Simulated Pointers
Limitations Of C++ Pointers

• May be used for internal data structures only.
• Data structure backup requires serialization and deserialization.
• No arithmetic.
Simulated-Pointer Memory Layout

Data structure memory is an array, and each array position has an element field (type $T$) and a next field (type $\text{int}$).
Node Representation

template <class T>
class simulatedNode
{
    public:
        // constructors defined here
    protected:
        T element;
        int next;
}
How It All Looks

next element

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>8</th>
<th>11</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>a</td>
<td>e</td>
<td>d</td>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

firstNode = 4
Still Drawn The Same

firstNode

a — b — c — d — e
Memory Management

Linked system (C++ or simulated pointer) requires:

- a way to keep track of the memory that is not in use (i.e., a storage pool)
- way to allocate a node
  C++ has the method `new`
- way to free a node that is no longer in use
  C++ has the method `delete`
Garbage Collection

The system determines which nodes/memory are not in use and returns these nodes (this memory) to the pool of free storage.

This is done in two or three steps:

Mark nodes that are in use.
Compact free space (optional).
Move free nodes to storage pool.
Marking

Unmark all nodes (set all mark bits to false).
Start at each program variable that contains a reference, follow all pointers, mark nodes that are reached.
Marking

Start at `firstNode` and mark all nodes reachable from `firstNode`. 🍀

Repeat for all pointer variables.
Compaction

Move all marked nodes (i.e., nodes in use) to one end of memory, updating all pointers as necessary.
Scan memory for unmarked nodes (if no compaction done), otherwise put single free block (unless no free memory) into pool.
Advantages Of Garbage Collection

- Programmer doesn’t have to worry about freeing nodes as they become free.
- However, for garbage collection to be effective, we must set reference variables to **null** when the object being referenced is no longer needed.
Disadvantage Of Garbage Collection

• Garbage collection time is linear in memory size (not in amount of free memory).
Alternative To Garbage Collection

Provide a method to free/deallocate a node.

e.g., delete method of C++

Now free nodes are always in storage pool.
Advantage Of Alternative

- Time to free nodes is proportional to number of nodes being freed and not to total memory size.
Disadvantages Of Alternative

• User must write methods to free data structure nodes.
• Time is spent freeing nodes that may not be reused.
• Application run time does not improve with increase in memory size.
Storage Pool Organization When All Nodes Have Same Size

- Maintain a chain of free nodes
- Allocate from front of chain
- Add node that is freed to chain front
Simulated-Pointer Memory Management

template <class T>
class simulatedSpace
{
    public:
    // constructor and other methods
    // defined here
    protected:
    int firstNode;
    simulatedNode<T> *node;
}
Constructor

template <class T>
simulatedSpace (int numberOfNodes)
{

    node = new simulatedNode<T> [numberOfNodes];
    // create nodes and link into a chain
    for (int i = 0; i < numberOfNodes - 1; i++)
        node[i].next = i + 1;
    // last node of array and chain
    node[numberOfNodes - 1].next = -1;
    // first node of chain of free nodes
    firstNode = 0;
}

Allocate A Node

```cpp
template <class T>
int allocateNode(T& element, int next) {
    // Allocate a free node and set its fields.
    if (firstNode == -1) {
        // double number of nodes, code omitted
    }
    int i = firstNode;  // allocate first node
    firstNode = node[i].next;
    node[i].element = element;
    node[i].next = next;
    return i;
}
```
template <class T>
void deallocateNode(int i)
{// Free node i.
    // make i first node on free space list
    node[i].next = firstNode;
    firstNode = i;
}