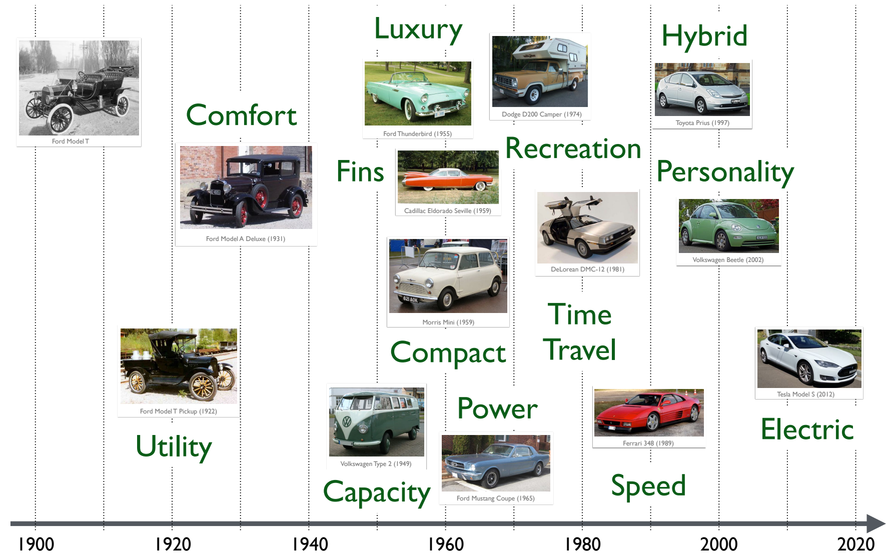
# Why Modern Programming Languages Matter

Mark P Jones, Portland State University

Winter 2017



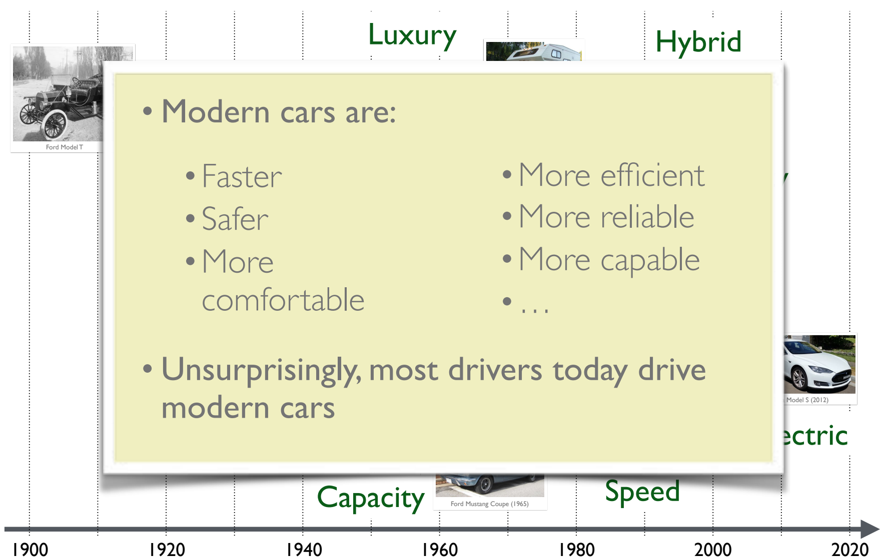
## A short history of the automobile




(Images via Wikipedia, subject to Creative Commons and Public Domain licenses)



## A short history of the automobile

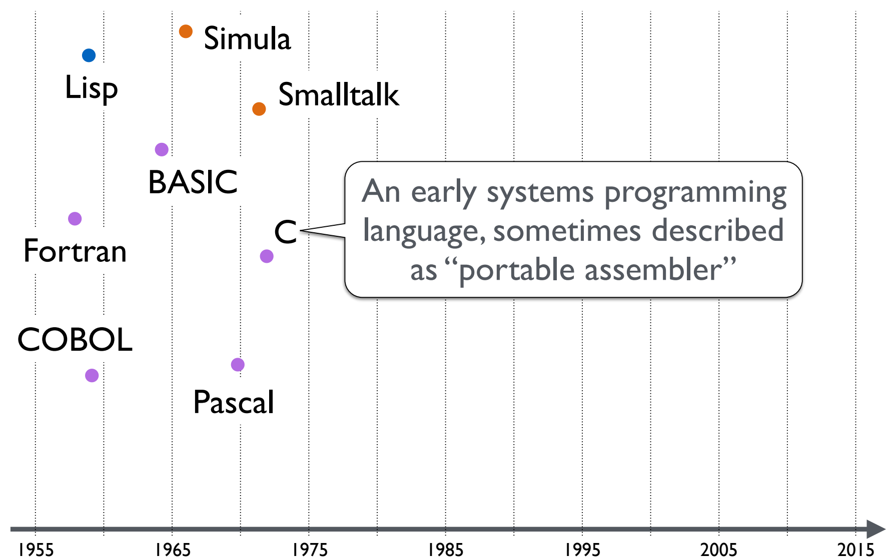



- Modern cars are:
  - Faster
  - Safer
  - More comfortable
  - More efficient
  - More reliable
  - More capable
  - …
- Unsurprisingly, most drivers today drive modern cars

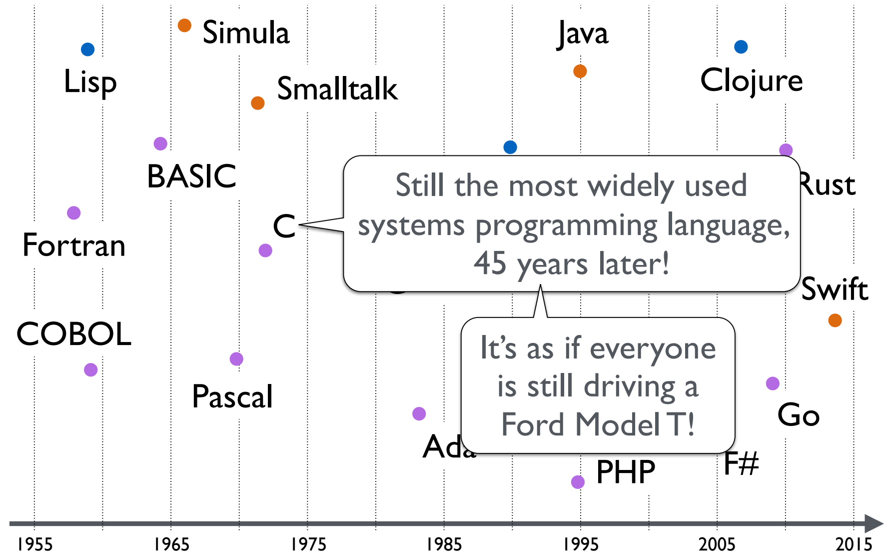
(Images via Wikipedia, subject to Creative Commons and Public Domain licenses)



## A short history of programming languages

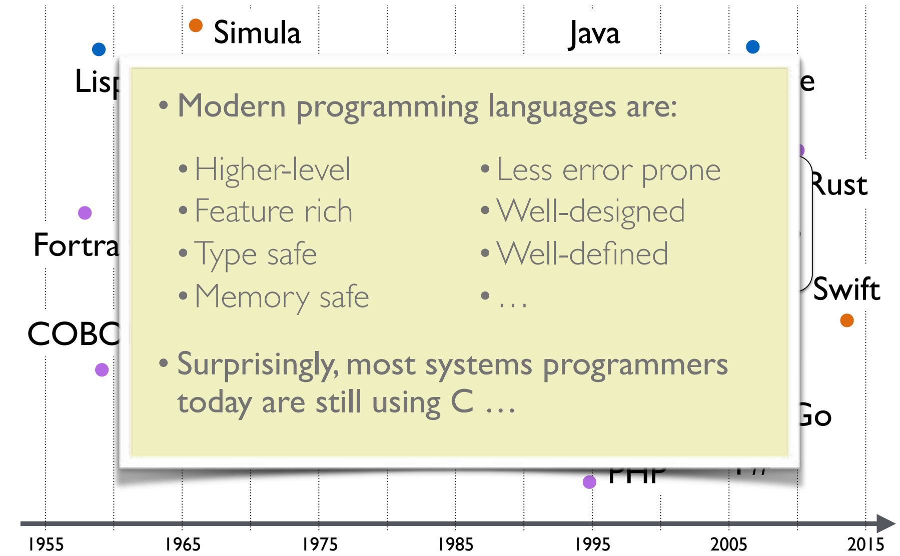


# A short history of programming languages






# A short history of programming languages




- Modern programming languages are:
  - Higher-level
  - Feature rich
  - Type safe
  - Memory safe
  - Less error prone
  - Well-designed
  - Well-defined
  - …
- Surprisingly, most systems programmers today are still using C …



# C is great … what more could you want?

- Programming in C gives systems developers:
  - Good (usually predictable) performance characteristics
  - Low-level access to hardware when needed
  - A familiar and well-established notation for writing imperative programs that will get the job done
- What can you do in modern languages that you can't already do with C?
- Do you really need the fancy features of newer object-oriented or functional languages?
- Are there any downsides to programming in C?



Impact: An application may be able to execute arbitrary code with kernel privileges

Description: Multiple memory corruption issues were addressed through improved input validation.

Impact: An application may be able to exec

Description: A use after free issue was add management.

Impact: An application may be able to execu

Description: A null pointer dereference was addressed through improved input validation.

Impact: A local user may be able to gain root privileges

Description: A type confusion issue was addressed through improved memory handling.

Impact: An application may be able to execute arbitrary code

Description: An out-of-bounds write issue was addressed by removing the vulnerable code.

Could a different language make it **impossible** to write programs with errors like these ?

# The Habit programming language

- "a dialect of Haskell that is designed to meet the needs of high assurance systems programming"

**Habit** = **Ha**skell + **bit**s

- Habit, like Haskell, is a functional programming language
- For people trained in using C, the syntax and features of Habit may be unfamiliar
- I won't assume familiarity with functional programming here
- We'll focus on how Habit uses **types** to detect and **prevent** common types of programming errors



# Division

- You can divide an integer by an integer to get an integer result
- In Habit:

"has type" 1st arg 2nd arg result

```
div :: Int ⟶ Int ⟶ Int
```

- This is a lie!
- **Correction**: You can divide an integer by a **non-zero integer** to get an integer result
- In Habit:

```
div :: Int ⟶ NonZero Int ⟶ Int
```

- But where do `NonZero Int` values come from?



# Where do `NonZero` values come from?

- **Option 1**: Integer literals - numbers like `1`, `7`, `63`, and `128` are clearly all `NonZero` integers
- **Option 2**: By checking at runtime

```
nonzero :: Int ⟶ Maybe (NonZero Int)
```

Values of type `Maybe t` are either:
- `Nothing`
- `Just x` for some `x` of type `t`

- These are the only two ways to get a `NonZero Int`!
- `NonZero` is an **abstract datatype**



# Examples using `NonZero` values

- Calculating the average of two values:

```
ave     :: Int ⟶ Int ⟶ Int
ave n m = (n + m) `div` 2
```

a non zero literal

- Calculating the average of a list of integers:

```
average :: List Int ⟶ Maybe Int
average nums
   = case nonzero (length nums) of
       Just d  ⟶  Just (sum nums `div` d)
       Nothing ⟶  Nothing
```

checked!

- Key point: If you forget the check, your code will not compile!

# Null pointer dereferences

- In C, a value of type `T*` is a pointer to an object of type `T`
- But this may be a lie!
- A `null` pointer has type `T*`, but does NOT point to an object of type `T`
- Attempting to read or write the value pointed to by a `null` pointer is called a "**null pointer dereference**" and often results in system crashes, vulnerabilities, or memory corruption
- Described by Tony Hoare (who introduced null pointers in the ALGOL W language in 1965) as his "billion dollar mistake"



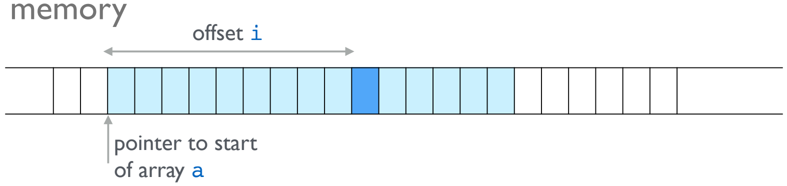
# Pointers and reference types

- Lesson learned: we should distinguish between
  - References (of type `Ref a`): guaranteed to point to values of type a
  - Pointers (of type `Ptr a`): either a reference or a `null`
- These types are not the same: `Ptr a = Maybe (Ref a)`
- You can only read or write values via a reference
- Code that tries to read from a pointer will fail to compile!
- Goodbye null pointer dereferences!



# Arrays and out of bounds indexes:

- Arrays are collections of values stored in contiguous locations in memory




- Address of `a[i]` = start address of `a` + `i`*(size of element)
- Simple, fast, … and dangerous!
- If `i` is not a valid index (an "out of bounds index"), then an attempt to access `a[i]` could lead to a system crash, memory corruption, buffer overflows, …
- A common path to "arbitrary code execution"



# Array bounds checking

- The designers of C knew that this was a potential problem … but chose not to address it in the language design:
  - We would need to store a length field in every array
  - We would need to check for valid indexes at runtime
- The designers of Java knew that this was a potential problem … and chose to address it in the language design:
  - Store a length field in every array
  - Check for valid indexes at runtime
- Performance **OR** Safety … pick **one**!

# Arrays in Habit

- Key idea: make array size part of the array type, do not allow arbitrary indexes:

```
(@) :: Ref (Array n t) ⟶ Ix n ⟶ Ref t
```

start address — index — element address

`a[i]` is written as `a@i` in Habit — array length, as part of the type — guaranteed to be ≥ `0` and < `n`

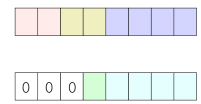
- Fast, no need for a runtime check, no need for a stored length
- `Ix n` is another abstract type:

```
maybeIx :: Int ⟶ Maybe (Ix n)
modIx   :: Int ⟶ Ix n
incIx   :: Ix n ⟶ Maybe (Ix n)
```



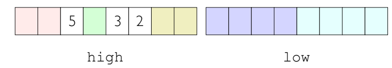
# Bit twiddling

- Given two 32 bit input values:
  - base:
  - limit:




Each box is one nibble (4 bits), least significant bits on the right

- Calculate a 64 bit descriptor:

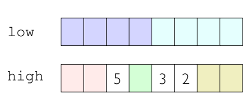


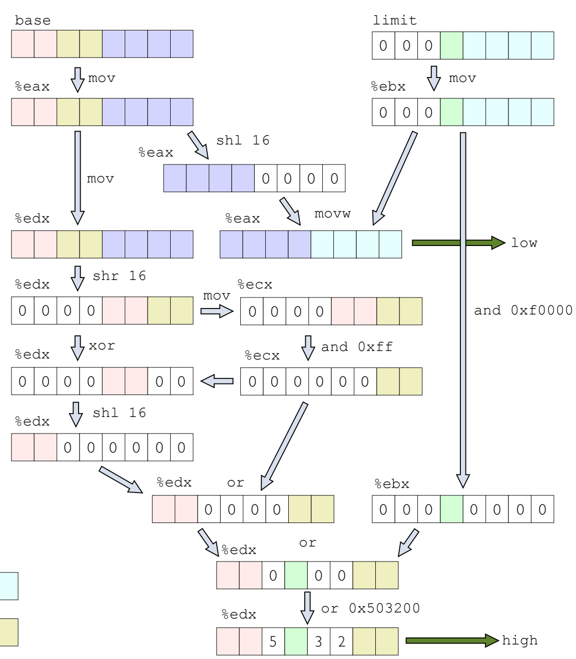

- Needed for the calculation of "Global Descriptor Table (GDT) entries" on the x86



# In assembly

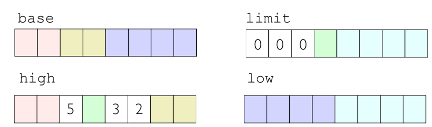
```
movl    base, %eax
movl    limit, %ebx

mov     %eax, %edx
shl     $16, %eax
mov     %bx, %ax
movl    %eax, low

shr     $16, %edx
mov     %edx, %ecx
andl    $0xff, %ecx
xorl    %ecx, %edx
shl     $16,%edx
orl     %ecx, %edx
andl    $0xf0000, %ebx
orl     %ebx, %edx
orl     $0x503200, %edx
movl    %edx, high
```





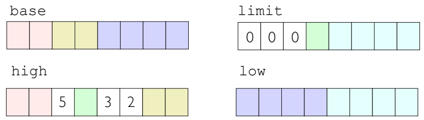

# In C




```
low  = (base << 16)           // purple
     | (limit & 0xffff);      // blue
high = (base & 0xff000000)    // pink
     | (limit & 0xf0000)      // green
     | ((base >> 16) & 0xff)  // yellow
     | 0x503200;              // white
```

- Examples like this show why we use high-level languages instead of assembly!
- But let's hope we don't get those offsets and masks wrong …
- And there is no safety net if we get the types wrong …

# In Habit




- Programmer describes layout in a type definition:

```
bitdata GDT
  = GDT [ pink   :: Bit 8 | 0x5  :: Bit 4
        | green  :: Bit 4 | 0x32 :: Bit 8
        | yellow :: Bit 8 | purple, blue :: Bit 16 ]
```

- Compiler tracks types and automatically figures out appropriate offsets and masks:

```
makeGDT :: Unsigned ⟶ Unsigned ⟶ GDT
makeGDT (pink # yellow # purple)  -- base
        (0 # green # blue)        -- limit
   = GDT [pink|green|yellow|purple|blue]

silly    :: GDT ⟶ Bit 8
silly gdt = gdt.pink + gdt.yellow
```

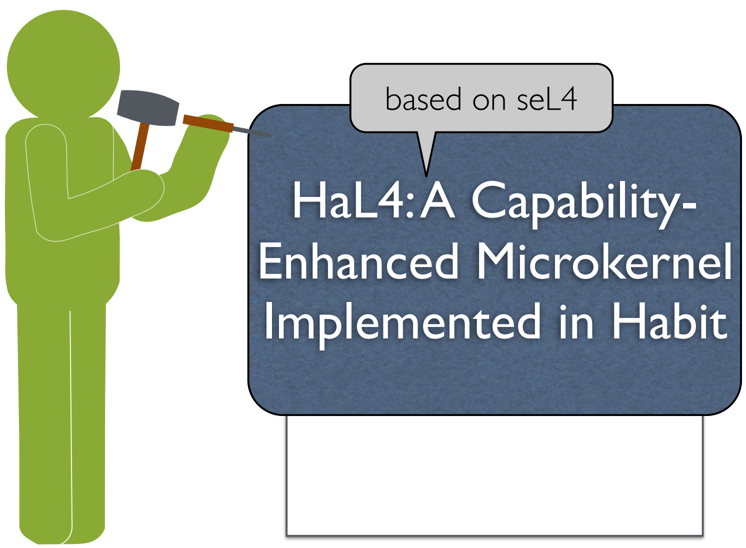


# Additional examples

- Layout and initialization of memory-based tables and data structures
- Distinguishing physical and virtual addresses
- Tracking (and limiting) side effects
  - ensuring some sections of code are "read only"
  - identifying/limiting code that uses privileged operations
  - preventing code that sleeps while holding a lock
  - …
- Reusable methods for concise and consistent input validation…
- …



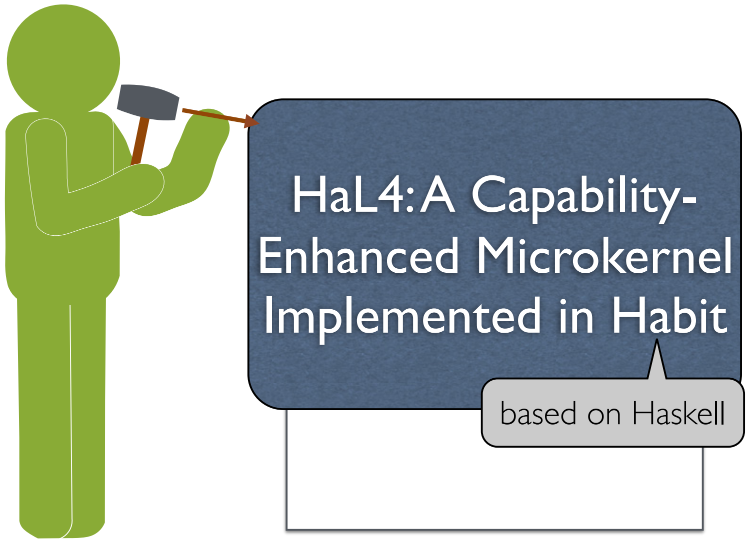
# Chipping away ...





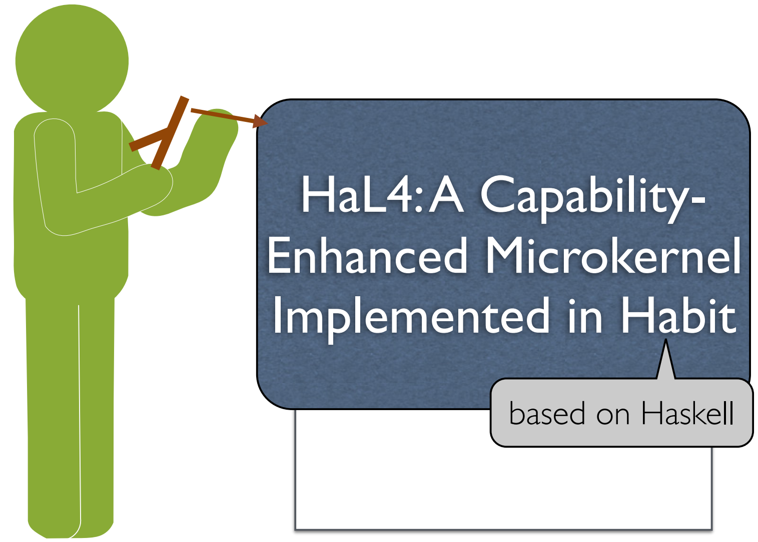

# Chipping away ...

## Using types …






## Using functional programming ...






## The CEMLaBS Project

- Three **technical** questions:
  - **Feasibility**: Is it possible to build an inherently "unsafe" system like seL4 in a "safe" language like Habit?
  - **Benefit**: What benefits might this have, for example, in reducing development or verification costs?
  - **Performance**: Is it possible to meet reasonable performance goals for this kind of system?
- A **social** question:
  - Can we persuade developers to try new languages?
- Maybe there is a role for modern programming languages …!?