



Memory Management

CS31

Pascal Van Hentenryck



Memory Management

What is it?

- high-level languages abstract many aspects of memory management
- Support varies with the language

Java (ML/Prolog/Lisp/Smalltalk)

- Highest level of abstraction

C++

- Explicit destruction of objects

C

- High-level assembly language

Memory Management

Two types of memory allocation

- Stack allocation
- Heap allocation

Stack allocation

- Parameters
- Return address
- Local variables

Heap Allocation

- Objects (Java, C++)
- Dynamic data in C

MM in Java

Memory allocation

- `new` allocates memory for an object

Memory deallocation

- No explicit deallocation
- Objects are deallocated whenever they are not needed
 - ▶ How do you know that?
- Garbage collection

Simple Lists

Conventions

- A list is a pointer to a list node
- A null reference/pointer indicates an empty list

List Node

- A key (e.g., an int)
- A pointer to the next element in the list
- The pointer/reference is null when there are no further element

Close to lists in Lisp/Scheme

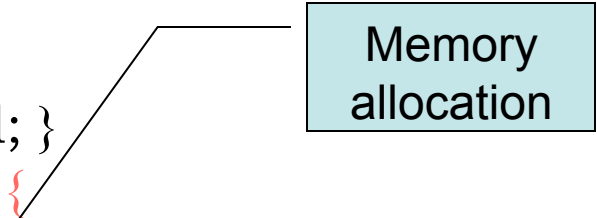
Simple Lists in Java

```
class List {  
    ListNode _head;  
    List() { _head = null; }  
}
```

```
Class ListNode {  
    int      _key;  
    ListNode _next;  
    ListNode(int key,ListNode n) {  
        _key = key; _next = n;  
    }  
    int getKey() { return _key; }  
    ListNode getNext() { return _next; }  
    void setNext(ListNode n) { _next = n;}  
}
```

MM in Java

```
class List {  
    ListNode _head;  
    List() { _head = null; }  
    void insert(int key) {  
        _head = new ListNode(key, _head);  
    }  
    void remove(int key) { ... }  
}
```



```
Class ListNode {  
    int    _key;  
    ListNode _next;  
    ListNode(int key, ListNode n) {  
        _key = key; _next = n; }  
    int getKey() { return _key; }  
    ListNode getNext() { return _next; }  
    void setNext(ListNode n) { _next = n; }  
  
    ListNode remove(int key) { ... }  
}
```

MM in Java

```
class ListNode {
```

```
...
```

```
ListNode remove(int key) {
```

```
    if (key == _key)
```

```
        return _next;
```

```
    else {
```

```
        _next = _next.remove(key);
```

```
        return this;
```

```
    }
```

```
}
```

Possible memory deallocation

Possible memory deallocation

```
class List {
```

```
...
```

```
void remove(int key) {
```

```
    _head = _head.remove(key);
```

```
}
```

```
}
```

Possible memory deallocation

MM in Java

```
class ListNode {
```

```
...
```

```
ListNode remove(int key) {
```

```
    if (key == _key)
```

```
        return _next;
```

```
    else {
```

```
        _next = _next.remove(key);
```

```
        return this;
```

```
    }
```

```
}
```

Possible memory deallocation

Possible memory deallocation

```
class List {
```

```
...
```

```
void remove(int key) {
```

```
    _head = _head.remove(key);
```

```
}
```

```
}
```

Possible memory deallocation

Space complexity?

- Linear in the worst case

MM in Java

Why is it linear in space?

- Each recursive call takes space on the stack
- There are as many recursive as they are elements in the list in the worst case
- This is a terrible implementation from an efficiency standpoint!

How to improve it?

- Do not use the stack!
- Use iterations/loops

MM in Java

```
void remove(int key) {
    ListNode cur = _head;
    ListNode prev = null;
    while (cur != null) {
        if (cur.getKey() == key)
            break;
        prev = cur;
        cur = cur.getNext();
    }
    if (cur != null) {
        if (prev != null)
            prev.setNext(cur.getNext());
        else
            _head = cur.getNext();
    }
}
```

MM in Java

```
ListNode remove(int key) {
    ListNode cur = _head;
    ListNode prev = null;
    while (cur != null) {
        if (cur.getKey() == key)
            break;
        prev = cur;
        cur = cur.getNext();
    }
    if (curr != null) {
        if (prev != null)
            prev.setNext(cur.getNext());
        else
            _head = cur.getNext();
    }
}
```

What happens to the removed element?

- Java will garbage-collect it at some point.

How do we figure out what is garbage?

- We do not need it any more

MM in C++

High-level View

- Similar to Java
- Explicit memory deallocation
- No garbage collection

Programmers are responsible to take their garbage out

Main Support

- Destructors for deallocation

This lecture

- Java with the C++ model
- C++ is much more complicated
 - ▶ pointers, superset of C

MM in C++

C++ automates many aspects

- Memory allocation
- Memory deallocation

Main abstraction

- No need to know how memory is allocated/deallocated

Main responsibility

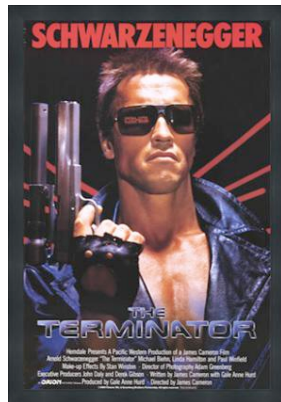
- Programmers must specify which objects are no longer needed

Nasty bugs

- Segmentation faults
- Bus error
- Memory leaks

MM in C++

```
class ListNode {  
    int    _key;  
    ListNode _next;  
    ListNode(int key, ListNode n) {  
        _key = key, _next = n;  
    }  
    ~ListNode() {}  
    ...  
}
```



Terminator

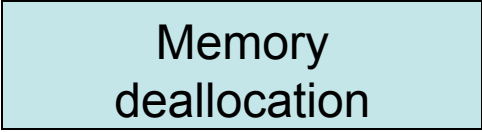
MM in C++

When do I need to delete?

- a node which is deleted
- the list and all its nodes when the list is deleted

MM in C++

```
void remove(int key) {  
    ListNode cur = _head;  
    ListNode prev = null;  
    while (cur != null) {  
        if (cur.getKey() == key)  
            break;  
        prev = cur;  
        cur = cur.getNext();  
    }  
    if (curr != null) {  
        if (prev != null)  
            prev.setNext(cur.getNext());  
        else  
            _head = cur.getNext();  
        delete curr;  
    }  
}
```



MM in C++

```
class List {  
    ListNode _head;  
    List() { ... }  
    ~List() {  
        ListNode n;  
        ListNode c = _head;  
        while (c!=null) {  
            n = c;  
            c = c.getNext();  
            delete n;  
        }  
    }  
    ...  
}
```

MM in C++



Memory bugs

- Using a freed object
 - ▶ The data will become incorrect
- Freeing an object twice
 - ▶ The space may be allocated twice
- Forgetting to release an object
 - ▶ Your program eats up the whole memory

Nasty bugs

- Time discrepancy between the bug and the symptoms
- Things generally look fine until the space is reallocated to another object, at which time values may start being overwritten.
- The new values may have no meaning in the original context.

MM in C

C is a high-level assembly language

- Mixture of Pascal and Assembly
- Predecessor was B
- Successor D never saw the light

Basic features

- Explicit memory management
- Bits and bytes and words ...
- Casting
- Pointer manipulation
- Pointers to any type (int, ...)
- Structures on the stack
- No objects
- Only functions

MM in C

```
struct ListNode {  
    int _key;  
}  
  
void test() {  
    ListNode n;  
    n._key = 5;  
}
```

Structure declaration

Structure instance

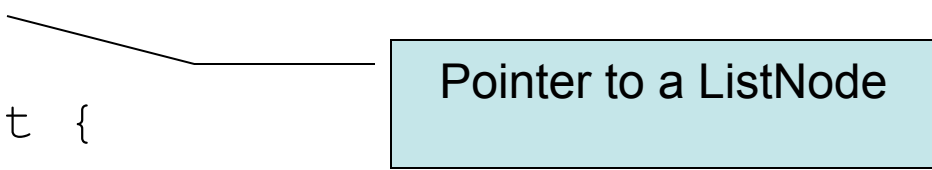
Accessing the structure fields

Observations

- `ListNode` is allocated on the stack
- It does not exist after the function call
- We need pointers to structures

MM in C

```
struct ListNode {
    int      _key;
    ListNode* _next;
}
struct List {
    ListNode* _head;
}
ListNode* search(List* l, int key) {
    ListNode* c = l->_head;
    while (c) {
        if (c->_key == key)
            return c;
        else
            c = c->_next;
    }
}
```



Pointer to a ListNode

We are manipulating pointers!

MM in C



How to allocate memory?

- `char* malloc(int sizeb)`
- returns a pointer to `sizeb` bytes of memory space
- `sizeof` can be used to know the size of structures and other types

How to deallocate memory?

- `free(void* ptr)`
- frees the space allocated to `ptr`



MM in C

```
ListNode*
createNode(int key, ListNode* next)
{
    ListNode* n;
    n =
    (ListNode*) malloc(sizeof(ListNode));
    n->_key = key;
    n->_next = next;
    return n;
}
```

More bugs

- Allocate the wrong space
- Change the pointers

```
n = n + 345;
```

- frees the wrong pointers
- Frees/returns objects from the stack

