Understanding Memory Management

Dipesh Kafle

Prerequisites

Memory Layout

Memory Layout

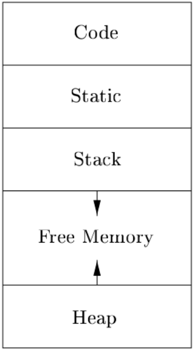


Figure 1: A rough classification of program's memory segments

Memory Layout

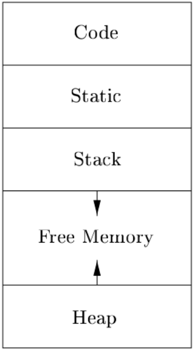


Figure 1: A rough classification of program's memory segments

- In practice, the stack grows towards lower addresses, the heap towards higher(the diagram has it the other way around, but that doesn't matter).

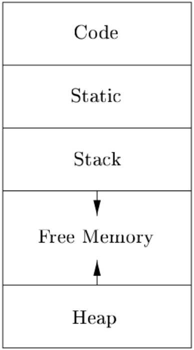


Figure 1: A rough classification of program's memory segments

- In practice, the stack grows towards lower addresses, the heap towards higher(the diagram has it the other way around, but that doesn't matter).
- What are all these things ??

Code and Static segments

- Code: Generated target code has a fixed size, allowing it to be stored in a statically determined area called Code, usually at the low end of memory.

Code and Static segments

- Code: Generated target code has a fixed size, allowing it to be stored in a statically determined area called Code, usually at the low end of memory.
- Static: Statically determined data objects, such as global constants and data generated by the compiler at compile time, can be stored in another area called Static.

Code and Static segments

- **Code**: Generated target code has a fixed size, allowing it to be stored in a statically determined area called Code, usually at the low end of memory.
- **Static**: Statically determined data objects, such as global constants and data generated by the compiler at compile time, can be stored in another area called Static.

```
const char* s = "Lorem Ipsum something something";
int main(){
  const char* string_arr[] = {"Made", "with", "love",
    "by", "Delta", "Force"};
  return 0;
}
```

All the strings used in the above code segment are stored in static section, while the instructions generated for the program will be in code section.

Stack and Stack Allocation

- The stack will store things such as local variables, return address from a function call, etc.

Stack and Stack Allocation(continued)

```c
int main(){
// All of this is
// doing stack
   allocation
  int a = 10;
  int b = 20;
  int arr[2] = {1,2};
  return 0;
}
```

Stack and Stack Allocation(continued)

```c
int main(){
// All of this is
// doing stack
    allocation
  int a = 10;
  int b = 20;
  int arr[2] = {1,2};
  return 0;
}
```

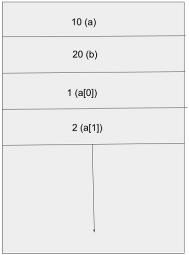


Figure 2: Stack Layout for above code

Heap and Heap Allocation

- Many programming languages allow the programmer to allocate and deallocate data under program control.

Heap and Heap Allocation

- Many programming languages allow the programmer to allocate and deallocate data under program control.
- The heap is used to manage long-lived data.

Heap and Heap Allocation

- Many programming languages allow the programmer to allocate and deallocate data under program control.
- The heap is used to manage long-lived data.
- Unavoidable when we want to allocate memory whose size is known only when the program is running(dynamic allocation).

Heap and Heap Allocation

- Many programming languages allow the programmer to allocate and deallocate data under program control.
- The heap is used to manage long-lived data.
- Unavoidable when we want to allocate memory whose size is known only when the program is running(dynamic allocation).
- C/C++ has `malloc/realloc/free` functions for doing heap memory management.

Heap and Heap Allocation

- Many programming languages allow the programmer to allocate and deallocate data under program control.
- The heap is used to manage long-lived data.
- Unavoidable when we want to allocate memory whose size is known only when the program is running(dynamic allocation).
- C/C++ has `malloc/realloc/free` functions for doing heap memory management.

```c
int* f(int n){
  return malloc(n*sizeof(int));
}
int main(){
    int n ;
    scanf("%d", &n);
    int *arr = f(n); // arr is heap allocated,returned from call to f
    free(arr); //Since, we're good programmers, we'll free the memory as
      ↪  well.
}
```

What exactly are malloc/realloc/free?

What exactly are malloc/realloc/free?

- `malloc(x)` : allocate x bytes in heap

What exactly are malloc/realloc/free?

- `malloc(x)` : allocate x bytes in heap
- `realloc(p, x)` : resize previously allocated heap memory

What exactly are malloc/realloc/free?

- `malloc(x)` : allocate x bytes in heap
- `realloc(p, x)` : resize previously allocated heap memory
- `free(p)` : return heap memory to the operating system

Introduction to Memory Management

- Memory management is all about using `heap memory` correctly.

- Memory management is all about using `heap memory` correctly.
- Stack allocations don't need to be freed; they're automatically managed with scopes (we'll talk about scopes in the next slide).

- Memory management is all about using `heap memory` correctly.
- Stack allocations don't need to be freed; they're automatically managed with **scopes** (we'll talk about scopes in the next slide).
- If it's done incorrectly, the program can crash or slow down.

- Memory management is all about using `heap memory` correctly.
- Stack allocations don't need to be freed; they're automatically managed with **scopes** (we'll talk about scopes in the next slide).
- If it's done incorrectly, the program can crash or slow down.
- There are different techniques for managing heap memory.

What is this scope thing??

What is this scope thing??

Easier to understand with an example

What is this scope thing??

Easier to understand with an example

```
// NOTE: This function won't compile
int f(){ // scope '1 starts
    int a = 10;
    { // scope '2 starts
       int b = 20;
       { // scope '3 stars
           int c = 50;
       } // scope '3 ends
    } // scope '2 ends

    if(a == 10){ // scope '4 starts
      int c = 30;
    } // scope '4 ends
    else { // scope '5 starts
       int c = 90;
    } // scope '5 ends
    return b; // This fails because it's not in scope
} // scope '1 ends
```

What is this scope thing?? (continued)

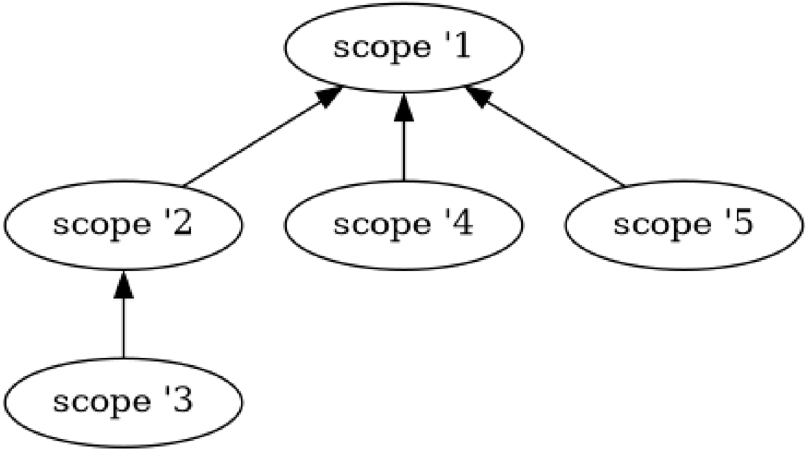


Figure 3: Pictorial representation of scope heirarchy

Why should I care about freeing memory? Is it really a problem?

Why should I care about freeing memory? Is it really a problem?

- If your system has infinite memory, you don't need to worry. However, since memory is finite, you must take care.

Why should I care about freeing memory? Is it really a problem?

- If your system has infinite memory, you don't need to worry. However, since memory is finite, you must take care.
- If one program uses up all the memory, other programs that require memory won't be able to function properly.

Why should I care about freeing memory? Is it really a problem?

- If your system has infinite memory, you don't need to worry. However, since memory is finite, you must take care.
- If one program uses up all the memory, other programs that require memory won't be able to function properly.
- Your program may crash if it requests more memory than the operating system can provide.

Why should I care about freeing memory? Is it really a problem?

- If your system has infinite memory, you don't need to worry. However, since memory is finite, you must take care.
- If one program uses up all the memory, other programs that require memory won't be able to function properly.
- Your program may crash if it requests more memory than the operating system can provide.
- Memory leaks can have a significant impact on long-running programs such as `web servers, editors, and IDEs`.

Why should I care about freeing memory? Is it really a problem?

- If your system has infinite memory, you don't need to worry. However, since memory is finite, you must take care.
- If one program uses up all the memory, other programs that require memory won't be able to function properly.
- Your program may crash if it requests more memory than the operating system can provide.
- Memory leaks can have a significant impact on long-running programs such as `web servers, editors, and IDEs`.

Important terminology

- **Memory Leak**: It happens when you ask the operating system for memory but don't return it back.

Ways to manage memory

We have two ways to do memory management.

We have two ways to do memory management.

- Manual Memory Management : Languages such as C, C++, Rust, etc have this

We have two ways to do memory management.

- Manual Memory Management : Languages such as C, C++, Rust, etc have this
- Automatic Memory Management: Languages such as Python, Java, Go, JavaScript, Swift, etc have this.

Manual Memory Management

Scenarios where you can go wrong

```
// NOTE: this is a dumb example to show where things can go wrong,
// I don't actually write code like this
int* allocate_and_throw_exn(int n){
   int *arr = malloc(n*sizeof(int));
   if (n < 10){
       throw runtime_error("n < 10");
   }
   return arr;
}
int main(){
    try {
        auto *arr = allocate_and_throw_exn(2);
        free(arr);
    } catch(const std::runtime_error &e){
        cout << "Error:" <<e.what() << endl;
    }
}
```

Scenarios where you can go wrong

```cpp
// NOTE: this is a dumb example to show where things can go wrong,
// I don't actually write code like this
int* allocate_and_throw_exn(int n){
   int *arr = malloc(n*sizeof(int));
   if (n < 10){
       throw runtime_error("n < 10");
   }
   return arr;
}
int main(){
    try {
        auto *arr = allocate_and_throw_exn(2);
        free(arr);
    } catch(const std::runtime_error &e){
        cout << "Error:" <<e.what() << endl;
    }
}
```

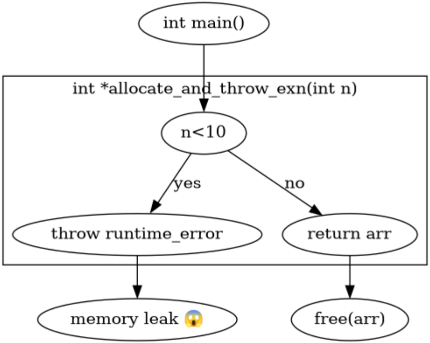


Figure 4: Flow for the leaking code

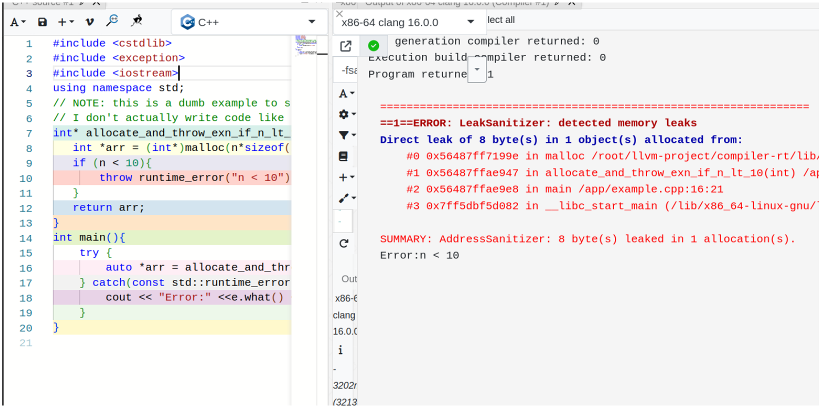


Figure 5: Memory Leak Detected by address sanitizer

How to fix it??

How to fix it??

- DON'T WRITE DUMB CODE LIKE I DID

How to fix it??

- DON'T WRITE DUMB CODE LIKE I DID
- More high level language(than C) like C++, Rust provide us with smart ways to manage memory

How to fix it??

- DON'T WRITE DUMB CODE LIKE I DID
- More high level language(than C) like C++, Rust provide us with smart ways to manage memory
- They come built-in with smart pointer types like `unique_ptr` (C++)

Fixing the code with smart pointer

```
# LEAKING CODE
int* allocate_and_throw_exn(int n){
   int *arr = malloc(n*sizeof(int));
   if (n < 10){
       throw runtime_error("n < 10");
   }
   return arr;
}
int main(){
    try {
        auto *arr =
         ↪  allocate_and_throw_exn(2);
        free(arr);
    } catch(const std::runtime_error
     ↪  &e){
        cout << "Error:" <<e.what() <<
         ↪  endl;
    }
}
```

Fixing the code with smart pointer

```
# LEAKING CODE
int* allocate_and_throw_exn(int n){
    int *arr = malloc(n*sizeof(int));
    if (n < 10){
        throw runtime_error("n < 10");
    }
    return arr;
}
int main(){
     try {
         auto *arr = allocate_and_throw_exn(2);
         free(arr);
     } catch(const std::runtime_error &e){
         cout << "Error:" <<e.what() << endl;
     }
}
```

```
# FIXED CODE
std::unique_ptr<int[]> allocate_and_throw_exn(int n){
    auto arr = std::unique_ptr<int[]>(new int[n]);
    if (n < 10){
        throw runtime_error("n < 10");
    }
    return arr;
}
int main(){
     try {
         auto arr = allocate_and_throw_exn(2);
     } catch(const std::runtime_error &e){
         cout << "Error:" <<e.what() << endl;
     }
}
```

Fixing the code with smart pointer

```
# LEAKING CODE
int* allocate_and_throw_exn(int n){
    int *arr = malloc(n*sizeof(int));
    if (n < 10){
        throw runtime_error("n < 10");
    }
    return arr;
}
int main(){
    try {
        auto *arr = allocate_and_throw_exn(2);
        free(arr);
    } catch(const std::runtime_error &e){
        cout << "Error:" <<e.what() << endl;
    }
}
```

```
# FIXED CODE
std::unique_ptr<int[]> allocate_and_throw_exn(int n){
    auto arr = std::unique_ptr<int[]>(new int[n]);
    if (n < 10){
        throw runtime_error("n < 10");
    }
    return arr;
}
int main(){
    try {
        auto arr = allocate_and_throw_exn(2);
    } catch(const std::runtime_error &e){
        cout << "Error:" <<e.what() << endl;
    }
}
```

What the hell just happened??

How do smart pointers work?

How do smart pointers work?

- Uses **scope** to track lifetime of a pointer(scopes mentioned in previous section)

How do smart pointers work?

- Uses **scope** to track lifetime of a pointer(scopes mentioned in previous section)
- C++ uses `destructors` to run code when an object goes out of scope. (due to RAII in C++)

What is RAII in C++?

RAII can be summarized as follows:

- encapsulate each resource into a class, where
  - the constructor acquires the resource and establishes all class invariants or throws an exception if that cannot be done,
  - the destructor releases the resource and never throws exceptions;

Figure 6: RAII

Destructors In Action

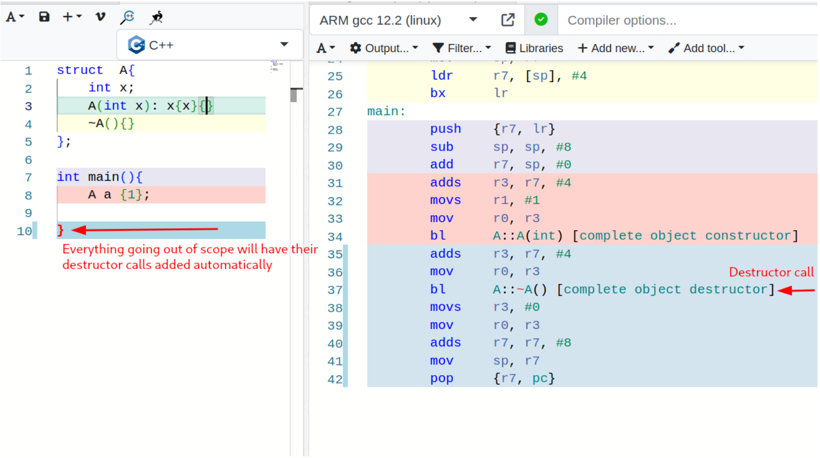


Figure 7: Destructor call added automatically

Let’s make our own unique_ptr

Let’s make our own unique_ptr

```cpp
#include <iostream>
using namespace std;
class int_ptr{
    int* x ;
public:
    int_ptr(int x): x{new int(x)}{}
    ~int_ptr(){
        delete x;
    }
    int& operator*(){
        return *x;
    }
};
int main(){
    int_ptr one(1);
    cout << *one << endl;
}
```

Figure 8: int_ptr working without any leaks

Is unique_ptr the only smart pointer??

Is unique_ptr the only smart pointer??

NO

Isn’t unique_ptr perfect already? Why would I need anything else?

Isn’t unique_ptr perfect already? Why would I need anything else?

- Problem with `unique_ptr` is that it can have only one owner.

Isn't unique_ptr perfect already? Why would I need anything else?

- Problem with `unique_ptr` is that it can have only one owner.

Case in Point

- A `unix file descriptor`

Isn’t unique_ptr perfect already? Why would I need anything else?

- Problem with `unique_ptr` is that it can have only one owner.

Case in Point

- A `unix file descriptor`
- A file descriptor can have multiple owners. It should only be freed when all the owners go out of scope.

File Descriptors Example and what's the problem

File Descriptors Example and what's the problem

DISCLAIMER: The code that I'm about to show is cursed and no one should model a file descriptor like this

File Descriptors Example and what's the problem

DISCLAIMER: The code that I'm about to show is cursed and no one should model a file descriptor like this

But, we'll do it still(for the purpose of this talk)

File Descriptors (continued)

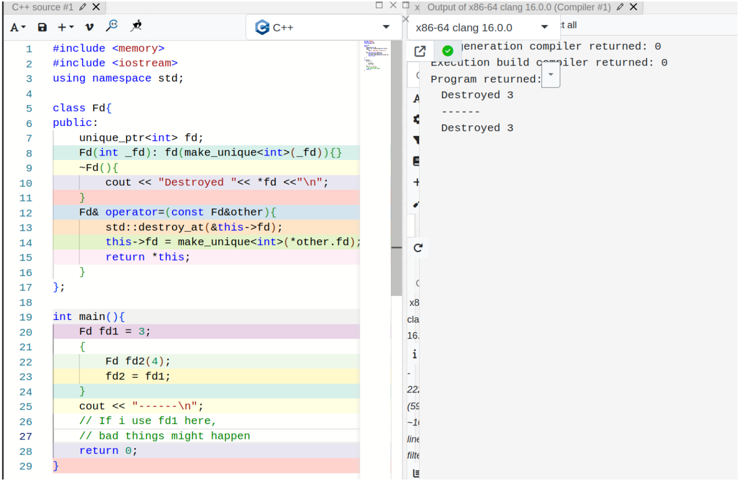


Figure 9: File Descriptor with unique_ptr

File Descriptors (continued)

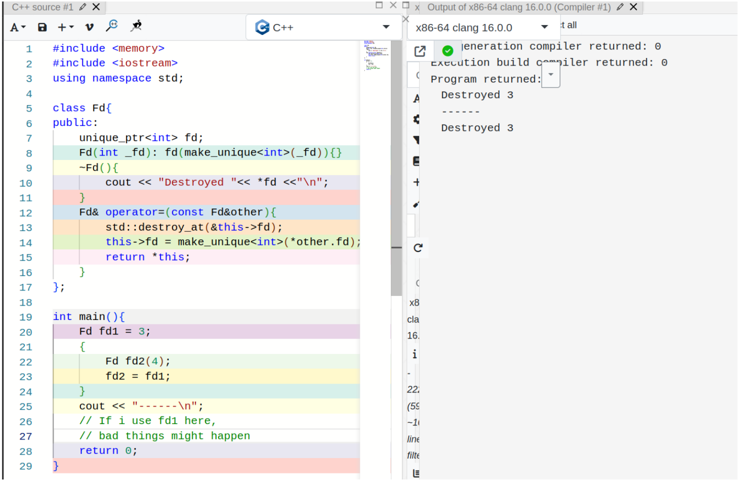


Figure 9: File Descriptor with unique_ptr

Not possible to model this correctly

Another classic example (why only unique_ptr isn't enough?)

- SPMC(Single Producer Multiple Consumer) and MPMC(Multiple Producer Multiple Consumer) problem

Another classic example (why only unique_ptr isn't enough?)

- SPMC(Single Producer Multiple Consumer) and MPMC(Multiple Producer Multiple Consumer) problem

Figure 10: MPMC queue

Another classic example (why only unique_ptr isn't enough?)

- SPMC(Single Producer Multiple Consumer) and MPMC(Multiple Producer Multiple Consumer) problem

Figure 10: MPMC queue

- Requires multiple owners for producer end as well as the consumer end

Introduction to Automatic Memory Management

- The programming language you're using is responsible for memory management. The language runtime handles everything.

Introduction to Automatic Memory Management

- The programming language you're using is responsible for memory management. The language runtime handles everything.
- Programmer doesn't have the mental burden of dealing with memory.

Introduction to Automatic Memory Management

- The programming language you're using is responsible for memory management. The language runtime handles everything.
- Programmer doesn't have the mental burden of dealing with memory.
- All the allocation/deallocation calls are hidden from the programmer.

Introduction to Automatic Memory Management

- The programming language you're using is responsible for memory management. The language runtime handles everything.
- Programmer doesn't have the mental burden of dealing with memory.
- All the allocation/deallocation calls are hidden from the programmer.

IMPORTANT TERMINOLOGY

- The programming language you're using is responsible for memory management. The language runtime handles everything.
- Programmer doesn't have the mental burden of dealing with memory.
- All the allocation/deallocation calls are hidden from the programmer.

IMPORTANT TERMINOLOGY

Data that cannot be referenced is generally known as **garbage**

Introduction to Automatic Memory Management

- The programming language you're using is responsible for memory management. The language runtime handles everything.
- Programmer doesn't have the mental burden of dealing with memory.
- All the allocation/deallocation calls are hidden from the programmer.

IMPORTANT TERMINOLOGY

Data that cannot be referenced is generally known as **garbage**

Two techniques used primarily

- The programming language you're using is responsible for memory management. The language runtime handles everything.
- Programmer doesn't have the mental burden of dealing with memory.
- All the allocation/deallocation calls are hidden from the programmer.

IMPORTANT TERMINOLOGY

Data that cannot be referenced is generally known as **garbage**

Two techniques used primarily

- We catch the transitions as `reachable`(in-scope) objects turn `unreachable`(out-of-scope).

Introduction to Automatic Memory Management

- The programming language you're using is responsible for memory management. The language runtime handles everything.
- Programmer doesn't have the mental burden of dealing with memory.
- All the allocation/deallocation calls are hidden from the programmer.

IMPORTANT TERMINOLOGY

Data that cannot be referenced is generally known as **garbage**

Two techniques used primarily

- We catch the transitions as `reachable`(in-scope) objects turn `unreachable`(out-of-scope). Example: Reference Counting based memory management in Python, Swift,etc., C++ shared_ptr, Rust's Rc and Arc types

Introduction to Automatic Memory Management

- The programming language you're using is responsible for memory management. The language runtime handles everything.
- Programmer doesn't have the mental burden of dealing with memory.
- All the allocation/deallocation calls are hidden from the programmer.

IMPORTANT TERMINOLOGY

Data that cannot be referenced is generally known as **garbage**

Two techniques used primarily

- We catch the transitions as `reachable`(in-scope) objects turn `unreachable`(out-of-scope). Example: Reference Counting based memory management in Python, Swift,etc., C++ shared_ptr, Rust's Rc and Arc types
- We periodically locate all the `reachable` objects and then infer that all the other objects are `unreachable`.

- The programming language you're using is responsible for memory management. The language runtime handles everything.
- Programmer doesn't have the mental burden of dealing with memory.
- All the allocation/deallocation calls are hidden from the programmer.

IMPORTANT TERMINOLOGY

Data that cannot be referenced is generally known as **garbage**

Two techniques used primarily

- We catch the transitions as `reachable`(in-scope) objects turn `unreachable`(out-of-scope). Example: Reference Counting based memory management in Python, Swift,etc., C++ shared_ptr, Rust's Rc and Arc types
- We periodically locate all the `reachable` objects and then infer that all the other objects are `unreachable`. Example: Trace Based Garbage Collection such as mark and sweep garbage collector used in language like Java, JavaScript, etc.

Reference Counting

- All the transitions from reachable to unreachable are caught immediately.

- All the transitions from reachable to unreachable are caught immediately.

Backed by simple rules

- All the transitions from reachable to unreachable are caught immediately.

Backed by simple rules

- Every allocated pointer has a reference count associated with it

Reference Counting

- All the transitions from reachable to unreachable are caught immediately.

Backed by simple rules

- Every allocated pointer has a reference count associated with it
- Every copy leads to reference count increment.

- All the transitions from reachable to unreachable are caught immediately.

Backed by simple rules

- Every allocated pointer has a reference count associated with it
- Every copy leads to reference count increment.
- Every time a reference of the pointer goes out of scope, reference count is decremented by 1.

- All the transitions from reachable to unreachable are caught immediately.

Backed by simple rules

- Every allocated pointer has a reference count associated with it
- Every copy leads to reference count increment.
- Every time a reference of the pointer goes out of scope, reference count is decremented by 1.
- When the reference count goes to 0, it is safe to `free` the object.

C++'s shared_ptr type

- It is a `reference-counted pointer`, manages the `reference count` elegantly with assignment operator overload, copy constructor overload, destructors, etc.

Let’s solve our file descriptor issue

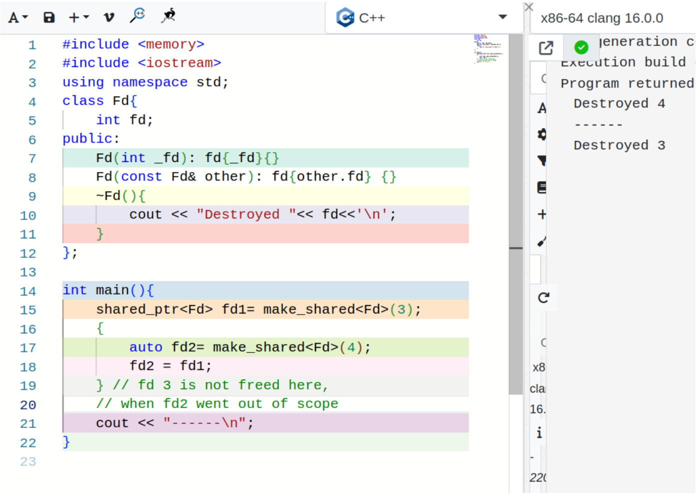


Figure 11: File Descriptor with shared_pointer

Trace Based Garbage Collection

- Periodically locates all the `reachable` objects and then infers that all the other objects are `unreachable`

Trace Based Garbage Collection

- Periodically locates all the `reachable` objects and then infers that all the other objects are `unreachable`
- A pointer is reachable $\Rightarrow$ There's a root somewhere(in stack, register, static, etc.) from where we can transitively access the pointer.

What’s a root?

What’s a root?

Think of them as root nodes of a graph, something that you have direct access to.

What’s a root?

Think of them as root nodes of a graph, something that you have direct access to.

A `GC` will treat all the stack variables, registers, static section, as roots.

What's a root?

Think of them as root nodes of a graph, something that you have direct access to.

A GC will treat all the stack variables, registers, static section, as roots.

```
type baz = {
    x: int;
    y: int
}
type bar = {
    a:  int;
    b:  baz
}
type foo = {
    field1 : bar;
    field2: int
}
let foo_instance: foo = {
    field1 = {
        a  = 10;
        b = { x= 2 ;y= 5; };
    };
    field2= 5;
}
```

What's a root?

Think of them as root nodes of a graph, something that you have direct access to.

A GC will treat all the stack variables, registers, static section, as roots.

```
type baz = {
    x: int;
    y: int
}
type bar = {
    a:  int;
    b:  baz
}
type foo = {
    field1 : bar;
    field2: int
}
let foo_instance: foo = {
    field1 = {
        a  = 10;
        b = { x= 2 ;y= 5; };
    };
    field2= 5;
}
```

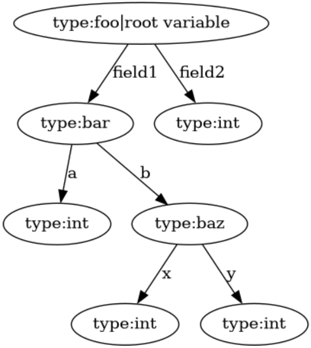


Figure 12: foo_instance variable as a graph

Root finding

- It involves many intricate details, which took me quite a while to understand.

- It involves many intricate details, which took me quite a while to understand.
- You'd probably want to design a `precise-collector` if you're building your own language.

- It involves many intricate details, which took me quite a while to understand.
- You'd probably want to design a `precise-collector` if you're building your own language. Precise Collector is when GC knows about root objects: where they're placed on stack,their size,lifetime etc.

- It involves many intricate details, which took me quite a while to understand.
- You'd probably want to design a `precise-collector` if you're building your own language. Precise Collector is when GC knows about root objects: where they're placed on stack,their size,lifetime etc. Most popular language runtimes use precise garbage collectors: Python, PyPy, JVM(Java), .NET(C#), Lua, V8(JavaScript) ,SpiderMonkey(JavaScript) and a lot of others.

Root finding

- It involves many intricate details, which took me quite a while to understand.
- You'd probably want to design a `precise-collector` if you're building your own language. Precise Collector is when GC knows about root objects: where they're placed on stack,their size,lifetime etc. Most popular language runtimes use precise garbage collectors: Python, PyPy, JVM(Java), .NET(C#), Lua, V8(JavaScript) ,SpiderMonkey(JavaScript) and a lot of others.
- Conservative collection does not know about types of traced GC objects.

- It involves many intricate details, which took me quite a while to understand.
- You'd probably want to design a `precise-collector` if you're building your own language. Precise Collector is when GC knows about root objects: where they're placed on stack,their size,lifetime etc. Most popular language runtimes use precise garbage collectors: Python, PyPy, JVM(Java), .NET(C#), Lua, V8(JavaScript) ,SpiderMonkey(JavaScript) and a lot of others.
- Conservative collection does not know about types of traced GC objects. It'll assume everything that's present in the stack, static and register are pointers and it'll follow it.

- It involves many intricate details, which took me quite a while to understand.
- You'd probably want to design a `precise-collector` if you're building your own language. Precise Collector is when GC knows about root objects: where they're placed on stack,their size,lifetime etc. Most popular language runtimes use precise garbage collectors: Python, PyPy, JVM(Java), .NET(C#), Lua, V8(JavaScript) ,SpiderMonkey(JavaScript) and a lot of others.
- Conservative collection does not know about types of traced GC objects. It'll assume everything that's present in the stack, static and register are pointers and it'll follow it. They are inherently unsafe.

- It involves many intricate details, which took me quite a while to understand.
- You'd probably want to design a `precise-collector` if you're building your own language. Precise Collector is when GC knows about root objects: where they're placed on stack,their size,lifetime etc. Most popular language runtimes use precise garbage collectors: Python, PyPy, JVM(Java), .NET(C#), Lua, V8(JavaScript) ,SpiderMonkey(JavaScript) and a lot of others.
- Conservative collection does not know about types of traced GC objects. It'll assume everything that's present in the stack, static and register are pointers and it'll follow it. They are inherently unsafe. Example: Boehm-Demers-Weiser conservative C/C++ Garbage Collector

All trace-based algorithms compute the set of reachable objects and then take the complement of this set. Memory is therefore recycled as follows:

All trace-based algorithms compute the set of reachable objects and then take the complement of this set. Memory is therefore recycled as follows:

1. The program or mutator runs and makes allocation requests.

Tracing GC Algorithms

All trace-based algorithms compute the set of reachable objects and then take the complement of this set. Memory is therefore recycled as follows:

1. The program or mutator runs and makes allocation requests.
2. The garbage collector discovers reachability by tracing.

All trace-based algorithms compute the set of reachable objects and then take the complement of this set. Memory is therefore recycled as follows:

1. The program or mutator runs and makes allocation requests.
2. The garbage collector discovers reachability by tracing.
3. The garbage collector reclaims the storage for unreachable objects.

All trace-based algorithms compute the set of reachable objects and then take the complement of this set. Memory is therefore recycled as follows:

1. The program or mutator runs and makes allocation requests.
2. The garbage collector discovers reachability by tracing.
3. The garbage collector reclaims the storage for unreachable objects.

Tracing Based GC Algorithms

All trace-based algorithms compute the set of reachable objects and then take the complement of this set. Memory is therefore recycled as follows:

1. The program or mutator runs and makes allocation requests.
2. The garbage collector discovers reachability by tracing.
3. The garbage collector reclaims the storage for unreachable objects.

Tracing Based GC Algorithms

- Copying Collector
- Mark and Sweep Collector

Copying Collector

- Copying Collector partitions memory into two semispaces, A and B.

Copying Collector

- Copying Collector partitions memory into two semispaces, A and B.
- All the allocations are performed in one semispace until it fills up.

- Copying Collector partitions memory into two semispaces, A and B.
- All the allocations are performed in one semispace until it fills up.
- The garbage collector then copies reachable objects to the other space.

Copying Collector

- Copying Collector partitions memory into two semispaces, A and B.
- All the allocations are performed in one semispace until it fills up.
- The garbage collector then copies reachable objects to the other space.
- Roles of the semispaces are reversed when garbage collection is complete.

Copying Collector

- Copying Collector partitions memory into two semispaces, A and B.
- All the allocations are performed in one semispace until it fills up.
- The garbage collector then copies reachable objects to the other space.
- Roles of the semispaces are reversed when garbage collection is complete.
- While all this is happening, the whole program is stopped. (This is why it's called a stop-the-world algorithm)

Copying Collector (A visual representation)

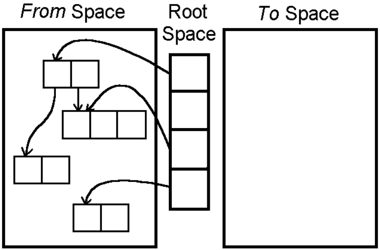


Figure 13: Before GC

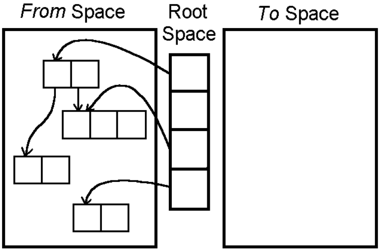


Figure 13: Before GC

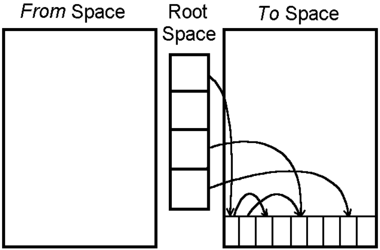


Figure 14: After GC

Some terminology first

Some terminology first

In a `mark-and-sweep` gc, An object may be in one of the 4 states: Free(Blue), Unreached(White), Unscanned(Gray), Scanned(Black)

Some terminology first

In a `mark-and-sweep` gc, An object may be in one of the 4 states: Free(Blue), Unreached(White), Unscanned(Gray), Scanned(Black)

It is also a stop-the-world algorithm.

Some terminology first

In a `mark-and-sweep` gc, An object may be in one of the 4 states: Free(Blue), Unreached(White), Unscanned(Gray), Scanned(Black)

It is also a stop-the-world algorithm.

Mark →responsible for marking reachable objects `Black`

Some terminology first

In a `mark-and-sweep` gc, An object may be in one of the 4 states: Free(Blue), Unreached(White), Unscanned(Gray), Scanned(Black)

It is also a stop-the-world algorithm.

Mark →responsible for marking reachable objects `Black`

Sweep →responsible for freeing all the unreachable(`White`) objects.

```
     /* marking phase */
1)   add each object referenced by the root set to list Unscanned
             and set its reached-bit to 1;
2)   while (Unscanned ≠ ∅) {
3)           remove some object o from Unscanned;
4)           for (each object o′ referenced in o) {
5)                   if (o′ is unreached; i.e., its reached-bit is 0) {
6)                           set the reached-bit of o′ to 1;
7)                           put o′ in Unscanned;
                     }
             }
     }
     /* sweeping phase */
8)   Free = ∅;
9)   for (each chunk of memory o in the heap) {
10)          if (o is unreached, i.e., its reached-bit is 0) add o to Free;
11)          else set the reached-bit of o to 0;
     }
```

Figure 15: Basic Mark and Sweep

Which is better ? (Automatic Memory Management or Manual Management)

Which is better ? (Automatic Memory Management or Manual Management)

Which is better ? (Automatic Memory Management or Manual Management)

DEPENDS

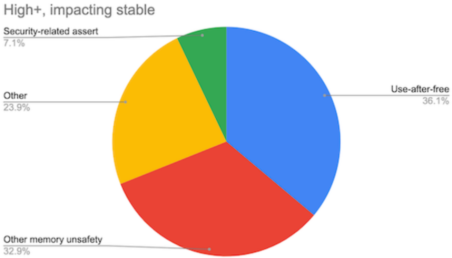
Figure 16: Segmentation Faults

Memory safety

The Chromium project finds that around 70% of our serious security bugs are memory safety problems. Our next major project is to prevent such bugs at source.

The problem

Around 70% of our high severity security bugs are memory unsafety problems (that is, mistakes with C/C++ pointers). Half of those are use-after-free bugs.



(Analysis based on 912 high or critical severity security bugs since 2015, affecting the Stable channel.)

These bugs are spread evenly across our codebase, and a high proportion of our non-security stability bugs share the same types of root cause. As well as risking our users' security, these bugs have real costs in how we fix and ship Chrome.

Figure 17: Use After Free

We frequently performed an operation we called the "gossip dance", where we'd take a node out of rotation to let it compact without taking traffic, bring it back in to pick up hints from Cassandra's hinted handoff, and then repeat until the compaction backlog was empty. We also spent a large amount of time tuning the JVM's garbage collector and heap settings, because GC pauses would cause significant latency spikes.

Figure 18: Discord Troubles(from a recent Discord Engineering Blog)

Case against Automatic Memory Management

> We frequently performed an operation we called the "gossip dance", where we'd take a node out of rotation to let it compact without taking traffic, bring it back in to pick up hints from Cassandra's hinted handoff, and then repeat until the compaction backlog was empty. We also spent a large amount of time tuning the JVM's garbage collector and heap settings, because GC pauses would cause significant latency spikes.

Figure 18: Discord Troubles(from a recent Discord Engineering Blog)

- Similar GC issues encountered by many tech giants

Advanced Topics in Memory Management

Advanced Topics in Memory Management

- Advanced type system like in Rust, which help with memory safety

- Advanced type system like in Rust, which help with memory safety
- Incremental GC

- Advanced type system like in Rust, which help with memory safety
- Incremental GC
- Parallel and Concurrent GC

Advanced Topics in Memory Management

- Advanced type system like in Rust, which help with memory safety
- Incremental GC
- Parallel and Concurrent GC
- Generational GC

- Advanced type system like in Rust, which help with memory safety
- Incremental GC
- Parallel and Concurrent GC
- Generational GC
- Reducing GC pauses

References

References

- https://rebelsky.cs.grinnell.edu/Courses/CS302/99S/Presentations/GC/
- https://discord.com/blog/how-discord-stores-trillions-of-messages
- https://gchandbook.org/
- Compilers: Principles, Techniques, and Tools by Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman
- cppreference