Summary of Dynamic Resizing Pros and Cons

dynamically resized array
- start with small constant
- multiply size by a constant as necessary

pros
- easy to implement
- array uses contiguous memory

cons (quantified for 2× multiplier)
- copying cost (≤ 2N for N players)
- waste space (~38%)
Where Can All the Pointers Go?

With **dynamic resizing**, we **used one player pointer** in the global data area:

```c
static player_t* player_list = NULL;
```

Where can we put more pointers?

We can use:
- a dynamically resized array of pointers.
- But … have we really solved the problem in that case? (An array of pointers does reducing copying and waste space.)

Solution: Add a Pointer to the Player Struct!

What if we add a `player_t*` to the player struct?

Mark the End of the List by Pointing to Nothing

What about the last player’s pointer? Set it to **NULL**.
Singly-Linked Lists are Common for Unordered Groups

The data structure shown
- is called a **singly-linked list**
- (or, frequently, just a **linked list**).
- Singly-linked lists are usually used when order is not important.

**How do we insert into a linked list?**
- Specifically, where should we insert a new element: at the start, or the end?

Insert at the start: it’s faster.

Inserting into a Singly-Linked List Requires Two Changes

Make two changes. In what order?

Correct Ordering of Changes is Important

First, change the **next** field of the new player. Otherwise, the old list is lost!

```c
new_player->next = player_list;
player_list = new_player;
```

That’s all.

How Can We Remove Player `p` from the List?

Need to change pointer (link) marked in blue.
Singly-Linked List Deletion is Linear in Size of List

Deletion is slower:
- to delete player p
- from a list that starts at player_list,
- we must walk over the list to find p,
- then change pointer to p to p->next.

In general,
- with N things in the list,
- we examine on average N/2.

Modify Player Structure to Use Dynamic Allocation

Before writing player_delete,
- let's modify our player structure
- to use dynamic allocation
- for the name* field.

*We treated the password field as a normal string before, but technically it should be hashed or encrypted to a fixed-length string.

Review: Example Player Structure

```c
struct player_t {
    char* name;  // points to a dynamically allocated block of memory.
    char password[20];
    int32_t age;
    int32_t num_games;
    int32_t score_dist[16];
    struct game_t* game;
};
```

Modify player_init to Dynamically Allocate the Name

Then, in player_init, we can write...

```c
p->name = malloc (strlen (n) + 1);
if (NULL == p->name) { return 0; }
strcpy (p->name, n);
```

or

```c
p->name = strdup (n);
if (NULL == p->name) { return 0; }
```

(recall that n is the new player's name).
First, Remove Player to Be Deleted from the List

Questions for you:

To delete “Y,” what needs to change?
The next field of player “Z.”

Free All Dynamically Allocated Data for the Player

Questions for you:

What needs to be freed to delete “Y?”
Both the player structure and the name.

Do Not Use Dynamic Data After Freeing It

Questions for you:

In what order?
First the name, then the player structure.

Ready to Write a Function to Delete a Player

Now we can write `player_delete`.

The function signature is:

```c
int32_t player_delete (player_t* p);
```

- `p` points to the player structure
to remove from the list and free
- function returns 1 on success, or 0 on failure
Use a `player_t**` to Find the Link to Change

```c
player_t** find;
for (find = &player_list;
    p != *find;
    find = &(*find)->next) {
    if (NULL == *find) {
        return 0;
    }
}
```

Using a `player_t**` makes the code simpler.

Initialize `find` to Point to the Pointer to the Head

```c
player_t** find;
for (find = &player_list;
    p != *find;
    find = &(*find)->next) {
    if (NULL == *find) {
        return 0;
    }
}
```

Point `find` first to the pointer to the head of the list.

Advance Until `find` Points to Pointer to Player to Delete

```c
player_t** find;
for (find = &player_list;
    p != *find;
    find = &(*find)->next) {
    if (NULL == *find) {
        return 0;
    }
}
```

Once `p == *find`, we have found the link to change.

Move `find` from `next` Field to `next` Field

```c
player_t** find;
for (find = &player_list;
    p != *find;
    find = &(*find)->next) {
    if (NULL == *find) {
        return 0;
    }
}
```

Advance by pointing `find` to the `next` field of the structure to which the pointer `find` points to points.
For Safety, Check for End of List in Loop Body

```c
player_t** find;
for (find = &player_list;
p != *find;
    find = &(*find)->next) {
    if (NULL == *find) {
        return 0;
    }
}
```

If we reach the end of the list, p is not in the list, so fail.

Remove the Player, Free the Blocks, and Return Success

```c
*find = p->next;
free (p->name);
free (p);
return 1;
```

Remember that find points to the pointer to be changed.

Free the name, then the player.

Return success.

Examine How player_delete Works in Detail

Let's do a detailed example of player_delete execution on a linked list of three players with variables shown in LC-3 memory.

Let's first identify where each variable resides:
- in the global data area,
- in the heap, or
- in the stack.

Dynamically Allocated Data Reside in the ...

The linked list is shown below (head on left).

Where are these data (global data, heap, or stack)?
**File-Scope Variables Reside in the ...**

The file-scope variable `player_list` points to the head of the list. **Where is `player_list` stored?**

- `player_list` points to the head of the list. 

- `Gao` points to `Charlie` points to `Baddie` points to `next`.

**Local Variables Reside in the ...**

And where is local variable `find`? 

- `find` points to `player_list`.

**Parameter `p` is Close to Local Variable `find`**

- Parameter `p` is close to local variable `find`.

**Start the Function by Initializing `find`**

Here's the loop again. 

```c
for (find = &player_list; 
p != *find; 
    find = &(*find)->next) {
    if (NULL == *find) {
        return 0; 
    } 
} 
```

Start by initializing `find`. 

© 2018 Steven S. Lumetta. All rights reserved.
Initialize `find` to `&player_list`

```
for (find = &player_list; 
p != *find; 
find = &(*find)->next) {
if (NULL == *find) {
    return 0;
}
}
```

Continue Executing the Loop

```
What happens next?
for (find = &player_list; 
p != *find; 
find = &(*find)->next) {
if (NULL == *find) {
    return 0;
}
}
```

```
Execute the loop body.
```
Is *find Equal to NULL?

Continue Executing the Loop

What happens next?
for (find = &player_list;
p != *find;
find = &(*find)->next) {
    if (NULL == *find) {
        return 0;
    }
}
Execute the loop update.

Where is (*find)->next?

Compiler Can Calculate Offsets for Each Field

struct player_t {
+*0  char* name;
+*01 char password[20];
+*15 int32_t age;
+*17 int32_t num_games;
+*19 int32_t score_dist[16];
+*39 struct game_t* game;
+*x3A player_t* next;
};
Continue Executing the Loop

And then...

```c
for (find = &player_list;
p != *find;
find = &(*find)->next) {
if (NULL == *find) {
    return 0;
}
}
```

Back to the loop test.

Continue Executing the Loop

After the loop test and the loop body...

```c
for (find = &player_list;
p != *find;
find = &(*find)->next) {
if (NULL == *find) {
    return 0;
}
}
```

Execute the loop update.
Where is (*find)->next?

Set find to &(*find)->next

Continue Executing the Loop

And then ...

for (find = &player_list; 
p != *find; 
find = &(*find)->next) {
  if (NULL == *find) {
    return 0;
  }
}

Back to the loop test.

Is *find Equal to p? Yes! Loop Test Fails...
Overwrite *find with \texttt{p->next}

Here's the code after the loop.

\begin{verbatim}
*find = p->next;
free (p->name);
free (p);
return 1;
\end{verbatim}

Notice that we overwrite *find.

Set the Bits at *\texttt{find} to \texttt{p->next}

```
*find = p->next;
free (p->name);
free (p);
return 1;
```

Finish the Rest of the Function

What's next?

```
*find = p->next;
free (p->name);
free (p);
return 1;
```

Free the name, then player \texttt{p}.

“Baddie” is No Longer in the List!

```
*find = p->next;
free (p->name);
free (p);
return 1;
```
Free the Two Blocks of Dynamically Allocated Data

The Function is Done

What's next?

*find = p->next;
free (p->name);
free (p);
return 1;
}

Return success!

The List After the Function has Returned