

Memory Management

Lecture 20 CS169

Announcements

- Shifting schedule by one lecture
 - Make time to discuss designs more
- Presentations
 - April 11, 13
- Midterm
 - April 18
- Design Doc
 - Due April 8th at 5pm

Outline

- Overview of memory management
 - Why it is a software engineering issue
- Styles of memory management
 - Malloc/free
 - Garbage collection
 - Regions

Memory Management

- A basic decision, because
 - Different memory management policies are difficult to mix
 - Best to stick with one in an application
 - Has a big impact on performance and quality
 - Different strategies better in different situations
 - Some more error prone than others

Distinguishing Characteristics

- Allocation is always explicit
- Deallocation
 - Explicit or implicit?
- Safety
 - Checks that explicit deallocation is safe?

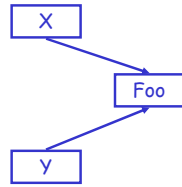
Explicit Memory Management

- Allocation and deallocation are explicit
 - Oldest style
 - C, C++

```
x = new Foo;  
...  
free(x);
```

A Problem: Dangling Pointers

```
X = new Foo;  
...  
Y = X;  
...  
free(X);  
...  
Y.bar();
```

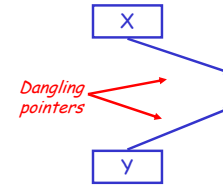


Prof. Brewer CS 169 Lecture 20

7

A Problem: Dangling Pointers

```
X = new Foo;  
...  
Y = X;  
...  
free(X);  
...  
Y.bar();
```



Prof. Brewer CS 169 Lecture 20

8

Notes

- Dangling pointers are bad
 - A system crash waiting to happen
- Storage bugs are hard to find
 - Visible effect far away (in time and program text) from the source
- Not the only potentially bad memory bug in C
 - But the other can be fixed by type systems

Prof. Brewer CS 169 Lecture 20

9

Notes, Continued

- Explicit deallocation is not all bad
- Gives the finest possible control over memory
 - May be important in memory-limited applications
- Programmer is very conscious of how much memory is in use
 - This is good and bad
- Allocation and deallocation fairly expensive

Prof. Brewer CS 169 Lecture 20

10

Automatic Memory Management

- I.e., automatic deallocation
- This is an old problem:
 - studied since the 1950s for LISP
- There are well-known techniques for completely automatic memory management
- Until recently unpopular outside of Lisp family languages

Prof. Brewer CS 169 Lecture 20

11

The Basic Idea

- When an object is created, unused space is automatically allocated
 - E.g., `new X`
 - As in all memory management systems
- After a while there is no more unused space
- Some space is occupied by objects that will never be used again
 - This space can be freed to be reused later

Prof. Brewer CS 169 Lecture 20

12

The Basic Idea (Cont.)

- How can we tell whether an object will "never be used again"?
 - in general, impossible to tell
 - use heuristics
- Observation: a program can use only the objects that it can find:
 - `A x = new A; x = y; ...`
 - After `x = y` there is no way to access the newly allocated object

Prof. Brewer CS 169 Lecture 20

13

Garbage

- An object `x` is reachable if and only if:
 - a register contains a pointer to `x`, or
 - another reachable object `y` contains a pointer to `x`
- You can find all reachable objects by starting from registers and following all the pointers
- An unreachable object can never be used
 - such objects are garbage

Prof. Brewer CS 169 Lecture 20

14

Reachability is an Approximation

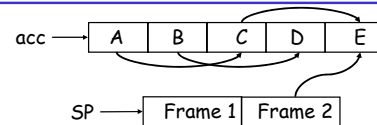
- Consider the program:

```
x = new A;
y = new B;
x = y;
if(alwaysTrue()){ x = new A } else { x.foo() }
```
- After `x = y` (assuming `y` becomes dead there)
 - the object `A` is unreachable
 - the object `B` is reachable (through `x`)
 - thus `B` is not garbage and is not collected
 - but object `B` is never going to be used

Prof. Brewer CS 169 Lecture 20

15

A Simple Example



- We start tracing from registers and stack
 - These are the *roots*
- Note `B` and `D` are unreachable from `acc` and stack
 - Thus we can reuse their storage

Prof. Brewer CS 169 Lecture 20

16

Elements of Garbage Collection

- Every garbage collection scheme has the following steps
 1. Allocate space as needed for new objects
 2. When space runs out:
 - a) Compute what objects might be used again (generally by tracing objects reachable from a set of "root" registers)
 - b) Free the space used by objects not found in (a)
- Some strategies perform garbage collection before the space actually runs out

Prof. Brewer CS 169 Lecture 20

17

Notes on Garbage Collection

- *Much* safer than explicit memory management
 - Crashes due to memory errors disappear
 - And easy to use
- But exacerbates other problems
 - Memory leaks can be hard to find
 - Because memory usage in general is hidden
 - Different GC approaches have different performance trade-offs

Prof. Brewer CS 169 Lecture 20

18

Notes (Continued)

- Fastest GCs do not perform well if live data is significant percentage of physical memory
 - Should be < 30%
 - If > 50%, quite dramatic performance degradation
- Pauses are not acceptable in some applications
 - Use real-time GC, which is more expensive
- Allocation can be very fast
- Amortized deallocation can be very fast, too

Prof. Brewer CS 169 Lecture 20

19

Finding Memory Leaks

- A simple automatic technique is effective at finding memory leaks
- Record allocations and accesses to objects
- Periodically check
 - Live objects that have not been used in some time
 - These are likely leaked objects
- This can find bugs even in GC languages!

Prof. Brewer CS 169 Lecture 20

20

A Different Approach: Regions

- Traditional memory management:

	free	GC
Safety	-	+
Control	+	-
Ease of use	-	+
Space usage	+	-

- A different approach: regions
safety and efficiency, expressiveness

Prof. Brewer CS 169 Lecture 20

21

Region-based Memory Management

- Regions represent areas of memory
- Objects are allocated "in" a given region
- Easy to deallocate a whole region

```
Region r = newregion();
for (i = 0; i < 10; i++) {
    int *x = ralloc(r, (i + 1) * sizeof(int));
    work(i, x);
}
deleteregion(r);
```

Prof. Brewer CS 169 Lecture 20

22

Policy Choices

- Deallocation
 - Garbage collection (GC)
 - per-object free (per-object)
 - **region deletion (all-at-once)**
 - implicit vs explicit
- Safety
 - none (none)
 - reachability (GC)
 - per-region reference counting (RC)
 - statically checked (static)

Prof. Brewer CS 169 Lecture 20

23

Why Regions ?

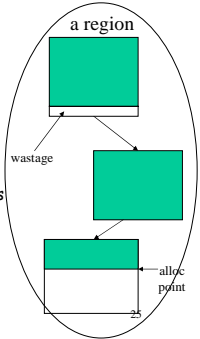
- Performance
- Locality benefits
- Expressiveness
- Memory safety

Prof. Brewer CS 169 Lecture 20

24

Region Performance: Allocation and Deallocation

- Applies to delete all-at-once only
- Basic strategy:
 - Allocate a big block of memory
 - Individual allocation is:
 - pointer increment
 - overflow test
 - Deallocation frees the list of big blocks
- * All operations are fast



Prof. Brewer CS 169 Lecture 20

Region Performance: Locality

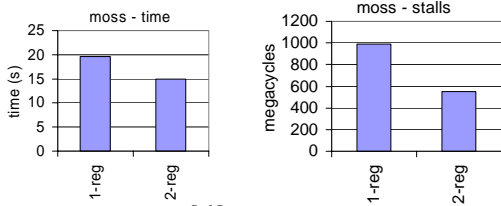
- Regions can express locality:
 - Sequential allocs in a region can share cache line
 - Allocs in different regions less likely to pollute cache for each other
- Example: moss (plagiarism detection software)
 - Small objects: short lived, many clustered accesses
 - Large objects: few accesses

Prof. Brewer CS 169 Lecture 20

26

Region Performance: Locality - moss

- 1-region version: small & large objects in 1 region
- 2-region version: small & large objects in 2 regions
- 45% less cycles lost to r/w stalls in 2-region version



Prof. Brewer CS 169 Lecture 20

27

Region Expressiveness

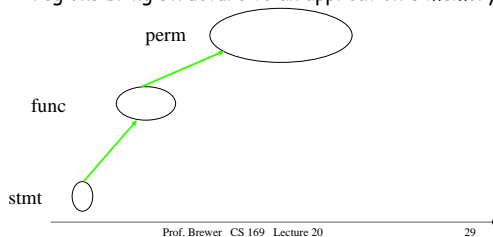
- Adds some structure to memory management
- Few regions:
 - Easier to keep track of
 - Delay freeing to convenient "group" time
 - End of an iteration, closing a device, etc
- No need to write "free this data structure" functions

Prof. Brewer CS 169 Lecture 20

28

Region Expressiveness: lcc

- The lcc C compiler, written using unsafe regions
 - regions bring structure to an application's memory

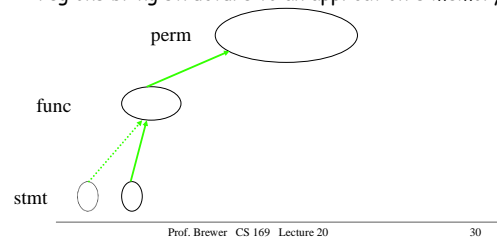


Prof. Brewer CS 169 Lecture 20

29

Region Expressiveness: lcc

- The lcc C compiler, written using unsafe regions
 - regions bring structure to an application's memory

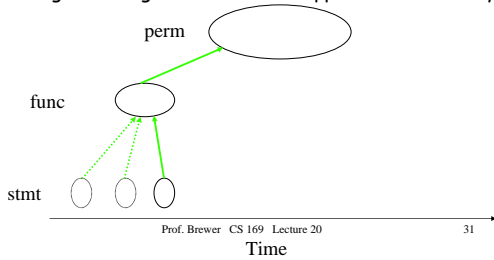


Prof. Brewer CS 169 Lecture 20

30

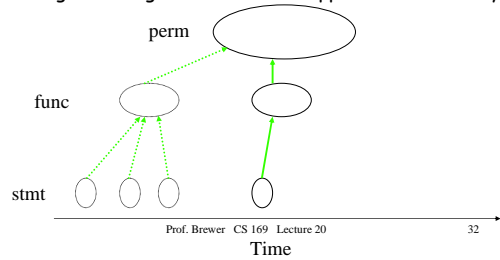
Region Expressiveness: lcc

- The lcc C compiler, written using unsafe regions
 - regions bring structure to an application's memory



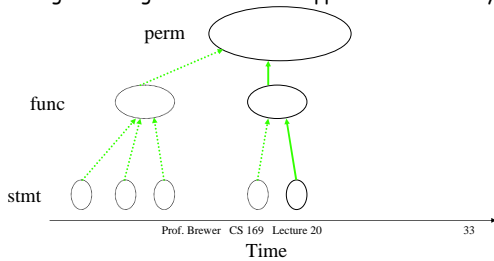
Region Expressiveness: lcc

- The lcc C compiler, written using unsafe regions
 - regions bring structure to an application's memory



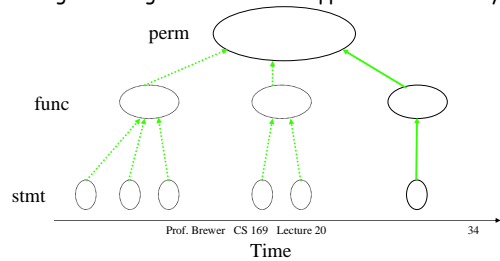
Region Expressiveness: lcc

- The lcc C compiler, written using unsafe regions
 - regions bring structure to an application's memory



Region Expressiveness: lcc

- The lcc C compiler, written using unsafe regions
 - regions bring structure to an application's memory



Summary

	regions	free	GC
Safety	+	-	+
Control	++	+	-
Ease of use	=	-	+
Space usage	+	+	-
Time	+	+	+

Region Notes

- Regions are fast
 - Very fast allocation
 - Very fast (amortized) deallocation
 - Can express locality
 - Only known technique for doing so
- Good for memory-intensive programs
 - Efficient and fast even if high % of memory in use

Region Notes (Continued)

- Does waste some memory
 - In between malloc/free and GC
- Requires more thought than GC
 - Have to organize allocations into regions

Prof. Brewer CS 169 Lecture 20

37

Summary

- You must pay attention to memory management
 - Can affect the design of many system components
- For applications with low-memory, no real time constraints, use GC
 - Easiest strategy for programmer
- For high-memory or high-performance applications, use regions
- Malloc/Free not really recommended

Prof. Brewer CS 169 Lecture 20

38