Pointers and
References, Chapter 8

Programming Languages

Objective

Today's primary objective:

- You can compare and contrast pointers and references
- You know that references are safer
- You have seen a presentation similar to the presentations you are to give

Pointers versus References

I will cover:

- Why this topic
- Context
- Description pointers and references
- Coding examples
- Implementation
- Pros and cons
- Questions
- Quiz

Why This Topic

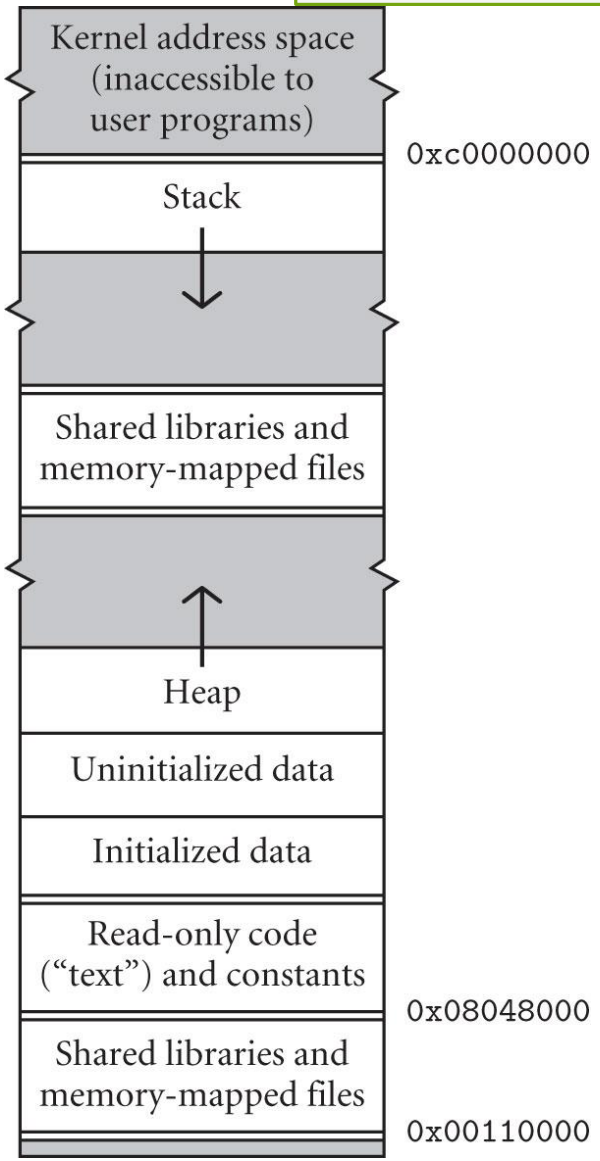
I choose this topic because:

- I remember how it was before there were reference types
- Both pointers and reference types are still used, and extensively
- References are relevant to functional languages

Context

Programming languages need variables that point to other variables:

- Recursive types – needed for list, trees
- Complex objects
- Dynamically obtained memory


Kernel address space
(inaccessible to
user programs)
0xc0000000
Stack
Shared libraries and
memory-mapped files
Heap
Uninitialized data
Initialized data
Read-only code
("text") and constants
0x08048000
Shared libraries and
memory-mapped files
0x00110000

Pointers / References in Languages

Pointers – C, C++, C# (unsafe mode), Go (Go implements differently to get advantages of references)

References – C++, C#, Java, Python, Ruby, Perl, PHP

C++ Pointers & References

Pointer – variable that holds the memory address of another variable

- Operators:
  - & - "address of"
  - * - value pointed to by this address

Reference – alias, that is, another name for an already existing variable

- Like a constant pointer with automatic indirection
  - & - "address of"
  - . or -> - value being referenced

Parameter Passing

Call-by-value – copy of value is passed

Definition	Call

void change (int num) {
num += 100;
}
	int x = 5;
	…
	change(x);
	… x is not changed…

Call-by-reference – address of the value is passed

Definition	Call

void change (int *num) {
(*num) += 100;
}
	int x = 5;
	…
	change(&x);
	… x is changed …

Value & Reference Types

Classification per language [edit]

Language	Value type	Reference type
C++[3]	booleans, characters, integer numbers, floating-point numbers, classes (including strings, lists, maps, sets, stacks, queues), enumerations	references, pointers
Java[4]	booleans, characters, integer numbers, floating-point numbers	arrays, classes (including immutable strings, lists, dictionaries, sets, stacks, queues, enumerations), interfaces, null pointer
C#[5]	structures (including booleans, characters, integer numbers, floating-point numbers, fixed-point numbers, lists, dictionaries, sets, stacks, queues, optionals), enumerations	classes (including immutable strings, arrays, tuples, lists, dictionaries, sets, stacks, queues), interfaces, pointers
Swift[6][7]	structures (including booleans, characters, integer numbers, floating-point numbers, fixed-point numbers, immutable strings, tuples, lists, dictionaries, sets, stacks, queues), enumerations (including optionals)	functions, classes, interfaces
Python[8]		classes (including immutable booleans, immutable integer numbers, immutab floating-point numbers, immutable complex numbers, immutable strings, byte strings, immutable byte strings, immutable tuples, immutable ranges, immutable memory views, lists, dictionaries, sets, immutable sets, null pointer
JavaScript[9]		immutable booleans, immutable floating-point numbers, immutable symbols, immutable strings, undefined, prototypes (including lists, null pointer)
OCaml[10][11]	immutable characters, immutable integer numbers, immutable floating-point numbers, immutable tuples, immutable enumerations (including immutable units, immutable booleans, immutable lists, immutable optionals), immutable exceptions, immutable formatting strings	arrays, immutable strings, byte strings, dictionaries (including pointers)

Boxing

Boxing – wrap a value-type variable to make it a reference type

Relevant to languages that distinguish between value and reference types

Variable Models

Value model of variables – variables are seen as named containers

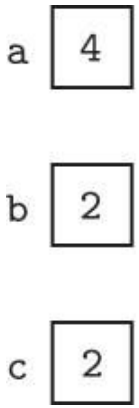
Reference model of variables – variables are a named reference to a variable

b:=2

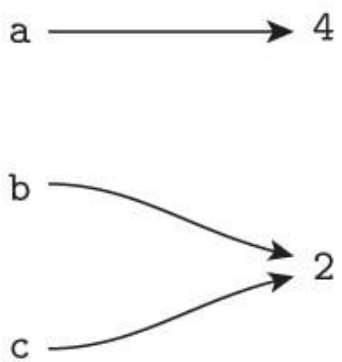
c:=b

a:=b+c

value model



reference model



Pointers versus References

Functional		Scheme Lisp, Haskell Ocaml
Imperative	C Ada Java built-in types C# built-in types, structs, C# can use pointers (unsafe mode)	SmallTalk Ruby Java objects C# objects
↑ Type of Language Model for variables →	Value model	Reference Model

Pointers versus References

Pointers correspond to a value model

References correspond to a reference model

C++ Example Declaration

Pointer – stores the address of another variable

Reference – variable which refers to another variable

```
int  i = 3;
int *ptr = &i;        // Pointer
int &ref = i;         // Reference
```

C++ Example Use

```
int  i = 3;
int *ptr = &i;
int &ref = i;

*ptr = 13;
ref = 13;
```

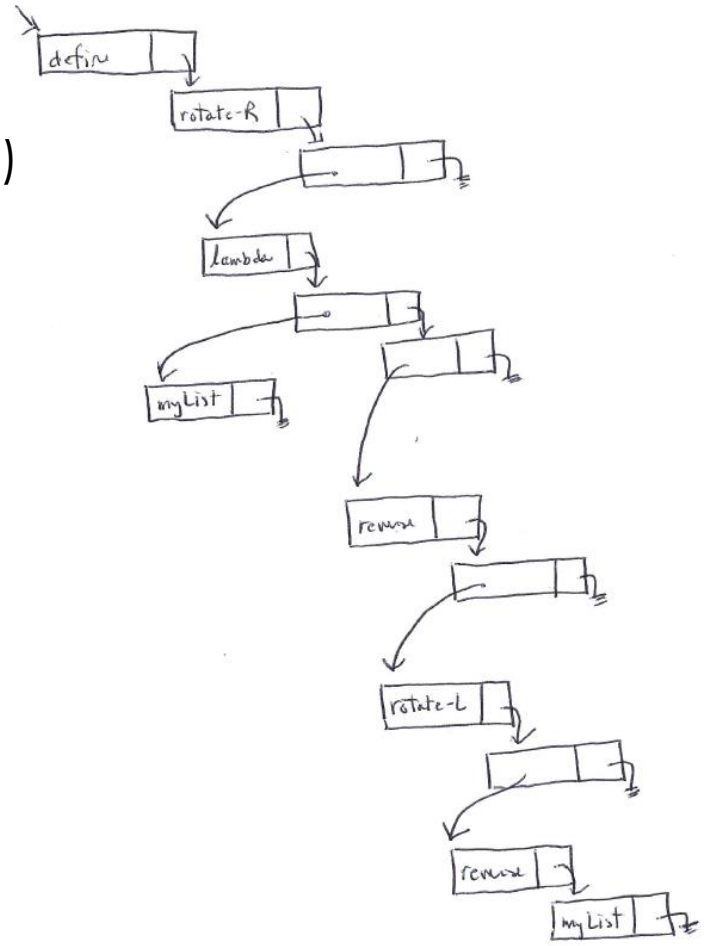
Haskell

Haskell - all data are immutable (like Scheme)

- no need to distinguish between passing an argument by value or by reference
- compiler is free to choose whichever representation it thinks will be the most efficient b
- lazy language (uses delayed computation)
- if compiler isn't confident that a function will evaluate its arguments, it passes the argument as a "thunk", which is a delayed computation, which is not passing by value or by reference
- “thunk” might be shared by multiple computations, and we only want it to be evaluated once (rather than every time it's used)

Scheme Example (references)

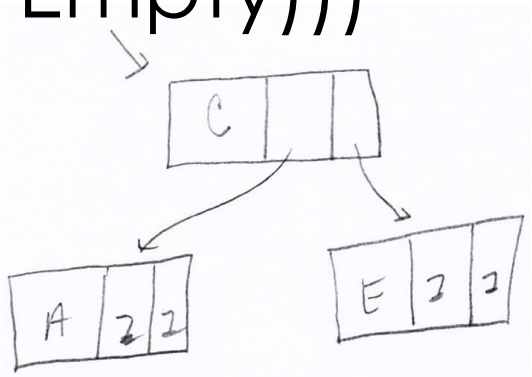
```
(define rotate-R
  (lambda (myList)
    (reverse (rotate-L (reverse myList)
  )
)
```



Ocaml Example (references)

```
type char_tree = Empty  |
    Node of char * chr_tree * chr_tree;;
```

Node ('C', Node ('A', Empty, Empty), Node ('E', Empty, Empty)))



Ada Example (pointers)

```
type chr_tree;
type chr_tree_ptr is acces chr_tree;
type chr_tree is record
        left, right : chr_tree_ptr;
        value : character;
end record;

my_ptr := new chr_tree
```

C Example (pointers)

```
struct chr_tree {
        struct chr_tree *left, *right;
        char val;
};


my_ptr = malloc(sizeof(struct chr_tree))
```

Implementation of References

C++ and Java standards carefully avoid dictating how a compiler implements references

Every C++ compiler implements references as pointers, consequently references require the same amount of storage as pointers

Implementation of References

With references - garbage collection possible

Garbage collection - automatic heap storage management

Programmer no longer:

- Forgets to reclaim storage - memory leak
- Claims storage too early – dangling references

Pointers

Pros	Cons
Flexible	Dangerous
Can use arithmetic for efficient array manipulation	Programmer needs to manage memory
During lifetime can point to different objects	Can be hard to debug
Can get the address	Memory leaks and dangling pointers are possible
Can be set to null	

References

Pros	Cons
Safe	Can't get the value of the address
During lifetime, only references one object (constant)	
Valid reference can never be null	
System can manage memory (garbage collection)	Run-time cost to garbage collect
Dereferencing is done automatically	
Compiler can do more optimization	Can't do pointer arithmetic

Question 1

An advantage of pointers over references is that:

a. Pointers can point to different objects over their lifetime
b. Compiler can do more optimization
c. Memory can be reclaimed via garbage collection
d. Safety
e. All of the above

Question 2

An advantage of references over pointers is that:

a. Dereferencing is done automatically
b. Memory can be reclaimed via garbage collection
c. Compiler can do more optimization
d. Safety
e. All of the above