ECE 573
L-values, R-values and expressions

In class today, we were discussing how to handle the distinction between L-values and R-values when processing expressions. The short rule is this:

> *If the value is definitely going to be used as an address, treat it as an L-value, otherwise it's an R-value*

Let's see how this applies to various constructs.

- **Variables/identifiers** When they are first encountered, the compiler does not know if variables are going to appear on the left side or right side of an assigment. However, what the compiler does know is that the variable will be used as an address, either to load from if it will ultimately be used as an R-value, or to store to, if it will ultimately be used as an L-value. Thus, we know the variable will definitely be used as an address first, so it is an **L-value**.

- **Expressions** Because expressions always operate on values and produce new values, they are **R-values**.

- $*$ **operator** This one is tricky. The $*$ operator says to treat its operand (which is itself an expression) as an address. We will then load from the address if the overall expression is on the right hand side, or load from it if the it is on the left hand side.

  First off, note that the $*$ operator itself operates on R-values. In a sense, the purpose of the operator is to turn R-values into L-values. This means that if the operand is an L-value, we must load from it to generate the R-value. (Think about what `*x` means: dereference whatever address is *stored in* x)

  Second, we note that after applying the $*$ operator, the result of the expression is now an address—and it is meant to be used as an address. This means that according to the rule, we will treat it as an **L-value**.

- & **operator** This one is also tricky. the & operator returns the address of a variable or any other L-valued expression. So what is that address? Well, it's just a value to begin with (and a constant, at that!). Which means that unless it becomes the operand of a $*$ operator, we don't know that it's going to be used as an address. It's an **R-value**.

Aside: converting between L- and R-values If an L-value is seen where a compiler expects an R-value, the *compiler* converts the L-value into the necessary R-value by *loading from it*. If an R-value needs to be used as an L-value (*i.e.*, it needs to be treated as an address), the *program* can convert the R-value into the appropriate L-value by applying the $*$ operator. Note that the compiler never turns R-values into L-values on its own.

Generating code for $*$ Recall that $*$ operates on R-values and turns them in to L-values. This means generating code is easy. If the temporary for the operand expression is an L-value, we need to turn it into an R-value, by loading from it (in exactly the same way that we load from L-valued variables when they're used in an expression). If the operand expression is already an R-value, we don't need to generate any more code. Now that we have the R-value, we just want the rest of the code to treat it as an address that can be loaded from or stored to, so we simply mark the temporary as an L-value. The rest of the code generation routines will do the right thing. Let's walk through a few examples to see how this works.

- `*(p)`

  1. First, we create the data object for `p`. There is no code here; we simply mark `p` as an L-value:

Code:	
Temp:	`p`
Type:	L-value

  2. This data object gets passed up to the routine that generates code for $*$. Because we received an L-value, we must first load from it to get the R-value, and then tell the rest of the code generation routines to treat the R-value as an address by marking the temporary as an L-value.

Code:	`LD p, t1`
Temp:	`t1`
Type:	L-value

  3. Now wherever `*(p)` gets used, the code generation routines will do the right thing. If it's on the left-hand-side of an assignment, we will store to `t1`. If it is used in another expression, or is on the right-hand-side of an assignment, we will load from it.

- `*(p + 1)`

  1. First, we create the data objects for `p` and `1`:

Code:	
Temp:	`p`
Type:	L-value

Code:	MV 1, t1
Temp:	t1
Type:	Const

2. Then, we generate the code for `p + 1`. Note that because `p` is an L-value, we must load from it, and because `1` is a constant, we can use the value directly, rather than the generated code. This creates an R-value:

Code:	`LD p, t2`
	`ADD t2, 1, t3`
Temp:	`t3`
Type:	R-value

3. Finally, we generate code for `*(p + 1)`. Because $*$ is operating on an R-value, note that we do not have to do any extra loading.

Code:	`LD p, t2`
	`ADD t2, 1, t3`
Temp:	`t3`
Type:	L-value

Interestingly, all we had to do here was tell the compiler to start treating `t3` as if it had an address in it; as if it were an L-value.

- `*(&x)`

1. Let us assume there is an instruction `AOF` which will move the address of a variable into a temporary. First we generate the code for `x`:

Code:	
Temp:	`x`
Type:	L-value

2. Then we generate the code for `&x`

Code:	`AOF x, t1`
Temp:	`t1`
Type:	R-value

Remember, the result of the & operator is an R-value.

3. Then we generate the code for `*(&x)`. Because `&x` is an R-value, we only need to tell the compiler to treat it as an L-value:

Code:	`AOF x, t1`
Temp:	`t1`
Type:	L-value

Compare the result of this to the code we generated for just `x`. Recall that `AOF`'s only purpose is to move the address of `x` into `t1`, which means that loading from `t1` is the same as loading from `x`. Thus, the two pieces of generated code do the same thing, which is what we would expect, since `*(&x) == x`