University of Illinois at Urbana-Champaign
Dept. of Electrical and Computer Engineering

ECE 220: Computer Systems & Programming

Fully Dynamic Allocation

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%)

Player Deletion with Dynamic Resizing

What about deletion?

If order doesn't matter,
- copy last element over deleted element,
- then reduce count
- (requires **constant time**).

If order matters, deletion can be expensive.

Allocating Individual Players Requires a Pointer for Each

Can we **use dynamic allocation**
- **to allocate one thing** (a player) at a time
- instead of resizing an array?

Yes, but first, we **need to solve a problem**:
- Every call to `malloc` returns a pointer.
- These pointers have no predictable relationship to one another.
- So we **need to store a pointer to each player**.

Where Can All the Pointers Go?

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

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

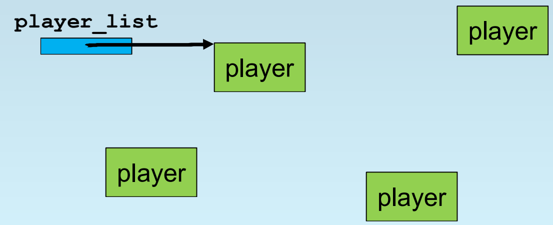
Where can we put more pointers?

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.)

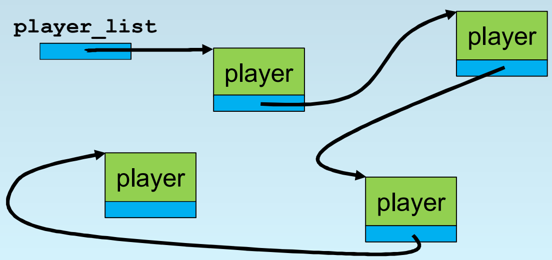
Where Can We Put More Pointers?

Can we do something else?



Solution: Add a Pointer to the Player Struct!

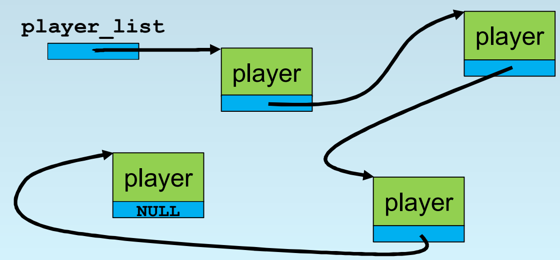
What if we add a `player_t*` to the player struct?



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

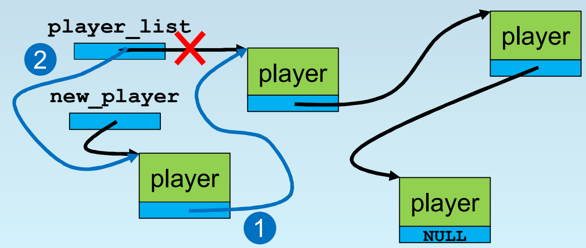
The data structure shown
- is called a **singly-linked list**
- (or, frequently, just a **linked list**).
- Singly-linked lists are usually used when order is not important.

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.

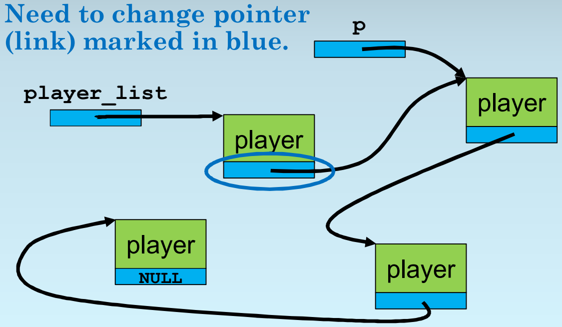
Otherwise, the old list is lost!

```
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

```
struct player_t {
    char name[32]; char* name;
    char password[20];
    int32_t age;
    int32_t num_games;
    int32_t score_dist[16];
    struct game_t* game;
    player_t* next;
};
```

`name` points to a dynamically allocated block of memory.

`next` is used for the linked list.

Modify `player_init` to Dynamically Allocate the Name

Then, in `player_init`, we can write…

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

or

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

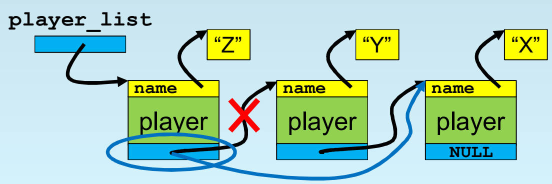
(recall that `n` is the new player's name).

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."

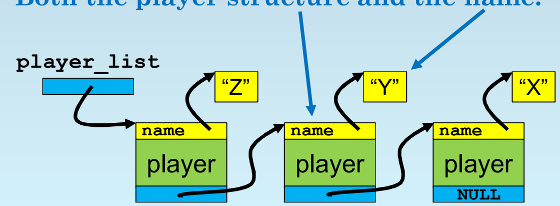


Free All Dynamically Allocated Data for the Player

Questions for you:

What needs to be freed to delete "Y?"

Both the player structure and the name.

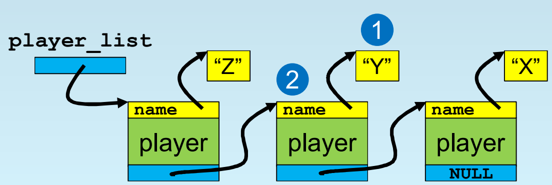


Do Not Use Dynamic Data After Freeing It

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:

```
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

```
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

```
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

```
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

```
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

```
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

Remember that **find** points to the pointer to be changed.

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

Free the name, then the player.

Return success.

Examine How **player_delete** Works in Detail

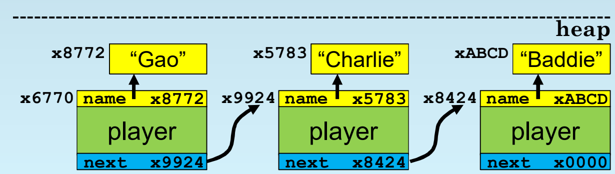
Let's do a **detailed example**
- **of player_delete execution**
- on a linked list of three players
- with variables shown in LC-3 memory.

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).

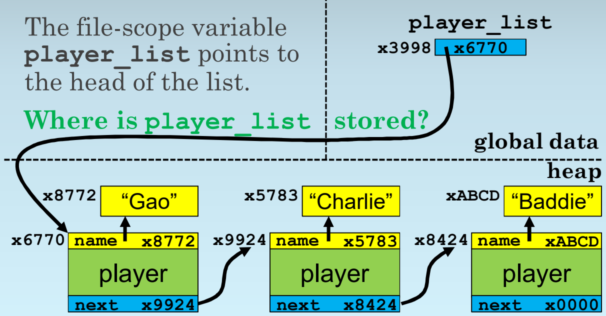
Where are these data (global data, heap, or stack)?



File-Scope Variables Reside in the …

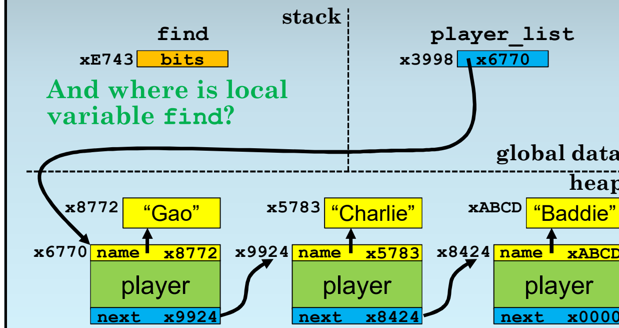
The file-scope variable **`player_list`** points to the head of the list.

Where is `player_list` stored?

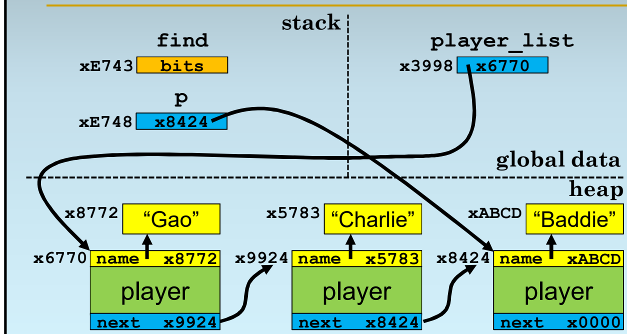


Local Variables Reside in the …

And where is local variable `find`?



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



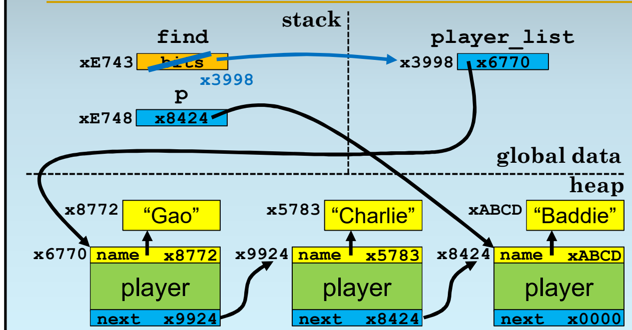
Start the Function by Initializing `find`

Here's the loop again.

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

Start by initializing **`find`**.

Initialize **find** to **&player_list**



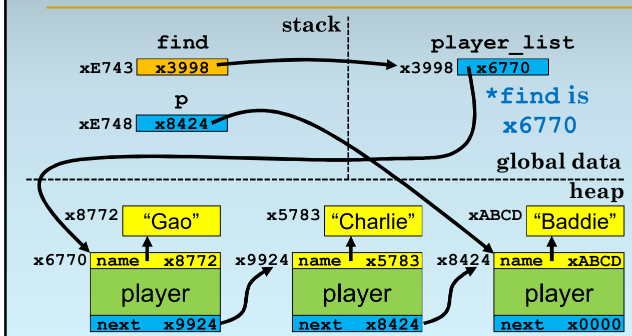
Continue Executing the Loop

What happens next?

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

Execute the loop test.

Is ***find** Equal to **p**?



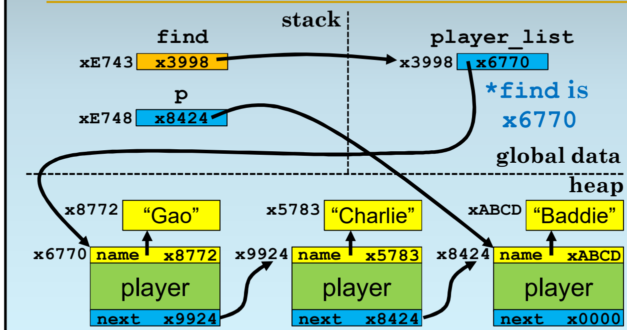
Continue Executing the Loop

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?



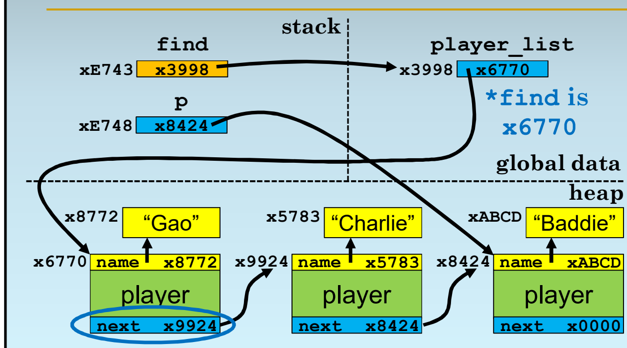
Continue Executing the Loop

What happens next?

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

Execute the loop update.

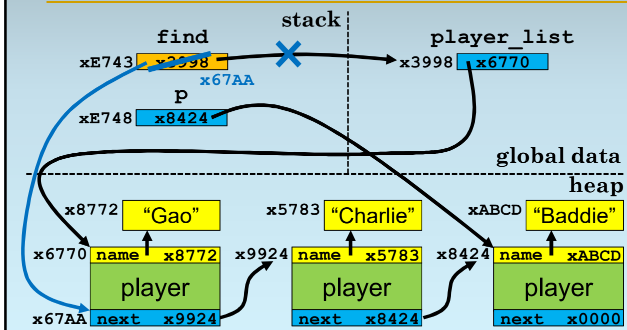
Where is (*find)->next?



Compiler Can Calculate Offsets for Each Field

```
     struct player_t {
+x00 char* name;
+x01 char password[20];
+x15 int32_t age;
+x17 int32_t num_games;
+x19 int32_t score_dist[16];
+x39 struct game_t* game;
+x3A player_t* next;
     };
```


Set `find` to `&(*find)->next`



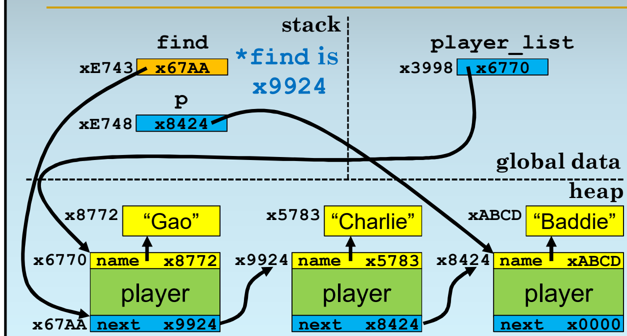
Continue Executing the Loop

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`? What About `NULL`?



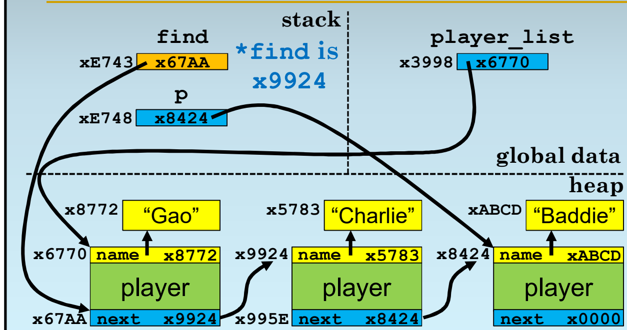
Continue Executing the Loop

After the loop test and the loop body…

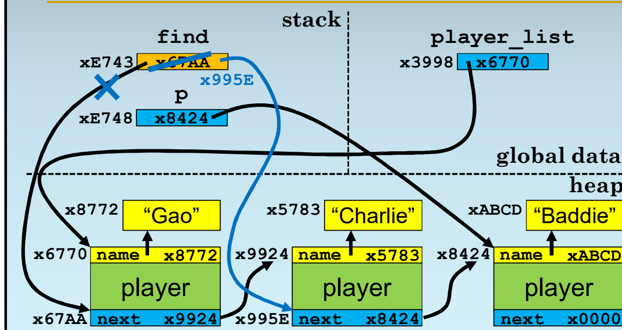
```
for (find = &player_list;
     p != *find;
     find = &(*find)->next) {
    if (NULL == *find) {
        return 0;
    }
}
```

Execute the loop update.

Where is `(*find)->next`?



Set `find` to `&(*find)->next`



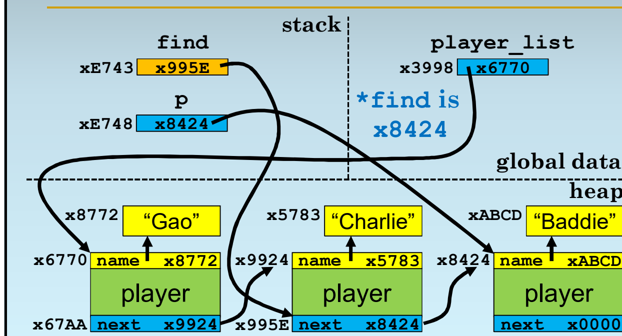
Continue Executing the Loop

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…



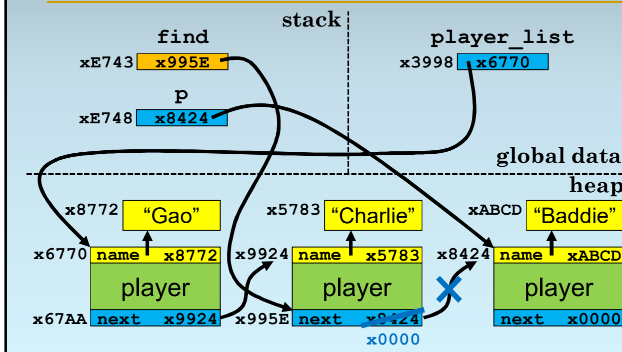
Overwrite *find with p->next

Here's the code after the loop.

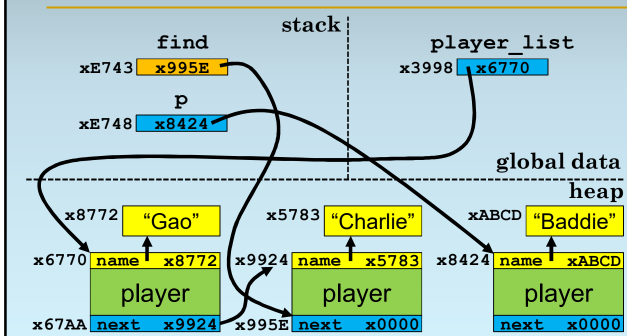
```
    *find = p->next;
    free (p->name);
    free (p);
    return 1;
}
```

Notice that we overwrite *find.

Set the Bits at *find to p->next



"Baddie" is No Longer in the List!



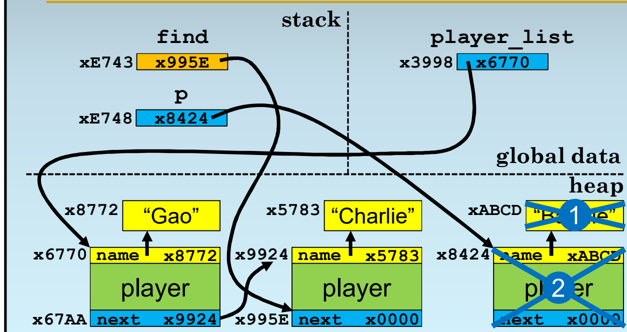
Finish the Rest of the Function

What's next?

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

Free the name, then player **p**.

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