Lecture 12: Arrays, Pointers, Recursive Types, & Garbage Collection

COMP 524 Programming Language Concepts
Aaron Block
February 27, 2007

Based on notes by N. Fisher, F. Hernandez-Campos, and D. Stotts
Goals

• Discuss Arrays, Pointers, Recursive Types, and Garbage Collection
Arrays

- Arrays are usually stored in contiguous locations.
Arrays

• Arrays are usually stored in contiguous locations
char days[][10] = {
    "sun", "mon", "tue",
    "wed", "thu", "fri",
    "sat"
};

char *days[] = {
    "sun", "mon", "tue",
    "wed", "thu", "fri",
    "sat"
};

\[
S3 = \text{size of element}
\]
\[
S2 = (U3-L3+1) \times S3
\]
\[
S1 = (U2 - L2 + 1) \times S2
\]

Address of \( A[i,j,k] \)

Address of \( A \) + \((i-L1) \times S1 + (j-L2) \times S2 + (k-L3) \times S3\)
A: array \([L1..U1]\) of array \([L2..U2]\) of array \([L3..U3]\) of element_type

Optimized
\[(i \times S1) + (j \times S2) + (k \times S3) + \text{address } A - ((L1 \times S1) + (L2 \times S2) + (L3 \times S3))] \]

the phrase \([((L1 \times S1) + (L2 \times S2) + (L3 \times S3))]\) can be determined at compile-time
Heap-based Allocation

- The **heap** is a region of storage in which sub-blocks can be allocated and deallocated.
Issues with Heap Allocation

• Pointers
  • Used in value model of variables
  • Not necessary for reference model

• Dangling References

• Garbage Collection
Pointers

• Pointers serve two purposes:
  • Efficient (and sometimes intuitive) access to elaborated objects (as in C)
  • Dynamic creation of linked data structures, in conjunction with a heap storage manager.

• Several Languages (e.g., Pascal) restrict pointers to accessing things in the heap

• Pointers are used with a value model of variables
  • They aren’t needed with a reference model (already implicit)
C Pointers and Arrays

- C Pointers and array

\[
\text{int } *a == \text{int a[]}
\]

\[
\text{int } **a == \text{int } *a[]
\]
C Pointers and Arrays

• But equivalencies don’t always hold
  • Specifically, a deceleration allocates an array if it specifies a size for the first dimension
  • otherwise it allocates a pointer

```
int **a, int *a[] // Pointer to pointer to int
int *a[n] // n-element array of row pointers
int a[n][m] // 2-D array
```
Recursive Types (now with Pointers!): Binary Tree

C

```c
struct chr_tree{
    struct chr_tree *l, *r;
    char var;
}
```

Ada

```ada
type chr_tree;  
type chr_tree_ptr is access chr_tree;  
type chr_tree is record  
    left,right:chr_tree_ptr;  
    val:characte;  
end record;
```
Binary Tree with Explicit Pointers
Lists

- A list is defined recursively as either the empty list or a pair consisting of an object (which may be either a list or an atom) and another (shorter) list
  - Lists are ideally suited to programming in functional and logical languages
    - In Lisp, in fact, a program is a list, and can extend itself at run time by constructing a list and executing it.
  - Lists can also be used in imperative programs
  - We’ll see more in the future
Problems with Explicit Reclamation

• Explicit reclamation of heap objects is problematic
  • The programmer may forget to deallocate some objects
    • Causing memory leaks
    • For example, in the previous example, the programmer may forget to include the delete statement
  • References to deallocated objects may not be reset
    • Creating dangling references

```cpp
ptr1

<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

ptr2
```
Problems with Explicit Reclamation

• Explicit reclamation of heap objects is problematic
  • The programmer may forget to deallocate some objects
    • Causing memory leaks
    • For example, in the previous example, the programmer may forget to include the delete statement
  • References to deallocated objects may not be reset
    • Creating dangling references
Dealing with Dangling References

• **Tombstones**: Use an intermediary device

• **Lock and Keys**: Use a universal key.
new(ptr1);

ptr2 := ptr1;

delete(ptr1);
new(ptr1);

ptr2 := ptr1;

delete(ptr1);
Garbage Collection

- Automatic reclamation of the **space used by objects that are no longer useful:**
  - Developed for functional languages
    - Essential in this programming paradigm. Why?
  - Getting more and more popular in imperative languages
    - Java, C#, Python

- Generally slower than manual reclamation, but it eliminates a very frequent programming error
Garbage Collection: Techniques

• When is an object no longer useful?
• There are several garbage collection techniques that answer this question in a different manner
  • Reference counting
  • Mark-and-sweep collection
Reference Counting

- Each object has an associated reference counter

- Keeps reference counter up to date, and recursively deallocates objects when the counter is zero
Reference Counting: Problems

- **Extra overhead** of storing and updating reference counts
- **Strong Typing required**
  - Impossible in language like C
  - It cannot be used for variant records
- It doesn’t work **with circular data structures**
  - This is a problem with this definition of useful object as an object with one or more references
Reference Counting: Circular Data Structures

[Diagram showing reference counting with stooges pointing to "larry" in heap with a reference count of 2, "moe" in heap with a reference count of 1, and "curly" in heap with a reference count of 1. The diagram also shows two instances of the heap with the same nodes and reference counts.]
Mark-and-Sweep Collection

• A better definition of *useless* is one that cannot be reached by following a chain of valid pointers starting from outside the heap.

• Mark-and-Sweep GC applies this definition
Mark-And-Sweep

Algorithm:

- Mark every block in the heap as useless
- Starting with all pointers outside the heap, recursively explore all linked data structures
- Add every block that remain marked to the free list.
- Run whenever free space is low
Mark-and-Sweep Collection: Problems

• Block must begin with an indication of its size

• A stack of depth proportional to the longest reference chain is required
  • AND! We are usually running low when running the GC
Mark-and-Sweep Collection: Problems

- Block must begin with an indication of its size
- A stack of depth proportional to the longest reference chain is required
- AND! We are usually running low when running the GC

If we use type descriptors, that indicate their size, then we don’t need to do this.
Mark-and-Sweep Collection: Problems

- Block must begin with an indication of its size
- A stack of depth proportional to the longest reference chain is required
  - AND! We are usually running out when running the GC

Possible to implement without a stack!
Mark-and-Sweep Collection: Pointer Reversal
Mark-and-Sweep Collection: Pointer Reversal
Mark-and-Sweep Collection: Pointer Reversal
Mark-and-Sweep Collection: Pointer Reversal
Mark-and-Sweep Collection: Pointer Reversal
Mark-and-Sweep Collection: Pointer Reversal
Store-and-Copy

• Use to reduce external fragmentation

• S-C divides the available space in half and allocates everything in that half until its full

• When that happens, copy each useful block to the other half, clean up the remaining block, and switch the roles of each half.