Understanding Memory Management

Dipesh Kafle

%

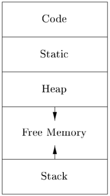

Figure 1: A program’s memory segments roughly classified

- 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).

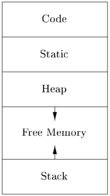

Figure 1: A program's memory segments roughly classified

- 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 ??

Memory Layout

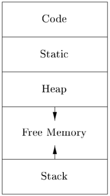

Figure 1: A program's memory segments roughly classified

- 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 ??
- We are mainly concerned withe the stack and the heap for the purpose of this talk, but we'll see what the other things are as well.

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.

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

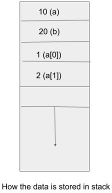

Figure 2: Stack Layout for above code

- 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.
- 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.
- C/C++ has `malloc/realloc/free` functions for doing heap memory management.
- Unavoidable when we want to allocate memory whose size is known only when the program is running(dynamic allocation).

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

```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?

Heap allocation functions:

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

What exactly are malloc/realloc/free?

Heap allocation functions:

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

What exactly are malloc/realloc/free?

Heap allocation functions:

- `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.
- If it's done incorrectly, the program can crash or slow down.

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

Introduction to Memory Management

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

Important terminology

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

What is this scope thing??

```
// NOTE: This function won't compile
int f(){ // scope '1 starts
    int a = 10;
    { // scope '2 starts
       int b = 20;
    } // scope '2 ends
    if(a == 10){ // scope '3 starts
      int c = 30;
    } // scope '3 ends
    return b; // This fails because it's not in scope
} // scope '1 ends
```

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`.

Ways to manage memory

Ways to manage memory

We have two ways to do memory management.

Ways to manage memory

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

```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_if_n_lt_10(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_if_n_lt_10(2);
        free(arr);
    } catch(const std::runtime_error &e){
        cout << "Error:" <<e.what() << endl;
    }
}
```

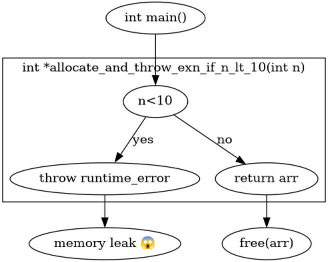

Figure 3: Flow for the leaking code

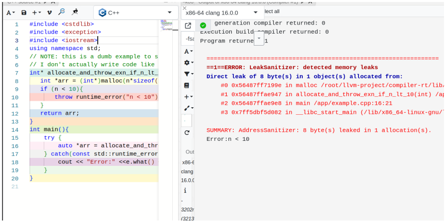

Figure 4: Memory Leak Detected by address sanitizer

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

```
std::unique_ptr<int[]> allocate_and_throw_exn_if_n_lt_10(int n){
    auto *arr = make_unique<int[]>(new int[n]);
    if (n < 10){
        throw runtime_error("n < 10");
    }
    return arr;
}
int main(){
     try {
         auto arr = allocate_and_throw_exn_if_n_lt_10(2);
     } catch(const std::runtime_error &e){
         cout << "Error:" <<e.what() << endl;
     }
}
```

Fixing the code with smart pointer

```cpp
std::unique_ptr<int[]> allocate_and_throw_exn_if_n_lt_10(int n){
    auto *arr = make_unique<int[]>(new int[n]);
    if (n < 10){
        throw runtime_error("n < 10");
    }
    return arr;
}
int main(){
    try {
        auto arr = allocate_and_throw_exn_if_n_lt_10(2);
    } catch(const std::runtime_error &e){
        cout << "Error:" <<e.what() << endl;
    }
}
```

What the hell just happened??

We're not even freeing anything?? How does this 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++)

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 5: RAII

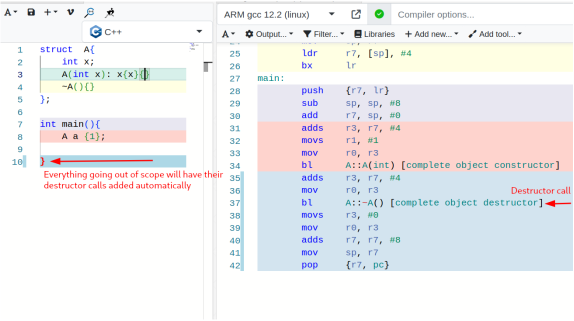

Figure 6: Destructor call added automatically

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;
}
```

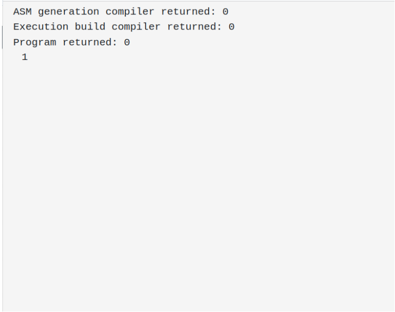

Figure 7: int_ptr working without any leaks

Is unique_ptr the only smart pointer??

NO

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.

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.

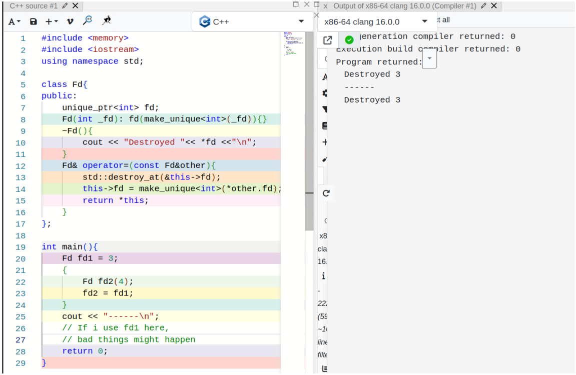

Figure 8: File Descriptor with unique_ptr(the code is very bad?)

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.

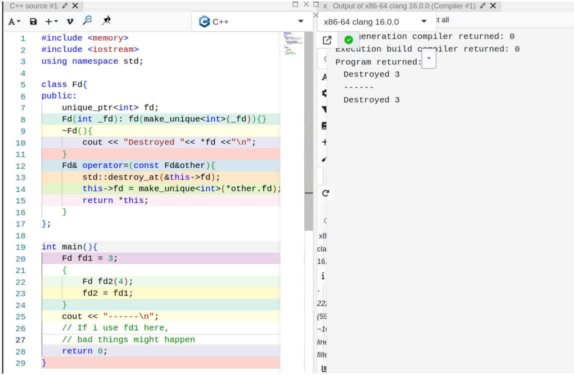

Figure 8: File Descriptor with unique_ptr(the code is very bad?)

Not possible to model this correctly

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.

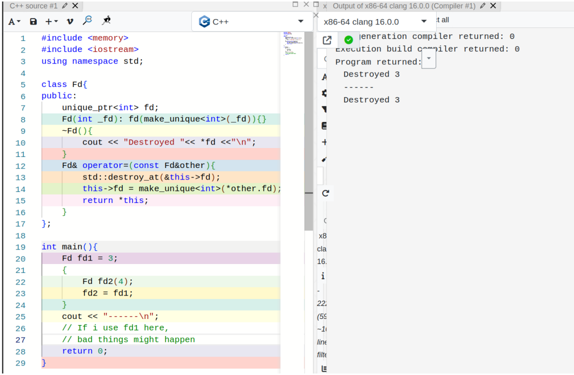

Figure 8: File Descriptor with unique_ptr(the code is very bad?)

Not possible to model this correctly

That’s why we need more

Introduction to Automatic Memory Management

Reference Counting

Trace Based Collection

Which is better?(Automatic Memory Management or Manual Management

Advanced Topics in Garbage Collection

Advanced Topics in Garbage Collection

- Incremental GC
- Parallel and Concurrent GC
- Precise and Conservative Garbage Collectors
- Reducing GC pause*

References

References

- Some presentation on GC, Grinnel college