

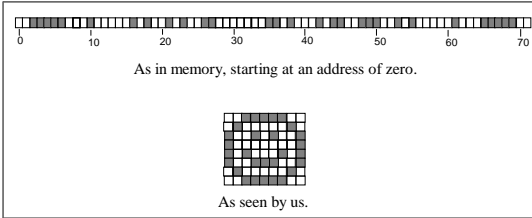
# Memory and Addressing

## How is memory organized?

How information is stored is important in understanding many programming languages. The memory of a computer is made up of sequential slots all given a certain number which represents its position, similar to how the number on a street acts as part of an address to a house (which contain people, furniture, and other things). Before I can store information, I have to know where I'm going to put it, and before I can retrieve information, I have to know from where I'm going to get it.

## Picture Illustration

To illustrate how the computer stores information, let's look at a picture formed of bits. The picture has 72 bits total, and each bit contains either a solid black or an empty block. The picture itself is easily seen by starting a new line after every 9 bits, but in memory the picture is actually just a string of bits. Notice how the picture stored in memory differs from what is displayed. Other data types used in computer programming are the same, what you see is not



necessarily organized in the same way you may think it is, and it doesn't need to be either. In our bit example, it would be nice to get bits and set bits. For those operations, we could make some pseudo assembly language instructions:

```

gbt <register>, <address register>      get bit
sbt <register>, <address register>      <register> = *<address register>
                                        set bit
                                        *<address register> = <register>

```

For simplicity, we'll assume the address is stored in a register. When programming in MIPS this is also the case when loading or storing information. In C the address is usually stored in a variable which has the same operational value as a register. We now need an instruction for putting an address into a register and putting a bit in a register.

```

lda <address register>, <address value>  load address
<address register> = <address value>
ldb <register>, <bit value>             load bit
<register> = <bit value>

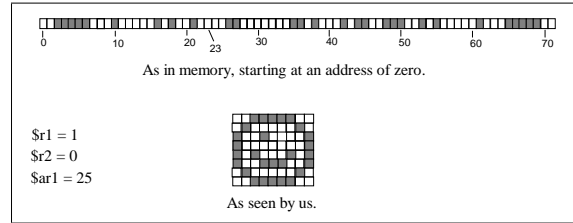
```

Now we can get and put bits on our picture. What if we used the following instructions on the picture above? (Assume \$ar is an address register and \$r is a bit register.)

```

lda $ar1, 23
gbit $r1, $ar1
ldb $r2, 0
sbit $r2, $ar1
lda $ar1, 25
gbit $r2, $ar1

```



It is also helpful if you can increment or decrement the address of a picture's address to advance it forward or backward. This is normally done by simply adding or subtracting a value from the currently stored value. Instructions might look like:

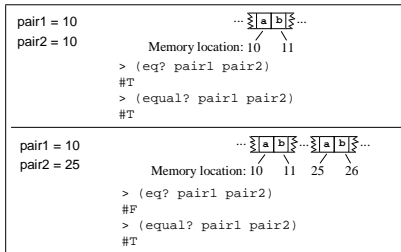
```

add <address register>, <index>        add
<address register> += <index>
sub <address register>, <index>        subtract
<address register> -= <index>

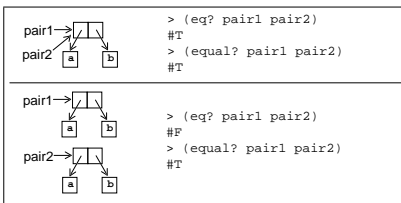
```

## Scheme and Pointers

Even in languages which don't explicitly allow you to use pointers, pointers are used extensively. Scheme is one of these languages. Recall in Scheme two variables pointing at pairs. If the variables point to the same pair, then *eq?* returns #T. If they look the same but aren't the same entity, *eq?* returns #F, even though *equal?* returns #T. What's the difference?



The real difference has to do with the way pairs are stored in memory. You can think of the arrows in Scheme as being actual pointers or addresses to locations in memory. The memory where a variable's data is kept is different than the location of the pair's actual data. Below is a diagram showing how the variable information may be stored in both examples:

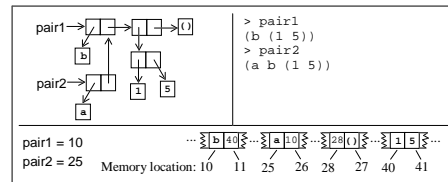


Notice that the variable isn't the pair at all! It's a number representing the position of the pair in memory! When a pair isn't the same entity as another pair, it means they don't reside in the same place in memory!

The variable is only given the starting address of the pair because all pairs take up two consecutive blocks in memory. Since Scheme knows this, it only needs to know what the first block's address is, and the second block's position can be found by incrementing the address by one.

Now you know why when you use *set!* on a variable name, the pair doesn't change. It just replaces the value representing the address of the pair without touching the pair! The reason why two *eq?* pairs change when you only change one of them is because it modifies the memory

that they both share! Just for fun, I'll show you what the memory would look like for pairs pointing to other pairs (if a number is not bold, it's a pointer).



## C and Pointers

Pointers in C are extremely similar to those found in the bit picture. The main difference is that you have more memory being addressed, it addresses bytes instead of bits, and most of it you can't touch. You must also let C determine what memory is useable before actually using it, or the program may cause a segmentation fault (we'll learn about this later in the course).

The two important symbols related to pointers in C are \* and &. The \* is used in two places in C. One is in defining a variable as being a pointer, and the other is in telling the compiler to use the location at where the pointer variable points, instead of the variable. Examples are as follows:

```

int *ptr;           // Defines ptr to be a pointer to data of type integer.
char *string;      // Defines string to be a pointer to data of type
                   // character.
ptr = 5;           // Sets the data at the location pointed to by ptr to 5.
char **ptr;        // Increments the location pointed to by string by the
                   // number located at the place ptr indicates.

```

The & is used to obtain the address of the given location where the variable's data is stored. (Do not confuse this with the symbol used in C++ for call-by-reference).

```

int num = 10;      // Defines a variable num and store the value 10 in it.
ptr = &num;        // Makes the integer pointer, ptr, point to where num
                   // stores its data.
*ptr += 5;         // Increases the variable num by five.

```

In C a string is really an array of characters, or a sequence of characters just like the picture was a sequence of bits. So if we had the following:

```

char c1, c2, c3;
char string[] = "Hello world!";
c1 = string[0];
c2 = *(string + 5);
c3 = *string;

```

What would *c1*, *c2*, and *c3* be?

```
c1 = 'H'  
c2 = 'o'  
c3 = 'H'
```

Notice that the array is already a pointer, so the *&* does not need to be used to access its elements.

### MIPS and Pointers

There are quite a few useful instructions for the MIPS to manipulate pointers, so I'll just list a few here that are similar to the ones in the bit instruction set. Notice that normal registers are the same as address registers in the MIPS architecture.

```
add <$r1>, <$r2>, <$r3>      <$r1> = <$r2> + <$r3>  
addi <$r1>, <$r2>, <num>    <$r1> = <$r2> + <num>  
lui <$r1>, <num>           <$r1> = <num> * 216  
lw <$r1>, <index>(<$r2>)   <$r1> = *(<$r2> + <index>)  
sw <$r1>, <index>(<$r2>)   *(<$r2> + <index>) = <$r1>
```

At this point it's not important to know how these actually work. Just know they exist. In order to load a full 32 bit number, you must use the following instructions. *Ori* just fills in the lower 16 bits with the 16 bits provided in the number.

```
lui <$r1>, <upper 16 bits>  
ori <$r1>, <$r1>, <lower 16 bits>
```