Code Generation: Expressions With Pointers

Pointer Operations

- We've seen how to generate code for addition, and can extend the same strategy to subtraction, multiplication, division, and modulo.
- WLP4 also allows **pointer operations**: dereference and address-of.
- The *new* operator (memory allocation) can also appear in expressions, but we'll discuss that later alongside *delete* statements.
- Our handling of dereference and address-of turns out to be tied to our handling of assignment statements, so we'll also discuss those.

Pointer Dereference

- There are two rules for pointer dereference:
  - factor → STAR factor
  - lvalue → STAR factor
- Recall that an **lvalue** is a special kind of expression that refers to a memory location in which a value can be stored.
  - The 'L' in lvalue comes from the fact that the expression on the left-hand side of an assignment statement must be an lvalue.
- So when generating code for the second rule, we have to consider that the dereference might be occurring in a context like this:

  *ptr = value;

Address-Of

- The address-of operator corresponds to the following rule:
  - factor → AMP lvalue (where AMP is ampersand, the & symbol)
- Notice lvalues appear again. It does not make sense to take the "address of" something unless it is a memory location.
- The only two contexts in which lvalues appear in WLP4 are in address-of expressions, and in assignment statements:
  - statement → lvalue BECOMES expr SEMI
- When implementing code generation for lvalue → STAR factor, we need to consider both of these contexts.

Other Lvalues

- There are actually three lvalue rules:
  - lvalue → ID (an lvalue can be a variable)
  - lvalue → STAR factor (an lvalue can be a dereferenced pointer expression)
  - lvalue → LPAREN lvalue RPAREN (an lvalue can be wrapped in parentheses)
- The third case does not change the meaning of the lvalue. It is just expressing the fact that we can wrap expressions in parentheses.
- So, there are fundamentally two kinds of lvalues we need to consider, variables and dereferenced pointer expressions.
- There are two contexts where they can appear, address-of expressions and assignment statements.

Code Generation Strategies For Lvalues

	statement → lvalue BECOMES expr SEMI	factor → AMP lvalue
lvalue → ID	?	?
lvalue → STAR factor	?	?

Address Of A Variable

	statement → lvalue BECOMES expr SEMI	factor → AMP lvalue
lvalue → ID	?	Goal: Return the address of the ID in $3
lvalue → STAR factor	?	?

Address Of A Variable

	statement → lvalue BECOMES expr SEMI	factor → AMP lvalue
lvalue → ID	?	Goal: Return the address of the ID in $3
lvalue → STAR factor	?	?

Hint: Variable locations are determined using offsets from the frame pointer!

Address Of A Variable

	statement → lvalue BECOMES expr SEMI	factor → AMP lvalue

lvalue → ID	?	`lis $3`
		`.word <offset of ID>`
		`add $3, $29, $3`

lvalue → STAR factor	?	?

Address Of A Dereferenced Pointer

	statement → lvalue BECOMES expr SEMI	factor → AMP lvalue

lvalue → ID	?	`lis $3`
		`.word <offset of ID>`
		`add $3, $29, $3`

lvalue → STAR factor	?	?

Address Of A Dereferenced Pointer

	statement → lvalue BECOMES expr SEMI	factor → AMP lvalue

lvalue → ID	?	`lis $3`
		`.word <offset of ID>`
		`add $3, $29, $3`

lvalue → STAR factor	?	?

Hint 1: This is something like
&*ptr or &*(array+i)

Address Of A Dereferenced Pointer

	statement → lvalue BECOMES expr SEMI	factor → AMP lvalue

lvalue → ID	?	`lis $3`
		`.word <offset of ID>`
		`add $3, $29, $3`

lvalue → STAR factor	?	?

Hint 2: Address-of and dereference are "inverse" operations (they cancel each other out)

Address Of A Dereferenced Pointer

	statement → lvalue BECOMES expr SEMI	factor → AMP lvalue

lvalue → ID	?	`lis $3`
		`.word <offset of ID>`
		`add $3, $29, $3`

lvalue → STAR factor	?	`code(AMP STAR factor)`
		`= code(factor)`
		(Just generate code for the pointer expression being dereferenced!)

Assignment To A Variable

	statement → lvalue BECOMES expr SEMI	factor → AMP lvalue

lvalue → ID	?	`lis $3`
		`.word <offset of ID>`
		`add $3, $29, $3`

lvalue → STAR factor	?	`code(AMP STAR factor)`
		`= code(factor)`
		(Just generate code for the pointer expression being dereferenced!)

Assignment To A Variable

	statement → lvalue BECOMES expr SEMI	factor → AMP lvalue

lvalue → ID	?	`lis $3`
		`.word <offset of ID>`
		`add $3, $29, $3`

lvalue → STAR factor	?	`code(AMP STAR factor)`
		`= code(factor)`
		(Just generate code for the pointer expression being dereferenced!)

- There are two approaches here which we'll call the "Direct Approach" and "Indirect Approach":
  - Indirect Approach: Reuse the address-of logic to implement assignment in a uniform way (compute the address of the lvalue, store the expr in that address).
  - Direct Approach: Implement assignment differently depending on what kind of lvalue we are dealing with (two different strategies for variables and dereferenced pointers).

Assignment: Indirect Approach

	statement → lvalue BECOMES expr SEMI	factor → AMP lvalue

lvalue → ID	addressOf(lvalue) ; places address in $3
	add $5, $3, $0
	code(expr) ; places expr value in $3
	sw $3, 0($5)
		lis $3
		.word <offset of ID>
		add $3, $29, $3

lvalue → STAR factor	?	code(AMP STAR factor) = code(factor) (Just generate code for the pointer expression being dereferenced!)

- Is this correct for the Indirect Approach?

Assignment: Indirect Approach

	statement → lvalue BECOMES expr SEMI	factor → AMP lvalue

lvalue → ID	`addressOf(lvalue) ; places address in $3`
	`add $5, $3, $0`
	`code(expr) ; places expr value in $3`
	`sw $3, 0($5)`
		`lis $3`
		`.word <offset of ID>`
		`add $3, $29, $3`

lvalue → STAR factor	?	`code(AMP STAR factor)`
		`= code(factor)`
		(Just generate code for the pointer expression being dereferenced!)

- Is this correct for the Indirect Approach?
- **No**, because our strategy for code(expr) uses $5 to hold values popped from the stack.

Assignment: Indirect Approach

	statement → lvalue BECOMES expr SEMI	factor → AMP lvalue
lvalue → ID	addressOf(lvalue) ; places address in $3 **push $3 to stack** code(expr) ; places expr value in $3 **pop into $5** sw $3, 0($5)	lis $3 .word <offset of ID> add $3, $29, $3
lvalue → STAR factor	?	code(AMP STAR factor) = code(factor) (Just generate code for the pointer expression being dereferenced!)

- Is this correct for the Indirect Approach?
- **No**, because our strategy for code(expr) uses $5 to hold values popped from the stack.
- We could use a different register (not $5) but this is a little sketchy, and might not work depending on how we implement procedures.
- Let's **use the stack ourselves** to preserve the lvalue address!

Assignment: Indirect Approach

	statement → lvalue BECOMES expr SEMI	factor → AMP lvalue

lvalue → ID	`addressOf(lvalue) ; places address in $3`
	`push $3 to stack`
	`code(expr) ; places expr value in $3`
	`pop into $5`
	`sw $3, 0($5)`
		`lis $3`
		`.word <offset of ID>`
		`add $3, $29, $3`

lvalue → STAR factor	`addressOf(lvalue) ; places address in $3`
	`push $3 to stack`
	`code(expr) ; places expr value in $3`
	`pop into $5`
	`sw $3, 0($5)`
		`code(AMP STAR factor)`
		`= code(factor)`
		(Just generate code for the pointer expression being dereferenced!)

- The Indirect Approach uses the same logic for assignment in all cases.
- The only difference is what code gets generated by "addressOf(lvalue)".

Assignment To A Variable: Direct Approach

	statement → lvalue BECOMES expr SEMI	factor → AMP lvalue

lvalue → ID	?	`lis $3`
		`.word <offset of ID>`
		`add $3, $29, $3`

lvalue → STAR factor	?	`code(AMP STAR factor)`
		`= code(factor)`
		(Just generate code for the pointer expression being dereferenced!)

- How could we implement assignment to a variable more directly, without relying on address-of?

Assignment To A Variable: Direct Approach

	statement → lvalue BECOMES expr SEMI	factor → AMP lvalue

lvalue → ID	?	`lis $3`
		`.word <offset of ID>`
		`add $3, $29, $3`

lvalue → STAR factor	?	`code(AMP STAR factor)`
		`= code(factor)`
		(Just generate code for the pointer expression being dereferenced!)

- How could we implement assignment to a variable more directly, without relying on address-of?
- Hint: After computing code(expr), only one more instruction is needed.

Assignment To A Variable: Direct Approach

	statement → lvalue BECOMES expr SEMI	**factor → AMP lvalue**

lvalue → ID	`code(expr) ; $3 = expr value`
	`sw $3, <offset of ID>($29)`
		`lis $3`
		`.word <offset of ID>`
		`add $3, $29, $3`

lvalue → STAR factor	?	`code(AMP STAR factor)`
		`= code(factor)`
		(Just generate code for the pointer expression being dereferenced!)

Assignment To Dereference: Direct Approach

	statement → lvalue BECOMES expr SEMI	factor → AMP lvalue

lvalue → ID	`code(expr) ; $3 = expr value`
	`sw $3, <offset of ID>($29)`
		`lis $3`
		`.word <offset of ID>`
		`add $3, $29, $3`

lvalue → STAR factor	?	`code(AMP STAR factor)`
		`= code(factor)`
		(Just generate code for the pointer expression being dereferenced!)

- What about assignment to a dereferenced pointer expression?
- Hint: This ends up being pretty similar to the indirect approach.

Assignment To Dereference: Direct Approach

	statement → lvalue BECOMES expr SEMI	factor → AMP lvalue

lvalue → ID	`code(expr) ; $3 = expr value`
	`sw $3, <offset of ID>($29)`
		`lis $3`
		`.word <offset of ID>`
		`add $3, $29, $3`

lvalue → STAR factor	?	`code(AMP STAR factor)`
		`= code(factor)`
		(Just generate code for the pointer expression being dereferenced!)

- What about assignment to a dereferenced pointer expression?
- Hint: This ends up being pretty similar to the indirect approach.

Assignment To Dereference: Direct Approach

	statement → lvalue BECOMES expr SEMI	factor → AMP lvalue

lvalue → ID	`code(expr) ; $3 = expr value`
	`sw $3, <offset of ID>($29)`
		`lis $3`
		`.word <offset of ID>`
		`add $3, $29, $3`

lvalue → STAR factor	`code(factor) ; places factor value in $3`
	`push $3 to stack`
	`code(expr) ; places expr value in $3`
	`pop into $5`
	`sw $3, 0($5)`
		`code(AMP STAR factor)`
		`= code(factor)`
		(Just generate code for the pointer expression being dereferenced!)

- The value of the dereferenced "factor" is the address we want to assign to.
- Since this could be a complex expression itself, we can't really take any shortcuts in computing it.

The Other Kind Of Pointer Dereference

- We now know how to handle the address-of operator, the assignment statement, and how to handle pointer dereferences in these contexts.
- …We never explained how to handle an ordinary pointer dereference though (the factor → STAR factor) rule.
- This rule is used when a pointer dereference appears in an expression context other than address-of or assignment, for example:

  return 1 + *array; // returns 1 plus the first element of the array

- Rather than producing the address of the pointer expression, we want to produce the value being pointed to.

The Other Kind Of Pointer Dereference

- What should our code generation strategy be for the rule factor → STAR factor?
- Hint 1: Start by generating code for the factor on the right-hand side.

  code(factor) ; places the value of the factor in $3
- What do we do after that?
- Hint 2: When you dereference a pointer, what value is returned?

The Other Kind Of Pointer Dereference

- What should our code generation strategy be for the rule factor → STAR factor?

```
code(factor) ; places the value of the factor in $3
lw $3, 0($3)
```

The Other Kind Of Pointer Dereference

- What should our code generation strategy be for the rule factor → STAR factor?

```
code(factor) ; places the value of the factor in $3
lw $3, 0($3)
```

- The value of the factor is the *address being pointed to*.
- Dereferencing a pointer *retrieves the value at this address*.
- That is exactly what the Load Word instruction is for.

NULL Pointers

- NULL is traditionally represented as 0 because 0 is normally not an accessible address.
- But in our simplified CS241 environment, 0 *is* a valid address and we often load code at this address.

```
code(factor) ; what if this puts 0 in $3?
lw $3, 0($3)
```

- Dereferencing NULL would *produce the first word of our code* and the program would continue on.
- This is bad, because we want dereferencing NULL to be an error!

NULL Pointers

- For this reason, our WLP4 to MIPS compiler will represent NULL using the value 1 instead of 0.
- Recall that in MIPS, you can only access addresses that are multiples of 4, so accessing address 1 with lw or sw will crash immediately.
- This is actually what we want (immediate crash when we try to load from or store to NULL).
- When dealing with these rules, use 1 for the value of NULL:
  - factor → NULL (NULL in expressions)
  - dcls → dcls dcl BECOMES NULL SEMI (initialization to NULL)

Pointer Arithmetic

- Four forms of pointer arithmetic are supported:
  - int* + int (adding an integer offset to a pointer)
  - int + int* (same as above, but with order changed)
  - int* – int (subtracting an integer offset from a pointer)
  - int* – int* (subtracting two pointers to find the “distance” between them)
- No other combinations are valid according to the type rules, other than arithmetic involving two ints.
- Once again, we need the type information to decide what to do, because each of these type combinations works differently from the case of two ints.

Adding or Subtracting an Offset

- For these type combinations:
  - int* + int (adding an integer offset to a pointer)
  - int + int* (same as above, but with order changed)
  - int* – int (subtracting an integer offset from a pointer)
- When adding or subtracting an offset, the offset should be multiplied by the *size of the pointed-to type*.
- For example, when we do “array + 2”, we want to access the element at index 2 of the array.
- But if each array element is 4 bytes, we have to add **8** to the starting address of the array.

Adding or Subtracting an Offset

- For these type combinations:
  - int* + int (adding an integer offset to a pointer)
  - int + int* (same as above, but with order changed)
  - int* – int (subtracting an integer offset from a pointer)
- When adding or subtracting an offset, the offset should be multiplied by the *size of the pointed-to type*.
- In WLP4, we only have pointers to ints, and ints are 4 bytes.
- So in all of the above cases, we must multiply the value of the int expression by 4 before adding it to or subtracting it from the address.

Adding or Subtracting an Offset

- Example: expr → expr PLUS term
  - int* + int (adding an integer offset to a pointer)

```
Generate code for the expr (address).
Push $3 to the stack.
Generate code for the term (offset).
mult $3, $4 ← Assuming $4 contains 4
mflo $3
Pop from the stack into $5.
add $3, $5, $3
```

- Pay close attention to the order of things to make sure you do not mix up the offset and address.

Subtracting Two Pointers

- This operation is a little obscure, but allowed in C/C++.
  - int* – int* (subtracting two pointers to find the “distance” between them)
- For example, if you have two pointers into an array, subtracting the smaller one from the larger one should produce the number of elements in between (inclusive of the smaller, exclusive of the larger).
  - “(array + index) – array” should produce “index”, which is the number of elements between a[0] and a[index], not including a[index] itself.
- This “distance” is signed though: subtracting the larger from the smaller should produce a *negative value*.
  - “array – (array + index)” should produce “-index”, i.e., index negated.

Subtracting Two Pointers

- This operation is a little obscure, but allowed in C/C++.
  - int* – int* (subtracting two pointers to find the “distance” between them)
- For an array of 4-byte ints, the address of a[i] is actually:

  (address of a) + (4*i)
- So for “(a+i) – a” if we just do subtraction, the value we actually get is:

  (address of a) + (4*i) – (address of a) = (4*i) !!
- To correct the value, we need to *divide* the result of subtraction by the size of the pointed-to type.
- In WLP4, we only have pointers to 4-byte ints, so we divide by 4.

Subtracting Two Pointers

- This operation is a little obscure, but allowed in C/C++.
  - int* – int* (subtracting two pointers to find the “distance” between them)
- In summary, when doing subtraction, check the type of *both* the left and right subexpressions. If *both* are pointers, divide the result by 4.

```
Generate code for the left subexpression.
Push $3 onto the stack.
Generate code for the right subexpression.
Pop from the stack into $5.
sub $3, $5, $3
div $3, $4
mflo $3
```

Code Generation: Statements

Statements in WLP4

- Assignment statements. (Done)
  - statement → lvalue BECOMES expr SEMI
- While loops.
  - statement → WHILE LPAREN test RPAREN LBRACE statements RBRACE
- Conditional (if/else) statements.
  - statement → IF LPAREN test RPAREN LBRACE statements RBRACE ELSE LBRACE statements RBRACE
- Printing integers.
  - statement → PRINTLN LPAREN expr RPAREN SEMI
- Deleting arrays.
  - statement → DELETE LBRACK RBRACK expr SEMI

While Loops

- statement → WHILE LPAREN test RPAREN LBRACE statements RBRACE
- Generate code for the **test** (comparison between two expressions).
- Depending on the result of the comparison, either enter the loop body (and run the code for the **statements**) or end the loop.
- We'll leave the actual MIPS code as an exercise (you should have written many MIPS loops at this point!)
- One issue is that MIPS loops normally involve **labels**.
- If there are multiple while loops in our program, all the labels we generate must have unique names!

Generating Unique Labels

```
while(...) { ... }
while(...) { ... }
```

- We cannot use the same label names for both loops.
- The basic idea is to append a unique number to the label name, e.g., by maintaining a global counter.

```
loop1:
...
end1:
loop2:
...
end2:
```

Generating Unique Labels

- Be careful about how you manage the counter.

```
while(...) { while(...) { ... } }
```

- The labels for the outer loop should *all use the same counter value*. If you aren't careful, you might end up with errors like this:

```
loop1:  ; outer loop start
...
 loop2: ; inner loop start, counter is incremented
 ...
 end2:  ; inner loop end
...
end2:   ; outer loop end, using wrong counter value!
```

Generating Unique Labels

- Be careful about how you manage the counter.

```
while(...) { while(...) { ... } }
```

- Generate all labels for a loop using the same counter before you generate any nested loops (which may increment the counter).

```
loop1:  ; outer loop start
...
 loop2: ; inner loop start
 ...
 end2:  ; inner loop end
...
end1:   ; outer loop end
```

Comparison Tests

- There are six comparison test rules:
  - test → expr LT expr (less than)
  - test → expr GT expr (greater than)
  - test → expr LE expr (less than or equal)
  - test → expr GE expr (greater than or equal)
  - test → expr EQ expr (equal)
  - test → expr NE expr (not equal)
- Use the same strategy as binary operations like addition.
  - Compute left, push, compute right, pop, compare (producing 0 or 1 in $3).
- LT and GT are straightforward to implement with slt. The others require a bit of extra math, but are not difficult.

Pointer Comparisons

- Pointers can be compared to determine which address is larger.
- It might seem like there is no difference from integer comparisons, but there is a minor one.
- In WLP4, int is a *signed type*, but memory addresses cannot be negative, so address comparisons should be *unsigned*.
- If signed comparison is used, large-enough memory addresses will be interpreted as negative two's complement numbers!
- For example, 0x7FFFFFFF < 0xFFFFFFFF but in two's complement, this is comparing $2^{31} - 1$ to -1, and the result is reversed.

Pointer Comparisons

- For pointer comparisons, *unsigned* comparison should be used instead of *signed* comparison.
- That is, the MIPS instruction **sltu** should be used instead of **slt**.
- To decide which instruction to use, we need the *type information* computed in the semantic analysis phase!
- All comparison rules look like:
  - test → expr OP expr (where OP is some comparison operator)
- Examine the type of one of the exprs to decide which instruction to use. (Type checking guarantees both exprs have the same type!)

Statements in WLP4

- Assignment statements. (Done)
  - statement → lvalue BECOMES expr SEMI
- While loops. (Done)
  - statement → WHILE LPAREN test RPAREN LBRACE statements RBRACE
- Conditional (if/else) statements. (Similar to while, left as exercise)
  - statement → IF LPAREN test RPAREN LBRACE statements RBRACE ELSE LBRACE statements RBRACE
- Printing integers.
  - statement → PRINTLN LPAREN expr RPAREN SEMI
- Deleting arrays.
  - statement → DELETE LBRACK RBRACK expr SEMI

External Libraries

- In Question 6, you wrote a print procedure.
- You could implement the **println** statement by including a copy of this procedure with every program you generate that uses println.
- However, complicated procedures like this are often stored externally and combined with the generated code using **linking!!**
- Next time, we'll discuss how to import and call external procedures for printing and memory management.
- We'll also discuss how to generate code for non-wain procedures and how to call those procedures.