CS 241 Honors
Memory

Ben Kurtovic Atul Sandur Bhuvan Venkatesh Brian Zhou
Kevin Hong

University of Illinois Urbana–Champaign

February 20, 2018

Overview

- Memory review
- Management techniques
  - Heap vs. stack
  - `free()`
  - `alloca()`
  - Smart pointers (C++)
- Garbage collection
  - Reference counting
  - Mark-and-sweep
  - Copying collector
  - Generational
  - In practice
- Concluding thoughts

Types of variables in program memory

- Static variables
  - Persist throughout the lifetime of the program
  - No runtime cost, but memory cannot be used for anything else

Types of variables in program memory

- Static variables
  - Persist throughout the lifetime of the program
  - No runtime cost, but memory cannot be used for anything else
- Local variables, function arguments
  - Live for the lifetime of the function call
  - Stored on the stack
  - Allocation and reclaiming of memory is really cheap—why?

Types of variables in program memory

- Static variables
  - Persist throughout the lifetime of the program
  - No runtime cost, but memory cannot be used for anything else
- Local variables, function arguments
  - Live for the lifetime of the function call
  - Stored on the stack
  - Allocation and reclaiming of memory is really cheap—why?
- Heap allocation
  - Slower than stack allocation because of bookkeeping by memory manager
  - This memory is not automatically reclaimed, so will eventually fill up
  - Call `free()`! But it isn't always obvious when you are done with something

`malloc()` and `free()`

What's the big deal?

You all know from using it—manual heap memory management is a pain

Why is heap management so difficult?

- Manual bookkeeping (programming error)
  - Dangling pointers, double deletion, memory leaks
  - Don't always know when to clean up memory

Why is heap management so difficult?

- Manual bookkeeping (programming error)
  - Dangling pointers, double deletion, memory leaks
  - Don't always know when to clean up memory
- Performance tradeoffs with heap memory management
  - Linked list of free blocks, how expensive to get a new block of memory to allocate?
  - Pre-coalesce blocks so allocation can be faster? `free` becomes expensive

Why is heap management so difficult?

- Manual bookkeeping (programming error)
  - Dangling pointers, double deletion, memory leaks
  - Don't always know when to clean up memory
- Performance tradeoffs with heap memory management
  - Linked list of free blocks, how expensive to get a new block of memory to allocate?
  - Pre-coalesce blocks so allocation can be faster? `free` becomes expensive
- Fragmentation
  - Bad locality of reference
  - Problem in virtual memory systems due to page faults, cache misses

free(), continued

Keeping mental track of your heap memory gets frustrating in complex programs, especially if you're trying to be careful about error handling

```
a = malloc(...);
b = malloc(...);

if (/* do something and it fails */) {
  free(a);
  free(b);
  return;
}

if (/* do another thing */) {
  if (/* do a third thing and it fails */) {
    free(a);
    free(b);
    return;
  }
  /* ... */
}

c = malloc(...);
if (/* do a fourth thing and it fails */) {
  free(c);
  free(b);
  free(a);
  return;
}

/* ... */
```

free(), continued (the "goto cleanup" idiom)

```c
a = malloc(...);
b = malloc(...);

if (/* do something and it fails */)
  goto cleanup;

if (/* do another thing */) {
  if (/* do a third thing and it fails */)
    goto cleanup;
  /* ... */
}

c = malloc(...);
if (/* do a fourth thing and it fails */)
  goto cleanup;

/* ... */

cleanup:
free(a);
free(b);
free(c);
```

free(), continued (the "goto cleanup" idiom)

```
a = malloc(...);
b = malloc(...);

if (/* do something and it fails */)
  goto cleanup;

if (/* do another thing */) {
  if (/* do a third thing and it fails */)
    goto cleanup;
  /* ... */
}

c = malloc(...);
if (/* do a fourth thing and it fails */)
  goto cleanup;

/* ... */

cleanup:
free(a);
free(b);
free(c);
```

- Reduces usage of `free()` and makes keeping track of variables easier—only need to think about it at the end of the function
- Can be generalized to other resource types (opening/closing files)

free(), continued (the "goto cleanup" idiom)

```c
a = malloc(...);
b = malloc(...);

if (/* do something and it fails */)
  goto cleanup;

if (/* do another thing */) {
  if (/* do a third thing and it fails */)
    goto cleanup;
  /* ... */
}

c = malloc(...);
if (/* do a fourth thing and it fails */)
  goto cleanup;

/* ... */

cleanup:
free(a);
free(b);
free(c);
```

- Reduces usage of `free()` and makes keeping track of variables easier—only need to think about it at the end of the function
- Can be generalized to other resource types (opening/closing files)
- Makes things easier, but still have manual heap management

So, how can we take advantage of the stack?

So, how can we take advantage of the stack?

Let's take a look at some of its limitations...

alloca()

Stack memory is cheap, but often limited by fixed sizes

```
char buffer[512];
puts("What's your name?");
fgets(buffer, sizeof(buffer), stdin);
```

alloca()

Stack memory is cheap, but often limited by fixed sizes

```
char buffer[512];
puts("What's your name?");
fgets(buffer, sizeof(buffer), stdin);
```

What if the person has a **really** long name?

alloca()

Stack memory is cheap, but often limited by fixed sizes

```
char buffer[512];
puts("What's your name?");
fgets(buffer, sizeof(buffer), stdin);
```

What if the person has a **really** long name?

```
if (/* entire name wasn't read */) {
  buffer = realloc(buffer, 2 * sizeof(buffer));
  fgets(...);
}
```

alloca()

Stack memory is cheap, but often limited by fixed sizes

```c
char buffer[512];
puts("What's your name?");
fgets(buffer, sizeof(buffer), stdin);
```

What if the person has a **really** long name?

```c
if (/* entire name wasn't read */) {
  buffer = realloc(buffer, 2 * sizeof(buffer));
  fgets(...);
}
```

```
*** Error in `./name': realloc(): invalid pointer: 0x00007fffb3af9250 ***
======= Backtrace: =========
/lib64/libc.so.6(+0x7bc17)[0x7f02fc21dc17]
/lib64/libc.so.6(realloc+0x31e)[0x7f02fc2228fe]
...
Aborted (core dumped)
```

You can't `realloc()` stack memory

alloca()

Stack memory is cheap, but often limited by fixed sizes

```c
char buffer[512];
puts("What's your name?");
fgets(buffer, sizeof(buffer), stdin);
```

What if the person has a **really** long name?

```c
if (/* entire name wasn't read */) {
  buffer = realloc(buffer, 2 * sizeof(buffer));
  fgets(...);
}
```

```
*** Error in `./name': realloc(): invalid pointer: 0x00007fffb3af9250 ***
======= Backtrace: =========
/lib64/libc.so.6(+0x7bc17)[0x7f02fc21dc17]
/lib64/libc.so.6(realloc+0x31e)[0x7f02fc2228fe]
...
Aborted (core dumped)
```

You can't `realloc()` stack memory... *or can you?*

alloca()

Stack memory is cheap, but often limited by fixed sizes

```c
char buffer[512];
puts("What's your name?");
fgets(buffer, sizeof(buffer), stdin);
```

What if the person has a **really** long name?

```c
if (/* entire name wasn't read */) {
  buffer = realloc(buffer, 2 * sizeof(buffer));
  fgets(...);
}
```

```
*** Error in `./name': realloc(): invalid pointer: 0x00007fffb3af9250 ***
======= Backtrace: =========
/lib64/libc.so.6(+0x7bc17)[0x7f02fc21dc17]
/lib64/libc.so.6(realloc+0x31e)[0x7f02fc2228fe]
...
Aborted (core dumped)
```

You can't `realloc()` stack memory... *or can you?* (you can't)

alloca()

But, there *is* another way

```c
#include <alloca.h>

void *alloca(size_t size);
```

alloca()

But, there *is* another way

```
#include <alloca.h>

void *alloca(size_t size);
```

Use it exactly like `malloc()`, but memory is allocated on the stack rather than heap

- No `free()` necessary! (It would crash, like before)
- Thus, memory leaks are impossible
- Cheap and fast (no heap memory management)

alloca()

But, there *is* another way

```
#include <alloca.h>

void *alloca(size_t size);
```

Use it exactly like `malloc()`, but memory is allocated on the stack rather than heap

- No `free()` necessary! (It would crash, like before)
- Thus, memory leaks are impossible
- Cheap and fast (no heap memory management)

What's wrong with this?

alloca()

But, there *is* another way

```
#include <alloca.h>

void *alloca(size_t size);
```

Use it exactly like `malloc()`, but memory is allocated on the stack rather than heap

- No `free()` necessary! (It would crash, like before)
- Thus, memory leaks are impossible
- Cheap and fast (no heap memory management)

What's wrong with this?

- Stack scoping rules: can't return a pointer to stack space!
- Overflow: stack space is limited; can't allocate too much memory

alloca()

But, there *is* another way

```
#include <alloca.h>

void *alloca(size_t size);
```

Use it exactly like `malloc()`, but memory is allocated on the stack rather than heap

- No `free()` necessary! (It would crash, like before)
- Thus, memory leaks are impossible
- Cheap and fast (no heap memory management)

What's wrong with this?

- Stack scoping rules: can't return a pointer to stack space!
- Overflow: stack space is limited; can't allocate too much memory
- Non-standard! (Not part of ANSI C, POSIX, etc...)

alloca()

But, there *is* another way

```
#include <alloca.h>

void *alloca(size_t size);
```

Use it exactly like `malloc()`, but memory is allocated on the stack rather than heap

- No `free()` necessary! (It would crash, like before)
- Thus, memory leaks are impossible
- Cheap and fast (no heap memory management)

What's wrong with this?

- Stack scoping rules: can't return a pointer to stack space!
- Overflow: stack space is limited; can't allocate too much memory
- Non-standard! (Not part of ANSI C, POSIX, etc...)

So, not a good idea

C++ smart pointers

New feature in C++11 standard library (although concept existed before)

C++ smart pointers

New feature in C++11 standard library (although concept existed before)

Concept of *ownership*:

C++ smart pointers

New feature in C++11 standard library (although concept existed before)

Concept of *ownership*:

- Instead of passing pointers around like mad, let's set some rules

C++ smart pointers

New feature in C++11 standard library (although concept existed before)

Concept of *ownership*:

- Instead of passing pointers around like mad, let's set some rules
- RAII (Resource Acquisition Is Initialization)—remember from CS 225, maybe?
  - Only allocate memory in your constructor and free it in your destructor
  - Put things on the stack as much as possible

C++ smart pointers

New feature in C++11 standard library (although concept existed before)

Concept of *ownership*:

- Instead of passing pointers around like mad, let's set some rules
- RAII (Resource Acquisition Is Initialization)—remember from CS 225, maybe?
  - Only allocate memory in your constructor and free it in your destructor
  - Put things on the stack as much as possible
- "Smart pointer" wraps an ordinary (raw) pointer and defines the semantics for how it is managed and who's managing it
- Using these rules, you should (almost) never need to manually `new`/`delete` memory

C++ smart pointers

Types of smart pointers

- `std::unique_ptr`: Exactly one owner (compile-time error to try to copy it)
  When that owner lets the pointer go out of scope, the memory is automatically released

C++ smart pointers

Types of smart pointers

- `std::unique_ptr`: Exactly one owner (compile-time error to try to copy it)
  When that owner lets the pointer go out of scope, the memory is automatically released
- `std::shared_ptr`: Reference-counted version (we'll discuss what this means later)
  In other words, it can have multiple owners, and is released when the last one lets go

C++ smart pointers

Types of smart pointers

- `std::unique_ptr`: Exactly one owner (compile-time error to try to copy it)
  When that owner lets the pointer go out of scope, the memory is automatically released
- `std::shared_ptr`: Reference-counted version (we'll discuss what this means later)
  In other words, it can have multiple owners, and is released when the last one lets go
- `std::weak_ptr`: Stores a "temporary" reference without owning it
  Can be used to break cycles if two objects refer to each other (e.g., doubly linked lists)
  Useful for caches, which store references to objects that *should* be deleted if the cache is the only thing still refering to it

C++ smart pointers

Old way:

```
bool func() {
  size_t bufsize = 1024 * 1024 * 128;
  int* data = new int[bufsize];

  for (int i = 0; i < bufsize; i++) {
    data[i] = get_data();
    if (!data[i]) {
      delete[] data;
      return false;
    }
  }

  // do something fancy with data
  delete[] data;
  return true;
}
```

C++ smart pointers

Old way:

```cpp
bool func() {
  size_t bufsize = 1024 * 1024 * 128;
  int* data = new int[bufsize];

  for (int i = 0; i < bufsize; i++) {
    data[i] = get_data();
    if (!data[i]) {
      delete[] data;
      return false;
    }
  }

  // do something fancy with data
  delete[] data;
  return true;
}
```

Using `std::unique_ptr`:

```cpp
bool func() {
  size_t bufsize = 1024 * 1024 * 128;
  auto data = std::make_unique<int[]>(
        bufsize);

  for (int i = 0; i < bufsize; i++) {
    data[i] = get_data();
    if (!data[i])
      return false;
  }

  // do something fancy with data
  return true;
}
```

C++ smart pointers

Old way:

```
bool func() {
  size_t bufsize = 1024 * 1024 * 128;
  int* data = new int[bufsize];

  for (int i = 0; i < bufsize; i++) {
    data[i] = get_data();
    if (!data[i]) {
      delete[] data;
      return false;
    }
  }

  // do something fancy with data
  delete[] data;
  return true;
}
```

Using `std::unique_ptr`:

```
bool func() {
  size_t bufsize = 1024 * 1024 * 128;
  auto data = std::make_unique<int[]>(
        bufsize);

  for (int i = 0; i < bufsize; i++) {
    data[i] = get_data();
    if (!data[i])
      return false;
  }

  // do something fancy with data
  return true;
}
```

No calls to `new[]` or `delete[]` necessary

C++ will automatically allocate the memory we want when we call `std::make_unique`, and release it when the unique pointer goes out of scope when the function exits

C++ smart pointers

...but not everyone uses C++

What about when you *need* the heap?

- You can't always rely on the stack because it can overflow quickly
- How do you easily manage large objects?

What about when you *need* the heap?

- You can't always rely on the stack because it can overflow quickly
- How do you easily manage large objects?
- Garbage collection to the rescue! - Automagic Memory Management

GC: Reference counting

- Each object has count of how many objects point to it
- Need to track count whenever object pointers are set or removed/freed

Manual Reference Counting


```
MyClass *obj1 = [[MyClass alloc] init];

MyClass *obj2 = [obj1 retain];

[obj2 release];

[obj1 release];
```

Demo

Python!

GC: Reference Counting Problems

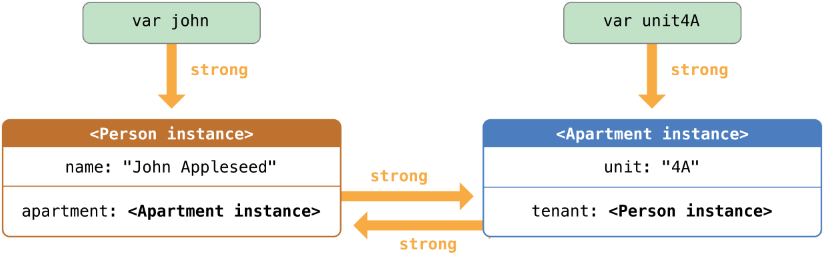
- Overhead of maintaining count

GC: Reference Counting Problems

- Overhead of maintaining count
- Lumpy deletions: tend to have clusters of related objects.

GC: Reference Counting Problems

- Overhead of maintaining count
- Lumpy deletions: tend to have clusters of related objects.
- Cycles



GC: Tracing

- Most common form of garbage collection

GC: Tracing

- Most common form of garbage collection
- "Trace" connections between references and allocations

GC: Tracing

- Most common form of garbage collection
- "Trace" connections between references and allocations

GC: Mark-and-Sweep

- Two Step Process

GC: Mark-and-Sweep

- Two Step Process
- Mark - Find and label all accessible objects

GC: Mark-and-Sweep

- Two Step Process
- Mark - Find and label all accessible objects
  - Follow all pointers from the "root set".
  - Perform DFS on pointers
  - Flag field set to marked

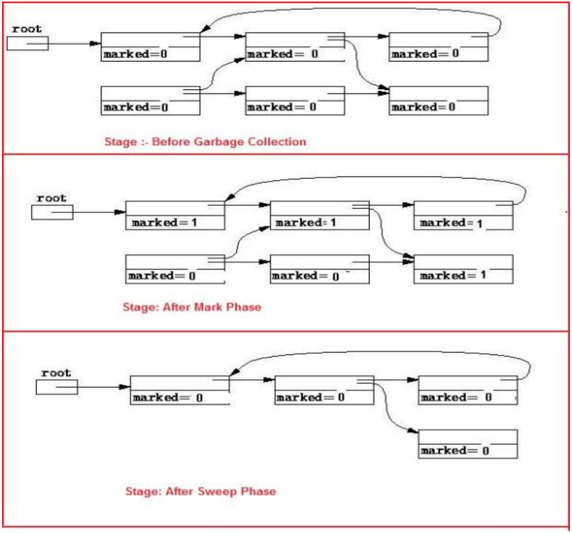
GC: Mark-and-Sweep

- Sweep - Remove all inaccessible objects

GC: Mark-and-Sweep

- Sweep - Remove all inaccessible objects
  - Iterate through memory
  - If marked, unmark
  - Else, free

GC: Mark-and-Sweep



GC: Mark-and-Sweep

- Pros

GC: Mark-and-Sweep

- Pros
  - Works with cyclic data structures
  - Effective and easy to implement

GC: Mark-and-Sweep

- Pros
  - Works with cyclic data structures
  - Effective and easy to implement
- Cons

GC: Mark-and-Sweep

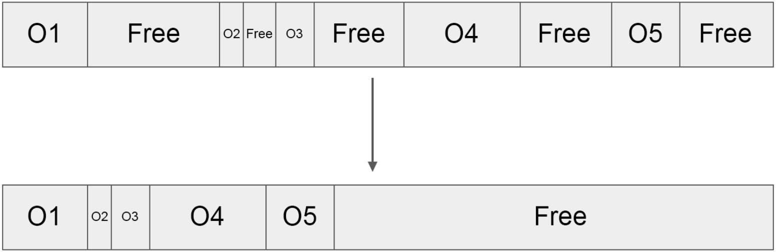
- Pros
  - Works with cyclic data structures
  - Effective and easy to implement
- Cons
  - Program must halt
  - Fragmentation

GC: Mark-and-Sweep with Compaction

- Alleviates fragmentation
- Sweep moves memory to the left

GC: Mark-and-Sweep with Compaction

- Alleviates fragmentation
- Sweep moves memory to the left



GC: Tri-Color Marking

- Incremental garbage collector

GC: Tri-Color Marking

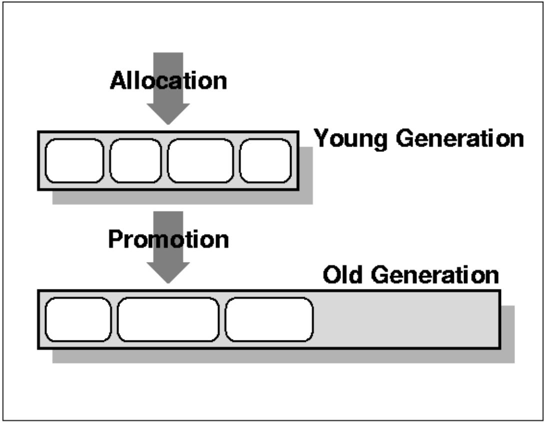
- Incremental garbage collector
- Gray Set - Need to be scanned blocks
- Black Set - Accessible blocks
- White Set - Inaccessible blocks

GC: Tri-Color Marking

- Incremental garbage collector
- Gray Set - Need to be scanned blocks
- Black Set - Accessible blocks
- White Set - Inaccessible blocks
- All blocks accessible from the root start in the gray set.
- All others start in the white set.
- Gray moves to black, all white set objects referenced moved to gray.

Generational GC

- Do we really need to scan everything all the time?
- Split the objects up into generations
- Scan each generation with a different frequency



Generational GC - Eden

- Compaction not important, needs fast GC
- When full, run mark phase and move objects up

Generational GC - Survivor Space 1

- Count for objects being copied back and forth
- When full, run mark phase and move objects up

Generational GC - Survivor Space 2

- Compaction not important
- Less frequent GC

Generational GC - Old Generation

- Mark/sweep with compaction since objects live longer
- Run less frequently, since slow process

Generational GC - Problems

- Objects in old generation could be pointing to objects in survivor space
- Mark phase in survivor space does not go through objects in old generation
- Maintain separate table for maintaining such pointers across generations
- Turns out such occurrences are less frequent in code bases
- Objects in old generation don't generally change much, so may not be too expensive in practice
- Add references from this table to root set of copying GC running in survivor space
- What about younger$\rightarrow$older generation references?

Garbage collection in C/C++?

- Doesn't come with one for performance reasons!
- Usually a bolted-on third party library
- Link the library and allow it to reclaim unused memory automatically
- `malloc` can be replaced with garbage collector's allocator, and `free` is a no-op

Garbage collection in C/C++?

```c
#include <assert.h>
#include <stdio.h>
#include <gc.h>

int main(void)
{
    int i;
    const size = 10000000;

    GC_INIT();
    for (i = 0; i < size; ++i)
    {
        int **p = GC_MALLOC(sizeof *p);
        int *q = GC_MALLOC_ATOMIC(sizeof *q);

        assert(*p == 0);
        *p = GC_REALLOC(q, 2 * sizeof *p);
        if (i == size-1)
            printf("Heap size = %zu\n", GC_get_heap_size());
    }

    return 0;
}
```

GC doesn't mean you forget about memory

- Explicitly let go of object by setting the reference to NULL. This is useful with global variables. Otherwise you hang onto system resources longer.
- Need to manage other system resources (like files); they are closed before GC by programmer

Conclusion

Thank you! Questions?