rate entities. When the compiler assigned the value of variable i to variable j, it will "copy" that value from one variable to the other.

GETTING TO THE POINT(ER)

As we mentioned earlier, when a variable is declared, the compiler assigns an appropriately sized chunk of memory to it. This chunk has an address, a number, much like a house. Should you want to access this address, you can do so with the reference (&) operator. This isn't very useful in itself, though. Fear not, the creators of C didn't add this ability to get the address without being able to doing something practical with it as well. This is where the pointer (*) operator enters the picture.

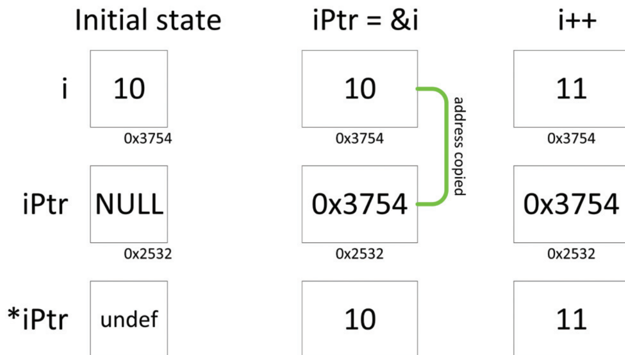
Previously, we saw that the r-value of a variable contains the actual value that we assigned. With a pointer, the r-value is in fact the address of the variable we're pointing at. But what if we want to see the value of this variable we're pointing at? We can do that with the dereference operator, also a "*". Combining all this new-found knowledge, we get the following:

```
// ptr tutorial example 2
task main()
{
  int i = 10;
  int *iPtr;
  iPtr = &i;  // iPtr now points at the address of i
  // This should print out
  // i: 10, iPtr: 656F6968, *iPtr: 10
  // (value for iPtr may vary)
  writeDebugStreamLine("i: %d, iPtr: %p, *iPtr:
%d", i, iPtr, *iPtr);

  i++;
  // This should print out
  // i: 11, iPtr: 656F6968, *iPtr: 11
  // (value for iPtr may vary)
  writeDebugStreamLine("i: %d, iPtr: %p, *iPtr:
%d", i, iPtr, *iPtr);
}
```

The output in the ROBOTC Debug Stream should look like this:

```
i: 10, iPtr: 656F6968, *iPtr: 10
i: 11, iPtr: 656F6968, *iPtr: 11
```

If you look at it from a memory perspective, our program will should look like this:



POINTER ARITHMETIC

So now that you know how what pointers are, let's have some fun with them. Consider the program below:

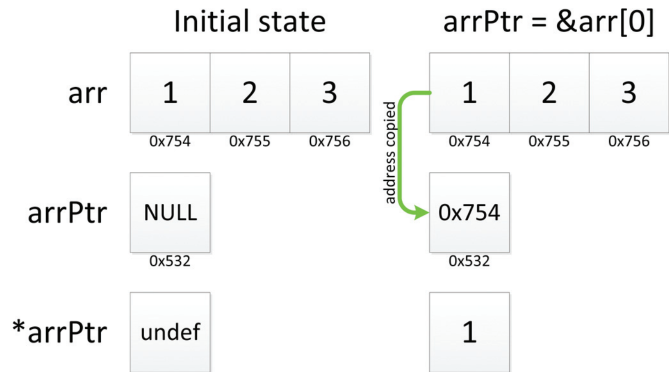
```
// ptr tutorial example 3
task main()
{
  ubyte arr[3] = {1, 2, 3};
  ubyte *arrPtr;

  arrPtr = &arr[0];  // arrPtr now points to the
address of arr[0]

  // This should print out
  // arr[0]: 1, arrPtr: 656F6968, *arrPtr: 1
  // (value for arrPtr may vary)
  writeDebugStreamLine("arr[0]: %d, arrPtr: %p,
*arrPtr: %d", arr[0], arrPtr, *arrPtr);

  arrPtr++;  // we're now pointing at arr[1]

  // This should print out
  // arr[1]: 2, arrPtr: 656F6969, *arrPtr: 2
  // (value for arrPtr may vary)
  writeDebugStreamLine("arr[1]: %d, arrPtr: %p,
*arrPtr: %d", arr[1], arrPtr, *arrPtr);
}
```

That should print out something like this in the Debug Stream:

```
arr[0]: 1, arrPtr: 656F6968, *arrPtr: 1
arr[1]: 2, arrPtr: 656F6969, *arrPtr: 2
```

Please note that the address arrPtr may be different in your case. It is important that on the second line, the address arrPtr points to is one higher than in the first one.

When you look at what it looks like in memory, you can get a good idea of what's going on:



INTERESTING FACT ABOUT POINTERS

When you use sizeof() on a normal variable, like a ubyte, int or long, you get the number of bytes this type takes up. In the case of a ubyte, that's one byte, int takes up two and a long uses four. Doing a "size-of" of a pointer is, well, pointless. You won't be getting the size of the item you're pointing at, it's always 4 in the case of ROBOTC. That's because the address stored in the pointer is a 4 byte number, a long. Keep that in mind when you want to use sizeof() to get the number of bytes you want to wipe with a memset(), for example.

PROJECT: DATALOGGING

If you've ever built a robot, may have found that it doesn't always do what you think it ought to do. Reality has a nasty habit of messing with your perfectly programmed robot. To deal with this, we have two choices: either change reality (not an easy task) or find out what's going on and change the behaviour of our robot. Sounds