Memory Management

Principles of Programming Languages

Colorado School of Mines

https://lambda.mines.edu

Review: Pointers and References

Pointers

- A **pointer** is a value that indicates location in memory.
- When we change the location the pointer points to, we say we **assign** the pointer a value.
- When we look at the data the pointer points to, we say we **dereference** the pointer.

Pointer Arithmetic

In C or C++:

- `p[0]` is equivalent to `*p`
- `p[n]` is equivalent to `*(p + n)`

Example in C or C++

```
main () {
    double nums[100];
    double *end = nums + 100;
    for (double *p = nums; p < end; p++)
        *p = rand() / RAND_MAX;
}
```

Pointer Arithmetic Oddities

```c
main() {
    char *name = "Jack";
    /* What does this print? */
    printf("%c\n", 2[name]);
}
```

Pointer Arithmetic Oddities

```
main() {
    char *name = "Jack";
    /* What does this print? */
    printf("%c\n", 2[name]);
}
```

Array accessess are written `BASE[OFFSET]`, which is equivalent to `*(BASE + OFFSET)`. Notice that `OFFSET[BASE]` is equivalent to `*(OFFSET + BASE)`, and since addition is commutative, the operations are equivalent.

References

- A **reference** is a special kind of pointer type
- References can be assigned to point to other data
- References can be dereferenced to get the corresponding data
- **Pointer arithmetic is not available on references**

Object Lifetimes

Object Lifetimes

- The **lifetime** of an object is the time between object creation and destruction; the time during which an object is bound to a memory cell
- Common lifetimes:
  - Static
  - Stack Dynamic
  - Explicit Heap
  - Implicit Heap
- Don't confuse these static/dynamic and explicit/implicit words with typing systems, we're referring to lifetimes here.

Static Lifetimes

- Static variables are bound to a memory cell **before execution** (typically, this means that binding time is during link time)

```c
void f() {
    static int x = 0;
    printf("%d\n", x++);
    if (x < 5) f();
}

main() {
    f();
}
```

Static Lifetimes: Advantages & Disadvantages

With your learning group:

1. What are the advantages of a static lifetime?
2. What are the disadvantages? Collectively come up with and write an example function (in any language, or even pseudocode) which could not be completed using just static lifetimes.

Be prepared to share with the class.

Static Lifetimes: Advantages & Disadvantages

Advantages:

- **Direct addressing** $\rightarrow$ efficiency
- Useful for history-sensitive applications: caching, memoization

Disadvantages:

- No recursion
- Bad for reliability

Stack Dynamic

- Lifetime: Created when declaration is encountered, destroyed when stack frame is cleared

```
int f(int x) {
    int z = x + 1;
    /* ... */
    if (z < 5) f(z);
}
```

Note

Variables might be allocated at beginning of method, not necessarily when declaration is encountered.
Languages may also have features which allow scopes which are not defined by a literal stack frame.

Stack Dynamic: Advantages & Disadvantages

With your learning group:

1. What are the advantages of a stack dynamic lifetime? Disadvantages?
2. How does a stack dynamic lifetime allow for recursion?
3. When might static lifetime be more memory efficient than stack dynamic? Vice-versa?
4. Does stack dynamic allow for dynamically sized arrays?

   Hint: `man 3 alloca`

Be prepared to share with the class.

Stack Dynamic: Advantages & Disadvantages

Advantages:

- Recursion possible (each call gets its own stack frame!)
- Conserves storage for short-lifetime variables

Disadvantages:

- Allocating and deallocating must be done at run time
- Functions cannot be history sensitive (which might be a good restriction...)
- Indirect addressing

Explicit Heap Dynamic

- Allocated and deallocated at runtime by **explicit directives**
- Accessed through pointers or references to heap memory (usually stored on stack)

```
main() {
    printf("How many numbers to input? ");
    int amount;
    scanf(" %d", &amount);
    int *nums = malloc(amount * sizeof(int));
    /* do some things ... */
    free(nums);
}
```

Explicit Heap Dynamic: Advantages & Disadvantages

With your learning group:

1. What are the advantages of an **explicit** heap dynamic lifetime? Disadvantages?
2. What might be some of the **dangers** of explicit heap dynamic lifetimes? Write a few examples with your group.

Be prepared to share with the class.

Dangers of Explicit Heap: Memory Leaks

A **memory leak** occurs when all references to a memory allocation have been lost but the memory has not been freed yet.

This becomes especially noticable when the routine that allocates memory is frequently called.

Memory Leak: Example

```c
/* excuse me caller, free this for me! */
char * f(int n) {
    char *A = malloc(n);
    /* ... */
    return A;
}

main() {
    char *result;
    for (int i = 1; i < 10000000; i++) {
        result = f(i);
        /* do some work, but don't free! */
    }
}
```

Dangers of Explicit Heap: Access after free

A programmer may accidentally access memory they already freed! A pointer which points to freed data is called a **dangling pointer**.

```
main() {
    char *A = NULL;
    int amount;
    for (int i = 1; i <= 10; i++) {
        scanf(" %d", &amount);
        if (i == 1 || amount > 0)
            A = malloc(amount + 1);
        A[amount] = '\0';
        /* ... */
        free(A);
    }
}
```

Dangers of Explicit Heap: Double Free

A programmer may accidentally free a pointer twice!

```
main() {
    char *buf = malloc(10);
    printf("What is your favorite word? ");
    scanf("%s\n", buf);
    if (!strcmp("computers", buf)) {
        printf("Me too!\n");
        free(buf);
    }
    free(buf);
}
```

Implicit Dynamic

- In an implicit dynamic lifetime, allocation is caused automatically by instantiation.
- Deallocation can be done a number of ways: reference counting, garbage collection, or simply not allowing references.

```
def f(x):
    return [0] * x

print(f(10))
```

Garbage Collection

Garbage collection provides means to automatically destroy inaccessible objects in an implicit dynamic system with references. The garbage collector is run occasionally.

To run the garbage collector:

1. Iterate over all visible variables (the roots) and find where they point in the heap.
2. For each of the corresponding entries in the heap, mark them as visited and visit all entries they reference.
3. Repeat until you cannot visit any more entries, the unvisited entries can be freed.

Garbage Collection: Example