Pointers
what are pointers?

• Up until now, we have only considered variables that represent values
  • A variable is a named box in memory that contains a value

```c
int x
```
what are pointers?

• Up until now, we have only considered variables that represent values
  • A variable is a named box in memory that contains a value

• But what if the box can contain the address of another box?

```
int x
```

```
int * y
```

```
int x
```
what are pointers?

• Up until now, we have only considered variables that represent values
  • A variable is a named box in memory that contains a value

  ![Diagram of a variable](int x)

• But what if the box can contain the address of another box?

  ![Diagram of a pointer](int * y int x)
what are pointers?

- Up until now, we have only considered variables that represent values
  - A variable is a named box in memory that contains a value
- But what if the box can contain the address of another box?
pointers vs references

• A **pointer** is a variable that holds an *address*
• That address can be treated as a *value* that can be computed over

```
int * p = &a  //p gets the address of a
int * q = p + 1  //q gets 4 + the address of a
```

• Some languages only have **references** instead of pointers
  • A reference *refers* to another memory location (under the hood: holds the address of another location in memory)
  • But cannot do pointer arithmetic
two key operations

- Two new **unary** operations:
  - & : address-of operation
    - Returns the address of a variable
      
      ```
p = &a //store the address of a in p
      ```
  - * : pointer dereference operation
    - Lets you load from, or store to, a pointer
      
      ```
P = 7 //store to the address stored in p
x = * p //load from the address stored in p
      ```
pointer types

• How do we build pointers into our type system?
• Can think of types as being defined by a grammar!

\[
T \rightarrow \text{int} \mid \text{float} \\
T \rightarrow \ast T
\]

• Type is either a base type or a pointer to another type
typing * and &

• What are the type rules for our two unary operations?

• * expr:

  If expr has type *T then * expr has type T

• & expr:

  If expr has type T then & expr has type *T
Code Generation for Pointers
l-values vs r-values

• Remember the distinction between **l-values** and **r-values**:
  • L-value: an address that can be loaded from or stored to
  • R-value: a piece of data that can be computed with

• Up until now, the only l-values we have had are variables (global variables, local variables)
l-values, r-values, and pointers, oh my!

- Semantically, what do & and * do?
- Convert between l-values and r-values!
- **Address-of operator**: take an l-value (an address) and treat it as an r-value (a piece of data)

  - `& x + 1`  
    take the *address* of `x`, treat it as a piece of data, and add 4 to it

  - `x + 1`  
    take the *value in x*, then add 1 to it
l-values, r-values, and pointers, oh my!

• Semantically, what do & and * do?
• Convert between l-values and r-values!
• De-reference operator: take an r-value (a piece of data) and treat it as an l-value (an address)

\[ * \ (x + 1) \]

- take the value in x, add 4 to it, then treat the result as an address so you can load from it or store to it

\[ x + 1 \]

- take the value in x, then add 1 to it

• Note that if the expression passed to * is an l-value, you load from it first to get an r-value, just like before
Pointer Codegen Example
code generation (assembly)

- Code generation in assembly is easy: keep the same CodeObject, but switch whether temporary is an l-val or an r-val
code generation (assembly)

\[ \ast x = \ast (y + 7) \]
code generation (assembly)

• $\star x = \star (y + 7)$
code generation (assembly)

\[ \ast \ x = \ast (y + 7) \]
code generation (assembly)

• \( *x = *(y + 7) \)
code generation (assembly)

\[ *x = * (y + 7) \]
code generation (assembly)

\[ *x = *(y + 7) \]

L/R: L
Var: t1
Code: `lw t1, 0x4004`

L/R: ---
Tmp: ---
Code:
`lw t1, 0x4004
lw t2, 0x4000
addi t3, t2, 28
lw t4, 0(t3)
sw t4, 0(t1)`

L/R: L
Tmp: t3
Code:
`lw t2, 0x4000
addi t3, t2, 28`
code generation (IR)

• Code generation for IR is similar

• Track whether IR temporary holds an l-value or an r-value (e.g., use ‘$’ as prefix for r-value, ‘@’ as prefix for l-value)

• Introduce two new IR nodes:
  
  • ADDROF a, b : store the address of operand b in a  
    (if b is a variable, a holds the variable address; if b is an l-value temporary, a is the temporary, just as an r-value)

  • DEREF a, b : store the value of operand b in a as an address  
    (if b is a variable or an l-value temporary, load from b and store the result in a as an l-value; if b is an r-value temporary, a is the temporary, just as an l-value)
register allocation

• Now that we have pointers, we have aliasing!

• Simple solution: treat all locals/globals as aliased to each other: cannot stay in registers. Write back on every store, free after every load

• Slightly more complicated: only variables that have ever had an ADDROF operation applied to them can be aliased

• More complex: perform **pointer analysis** (stay tuned!)
Memory Allocation
reserving space in memory

• How do we decide what address to put in a pointer?
• Can point to the address of an existing variable

\[ p = \& x \]

• Means addresses point either to:
  • global memory segment (global variables)
  • stack (local variables)

• Can we point elsewhere?
program heap

- Memory space of executing program also contains a large region called the **heap**

- Used for *dynamically allocated* data
  - Data not associated with a local variable or a global variable
  - Pointed to by pointers
  - No fixed location in memory

- How do we allocate that?
malloc/free

- `malloc(n)` : allocate (reserve) `n` bytes of data in the heap, return the *address* of the first byte of the allocated region

- `free(a)` : free the allocated region at address `a`
• `malloc(n)` : allocate (reserve) $n$ bytes of data in the heap, return the address of the first byte of the allocated region

```
x = malloc(10)
```

- 10 bytes

• `free(a)` : free the allocated region at address $a$
malloc/free

- `malloc(n)`: allocate (reserve) \( n \) bytes of data in the heap, return the \textit{address} of the first byte of the allocated region

\[
x = \text{malloc}(10)
y = \text{malloc}(8)
\]

- `free(a)`: free the allocated region at address \( a \)
malloc/free

- `malloc(n)`: allocate (reserve) $n$ bytes of data in the heap, return the address of the first byte of the allocated region

  $$x = \text{malloc}(10)$$
  $$y = \text{malloc}(8)$$

- `free(a)`: free the allocated region at address $a$

  `free(x)`
malloc/free

• `malloc(n)` : allocate (reserve) `n` bytes of data in the heap, return the address of the first byte of the allocated region

  \[
  \begin{align*}
  x &= \text{malloc}(10) \\
  y &= \text{malloc}(8)
  \end{align*}
  \]

• `free(a)` : free the allocated region at address `a`

\[
\begin{align*}
\text{free}(x) \\
\text{free}(y)
\end{align*}
\]
malloc/free

• malloc(n) : allocate (reserve) \(n\) bytes of data in the heap, return the address of the first byte of the allocated region

\[
\begin{align*}
x &= \text{malloc}(10) \\
y &= \text{malloc}(8)
\end{align*}
\]

• free(a) : free the allocated region at address \(a\)

\[
\begin{align*}
\text{free}(x) \\
\text{free}(y)
\end{align*}
\]

• Guarantee: malloc will not return a region that overlaps with a current allocation
implementing malloc and free

• Implementation of memory allocator (malloc/free) is the responsibility of the operating system or the virtual machine

• Language usually provides a standard library that interfaces with the operating system to perform memory allocation

• In our course project, we don’t have a standard library or an operating system
  • But the RISC simulator is essentially a virtual machine!

• malloc/free implemented as “magic” instructions in the simulator
  • Compiler should detect invocations of malloc/free and generate magic instructions
next: arrays