zeroes make more zeroes, leaving only the rightmost one.

Multiplication and Division
Consider the following:

$$x << i = x * 2^i$$

This is because x is represented as $x_{w-1}x_{w-2}...x_2x_1x_0$. After shifting i to the left, we are left with y represented as $x_2x_1x_00...0$, where there are i zeroes. More precisely:

$$y = \sum_{k=0}^{w-1} y_k = \sum_{k=1}^{w-1} 2^k x_{k-i} = \sum_{k=0}^{w-1-i} 2^i 2^k * x_k = 2^i \sum 2^k x_k = 2^i x$$

Similarly, consider this:

$$x >> i = x / 2^i$$

Shifts are faster than their regular mathematical counterparts when it comes to division and multiplication, owing primarily to the fact that each bit is affected by one single bit as opposed to each bit being affected by a different bit.

Also, the following use of & is faster than modulo:

$$x \,\&\, (2^i - 1) = x \,\%\, 2^i$$

Signed Arithmetic
Due to Two's complement, signed addition and unsigned addition produce the same bit pattern. The same is true for signed and unsigned subtraction, and also for signed and unsigned multiplication (since multiplication amounts to nothing more than repeated addition). However, signed and unsigned division do not return the same bit patterns. For example:

$$-1/2 = 0$$
$$15/2 = 7$$

-1 and 15 have the same bit representations, but they return answers that do not share the same bit pattern.

C's Binary Logical Operators
C's binary logical operators are: !, &&, ||, == !=, <, >, <=, and >=.

N.B. Logical operators operate on truth values and only return 1 or 0, where 1 is true and 0 is false. For example, !!x and x != 0 are equivalent (this is as trick for casting t`o bool`).

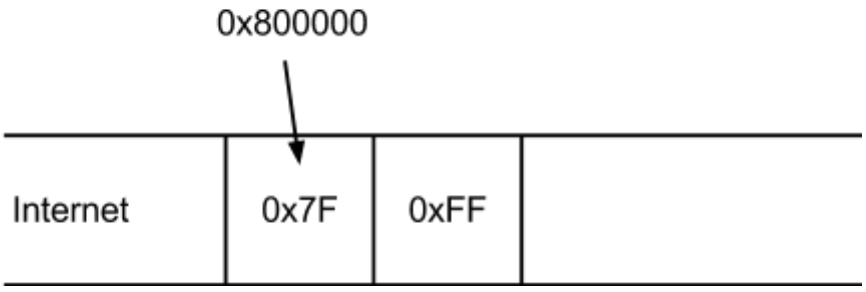
Data Representation
Memory can be thought of as an enormous array of unsigned char. On a 32-bit machine, the largest address possible is $2^{32} - 1$.

The range of values for an object of type signed `short int` is -32768 to 32767. An unsigned `short int` can represent values from 0 to 65535. A `short int` is 16-bit object (2 bytes).

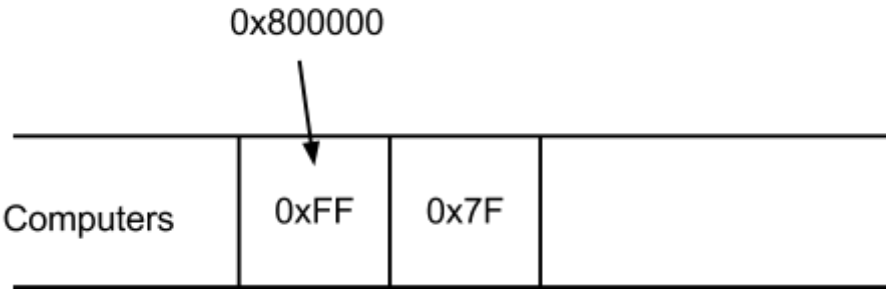
Here's an example of how an object is stored in memory. Pick a `short int`, say 32767 == 0x7FFF. Let's say we're storing this in the address 0x800000. On the internet, this address

holds 0x7F and the next address holds 0xFF, meaning that more significant bytes are transferred and stored first.

This style of ordering the individually addressable sub-components (bytes) within the representation of the larger data item (the `short int`) is called *big endian*.



On the other hand, in most PCs (in particular those powered by Intel chips, i.e., almost all) 0xFF goes in the lower numbered address and 0x7F goes in the next one, meaning the least significant byte is stored first - this ordering style is referred to as *little endian*.



Let's briefly review the size of a few C data types on a 32-bit machine. An `int` is 4 bytes, a `long long` is 8 bytes, and a `char *` is 4 bytes (8 bytes on a 64-bit machine).

Now let's take a look at arrays. As stated previously, memory can be thought of as an enormous array of `unsigned char`. What is the best way to represent an array of `ints`?

To store 10 `ints`, allocate a contiguous 40-byte chunk of memory. Why 40 bytes? The amount of memory you must allocate is given by multiplying the number of objects you want to store by the byte size of the object. In this case, an `int` is 4 bytes long, and you want to store 10 of them, hence you need 40 bytes. Let's say the chunk of memory you have allocated starts at 0x70000; it will end at 0x70028. Such a contiguously allocated subset of memory is referred to as a *homogenous collection.* An array is an example of a homogenous collection.

N.B. Given an array $x[]$ of type T, where the address of the array is A, the address of item i is $A + i * sizeof(T)$.

Heterogenous collections are those where every element is not the same (and hence the size of every element is not the same). A good example of a heterogenous collection C is a `struct`:

```
struct foo {
    int a;
    char b;
    unsigned char c;
    int *p;
}
```

Like the array, the `struct` is also laid out in a contiguous block. However, the objects stored within this block are not of uniform size, as they were in the array.

We can inspect the locations of each of these items in memory:

```
int main () {
    foo p;
    printf ("%p %p %p\n", &p, &p.a, &p.b, &p.c, &p.p);
}
```

The `struct` takes up 12 bytes in memory. The difference between the addresses of `p` and `p.a` is 0 bytes (they're at the same location), while the difference between `p.a` and `p.b` is 4 bytes, the difference between `p.b` and `p.c` is 1 byte, and the difference between `p.c` and `p.p` is 3 bytes.

Each type has an alignment number that should evenly divide its address. The alignment number of the largest object is always used. In this example, the largest objects are the `int` and the `int *` (each of size 4 bytes), and so the alignment number is used is 4. Padding is added at the end of the `struct` to make its size a multiple of 4, since as we have just discussed, the `struct` needs to be aligned on a 4-byte boundary. This use of padding explains the 3 byte difference between `p.c` and `p.p` (2 bytes of padding are added between them).

If we eliminate the `int *` from the struct, we might reasonably expect the size to be 6 bytes, but in fact the size comes to 8 bytes. This is again due to the fact that the `struct` must be aligned on a 4-byte boundary, since it still contains an `int`.

Here is an example of our third type of collection, the *overlapping collection:*

```
union foo{
    int a;
    char b;
    unsigned char c;
```

```
    int *p;
}
```

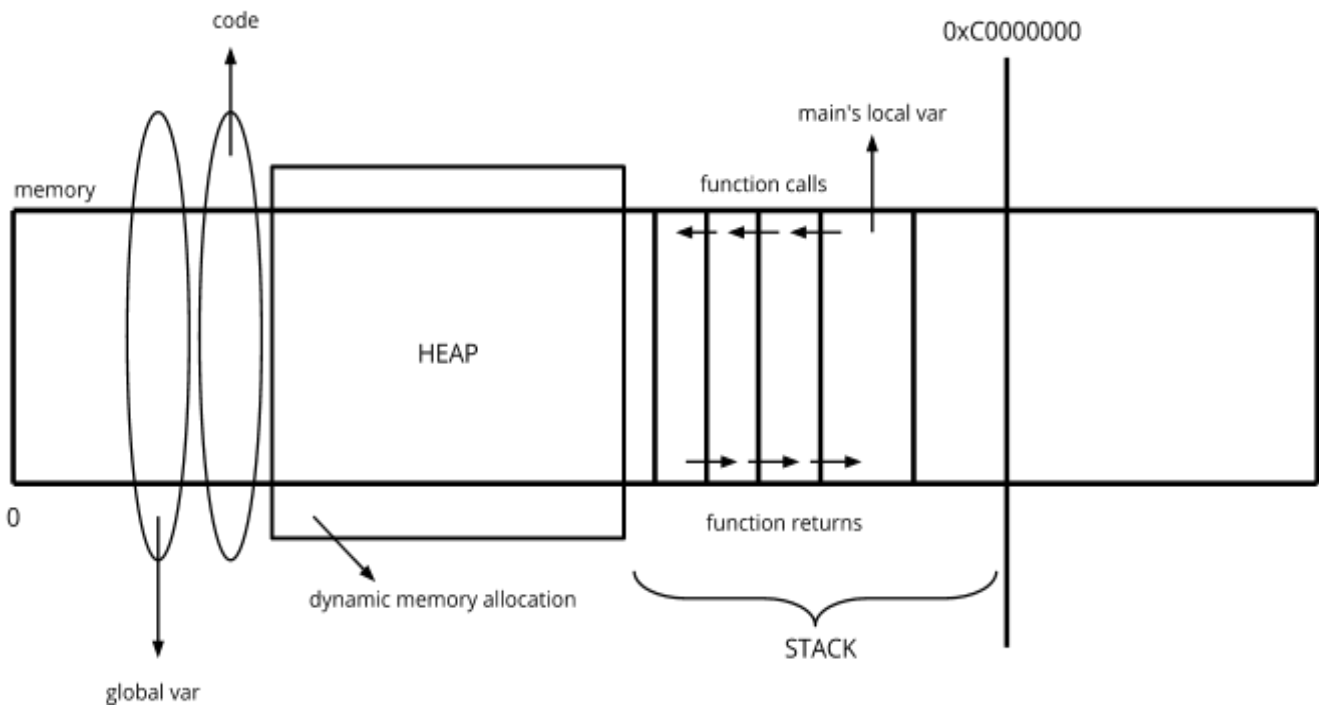
The type `union` can be viewed as a variable type that can contain many different variables (like a `struct`), but only actually holds one of them at a time (not like a `struct`). The size of a `union` is equal to the size of its largest data member. In other words, the C compiler allocates just enough space for the largest member. This is because only one member can be used at a time, so the size of the largest is the most that will be needed. If we print out the addresses of the elements of this `union`, each element has the same memory address; the union is aligned according to the alignment number of the largest element.

Finally, we will inspect the memory usage of the following recursive function used to compute $n!$:

```
int factorial(int n){
    if (n==0)
        return 1;
    else
        return factorial(n-1)*n;
}
```

Each time the `factorial` function is called, a new argument `n` is generated. Each `n` has its own address. Where in memory are these `n`'s stored? We can answer this question by printing the address of each `n`. By doing this, we observe that each subsequent `n` is located at a lower address than the previous. Also note that each time the program is run anew, new addresses are used for all `n`'s; this is for security purposes.

How are functions laid out in memory? As we saw, `main's` local variables are stored somewhere to the right of center (near, but below, the address 0xC0000000). Function calls move to progressively lower memory addresses, and as function calls return they move back up, reusing the space. Code itself is stored in markedly lower memory addresses, with global variables even lower. The region located in higher addresses, containing local variables and function calls, is called the *stack*. The region located in lower in lower addresses, between the stack and the code, is called the *heap,* and is used for dynamic memory allocation.


code
0xC0000000
main's local var
memory
function calls
HEAP
0
function returns
dynamic memory allocation
STACK
global var