MEMORY MANAGEMENT IN PROGRAMMING LANGUAGES

LECTURE # 5

WHAT IS MEMORY MANAGEMENT?

- Memory management is the process of controlling and coordinating the way a software application access **computer memory**
- When a software runs on a target Operating system on a computer it needs access to the computers **RAM**(Random-access memory) to:
  - load its own **bytecode** that needs to be executed
  - store the **data values** and **data structures** used by the program that is executed
  - load any **run-time systems** that are required for the program to execute

REVIEW DEFINITIONS

- Method: any subprogram (function, procedure, subroutine) – depends on language terminology.
- Environment of an active method: the variables it can currently access plus their addresses (a set of ordered pairs)
- State of an active method: variable/value pairs

THREE CATEGORIES OF MEMORY (FOR DATA STORE)

- Static: storage requirements are known prior to run time; lifetime is the entire program execution
- Run-time stack: memory associated with active functions
  - Structured as stack frames (activation records)
- Heap: dynamically allocated storage; the least organized and most dynamic storage area

STATIC DATA MEMORY

- Simplest type of memory to manage.
- Consists of anything that can be completely determined at compile time; *e.g.*, global variables, constants (perhaps), code.
- Characteristics:
  - Storage requirements known prior to execution
  - Size of static storage area is constant throughout execution

RUN-TIME STACK

- The stack is a contiguous memory region that grows and shrinks as a program runs.
- Its purpose: to support method calls
- It grows (storage is allocated) when the activation record (or stack frame) is pushed on the stack at the time a method is called (activated).
- It shrinks when the method terminates and storage is de-allocated.

RUN-TIME STACK

- The stack frame has storage for local variables, parameters, and return linkage.
- The size and structure of a stack frame is known at compile time, but actual contents and time of allocation is unknown until runtime.
- How is variable lifetime affected by stack management techniques?

HEAP MEMORY

- Heap objects are allocated/deallocated dynamically as the program runs (not associated with specific event such as function entry/exit).
- The kind of data found on the heap depends on the language
  - Strings, dynamic arrays, objects, and linked structures are typically located here.
  - Java and C/C++ have different policies.

HEAP MEMORY

- Special operations (e.g., **malloc, new)** may be needed to allocate heap storage.
- When a program deallocates storage (**free, delete**) the space is returned to the heap to be re-used.
- Space is allocated in variable sized blocks, so deallocation may leave "holes" in the heap (fragmentation).
  - Compare to deallocation of stack storage

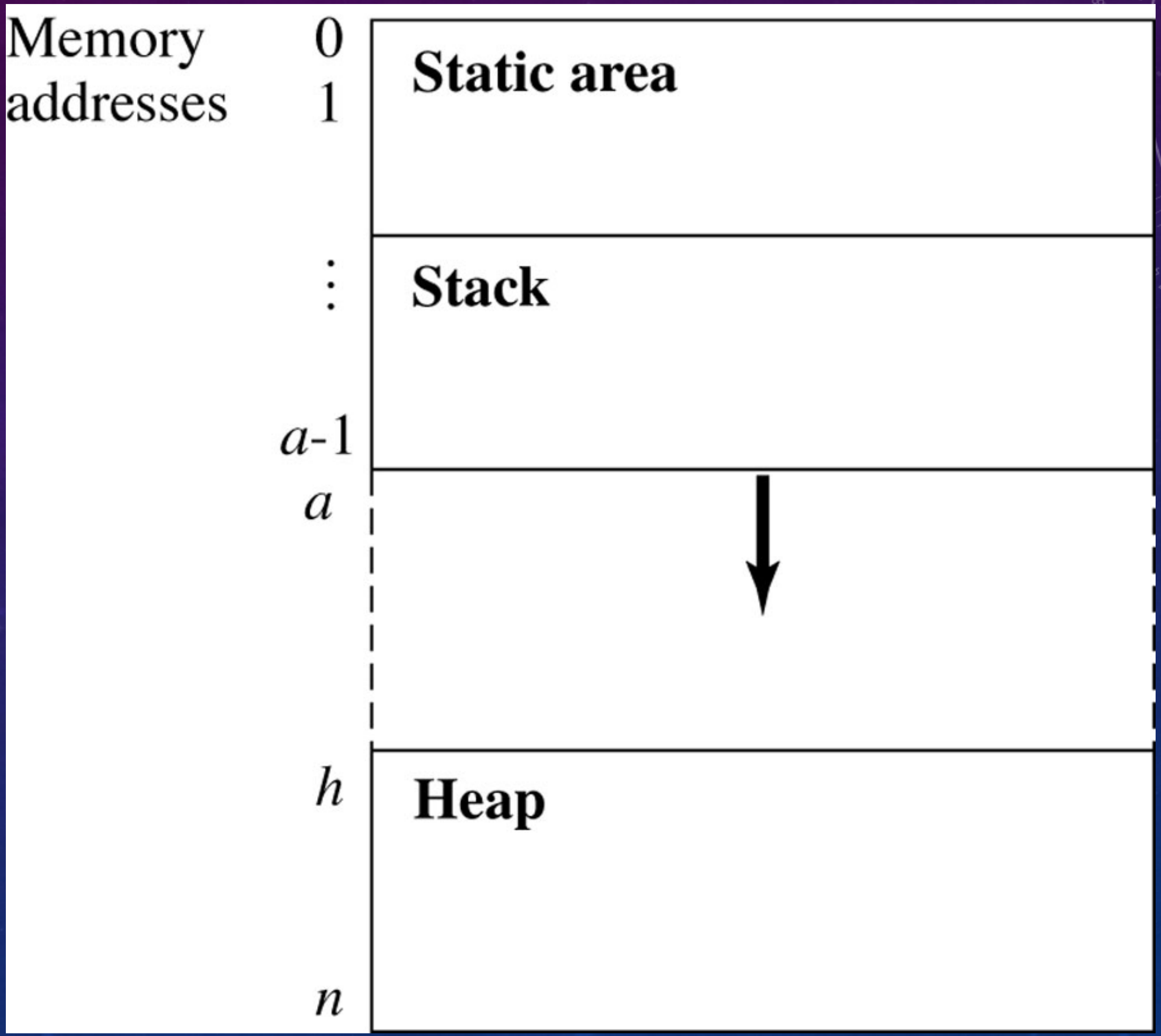
HEAP MANAGEMENT

- Some languages (e.g. C, C++) leave heap storage deallocation to the programmer
  - `delete`
- Others (e.g., Java, Perl, Python, list-processing languages) employ garbage collection to reclaim unused heap space.

The Structure of Run-Time Memory

Figure 11.1

These two areas grow towards each other as program events require.



STACK OVERFLOW

- The following relation must hold:
  $0 \leq a \leq h \leq n$
- In other words, if the stack top bumps into the heap, or if the beginning of the heap is greater than the end, there are problems!

ARRAY OUT-OF-BOUNDS VIOLATIONS IN C

- No run-time type-checking is taking place as a C program executes.
- No-one is tracking array bounds violations

HEAP STORAGE STATES

- For simplicity, we assume that memory words in the heap have one of three states:
  - Unused: not allocated to the program yet
  - Undef: allocated, but not yet assigned a value by the program
  - Contains some actual value

HEAP MANAGEMENT FUNCTIONS

- *new* returns the start address of a block of *k* words of unused heap storage and changes the state of the words from *unused* to *undef*.
  - $n \leq k$, where n is the number of words of storage needed; *e.g.*, suppose a Java class Point has data members x,y,z which are floats.
  - If floats require 4 bytes of storage, then
    Point firstCoord = new Point( )
    calls for 3 X 4 bytes (at least) to be allocated and initialized to some predetermined state.

HEAP OVERFLOW

- Heap overflow occurs when a call to *new* occurs and the heap does not have a contiguous block of *k* unused words
- So *new* either fails, in the case of heap overflow, or returns a pointer to the new block

HEAP MANAGEMENT FUNCTIONS

- *delete* returns a block of storage to the heap
- The status of the returned words are returned to *unused*, and are available to be allocated in response to a future *new* call.
- One cause of heap overflow is a failure on the part of the program to return unused storage.

The New (5) Heap Allocation Function Call: Before and After

h

7	*undef*	12	0
3	*unused*	*unused*	*unused*
undef	0	*unused*	*unused*
unused	*unused*	*unused*	*unused*

...

n

h

7	*undef*	12	0
3	*unused*	*unused*	*unused*
undef	0	*undef*	*undef*
undef	*undef*	*undef*	*unused*

...

n

A before and after view of the heap. The "after" shows the affect of an operation requesting a size-5 block. (Note difference between "undef" and "unused".) Deallocation reverses the process.

HEAP ALLOCATION

- Heap space isn't necessarily allocated and deallocated from one end (like the stack) because the memory is not allocated and deallocated in a predictable (first-in, first-out or last-in, first-out) order.
- As a result, the location of the specific memory cells depends on what is available at the time of the request.

MEMORY ATTRIBUTES

- Memory to store data in programming languages has the following lifecycle
  - Allocation: When the memory is allocated to the program
  - Lifetime: How long allocated memory is used by the program
  - Recovery: When the system recovers the memory for reuse

MEMORY CLASSES

- **Static memory – Usually at a fixed address**
  - Lifetime – The execution of program
  - Allocation – For entire execution
  - Recovery – By system when program terminates
  - Allocator – Compiler
- **Automatic (LIFO) memory – Usually on a stack**
  - Lifetime – Activation of method using that data
  - Allocation – When method is invoked
  - Recovery – When method terminates
  - Allocator – Typically compiler, sometimes programmer

MEMORY CLASSES …CONT

- Dynamic memory – Addresses allocated on demand in an area called the heap
  - Lifetime – As long as memory is needed
  - Allocation – Explicitly by programmer, or implicitly by compiler
  - Recovery – Either by programmer or automatically (when possible and depends upon language)
  - Allocator – Manages free/available space in heap

MEMORY MANAGEMENT IN C

- Local variables live on the stack
  - Allocated at function invocation time
  - Deallocated when function returns
  - Storage space reused after function returns
- Space on the heap allocated with malloc()
  - Must be explicitly freed with free()
  - Called explicit or manual memory management
  - Deletions must be done by the user

MEMORY MANAGEMENT

- Computer programs need to allocate memory to store data values and data structures.
- Memory is also used to store the program itself and the run-time system needed to support it.
- If a program allocates memory and never frees it, and that program runs for a sufficiently long time, eventually it will run out of memory.
- Even in the presence of virtual memory, memory consumption is still a major issue because it is considerably less efficient to access virtual memory than to access physical memory

MANUAL & AUTOMATIC MEMORY MANAGEMENT

- **Automatic memory management**
- **Which ask the programmer to allocate and free memory manually.**
  - The C language requires the programmer to implement memory management each time, for each application program.
  - Modern programming languages such as Java, C#, Caml, Cyclone and Ruby provide automatic memory management with garbage collection

MANUAL MEMORY MANAGEMENT

- In C, where there is no garbage collector, the programmer must allocate and free memory explicitly.
  - The key functions are malloc and free.
- The malloc function takes as a parameter the size in bytes of the memory area to be allocated.
- The size of a type can be obtained using sizeof.
- The resulting area of memory does not represent a value of the correct type, so it then needs to be cast to the correct type.

```
p = (Type_t*) malloc(sizeof(Type_t));
```

DANGLING POINTER PROBLEM

- A significant problem with manual memory management is that it is possible to attempt to use a pointer after it has been freed.
- This is known as the dangling pointer problem. Dangling pointer errors can arise whenever there is an error in the control flow logic of a program.
- This can lead to allocation, use and deallocation happening in the wrong order in some circumstances
- Use before allocation may be a fatal run-time error. Use after deallocation is not always fatal. Neither of these is a good thing.

DANGLING POINTER

```
{ int *x = ...malloc();
  free(x);
  *x = 5; /* oops! */
}
```

SPACE/MEMORY LEAK

- Another potential problem of manual memory management is not remembering to free allocated memory when it should be freed.
- The reference to an allocated area of memory can be lost when a variable in a block-structured language goes out of scope.
- This problem is perhaps more subtle than the dangling pointer problem because it may only become manifest for long-running applications.
- When memory is lost and cannot be reclaimed we term this a space leak. Space cannot be lost forever without reaching the limit on the available memory.
- sA long-running program with a space leak will eventually crash.

MEMORY LEAK

```
{ int *x = (int *) malloc(sizeof(int)); }
```

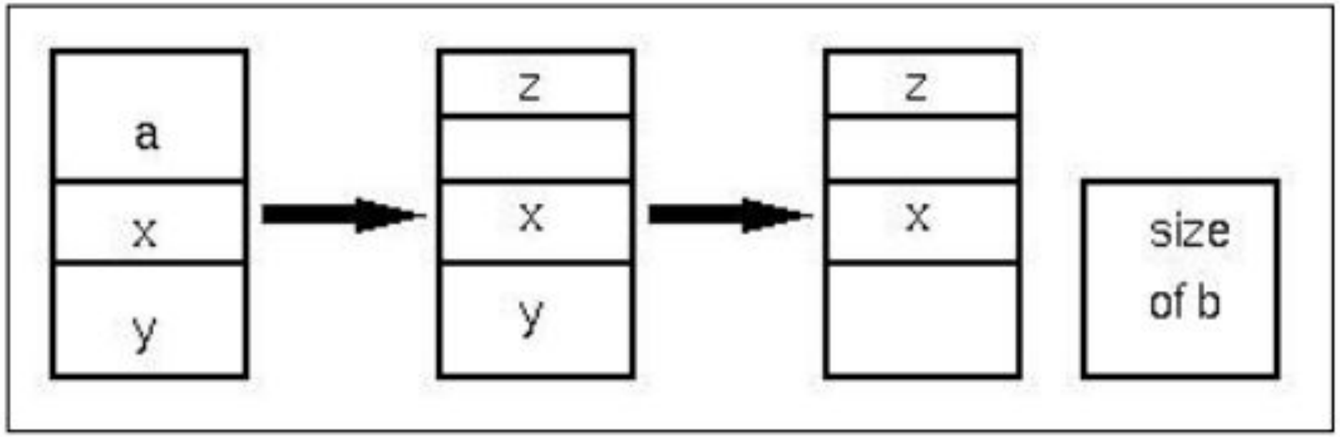
MAY FREE SOMETHING TWICE

```
{ int *x = ...malloc(); free(x); free(x); }
```

FRAGMENTATION

- Another memory management problem
- Example sequence of calls

```
allocate(a);
allocate(x);
allocate(y);
free(a);
allocate(z);
free(y);
allocate(b);
```



⇒ Not enough contiguous space for b

AUTOMATIC MEMORY MANAGEMENT

- Primary goal: automatically reclaim dynamic memory.
- Secondary goal: also avoid fragmentation