

Concepts of Programming Languages

Memory management

Prof. Dr. Guido Salvaneschi

Outline

- **Why memory management?**
- **Reference counting**
- **Garbage collection**
- **Summary**

Creating Mutable Boxes

- Interpret `valueExpr`
 - resulting in a `value` and potentially updated `store`.
- Get new location in the most recent store.
- Put value into new location and store this location in a box value.
- Return the box value and the updated store.

```
def interp(expr: SCFLAE, env: Env, store: Store): (Val, Store) = expr match {  
  //...  
  case NewBox(valueExpr) => {  
    val (value, boxStore) = interp(valueExpr, env, store)  
    val newLoc = nextLocation  
    (Box(newLoc), boxStore + (newLoc -> value))  
  }  
}
```

We allocate a new memory cell

Adding State Syntactically

- ... will add constructs for defining boxes, for setting/getting the value in a box, for changing the value of a variable, and for sequencing two expressions.

```
<BCFLAE> ::= <num>
           | {+ <SCFLAE> <SCFLAE>}
           | {- <SCFLAE> <SCFLAE>}
           | {with {<id> <SCFLAE>} <SCFLAE>}
           | <id>
           | {fun {<id>} <SCFLAE>}
           | {<SCFLAE> <SCFLAE>}
           | {* <SCFLAE> <SCFLAE>}
           | {if0 <SCFLAE> <SCFLAE> <SCFLAE>}
```

```
| {newbox <SCFLAE>}
| {setbox <SCFLAE> <SCFLAE>}
| {openbox <SCFLAE>}
| {set <id> <SCFLAE>}
| {seqn <SCFLAE> <SCFLAE>}
```

**No operation for deallocation.
What will happen eventually?**

```
def whatDoesThisDo(n: Int) : Expr = {  
  var v: Expr = Num(17)  
  for (i <- 1 to n)  
    v = Seqn(NewBox(i), v)  
  v  
}
```

```
interp(whatDoesThisDo(10))  
interp(whatDoesThisDo(100))  
interp(whatDoesThisDo(1000))  
interp(whatDoesThisDo(10000))  
interp(whatDoesThisDo(100000))  
interp(whatDoesThisDo(1000000))
```

Quiz: What do these code pieces evaluate to?

**Heap overflow:
out of memory**



Adding State Syntactically

- ... will add constructs for defining boxes, for setting/getting the value in a box, for changing the value of a variable, and for sequencing two expressions.

```
<BCFLAE> ::= <num>
          | {+ <SCFLAE> <SCFLAE>}
          | {- <SCFLAE> <SCFLAE>}
          | {with {<id> <SCFLAE>} <SCFLAE>}
          | <id>
          | {fun {<id>} <SCFLAE>}
          | {<SCFLAE> <SCFLAE>}
          | {* <SCFLAE> <SCFLAE>}
          | {if0 <SCFLAE> <SCFLAE> <SCFLAE>}
```

```
| {newbox <SCFLAE>}
| {setbox <SCFLAE> <SCFLAE>}
| {openbox <SCFLAE>}
| {set <id> <SCFLAE>}
| {seqn <SCFLAE> <SCFLAE>}
```

No operation for deallocation.

Quiz: Should we add a deallocation operation?

Manual memory management

- Popular until mid-1990s
- Today: mostly used in C and C++
- Problems:
 - **Memory leak**: developer forgets to release unused object
 - **Heap corruption**: developer releases a single object multiple times
 - **Dangling pointers**: after release, pointers to the released object become dangling
 - **Distraction** from actual programming task

What happens in a double delete?



15



6

```
Obj *op = new Obj;  
Obj *op2 = op;  
delete op;  
delete op2; // What happens
```

What's the worst that can happen when the compiler going to throw an error?

c++

pointers

memory-leaks



14



It causes undefined behaviour. Anything can happen. In practice, a runtime crash is probably what I'd expect.

share edit flag

answered Feb 7 '12 at 1:23



Carl Norum

149k • 19 • 272 • 355

6 Random puppy might die as well. – [Petr](#) Nov 21 '15 at 9:14

1 There was a known case of a persons monitor lighting on fire due to undefined behavior. – [Enn Michael](#) Jul 5 at 10:35

[add a comment](#)



17



Undefined behavior. There are no guarantees whatsoever made by the standard. Probably your operating system makes some guarantees, like "you won't corrupt another process", but that doesn't help your program very much.

Your program could crash. Your data could be corrupted. The direct deposit of your next paycheck could instead take 5 million dollars out of your account.

share edit flag

answered Feb 7 '12 at 1:24



Ben Voigt

205k • 21 • 242 • 459

19 Or it could order pizza with your credit card. – [Seth Carnegie](#) Feb 7 '12 at 1:29

Automatic memory management

In this lecture, we want to investigate mechanisms for automatic memory management:

- **Reference counting:**
keeping track of references to an *object*
- **Garbage collection:**
scan heap for unused objects

Outline

- **Why memory management?**
- **Reference counting**
- **Garbage collection**
- **Summary**

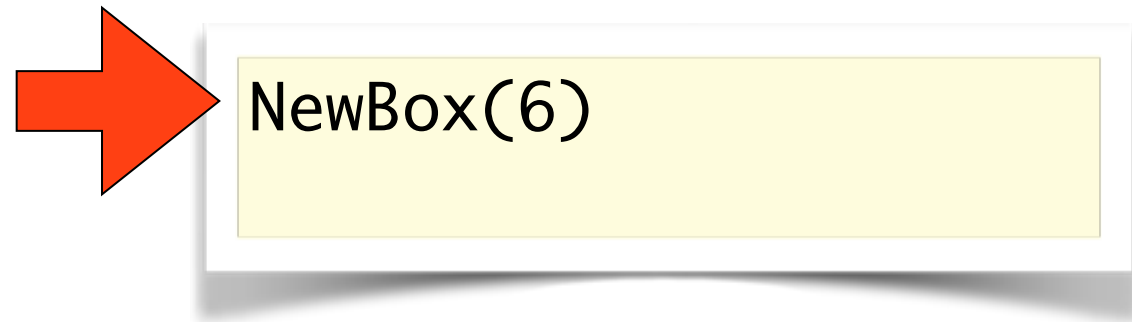
Reference counting

Simple idea:

- Keeping track of number of references for each object
- If the number reaches 0, the object can be released

Reference counting

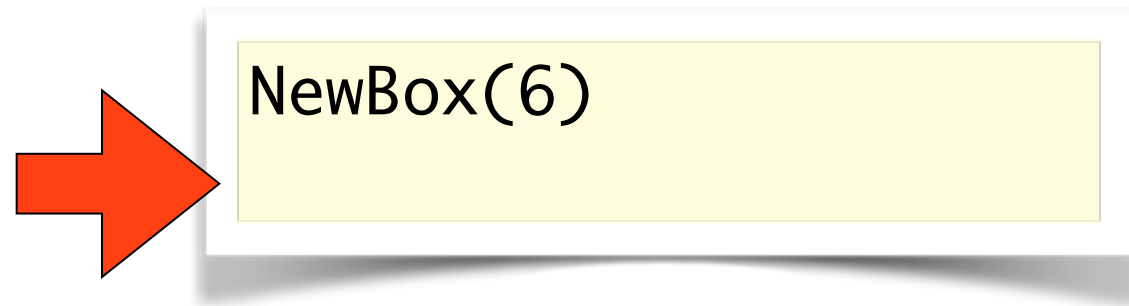
Example:



1	(num 6)	1

Reference counting

Example:

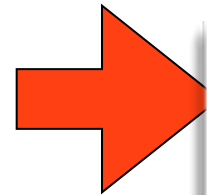


1	(num 6)	0

No reference to cell 1 => release

Reference counting

Example:

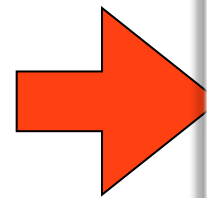


```
{val x = NewBox(6)  
  val y = NewBox(x)}
```

1	(num 6)	1

Reference counting

Example:

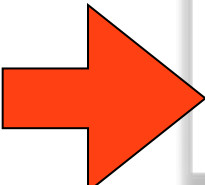


```
{val x = NewBox(6)  
 val y = NewBox(x)}
```

1	(num 6)	2
2	(box 1)	1

Reference counting

Example:



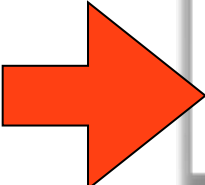
```
{val x = NewBox(6)  
  val y = NewBox(x)}
```

1	(num 6)	1
2	(box 1)	0

No reference to cell 2 => release

Reference counting

Example:



```
{val x = NewBox(6)  
  val y = NewBox(x)}
```

1	(num 6)	0
2		0

No reference to cell 2 => release

No reference to cell 1 => release

Reference counting

Simple idea:

- keeping track of number of references for each object
- if the number reaches 0, the object can be released
- releasing a single object can free other objects

Let's implement this in our interpreter.

Issue with reference counting

Simple idea:

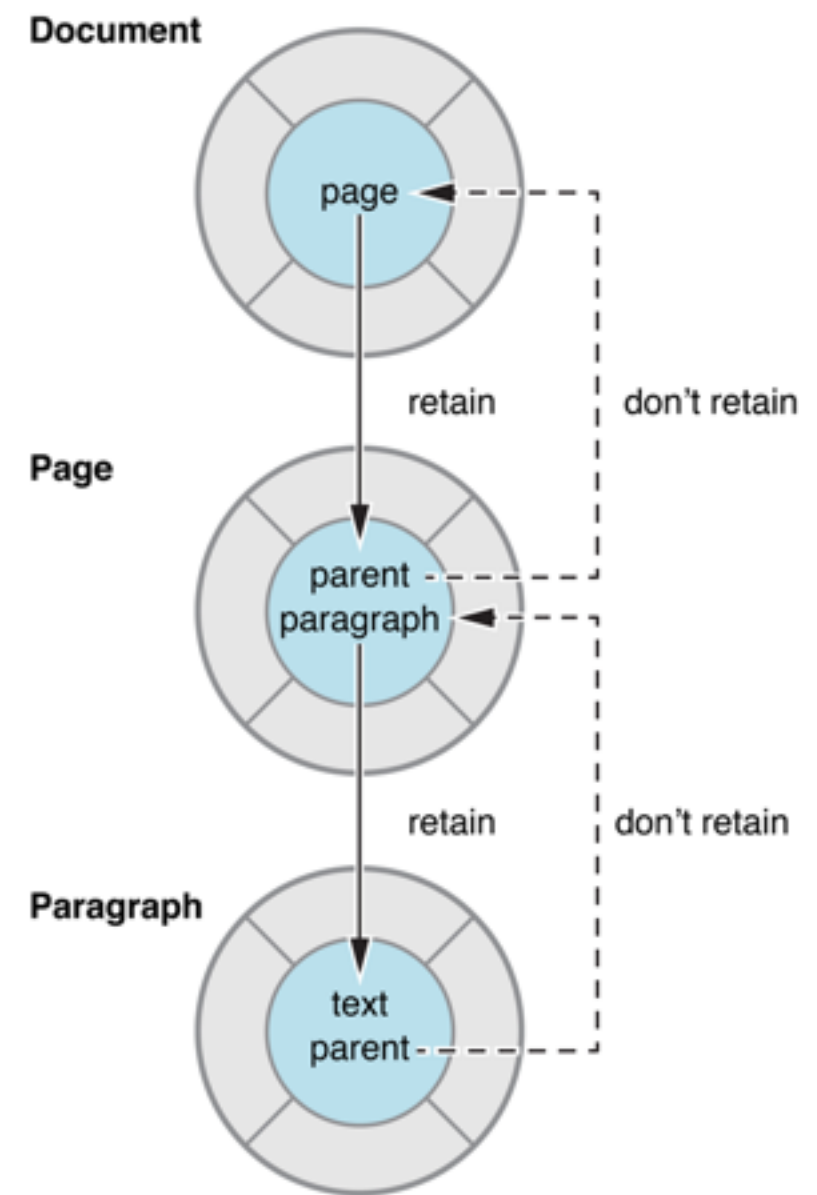
- keeping track of number of references for each object
- if the number reaches 0, the object can be released
- releasing a single object can free other objects

Quiz: Reference counting has an important limitation. What is it?

- no support for cyclic data structures: reference count never reaches 0

Reference Counting: Handling Cycles

- Design the system to avoid creating them.
 - E.g., forbid reference cycles like in file systems with hard links.
- Weak references
 - "weak" references are non-counted references
 - E.g., in Cocoa recommends using "strong" references for parent-to-child relationships and "weak" references for child-to-parent relationships.



Cocoa (Manual) Reference Counting

```
- (void)createAndGiveAwayTheGroceryList
{
    // Create a list
    GroceryList *g = [[GroceryList alloc] init];

    // (The retain count of g is 1)

    // Share it with your friend who retains it
    [smartFriend takeGroceryList:g];

    // (The retain count of g is 2)

    // Give up ownership
    [g release];

    // (The retain count of g is 1)
    // But we don't really care here, as this
    // method's responsibility is finished.
}
```

```
- (void)takeGroceryList:(GroceryList *)x
{
    // Take ownership
    [x retain];

    // Hold onto a pointer to the object
    myList = x;
}
```

```
- (id)retain
{
    retainCount++;
    return self;
}
- (void)release
{
    retainCount--;
    if (retainCount == 0)
        [self dealloc];
}
```

Automatic Reference Counting

- Reference counting is accomplished by the programmer.
 - E.g., by sending `retain` and `release` messages to objects
- With **Automatic** Reference Counting these messages are inserted by the compiler as needed.
 - Such feature, based on the Clang compiler, was added in iOS 5 and Mac OS X 10.7.

**The interpreter uses
automatic RC.**

- Note: retrofitting GC is hard!
 - Mac OS X 10.5 introduced a tracing garbage collector as an alternative to reference counting, but it was deprecated in OS X 10.8.

Outline

- **Why memory management?**
- **Reference counting**
- **Garbage collection**
- **Summary**

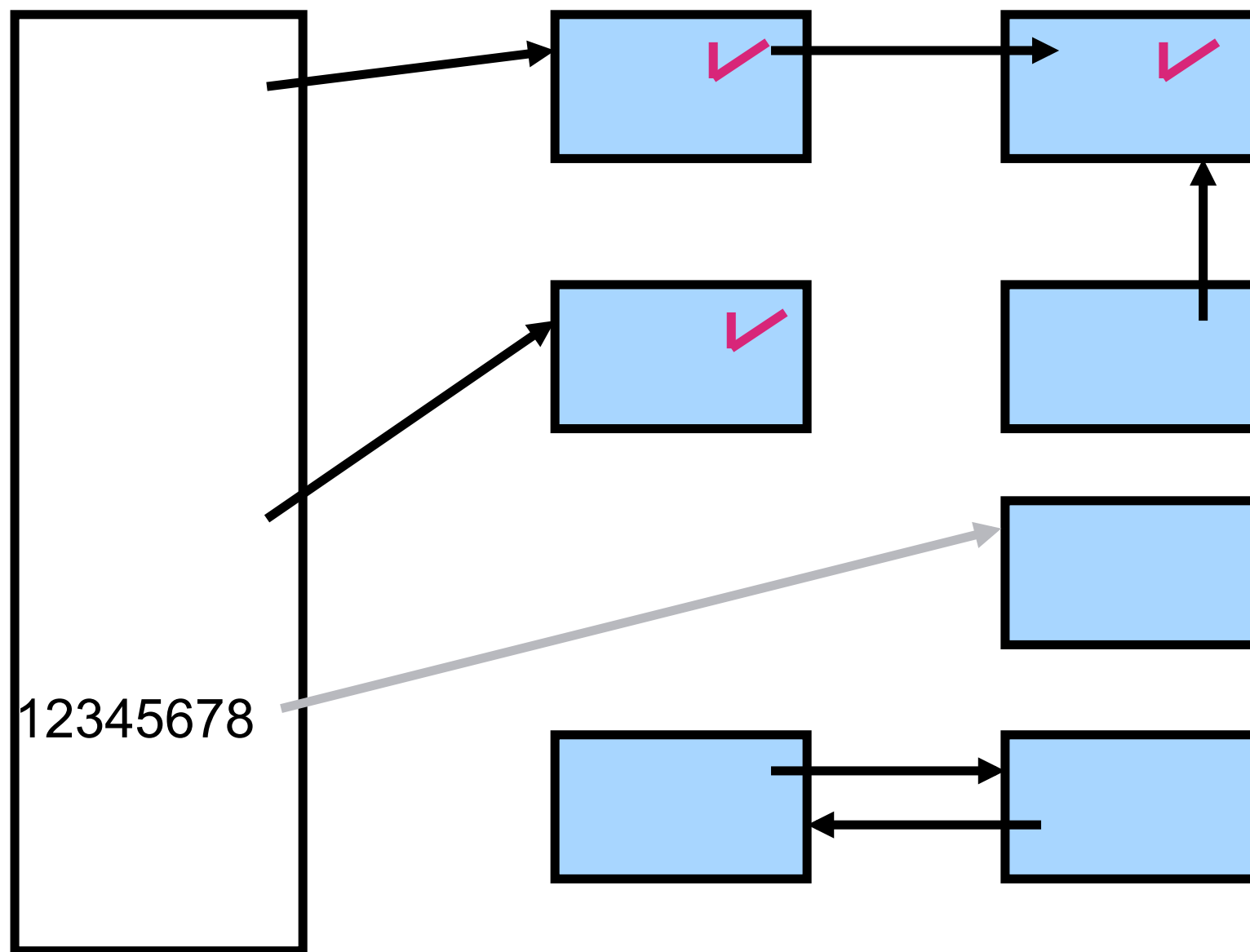
Mark&Sweep garbage collection

Simple idea of Mark&Sweep:

- when memory runs low, scan the heap
- mark objects that are referenced on the stack
 - transitively: mark objects referenced by marked objects
- sweep: release all objects without a mark
- reset the marks

Mark&Sweep garbage collection

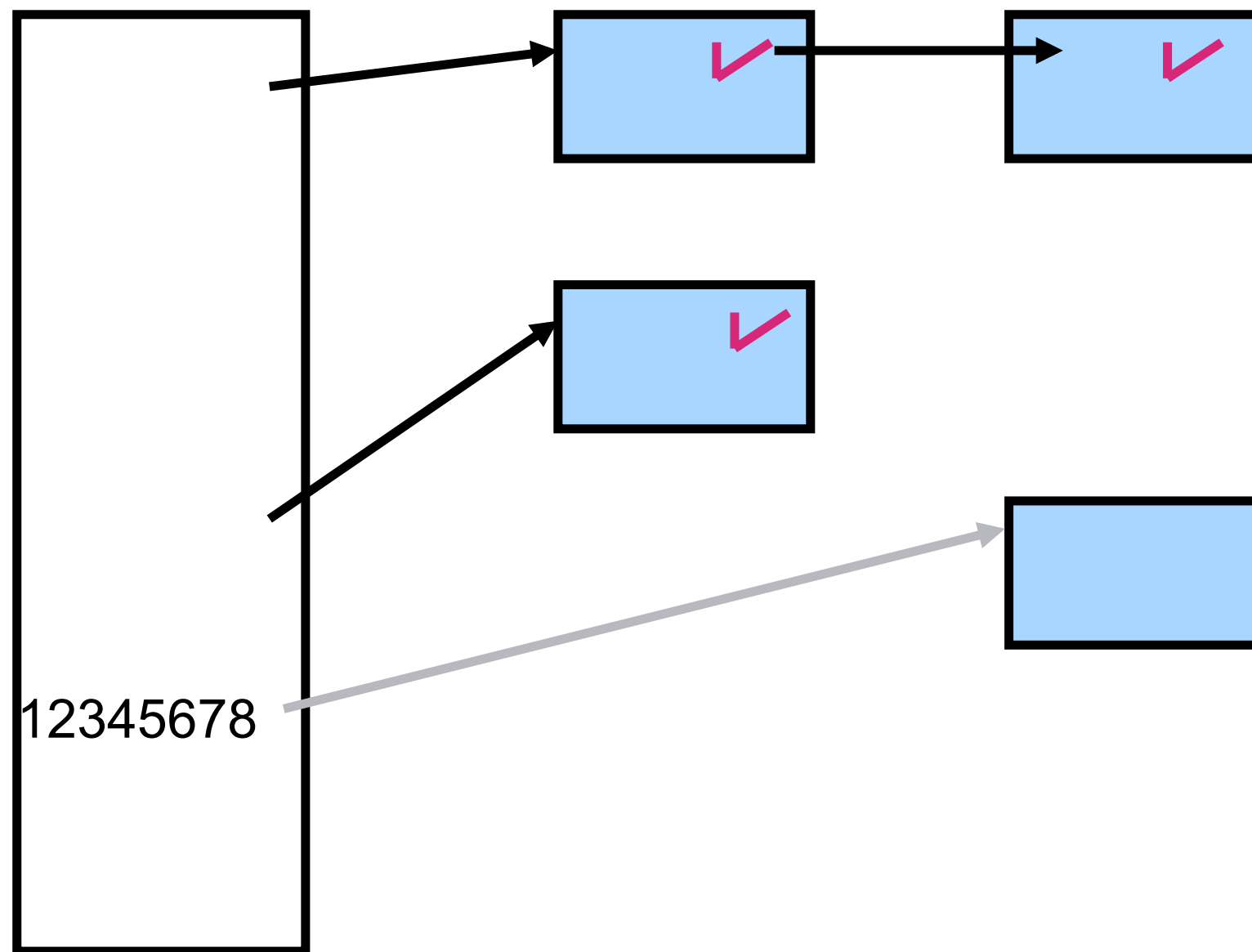
Example:



Stack w/ pointer variables

Mark&Sweep garbage collection

Example:



Stack w/ pointer variables

Mark&Sweep garbage collection

Simple idea of Mark&Sweep:

- when memory runs low, scan the heap
- mark objects that are referenced from the stack
 - transitively: mark objects referenced by marked objects
- sweep: release all objects without a mark
- reset the marks

Let's implement this in our interpreter.

Issues with Mark&Sweep GC

Simple idea of Mark&Sweep:

- when memory runs low, scan the heap
- mark objects that are referenced on the stack
 - transitively: mark objects referenced by marked objects
- sweep: release all objects without a mark
- reset the marks

Quiz: Our GC has some limitations. What are they?

stop-the-world
touches all (virtual) memory pages
heap fragmentation

Outline

- **Why memory management?**
- **Reference counting**
- **Garbage collection**
- **Summary**

Comparison

Reference counting:

- + incremental: deallocate as soon as object is unreferenced
- + re-use freed cells immediately
- space and time overhead: keeping track of references
- no support for cyclic structures

Mark&sweep GC:

- + handles cycles
- + small space overhead (1 bit per cell)
- requires stopping of regular program execution
- fragmentation: slower allocation, cache misses

Other GC techniques

Moving: collect live objects in one part of heap

- + faster allocation, no mark reset for free memory region
- all references must be managed by GC, no pointer arithmetic

Copying: Divide heap into from-space and to-space. When from-space is full, copy only live objects to to-space. Flip roles of spaces.

- + very fast allocation
- can only use half of available memory

Generational: young objects are more likely to die => divide heap into different generations. Run GC on younger generations more frequently.

Garbage Collector



No GC

Areas that flash bright green or yellow are memory reads or writes

The color decays over time so you can see how memory was used, but also see current activity.

Patterns emerge where the program begins to ignore some memory

Garbage Collector

Reference Counting

Red = RF activity

Memory is free as soon as possible

Arithmetic is fast but memory is slow: need to continuously access counters

Garbage Collector

Mark and Sweep

Handles cycles

No counters: less overhead

Pass on all memory can take a long time

Garbage Collector

Mark Compact

Copy objects to remove gaps

Cheap object allocation: simply use the first available cell in memory.

Better memory access patterns

Garbage Collector

Copy GC

Live objects are copied to another space

No mark phase for dead objects