Programming Languages

Memory Allocation, Garbage Collection

CSCI.GA-2110-001
Spring 2023

Dynamic memory management

For most languages, the amount of memory used by a program cannot be determined at compile time

- earlier versions of FORTRAN are exceptions!

Some features that require dynamic memory allocation:

- recursion
- pointers, explicit allocation (e.g., `new`)
- higher order functions

Types of Allocation

- Static – absolute address retained throughout program's execution.
  - Static variables
  - Global variables
  - Certain fixed data (e.g., string literals, constants)
- Stack – last-in, first-out (LIFO) ordering.
  - Subroutine arguments
  - Local variables
  - Runtime system data structures (displays, etc.)
- Heap – general storage, for allocation at arbitary times.
  - Explicitly or automatically allocated
  - Resizable types (e.g., String)
  - Java class instances
  - All objects and data structures in Python

Heap Allocation

The heap is finite – if we allocate too much space, we will run out.

Solution: deallocate space when it is no longer necessary.

Methods:

- Manual deallocation, with e.g., `free`, `delete` (C, Pascal)
- Automatic deallocation via garbage collection (Java, C#, Scheme, ML, Perl)
- Semi-automatic deallocation, using destructors (C++, Ada)
  - Automatic because the destructor is called at certain points automatically
  - Manual because the programmer writes the code for the destructor

Manual deallocation is dangerous (because not all current references to an object may be visible).

Heap Allocation

Most languages permit custom memory allocation/deallocation.
Some permit overloading the allocation/deallocation operators (`new`, `delete`, etc.). in C++:

```
class Foo {
  // data members here

public:
static void* operator new (unsigned int num_bytes) {}
static void operator delete(void* p) {}
};
```

Usage:

```
Foo* f = new Foo;
```

Heap Allocation

Programming language C contains a library of helpful memory functions:

1. `malloc` : allocate memory from the heap.
2. `alloca` : allocates memory from the stack. Automatically freed.
3. `calloc` : allocate zero-initialized memory from the heap.
4. `realloc` : increases the size of an already allocated block.

Use `free` to deallocate memory allocated above (except `alloca`).

Heap Allocation

Control over allocation is essential in some applications.

Object *construction* often accompanies allocation. C++ example:

```
Foo myArray[250]; // allocate and call constructor 250 times.
```

Sometimes we can't afford to slow down the program like this. Also, C++ won't let us use anything but a default constructor.

Solution: allocate the memory now, construct objects later.

```
Foo* myArray = (Foo*)malloc(sizeof(Foo)*250);
...
new (myArray+x) Foo(); // invoke constructor at myArray[x]
```

This is called *placement-new*. *Any* constructor can be called (not just default). Call be invoked again at any time without deallocating/allocating memory.

Allocation Methods

Two basic methods:

- free list – typically for manual and semi-automatic deallocation
- heap pointer – typically for automatic deallocation

Free list method:

- a linked list of unused blocks of memory is maintained (the *free list*)
- **Allocation**: a search is done to find a free block of adequate size; it's removed from the free list
  - first-fit, best-fit
- **Deallocation**: the block is placed on the free list

Problems:

- may take some time to find a free block of the right size
- memory eventually becomes fragmented

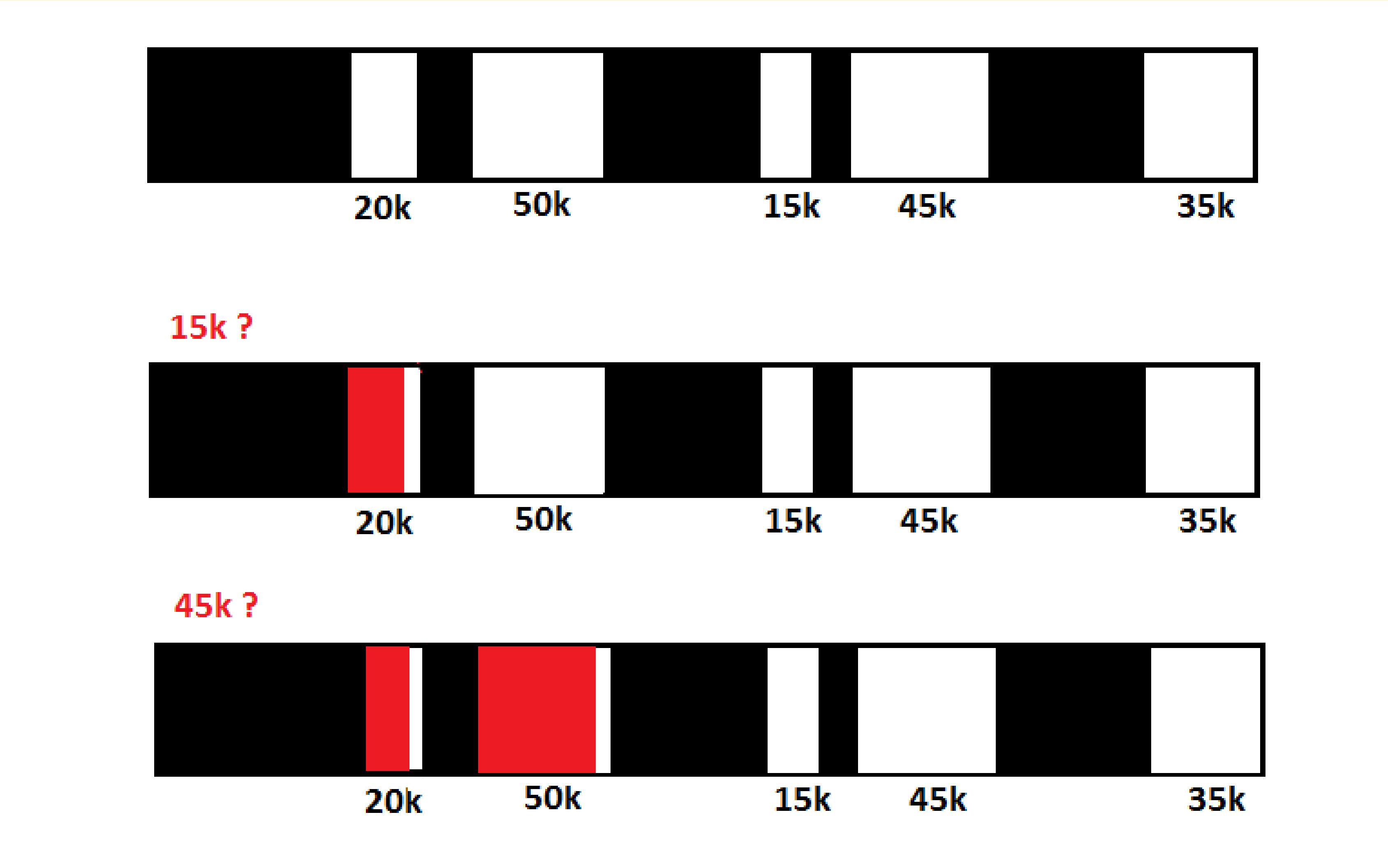
Allocation Methods

- First fit: select the first block large enough to satisfy the request.
- Best fit: select the *smallest* block large enough to satisfy the request.
- Worst fit: always select the *largest* available block.

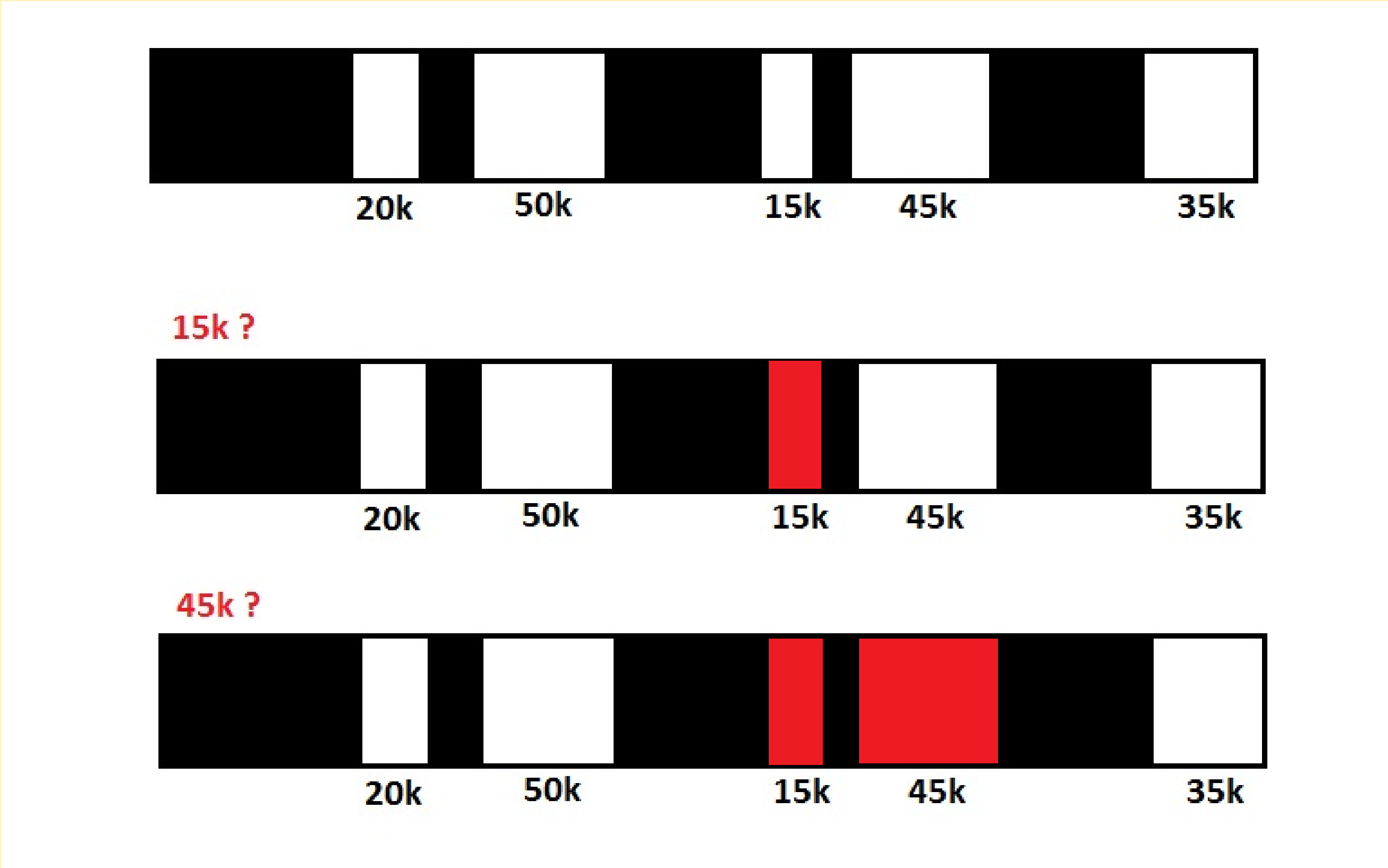
All may suffer from fragmentation:

- Internal fragmentation: memory allocated but not used. Typical for fixed block allocation.
- External fragmentation: available memory blocks too small to be used.

First Fit


20k
50k
15k
45k
35k
15k ?
20k
50k
15k
45k
35k
45k ?
20k
50k
15k
45k
35k

Best Fit



20k
50k
15k
45k
35k
15k ?
20k
50k
15k
45k
35k
45k ?
20k
50k
15k
45k
35k

Allocation: Heap pointer

Heap pointer method:

- initially, the heap pointer is set to bottom of heap
- **Allocation**: the heap pointer is incremented an appropriate amount
- **Deallocation**: defragmentation eventually required

Problems:

- requires moving live objects in memory

Automatic deallocation

An automatically deallocated language is said to be *garbage collected*.

What languages use this?

- All functional languages (Lisp, ML, Scheme, etc.)
- Modula-3
- Eiffel
- Ruby
- Java
- JavaScript
- .NET ("managed" languages)

Even languages that don't natively support garbage collection can be garbage collected (e.g., C++).

Automatic deallocation

Basic garbage collection algorithms:

- mark/sweep – needs run-time support
  - variant: compacting
  - variant: non-recursive
  - variant: concurrent
- copying – needs run-time support
  - variant: incremental
  - variant: generational
- hybrid – combination copy and mark & sweep
  - Most production collectors use hybrid
  - variant: Garbage First (G1)
- reference counting – usually done by programmer

Mark/sweep & Copying GC

An object `x` is *live* (i.e., can be referenced) if:

- `x` is pointed to by some variable located
  - on the stack (e.g., in an activation record)
  - in static memory
- there is a register (containing a temporary or intermediate value) that points to `x`
- there is another object on the heap (e.g., `y`) that is live and points to `x`

All live objects in the heap can be found by a graph traversal:

- start at the *roots* – local variables on the stack, static memory, registers.
- any object not reachable from the roots is *dead* and can be reclaimed

Mark/sweep

- each object has an extra bit called the *mark bit*
- **mark phase**: the collector traverses the heap and sets the mark bit of each object encountered
- **sweep phase**: each object whose mark bit is not set goes on the free list

name	definition

`GC()`	`for each root pointer p do`
	`   mark(p);`
	`sweep();`

`mark(p)`	`if p->mark /= 1 then`
	`   p->mark = 1;`
	`   for each pointer field p->x do`
	`      mark(p->x);`

`sweep()`	`for each object x in heap do`
	`   if x.mark = 0 then insert(x, free_list);`
	`                   else x.mark = 0;`

Copying

- heap is split into 2 parts: **FROM** space, and **TO** space
- objects allocated in **FROM** space
- when **FROM** space is full, garbage collection begins
- during traversal, each encountered object is copied to **TO** space
- when traversal is done, all live objects are in **TO** space
- now we flip the spaces – **FROM** space becomes **TO** space and vice versa
- Note: since we are moving objects, any pointers to them must be updated
  This is done by leaving a *forwarding address*

heap pointer method used for allocation – fast

Copying

name	definition
`GC()`	`for each root pointer p do` `    p := traverse(p);`

`traverse(p)`	`if *p contains forwarding address then`
	`   p := *p;  // follow forwarding address`
	`   return p;`
	`else {`
	`   new_p := copy (p, TO_SPACE);`
	`   *p := new_p; // write forwarding address`
	`   for each pointer field p->x do`
	`      new_p->x := traverse(p->x);`
	`   return new_p;`
	`}`

Generational GC

- a variant of a copying garbage collector
- Observation: the older an object gets, the longer it is expected to stay around.
  Why?
  - many objects are very short-lived (e.g., intermediate values)
  - objects that live for a long time tend to make up central data structures in the program, and will probably be live until the end of the program
- Idea: instead of 2 heaps, use many heaps, one for each “generation”
  - younger generations collected more frequently than older generations (because younger generations will have more garbage to collect)
  - when a generation is traversed, live objects are copied to the next-older generation
  - when a generation fills up, we garbage collect it

Garbage First (G1) in Java

- Another variant of a copying garbage collector
- Intended to replace Concurrent Mark and Sweep.
- Default collector in Java 9.
- Goal: reduce pause times and make them more predictable.
- Idea: split up heap memory into a matrix of fixed-sized smaller *regions*.
  - number/size of regions chosen by JVM (usually ˜2,000).
  - same generations as before (Eden, survivors, etc.) but now partitioned into regions—physically scattered throughout heap memory.
  - copy collection occurs between 2 regions.
  - heaps with the most dead objects preferred for collection first.
  - Also: "humongous" region, multiple contiguous regions.

Collection Modes

- **Serial**: Single thread. Application must be stopped during collection.
- **Parallel**: Multi-threaded. Perform collections frequently, but pauses the application each time. Max throughput.
- **Concurrent**: Multi-threaded. Performs collections while the application is running. Max response time.

Reference Counting

The problem:

- we have several references to some data on the heap
- we want to release the memory when there are no more references to it
- may not have “built-in” garbage collection

Idea: Keep track of how many references point to the data, and free it when there are no more.

- set reference count to 1 for newly created objects
- increment reference count whenever we make a copy of a pointer to the object
- decrement reference count whenever a pointer to the object goes out of scope or stops pointing to the object
- when an object’s reference count becomes 0, we can free it

Reference Counting

Advantages:

- Memory can be reclaimed as soon as no longer needed.
- Simple, can be done by the programmer for languages not supporting GC.

Disadvantages:

- Additional space needed for the reference count.
- Will not reclaim circular references.
- Can be inefficient (e.g., if many objects are reclaimed at once).

Why reference counting is slow

Consider this seemingly innocuous code:

```
ptr2 = ptr1
```

Look at all the events required just to do this:

1. Lock `ptr2` (possible contention).
2. Decrement old.
3. Test old against 0.
4. Possible deletion.
5. Unlock `ptr2`.
6. Lock `ptr1` (possible contention).
7. Increment new.
8. Unlock `ptr1`.

C++: important lifetime events

Event	what gets called (declaration)
Creation	`C (...)        // constructors`
Pass by value	`C (const C&) //copy constructor`
Assignment	`C& operator= (const C&)`
Destruction	`~C ()          // destructor`

A chief reason C++ has destructors is to enable implementation of *reference counting*.

Reference Counting: Example

```
class C {
public:
   C () : p(NULL) { }
   C (const C& c) : p(c.p) { if (p) p->refCount++; }
   ~C () { if (p && --p->refCount == 0) delete p; }
   C& operator= (const C&);
   ...
private:
   struct RefCounted {
      int refCount;
      ...
      RefCounted (...) : refCount(1), ... { ... }
   };
   RefCounted *p;
}
```

Reference Counting: assignment

```
const C& C::operator= (const C& c) {
  if (c.p)
    c.p->refCount++;

  if (p && --p->refCount == 0) delete p;

  p = c.p;

  return *this;
}
```

Conservative collection

- What about weakly typed languages?
- What about languages not designed for GC? (hostile environments)

It turns out that strong typing is not necessary for garbage collection.

Approach: traverse the roots and *guess* whether bit patterns "look like" a pointer.

- Let x be a bit pattern reachable from the roots.
- Was a memory block beginning at address x was previously allocated? If so, assume the bit pattern is a pointer. If not, skip.
- If assumed a pointer, include it in the garbage collection.
- Consequence: there may be a bit pattern that "accidentally" coincides with a block of previously allocated memory. Worst case: the memory won't be reclaimed.
- Algorithm is safe: memory will never be reclaimed if still referenced somewhere.