Check-in 30

Review – Other Codegen

Translate the following to x64

```
int main(){
  int8_t a;
  int8_t b;
  return a + -b;
}
```

Check-in 30 Solution

Review – Other Codegen

Translate the following to x64

```
int main(){
  int8_t a;
  int8_t b;
  return a + -b;
}
```

-17(%rbp)

-18(%rbp)

```
pushq %rbp
movq %rsp, %rbp
addq $16, %rbp
subq $16, %rsp
movb -18(%rbp), %al
~~subb $0, %al~~
movb $0, %bl
subb %al, %bl
movb -17(%rbp), %al
addb %bl, %al
```

```
addq $16, %rbp
popq %rbp
retq
```

movsx %al, %eax

Announcements

Administrivia

Quiz 3 Friday

- Review session TOMORROW at 7:00 (room TBA ☹)

```
main : () -> int {
    a : immutable int = 3
    maybe a means | otherwise 2; X
    a = 2; X
    a++; X
}
```

EECS 665

COMPILER CONSTRUCTION

Heap Management

Previously…
Other Code Generation

Other constructs

- Shorter primitive types
- Arrays
- Pointers
- Strings
- Structs

You Should Know

- How to compile programs with strings
- How to compile programs with arrays
- The general idea behind pointers and shorter primitive types

Machine Codegen

Today’s Outline

Heap Management

Heap Memory

- Using the heap
- OS interface

Garbage collection

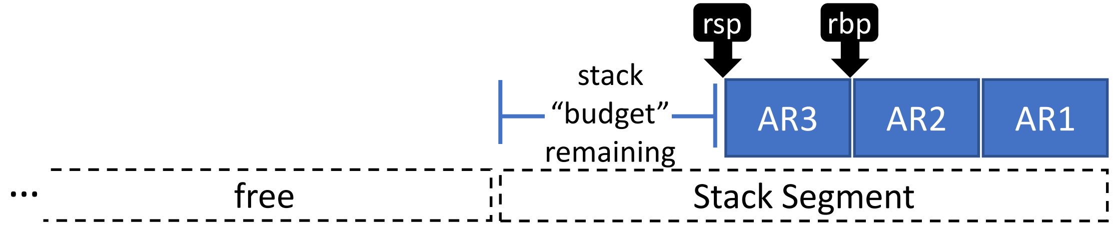
- Reference Counting
- Mark and Sweep

Machine Codegen

The Stack "Budget"

Fixed overall budget, managed internally

(On Linux):



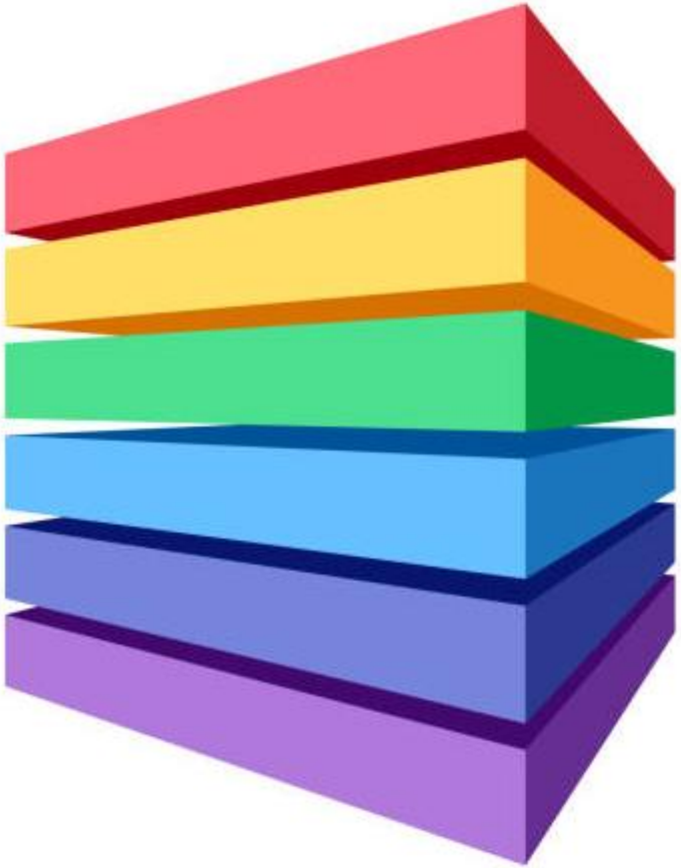
- The stack segment is actually pretty small (10 MB)!

When the Stack isn't enough

Heap Management – Heap Memory

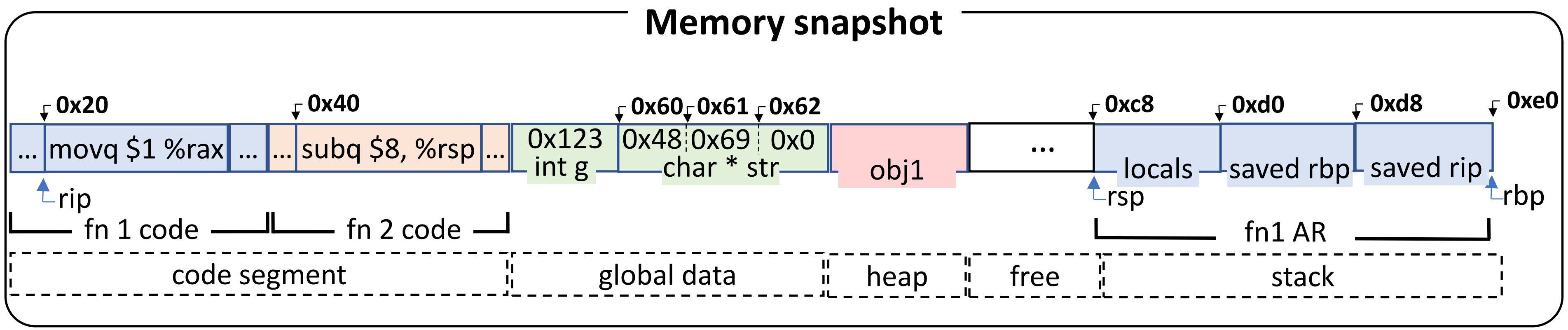
Stack memory is *efficient* but *constrained*

- (De)Allocation is easy (just move the stack ptr)
- Object lifetime is at most the lifetime of the activation record
  - This is a significant limitation!



Don’t Forget the Heap!

Heap Management – Heap Memory



Expressiveness/Efficiency Limitations

Heap Management

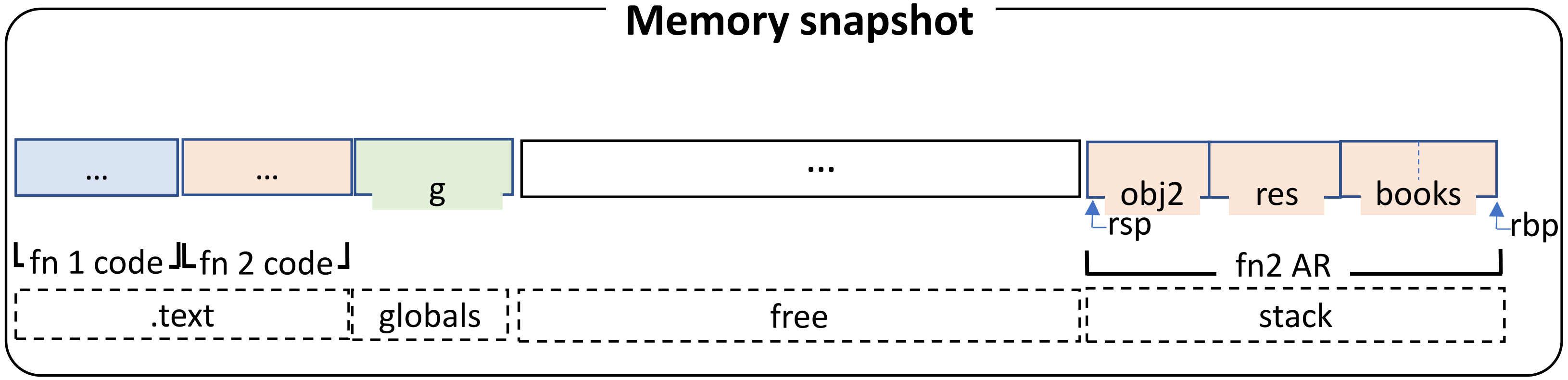
```
int[200] getArrayOf5s(){
  int[200] res;
  for (int i = 0 ; i < 200 ; i++){
    res[i] = 5;
  }
  return res;
}
main(){
  int[200] fives = getArrayOf5s();
}
```

Would like res allocated in the callee but alive in the caller

The Heap: Basic Idea

Heap Management

Disassociate memory region from ARs

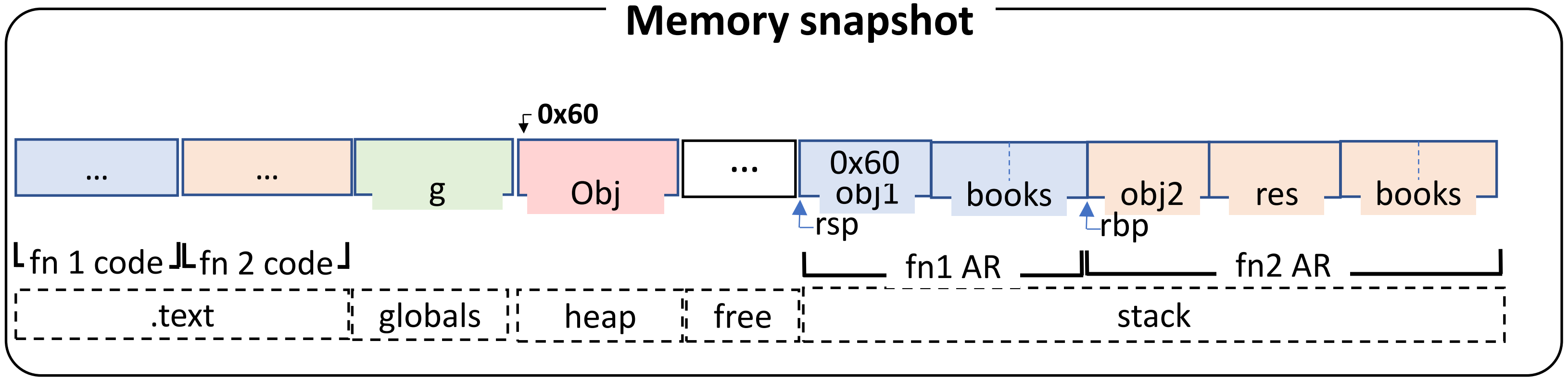



```
Obj * g;
Obj * fn1(){
  Obj * obj1 = new Obj();
  return obj1;
}
void fn2(){
 Obj * res = fn1();
 Obj * obj2 = new Obj();
 g = new Obj();
}
```

The Heap: Basic Idea

Heap Management

Disassociate memory region from ARs

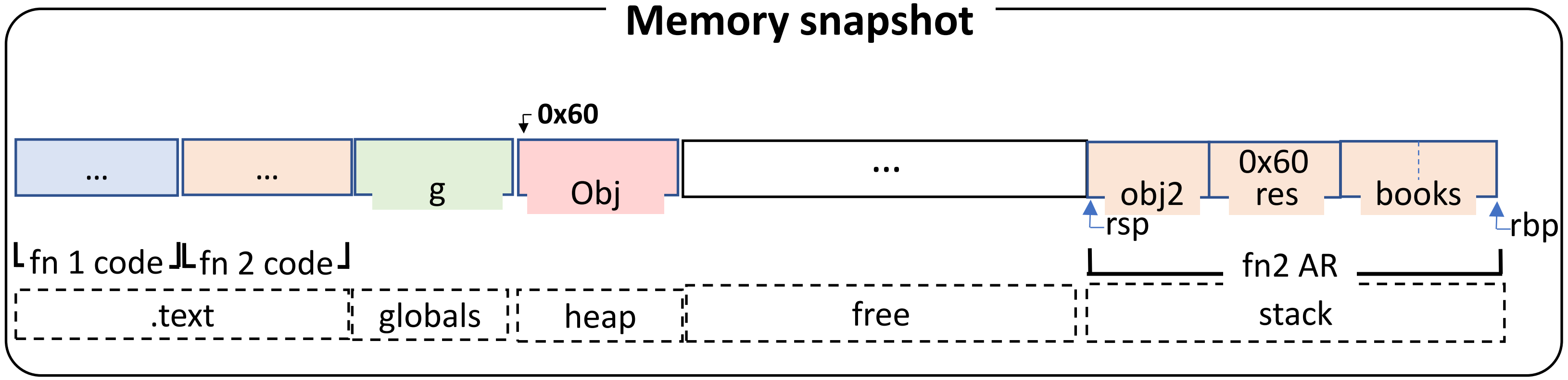



```
Obj * g;
Obj * fn1(){
  Obj * obj1 = new Obj();
  return obj1;
}
void fn2(){
  Obj * res = fn1();
  Obj * obj2 = new Obj();
  g = new Obj();
}
```

The Heap: Basic Idea

Heap Management

Disassociate memory region from ARs

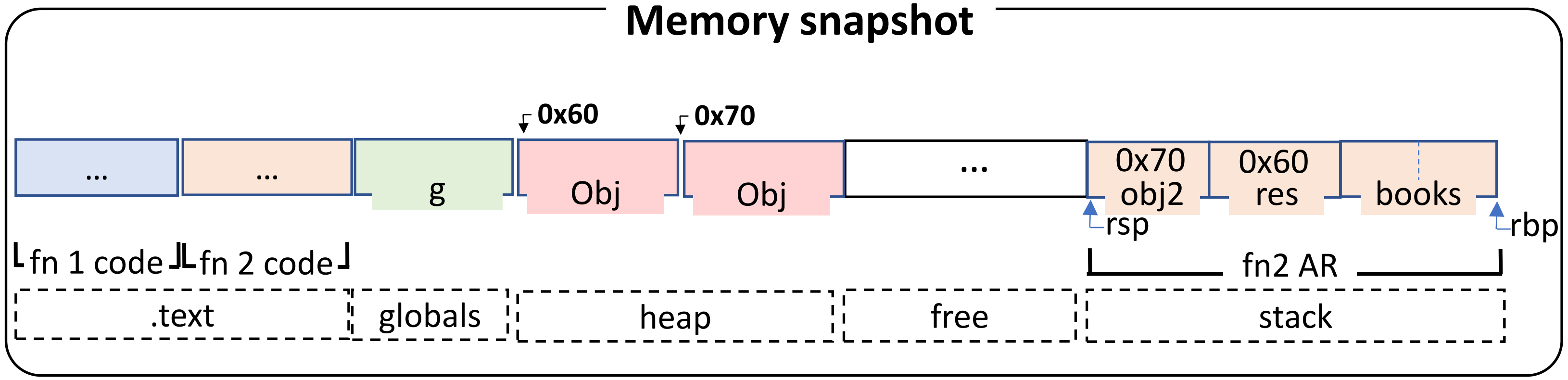



```
Obj * g;
Obj * fn1(){
  Obj * obj1 = new Obj();
  return obj1;
}
void fn2(){
 Obj * res = fn1();
 Obj * obj2 = new Obj();
 g = new Obj();
}
```

The Heap: Basic Idea

Heap Management

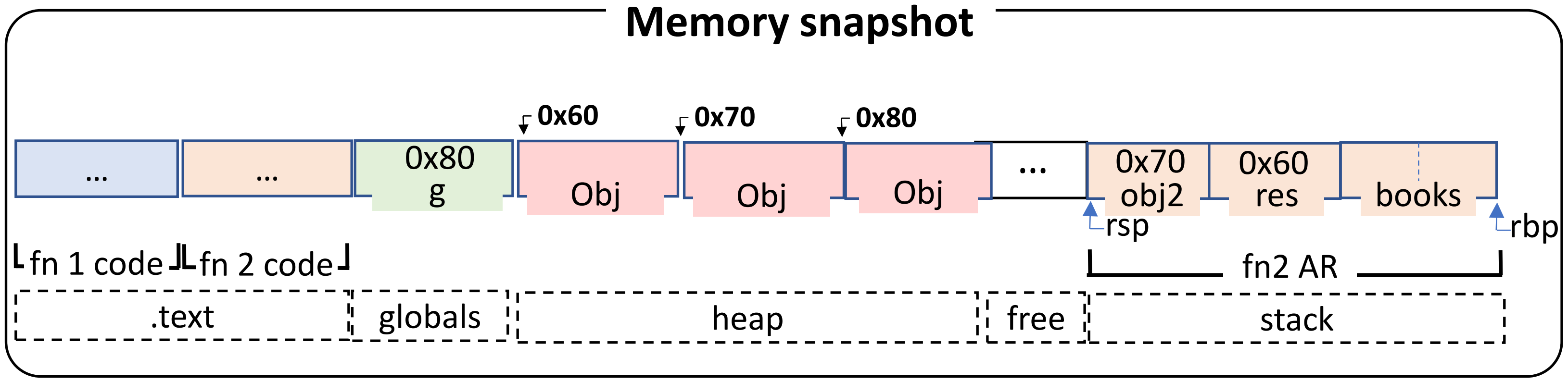
Disassociate memory region from ARs




```
Obj * g;
Obj * fn1(){
  Obj * obj1 = new Obj();
  return obj1;
}
void fn2(){
 Obj * res = fn1();
 Obj * obj2 = new Obj();
 g = new Obj();
}
```

The Heap: Basic Idea
Heap Management

Disassociate memory region from ARs




```
Obj * g;
Obj * fn1(){
  Obj * obj1 = new Obj();
  return obj1;
}
void fn2(){
 Obj * res = fn1();
 Obj * obj2 = new Obj();
 g = new Obj();
}
```

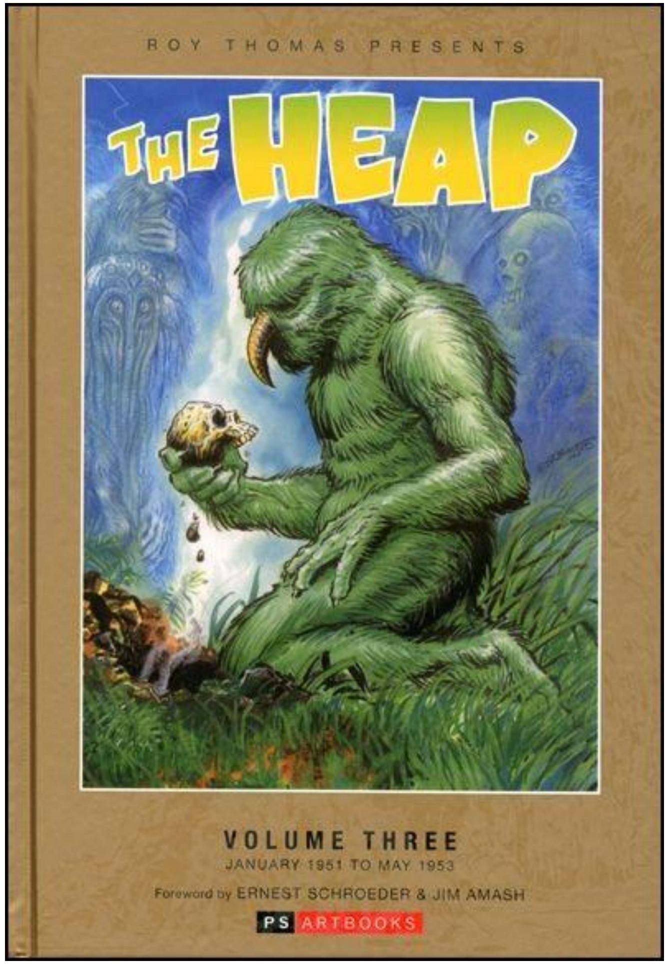
About the Heap
Heap Management

Appropriately named:

- Not as well-ordered compared to the stack

Benefits

- Reduces data copied between caller and callee



About the Heap

Heap Management

Appropriately named:

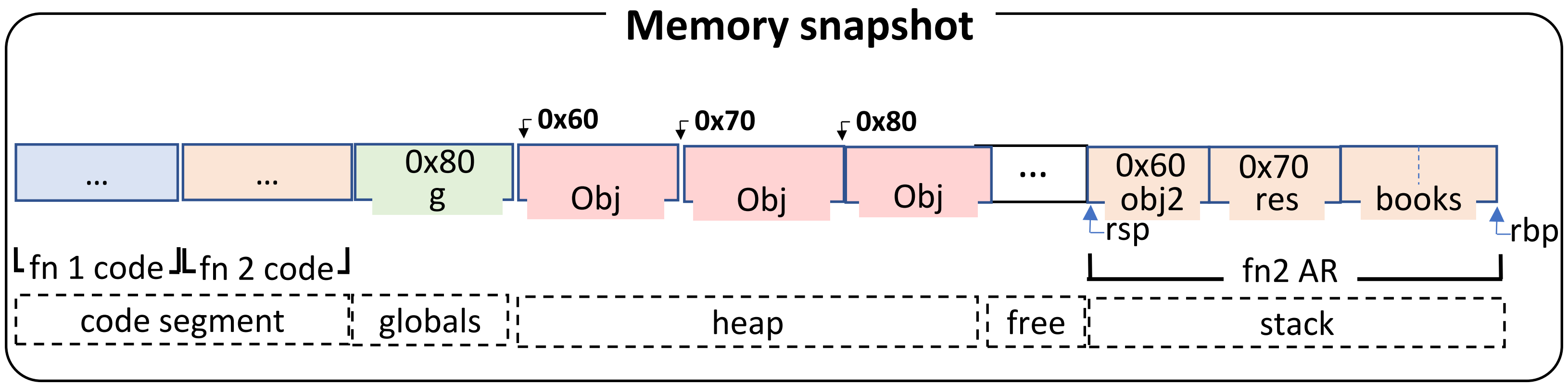
- Not as well-ordered compared to the stack

Benefits

- Reduces data copied between caller and callee

Just return address
0x60 in register

```
Obj * g;
Obj * fn1(){
   Obj * obj1 = new Obj();
   return obj1;
}
void fn2(){
  Obj * res = fn1();
  Obj * obj2 = new Obj();
  g = new Obj();
}
```



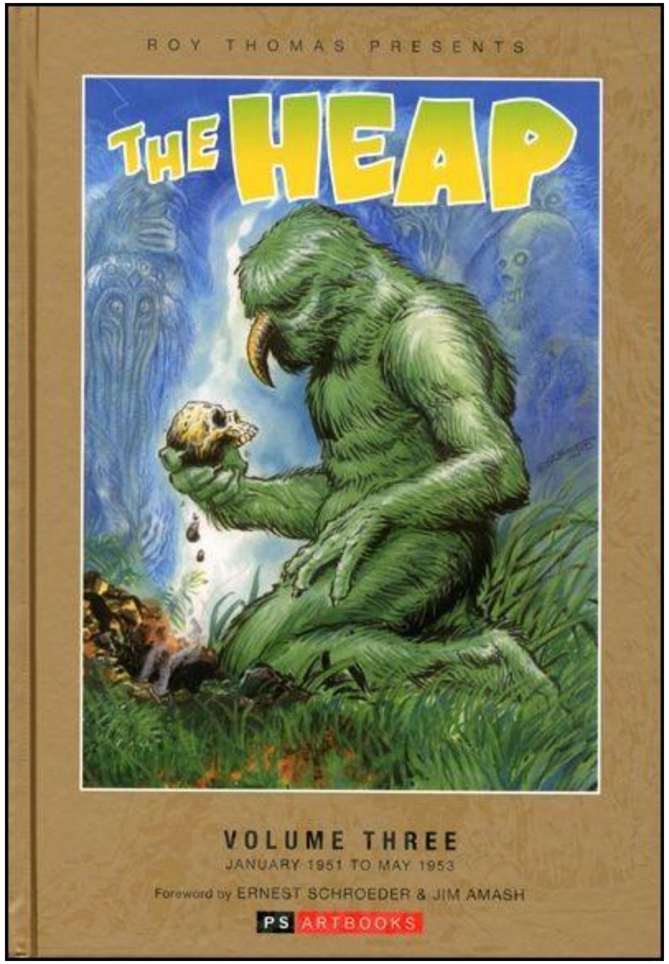
About the Heap
Heap Management

Appropriately named:

- Not as well-ordered compared to the stack

Benefits

- Reduces data copied between caller and callee
- Flexible lifetime
- Allows for various non-stack abstractions



Heap-Appropriate Abstractions

Some Functions don’t fit the tradition stack-based lifecycle

- First-class functions
- Function closures

```
def outer {
  int a;
  def inner(){
    a = 1;
  }
  return inner;
}
```

Simply allocate the closure on the heap

Heap Allocation

Heap Management

Naïve approach 1:

- Allocate all process memory at load time
- Incredibly wasteful (probably not even possible)!

A modern 64-bit OS will actually limit heap / stack size to discrete, never-overlapping segments

- This might seem like a limitation – it isn't

2^{64} bytes > 18 million terabytes of RAM

Managing the Heap

Only use the memory you need

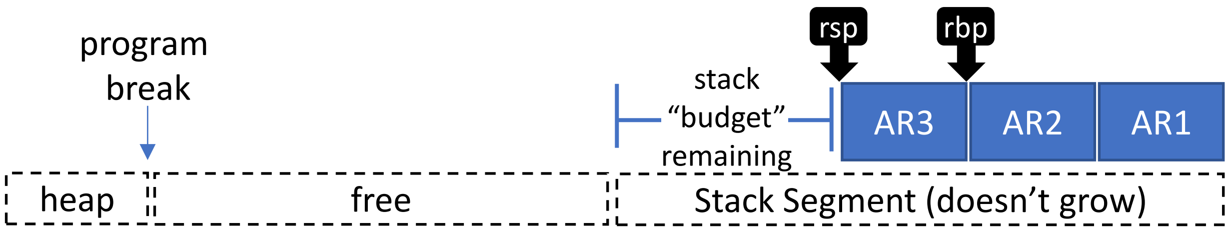
- The whole point is to allocate memory dynamically

Heap Allocation: brk / sbrk

Heap Management

Linux syscall for growing the heap

- `int brk(void *addr);`
  - Set the position of the program break
  - Linux: when addr is 0, returns current program break

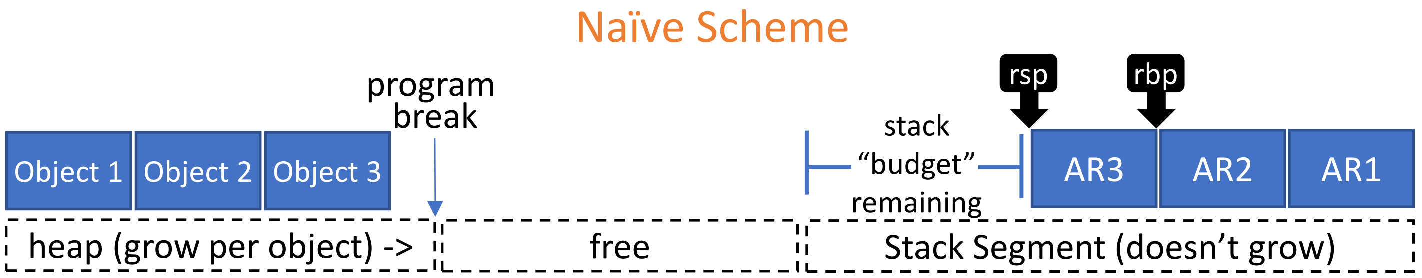


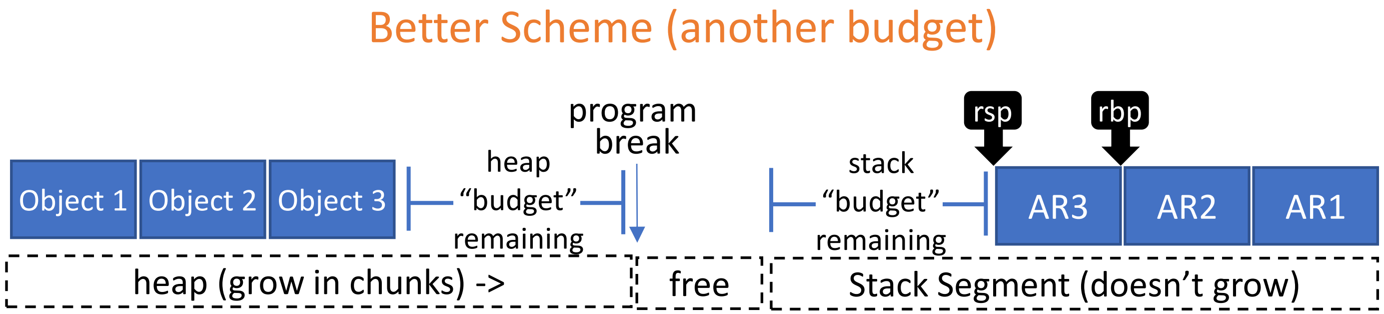
Heap Allocation

Heap Management

Naïve approach 2:

- Ask the OS to allocate exactly the number of bytes we need for each new object
- Very slow!





Heap Deallocation

Heap Management

When the heck do you free up heap memory?

```
Obj * g;
Obj * fn1(){
   Obj * obj1 = new Obj();
   return obj1;
}
void fn2(){
  Obj * res = fn1();
  Obj * obj2 = new Obj();
  g = new Obj();
}
```

```
Obj * g;
Obj * fn1(){
   Obj * obj1 = new Obj();
   return obj1;
}
void fn2(){
  fn1();
  Obj * obj2 = new Obj();
  g = new Obj();
}
```

Heap Deallocation

Heap Management

When the heck do you free up heap memory?

Whose job is it?

- Simplest approach: rely on the programmer

Heap Deallocation

Heap Management

When the heck do you free up heap memory?

Whose job is it?

- Simplest approach: rely on the programmer
  - The C/C++ way
  - Still some complexity in managing the heap
    - Heap compaction
- “Modern” approach: free heap space automatically

Heap Management Terminology

- **Cells:** data items on the heap
  - Cells are pointed to by other cells, or by registers, stack pointers, global variables
- **Roots:** registers, stack pointers, global variables
- A cell is **live** if it pointed to by a root or another live cell

Garbage Collection

Heap Management: Garbage Collection Overview

- **Garbage:** A memory block that cannot be (validly) accessed by the program
    - Obviously: a cell that is no longer live
    - Less Obviously: An explicitly deallocated cell still pointed-to
- **Garbage collection:** Automatically reclaiming garbage for use in future allocation

Garbage Collection: Considerations

Because it’s automatic it can be unpredictable

- It better not be too disruptive to performance
- It better be correct
  - Don’t deallocate live cells / minimize memory leaks

Garbage Collection: Real-Time Issue

Because it's automatic it can be unpredictable

- When is the garbage collector kick in?
- How long will it take to run?

The software product may contain support for programs written in Java. Java technology is not fault tolerant and is not designed, manufactured, or intended for use or resale as on-line control equipment in hazardous environments requiring fail-safe performance, such as in the operation of nuclear facilities, aircraft navigation or communication systems, air traffic control, direct life support machines, or weapon systems, in which the failure of Java technology could lead directly to death, personal injury, or severe physical or environmental damage.

- From the Windows EULA

Today's Outline
Heap Management: Garbage Collection

Heap Memory

- Using the heap
- OS interface

Garbage collection

- Reference Counting
- Mark and Sweep

Machine Codegen

Naïve Reference Counting

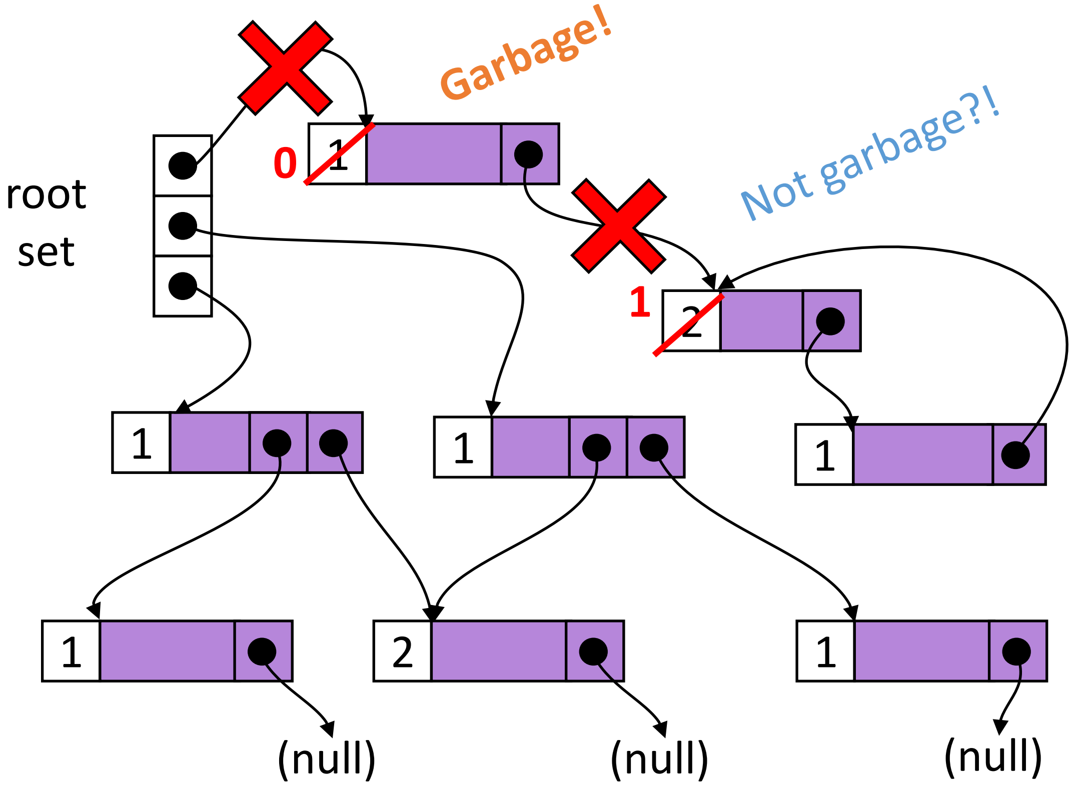
Heap Management: Garbage Collection

Associate a count with each Heap cell

- When a pointer is assigned to the cell, increment count
- When a pointer leaves scope (i.e. dies), decrement count

Predictable, fairly fast

- Used by C++ smart pointers / Python



Naïve Reference Counting: Limitations

Heap Management: Garbage Collection

Space Overhead

- 1 counter per cell

Time Overhead

- Fix up counts
- Check for self-loops

Potential leaks

- Cycles

Reference Counting: Summary

Associate a count with each Heap cell

- When a pointer is assigned to the cell, increment the count
- When a pointer goes out of scope/goes dead, decrement the count

Pretty predictable, relatively fast

- Used by C++ smart pointers / Python

Mark and Sweep

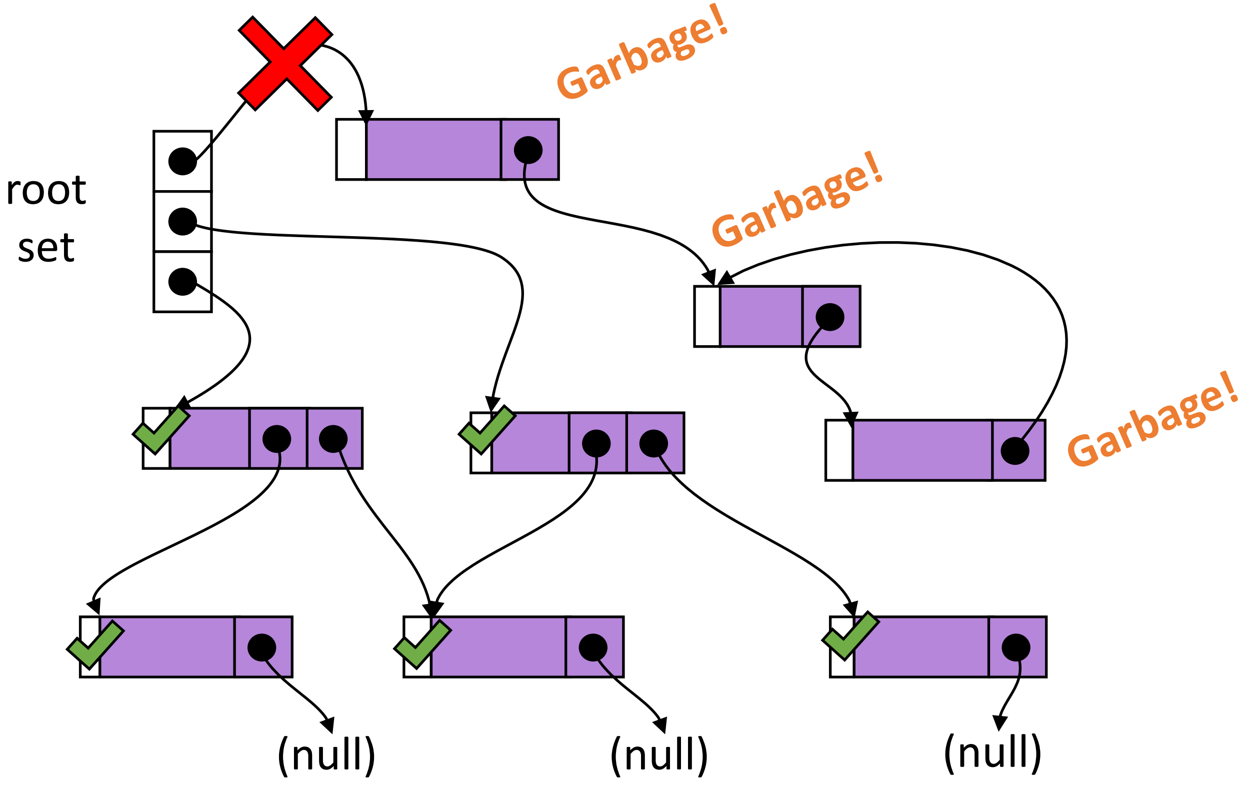
Heap Management: Garbage Collection

“Lazy” garbage collection

- (Can be) performed when needed

Two-phase approach:

- Mark – Traverse memory from the roots, set a “mark bit” on each cell
- Sweep – Free all memory that wasn’t marked



Mark and Sweep - Tradeoffs

Heap Management: Garbage Collection

Space Overhead - Low

- Only need 1 bit per cell

Time Overhead - High

- Need to traverse all data structures

Summary

Heap Management: Garbage Collection

Compiler-adjacent topic

- Probably implemented in a library and linked into the code
- Still an important aspect of the design and implementation of a language!

Finished the basic workflow for the compiler!

Next Time

Lecture Preview

How do we go from assembly code to an executable?

- The postcompilation toolchain
  - The assembler
  - The linker
  - The loader