

# Linked Lists

## Part Three

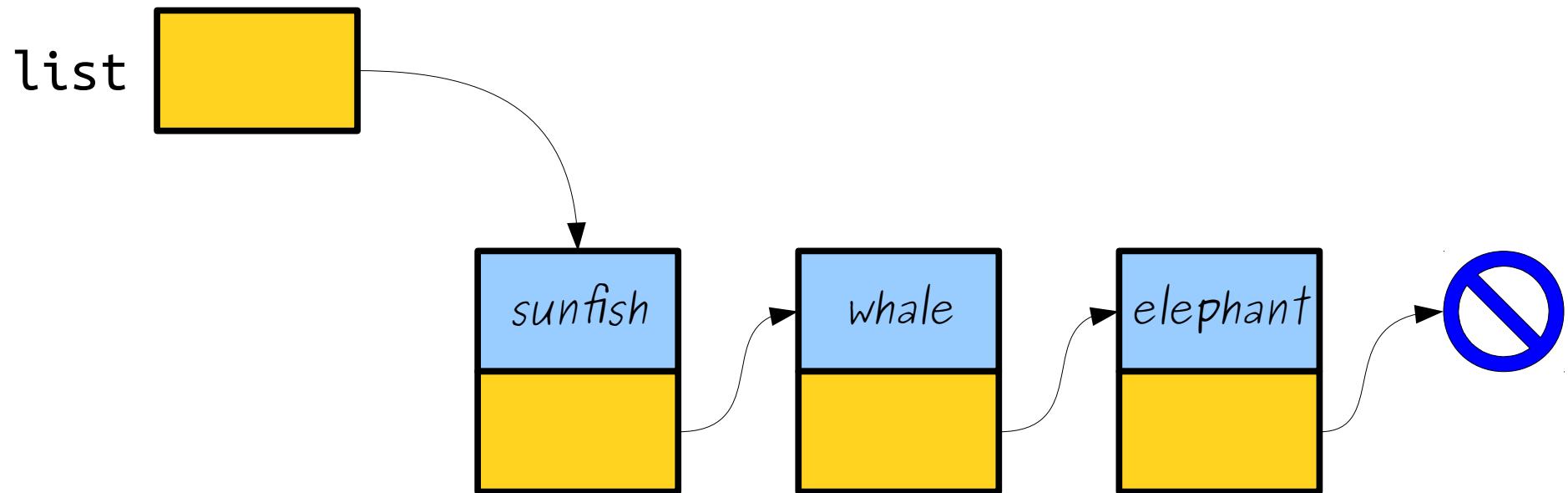
# Outline for Today

- ***Pointers by Reference***
  - Changing where you're looking.
- ***Tail Pointers***
  - Speeding up list operations.
- ***Doubly-Linked Lists***
  - A preview of things to come.

# Pointers and References

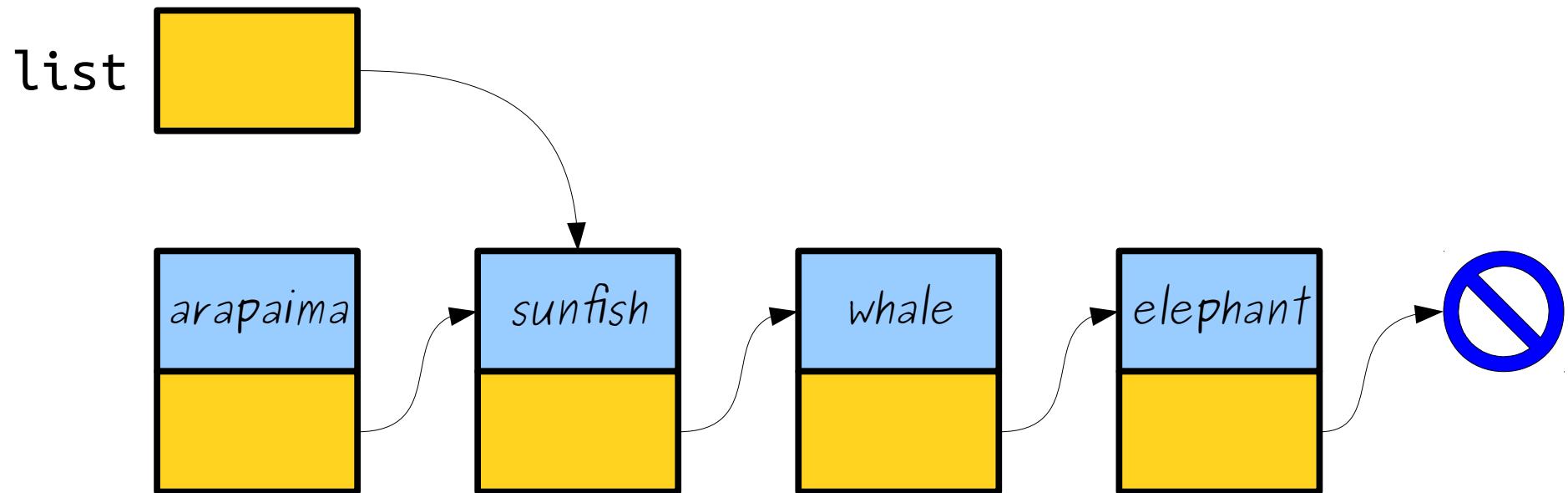
# Prepending an Element

- Suppose that we want to write a function that will add an element to the front of a linked list.
- What might this function look like?



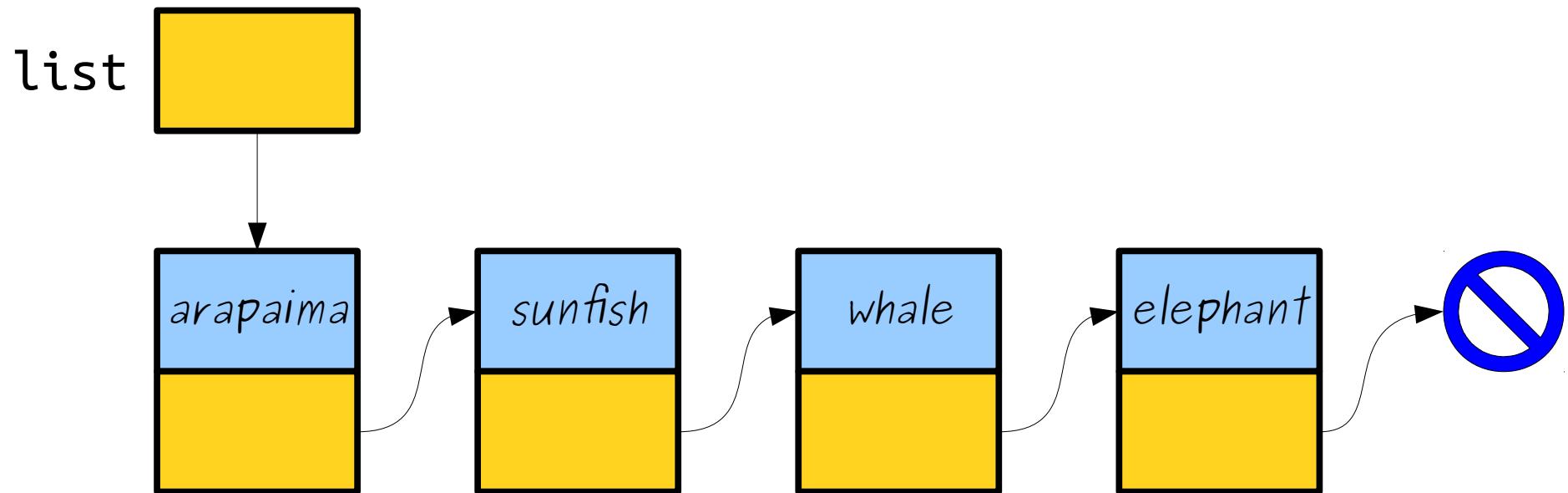
# Prepending an Element

- Suppose that we want to write a function that will add an element to the front of a linked list.
- What might this function look like?



# Prepending an Element

- Suppose that we want to write a function that will add an element to the front of a linked list.
- What might this function look like?



What went wrong?

```
int main() {
    Cell* list = nullptr;
    prependTo(list, "Sartre");
    prependTo(list, "Camus");
    prependTo(list, "Nietzsche");

    return 0;
}
```

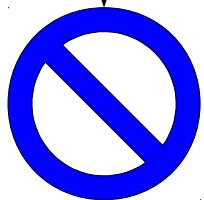
```
int main() {
    Cell* list = nullptr;
    prependTo(list, "Sartre");
    prependTo(list, "Camus");
    prependTo(list, "Nietzsche");

    return 0;
}
```

```
int main() {
    Cell* list = nullptr;
    prependTo(list, "Sartre");
    prependTo(list, "Camus");
    prependTo(list, "Nietzsche");

    return 0;
}
```

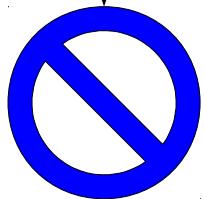
list



```
int main() {
    Cell* list = nullptr;
    prependTo(list, "Sartre");
    prependTo(list, "Camus");
    prependTo(list, "Nietzsche");

    return 0;
}
```

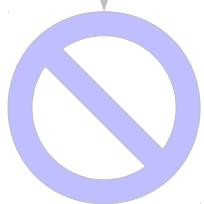
list



```
int main() {
    cell* list = nullptr;
    prependTo(list, "Sartre");
    prependTo(list, "Camus");
    prependTo(list, "Nietzsche");

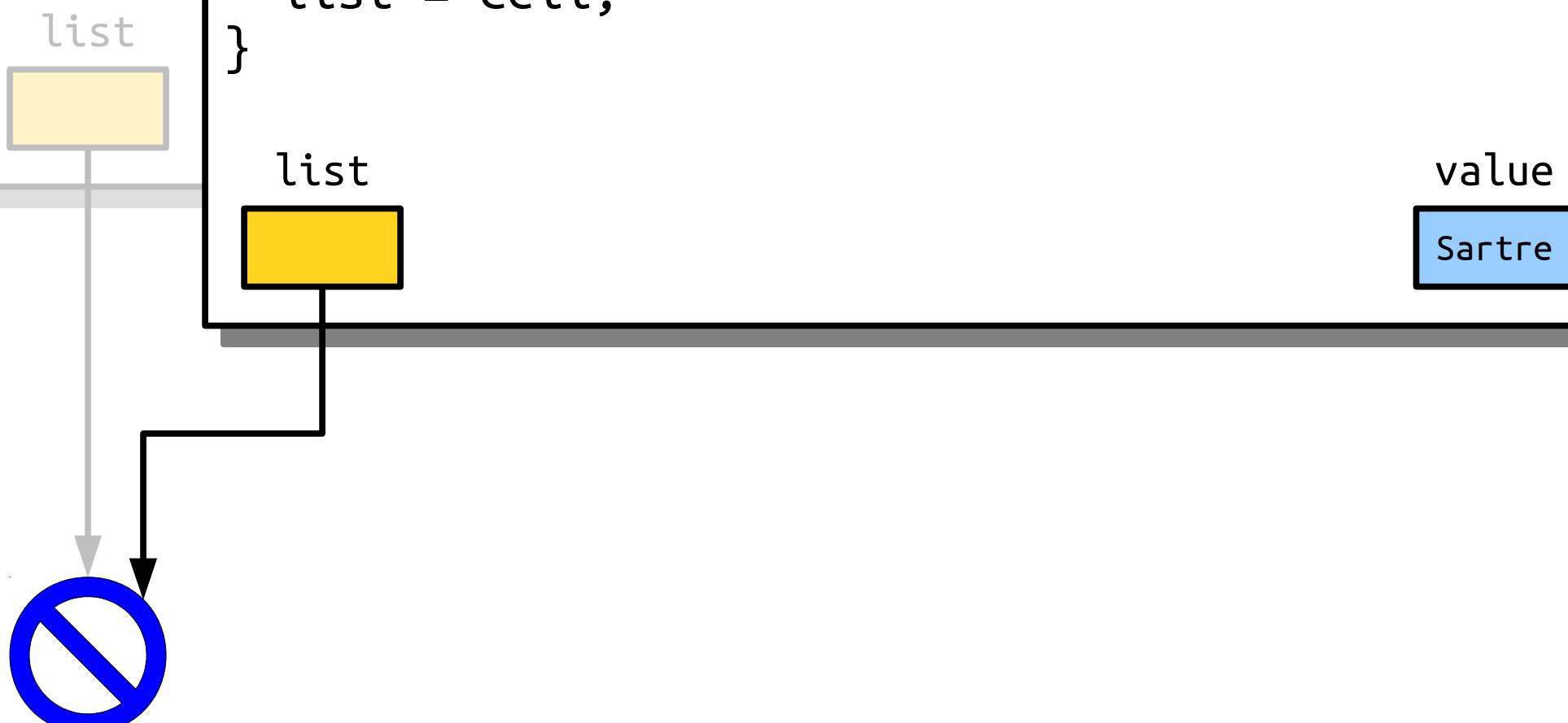
    return 0;
}
```

list



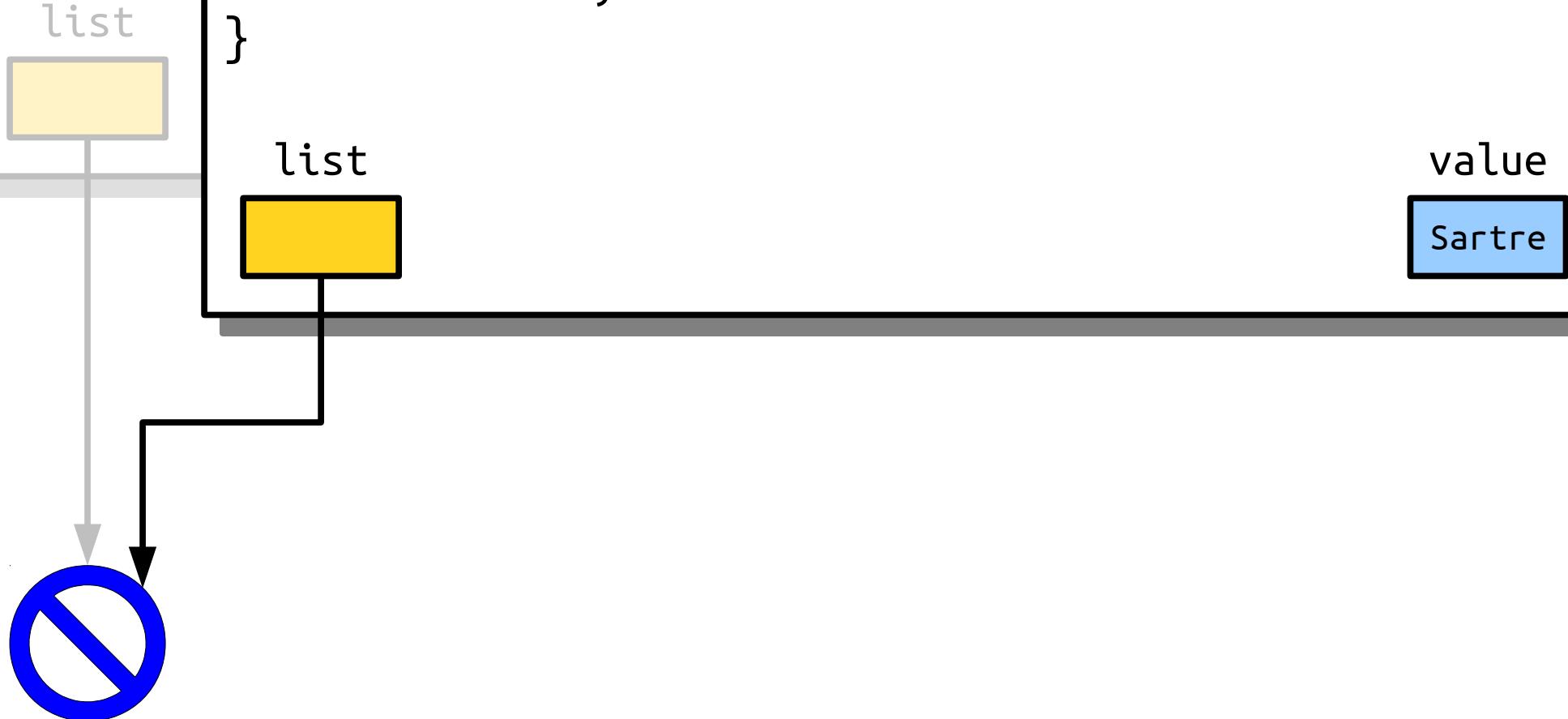
```
int main() {  
    Cell* list = nullptr;  
    prep  
    prep  
    prep  
    return 0;  
}
```

```
void prependTo(Cell* list, const string& value) {  
    Cell* cell = new Cell;  
    cell->value = value;  
    cell->next = list;  
    list = cell;  
}
```



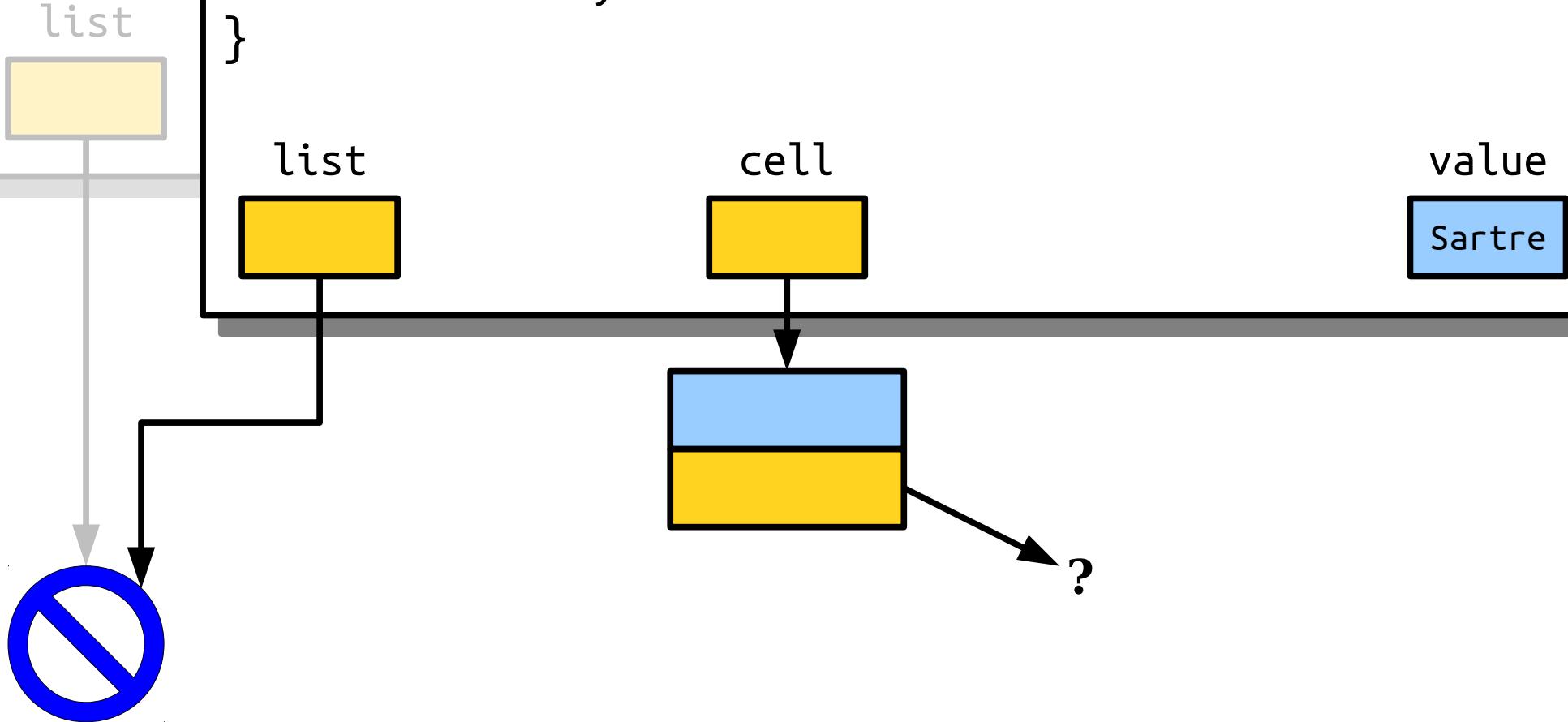
```
int main() {
    cell* list = nullptr;
    prep();
    prep();
    prep();
    return 0;
}

void prependTo(Cell* list, const string& value) {
    Cell* cell = new Cell;
    cell->value = value;
    cell->next = list;
    list = cell;
}
```



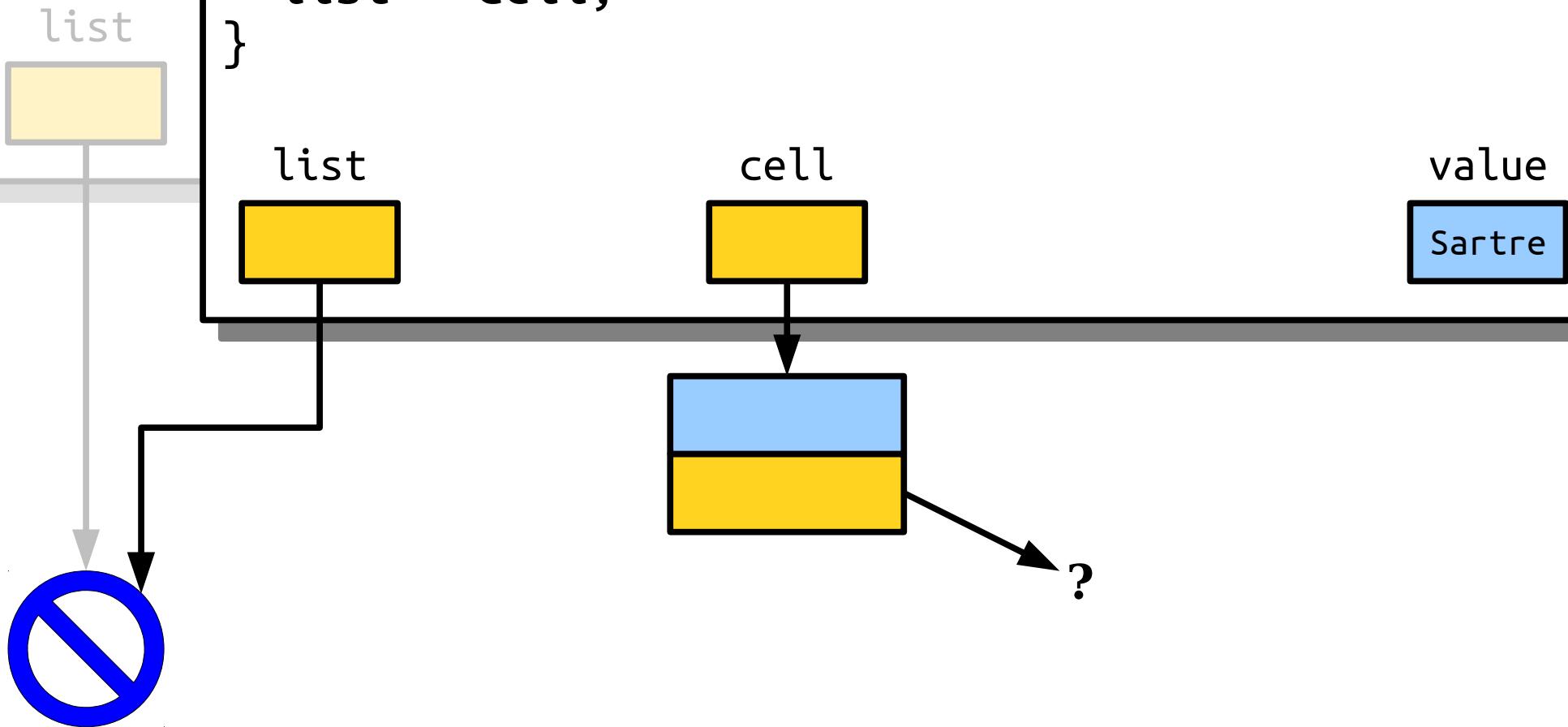
```
int main() {  
    cell* list = nullptr;  
    prep  
    prep  
    prep  
    return 0;  
}
```

```
void prependTo(Cell* list, const string& value) {  
    Cell* cell = new Cell;  
    cell->value = value;  
    cell->next = list;  
    list = cell;  
}
```



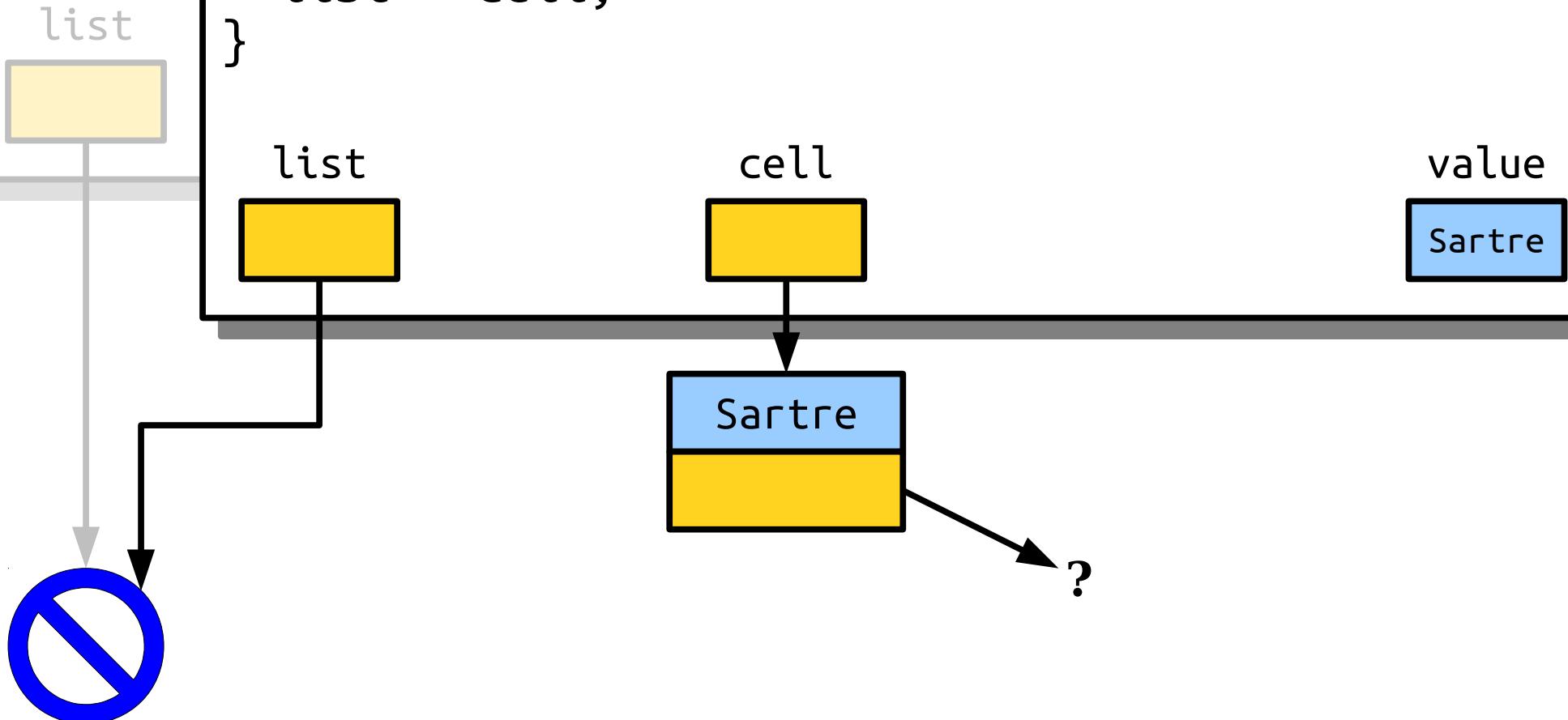
```
int main() {  
    Cell* list = nullptr;  
    prep  
    prep  
    prep  
    return 0;  
}
```

```
void prependTo(Cell* list, const string& value) {  
    Cell* cell = new Cell;  
    cell->value = value;  
    cell->next = list;  
    list = cell;  
}
```



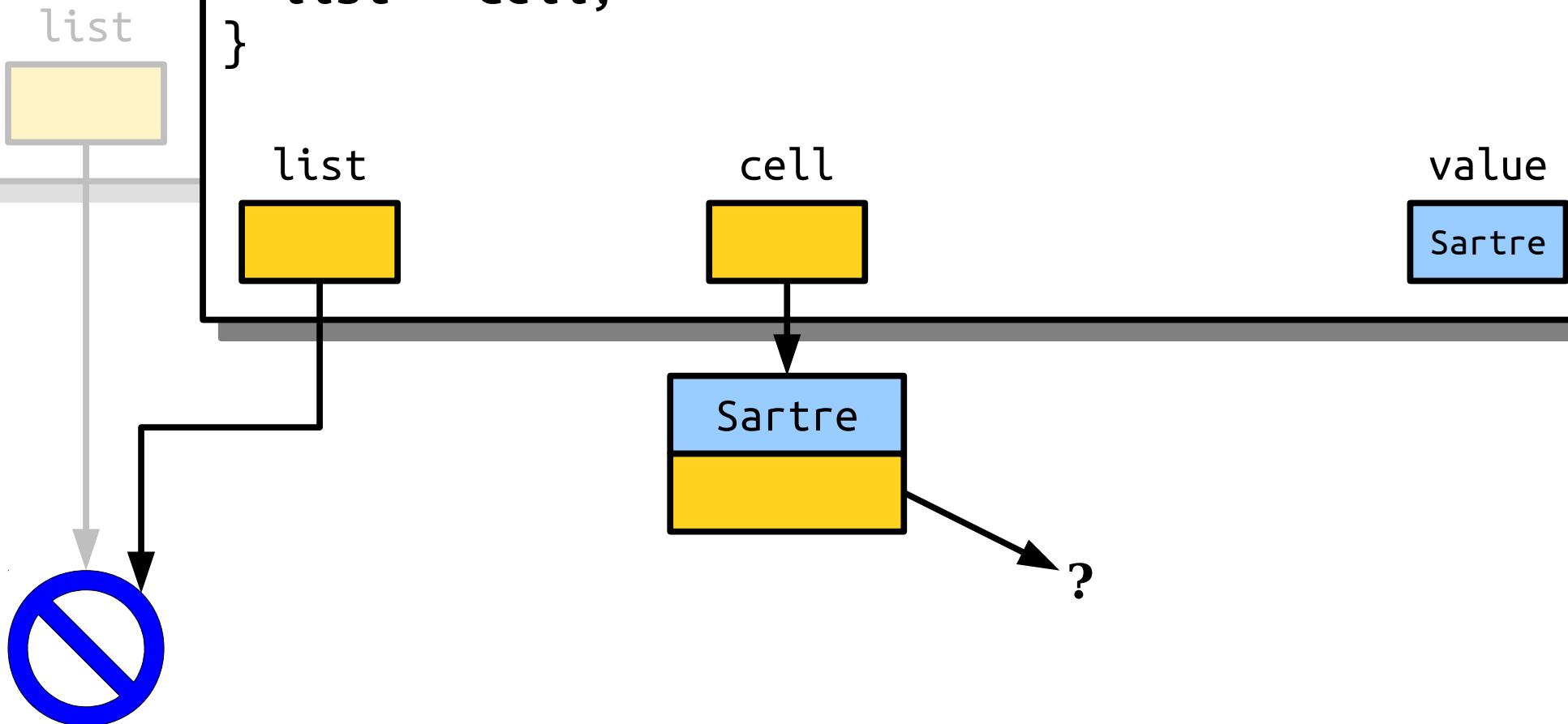
```
int main() {
    Cell* list = nullptr;
    prep
    prep
    prep
    return 0;
}
```

```
void prependTo(Cell* list, const string& value) {
    Cell* cell = new Cell;
    cell->value = value;
    cell->next = list;
    list = cell;
}
```



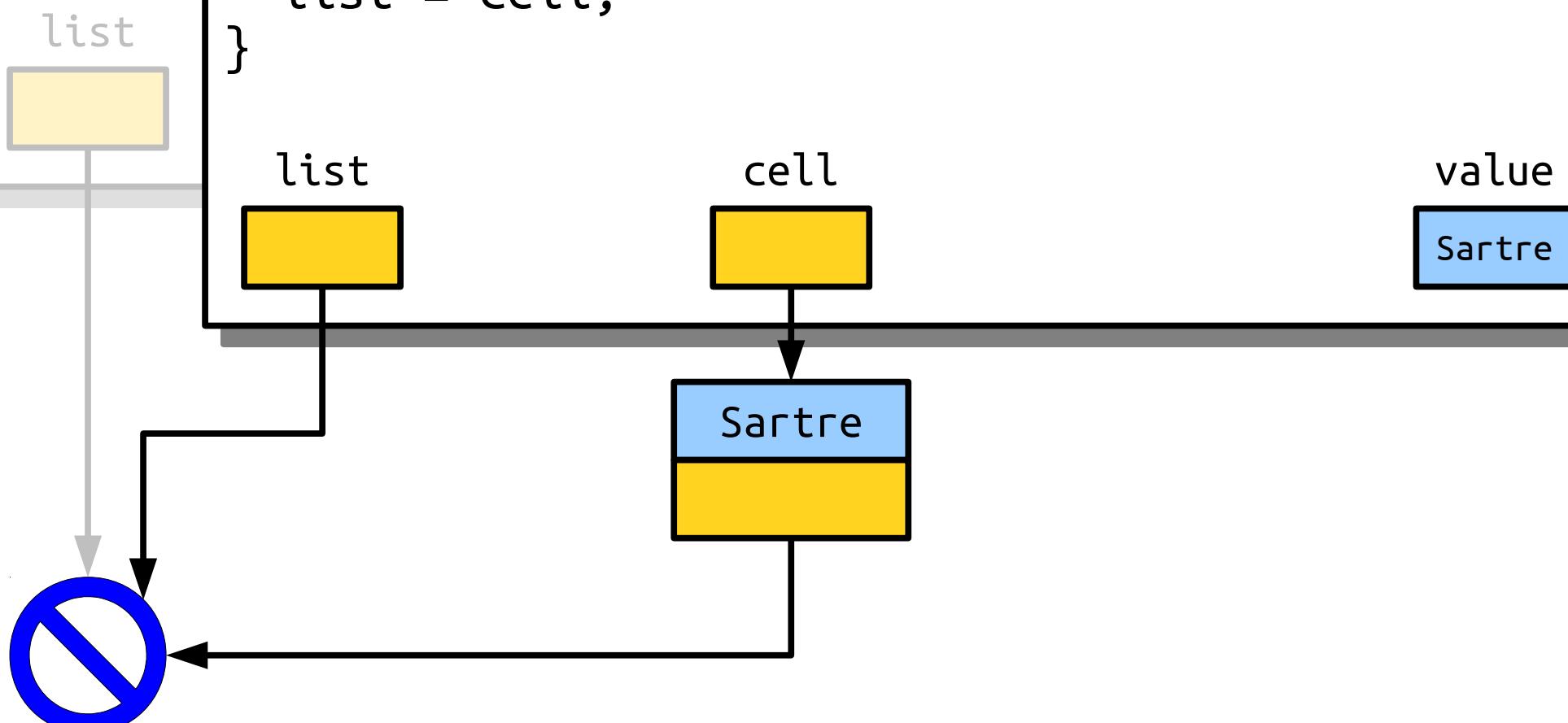
```
int main() {  
    cell* list = nullptr;  
    prep  
    prep  
    prep  
    return 0;  
}
```

```
void prependTo(Cell* list, const string& value) {  
    Cell* cell = new Cell;  
    cell->value = value;  
    cell->next = list;  
    list = cell;  
}
```



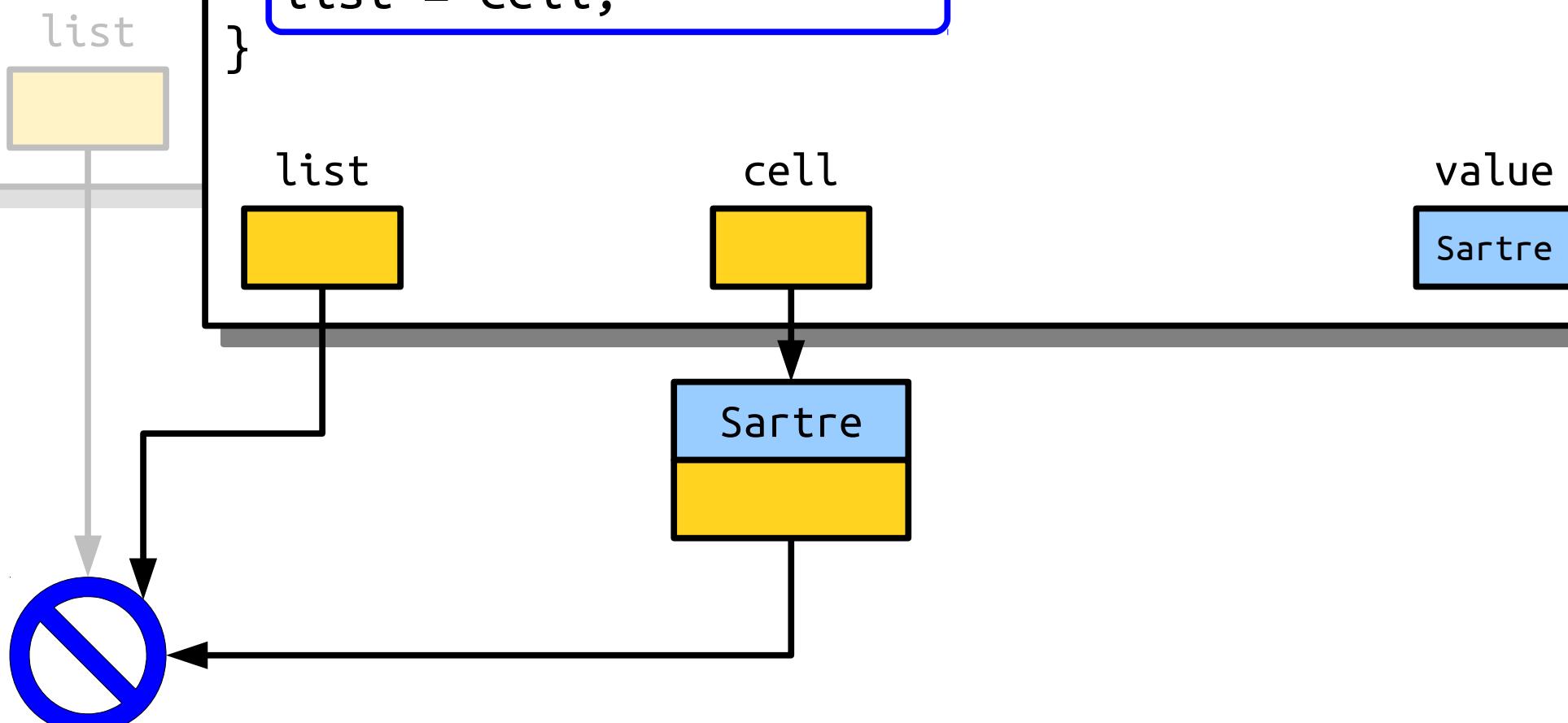
```
int main() {  
    cell* list = nullptr;  
    prep  
    prep  
    prep  
    return 0;  
}
```

```
void prependTo(Cell* list, const string& value) {  
    Cell* cell = new Cell;  
    cell->value = value;  
    cell->next = list;  
    list = cell;  
}
```



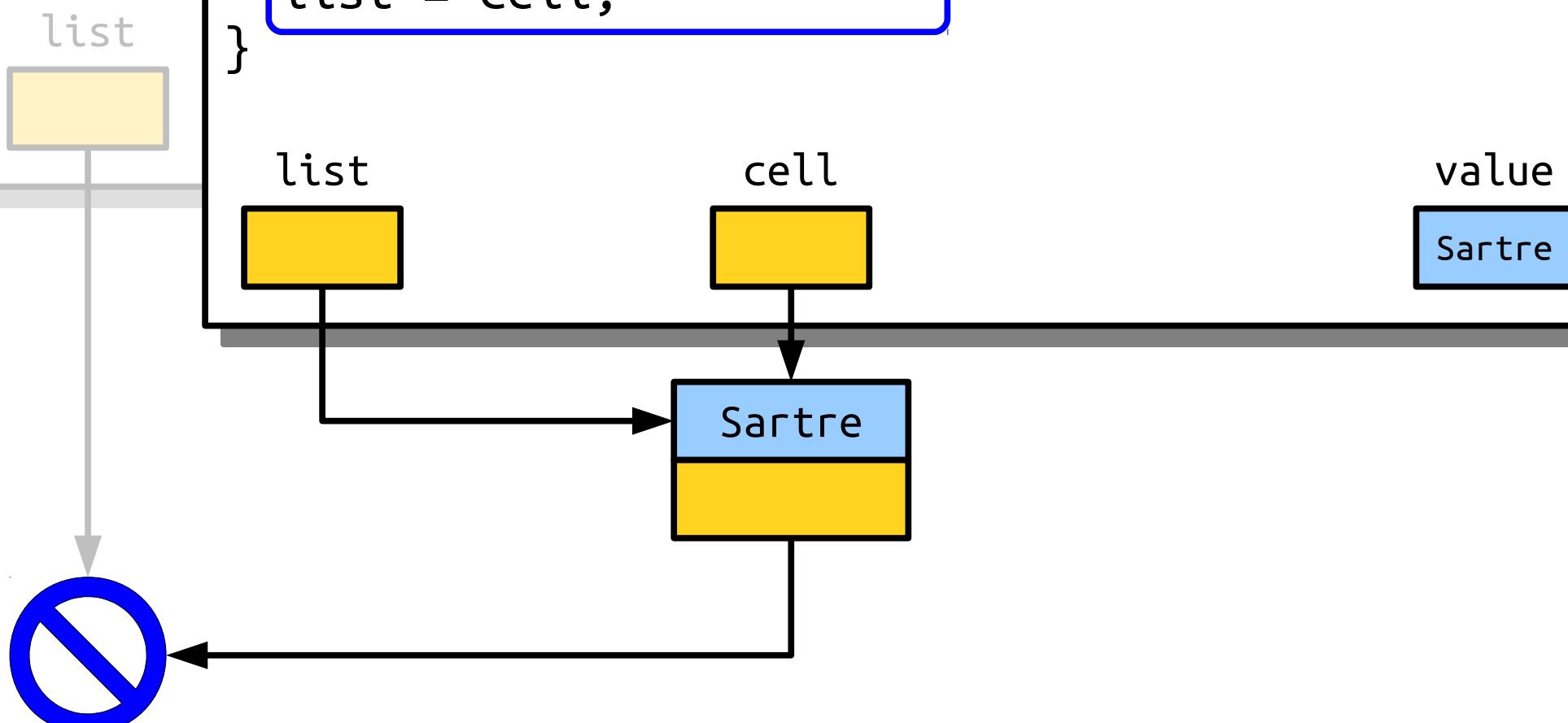
```
int main() {  
    cell* list = nullptr;  
    prep  
    prep  
    prep  
    return 0;  
}
```

```
void prependTo(Cell* list, const string& value) {  
    Cell* cell = new Cell;  
    cell->value = value;  
    cell->next = list;  
    list = cell;  
}
```



```
int main() {  
    cell* list = nullptr;  
    prep  
    prep  
    prep  
    return 0;  
}
```

```
void prependTo(Cell* list, const string& value) {  
    Cell* cell = new Cell;  
    cell->value = value;  
    cell->next = list;  
    list = cell;  
}
```



```
int main() {
    cell* list = nullptr;
    prependTo(list, "Sartre");
    prependTo(list, "Camus");
    prependTo(list, "Nietzsche");

    return 0;
}
```

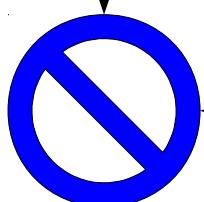
list



```
int main() {
    Cell* list = nullptr;
    prependTo(list, "Sartre");
    prependTo(list, "Camus");
    prependTo(list, "Nietzsche");

    return 0;
}
```

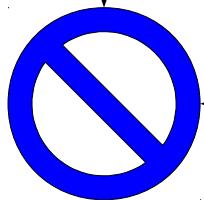
list



```
int main() {
    Cell* list = nullptr;
    prependTo(list, "Sartre");
    prependTo(list, "Camus");
    prependTo(list, "Nietzsche");

    return 0;
}
```

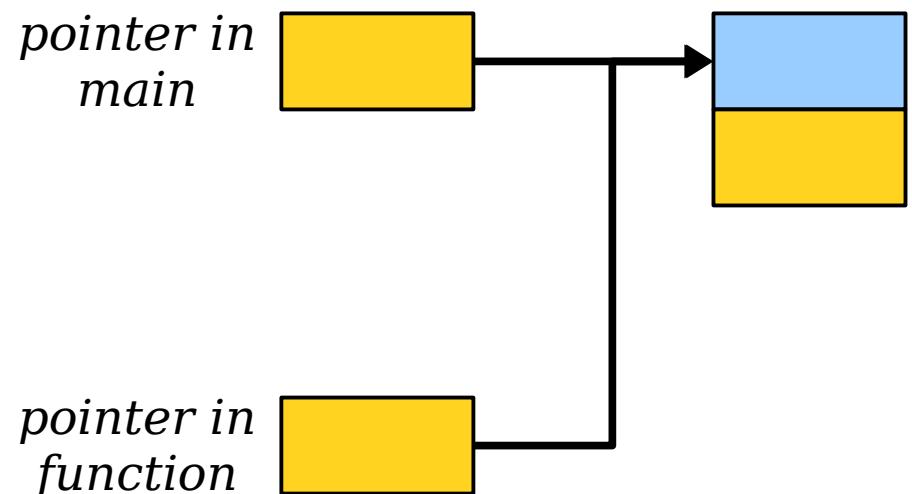
list



*Hell is other pointers*

# Pointers By Value

- Unless specified otherwise, function arguments in C++ are passed by value.
- This includes pointers!
- A function that takes a pointer as an argument gets a copy of the pointer.
- We can change where the *copy* points, but not where the original pointer points.



# Pointers by Reference

- To resolve this problem, we can pass the linked list pointer by reference.
- Our new function:

```
void prependTo(Cell*& list, const string& value) {  
    Cell* cell = new Cell;  
    cell->value = value;  
    cell->next = list;  
    list = cell;  
}
```

# Pointers by Reference

- To resolve this problem, we can pass the linked list pointer by reference.
- Our new function:

```
void prependTo(Cell*& list, const string& value) {  
    Cell* cell = new Cell;  
    cell->value = value;  
    cell->next = list;  
    list = cell;  
}
```

# Pointers by Reference

- To resolve this problem, we can pass the linked list pointer by reference.
- Our new function:

```
void prependTo(Cell*& list, const string& value) {  
    Cell* cell = new Cell;  
    cell->value = value;  
    cell->next = list;  
    list = cell;  
}
```

This is a reference to a pointer to a Cell. If we change where list points in this function, the changes will stick!

```
int main() {
    Cell* list = nullptr;
    prependTo(list, "Descartes");
    prependTo(list, "Kant");
    prependTo(list, "Bentham");

    return 0;
}
```

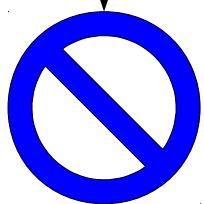
```
int main() {
    Cell* list = nullptr;
    prependTo(list, "Descartes");
    prependTo(list, "Kant");
    prependTo(list, "Bentham");

    return 0;
}
```

```
int main() {
    Cell* list = nullptr;
    prependTo(list, "Descartes");
    prependTo(list, "Kant");
    prependTo(list, "Bentham");

    return 0;
}
```

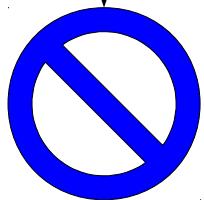
list



```
int main() {
    Cell* list = nullptr;
    prependTo(list, "Descartes");
    prependTo(list, "Kant");
    prependTo(list, "Bentham");

    return 0;
}
```

list



```
int main() {  
    cell* list = nullptr;  
    prep  
    prep  
    prep  
    return 0;  
}
```

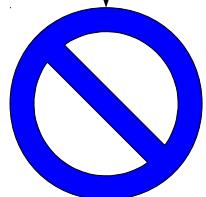
```
void prependTo(Cell*& list, const string& value) {  
    Cell* cell = new Cell;  
    cell->value = value;  
    cell->next = list;  
    list = cell;  
}
```

list



value

Descartes



```
int main() {
    Cell* list = nullptr;
    prep();
    prep();
    prep();
    return 0;
}

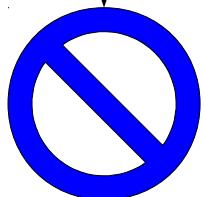
void prependTo(Cell*& list, const string& value) {
    Cell* cell = new Cell;
    cell->value = value;
    cell->next = list;
    list = cell;
}
```

list



value

Descartes



```
int main() {  
    Cell* list = nullptr;  
    prep();  
    prep();  
    prep();  
    return 0;  
}
```

```
void prependTo(Cell*& list, const string& value) {  
    Cell* cell = new Cell;  
    cell->value = value;  
    cell->next = list;  
    list = cell;  
}
```

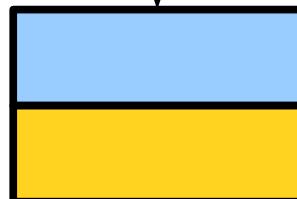
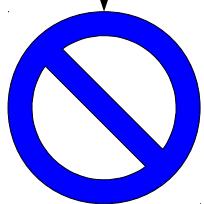
list



cell



value



```
int main() {  
    cell* list = nullptr;  
    prep  
    prep  
    prep  
    return 0;  
}
```

```
void prependTo(Cell*& list, const string& value) {  
    Cell* cell = new Cell;  
    cell->value = value;  
    cell->next = list;  
    list = cell;  
}
```

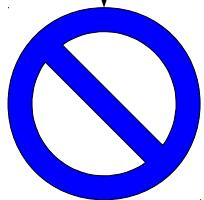
list



cell



value



```
int main() {  
    Cell* list = nullptr;  
    prep  
    prep  
    prep  
    return 0;  
}
```

```
void prependTo(Cell*& list, const string& value) {  
    Cell* cell = new Cell;  
    cell->value = value;  
    cell->next = list;  
    list = cell;  
}
```

list



cell



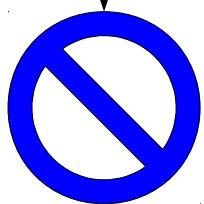
value



Descartes



?



```
int main() {  
    cell* list = nullptr;  
    prep  
    prep  
    prep  
    return 0;  
}
```

```
void prependTo(Cell*& list, const string& value) {  
    Cell* cell = new Cell;  
    cell->value = value;  
    cell->next = list;  
    list = cell;  
}
```

list



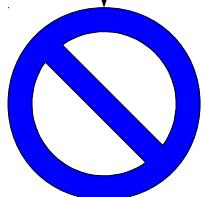
cell



value



Descartes



```
int main() {  
    Cell* list = nullptr;  
    prep  
    prep  
    prep  
    return 0;  
}
```

```
void prependTo(Cell*& list, const string& value) {  
    Cell* cell = new Cell;  
    cell->value = value;  
    cell->next = list;  
    list = cell;  
}
```

list



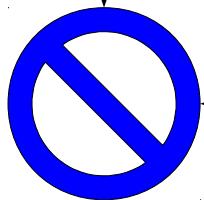
cell



value



Descartes



```
int main() {  
    Cell* list = nullptr;  
    prep  
    prep  
    prep  
    return 0;  
}
```

```
void prependTo(Cell*& list, const string& value) {  
    Cell* cell = new Cell;  
    cell->value = value;  
    cell->next = list;  
    list = cell;  
}
```

list



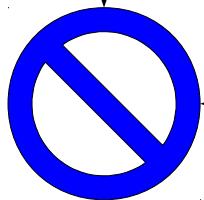
cell



value



Descartes



```
int main() {  
    cell* list = nullptr;  
    prep  
    prep  
    prep  
    return 0;  
}
```

list



```
void prependTo(Cell*& list, const string& value) {  
    Cell* cell = new Cell;  
    cell->value = value;  
    cell->next = list;  
    list = cell;  
}
```

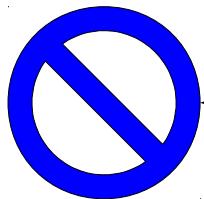
cell



value



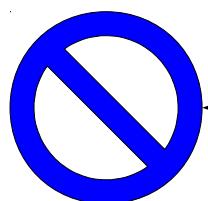
Descartes



```
int main() {
    cell* list = nullptr;
    prependTo(list, "Descartes");
    prependTo(list, "Kant");
    prependTo(list, "Bentham");

    return 0;
}
```

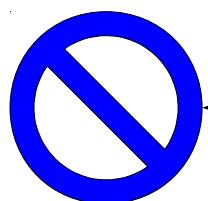
list



```
int main() {
    Cell* list = nullptr;
    prependTo(list, "Descartes");
    prependTo(list, "Kant");
    prependTo(list, "Bentham");

    return 0;
}
```

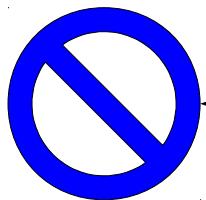
list



```
int main() {
    Cell* list = nullptr;
    prependTo(list, "Descartes");
    prependTo(list, "Kant");
    prependTo(list, "Bentham");

    return 0;
}
```

list



*I link,  
therefore I am.*

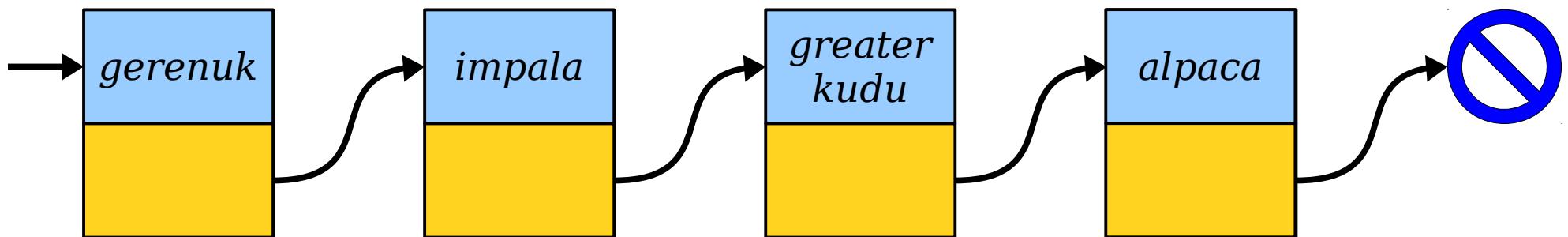
# Pointers by Reference

- If you pass a pointer into a function *by value*, you can change the contents at the object you point at, but not *which* object you point at.
- If you pass a pointer into a function *by reference*, you can *also* change *which* object is pointed at.

# Appending to a List

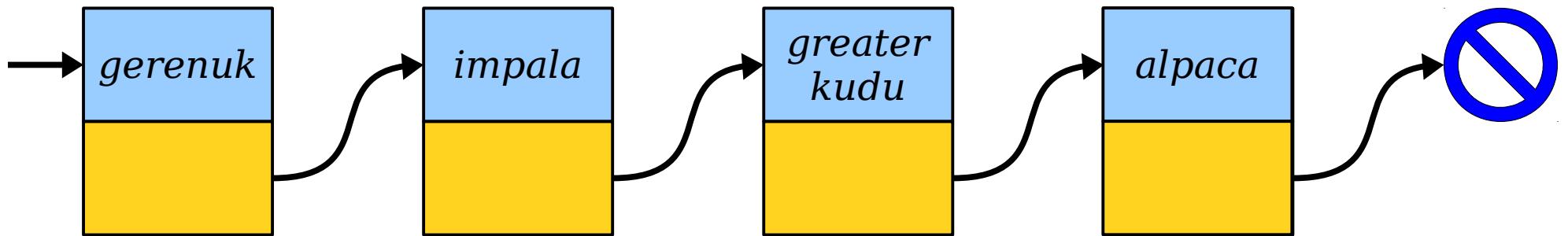
# Appending to a List

- Think about which link needs to get changed to append something to this list:



# Appending to a List

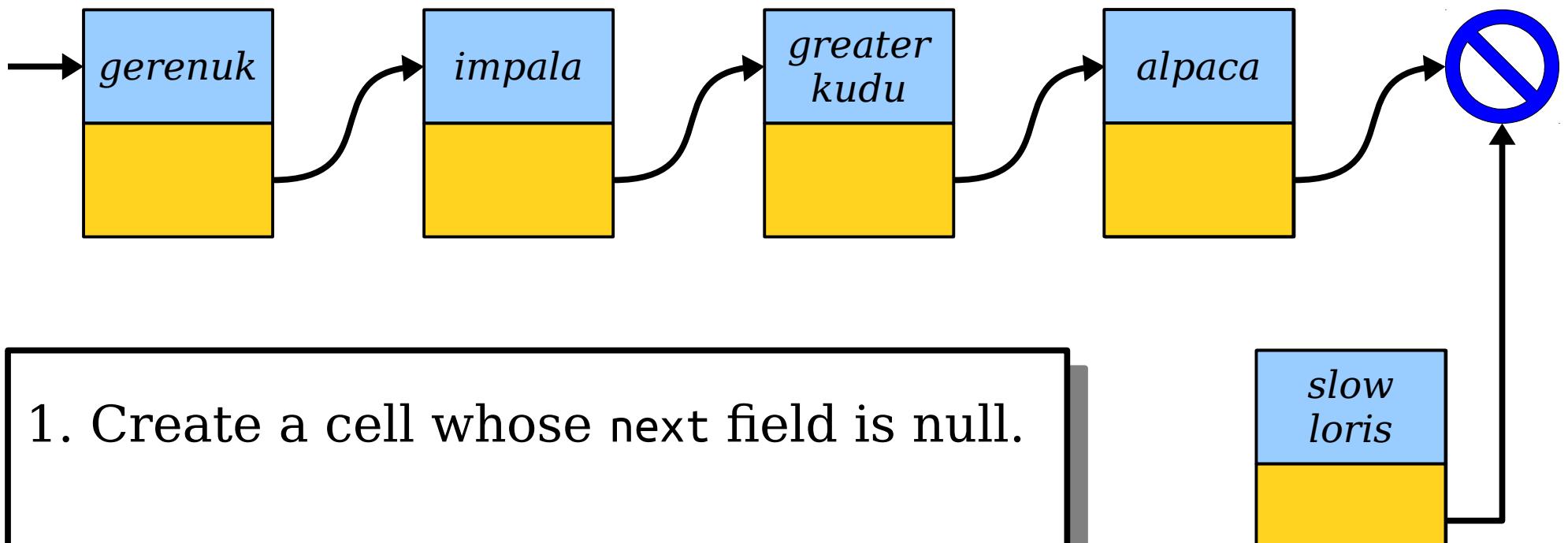
- Think about which link needs to get changed to append something to this list:



1. Create a cell whose `next` field is null.

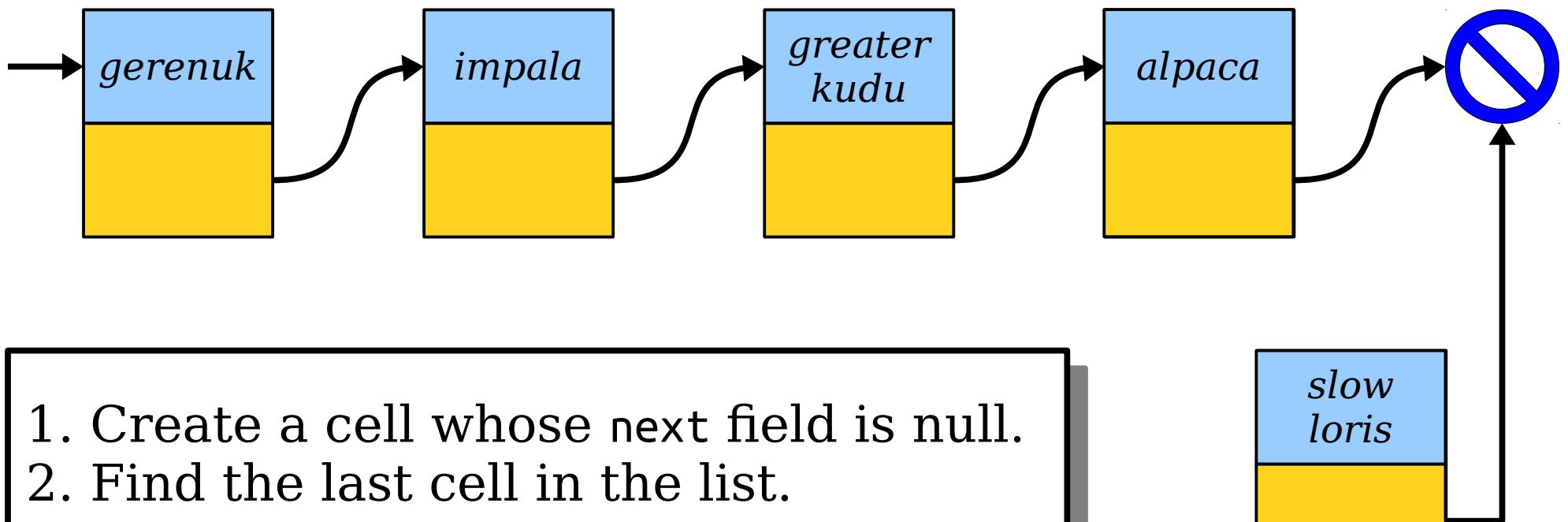
# Appending to a List

- Think about which link needs to get changed to append something to this list:



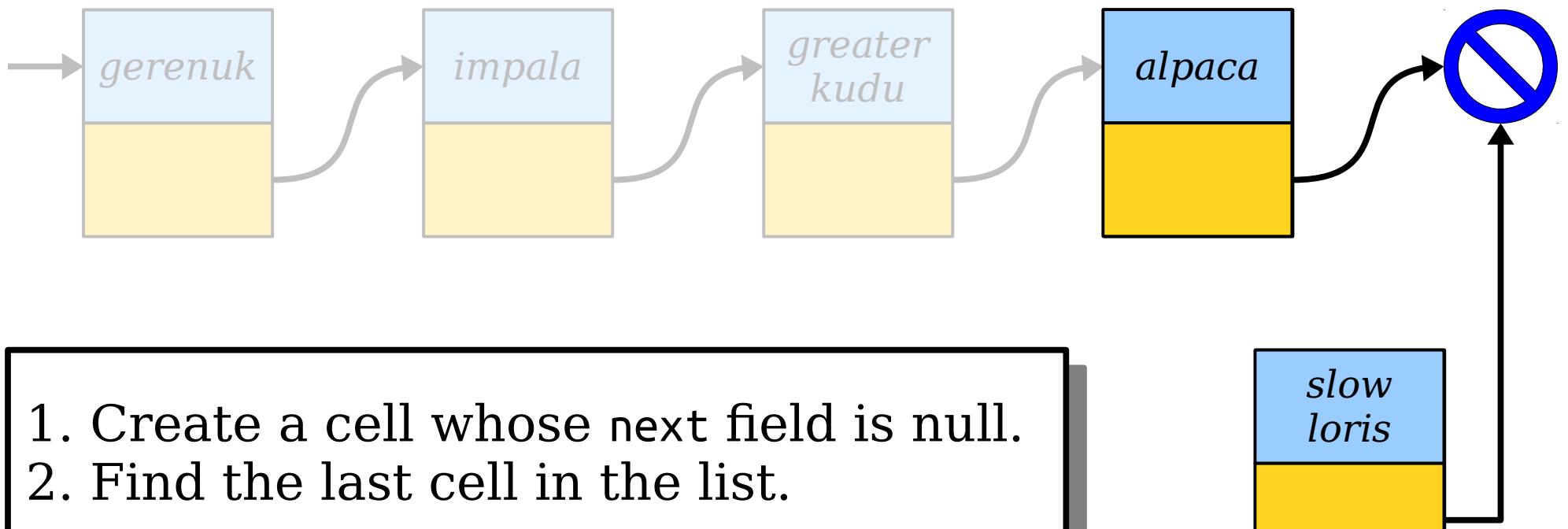
# Appending to a List

- Think about which link needs to get changed to append something to this list:



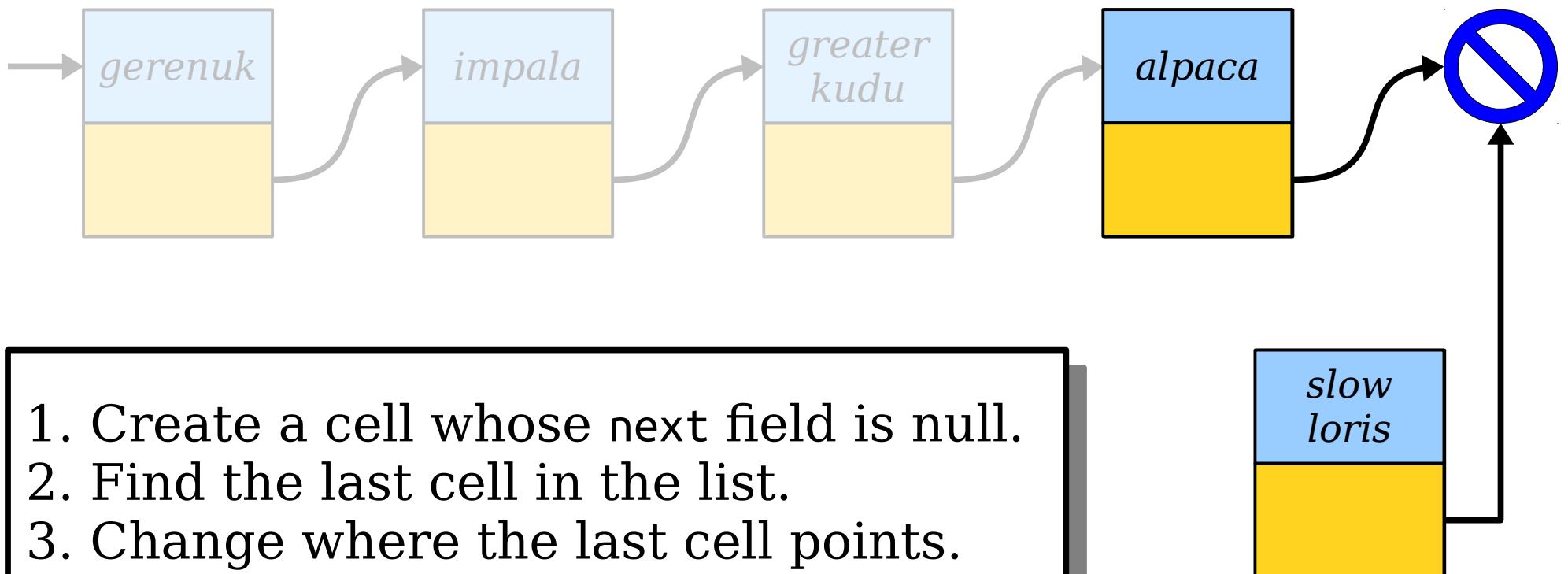
# Appending to a List

- Think about which link needs to get changed to append something to this list:



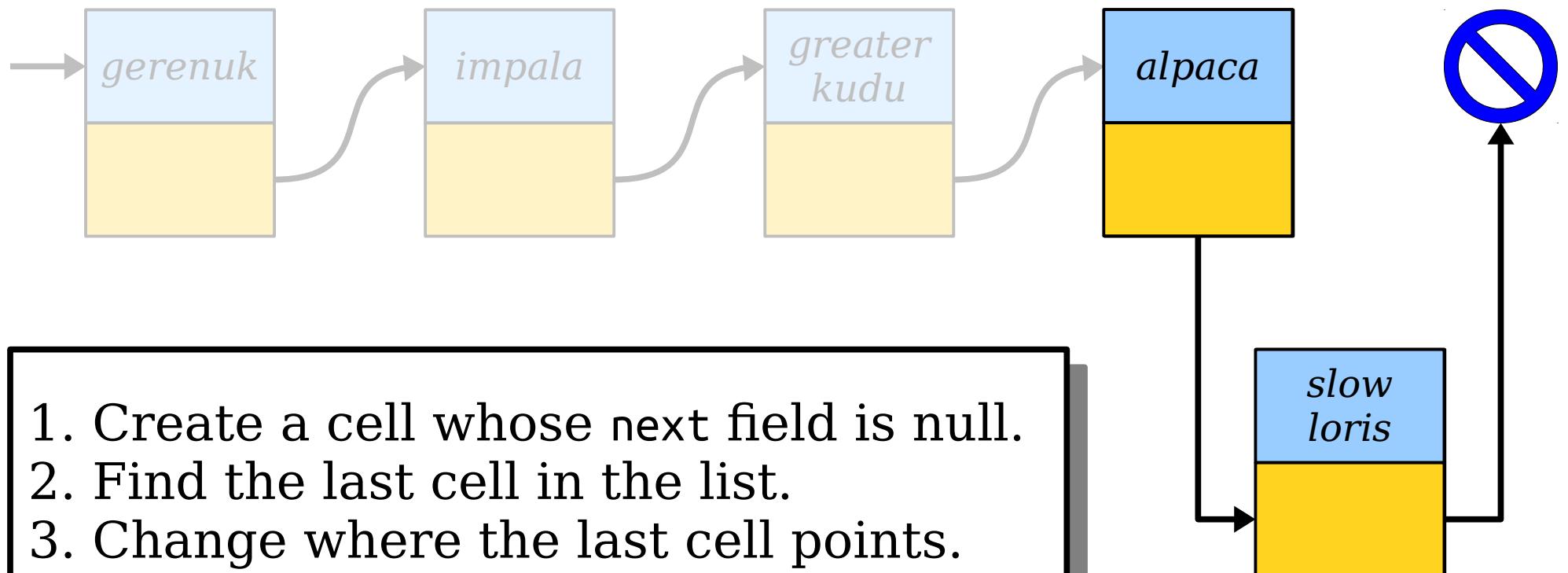
# Appending to a List

- Think about which link needs to get changed to append something to this list:



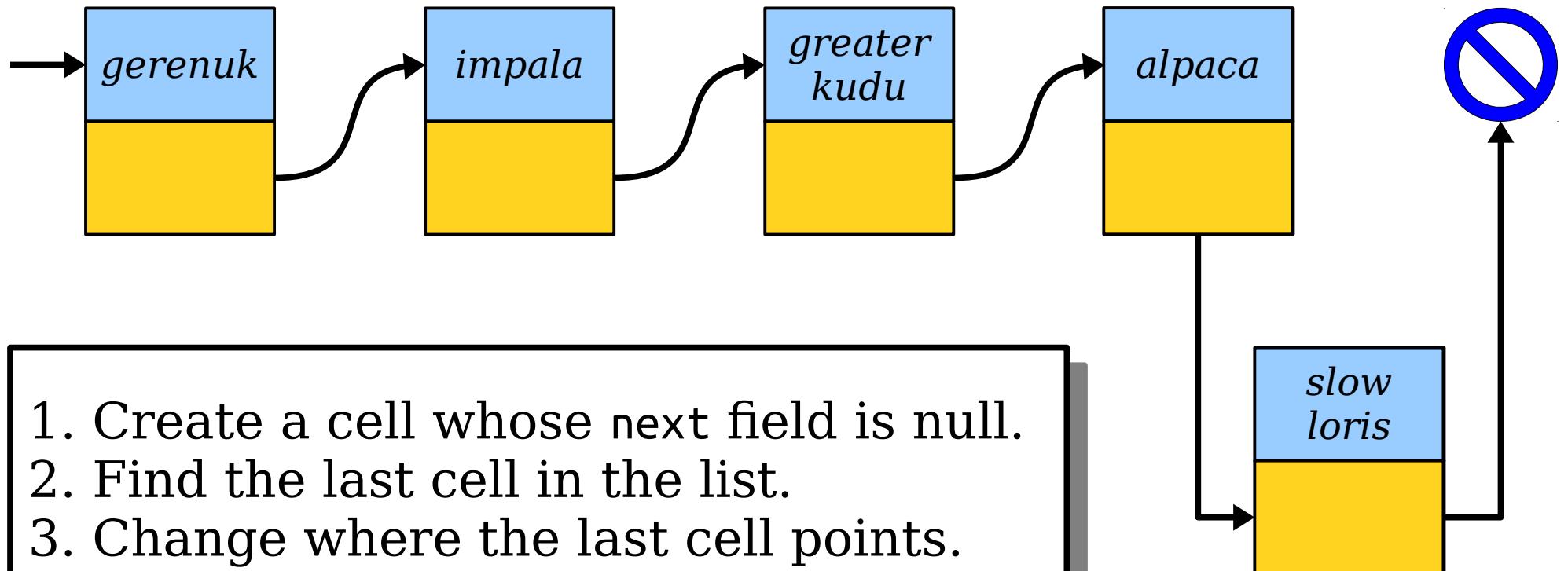
# Appending to a List

- Think about which link needs to get changed to append something to this list:



# Appending to a List

- Think about which link needs to get changed to append something to this list:



# What Went Wrong?

```
int main() {
    Cell* list = nullptr;
    appendTo(list, "Last");
    appendTo(list, "Final");
    appendTo(list, "Ultimate");
    appendTo(list, "Terminal");

    /* ... other listy things. ... */
}
```

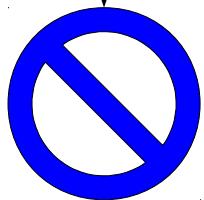
```
int main() {
    Cell* list = nullptr;
    appendTo(list, "Last");
    appendTo(list, "Final");
    appendTo(list, "Ultimate");
    appendTo(list, "Terminal");

    /* ... other listy things. ... */
}
```

```
int main() {
    Cell* list = nullptr;
    appendTo(list, "Last");
    appendTo(list, "Final");
    appendTo(list, "Ultimate");
    appendTo(list, "Terminal");

    /* ... other listy things. ... */
}
```

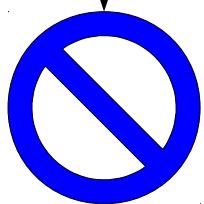
list



```
int main() {
    Cell* list = nullptr;
    appendTo(list, "Last");
    appendTo(list, "Final");
    appendTo(list, "Ultimate");
    appendTo(list, "Terminal");

    /* ... other listy things. ... */
}
```

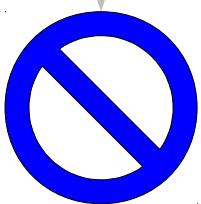
list



```
int main() {
    Cell* list = nullptr;
    appendTo(list, "Last");
    appendTo(list, "Final");
    appendTo(list, "Ultimate");
    appendTo(list, "Terminal");

    /* ... other listy things. ... */
}
```

list



```
int main() {
    Cell* list;
    appendTo(list, "First");
    appendTo(list, "Second");
    appendTo(list, "Third");
    appendTo(list, "Fourth");

    /* ... other code ...
    }

    list
```

```
void appendTo(Cell* list, const string& value) {
    Cell* cell = new Cell;
    cell->value = value;
    cell->next = nullptr;

    while (list->next != nullptr) {
        list = list->next;
    }

    list->next = cell;
}
```

value  
Last



```
int main() {
    Cell* list;
    appendTo(list, "First");
    appendTo(list, "Second");
    appendTo(list, "Third");
    appendTo(list, "Fourth");

    /* ... other code ...
    }

    list
```

```
void appendTo(Cell* list, const string& value) {
    Cell* cell = new Cell;
    cell->value = value;
    cell->next = nullptr;
```

```
    while (list->next != nullptr) {
        list = list->next;
    }
```

```
    list->next = cell;
}
```

```
list
```

value

Last



```
int main() {  
    Cell* list;  
    appendTo(list, "First");  
    appendTo(list, "Second");  
    appendTo(list, "Third");  
    appendTo(list, "Fourth");  
  
    /* ... other code */  
}  
  
list
```

```
void appendTo(Cell* list, const string& value) {  
    Cell* cell = new Cell;  
    cell->value = value;  
    cell->next = nullptr;
```

```
        while (list->next != nullptr) {  
            list = list->next;
```

```
        }  
        list->next = cell;
```

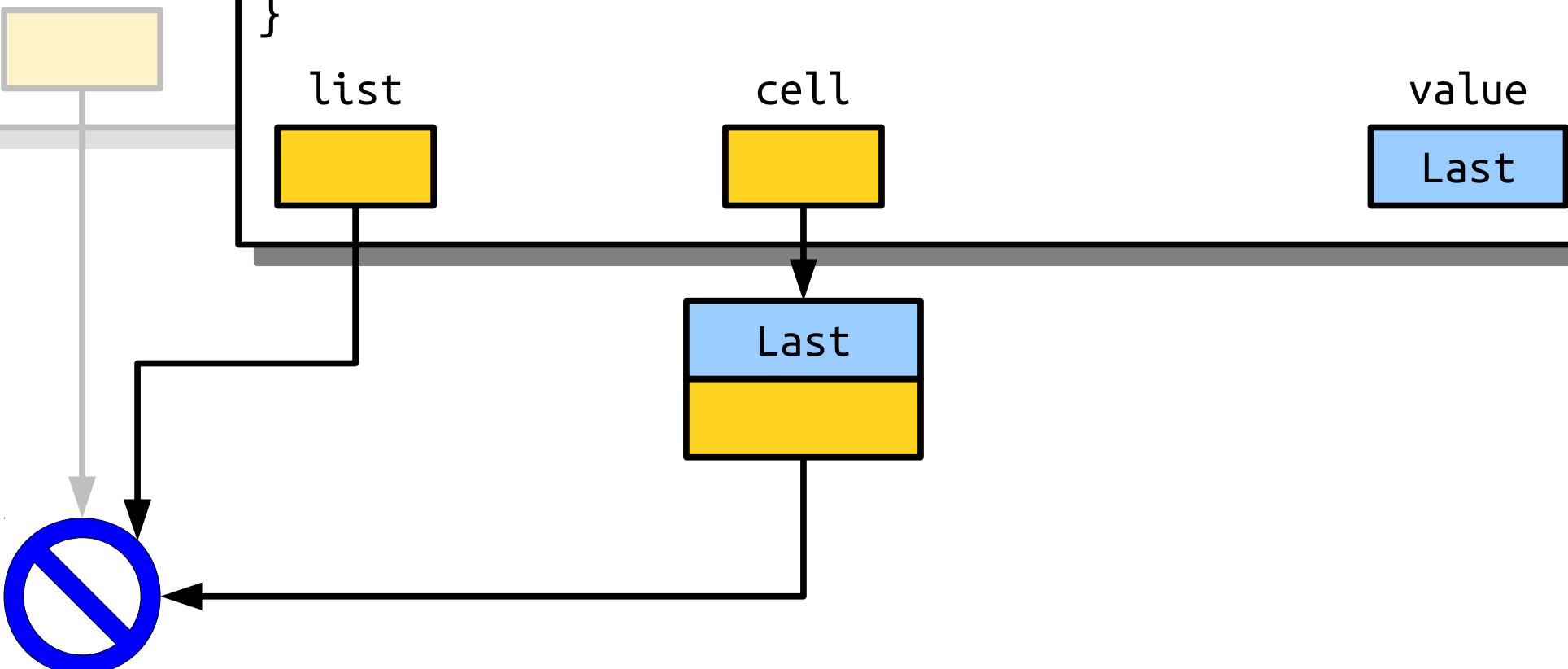
```
}
```

list

cell

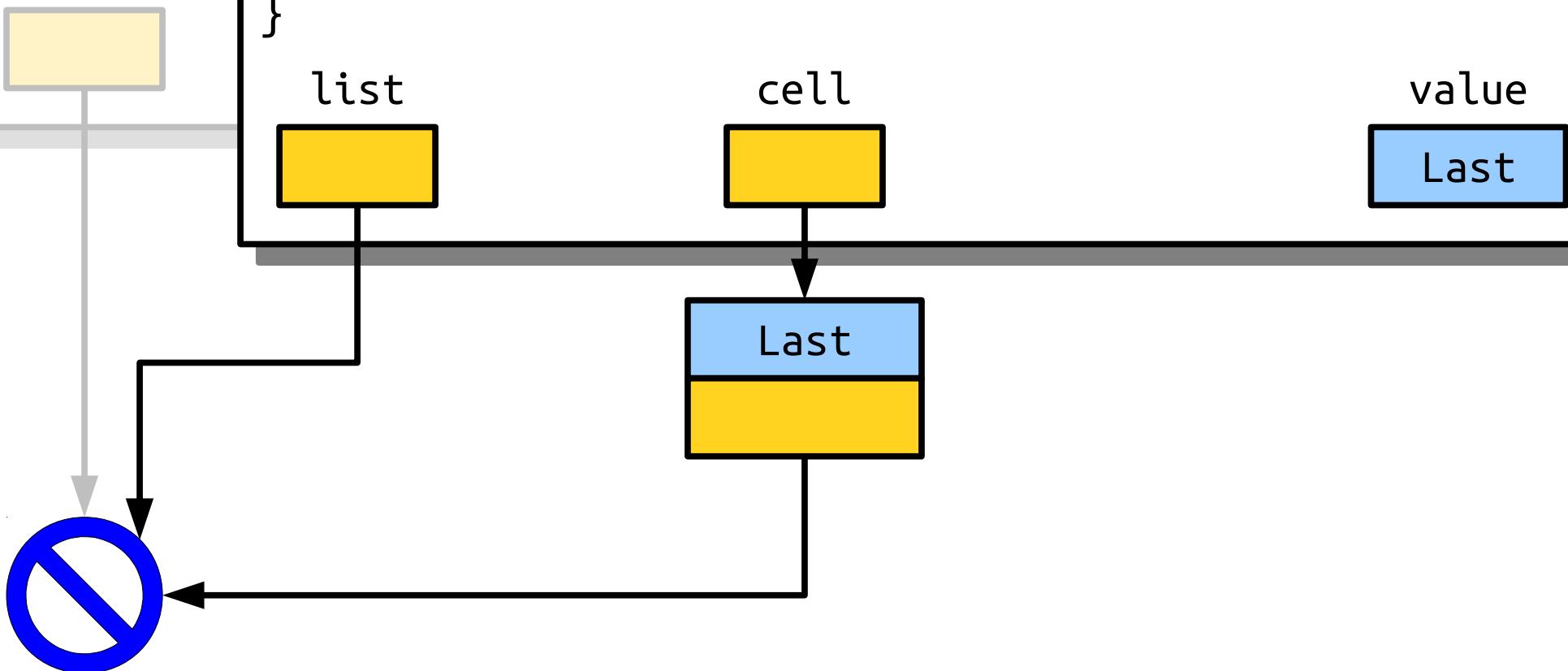
value

Last



```
int main() {  
    Cell* list;  
    appendTo(list, "First");  
    appendTo(list, "Second");  
    appendTo(list, "Third");  
    appendTo(list, "Fourth");  
    /* ... other code */  
}  
list
```

```
void appendTo(Cell* list, const string& value) {  
    Cell* cell = new Cell;  
    cell->value = value;  
    cell->next = nullptr;  
  
    while (list->next != nullptr) {  
        list = list->next;  
    }  
  
    list->next = cell;  
}
```



```

int main() {
    Cell* list;
    appendTo(list, "First");
    appendTo(list, "Second");
    appendTo(list, "Third");
    appendTo(list, "Fourth");
    /* ... other code ...
}
list

```

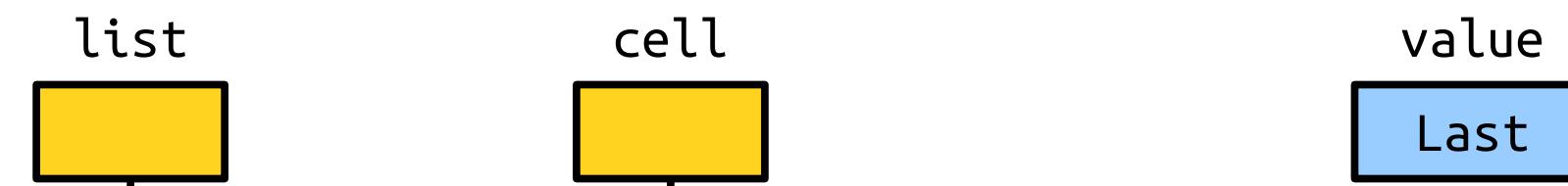
**void appendTo(Cell\* list, const string& value) {**

    Cell\* cell = **new** Cell;  
 cell->value = value;  
 cell->next = **nullptr**;

**while** (list->next != **nullptr**) { // Uh oh!

        list = list->next;
 }

    list->next = cell;
}



**Null Pointer  
Dereference!**

# Appending to a List

- There's an edge case we missed! We need to account for the list being empty.
- If the list is empty, we should change the list pointer to point to our new cell.
- Let's change things up and see if we can fix this problem.

What Went Wrong (This Time)?

What Went Wrong (This Other Time)?

```
int main() {
    Cell* list = nullptr;
    appendTo(list, "Last");
    appendTo(list, "Final");
    appendTo(list, "Ultimate");
    appendTo(list, "Terminal");

    /* ... other listy things. ... */
}
```

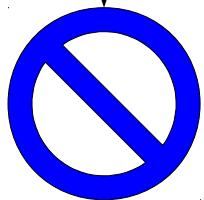
```
int main() {
    Cell* list = nullptr;
    appendTo(list, "Last");
    appendTo(list, "Final");
    appendTo(list, "Ultimate");
    appendTo(list, "Terminal");

    /* ... other listy things. ... */
}
```

```
int main() {
    Cell* list = nullptr;
    appendTo(list, "Last");
    appendTo(list, "Final");
    appendTo(list, "Ultimate");
    appendTo(list, "Terminal");

    /* ... other listy things. ... */
}
```

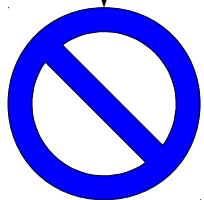
list



```
int main() {
    Cell* list = nullptr;
    appendTo(list, "Last");
    appendTo(list, "Final");
    appendTo(list, "Ultimate");
    appendTo(list, "Terminal");

    /* ... other listy things. ... */
}
```

list



```
int main() {
    Cell* list;
    appendTo(list, "First");
    appendTo(list, "Second");
    appendTo(list, "Third");
    appendTo(list, "Fourth");
    /* ... other code ...
    }

    list

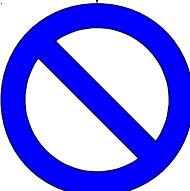
}

void appendTo(Cell*& list, const string& value) {
    Cell* cell = new Cell;
    cell->value = value;
    cell->next = nullptr;

    while (list != nullptr && list->next != nullptr) {
        list = list->next;
    }

    if (list == nullptr) {
        list = cell;
    } else {
        list->next = cell;
    }
}
```

value  
Last

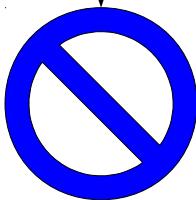


```
int main() {
    Cell* list;
    appendTo(list, "First");
    appendTo(list, "Second");
    appendTo(list, "Third");
    appendTo(list, "Fourth");
    /* ... other code ...
    }

    list
     
}

void appendTo(Cell*& list, const string& value) {
    Cell* cell = new Cell;
    cell->value = value;
    cell->next = nullptr;
    while (list != nullptr && list->next != nullptr) {
        list = list->next;
    }
    if (list == nullptr) {
        list = cell;
    } else {
        list->next = cell;
    }
}
```

value  
Last



```
int main() {
    Cell* list;
    appendTo(list, "First");
    appendTo(list, "Second");
    appendTo(list, "Third");
    appendTo(list, "Last");
    /* ... other code */
}

list
```

```
void appendTo(Cell*& list, const string& value) {
    Cell* cell = new Cell;
    cell->value = value;
    cell->next = nullptr;

    while (list != nullptr && list->next != nullptr) {
        list = list->next;
    }

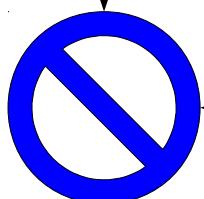
    if (list == nullptr) {
        list = cell;
    } else {
        list->next = cell;
    }
}
```

cell

value

Last

Last



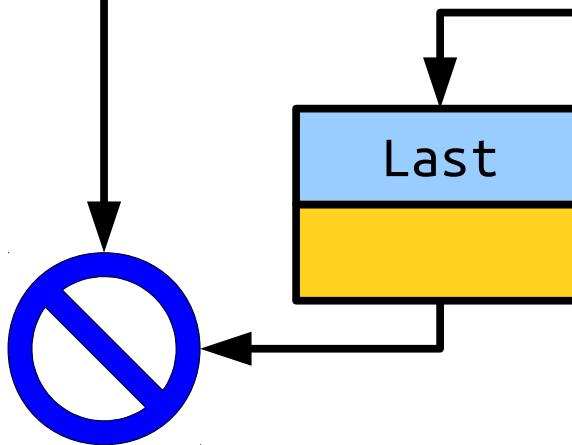
```
int main() {
    Cell* list;
    appendTo(list, "First");
    appendTo(list, "Second");
    appendTo(list, "Third");
    appendTo(list, "Last");
    /* ... other code */
}

list
```

```
void appendTo(Cell*& list, const string& value) {
    Cell* cell = new Cell;
    cell->value = value;
    cell->next = nullptr;

    while (list != nullptr && list->next != nullptr) {
        list = list->next;
    }

    if (list == nullptr) {
        list = cell;
    } else {
        list->next = cell;
    }
}
```



```
int main() {
    Cell* list;
    appendTo(list, "First");
    appendTo(list, "Second");
    appendTo(list, "Third");
    appendTo(list, "Fourth");
    /* ... other code */
}

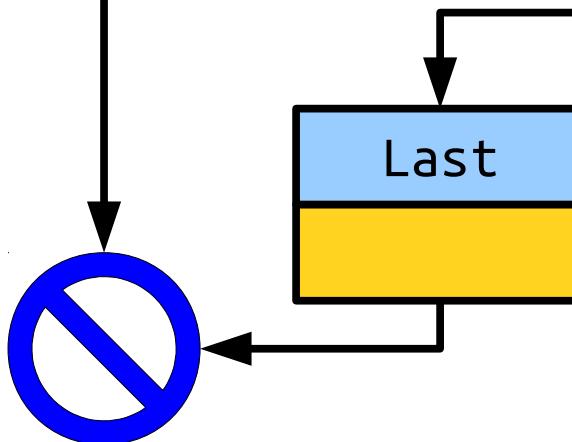
list
```

```
void appendTo(Cell*& list, const string& value) {
    Cell* cell = new Cell;
    cell->value = value;
    cell->next = nullptr;

    while (list != nullptr && list->next != nullptr) {
        list = list->next;
    }

    if (list == nullptr) {
        list = cell;
    } else {
        list->next = cell;
    }
}
```

The diagram illustrates the state of variables during the execution of the `appendTo` function. The variable `list` is represented by a yellow box pointing to a blue box labeled "Last". The variable `cell` is represented by a yellow box pointing to a blue box labeled "Last". The variable `value` is represented by a blue box containing the word "Last". This visualizes the state where the list is empty (null), and the new node being appended has the same value as the current last node.



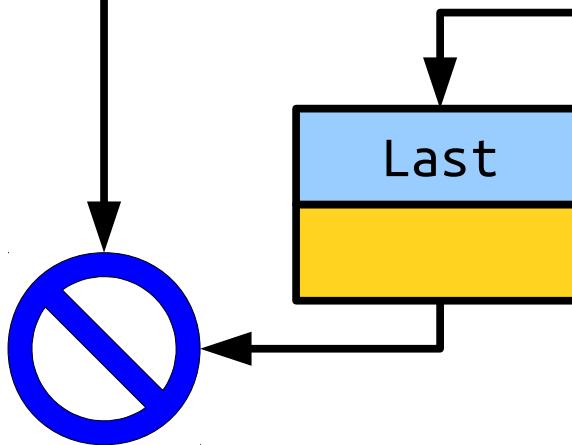
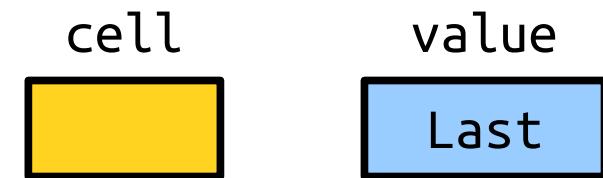
```
int main() {
    Cell* list;
    appendTo(list, "First");
    appendTo(list, "Second");
    appendTo(list, "Third");
    appendTo(list, "Fourth");
    /* ... other code ...
    }

    list
    list
}

void appendTo(Cell*& list, const string& value) {
    Cell* cell = new Cell;
    cell->value = value;
    cell->next = nullptr;

    while (list != nullptr && list->next != nullptr) {
        list = list->next;
    }

    if (list == nullptr) {
        list = cell;
    } else {
        list->next = cell;
    }
}
```



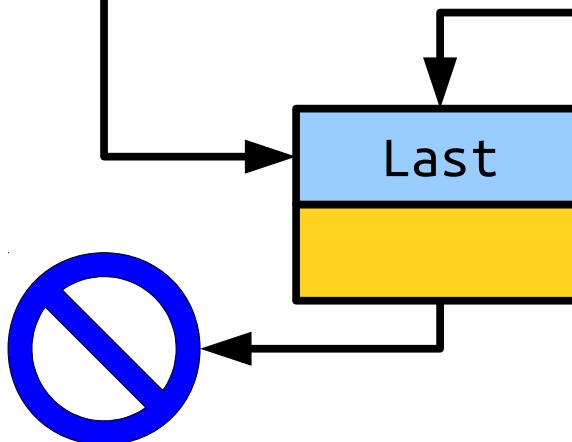
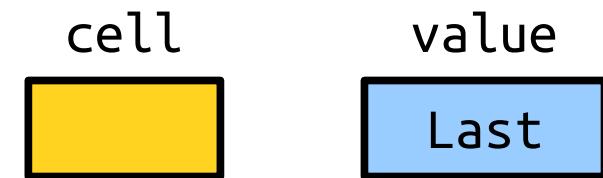
```
int main() {
    Cell* list;
    appendTo(list, "First");
    appendTo(list, "Second");
    appendTo(list, "Third");
    appendTo(list, "Fourth");
    /* ... other code ...
    }

    list
    list
}

void appendTo(Cell*& list, const string& value) {
    Cell* cell = new Cell;
    cell->value = value;
    cell->next = nullptr;

    while (list != nullptr && list->next != nullptr) {
        list = list->next;
    }

    if (list == nullptr) {
        list = cell;
    } else {
        list->next = cell;
    }
}
```

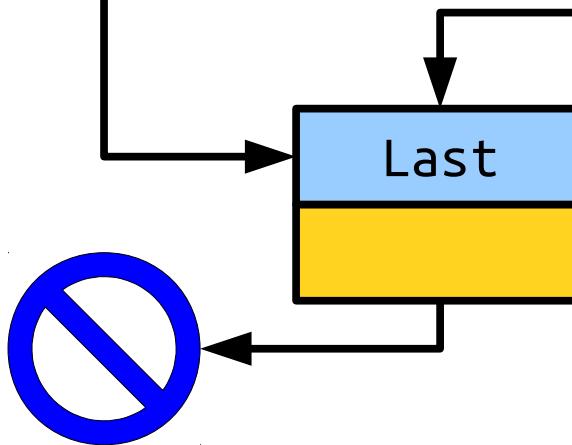


```
int main() {
    Cell* list;
    appendTo(list, "Last");
    appendTo(list, "Second");
    appendTo(list, "First");
    appendTo(list, "Fourth");
    /* ... other code ...
}
list
```

```
void appendTo(Cell*& list, const string& value) {
    Cell* cell = new Cell;
    cell->value = value;
    cell->next = nullptr;

    while (list != nullptr && list->next != nullptr) {
        list = list->next;
    }

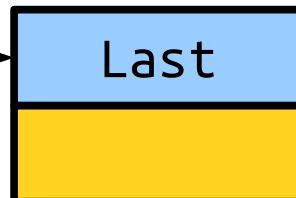
    if (list == nullptr) {
        list = cell;
    } else {
        list->next = cell;
    }
}
```



```
int main() {
    Cell* list = nullptr;
    appendTo(list, "Last");
    appendTo(list, "Final");
    appendTo(list, "Ultimate");
    appendTo(list, "Terminal");

    /* ... other listy things. ... */
}
```

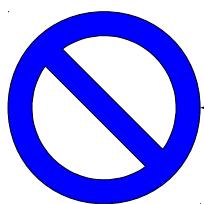
list



```
int main() {
    Cell* list = nullptr;
    appendTo(list, "Last");
    appendTo(list, "Final");
    appendTo(list, "Ultimate");
    appendTo(list, "Terminal");

    /* ... other listy things. ... */
}
```

list



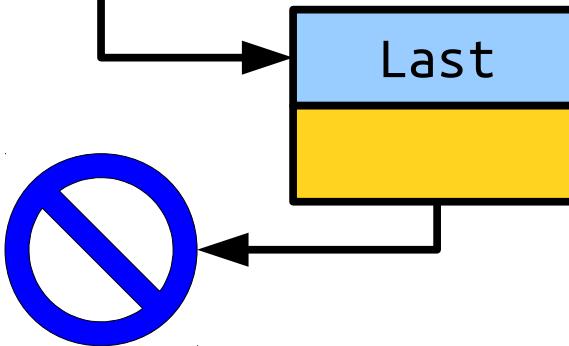
```
int main() {
    Cell* list;
    appendTo(list, "Initial");
    appendTo(list, "Middle");
    appendTo(list, "Final");
    appendTo(list, "Last");
    /* ... other code ...
    }

    list
    Initial
}

void appendTo(Cell*& list, const string& value) {
    Cell* cell = new Cell;
    cell->value = value;
    cell->next = nullptr;

    while (list != nullptr && list->next != nullptr) {
        list = list->next;
    }

    if (list == nullptr) {
        list = cell;
    } else {
        list->next = cell;
    }
}
```

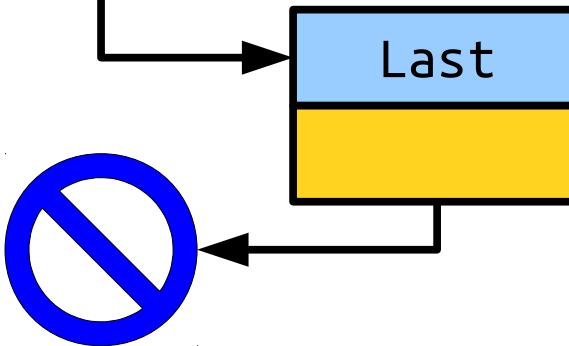


value  
Final

```
int main() {
    Cell* list;
    appendTo(list, "Initial");
    appendTo(list, "Middle");
    appendTo(list, "Final");
    appendTo(list, "Last");
    /* ... other code ...
    }

    list
    Initial
}

void appendTo(Cell*& list, const string& value) {
    Cell* cell = new Cell;
    cell->value = value;
    cell->next = nullptr;
    while (list != nullptr && list->next != nullptr) {
        list = list->next;
    }
    if (list == nullptr) {
        list = cell;
    } else {
        list->next = cell;
    }
}
```



value  
Final

```
int main() {
    Cell* list;
    appendTo(list, "Initial");
    appendTo(list, "Middle");
    appendTo(list, "Final");
    appendTo(list, "Last");
    /* ... other code ...
    }

    list
```

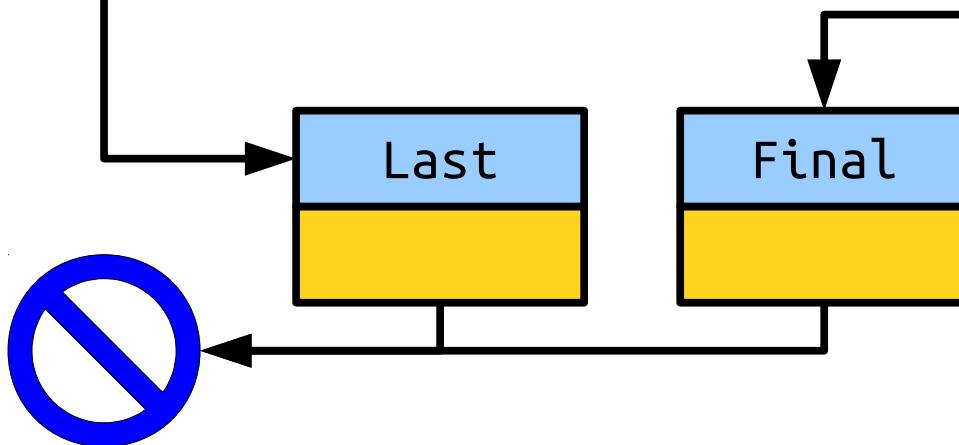
```
void appendTo(Cell*& list, const string& value) {
    Cell* cell = new Cell;
    cell->value = value;
    cell->next = nullptr;

    while (list != nullptr && list->next != nullptr) {
        list = list->next;
    }

    if (list == nullptr) {
        list = cell;
    } else {
        list->next = cell;
    }
}
```

cell

value  
Final



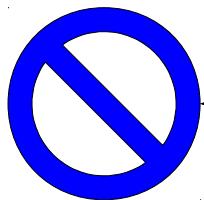
```
int main() {
    Cell* list;
    appendTo(list, "Initial");
    appendTo(list, "Middle");
    appendTo(list, "Final");
    appendTo(list, "Last");
    /* ... other code */
}

list
```

```
void appendTo(Cell*& list, const string& value) {
    Cell* cell = new Cell;
    cell->value = value;
    cell->next = nullptr;

    while (list != nullptr && list->next != nullptr) {
        list = list->next;
    }

    if (list == nullptr) {
        list = cell;
    } else {
        list->next = cell;
    }
}
```



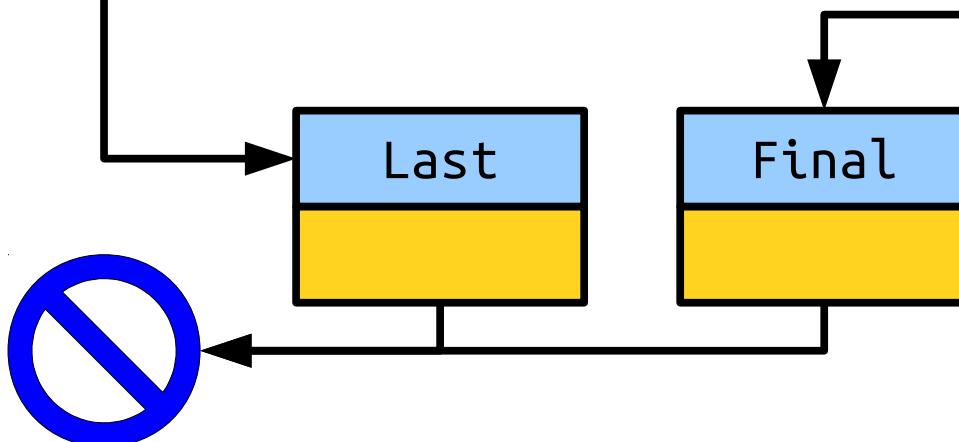
```
int main() {  
    Cell* list = appendTo(list, "Initial");  
    appendTo(list, "Final");  
    /* ... other code */  
}  
  
list
```

```
void appendTo(Cell*& list, const string& value) {  
    Cell* cell = new Cell;  
    cell->value = value;  
    cell->next = nullptr;  
  
    while (list != nullptr && list->next != nullptr) {  
        list = list->next;  
    }  
  
    if (list == nullptr) {  
        list = cell;  
    } else {  
        list->next = cell;  
    }  
}
```

cell

value

Final



```
int main() {
    Cell* list;
    appendTo(list, "Initial");
    appendTo(list, "Middle");
    appendTo(list, "Final");
    appendTo(list, "Last");
    /* ... other code ...
    }

    list
```

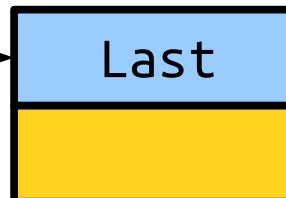
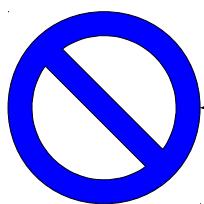
```
void appendTo(Cell*& list, const string& value) {
    Cell* cell = new Cell;
    cell->value = value;
    cell->next = nullptr;

    while (list != nullptr && list->next != nullptr) {
        list = list->next;
    }

    if (list == nullptr) {
        list = cell;
    } else {
        list->next = cell;
    }
}
```

cellvalue

InitialFinal



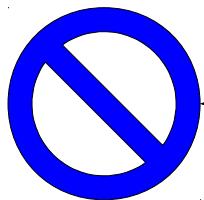
```
int main() {
    Cell* list;
    appendTo(list, "Initial");
    appendTo(list, "Middle");
    appendTo(list, "Final");
    appendTo(list, "Last");
    /* ... other code ...
    }

    list
```

```
void appendTo(Cell*& list, const string& value) {
    Cell* cell = new Cell;
    cell->value = value;
    cell->next = nullptr;

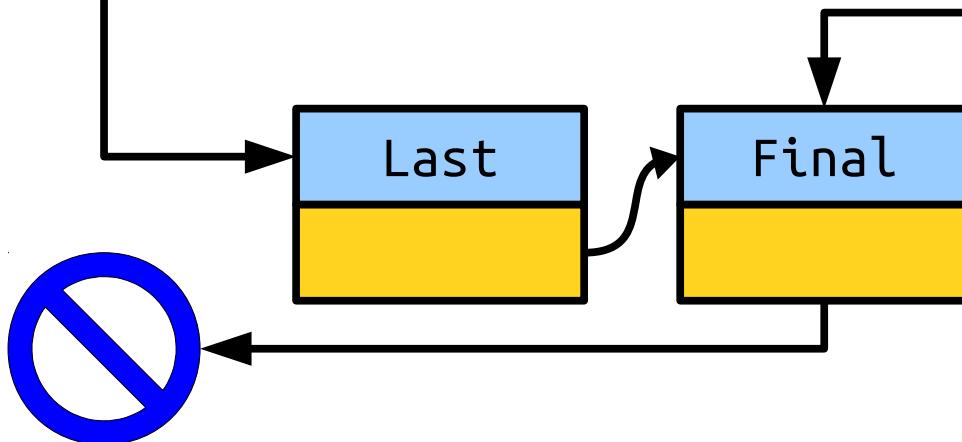
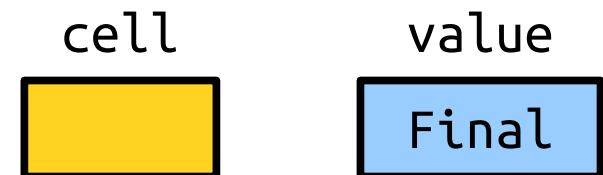
    while (list != nullptr && list->next != nullptr) {
        list = list->next;
    }

    if (list == nullptr) {
        list = cell;
    } else {
        list->next = cell;
    }
}
```



```
int main() {  
    Cell* list;  
    appendTo(list, "Initial");  
    appendTo(list, "Middle");  
    appendTo(list, "Final");  
    /* ... other code */  
}  
  
list
```

```
void appendTo(Cell*& list, const string& value) {  
    Cell* cell = new Cell;  
    cell->value = value;  
    cell->next = nullptr;  
  
    while (list != nullptr && list->next != nullptr) {  
        list = list->next;  
    }  
  
    if (list == nullptr) {  
        list = cell;  
    } else {  
        list->next = cell;  
    }  
}
```



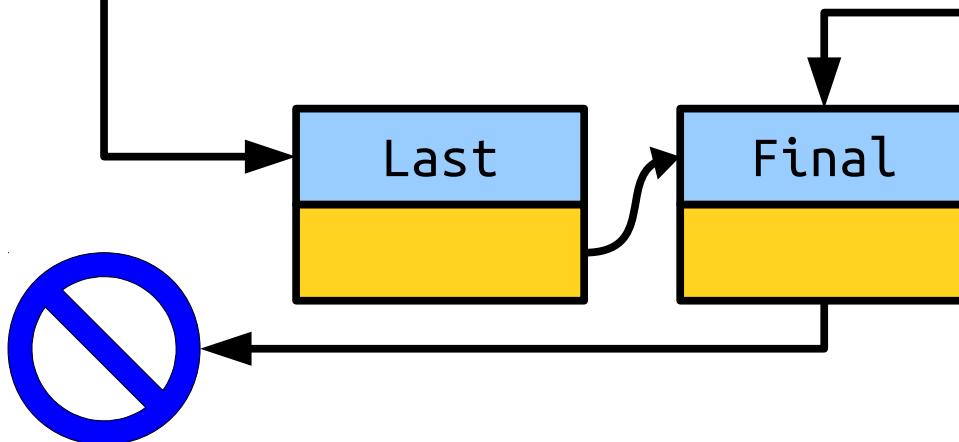
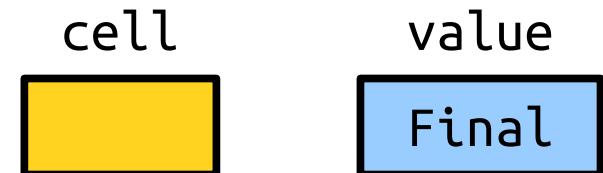
```
int main() {
    Cell* list;
    appendTo(list, "Initial");
    appendTo(list, "Middle");
    appendTo(list, "Final");
    appendTo(list, "Last");
    /* ... other code ...
    }

    list
```

```
void appendTo(Cell*& list, const string& value) {
    Cell* cell = new Cell;
    cell->value = value;
    cell->next = nullptr;

    while (list != nullptr && list->next != nullptr) {
        list = list->next;
    }

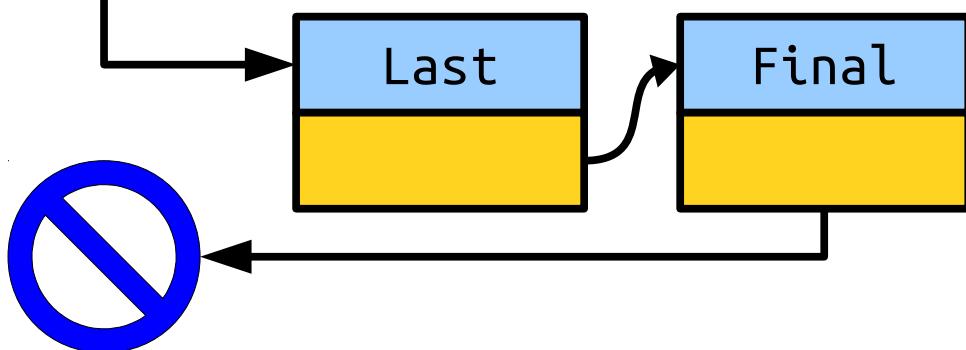
    if (list == nullptr) {
        list = cell;
    } else {
        list->next = cell;
    }
}
```



```
int main() {
    Cell* list = nullptr;
    appendTo(list, "Last");
    appendTo(list, "Final");
    appendTo(list, "Ultimate");
    appendTo(list, "Terminal");

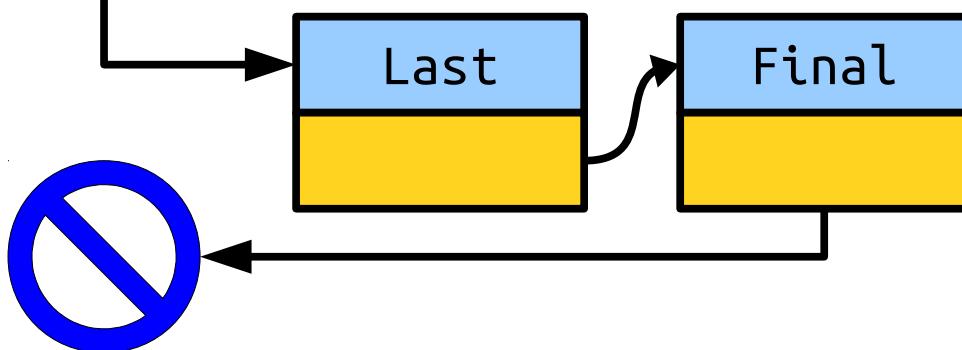
    /* ... other listy things. ... */
}
```

list



```
int main() {
    Cell* list = nullptr;
    appendTo(list, "Last");
    appendTo(list, "Final");
    appendTo(list, "Ultimate");
    appendTo(list, "Terminal");
    /* ... other listy things. ... */
}
```

list



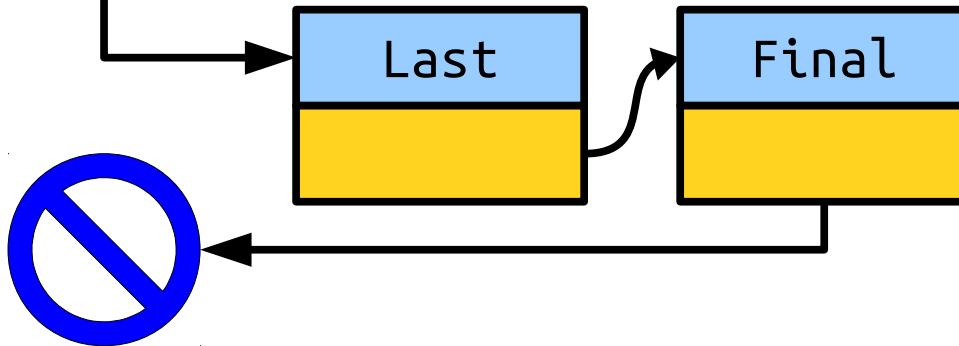
```
int main() {
    Cell* list;
    appendT
    appendT
    appendT
    appendT
    appendT
    /* ... ot
}
list
```

```
void appendTo(Cell*& list, const string& value) {
    Cell* cell = new Cell;
    cell->value = value;
    cell->next = nullptr;

    while (list != nullptr && list->next != nullptr) {
        list = list->next;
    }

    if (list == nullptr) {
        list = cell;
    } else {
        list->next = cell;
    }
}
```

value  
Ultimate



```
int main() {
    Cell* list;
    appendTo(list, "Ultimate");
    appendTo(list, "Final");
    appendTo(list, "Last");
    appendTo(list, "Initial");
    /* ... other code */
}

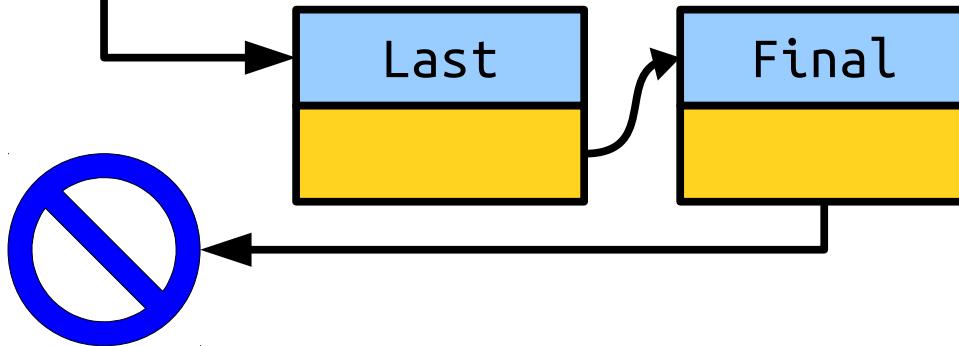
list
```

```
void appendTo(Cell*& list, const string& value) {
    Cell* cell = new Cell;
    cell->value = value;
    cell->next = nullptr;

    while (list != nullptr && list->next != nullptr) {
        list = list->next;
    }

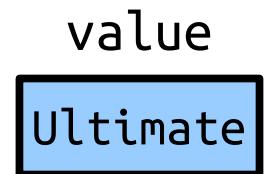
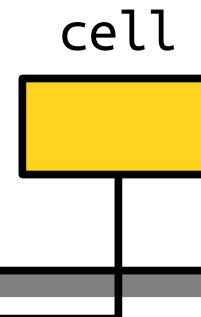
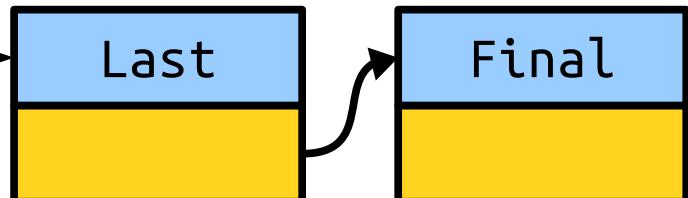
    if (list == nullptr) {
        list = cell;
    } else {
        list->next = cell;
    }
}
```

value  
Ultimate



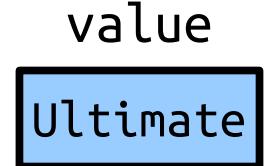
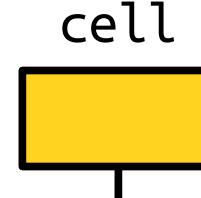
```
int main() {  
    Cell* list;  
    appendTo(list, "Initial");  
    appendTo(list, "Middle");  
    appendTo(list, "Ultimate");  
    appendTo(list, "Final");  
    /* ... other code */  
}  
  
list
```

```
void appendTo(Cell*& list, const string& value) {  
    Cell* cell = new Cell;  
    cell->value = value;  
    cell->next = nullptr;  
  
    while (list != nullptr && list->next != nullptr) {  
        list = list->next;  
    }  
  
    if (list == nullptr) {  
        list = cell;  
    } else {  
        list->next = cell;  
    }  
}
```



```
int main() {  
    Cell* list;  
    appendT...  
    appendT...  
    appendT...  
    appendT...  
    appendT...  
    /* ... other code */  
}  
  
list
```

```
void appendTo(Cell*& list, const string& value) {  
    Cell* cell = new Cell;  
    cell->value = value;  
    cell->next = nullptr;  
  
    while (list != nullptr && list->next != nullptr) {  
        list = list->next;  
    }  
  
    if (list == nullptr) {  
        list = cell;  
    } else {  
        list->next = cell;  
    }  
}
```



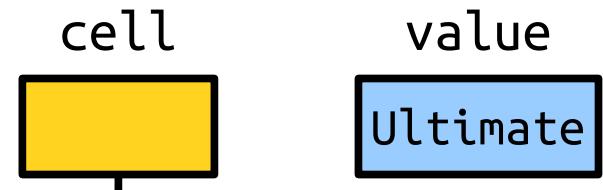
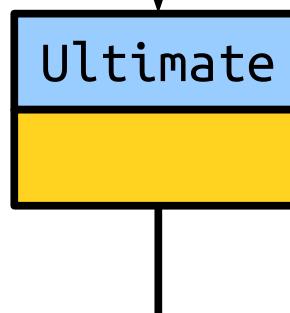
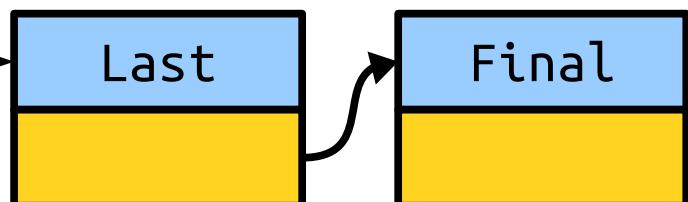
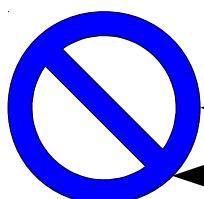
```
int main() {
    Cell* list;
    appendTo(list, "Initial");
    appendTo(list, "Middle");
    appendTo(list, "Final");
    appendTo(list, "Ultimate");
    /* ... other code */
}

list
```

```
void appendTo(Cell*& list, const string& value) {
    Cell* cell = new Cell;
    cell->value = value;
    cell->next = nullptr;

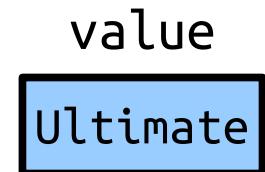
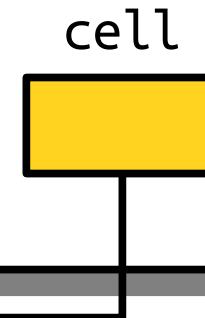
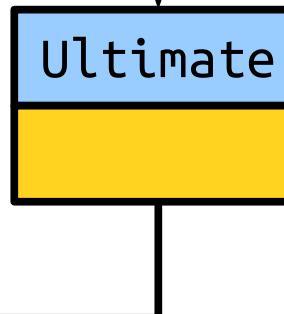
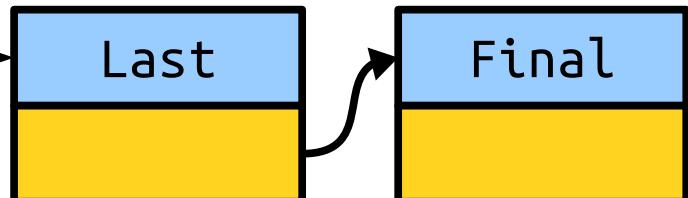
    while (list != nullptr && list->next != nullptr) {
        list = list->next;
    }

    if (list == nullptr) {
        list = cell;
    } else {
        list->next = cell;
    }
}
```



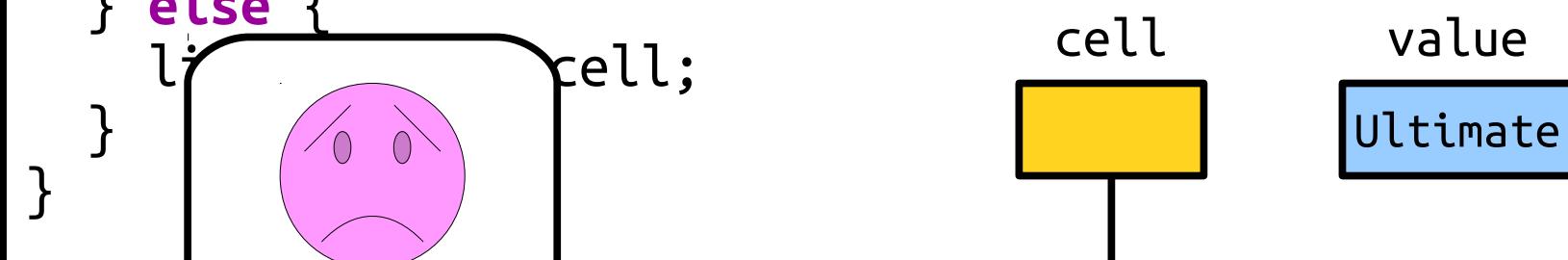
```
int main() {  
    Cell* list;  
    appendTo(list, "Initial");  
    appendTo(list, "Middle");  
    appendTo(list, "Ultimate");  
    appendTo(list, "Final");  
    /* ... other code */  
}  
  
list
```

```
void appendTo(Cell*& list, const string& value) {  
    Cell* cell = new Cell;  
    cell->value = value;  
    cell->next = nullptr;  
  
    while (list != nullptr && list->next != nullptr) {  
        list = list->next; // Uh oh!  
    }  
  
    if (list == nullptr) {  
        list = cell;  
    } else {  
        list->next = cell;  
    }  
}
```



```
int main() {  
    Cell* list;  
    appendTo(list, "Ultimate");  
    appendTo(list, "Final");  
    appendTo(list, "Last");  
    /* ... other code */  
}  
  
list
```

```
void appendTo(Cell*& list, const string& value) {  
    Cell* cell = new Cell;  
    cell->value = value;  
    cell->next = nullptr;  
  
    while (list != nullptr && list->next != nullptr) {  
        list = list->next; // Uh oh!  
    }  
  
    if (list == nullptr) {  
        list = cell;  
    } else {  
        list->next = cell;  
    }  
}
```



```

int main() {
    Cell* list;
    appendTo(list, "Initial");
    appendTo(list, "Middle");
    appendTo(list, "Final");
    appendTo(list, "Ultimate");
    /* ... other code ...
    }

    list
    [ ]
}

```

**void appendTo(Cell\*& list, const string& value) {**

```

    Cell* cell = new Cell;
    cell->value = value;
    cell->next = nullptr;

```

**while (list != nullptr && list->next != nullptr) {**

```

    list = list->next; // Uh oh!
}

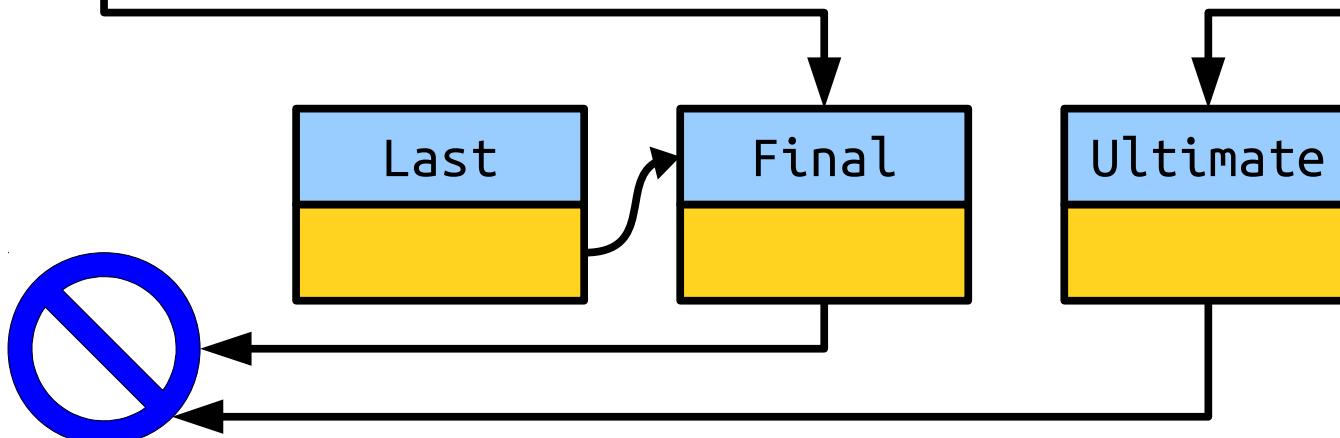
```

**if (list == nullptr) {**

```

    list = cell;
} else {
    list->next = cell;
}

```

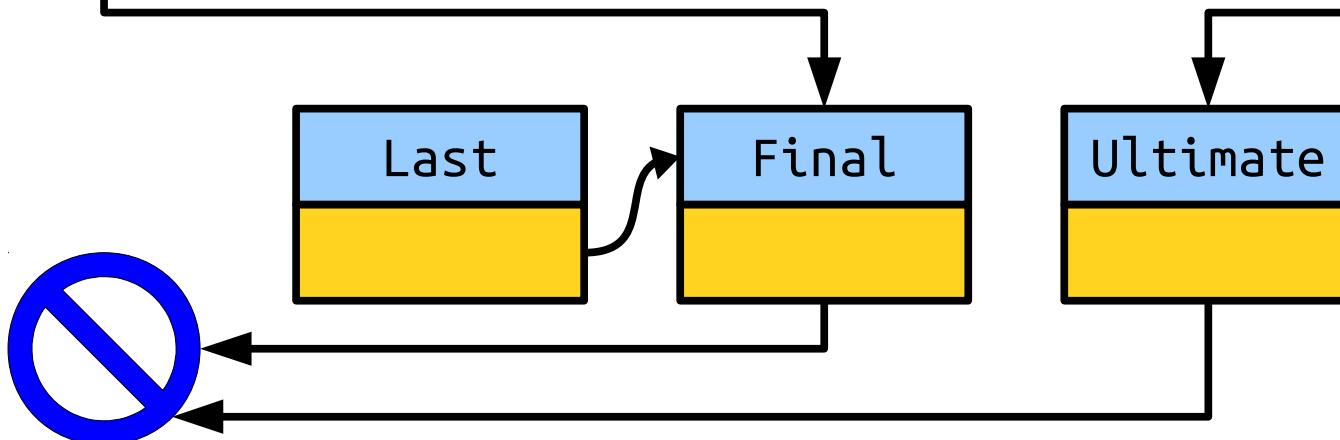


```
int main() {  
    Cell* list;  
    appendTo(list, "Initial");  
    appendTo(list, "Middle");  
    appendTo(list, "Ultimate");  
    appendTo(list, "Final");  
    /* ... other code */  
}  
  
list
```

```
void appendTo(Cell*& list, const string& value) {  
    Cell* cell = new Cell;  
    cell->value = value;  
    cell->next = nullptr;  
  
    while (list != nullptr && list->next != nullptr) {  
        list = list->next; // Uh oh!  
    }  
  
    if (list == nullptr) {  
        list = cell;  
    } else {  
        list->next = cell;  
    }  
}
```

```
cell
```

```
value  
Ultimate
```



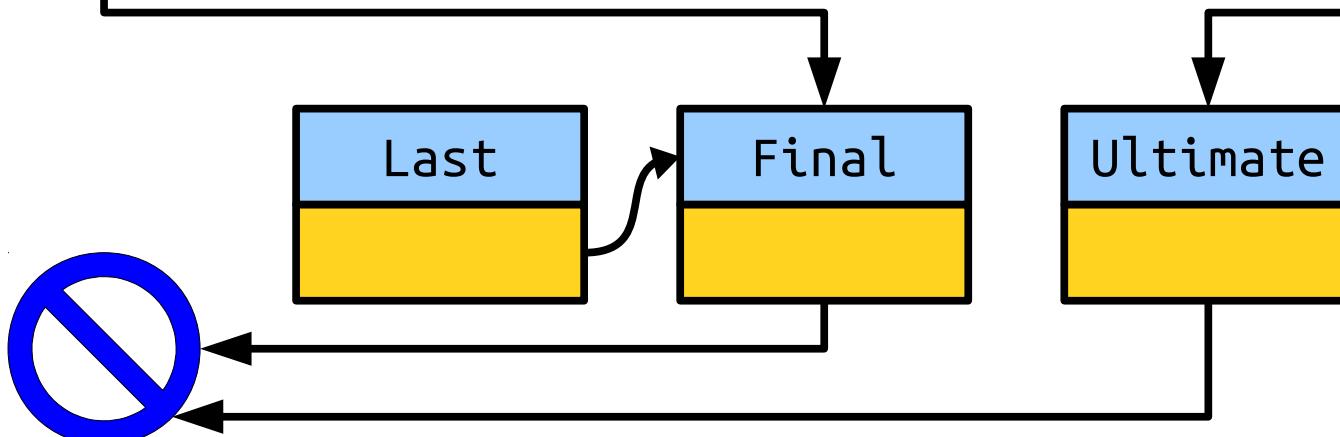
```
int main() {
    Cell* list;
    appendTo(list, "Initial");
    appendTo(list, "Middle");
    appendTo(list, "Final");
    appendTo(list, "Ultimate");
    /* ... other code ...
    }

    list
```

```
void appendTo(Cell*& list, const string& value) {
    Cell* cell = new Cell;
    cell->value = value;
    cell->next = nullptr;

    while (list != nullptr && list->next != nullptr) {
        list = list->next; // Uh oh!
    }

    if (list == nullptr) {
        list = cell;
    } else {
        list->next = cell;
    }
}
```



```

int main() {
    Cell* list;
    appendTo(list, "Initial");
    appendTo(list, "Middle");
    appendTo(list, "Final");
    appendTo(list, "Ultimate");
    /* ... other code ...
    }

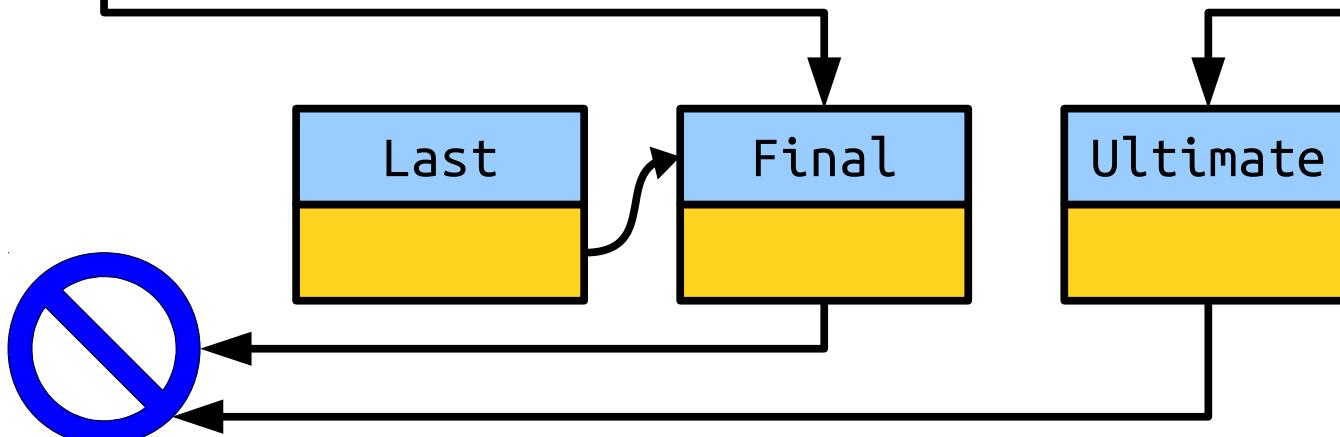
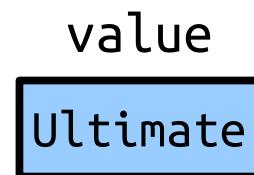
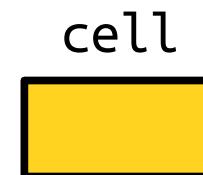
    list
    Cell
}

void appendTo(Cell*& list, const string& value) {
    Cell* cell = new Cell;
    cell->value = value;
    cell->next = nullptr;

    while (list != nullptr && list->next != nullptr) {
        list = list->next; // Uh oh!
    }

    if (list == nullptr) {
        list = cell;
    } else {
        list->next = cell;
    }
}

```



```

int main() {
    Cell* list;
    appendTo(list, "Initial");
    appendTo(list, "Final");
    appendTo(list, "Ultimate");
    appendTo(list, "Last");
    /* ... other code ...
    }

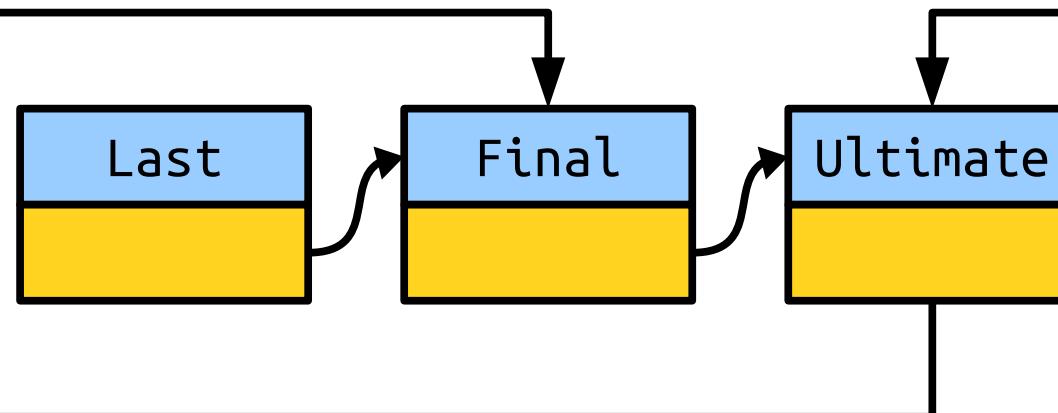
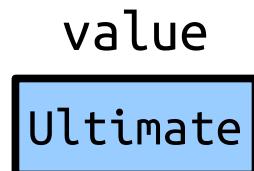
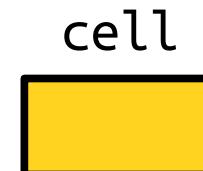
    list
    Cell
}

void appendTo(Cell*& list, const string& value) {
    Cell* cell = new Cell;
    cell->value = value;
    cell->next = nullptr;

    while (list != nullptr && list->next != nullptr) {
        list = list->next; // Uh oh!
    }

    if (list == nullptr) {
        list = cell;
    } else {
        list->next = cell;
    }
}

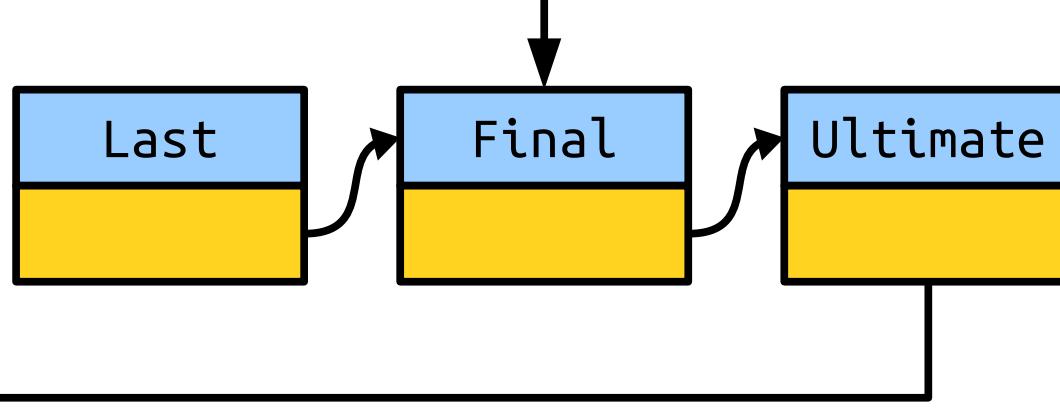
```



```
int main() {
    Cell* list = nullptr;
    appendTo(list, "Last");
    appendTo(list, "Final");
    appendTo(list, "Ultimate");
    appendTo(list, "Terminal");

    /* ... other listy things. ... */
}
```

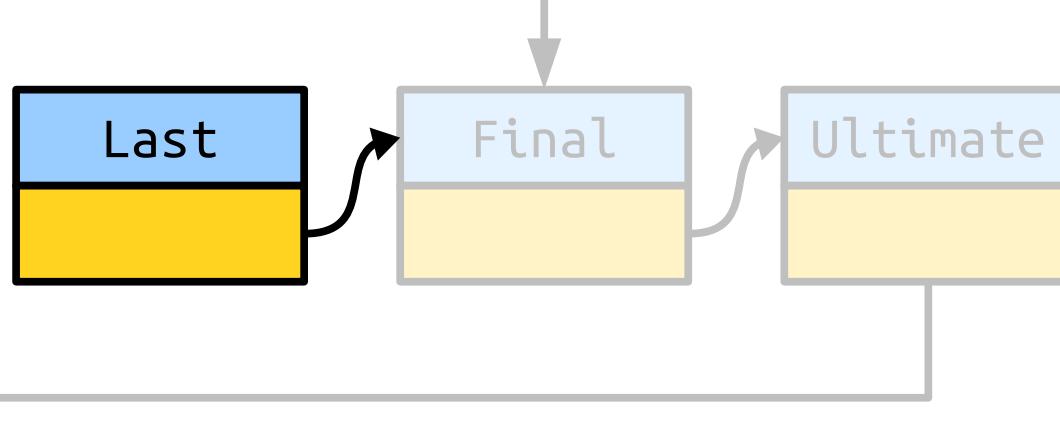
list



```
int main() {
    Cell* list = nullptr;
    appendTo(list, "Last");
    appendTo(list, "Final");
    appendTo(list, "Ultimate");
    appendTo(list, "Terminal");

    /* ... other listy things. ... */
}
```

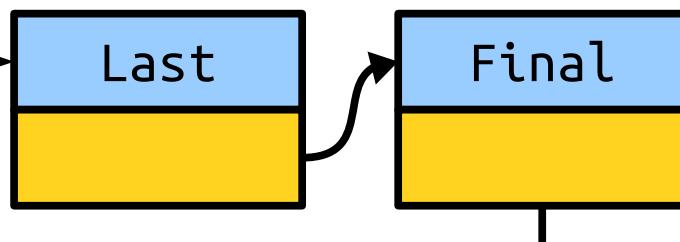
list



When passing in pointers by reference,  
be careful not to change the pointer  
unless you really want to change where it's  
pointing!

```
int main() {
    Cell* list = nullptr;
    appendTo(list, "Last");
    appendTo(list, "Final");
    appendTo(list, "Ultimate");
    appendTo(list, "Terminal");
    /* ... other listy things. ... */
}
```

list



```
int main() {
    Cell* list;
    appendTo(list, "Ultimate");
    appendTo(list, "Final");
    appendTo(list, "Last");
    /* ... other code */
}

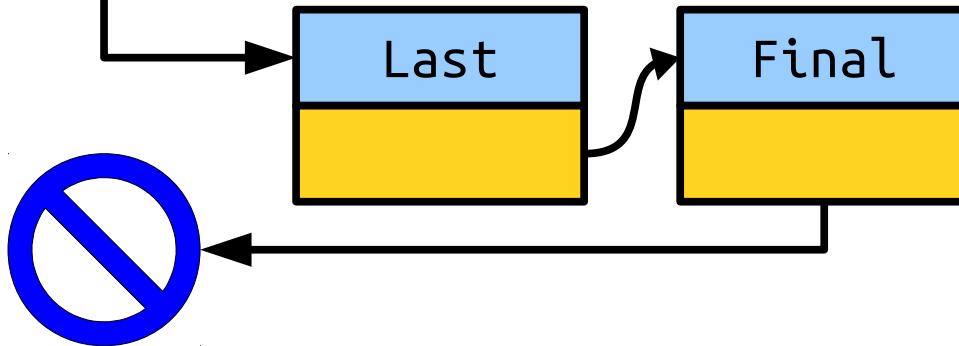
list
```

```
void appendTo(Cell*& list, const string& value) {
    Cell* cell = new Cell;
    cell->value = value;
    cell->next = nullptr;

    Cell* end = list;
    while (end != nullptr && end->next != nullptr) {
        end = end->next;
    }

    if (list == nullptr) {
        list = cell;
    } else {
        end->next = cell;
    }
}
```

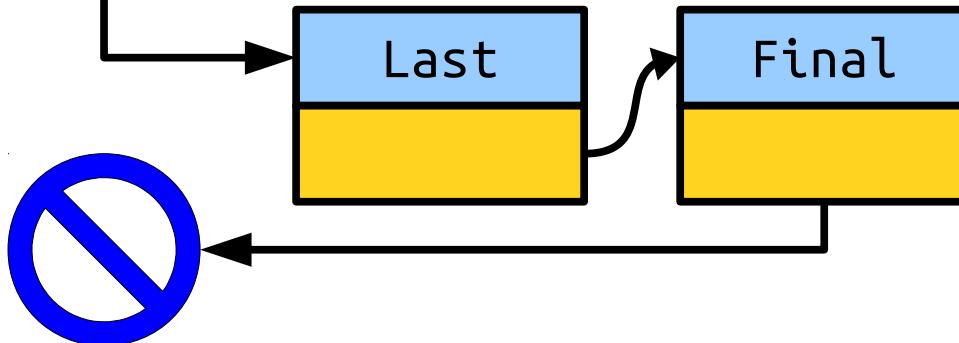
value  
Ultimate



```
int main() {
    Cell* list;
    appendTo(list, "Ultimate");
    appendTo(list, "Final");
    appendTo(list, "Last");
    appendTo(list, "Initial");
    /* ... other code */
}

list
value
Ultimate
```

The code shows a function `appendTo` that adds a new `Cell` node to the end of a linked list. The function takes a reference to the list head and a string value. It creates a new `Cell` node, sets its value, and points its `next` pointer to `nullptr`. Then it iterates through the list until it finds the last node (where `end->next == nullptr`). If the list is empty (`list == nullptr`), it sets `list` to the new node. Otherwise, it sets the `next` pointer of the last node to the new node.



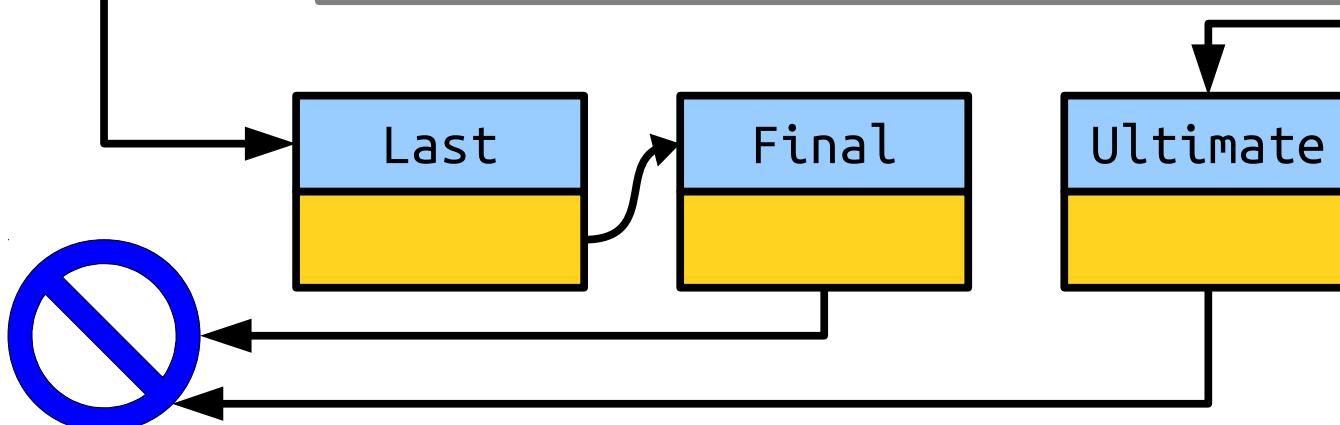
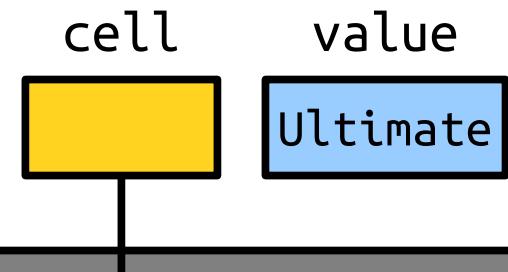
```
int main() {
    Cell* list;
    appendTo(list, "Ultimate");
    appendTo(list, "Final");
    appendTo(list, "Last");
    /* ... other code ...
    }

    list
```

```
void appendTo(Cell*& list, const string& value) {
    Cell* cell = new Cell;
    cell->value = value;
    cell->next = nullptr;
```

```
    Cell* end = list;
    while (end != nullptr && end->next != nullptr) {
        end = end->next;
    }

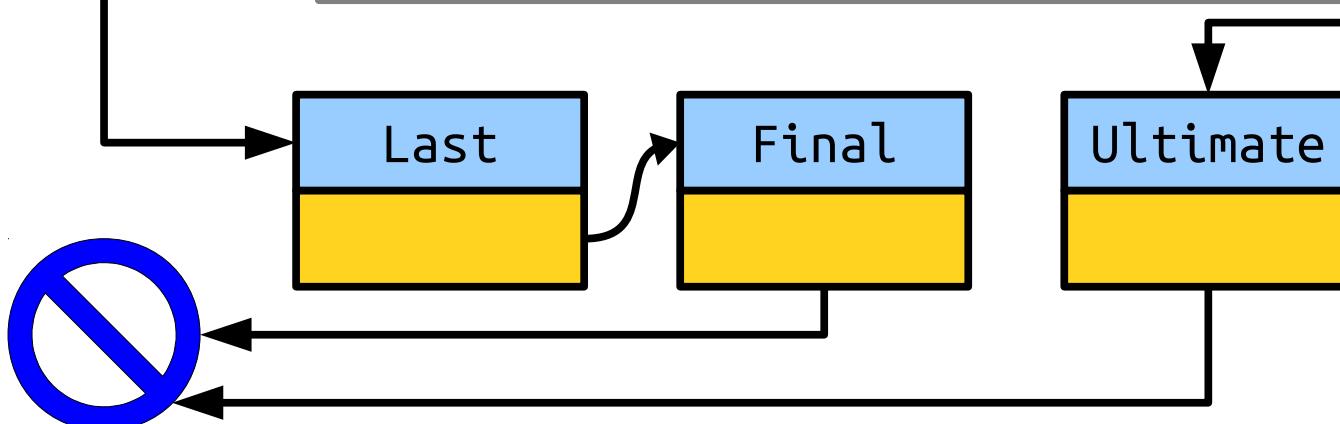
    if (list == nullptr) {
        list = cell;
    } else {
        end->next = cell;
    }
}
```



```
int main() {
    Cell* list;
    appendT
    appendT
    appendT
    appendT
    appendT
    /* ... ot
}
list
```

```
void appendTo(Cell*& list, const string& value) {
    Cell* cell = new Cell;
    cell->value = value;
    cell->next = nullptr;
    Cell* end = list;
    while (end != nullptr && end->next != nullptr) {
        end = end->next;
    }
    if (list == nullptr) {
        list = cell;
    } else {
        end->next = cell;
    }
}
```

```
cell      value
          Ultimate
```



```
int main() {
    Cell* list;
    appendT
    appendT
    appendT
    appendT
    appendT
    /* ... ot
}
list
```

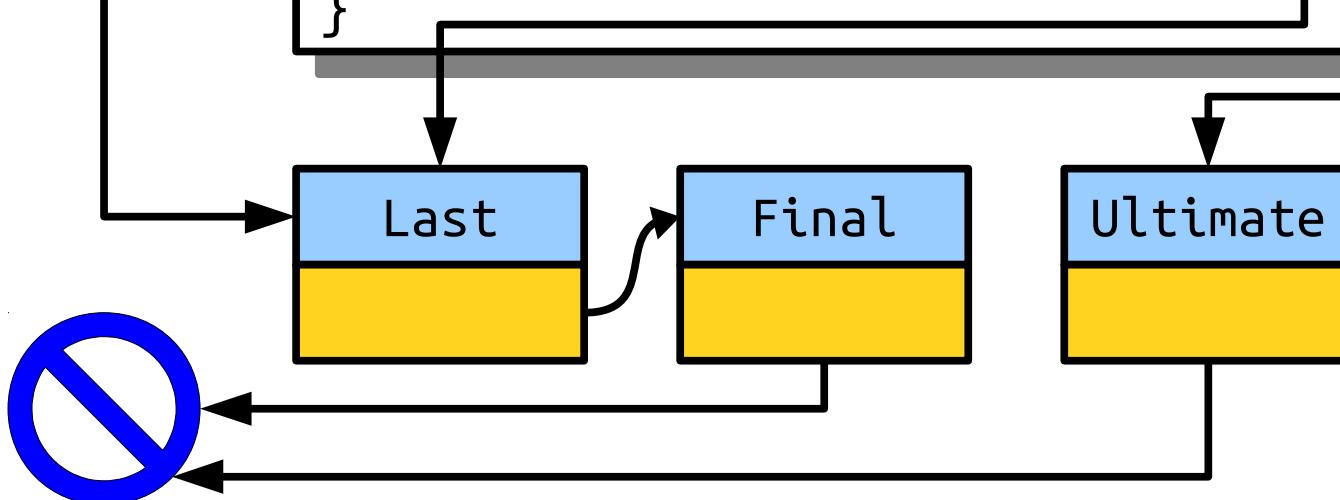
```
void appendTo(Cell*& list, const string& value) {
    Cell* cell = new Cell;
    cell->value = value;
    cell->next = nullptr;
    Cell* end = list;
    while (end != nullptr && end->next != nullptr) {
        end = end->next;
    }
    if (list == nullptr) {
        list = cell;
    } else {
        end->next = cell;
    }
}
```

end

cell

value

Ultimate



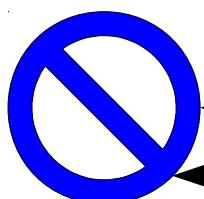
```
int main() {
    Cell* list;
    appendT
    appendT
    appendT
    appendT
    appendT
    /* ... ot
}
list
```

```
void appendTo(Cell*& list, const string& value) {
    Cell* cell = new Cell;
    cell->value = value;
    cell->next = nullptr;

    Cell* end = list;
    while (end != nullptr && end->next != nullptr) {
        end = end->next;
    }

    if (list == nullptr) {
        list = cell;
    } else {
        end->next = cell;
    }
}
```

The diagram illustrates the state of pointers during the execution of the `appendTo` function. The variable `list` (yellow box) initially points to the `Last` cell (blue box). The `Last` cell's `next` pointer loops back to itself, and it also points to the `Final` cell (blue box). The `Final` cell points to the `Ultimate` cell (blue box). The `cell` pointer (blue box) is assigned to point to the `Ultimate` cell. The `value` parameter (blue box) contains the string "Ultimate". A red circle with a slash indicates that the current operation is prohibited or incorrect.



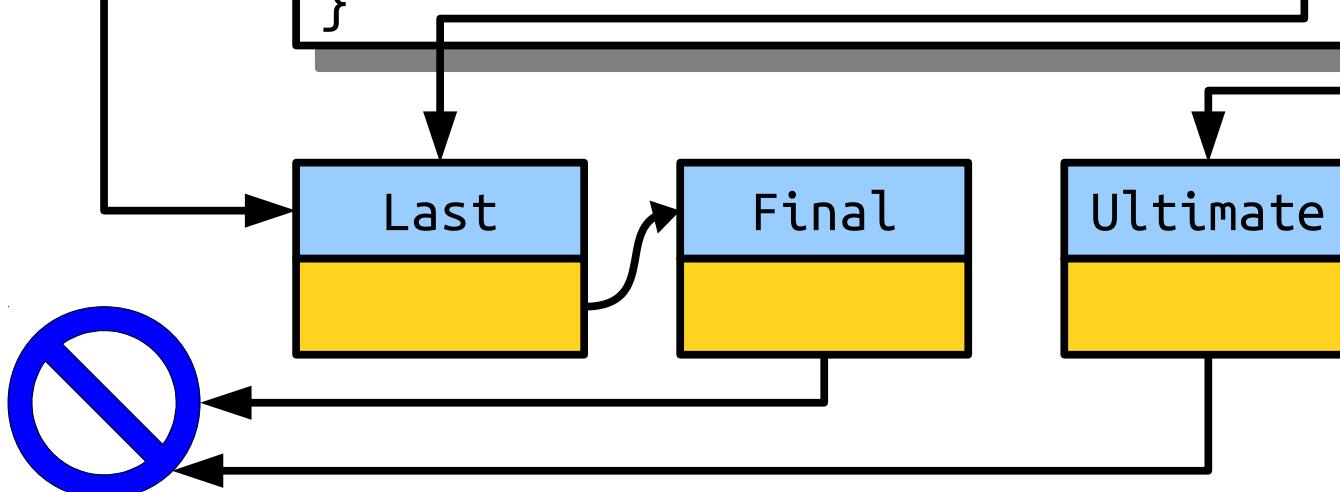
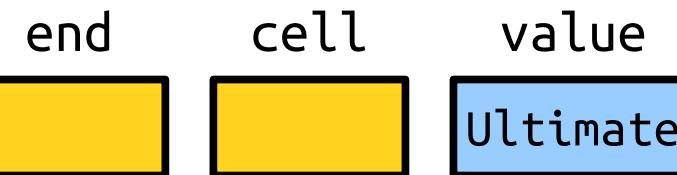
```
int main() {
    Cell* list;
    appendT
    appendT
    appendT
    appendT
    appendT
    /* ... other code */
}

list
```

```
void appendTo(Cell*& list, const string& value) {
    Cell* cell = new Cell;
    cell->value = value;
    cell->next = nullptr;

    Cell* end = list;
    while (end != nullptr && end->next != nullptr) {
        end = end->next;
    }

    if (list == nullptr) {
        list = cell;
    } else {
        end->next = cell;
    }
}
```



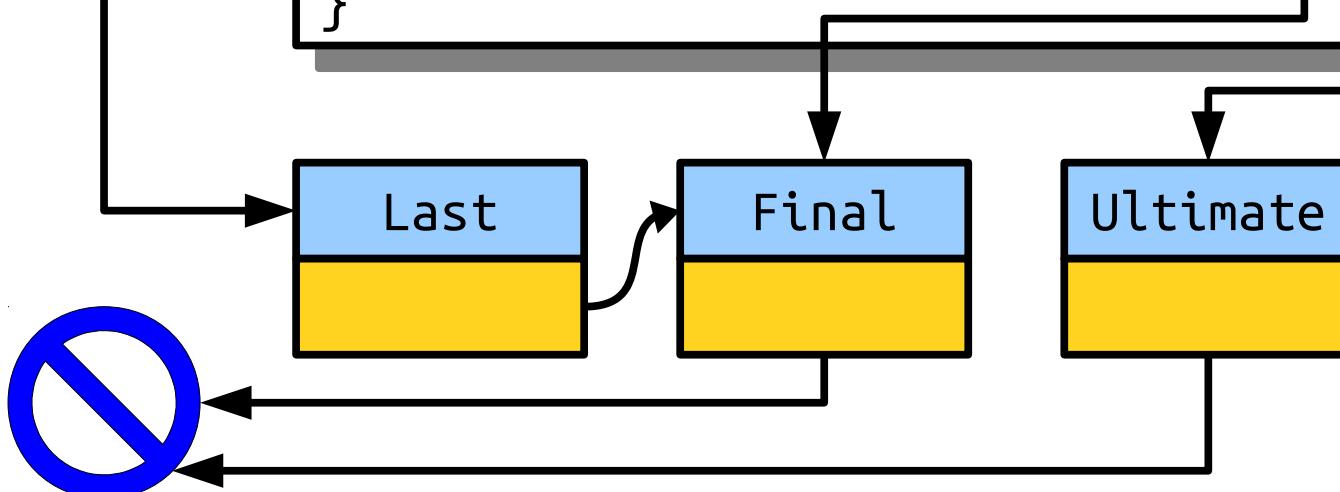
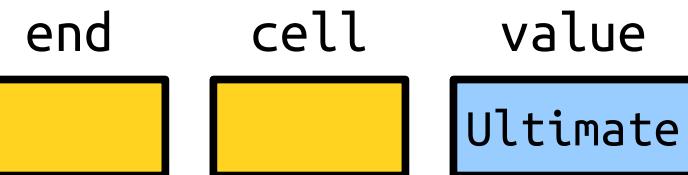
```
int main() {
    Cell* list;
    appendTo(list, "Ultimate");
    appendTo(list, "Final");
    appendTo(list, "Last");
    /* ... other code */
}

list
```

```
void appendTo(Cell*& list, const string& value) {
    Cell* cell = new Cell;
    cell->value = value;
    cell->next = nullptr;

    Cell* end = list;
    while (end != nullptr && end->next != nullptr) {
        end = end->next;
    }

    if (list == nullptr) {
        list = cell;
    } else {
        end->next = cell;
    }
}
```



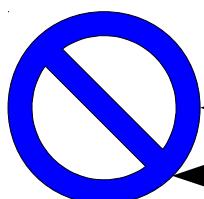
```
int main() {
    Cell* list;
    appendT
    appendT
    appendT
    appendT
    appendT
    /* ... ot
}
list
```

```
void appendTo(Cell*& list, const string& value) {
    Cell* cell = new Cell;
    cell->value = value;
    cell->next = nullptr;

    Cell* end = list;
    while (end != nullptr && end->next != nullptr) {
        end = end->next;
    }

    if (list == nullptr) {
        list = cell;
    } else {
        end->next = cell;
    }
}
```

```
graph TD
    list[Yellow] --> Last[Last]
    Last --> Final[Final]
    Final --> Ultimate[Ultimate]
    end[Blue] --> Ultimate
    cell[Blue] --> Ultimate
    value[Blue] --> Ultimate
```

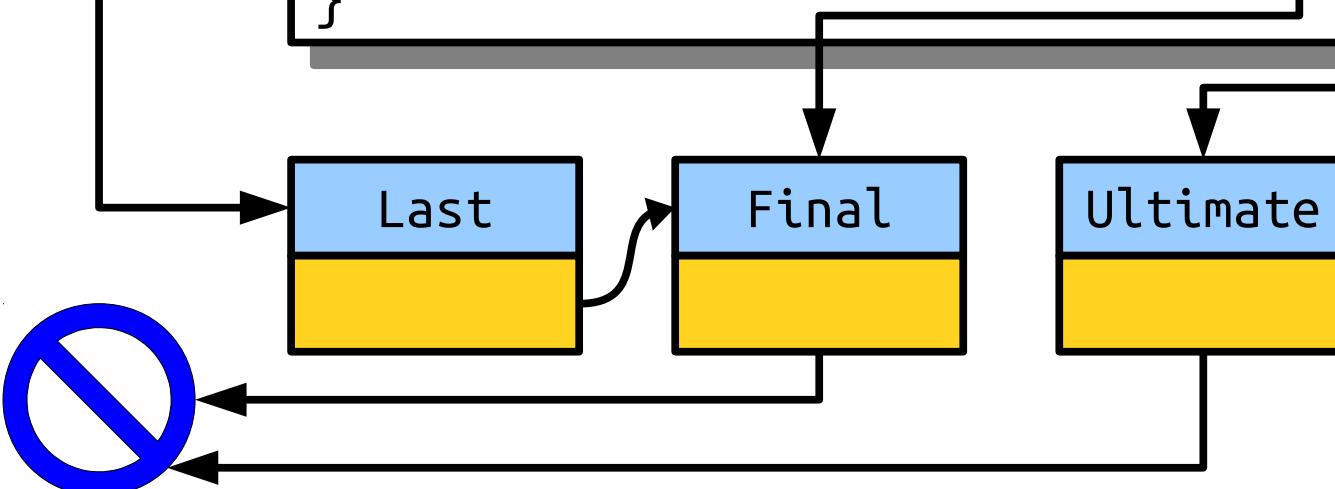
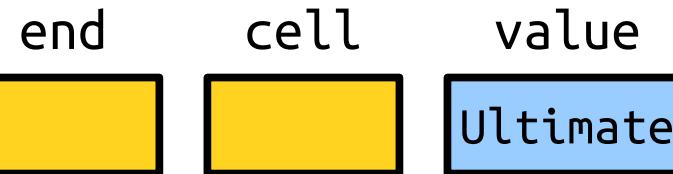


```
int main() {
    Cell* list;
    appendT
    appendT
    appendT
    appendT
    appendT
    /* ... ot
}
list
```

```
void appendTo(Cell*& list, const string& value) {
    Cell* cell = new Cell;
    cell->value = value;
    cell->next = nullptr;

    Cell* end = list;
    while (end != nullptr && end->next != nullptr) {
        end = end->next;
    }

    if (list == nullptr) {
        list = cell;
    } else {
        end->next = cell;
    }
}
```



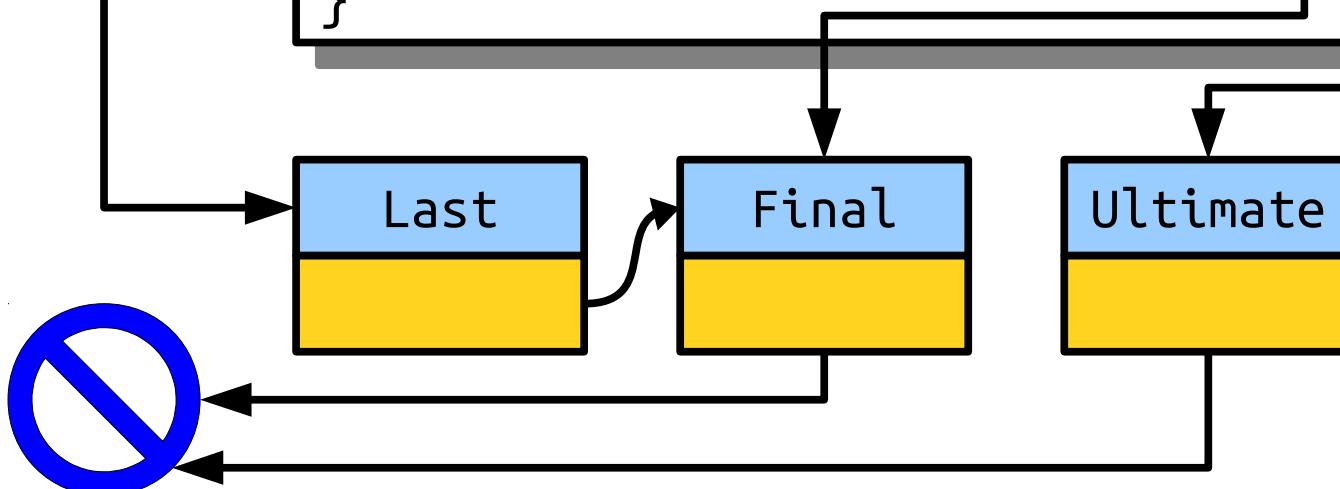
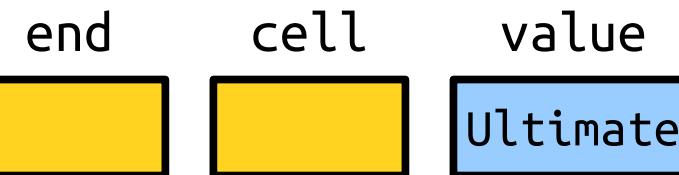
```
int main() {
    Cell* list;
    appendTo(list, "Ultimate");
    appendTo(list, "Final");
    appendTo(list, "Last");
    /* ... other code ...
    }

    list
```

```
void appendTo(Cell*& list, const string& value) {
    Cell* cell = new Cell;
    cell->value = value;
    cell->next = nullptr;

    Cell* end = list;
    while (end != nullptr && end->next != nullptr) {
        end = end->next;
    }

    if (list == nullptr) {
        list = cell;
    } else {
        end->next = cell;
    }
}
```



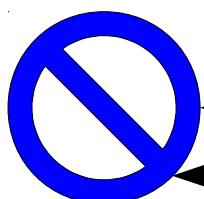
```
int main() {
    Cell* list;
    appendTo(list, "Ultimate");
    appendTo(list, "Final");
    appendTo(list, "Last");
    /* ... other code ...
    }

    list
```

```
void appendTo(Cell*& list, const string& value) {
    Cell* cell = new Cell;
    cell->value = value;
    cell->next = nullptr;

    Cell* end = list;
    while (end != nullptr && end->next != nullptr) {
        end = end->next;
    }

    if (list == nullptr) {
        list = cell;
    } else {
        end->next = cell;
    }
}
```



```
int main() {
    Cell* list;
    appendT
    appendT
    appendT
    appendT
    appendT
    /* ... ot
}
list
```

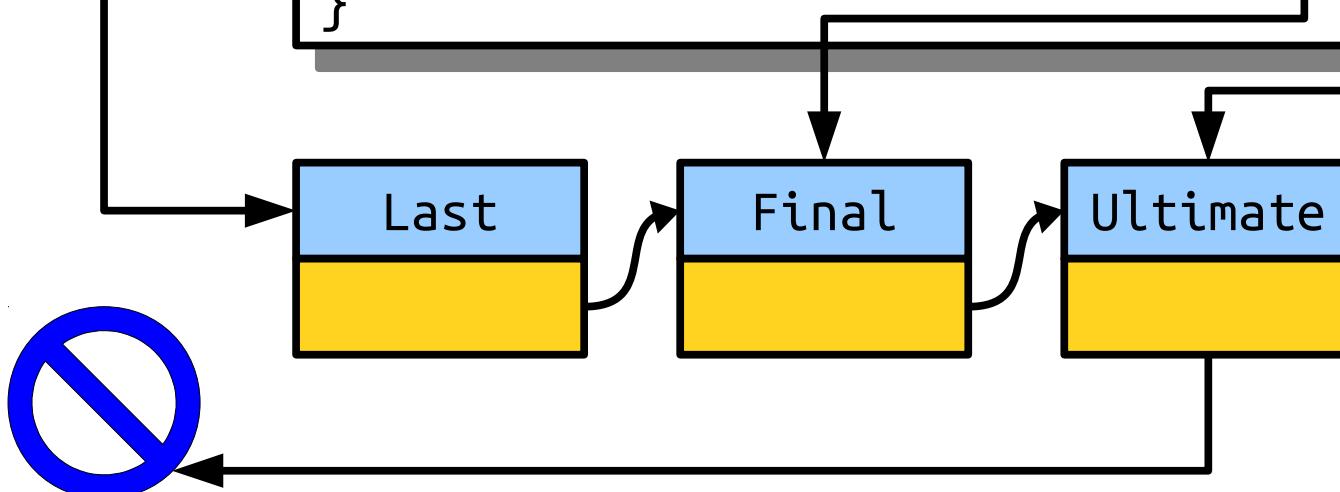
```
void appendTo(Cell*& list, const string& value) {
    Cell* cell = new Cell;
    cell->value = value;
    cell->next = nullptr;

    Cell* end = list;
    while (end != nullptr && end->next != nullptr) {
        end = end->next;
    }

    if (list == nullptr) {
        list = cell;
    } else {
        end->next = cell;
    }
}
```

Diagram illustrating pointer assignments:

- The variable `list` points to a yellow box labeled "Ultimate".
- The variable `end` points to a yellow box labeled "Ultimate".
- The variable `cell` points to a yellow box labeled "Ultimate".
- The variable `value` points to a blue box labeled "Ultimate".



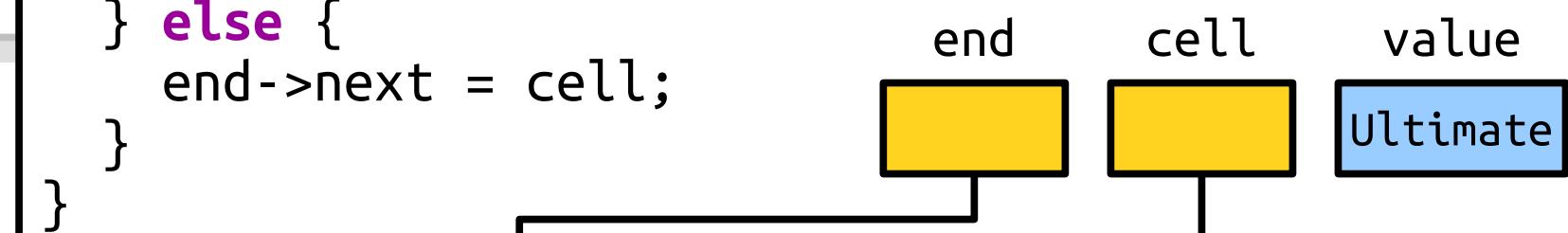
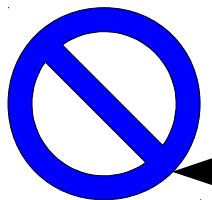
```
int main() {
    Cell* list;
    appendTo(list, "Ultimate");
    appendTo(list, "Final");
    appendTo(list, "Last");
    /* ... other code */
}

list
```

```
void appendTo(Cell*& list, const string& value) {
    Cell* cell = new Cell;
    cell->value = value;
    cell->next = nullptr;

    Cell* end = list;
    while (end != nullptr && end->next != nullptr) {
        end = end->next;
    }

    if (list == nullptr) {
        list = cell;
    } else {
        end->next = cell;
    }
}
```



```
int main() {
    Cell* list = nullptr;
    appendTo(list, "Last");
    appendTo(list, "Final");
    appendTo(list, "Ultimate");
    appendTo(list, "Terminal");

    /* ... other listy things. ... */
}
```

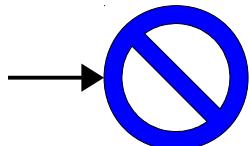
list



# What Went Wrong (Yet Again)?

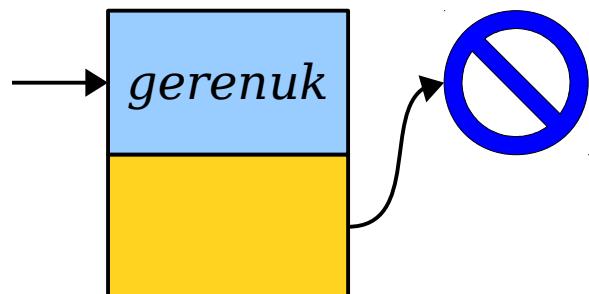
# Appending to a List

- What is the big-O complexity of appending to the back of a linked list using our algorithm?
- **Answer:  $O(n)$** , where  $n$  is the number of elements in the list, since we have to find the last position each time.



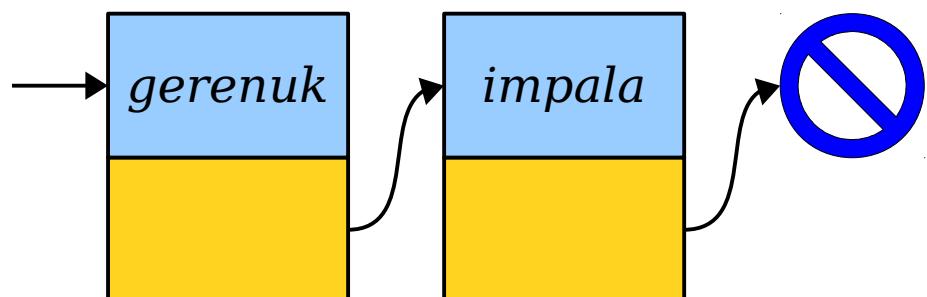
# Appending to a List

- What is the big-O complexity of appending to the back of a linked list using our algorithm?
- **Answer:  $O(n)$** , where  $n$  is the number of elements in the list, since we have to find the last position each time.



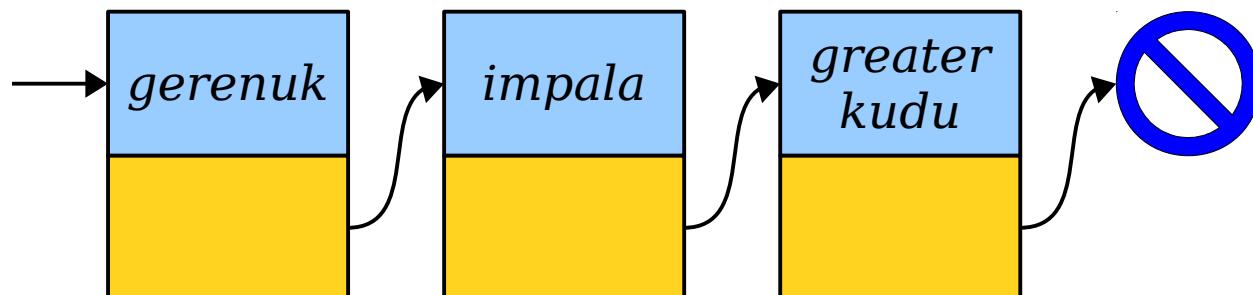
# Appending to a List

- What is the big-O complexity of appending to the back of a linked list using our algorithm?
- **Answer:  $O(n)$** , where  $n$  is the number of elements in the list, since we have to find the last position each time.



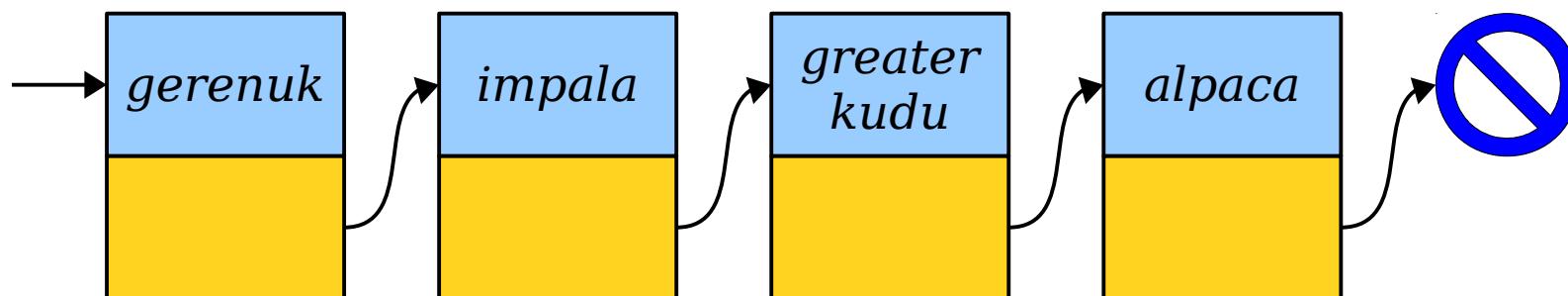
# Appending to a List

- What is the big-O complexity of appending to the back of a linked list using our algorithm?
- **Answer:  $O(n)$** , where  $n$  is the number of elements in the list, since we have to find the last position each time.



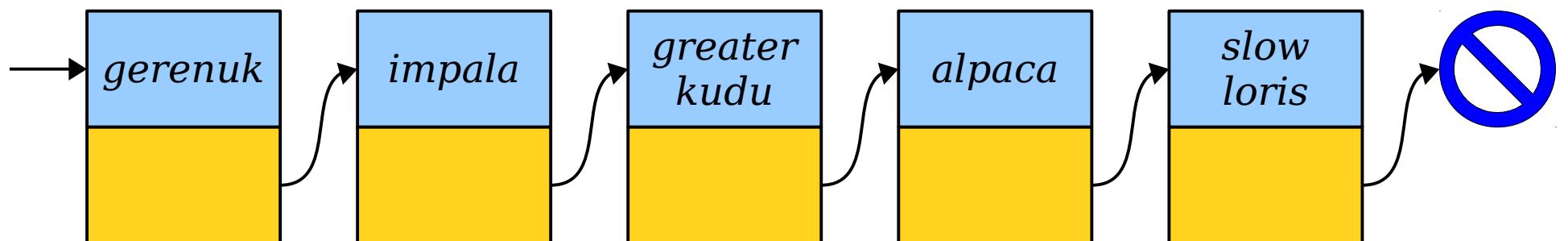
# Appending to a List

- What is the big-O complexity of appending to the back of a linked list using our algorithm?
- **Answer:  $O(n)$** , where  $n$  is the number of elements in the list, since we have to find the last position each time.



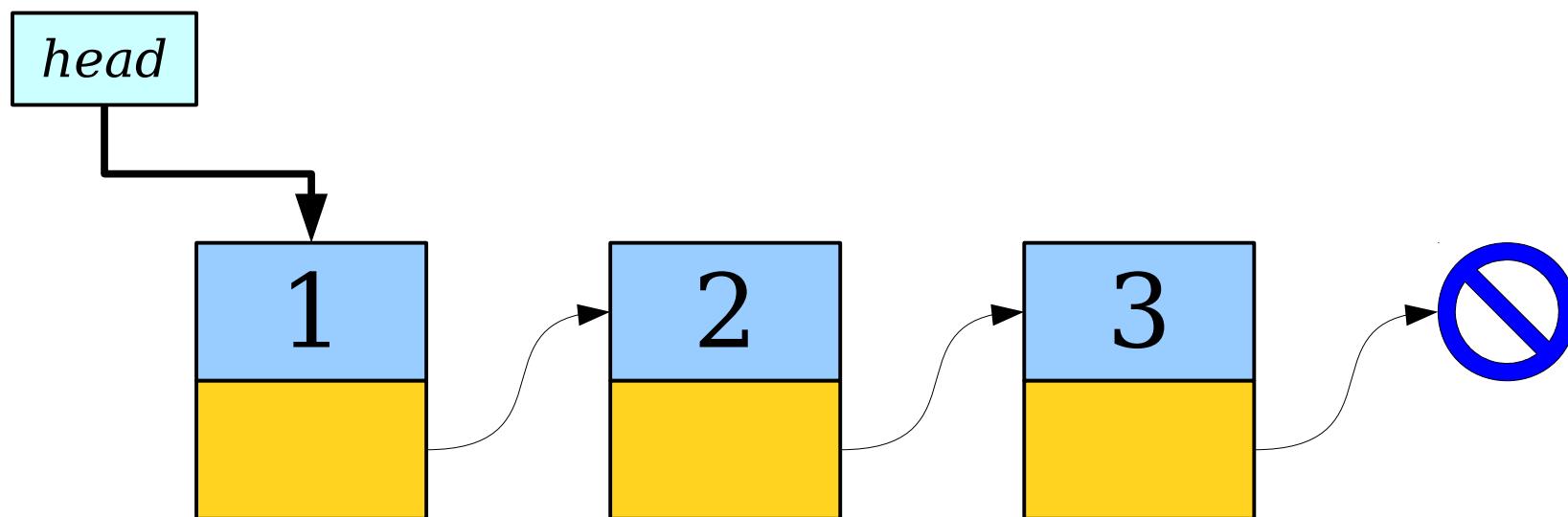
# Appending to a List

- What is the big-O complexity of appending to the back of a linked list using our algorithm?
- **Answer: O(n)**, where  $n$  is the number of elements in the list, since we have to find the last position each time.



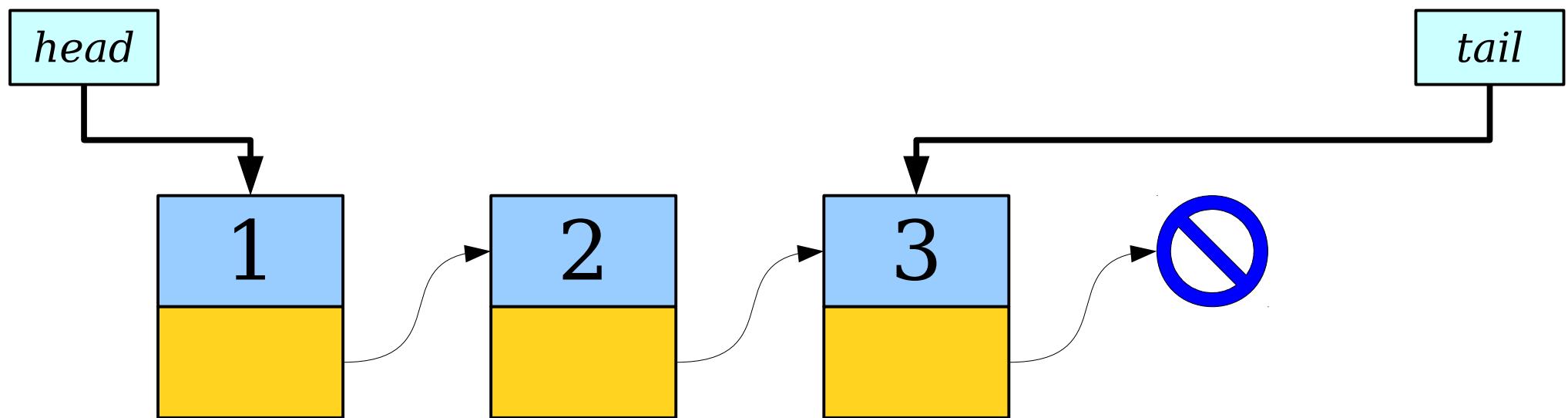
# Tail Pointers

- A ***tail pointer*** is a pointer to the last element of a linked list.
- Tail pointers make it easy and efficient to add new elements to the back of a linked list.



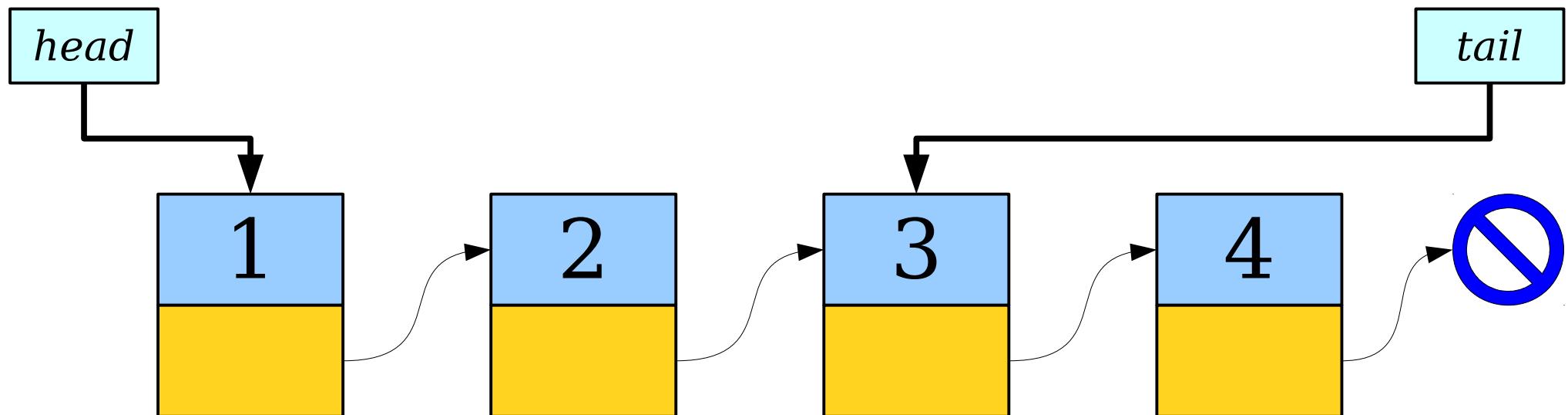
# Tail Pointers

- A ***tail pointer*** is a pointer to the last element of a linked list.
- Tail pointers make it easy and efficient to add new elements to the back of a linked list.



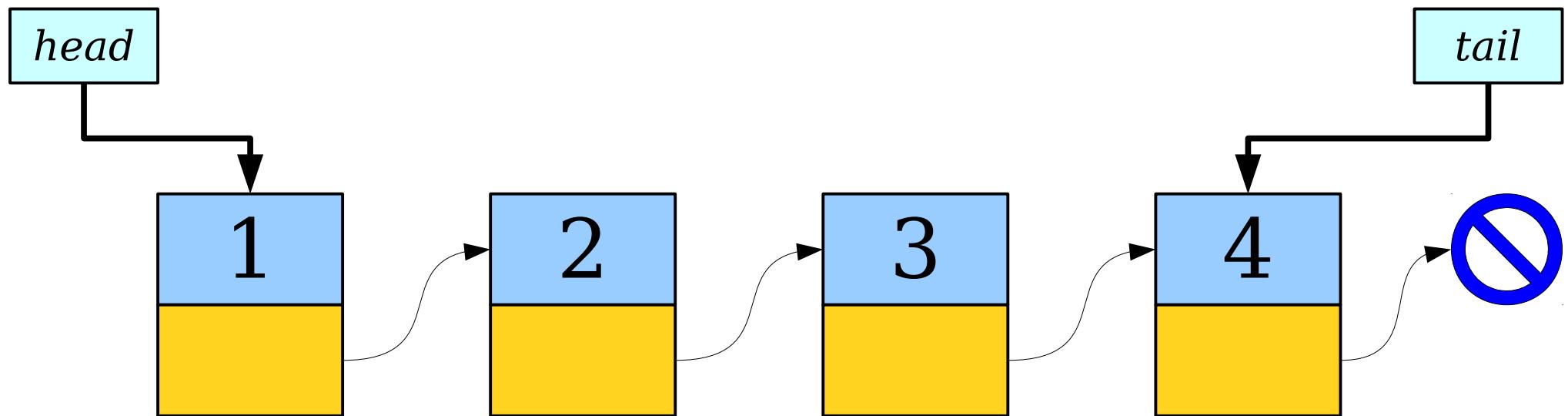
# Tail Pointers

- A ***tail pointer*** is a pointer to the last element of a linked list.
- Tail pointers make it easy and efficient to add new elements to the back of a linked list.



# Tail Pointers

- A ***tail pointer*** is a pointer to the last element of a linked list.
- Tail pointers make it easy and efficient to add new elements to the back of a linked list.



# Appending Things Quickly

- **Case 1:** The list is empty.

head

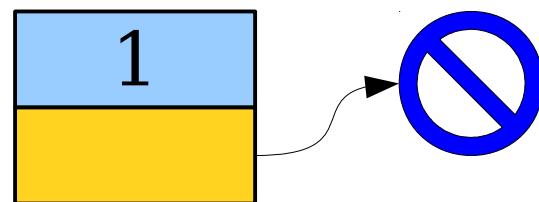
tail

# Appending Things Quickly

- **Case 1:** The list is empty.

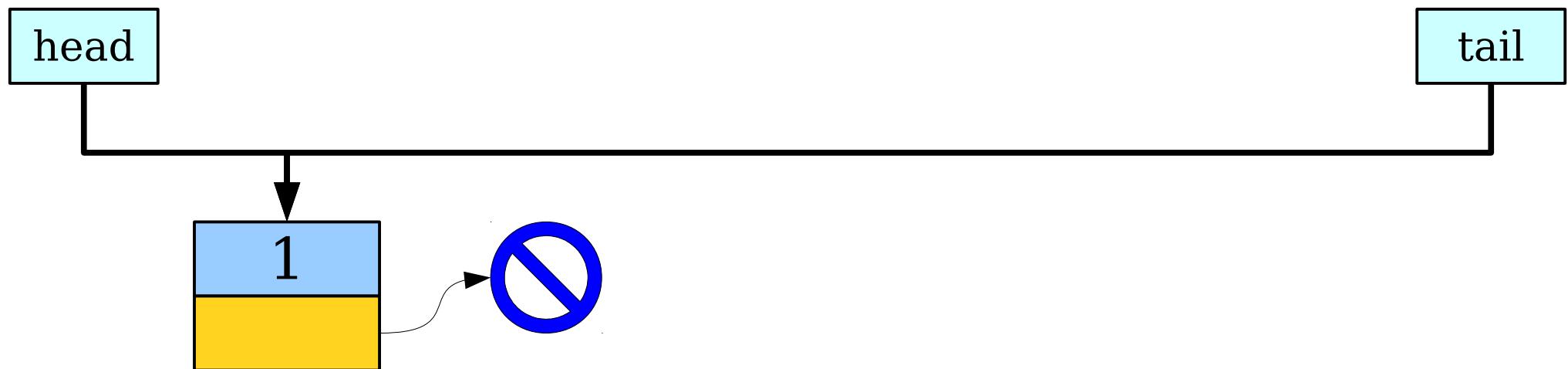
head

tail



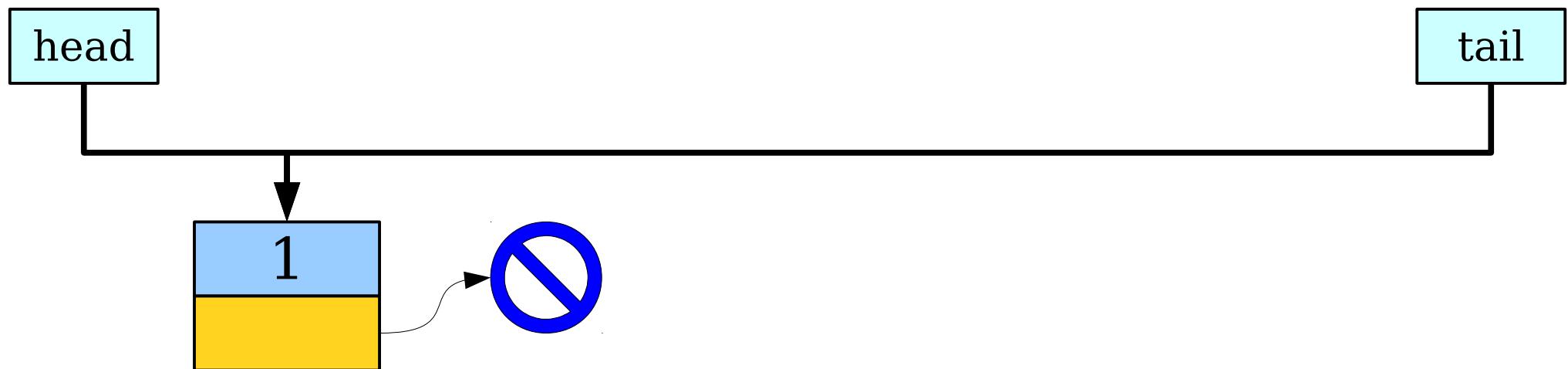
# Appending Things Quickly

- **Case 1:** The list is empty.

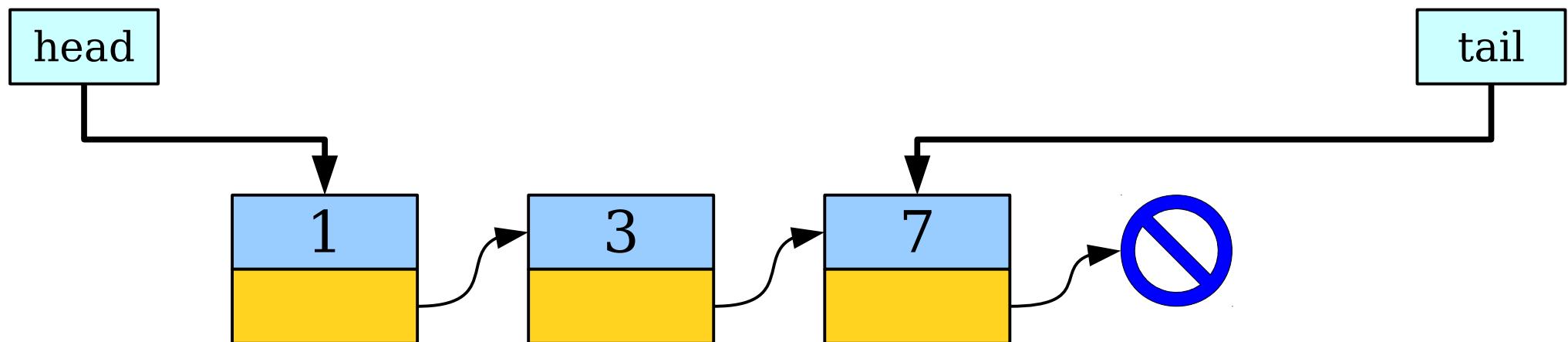


# Appending Things Quickly

- **Case 1:** The list is empty.

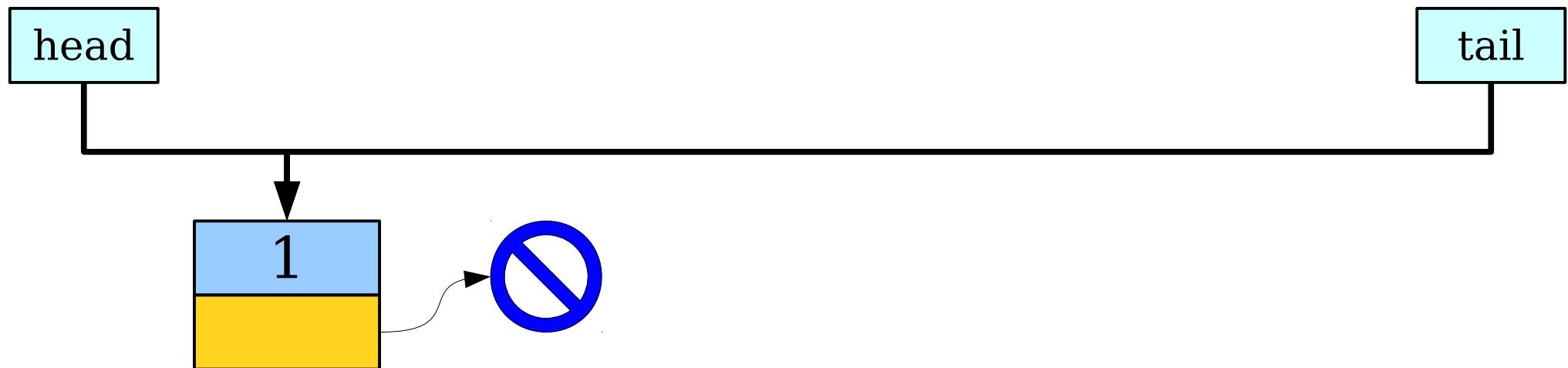


- **Case 2:** The list is not empty.

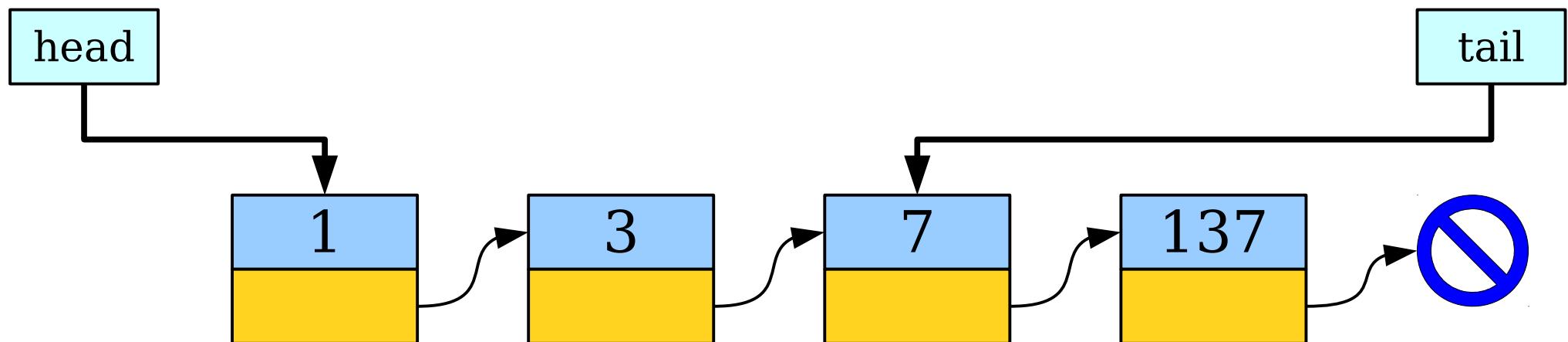


# Appending Things Quickly

- **Case 1:** The list is empty.

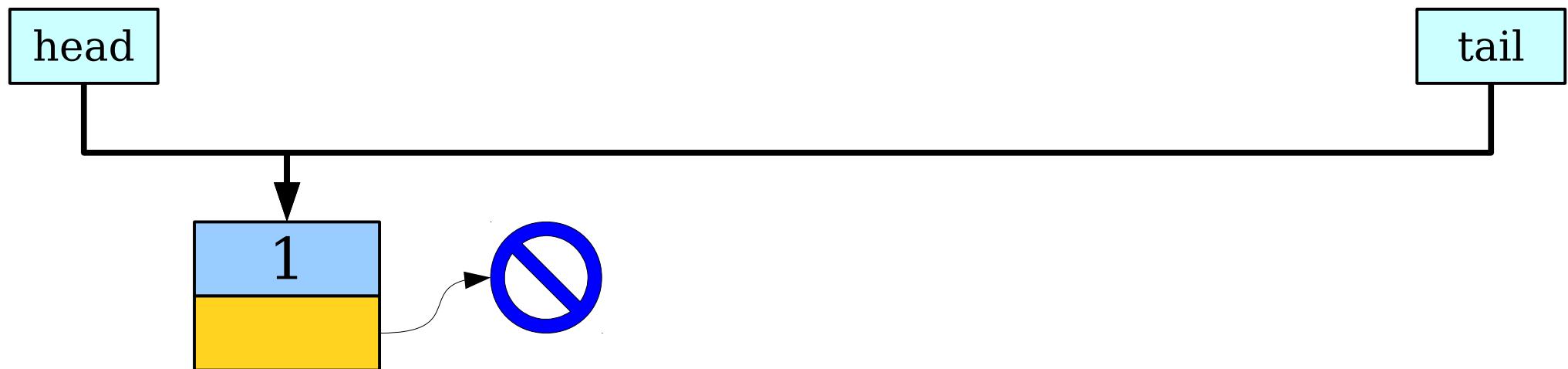


- **Case 2:** The list is not empty.

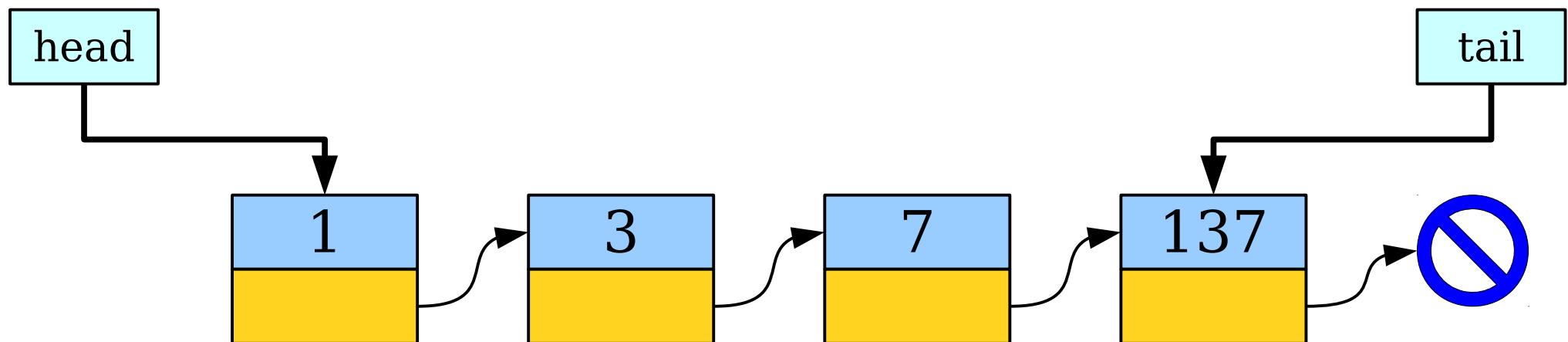


# Appending Things Quickly

- **Case 1:** The list is empty.



- **Case 2:** The list is not empty.



## ***Coda:*** Doubly-Linked Lists

# Doubly-Linked Lists

- There's a strange asymmetry in a linked list: you can easily move forward in a list, but there's no easy way to move backwards.
- A ***doubly-linked list*** is a list where each cell stores two pointers: one to the next element in the list, and one to the previous element.



# Doubly-Linked Lists

- In many cases, doubly-linked lists are similar to singly-linked lists.
- For example, if you're just moving from the left to the right, then code on doubly-linked lists looks really similar to code on singly-linked lists.



# Doubly-Linked Lists

- In many cases, doubly-linked lists are similar to singly-linked lists.
- For example, if you're just moving from the left to the right, then code on doubly-linked lists looks really similar to code on singly-linked lists.

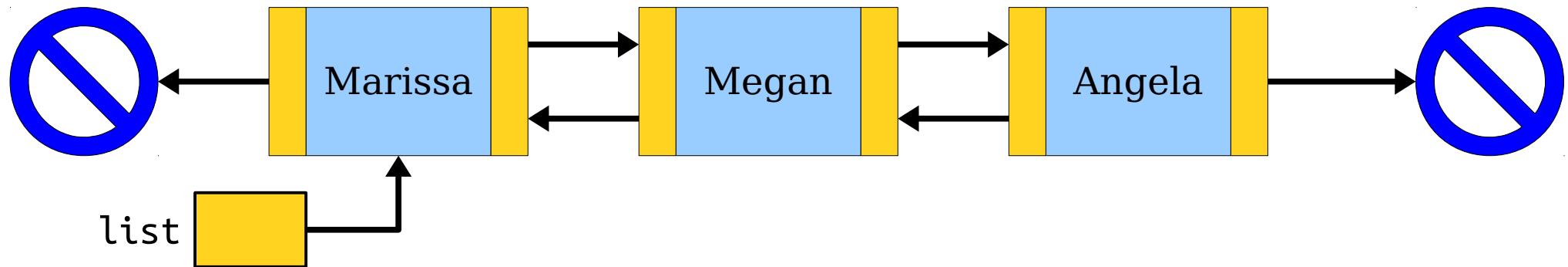
```
Cell* list = /* first cell */;
```



# Doubly-Linked Lists

- In many cases, doubly-linked lists are similar to singly-linked lists.
- For example, if you're just moving from the left to the right, then code on doubly-linked lists looks really similar to code on singly-linked lists.

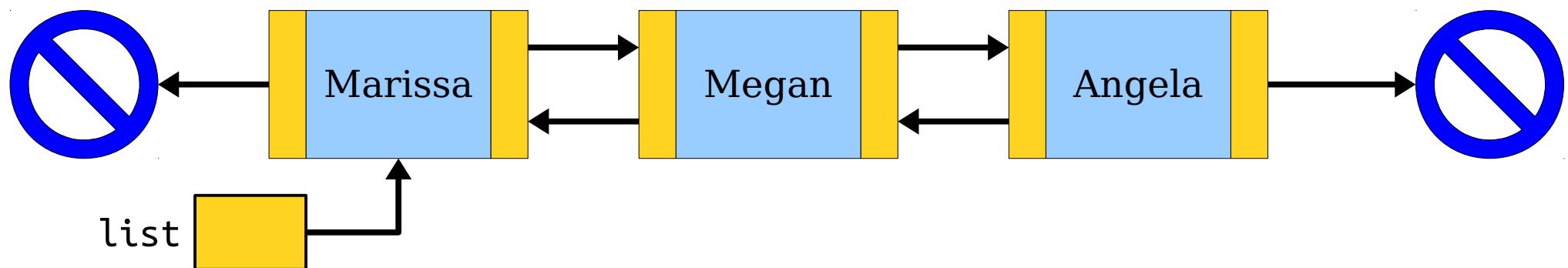
```
Cell* list = /* first cell */;
```



# Doubly-Linked Lists

- In many cases, doubly-linked lists are similar to singly-linked lists.
- For example, if you're just moving from the left to the right, then code on doubly-linked lists looks really similar to code on singly-linked lists.

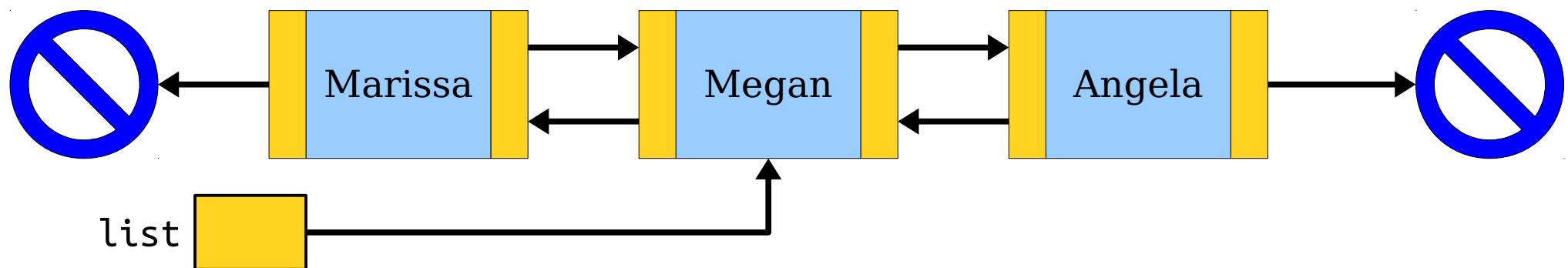
```
Cell* list = /* first cell */;  
list = list->next;
```



# Doubly-Linked Lists

- In many cases, doubly-linked lists are similar to singly-linked lists.
- For example, if you're just moving from the left to the right, then code on doubly-linked lists looks really similar to code on singly-linked lists.

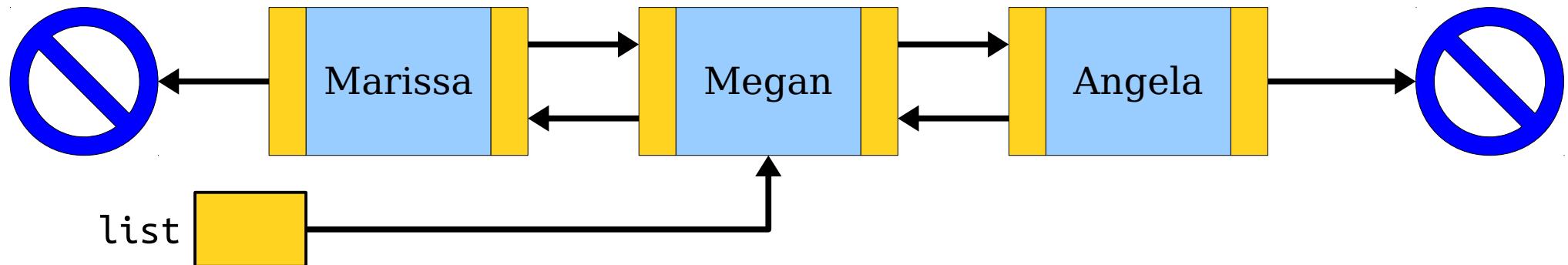
```
Cell* list = /* first cell */;  
list = list->next;
```



# Doubly-Linked Lists

- We can also move backwards in a doubly-linked list.
- Many algorithms are a lot easier to write if you can do this!

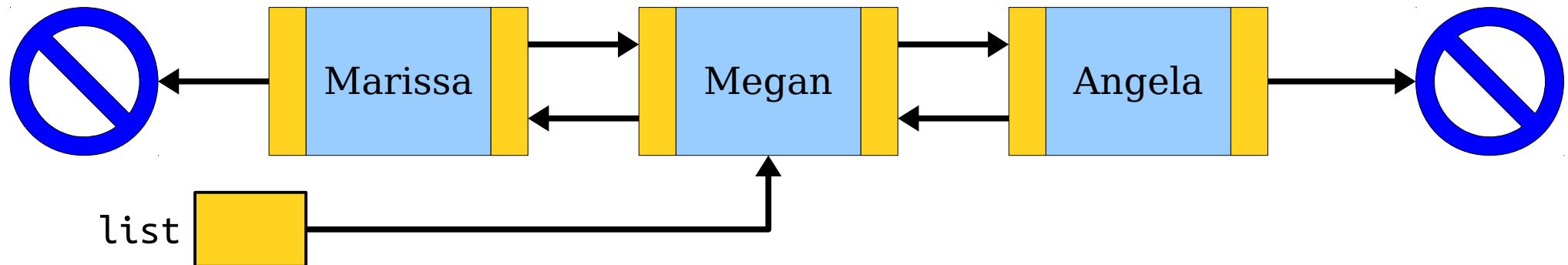
```
Cell* list = /* first cell */;  
list = list->next;
```



# Doubly-Linked Lists

- We can also move backwards in a doubly-linked list.
- Many algorithms are a lot easier to write if you can do this!

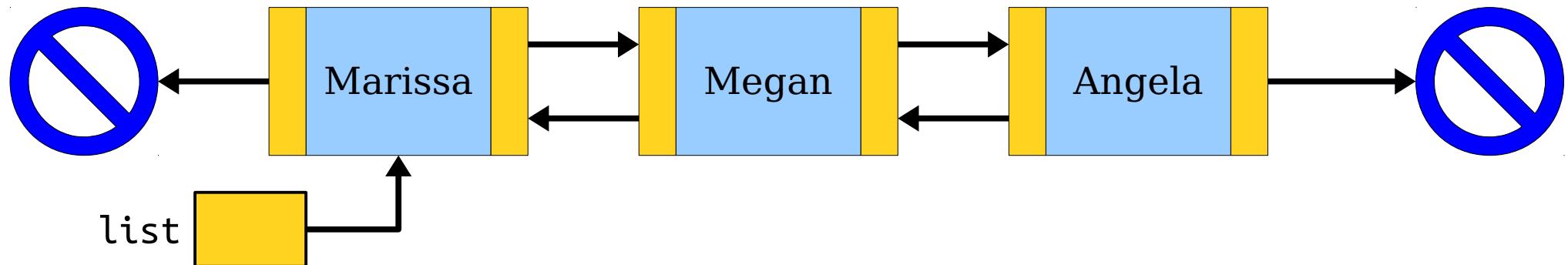
```
Cell* list = /* first cell */;  
list = list->next;  
list = list->prev;
```



# Doubly-Linked Lists

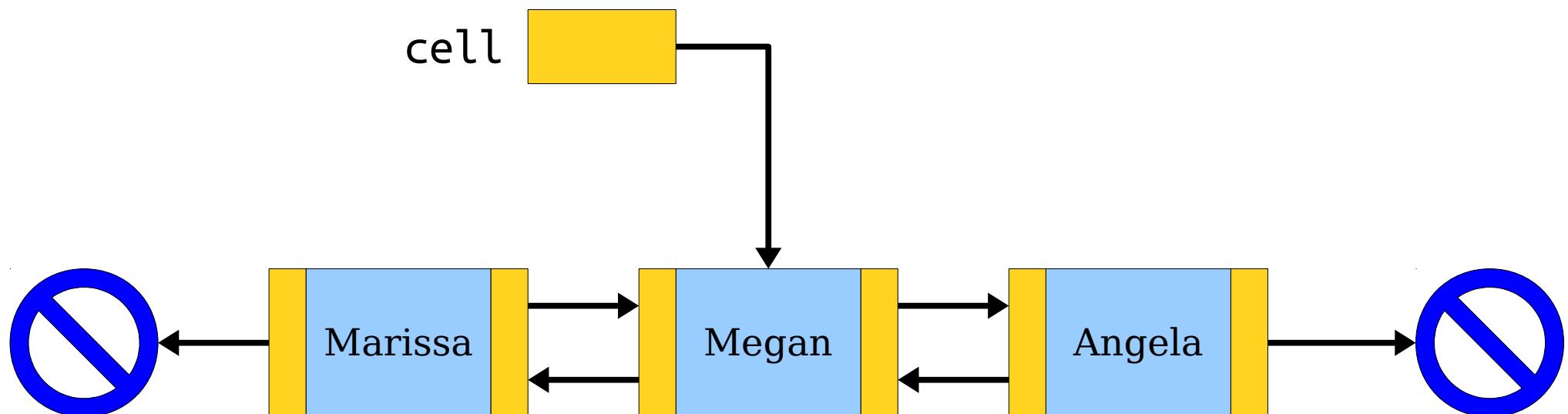
- We can also move backwards in a doubly-linked list.
- Many algorithms are a lot easier to write if you can do this!

```
Cell* list = /* first cell */;  
list = list->next;  
list = list->prev;
```



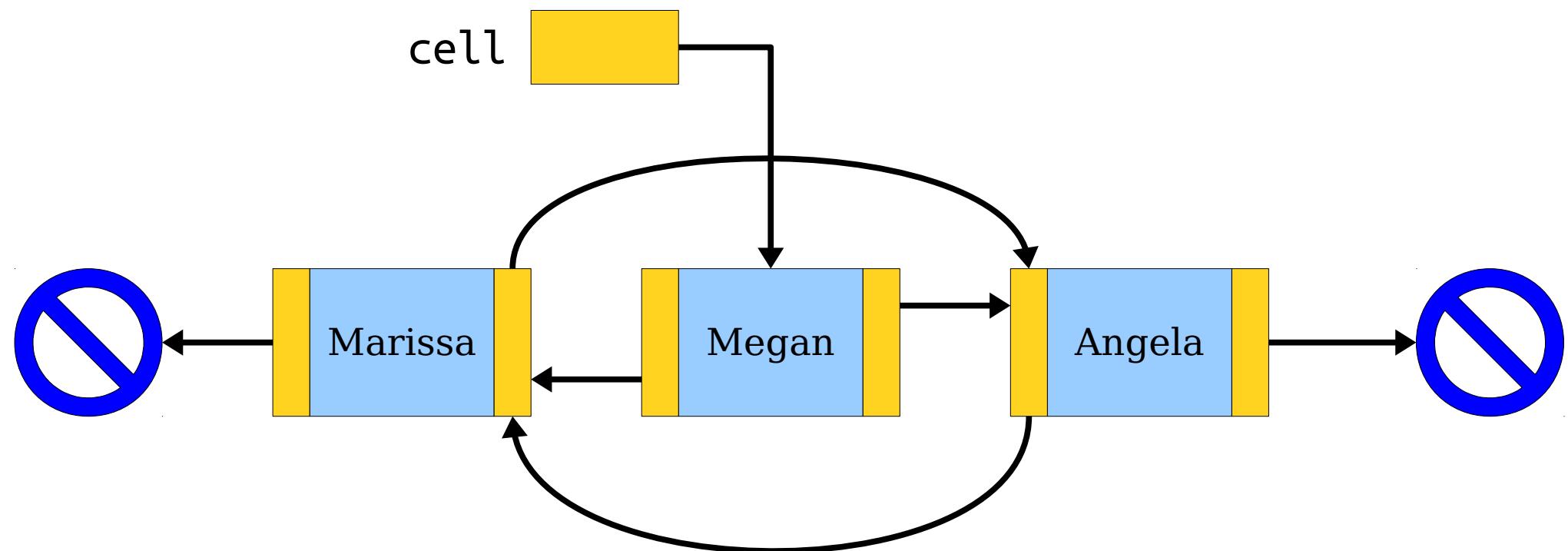
# Doubly-Linked Lists

- It's easy to remove a cell from a doubly-linked list: just wire the nodes next to it around it.
- (Don't forget to handle edge cases!)



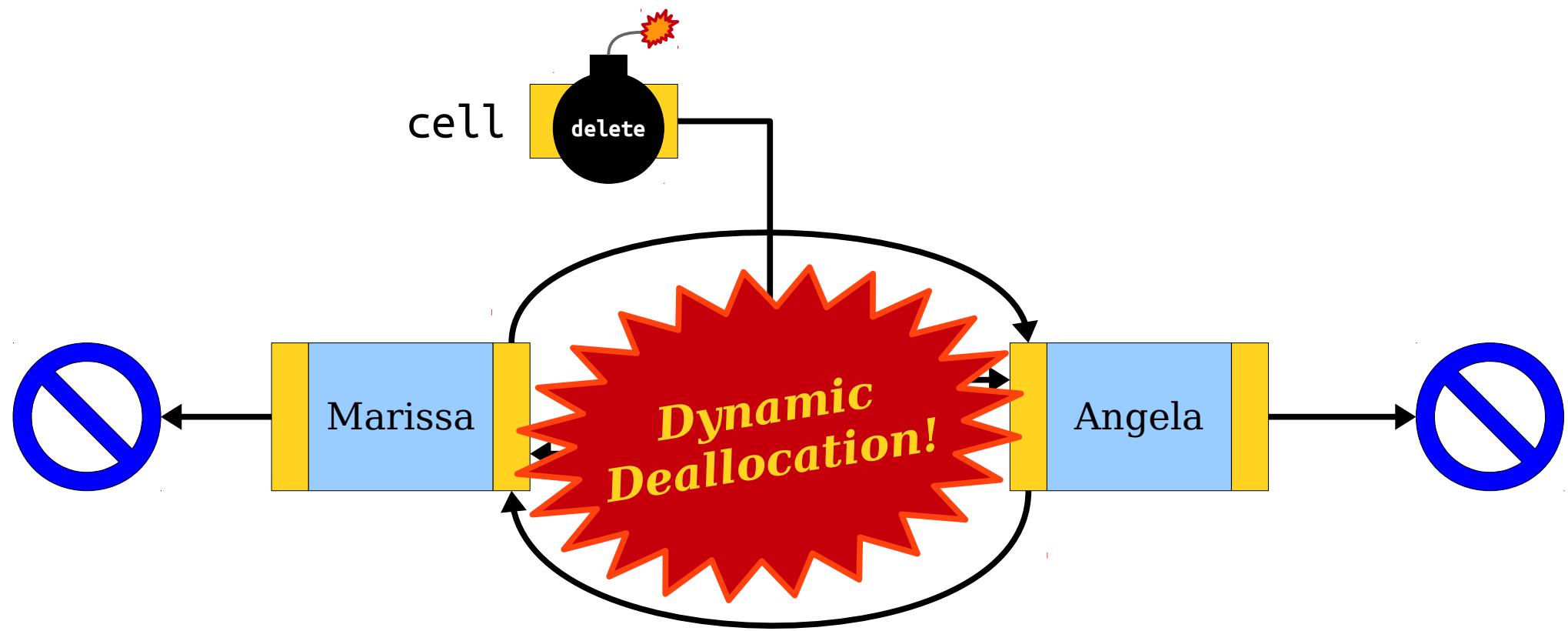
# Doubly-Linked Lists

- It's easy to remove a cell from a doubly-linked list: just wire the nodes next to it around it.
- (Don't forget to handle edge cases!)



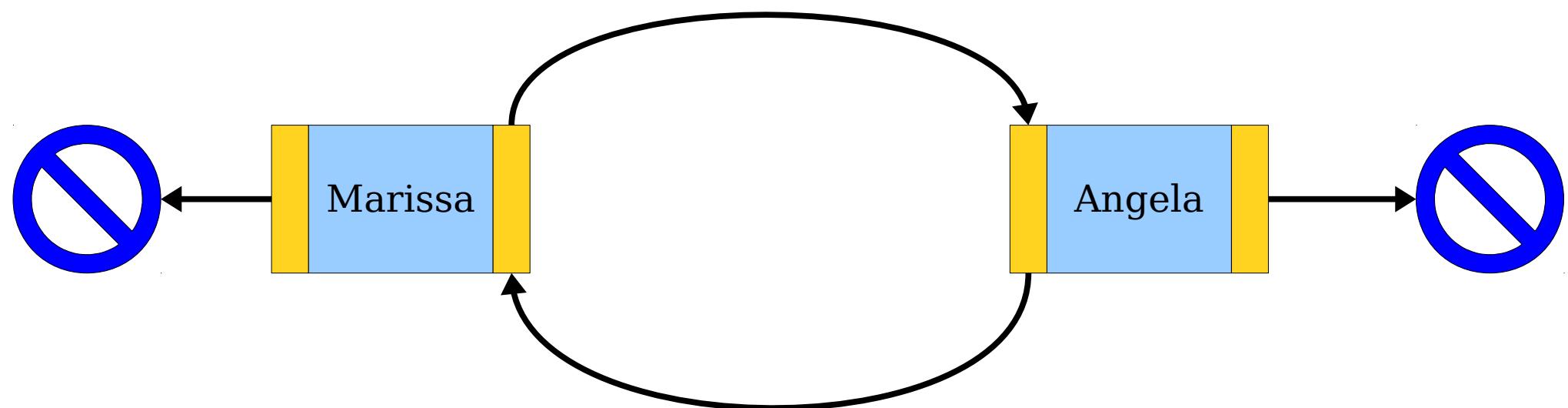
# Doubly-Linked Lists

- It's easy to remove a cell from a doubly-linked list: just wire the nodes next to it around it.
- (Don't forget to handle edge cases!)



# Doubly-Linked Lists

- It's easy to remove a cell from a doubly-linked list: just wire the nodes next to it around it.
- (Don't forget to handle edge cases!)



For more on doubly-linked lists, check  
***Section Handout 7*** and ***Chapter 13*** of  
the textbook.

# To Recap

- If you want a function to change *which object* a pointer points to, pass that pointer in by reference.
- When passing in pointers by reference, make sure not to change the pointer unless you really mean it.
- Tail pointers make it easy to find the end of a linked list - a handy tool to keep in mind!
- Doubly-linked lists have each cell store pointers to both the next and previous cells in the list. They're useful for when you need to remove out of a list.

# Your Action Items

- ***Read Chapter 13.***
  - It's all about different representations for data and the relative tradeoffs. And there's some great coverage of linked lists in there!
- ***Start Assignment 7.***
  - It's all about linked lists! Working through this is a great

# Next Time

- ***Tree Structures***
  - Representing branching structures in code.
- ***Binary Search Trees***
  - Maintaining order at a low cost!