CS 420 Advanced Programming Languages
Fall Semester, 2022

Doc 2 Stack & Heap

Aug 23, 2022

Sample Functions

```c
int calculating(int number1, int number2) {
    int x = 2 * number1;
    int y = 3 * number2;
    return x + y;
}
```

C

```python
def calculating(number1, number2):
    x = 2 * number1
    y = 3 * number2
    return x + y
```

Python

```julia
function calculating(number1, number2)
    x = 2number1
    y = 3number2
    x + y
end
```

Julia

C Execution

```
#include <stdio.h>

int calculating(int number1, int number2) {
   int x = 2 * number1;
   int y = 3 * number2;
   return x + y;
}

int bar(int x) {
    int y = x + 1;
    return calculating(y,5);
}

int main() {
    int answer1 = bar(3);
    int answer2 = calculating(3,5);
    printf("%d", answer1);
    return 0;
}
```

3

Python's Stack Problem

```
def foo():
    x = 5
    x = 12.23
    x = "This is a long string. It takes a lot more space to store than an integer"
    n = int(input('Enter a number: '))
    x = [y ** 2 for y in range(n)]
    return x
```

Can not store x in the stack

- Unknown size
- What happens to the return value

Python

py_num = 42

Create a PyObject; allocating enough memory to an address

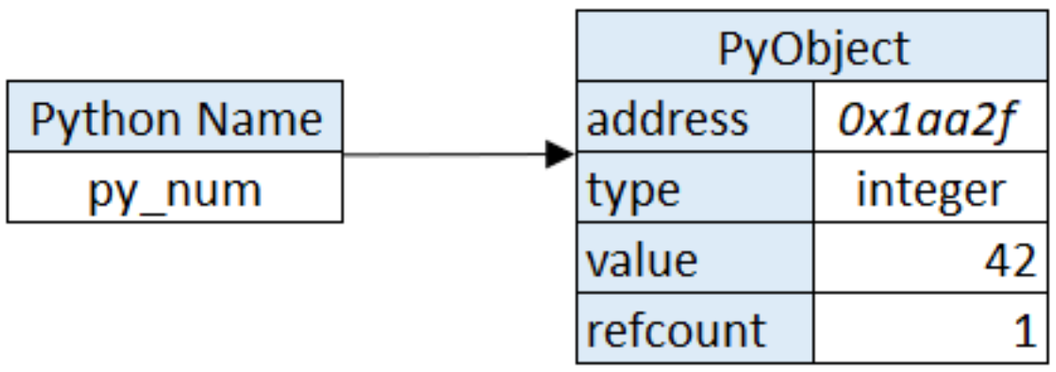
Set the PyObject’s typecode to integer (as determined by the interpreter)

Set the PyObject’s value to 42

Create a name called py_num

Point py_numto the Pyobject

Increment the PyObject’s refcount by 1



https://towardsdatascience.com/why-is-python-so-slow-and-how-to-speed-it-up-485b5a84154e

Memory Heap

Stores data independent of stack

Allows dynamic sizing of data

Slower than stack

Needs a separate mechanism to manage

Fragmentation

Ways of Handling Heap Memory

Garbage Collection

Reference Counting

Manual

Python Reference Counting

```
def foo():
    x = 5
    y = 12.23
    return x + y

print(foo())
```

Python Code

```
def bar(x):
    y = x + 1
    return calculating(y,5)

def calculating(number1, number2):
    x = 2 * number1
    y = 3 * number2
    return x + y

answer = bar(2)
print(answer)
```

We can use calculating before defining it

Python Code

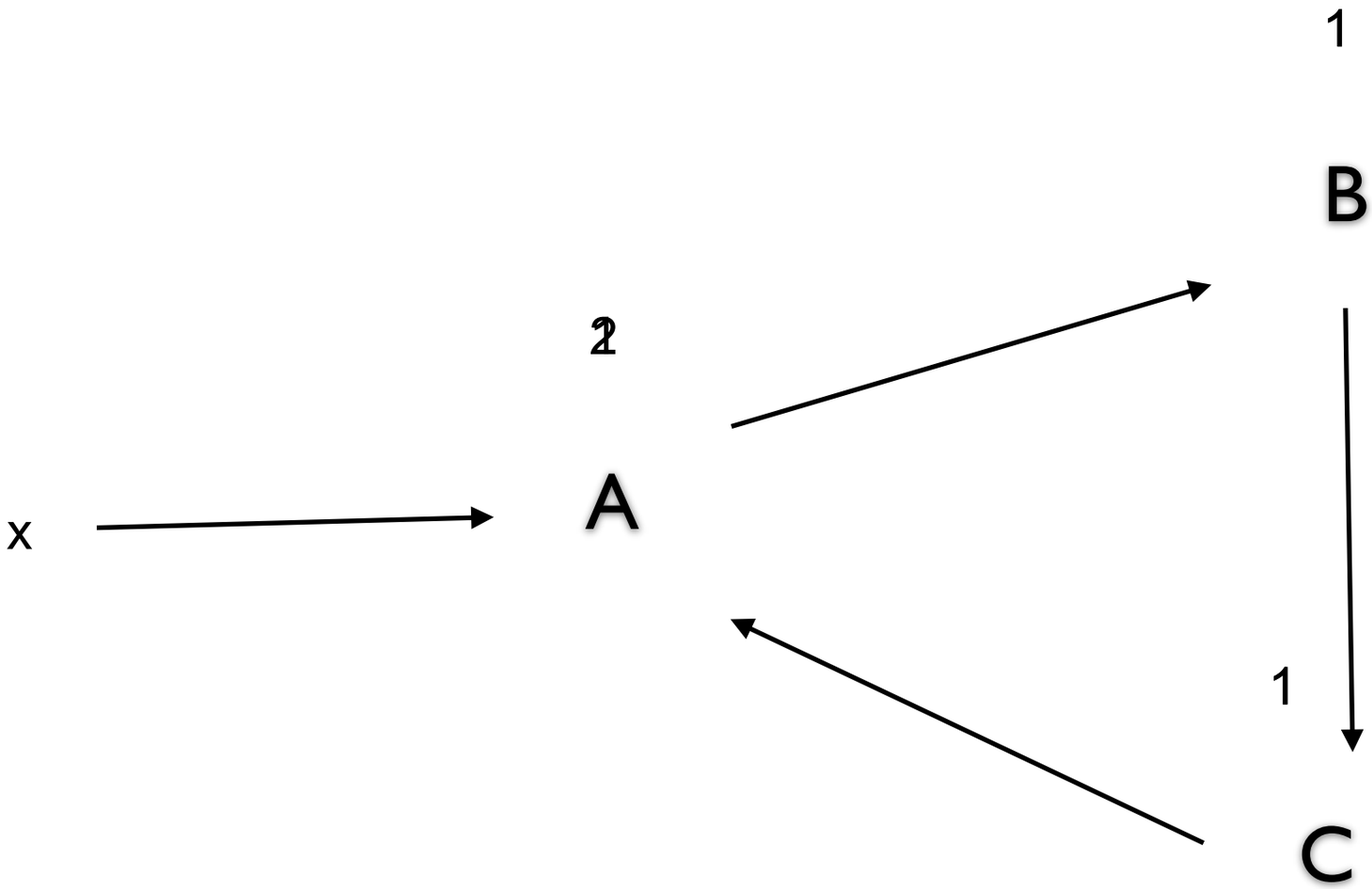
```
def bar(x):
    y = x + 1
    return calculating(y,5)

def calculating(number1, number2):
    x = 2 * number1
    y = 3 * number2
    return x + y
```

```
answer = calculating("cat", "dog")
print(answer)
```

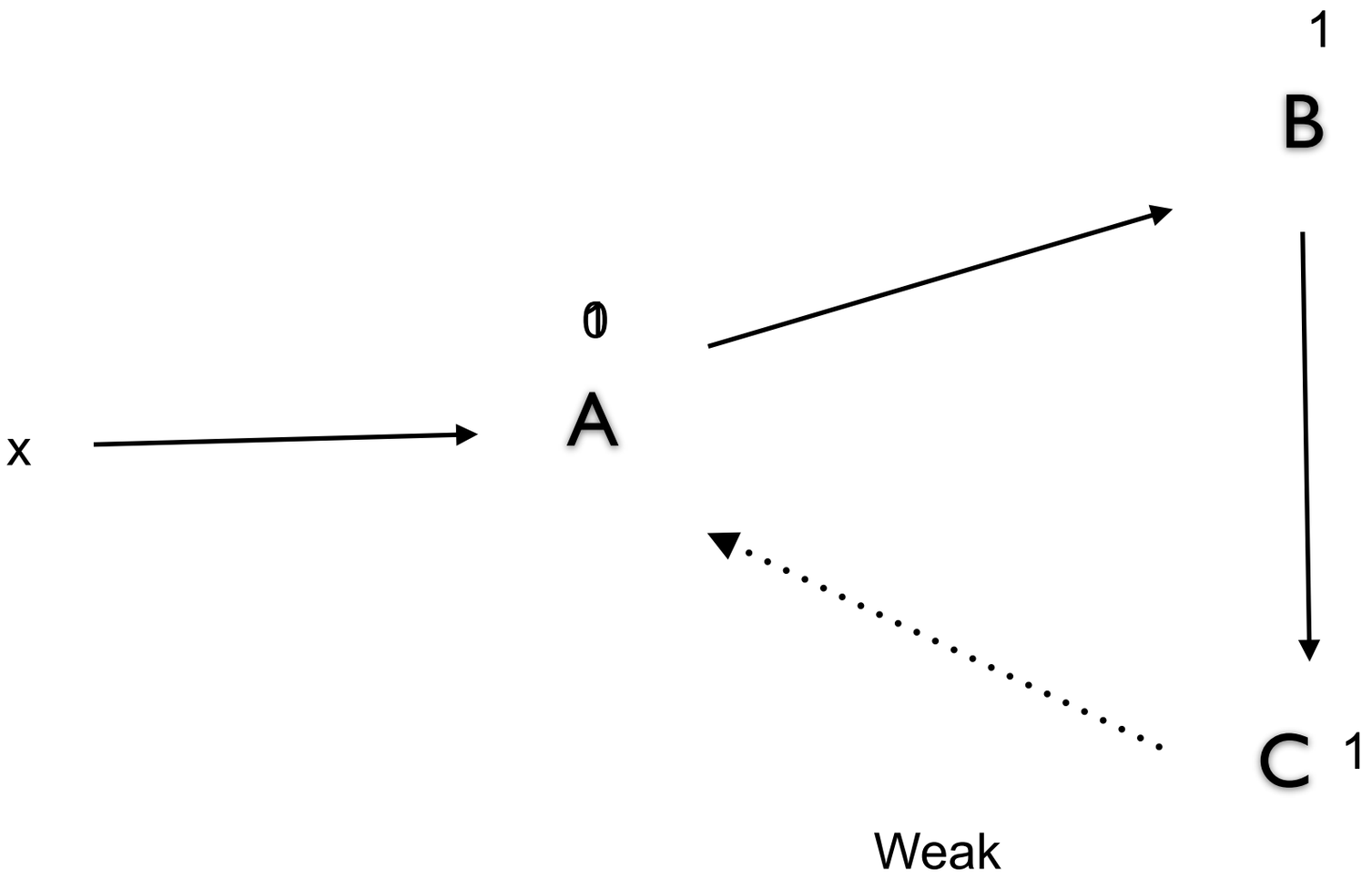
Why Is Reference Counting not Enough?

Weak

Reference count & Circular References



Weak Reference to Break Circular References



Ways of Handling Heap Memory

Garbage Collection

Runtime process scans all memory

Frees unused Heap memory

Java C# Python Ruby Go

Standard in all most all modern languages

Removes nearly all errors caused by manual memory management

Runtime Performance

5x more memory - same as program using manual memory management
3x more memory - ~17% slower
2x more memory - ~70% slower

Why Does Garbage Collection need more Space

Generational garbage collection copies active heap memory to new location

Compacting memory

Julia

```
bar(x) = calculating(x + 1, 5)

function calculating(number1, number2)
    x = 2number1
    y = 3number2
    x + y
    end

bar(2)
```

Type Inference

Compiler generates

Multiple versions calculating

Manual Memory Management

Example C

malloc - Allocate memory from heap

free - Release memory

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
	int* ptr;
	int i;
	ptr = (int*)malloc(10 * sizeof(int));

	if (ptr == NULL) {
		printf("Memory not allocated.\n");
		exit(0);
	}
	else {
		for (i = 0; i < 10; ++i) {
			ptr[i] = i + 1;
		}
	}
	return 0;
}
```

```c
#include <stdio.h>
#include <stdlib.h>

int* allocate(int n) {
    int *p = (int*)malloc(n * sizeof(int));
    if (p == NULL) {
        printf("Error: malloc failed in allocate");
        exit(1);
    }
    for (int i = 0; i < n; i++) {
        p[i] = i;
    }
    return p;
}

int main() {
    int *p = allocate(10);
    printf("%d ",p[5]);
    return 0;
}
```

Zombies

```
#include <stdio.h>
#include <stdlib.h>

int* allocate(int n) {
    int *p = (int*)malloc(n * sizeof(int));
    if (p == NULL) {
        printf("Error: malloc failed in allocate");
        exit(1);
    }
    for (int i = 0; i < n; i++) {
        p[i] = i;
    }
    free(p);
    return p;
}

int main() {
    int *p = allocate(10);
    printf("%d ",p[5]);
    return 0;
}
```

Memory Leak

```
#include <stdio.h>
#include <stdlib.h>

void allocate(int n) {
    int *p = malloc(n * sizeof(int));
    if (p == NULL) {
        printf("Error: malloc failed in allocate");
        exit(1);
    }
    for (int i = 0; i < n; i++) {
        p[i] = i;
    }
}

int main() {
    for (int k = 0; k < 100,000;k++) {
        allocate(k);
    }
    return 0;
}
```

Rust in the Android platform

"Memory safety bugs in C and C++ continue to be the most-difficult-to-address source of incorrectness.

We invest a great deal of effort and resources into detecting, fixing, and mitigating this class of bugs, and these efforts are effective in preventing a large number of bugs from making it into Android releases.

Yet in spite of these efforts, **memory safety bugs** continue to be a top contributor of stability issues, and consistently represent ~70% of Android's high severity security vulnerabilities."

https://security.googleblog.com/2021/04/rust-in-android-platform.html

Recursion

```
def fact(n):
    if (n == 0):
        return 1
    return n * fact(n-1)
```

```
if __name__ == '__main__':
    print(fact(5))
```

```
if __name__ == '__main__':
    print(fact(998))
```

RecursionError: maximum recursion depth exceeded in comparison

https://www.geeksforgeeks.org/tail-recursion/

Tail Recursion

```
def fact(n, a=1):

    if (n == 1):
        return a

    return fact(n - 1, n * a)

print(fact(5))
```

If last line of function is a recursive call

A smart compiler can reuse the activation record

```
print(fact(999))
```

Python does not support Tail recursion

Static Chains

```
function main(){
    var x;
    function bigsub() {
        var a, b, c;
        function sub1 {
            var a, d;
            ...
             a = b + c;
        } // end of sub1
        function sub2(x) {
            var b, e;
            function sub3() {
                var c, e;
                ...
                sub1();
                ...
                e = b + a;
            } // end of sub3
            ...
            sub3();
            ...
            a = d + e;
        } // end of sub2
        sub2(7);
} // end of bigsub
```

Application Binary Interface (ABI)

How software programs can communicate with compiled binaries

- How functions' arguments are passed and return values are returned
- How datatype instances are laid out in memory

Closure

```
function create_adder(n)
    sum = n
    function adder(k)
        sum = sum + k
        return sum
    end
    return adder
end
```

```
accumulator10 = create_adder(10)
accumulator5 = create_adder(5)
```

```
accumulator10(2)
accumulator5(3)
accumulator10(8)
accumulator5( -4)
```

```
print( accumulator10(0))
print( accumulator5(0))
```

What is printed out?

How does it work?