Object Lifetime and Pointers

CSCI 400

Colorado School of Mines

31 August 2017

Object Lifetime

Why do we care?

Could affect:

- Performance
- Reliability
  - e.g. Ease of debugging
- Language choice

Object Lifetime

- **Lifetime of a variable**
  - Time during which the variable is bound to a particular memory cell
- Ruby built-in objects created when value assigned
  - e.g. `x = 5`
  - Other classes create with `new`
- Factory methods also create objects
- Ruby uses *garbage collection*
  - Destroys objects that are no longer reachable

Object Lifetimes

1. Static
2. Stack
3. Explicit heap
4. Implicit heap

Variables by Lifetime: (1) Static

- Bound to memory cells *before execution begins*
  - Not allocated on stack or heap
- Remains bound to same memory *throughout execution*
- Usage: Similar to global variables, but always local to declaring file
- Examples
  - All FORTRAN 77 variables, C `static` variables
  - But *not* C++ class variables

Variables by Lifetime: (1) Static

Example

```
void fn() {
    static int count = 0;
    count ++;
    std::cout << count;
}
fn();
fn();
```

Variables by Lifetime: (1) Static

- Advantages
  - Efficiency – *direct addressing*
  - A subprogram can use across multiple executions
- Disadvantages
  - Bad when value needs to be *reinitialized* (e.g. recursion)
  - Storage can't be shared betweeen subprograms

Variables by Lifetime: (2) Stack

- Created when *execution reaches code*
- Allocated to runtime stack
- Variables may be allocated at beginning of method, even if declared later

Variables by Lifetime: (2) Stack

Example

```
// param, temp, temp2 not allocated here
void fn(int param) {
    int temp;
    int temp2;
}
// param, temp, temp2 now allocated
```

Variables by Lifetime: (2) Stack

- Advantages
  - Good when value needs to be *reinitialized* (e.g. recursion)
  - Conserves storage (deallocated once out of scope)
- Disadvantages
  - Overhead of allocation/deallocation
    - Not too bad, since all memory allocated/deallocated together
  - Subprograms cannot be history-sensitive
  - Inefficient references – *indirect addressing*

Variables by Lifetime: (3) Explicit Heap

- (De)Allocated at runtime by explicit directives
  - e.g. `new`/`delete`, `malloc`/`free`
- Accessed only through *pointers* or *references*
- Examples
  - Dynamic objets in C++
  - All obects in Java

Variables by Lifetime: (3) Explicit Heap

Examples

```
void fn1() {
    int* nums = new int[5];
    // ...
}

public void fn2() {
    Point point = new Point();
    // ...
}
```

Variables by Lifetime: (3) Explicit Heap

- Advantage
  - Don't need to predict exact memory requirements beforehand
  - Can modify if needed, e.g. resizing an array
- Disadvantages
  - Inefficient – *Heap fragmentation* (see next slide)
  - Unreliable – *Dangling pointers*, *memory leaks*

Heap Fragmentation

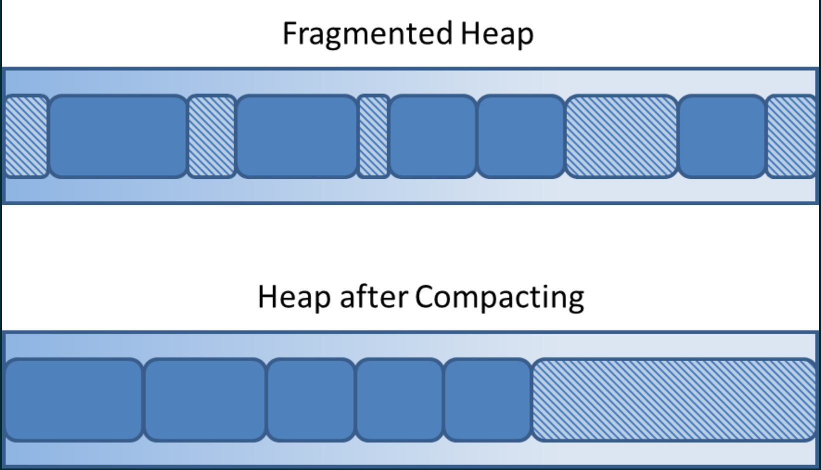

Figure 1: Heap fragmentation example

Variables by Lifetime: (4) Implicit Heap

- Basically same as **Explicit Heap**, except...
  - No `new`/`delete` – these are *implied*
- Identifiers (often) don't have explicit types
  - `x = 3; x = "bob";`
- Examples
  - All variables in APL
  - All strings and arrays in Perl, Javascript

Variables by Lifetime: (4) Implicit Heap

Examples

```
# memory allocation (onto heap) + type binding done at
# declaration
list = [2, 4.33, 6, 8]
```

Variables by Lifetime: (4) Implicit Heap

- Advantage
  - Writeability – Compiler/interpreter handles details
  - Flexibility – Types are implicit
- Disadvantages
  - Inefficient – *Heap fragmentation*
  - Unreliable – Difficult to detect errors (e.g. type errors)

Pointers and References

Pointer Operations (Review)

Two fundamental operations:

1. **Assignment** – used to set pointer variable's value to some useful address
   - `int *ptr = new int;`
2. **Dereferencing** – yields the value stored at pointer's address
   - `*ptr = 206`
   - `int j = *ptr`

Pointers

- Stores a *memory address*
  - Often has special value, e.g. `NULL` or `nil`, but not always (Rust)
- Provide means of *dynamic memory management*
  - Can use to access area where storage is dynamically created (the *heap*)
- Not necessary for all pointers to reference the heap
  - C++ example?

Pointer to Stack Address

In C/C++, it is not necessary for all pointers to reference the heap:

```
int x = 5;
int *ptr = &x;
```

Pointer Operations

- Dereferencing can be *implicit* or *explicit*
- C++ uses an *explicit* operation, via *
  - `j = *ptr; // set j to value stored at ptr`
  - `*ptr = 5; // set value stored at ptr to 5`
- C++ also does *implicit* dereferencing of *reference variables*

```
void fn(int& x) {
    x = 5; // value also changed for caller
}
```

Pointer Arithmetic in C/C++

```
float arr[20]
float *ptr;
ptr = &arr;
```

- `ptr` is an *alias* for `arr`
  - `*(ptr+i)` is equivalent to `stuff[i]` and `ptr[i]`

Pointer Arithmetic in C/C++

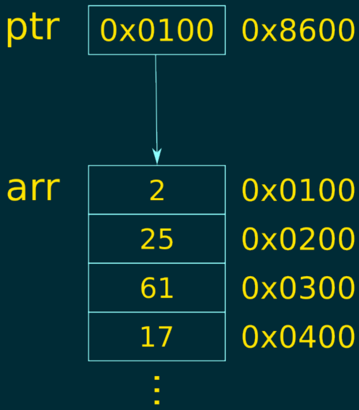

Figure 2: Pointer as alias to Array

Pointers in C/C++: `void*`

- Domain type need not be fixed: `void*`
  - `void*` can point to *any type*
    - Use typecasts when needed, e.g. `(int*)void_ptr ...`
  - `void*` *cannot* be dereferenced
  - `void*` often used in C to pass as arguments
- In C++, generally better to use templates so compiler can do appropriate type-checking

Question

Do you remember the difference between a *dangling pointer* and a *memory leak*?

Problems with Pointers (review)

- Dangling pointers
  - Pointer pointing to *heap-dynamic* variable that has been deallocated
  - That memory *may* have been reallocated
  - Value no longer meaningful
  - Writing to it could corrupt memory
- Example

```
Point p = new Point(3, 4);
delete p; // dangling -- p still has address!
std::cout << p.getX(); // bad!
```

Problems with Pointers (review)

- Memory leak
  - Memory has *not* been deleted/returned to heap manager
  - *Inaccesible:* No variables contain the address
- When is this a problem?
  - One-off programs, small school assignments? No...
  - Long running programs, e.g. web servers? Yep...

```
int[] p = new int[5000];
p = new int[10000]; // p contains new address
```

Reference Types

C++ includes a special kind of pointer typed, called a *reference type*

- Used primarily for formal parameters
- Constant pointer* that is always *implicitly dereferenced*
  - Notice no * in the code below

```
void fn(int &y) {
    y = y + 1;
}
```

*What does *constant pointer* mean?

Reference Types: Point of confusion

- *Constant pointer*
  - *Can't* change *where* it points
  - *Can* change *contents*
- Java
  - Uses references to objects, but *can* change address it references
  - Implicitly dereferenced
  - No pointer arithmetic – Java *does not* have pointers
- C# has references like Java and pointers like C++

Pointers vs. References

Broadly speaking:

- Pointers
  - *Do* support pointer arithmetic
  - Must be *explicitly* dereferenced
- References
  - *Do not* support pointer arithmetic
  - Are *implicitly* dereferenced

Mutability of address/contents depends on context

What about Ruby?

- Does Ruby have references or pointers?
  - A: References (read)
- Ruby also has *garbage collection (GC)**

*What problem does GC solve? (Dangling pointers, memory leaks?)