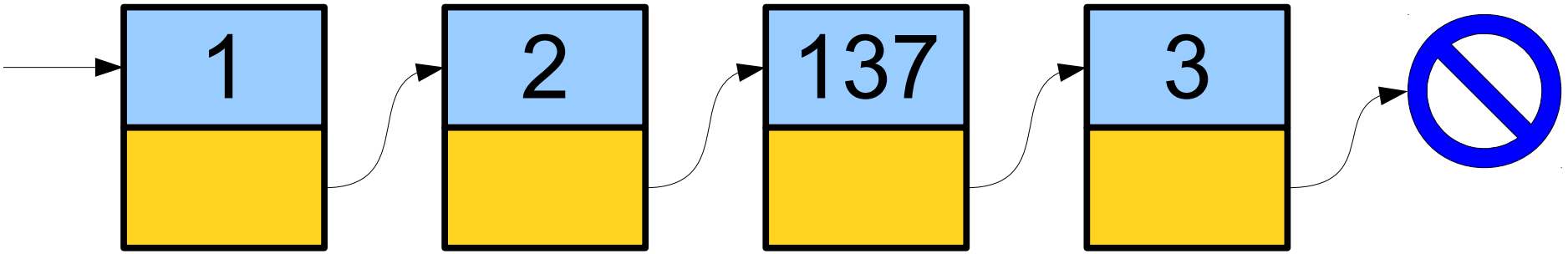
Linked Lists

Part Two

Recap from Last Time

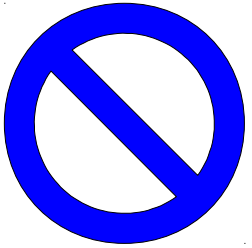
Linked Lists

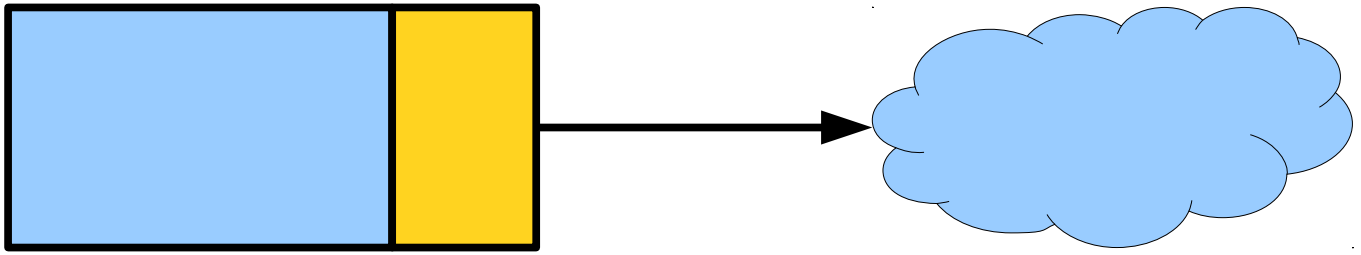
- A ***linked list*** is a data structure for storing a sequence of elements.
- Each element is stored separately from the rest.
- The elements are then chained together into a sequence.
- The end of the list is marked with some special indicator.



A Linked List is Either…

…an empty list, represented by **nullptr**, or…





a single linked list cell that points…

… at another linked list.

```
struct Cell {
    Type data;
    Cell* next;
};
```

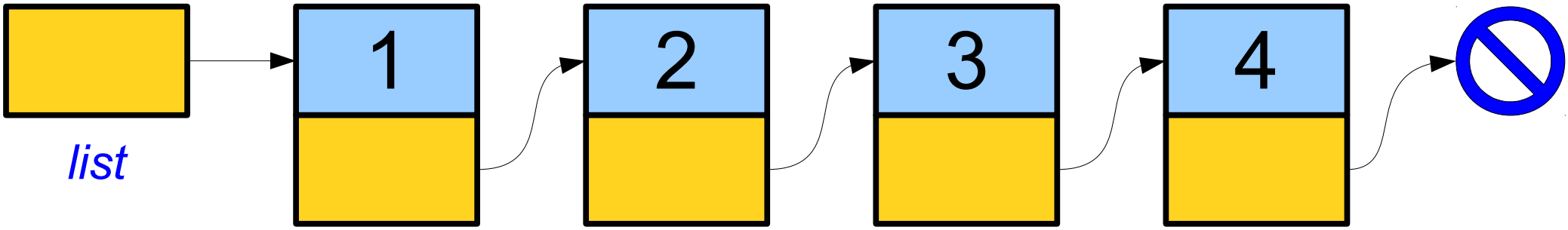
Processing Lists Recursively

Processing Lists Iteratively

Linked Lists, Iteratively

- You can navigate a linked list using a traditional `for` loop:

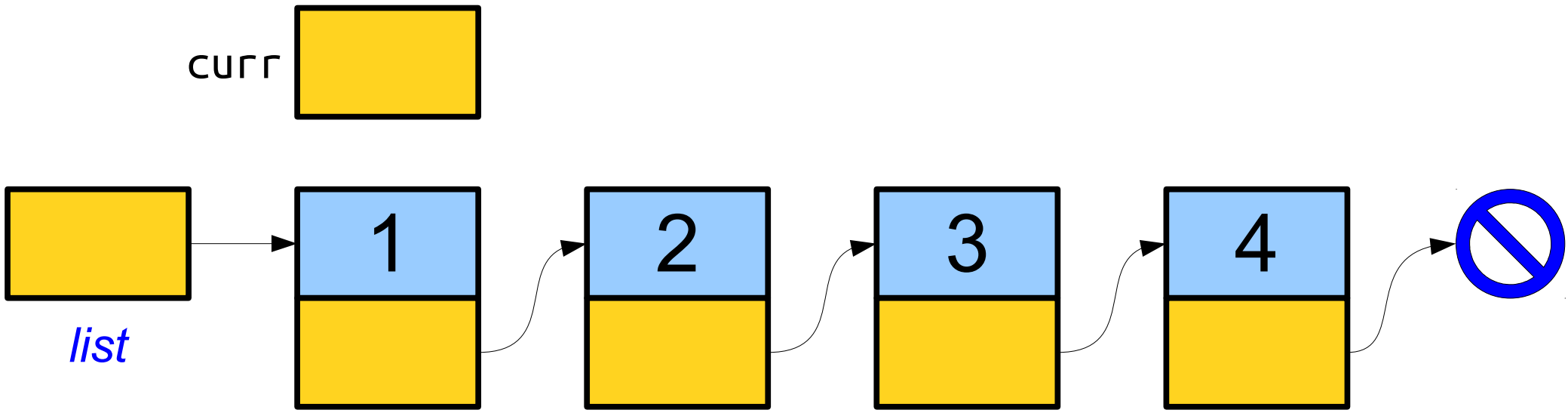
```
for (Cell* curr = list; curr != nullptr; curr = curr->next) {
    /* … do something with curr->value … */
}
```



Linked Lists, Iteratively

- You can navigate a linked list using a traditional `for` loop:

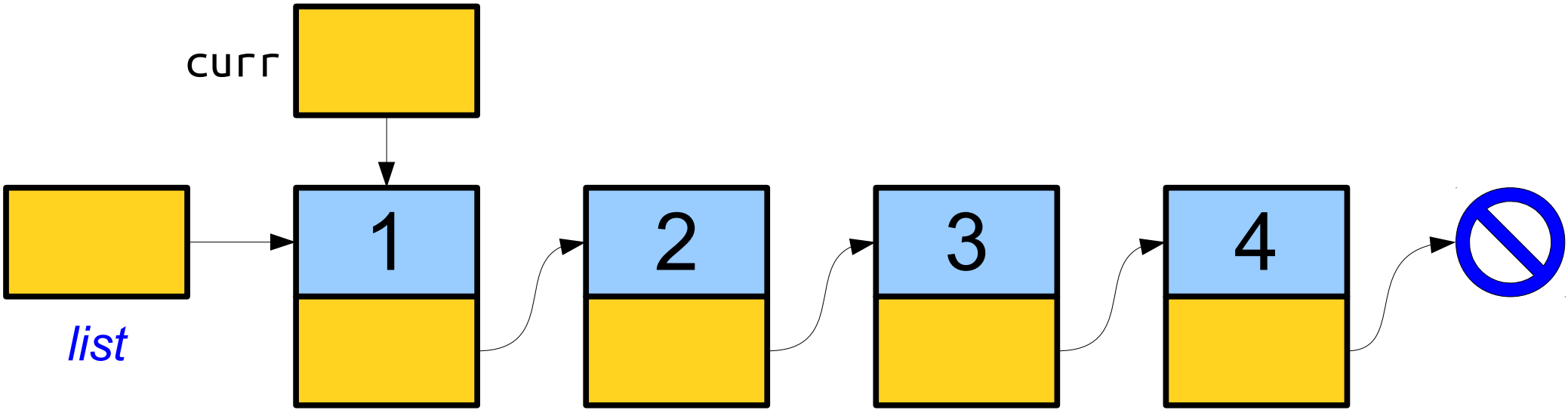
```
for (Cell* curr = list; curr != nullptr; curr = curr->next) {
    /* … do something with curr->value … */
}
```



Linked Lists, Iteratively

- You can navigate a linked list using a traditional `for` loop:

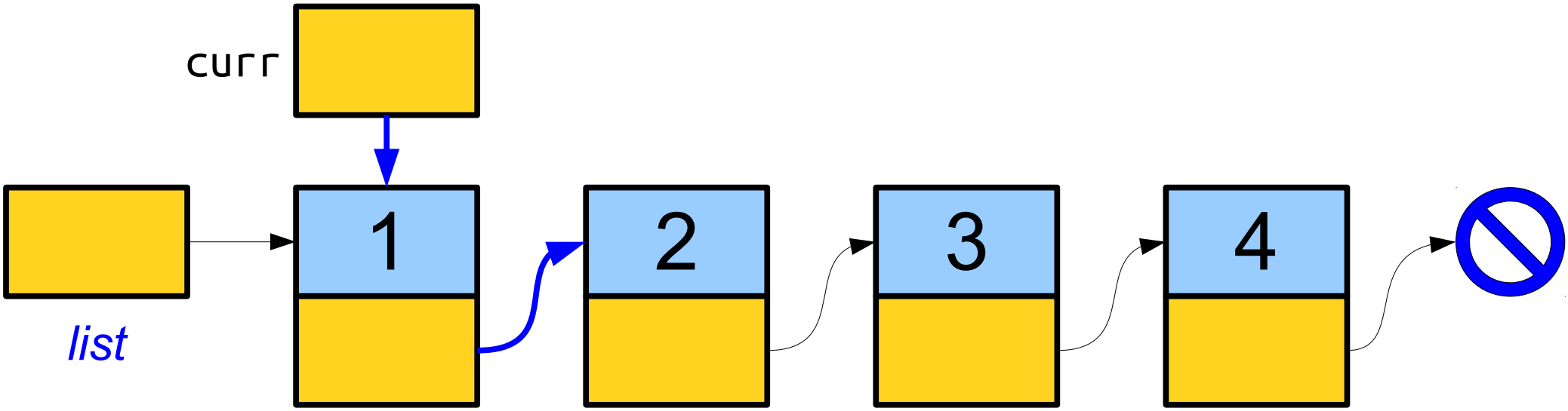
```
for (Cell* curr = list; curr != nullptr; curr = curr->next) {
    /* … do something with curr->value … */
}
```



Linked Lists, Iteratively

- You can navigate a linked list using a traditional `for` loop:

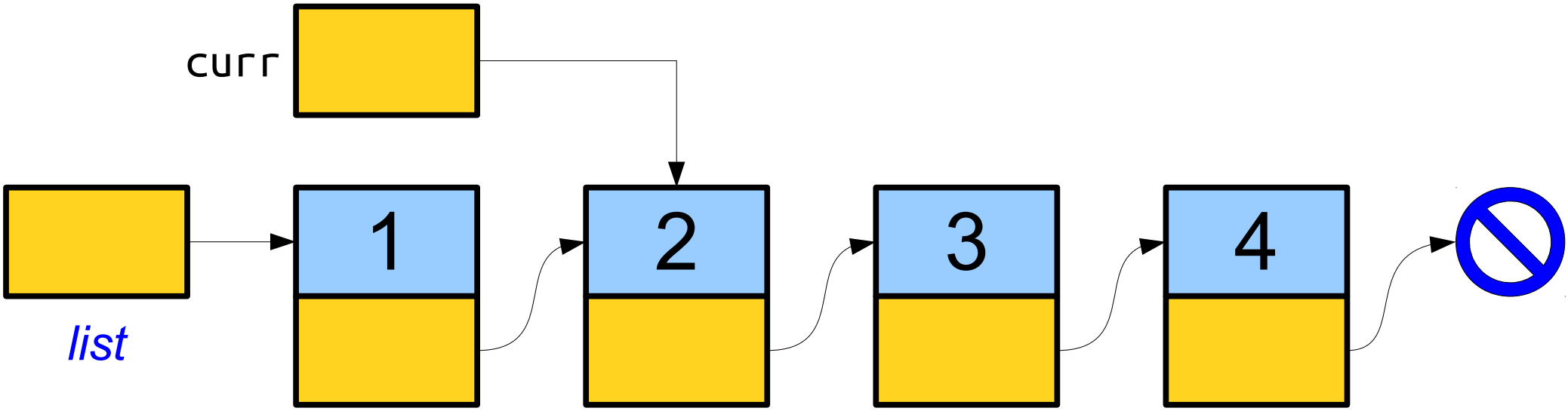
```
for (Cell* curr = list; curr != nullptr; curr = curr->next) {
    /* … do something with curr->value … */
}
```



Linked Lists, Iteratively

- You can navigate a linked list using a traditional `for` loop:

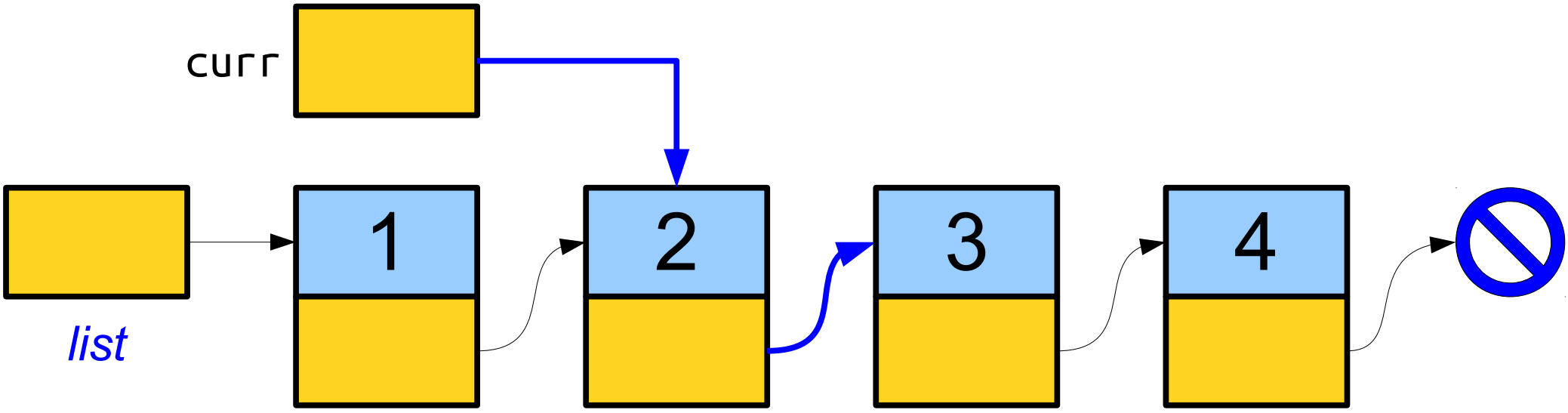
```
for (Cell* curr = list; curr != nullptr; curr = curr->next) {
    /* … do something with curr->value … */
}
```



Linked Lists, Iteratively

- You can navigate a linked list using a traditional `for` loop:

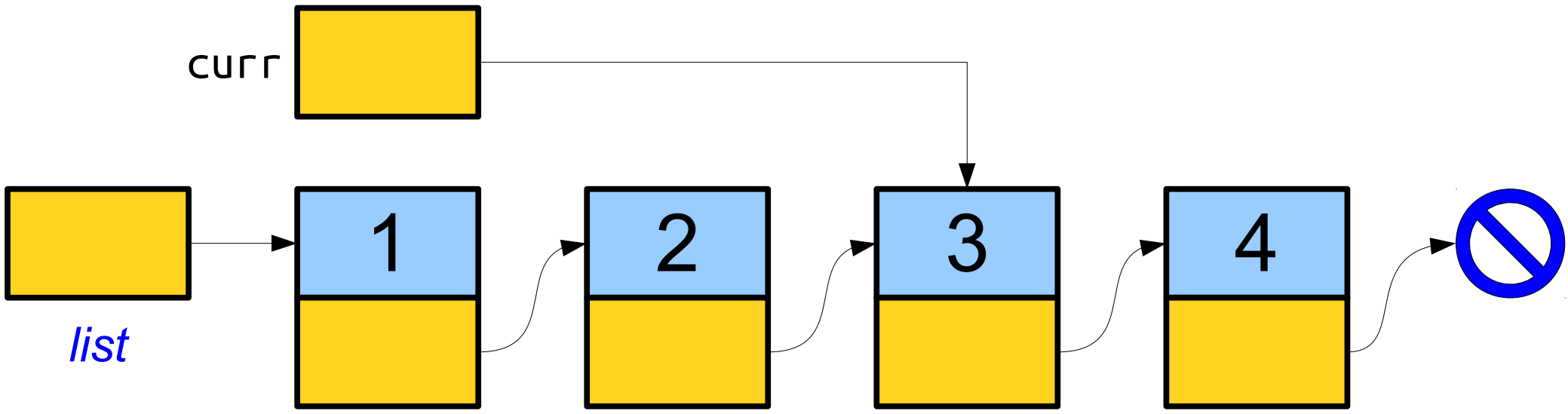
```
for (Cell* curr = list; curr != nullptr; curr = curr->next) {
    /* … do something with curr->value … */
}
```



Linked Lists, Iteratively

- You can navigate a linked list using a traditional `for` loop:

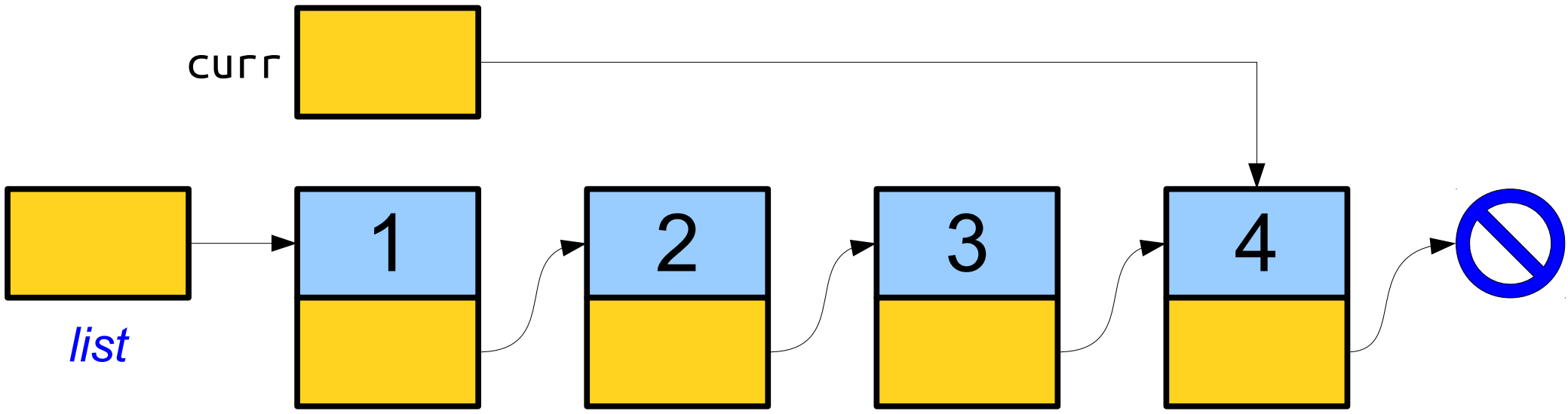
```
for (Cell* curr = list; curr != nullptr; curr = curr->next) {
    /* … do something with curr->value … */
}
```



Linked Lists, Iteratively

- You can navigate a linked list using a traditional `for` loop:

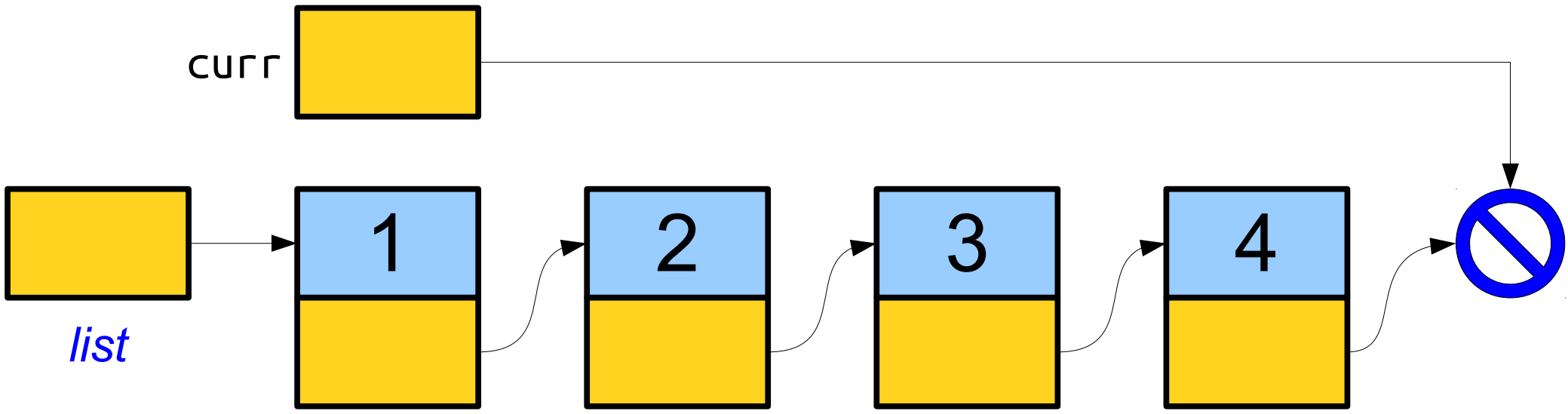
```
for (Cell* curr = list; curr != nullptr; curr = curr->next) {
    /* … do something with curr->value … */
}
```



Linked Lists, Iteratively

- You can navigate a linked list using a traditional `for` loop:

```
for (Cell* curr = list; curr != nullptr; curr = curr->next) {
    /* … do something with curr->value … */
}
```



New Stuff!

ap·op·to·sis

the death of cells which occurs as a normal and controlled part of an organism's growth or development.

Source: Google

Endearing C++ Quirks

- If you allocate memory using the `new[]` operator (e.g. `new int[137]`), you have to free it using the `delete[]` operator.

```
delete[] ptr;
```

- If you allocate memory using the `new` operator (e.g. `new Cell`), you have to free it using the `delete` operator.

```
delete ptr;
```

- ***Make sure to use the proper deletion operation***. Mixing these up leads to Undefined Behavior.

Freeing a Linked List

- All good things must come to an end, and we eventually need to reclaim the memory for a linked list.
- The following code triggers *undefined behavior*. ***Don't do this!***

```
for (Cell* ptr = list; ptr != nullptr; ptr = ptr->next) {
    delete ptr;
}
```

Freeing a Linked List

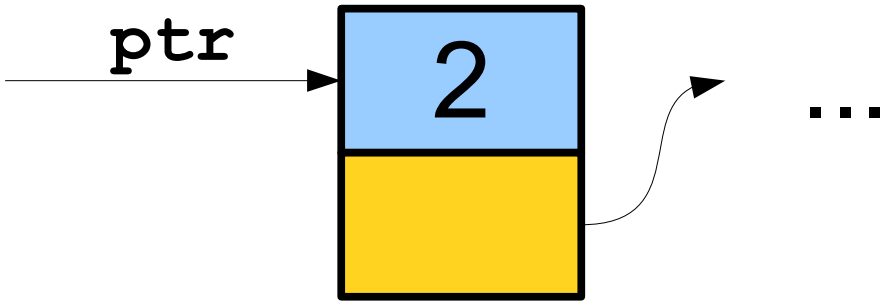
- All good things must come to an end, and we eventually need to reclaim the memory for a linked list.
- The following code triggers *undefined behavior*. ***Don't do this!***

```
for (Cell* ptr = list; ptr != nullptr; ptr = ptr->next) {
    delete ptr;
}
```

Freeing a Linked List

- All good things must come to an end, and we eventually need to reclaim the memory for a linked list.
- The following code triggers *undefined behavior*. ***Don't do this!***

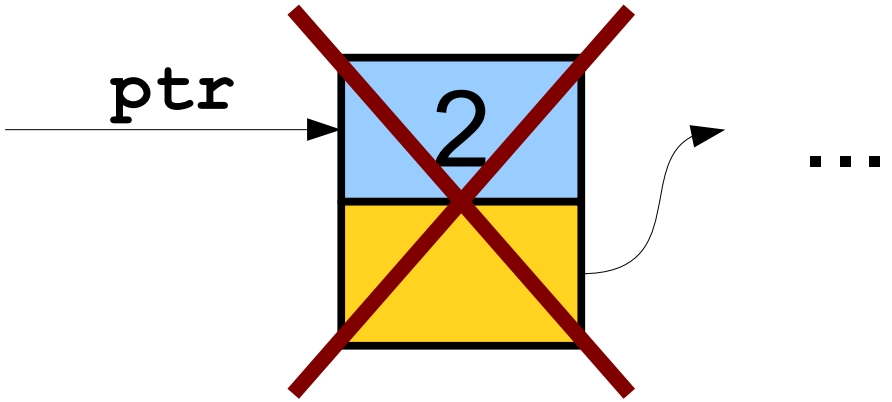
```
for (Cell* ptr = list; ptr != nullptr; ptr = ptr->next) {
    delete ptr;
}
```



Freeing a Linked List

- All good things must come to an end, and we eventually need to reclaim the memory for a linked list.
- The following code triggers *undefined behavior*. ***Don't do this!***

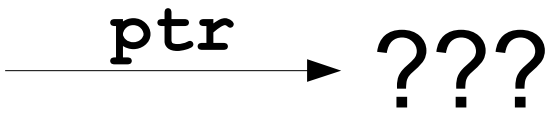
```
for (Cell* ptr = list; ptr != nullptr; ptr = ptr->next) {
    delete ptr;
}
```



Freeing a Linked List

- All good things must come to an end, and we eventually need to reclaim the memory for a linked list.
- The following code triggers *undefined behavior*. ***Don't do this!***

```
for (Cell* ptr = list; ptr != nullptr; ptr = ptr->next) {
    delete ptr;
}
```

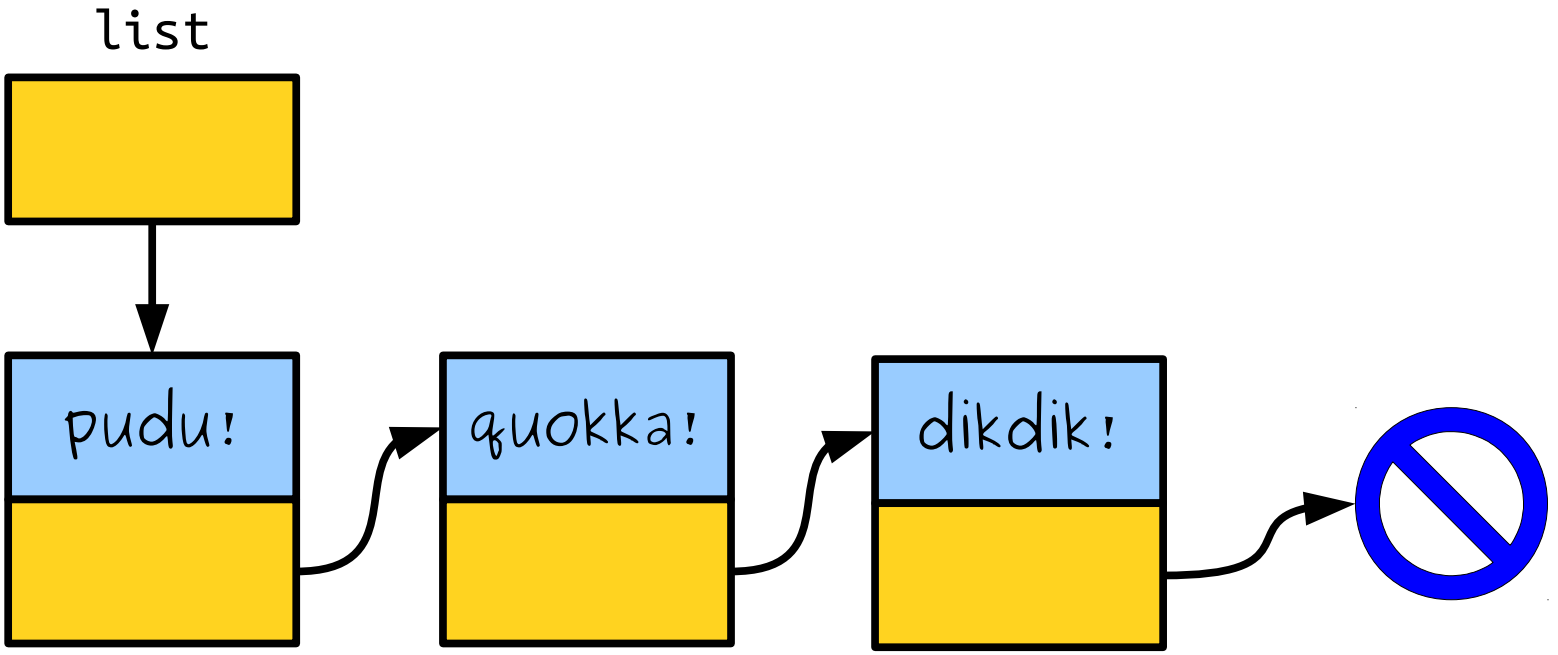


Freeing a Linked List Properly

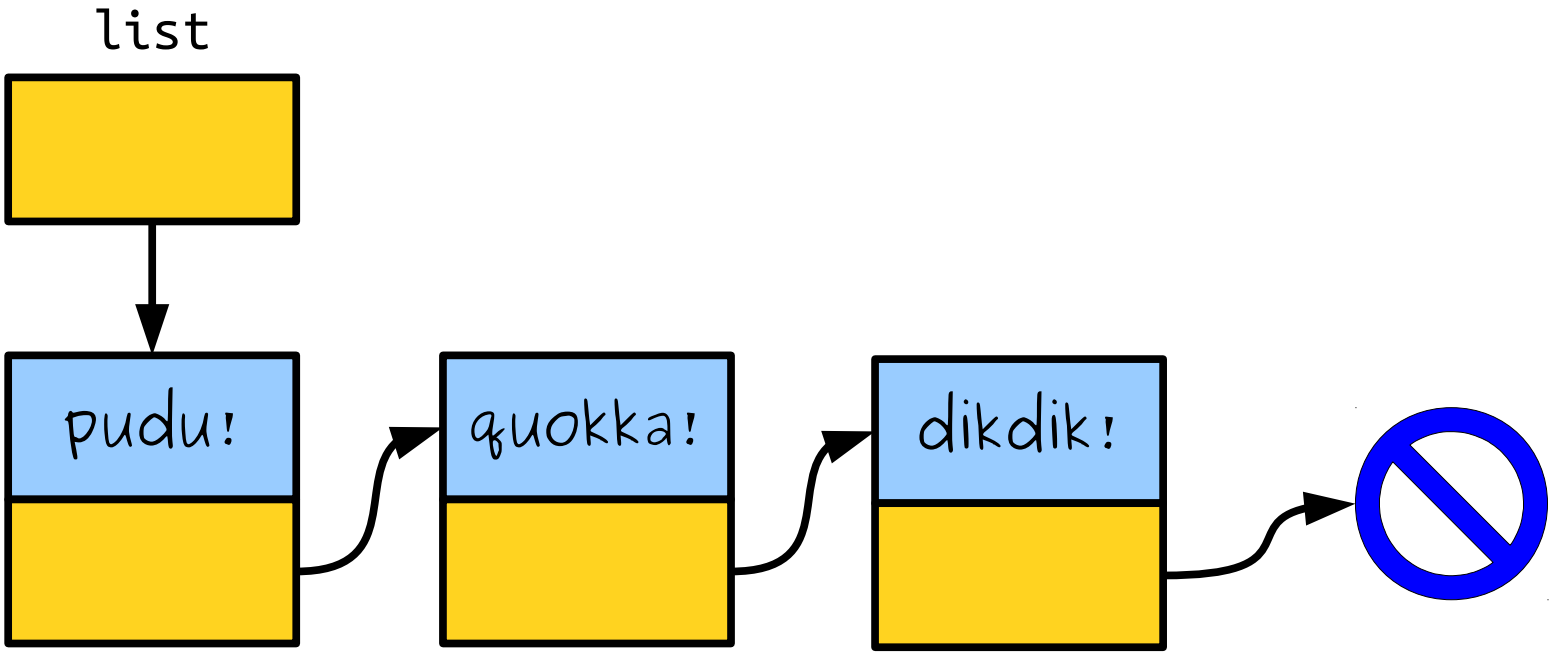
- To properly free a linked list, we have to be able to
  - destroy a cell, and
  - advance to the cell after it.
- How might we accomplish this?

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

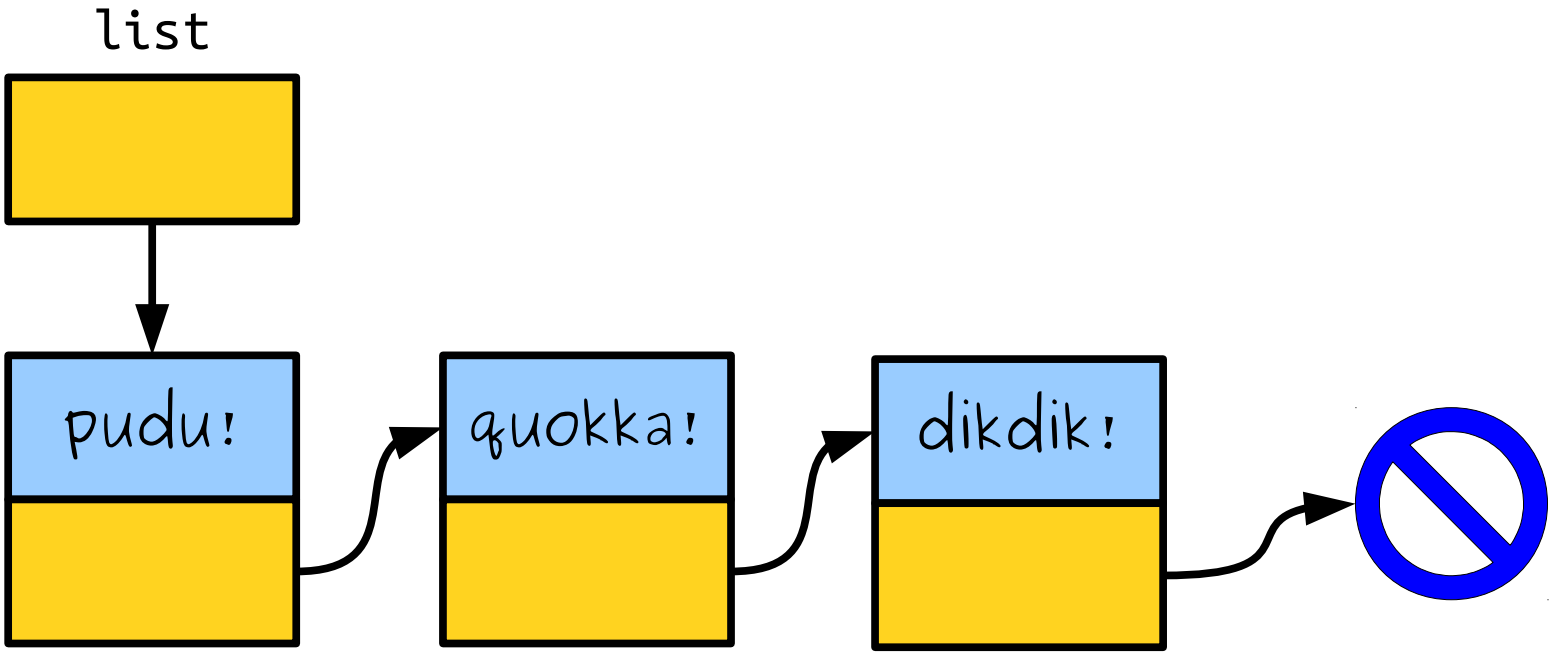
```
while (list != nullptr) {
    Cell* next = list->next;
    delete list;
    list = next;
}
```



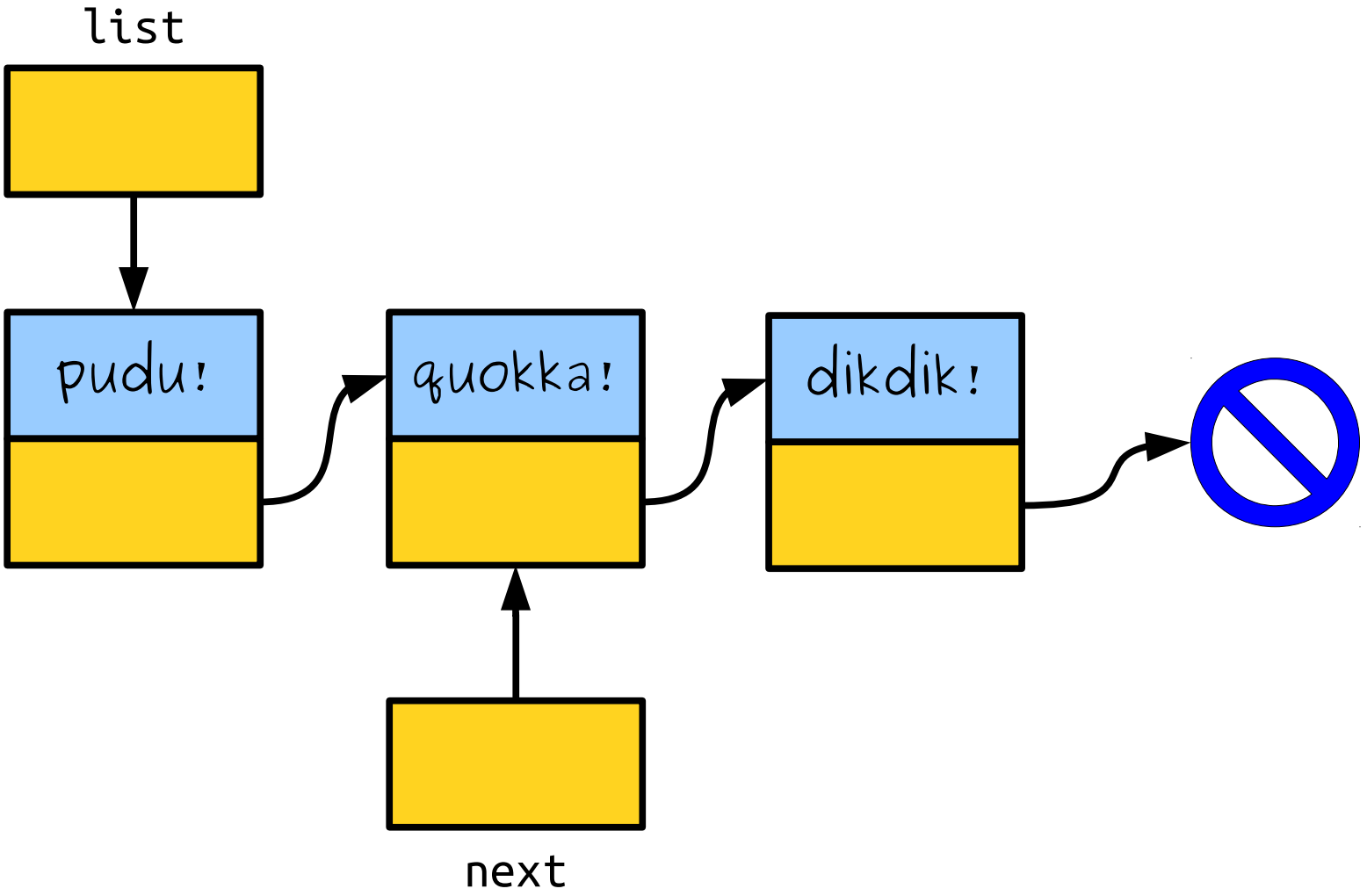
```
while (list != nullptr) {
    Cell* next = list->next;
    delete list;
    list = next;
}
```



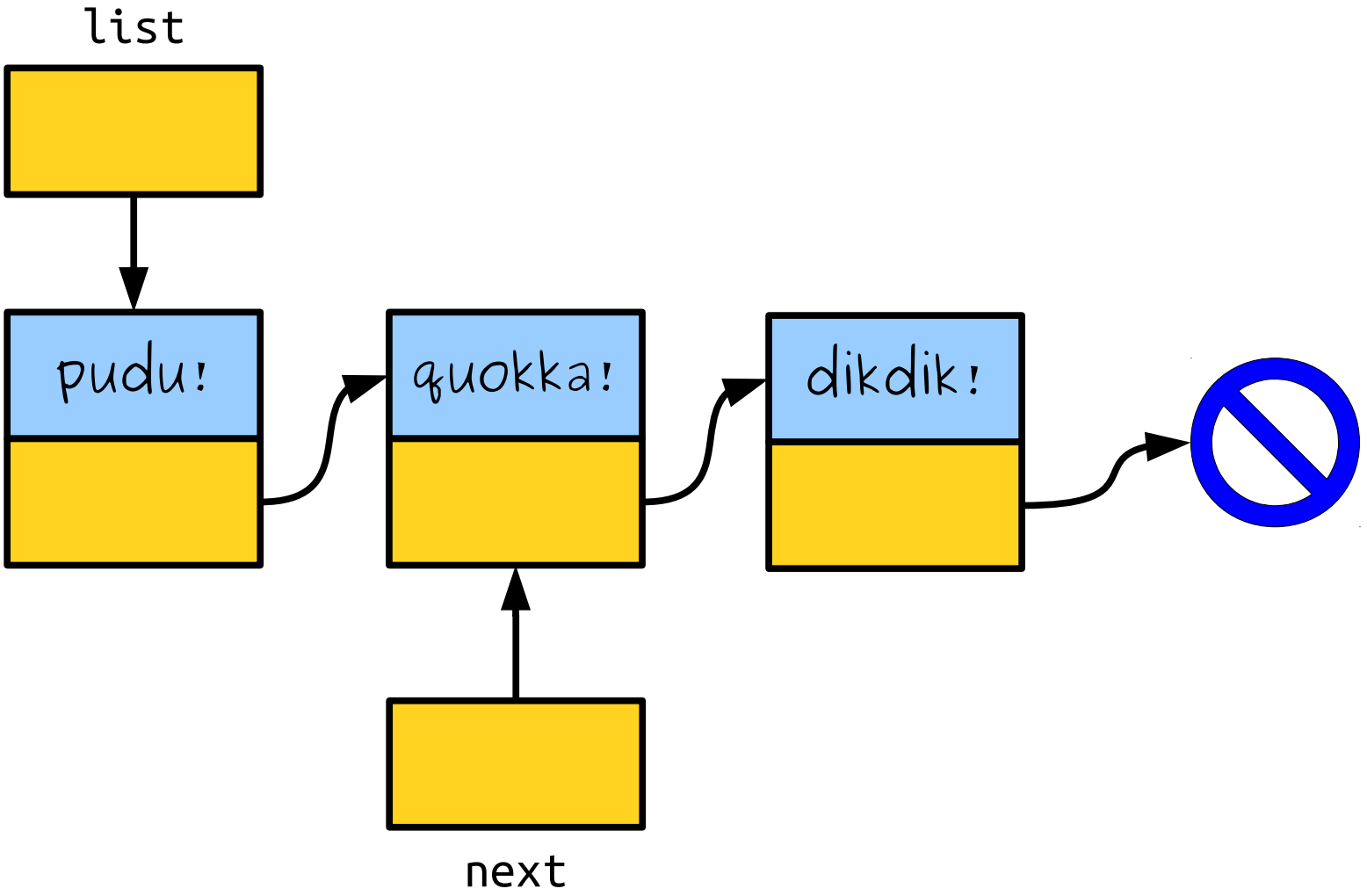
```
while (list != nullptr) {
    Cell* next = list->next;
    delete list;
    list = next;
}
```



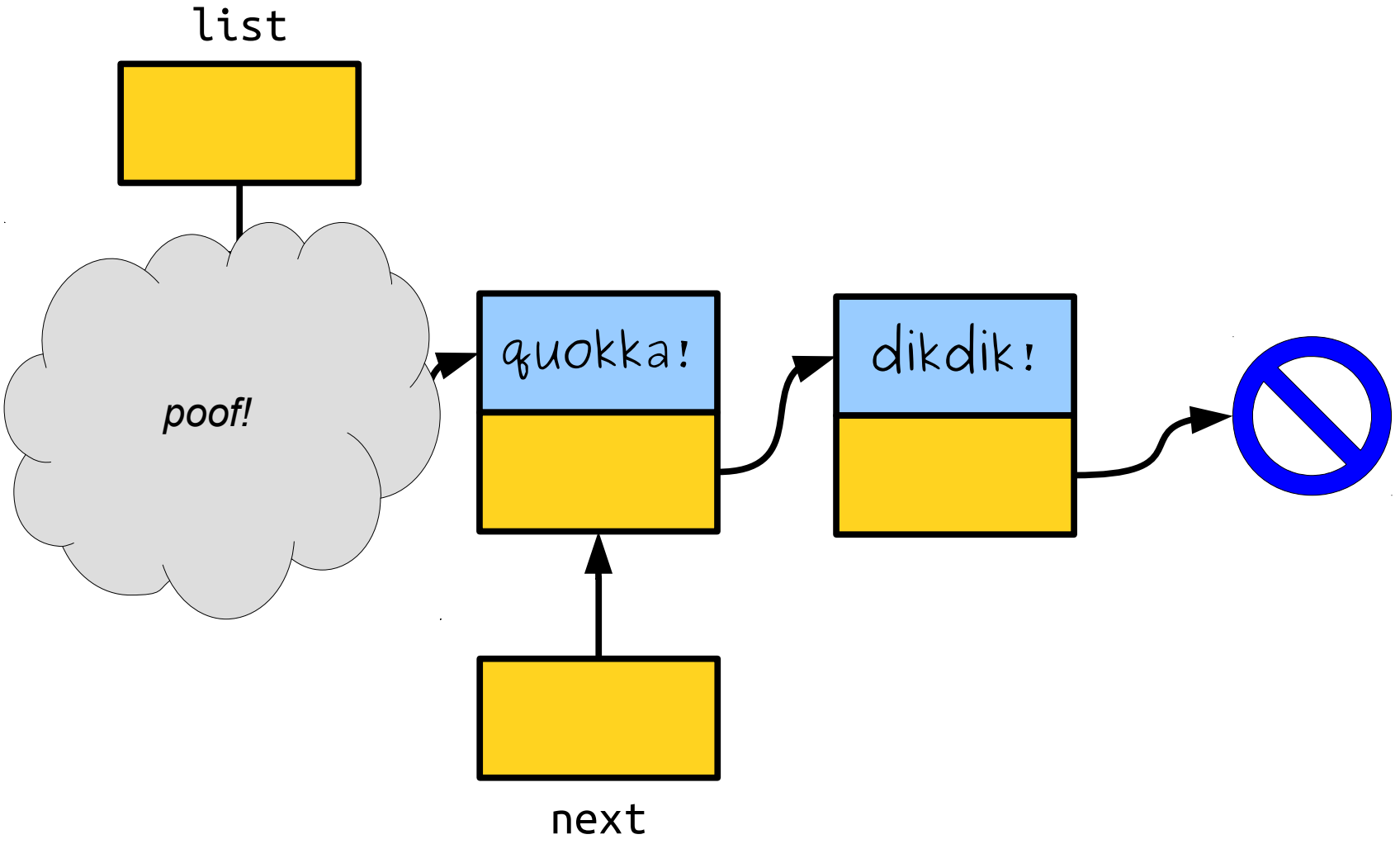
```
while (list != nullptr) {
    Cell* next = list->next;
    delete list;
    list = next;
}
```



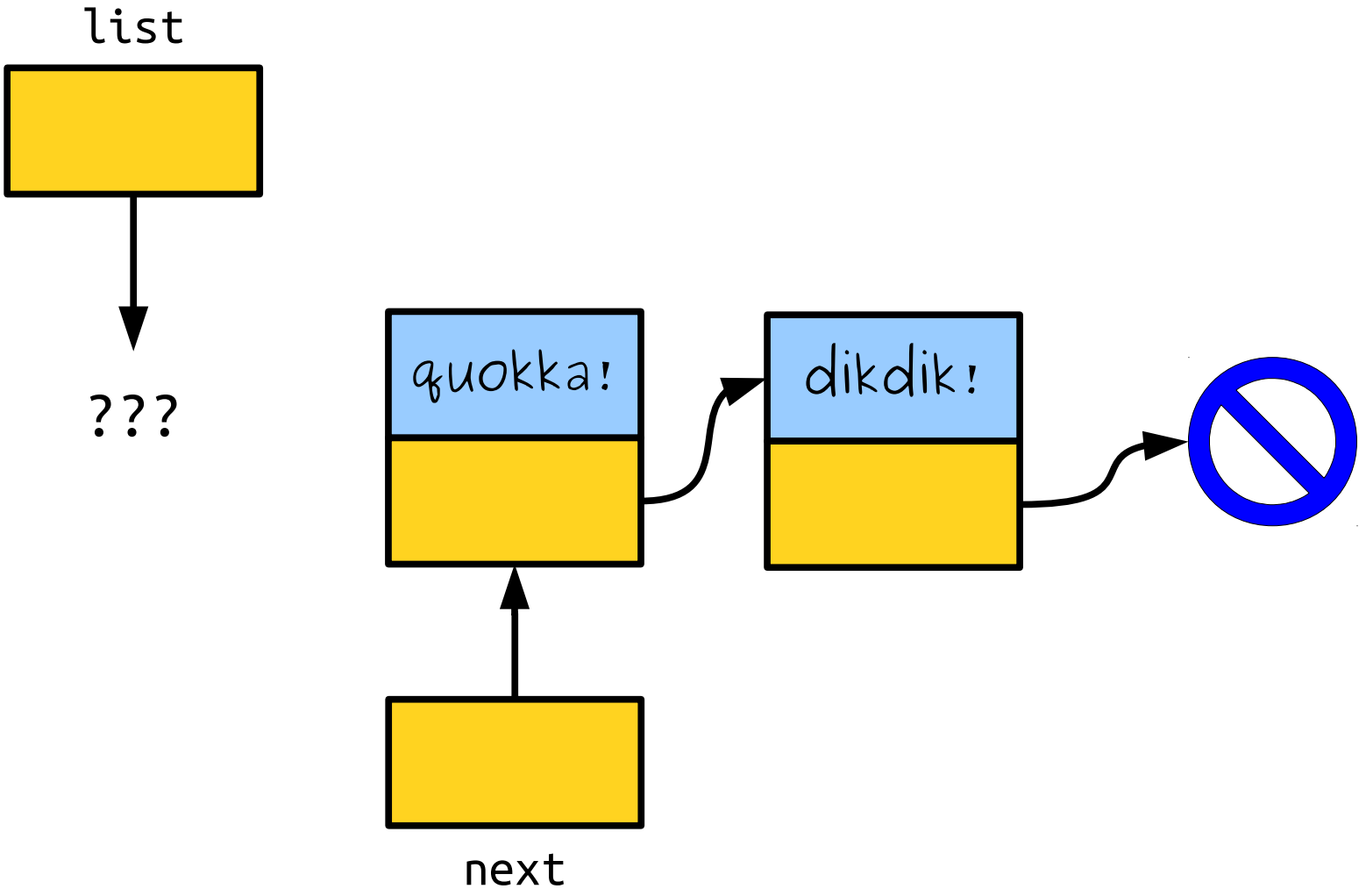
```
while (list != nullptr) {
    Cell* next = list->next;
    delete list;
    list = next;
}
```



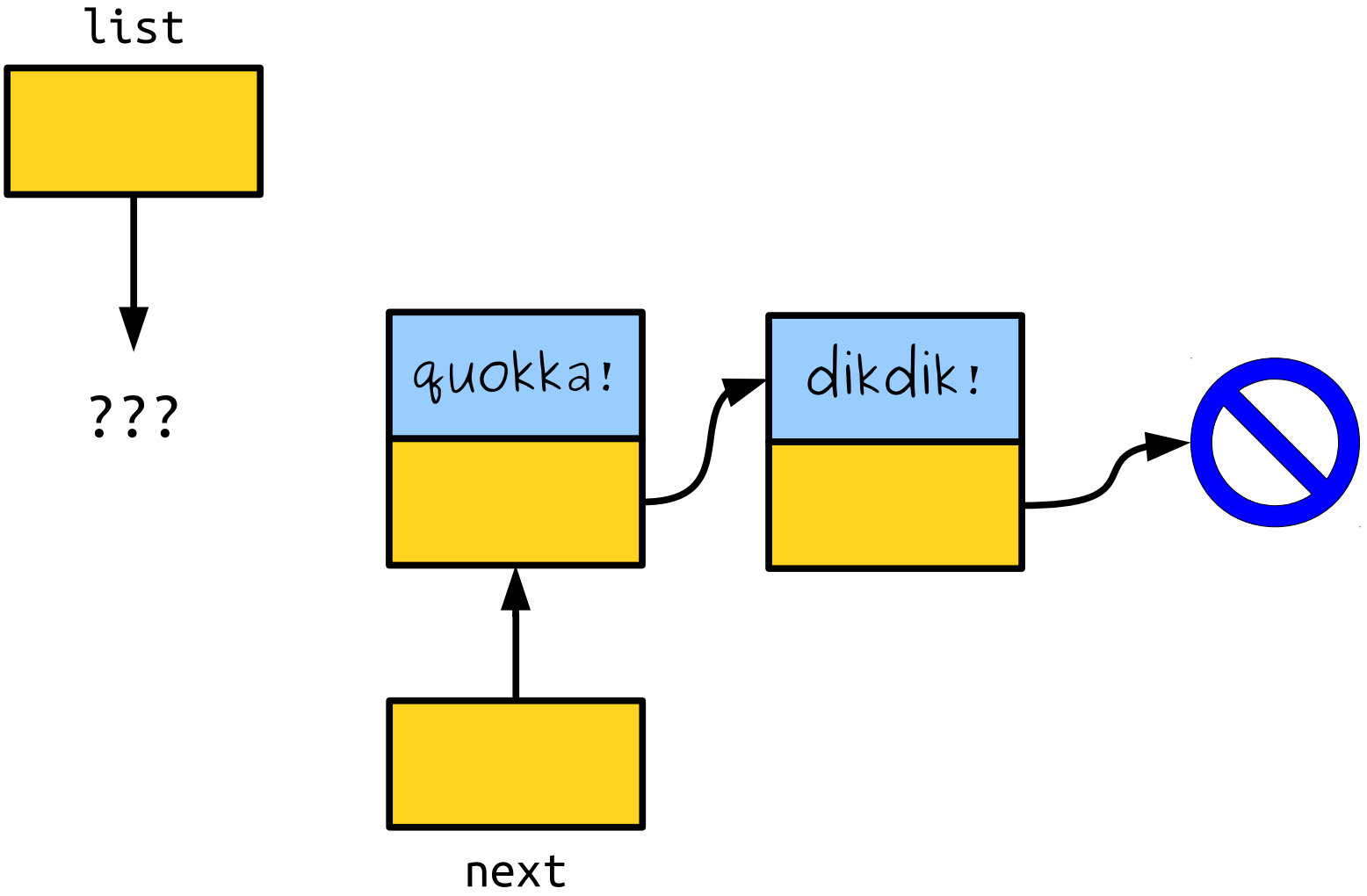
```
while (list != nullptr) {
    Cell* next = list->next;
    delete list;
    list = next;
}
```



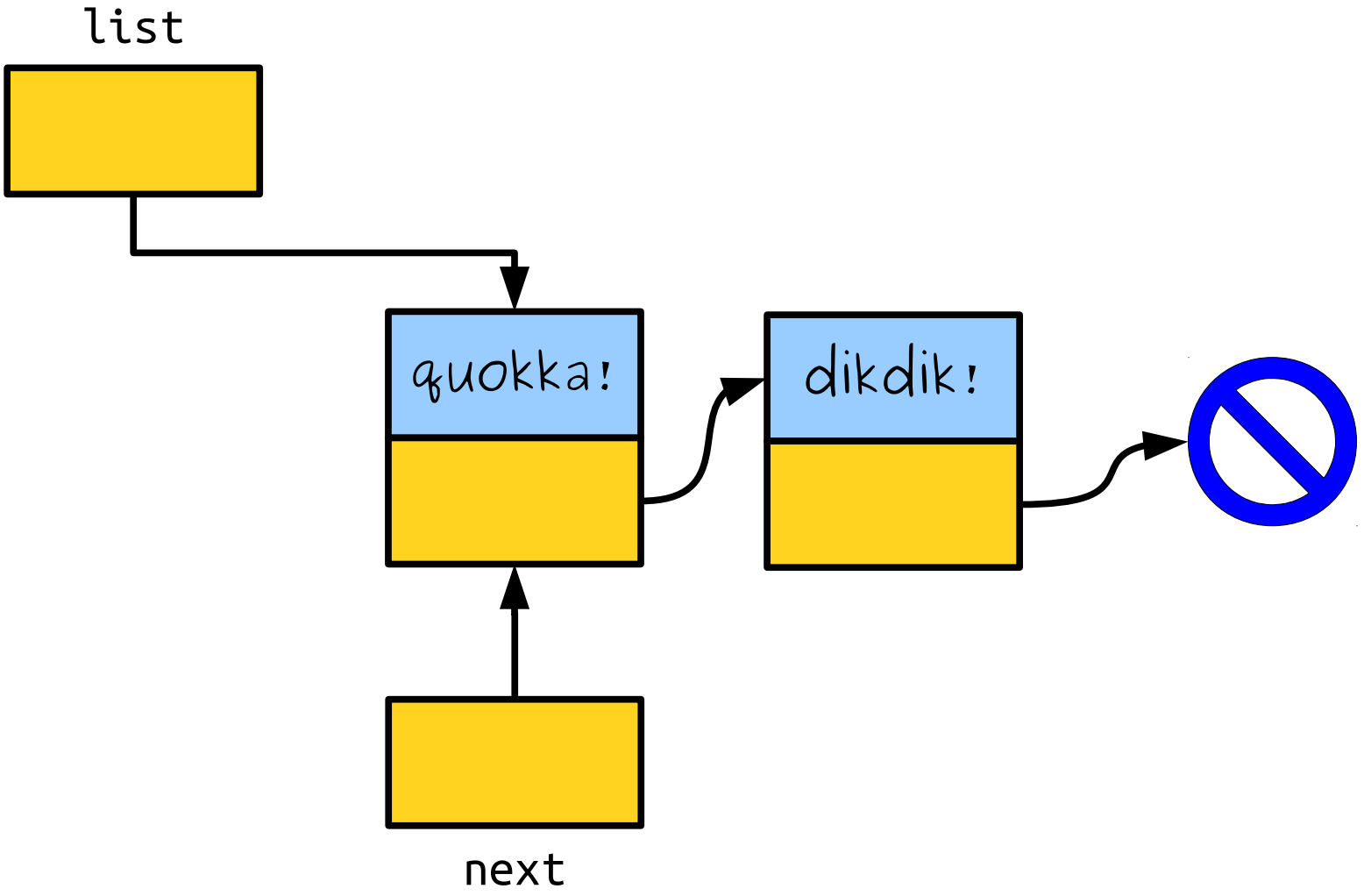
```
while (list != nullptr) {
    Cell* next = list->next;
    delete list;
    list = next;
}
```



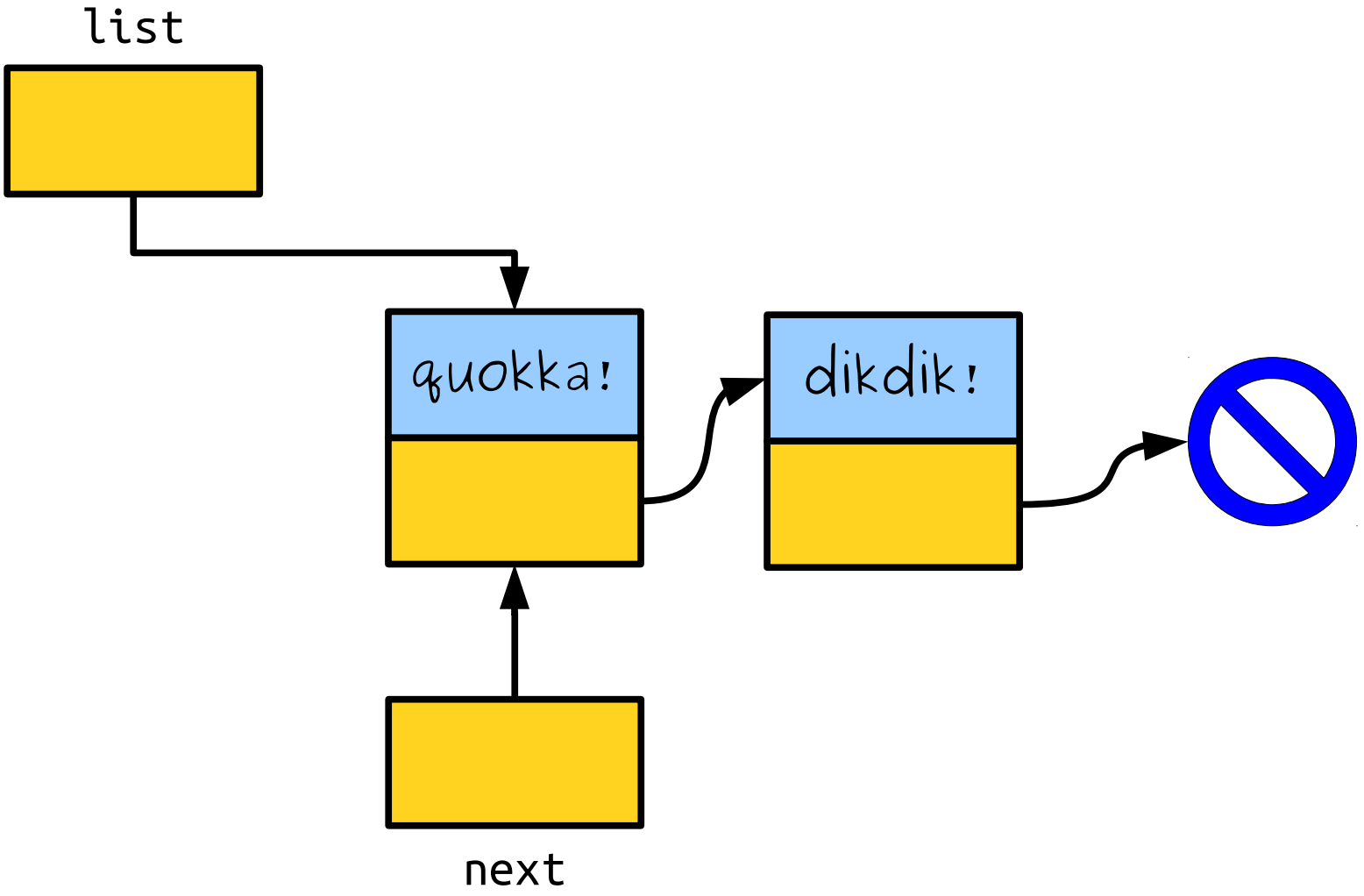
```
while (list != nullptr) {
    Cell* next = list->next;
    delete list;
    list = next;
}
```



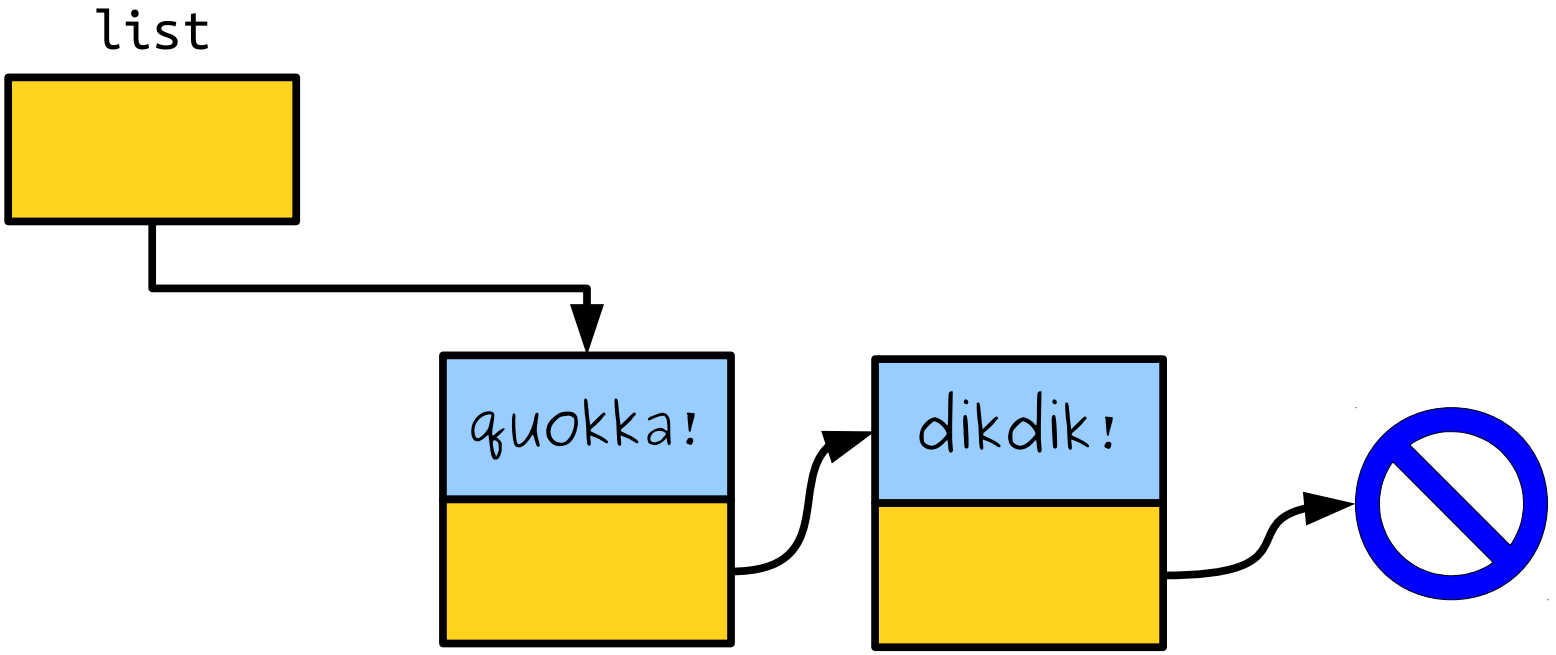
```
while (list != nullptr) {
    Cell* next = list->next;
    delete list;
    list = next;
}
```



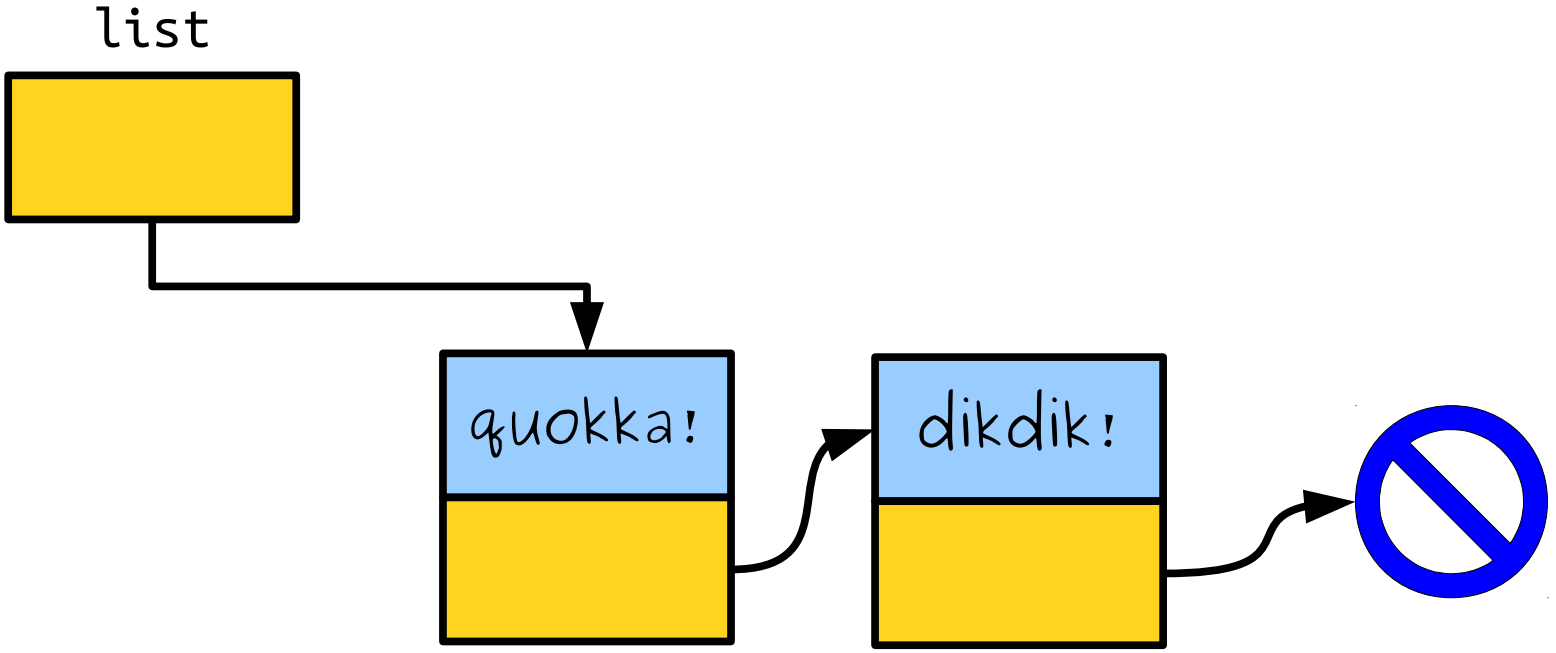
```
while (list != nullptr) {
    Cell* next = list->next;
    delete list;
    list = next;
}
```



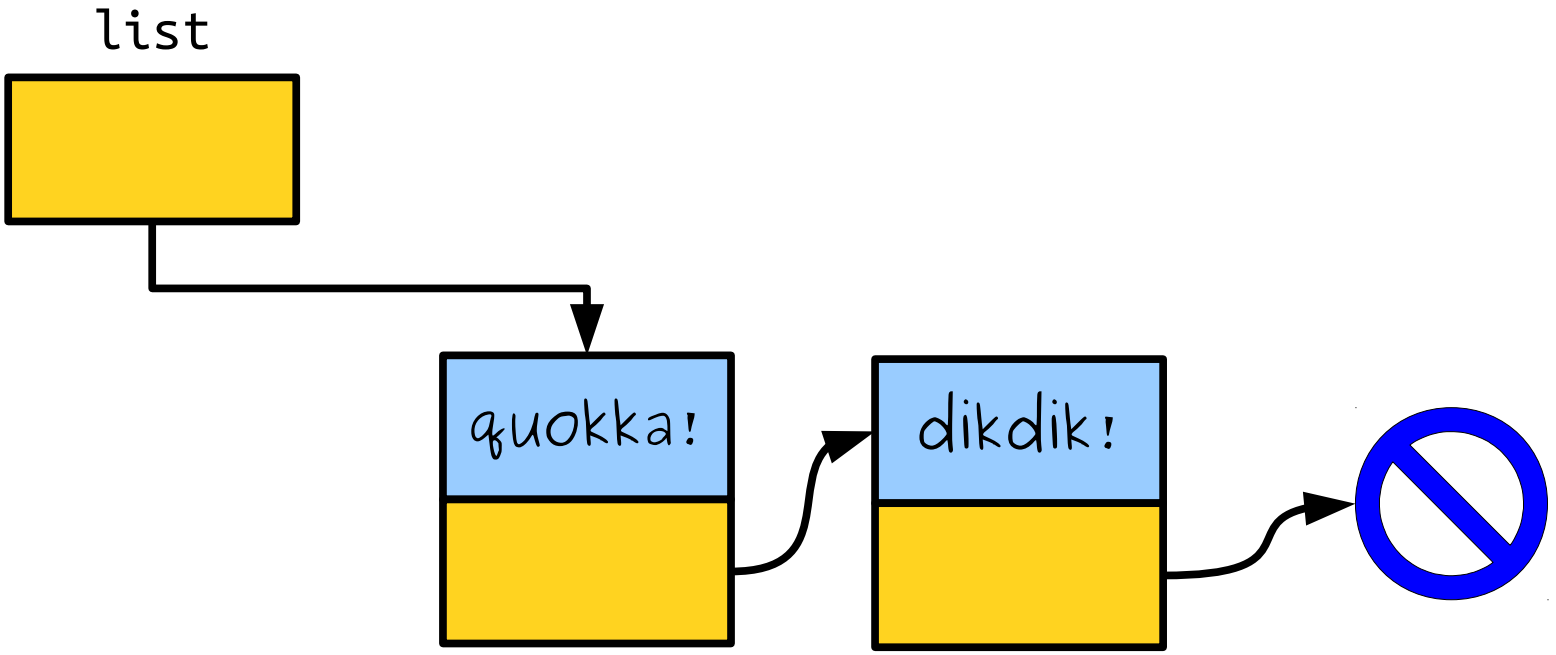
```
while (list != nullptr) {
    Cell* next = list->next;
    delete list;
    list = next;
}
```



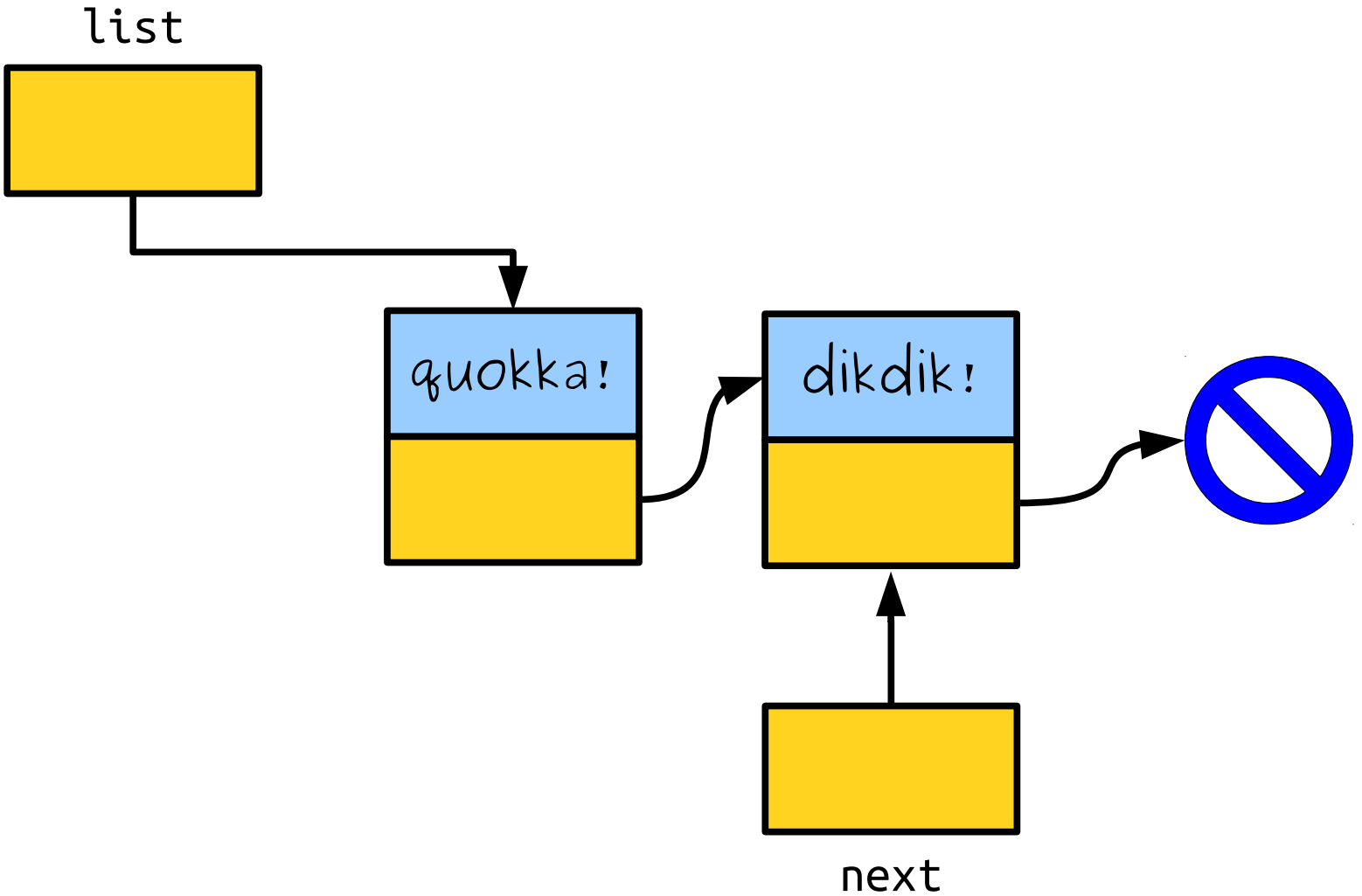
```
while (list != nullptr) {
    Cell* next = list->next;
    delete list;
    list = next;
}
```



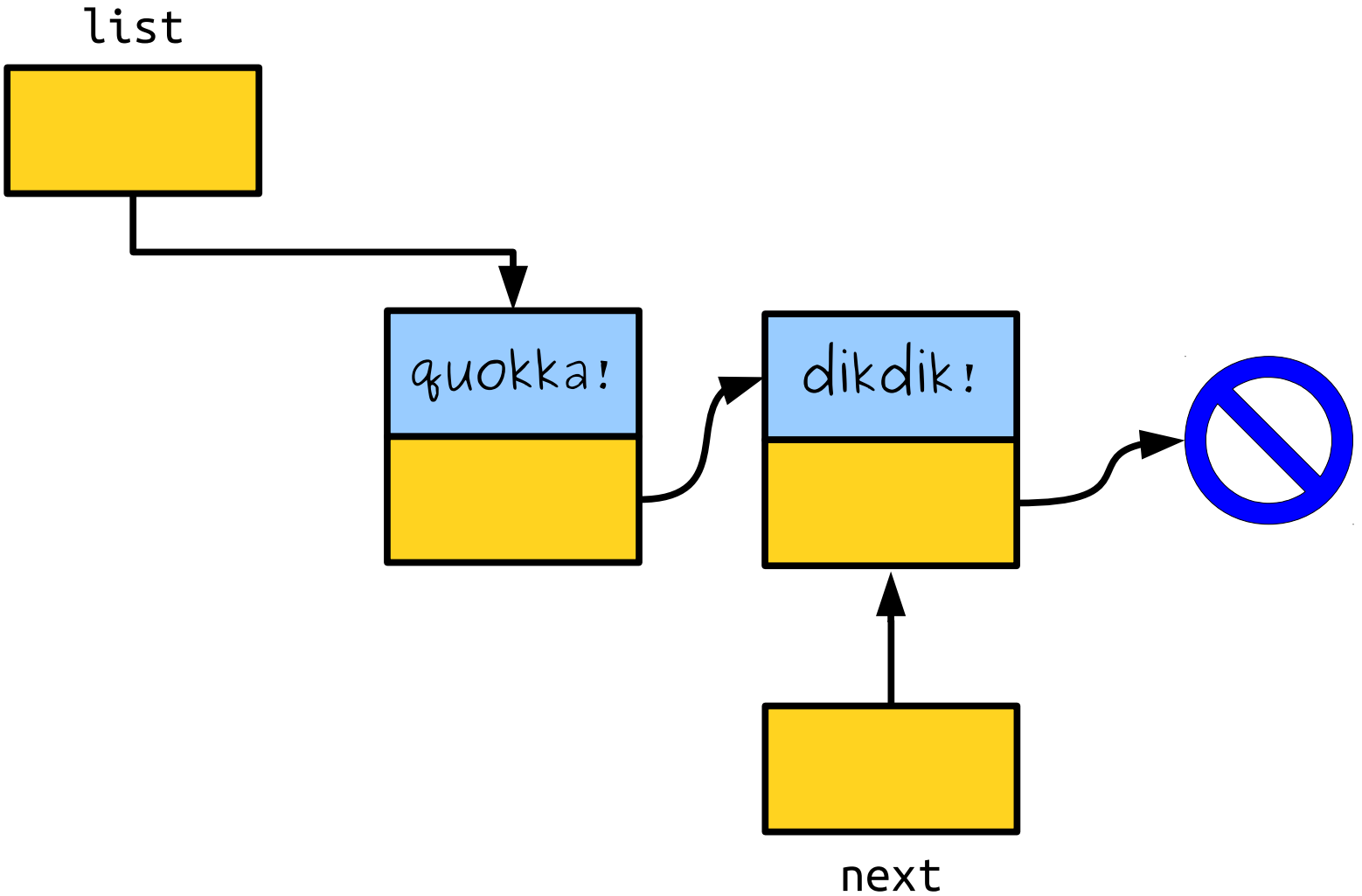
```
while (list != nullptr) {
    Cell* next = list->next;
    delete list;
    list = next;
}
```



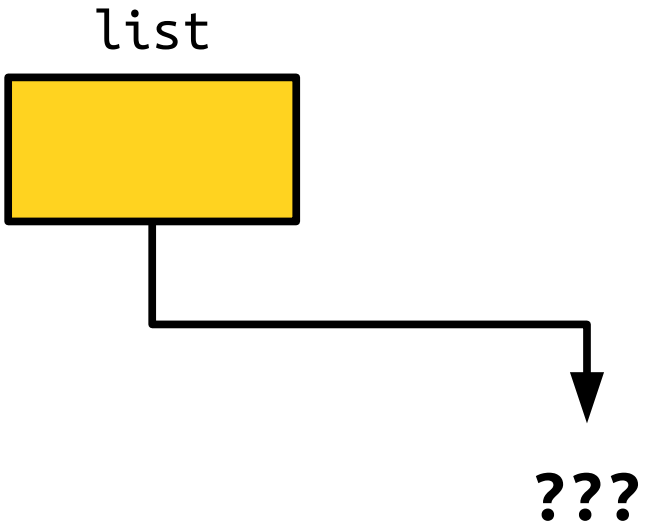
```
while (list != nullptr) {
    Cell* next = list->next;
    delete list;
    list = next;
}
```

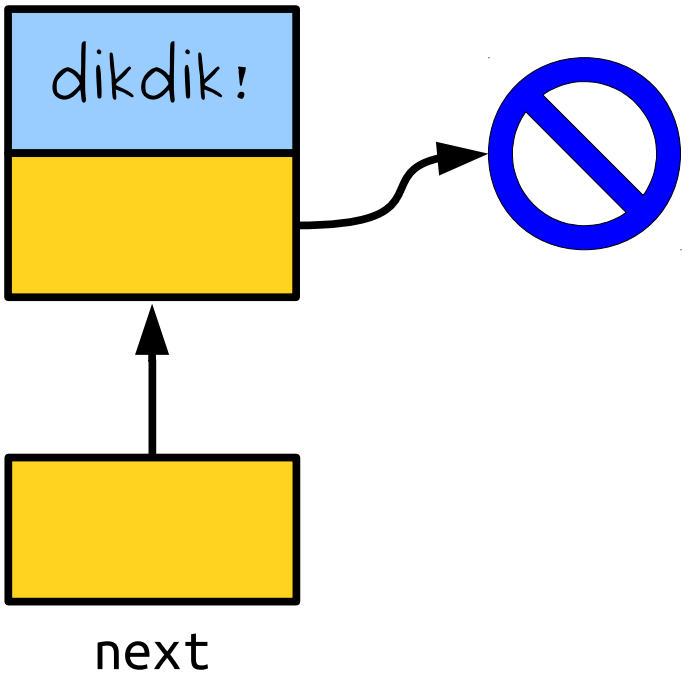


```
while (list != nullptr) {
    Cell* next = list->next;
    delete list;
    list = next;
}
```

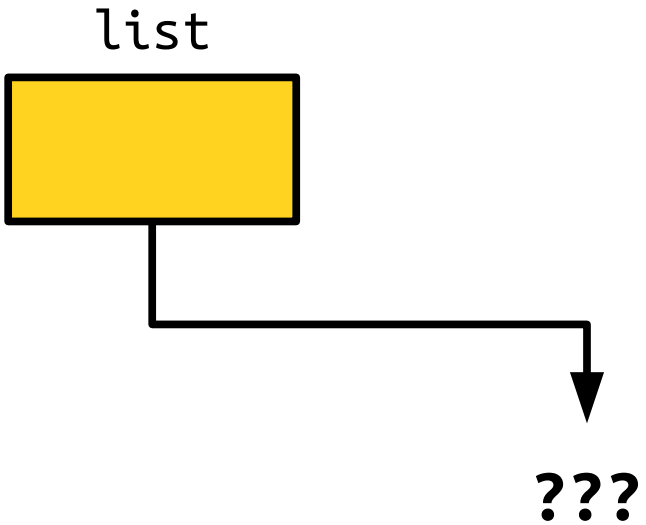


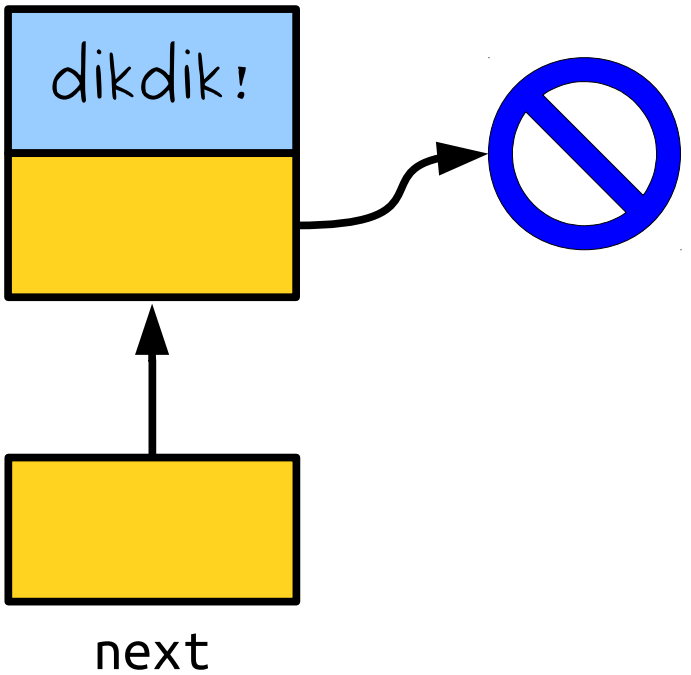
```
while (list != nullptr) {
    Cell* next = list->next;
    delete list;
    list = next;
}
```



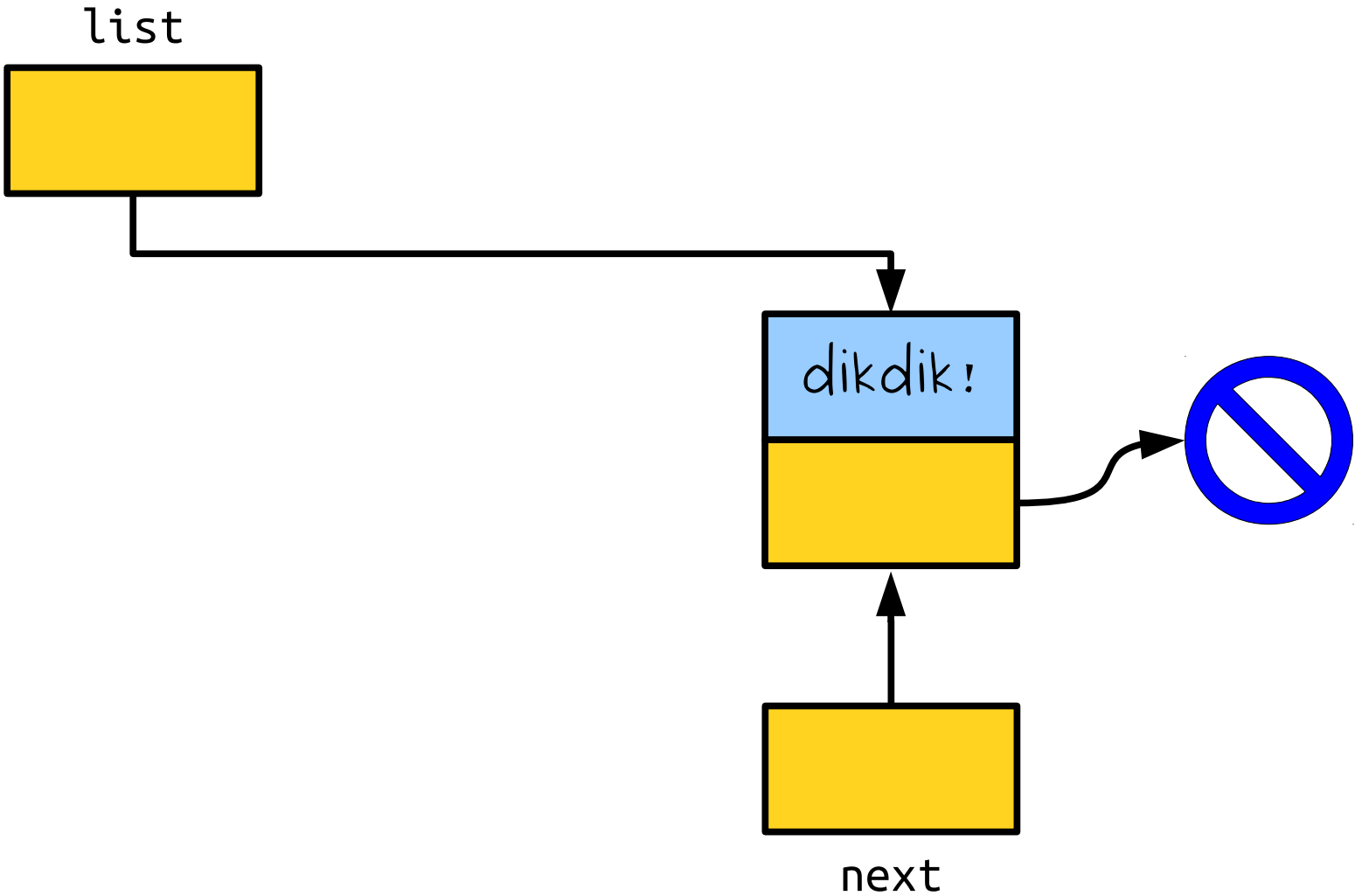


```
while (list != nullptr) {
    Cell* next = list->next;
    delete list;
    list = next;
}
```

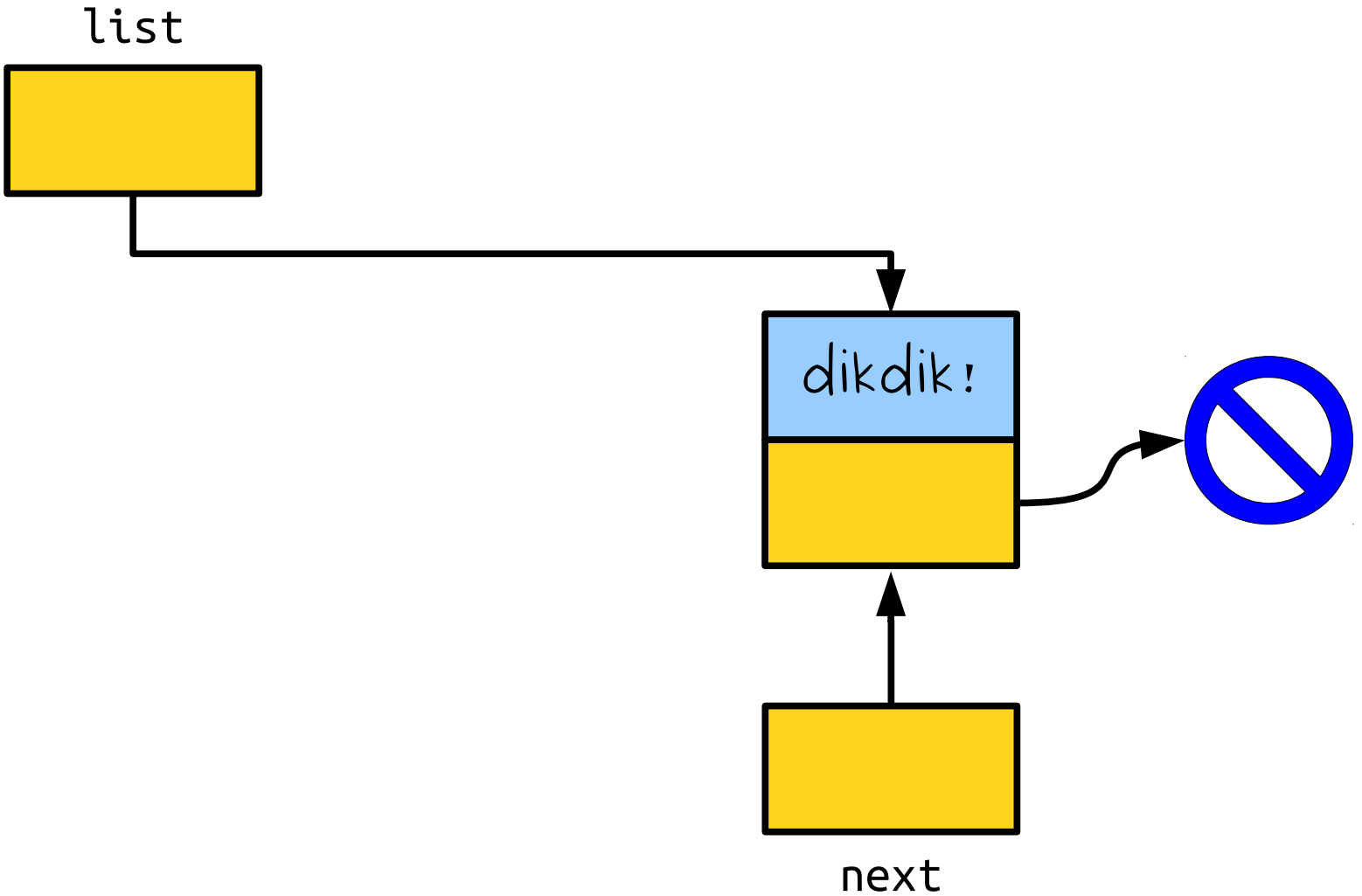




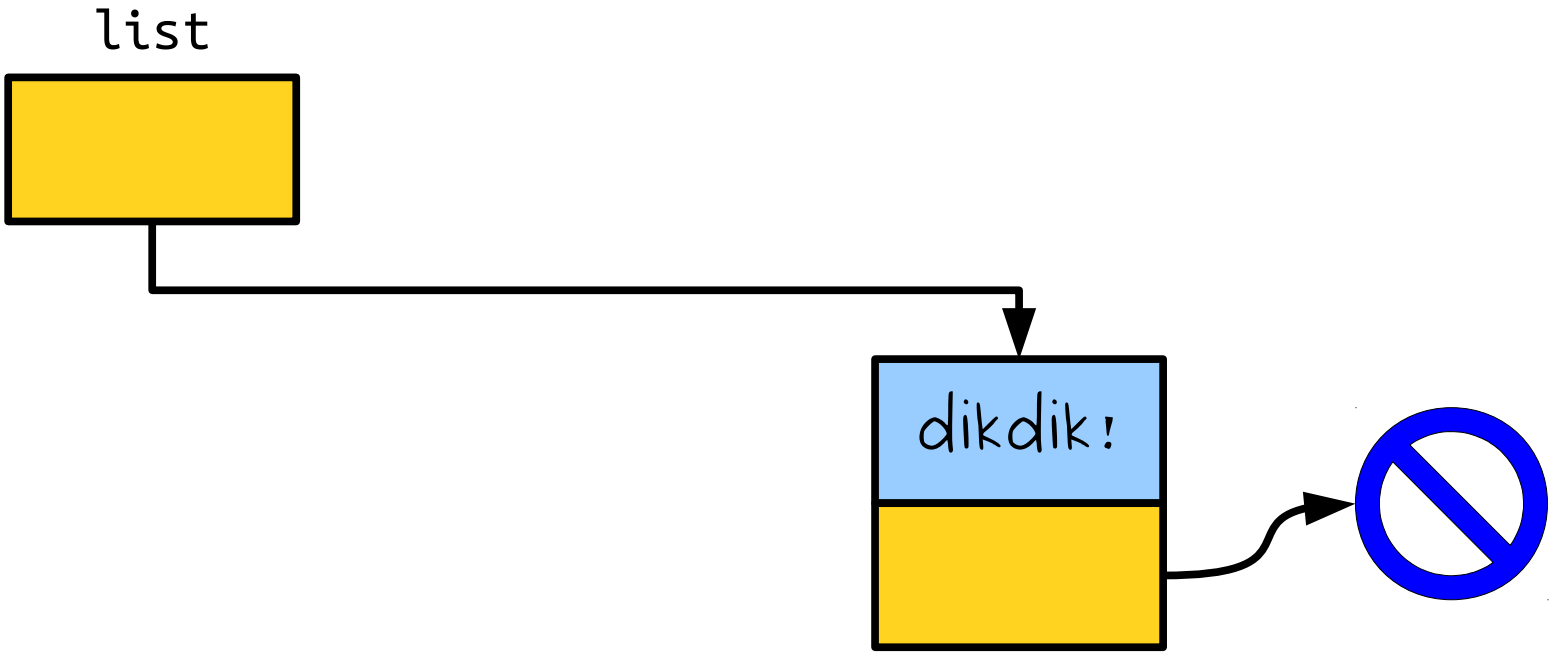
```
while (list != nullptr) {
    Cell* next = list->next;
    delete list;
    list = next;
}
```



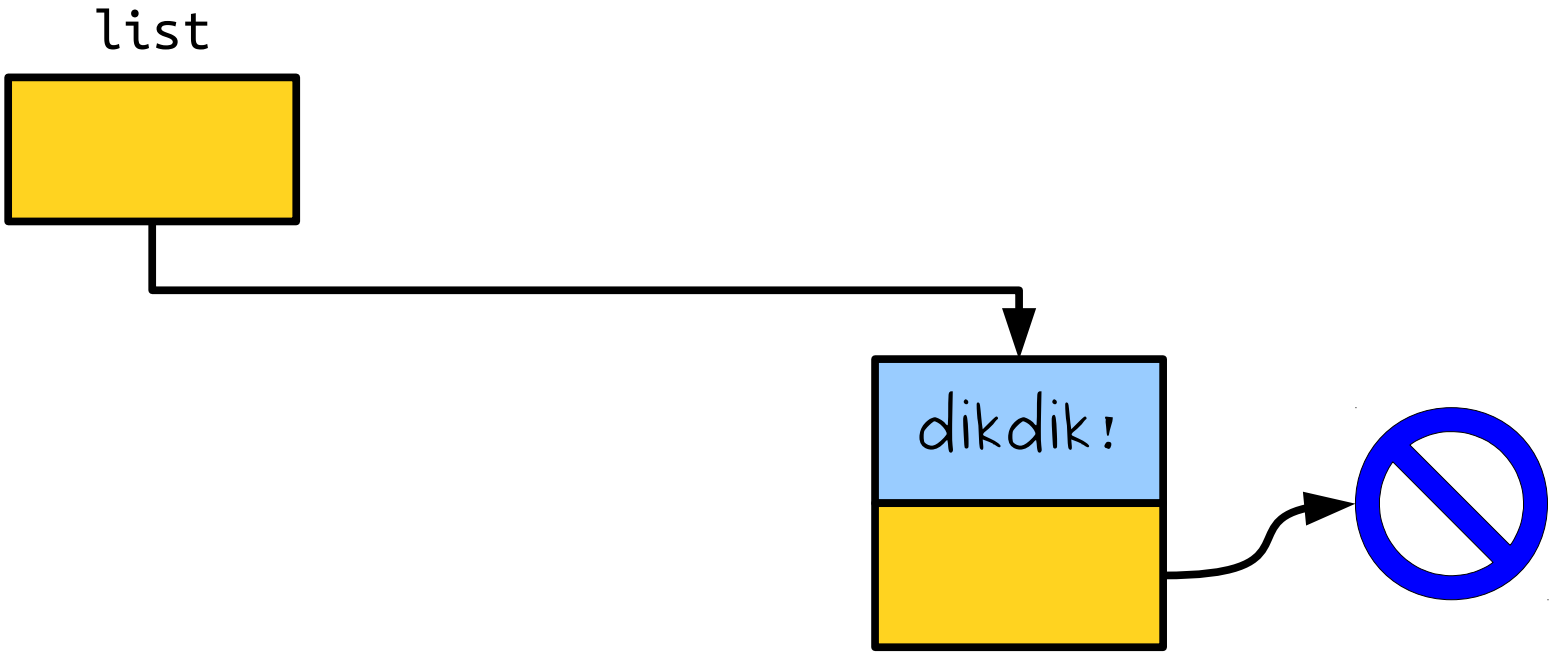
```
while (list != nullptr) {
    Cell* next = list->next;
    delete list;
    list = next;
}
```



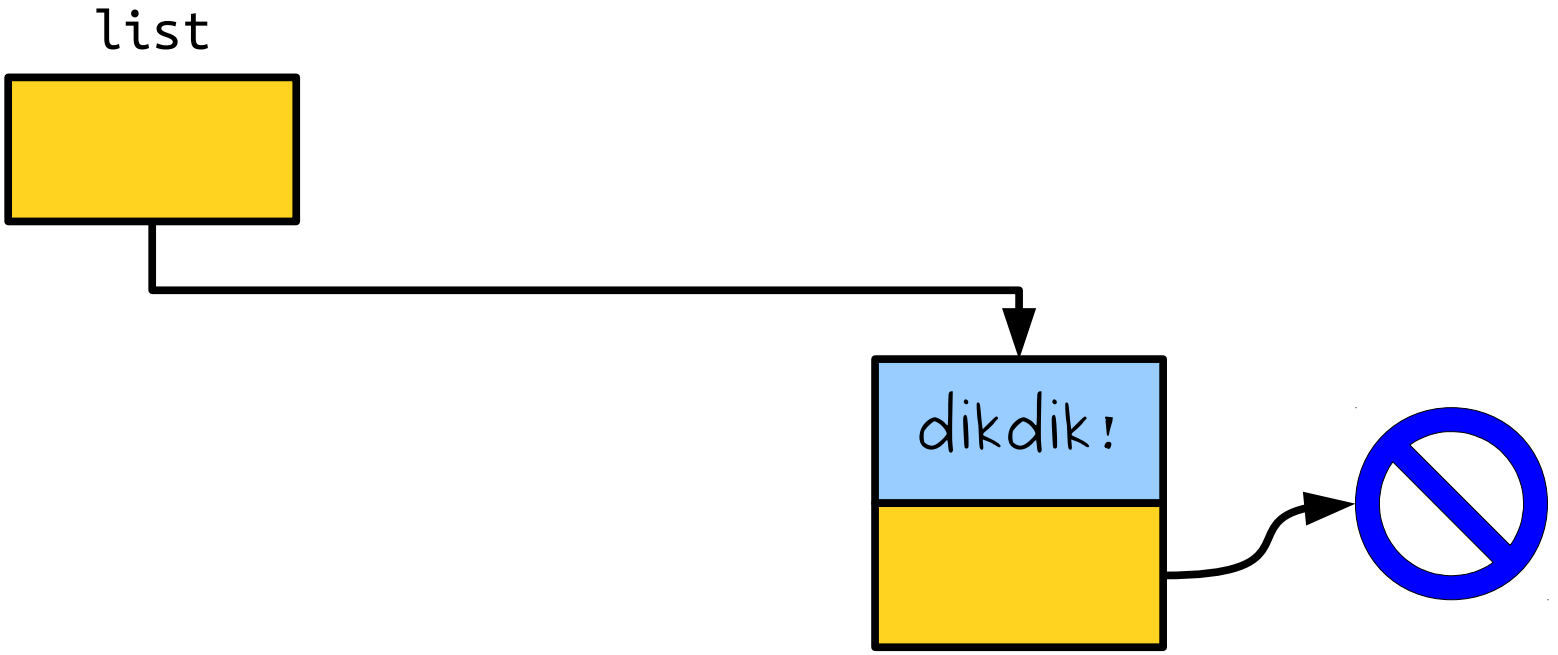
```
while (list != nullptr) {
    Cell* next = list->next;
    delete list;
    list = next;
}
```



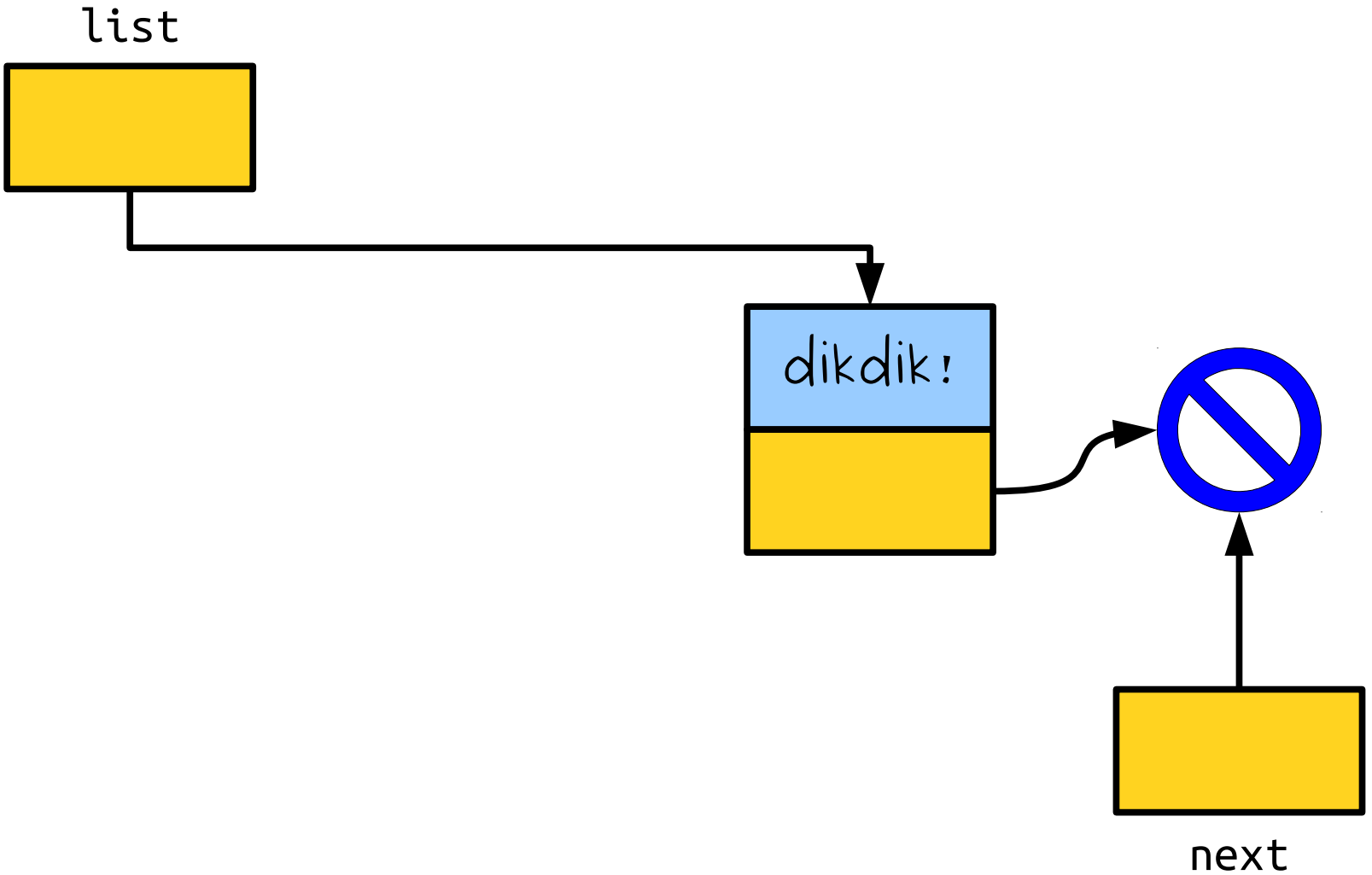
```
while (list != nullptr) {
    Cell* next = list->next;
    delete list;
    list = next;
}
```



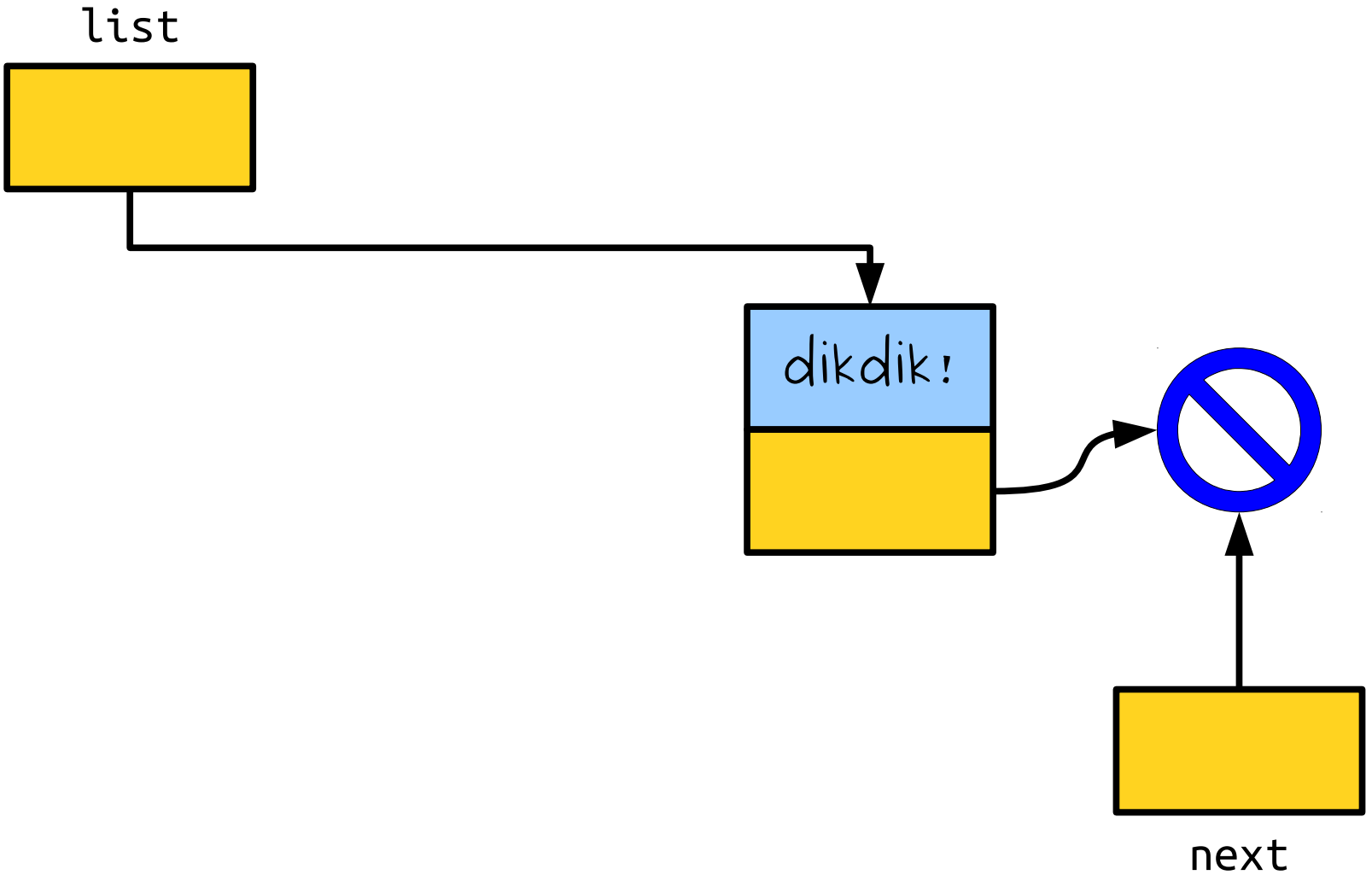
```
while (list != nullptr) {
    Cell* next = list->next;
    delete list;
    list = next;
}
```



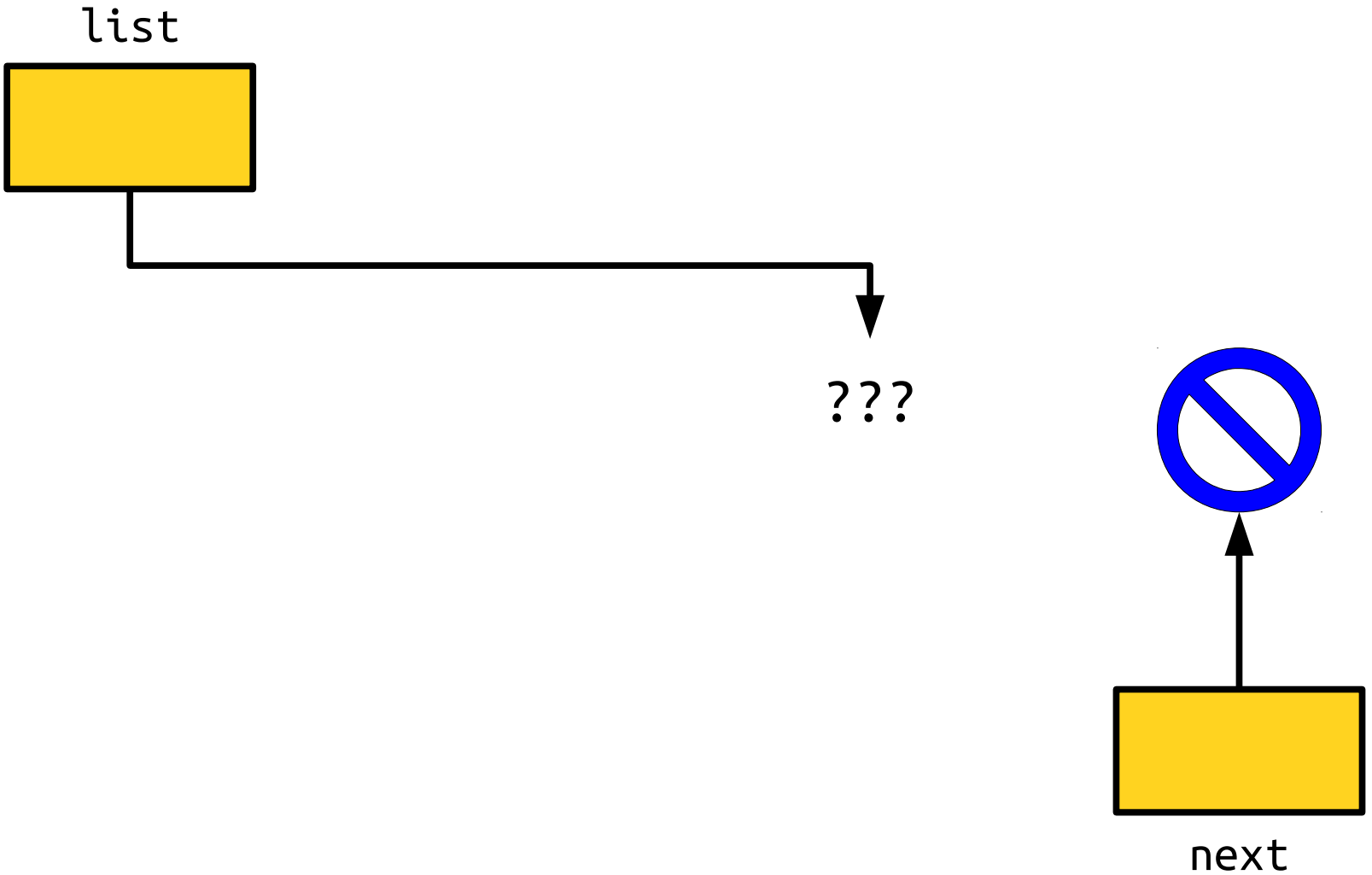
```
while (list != nullptr) {
    Cell* next = list->next;
    delete list;
    list = next;
}
```



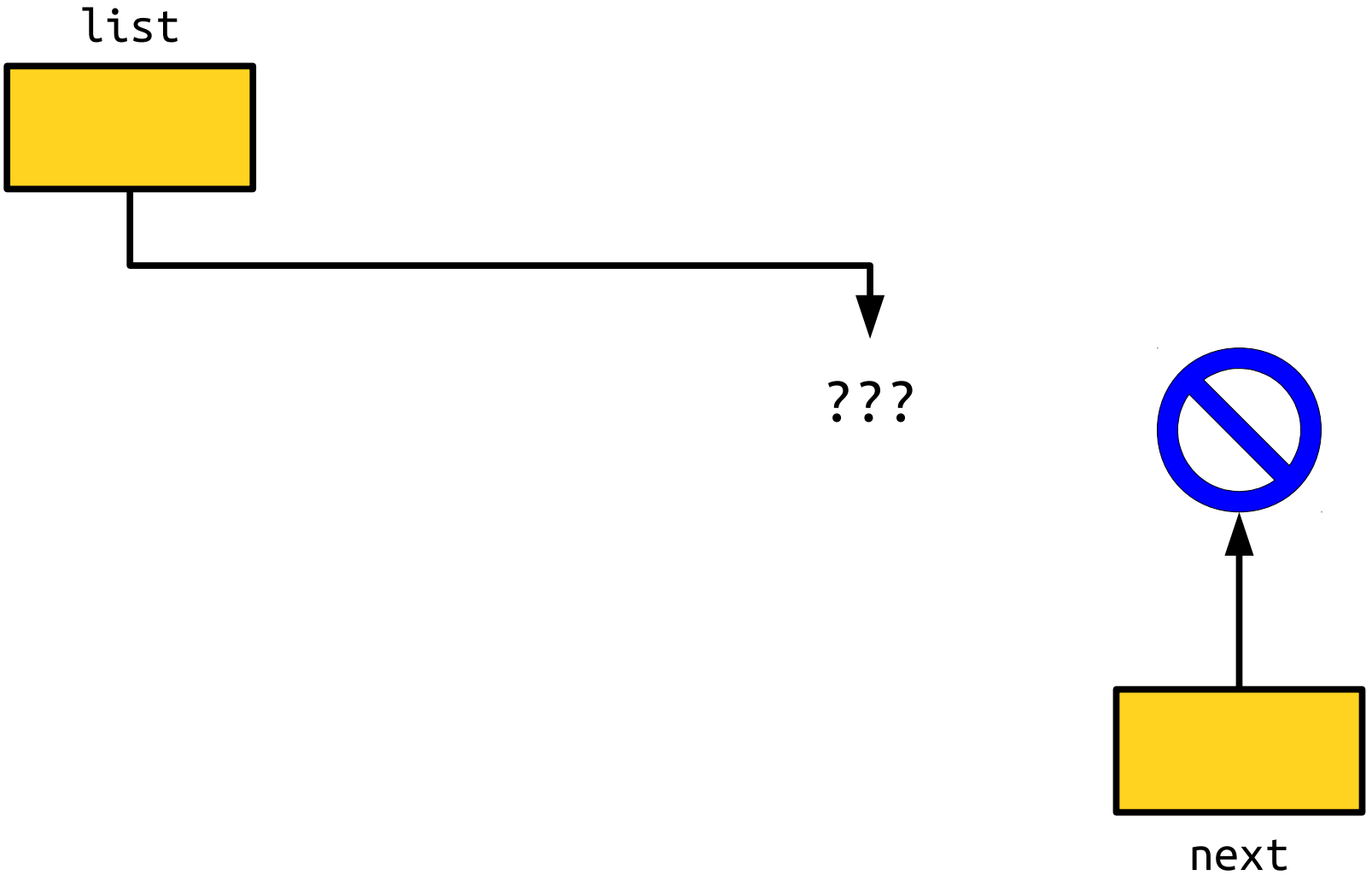
```
while (list != nullptr) {
    Cell* next = list->next;
    delete list;
    list = next;
}
```



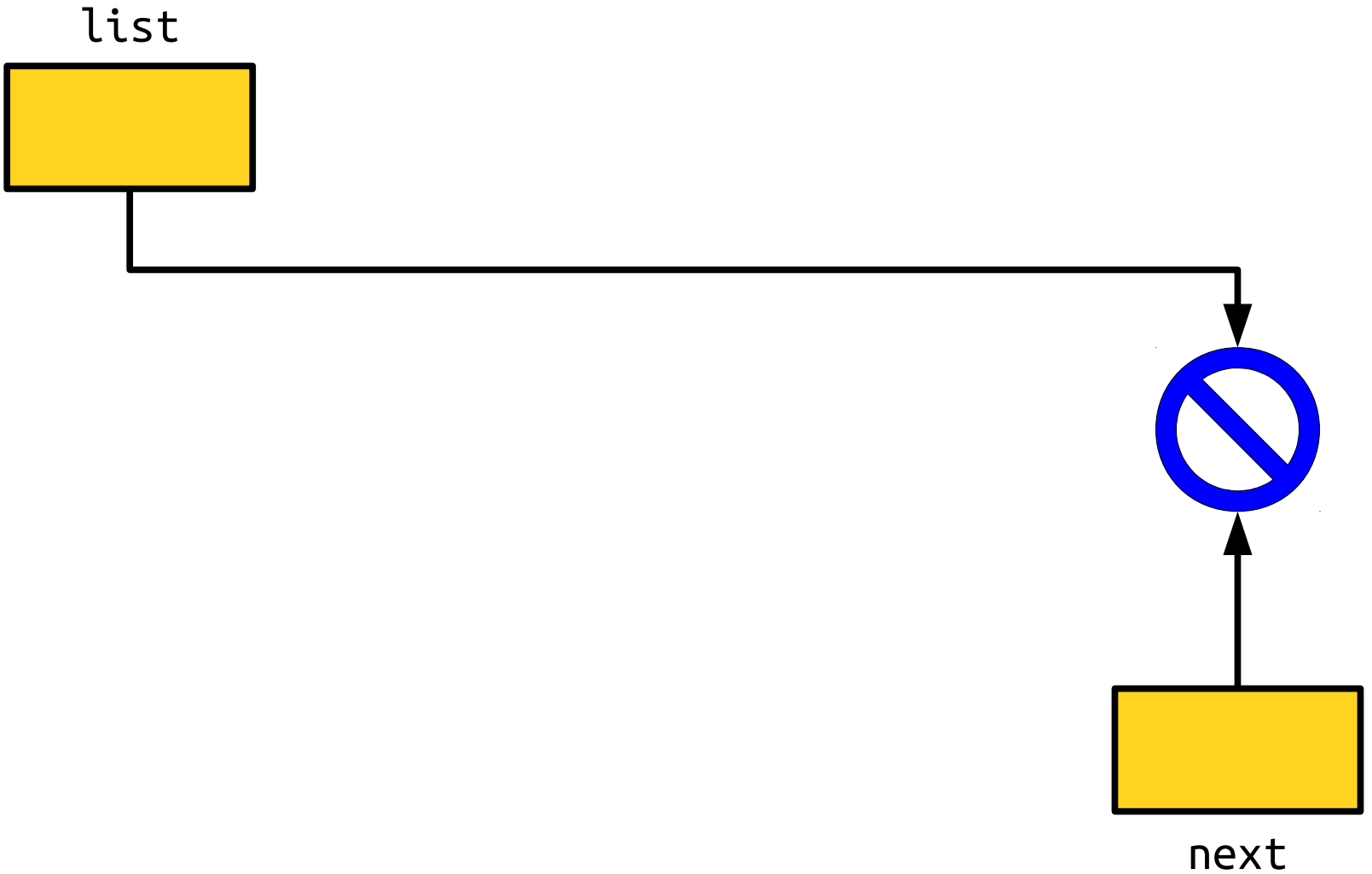
```
while (list != nullptr) {
    Cell* next = list->next;
    delete list;
    list = next;
}
```



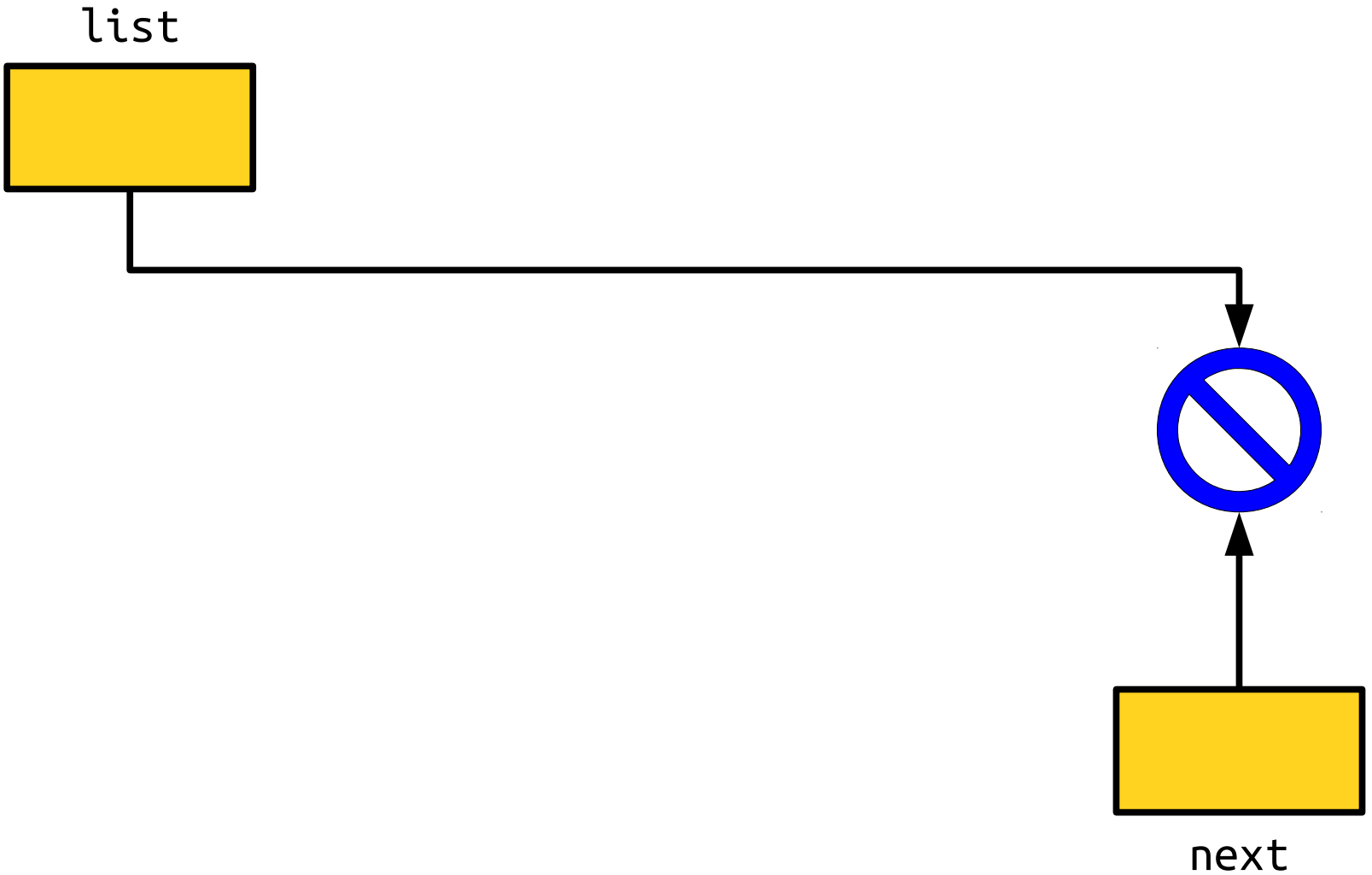
```
while (list != nullptr) {
    Cell* next = list->next;
    delete list;
    list = next;
}
```



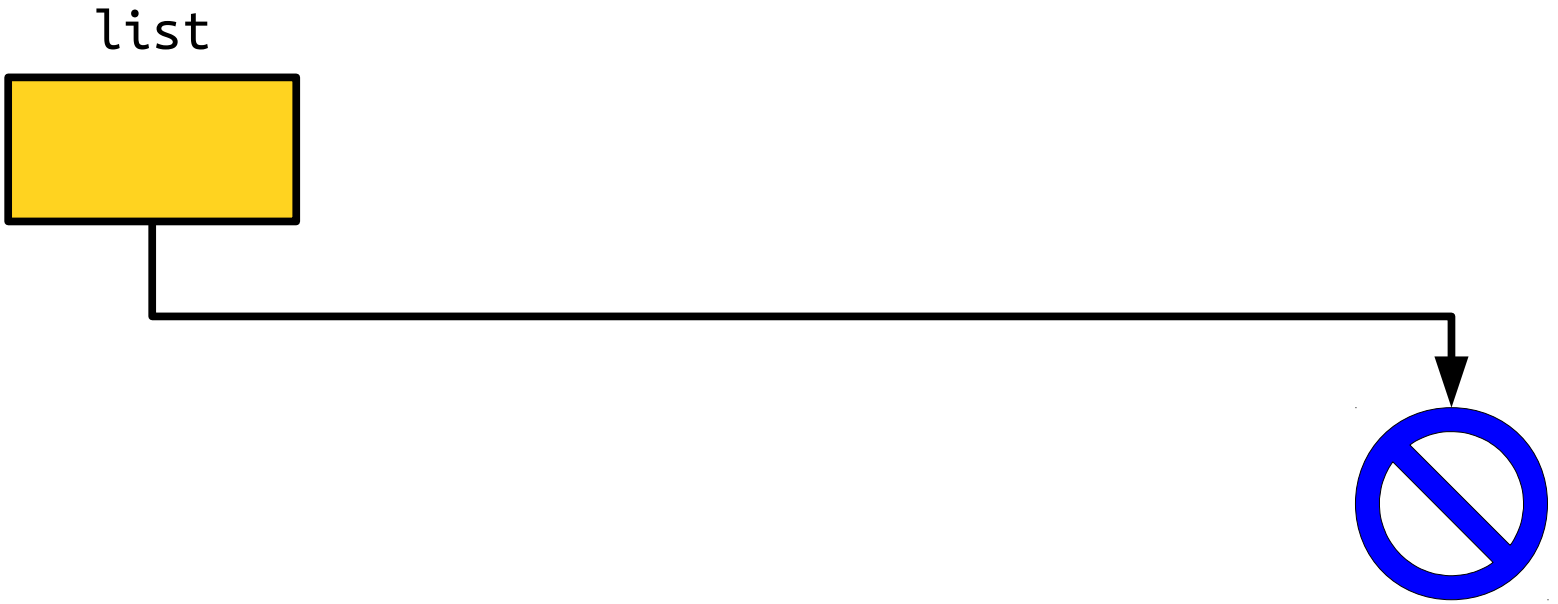
```
while (list != nullptr) {
    Cell* next = list->next;
    delete list;
    list = next;
}
```



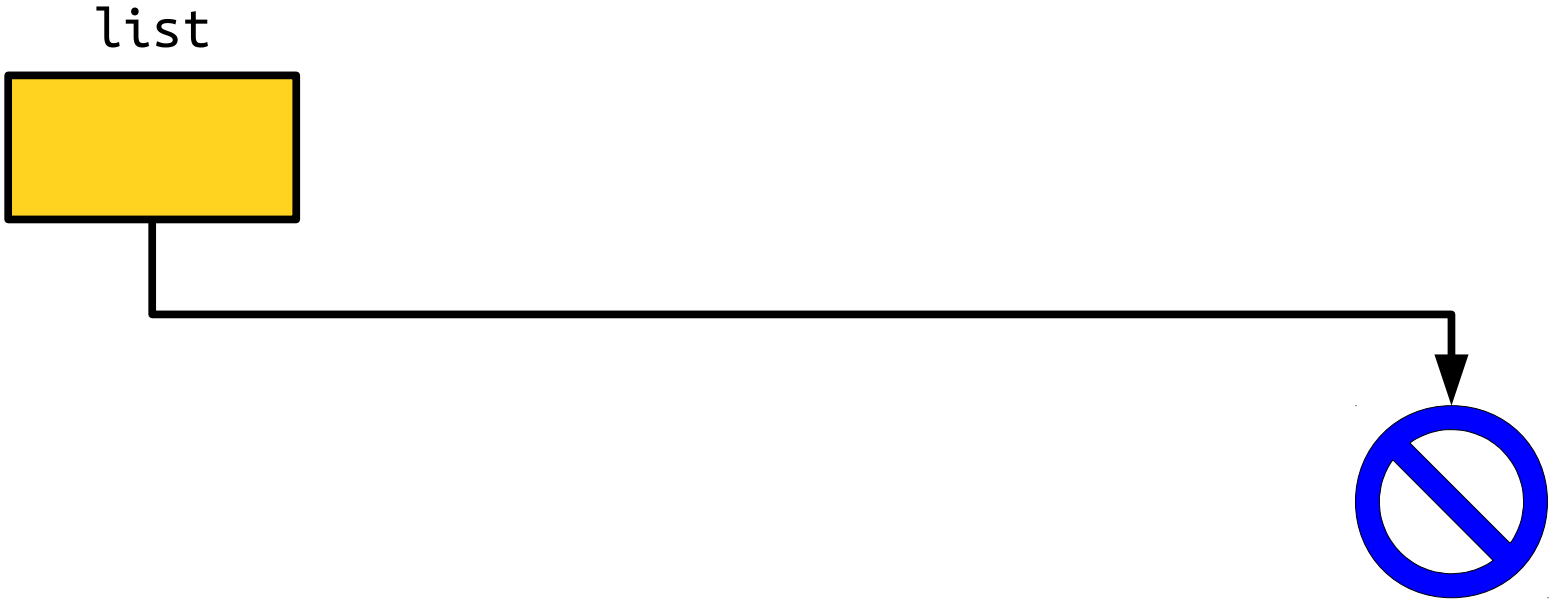
```
while (list != nullptr) {
    Cell* next = list->next;
    delete list;
    list = next;
}
```



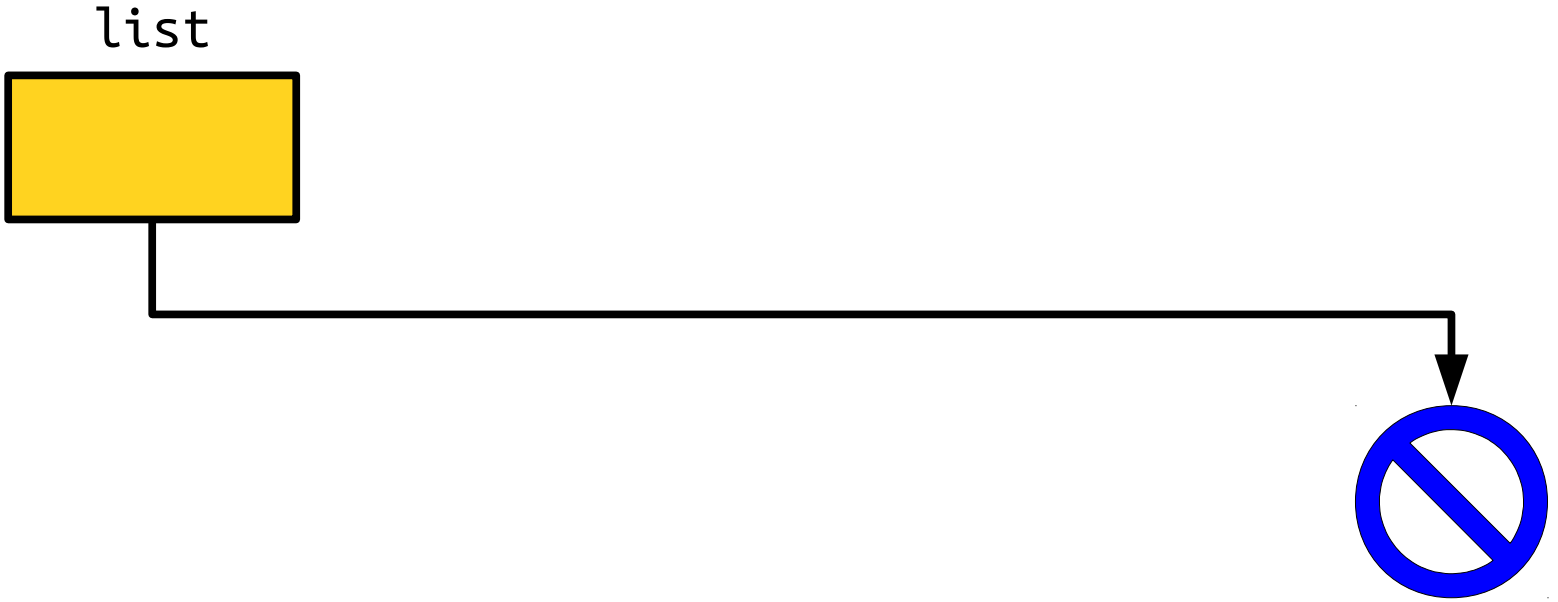
```
while (list != nullptr) {
    Cell* next = list->next;
    delete list;
    list = next;
}
```



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

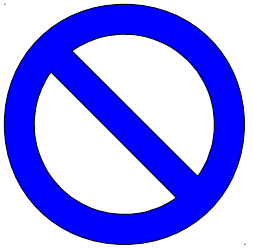


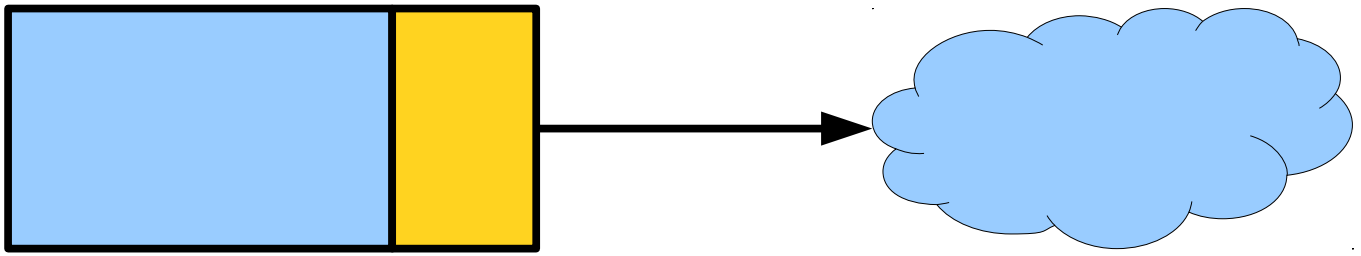
```
while (list != nullptr) {
    Cell* next = list->next;
    delete list;
    list = next;
}
```



A Linked List is Either…

…an empty list, represented by `nullptr`, or…





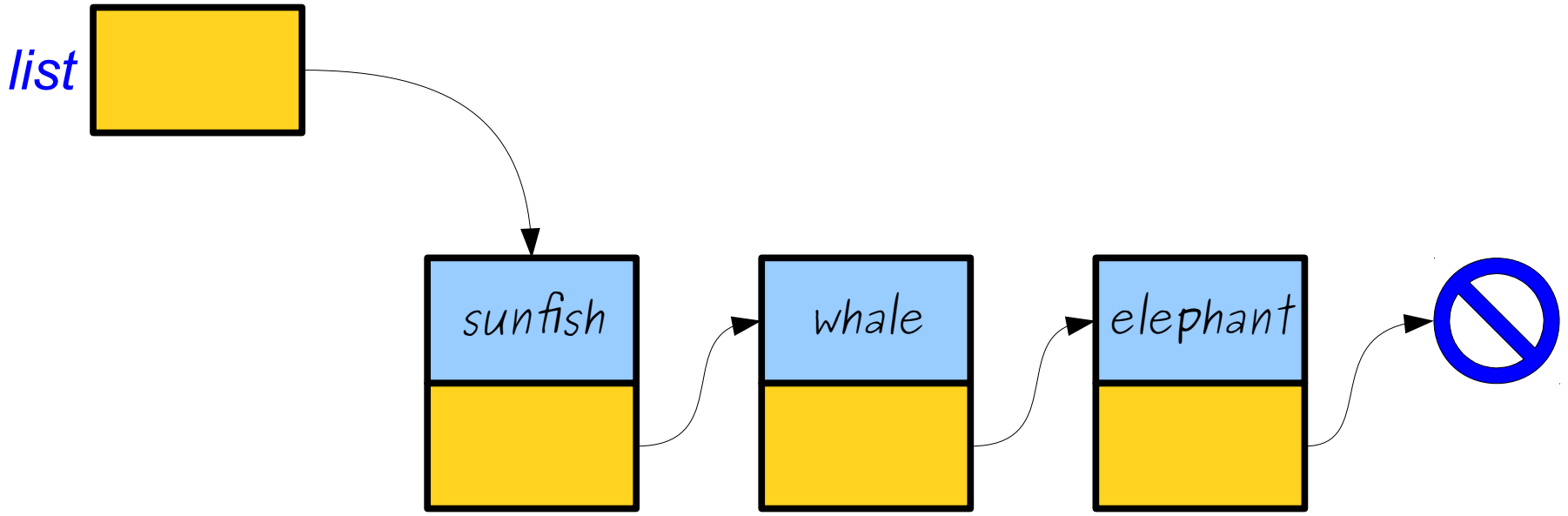
a single linked list cell that points…

… at another linked list.

Pointers and References

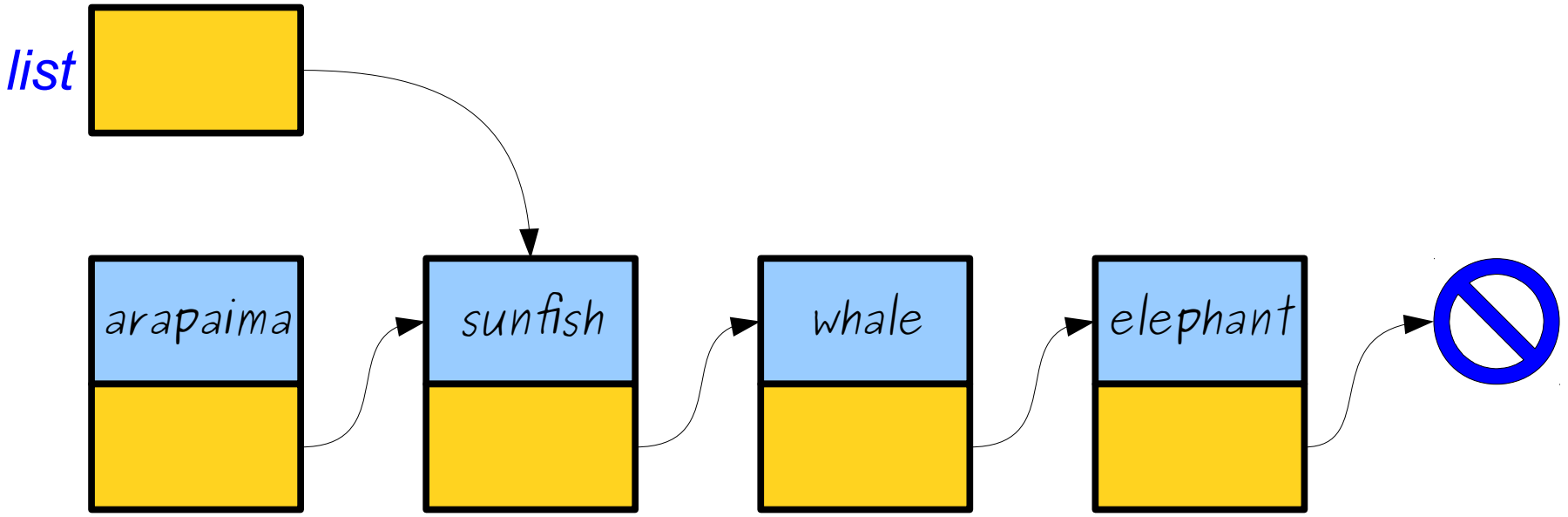
Prepending an Element

- 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?



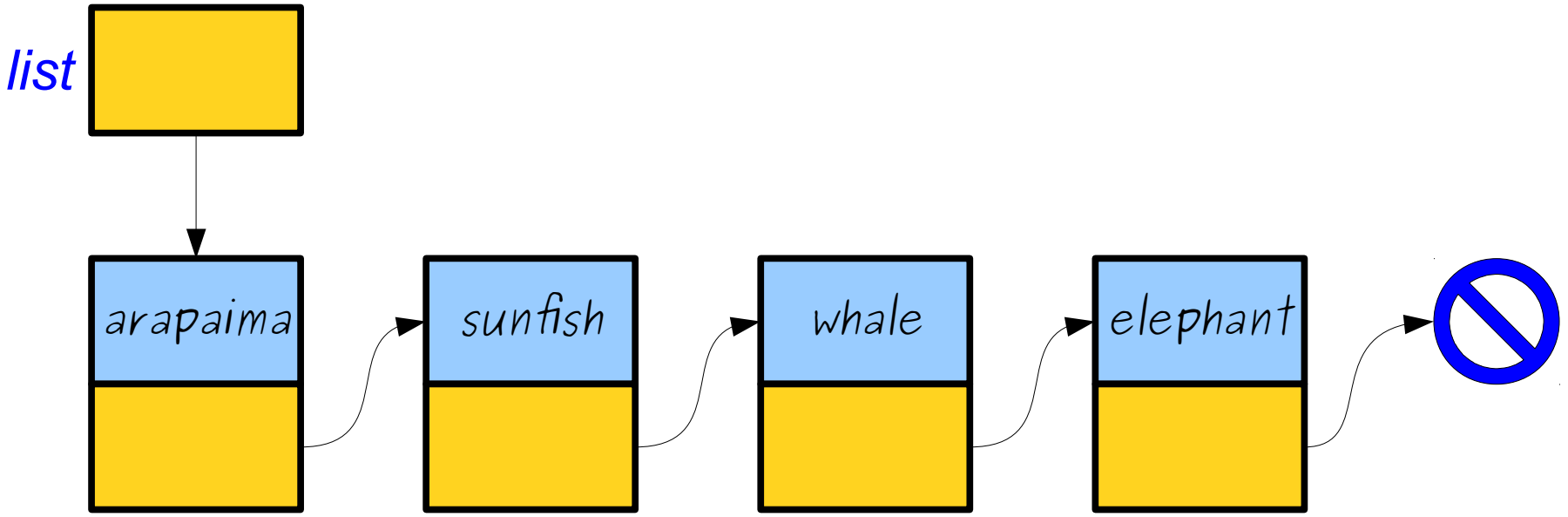
Prepending an Element

- 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?

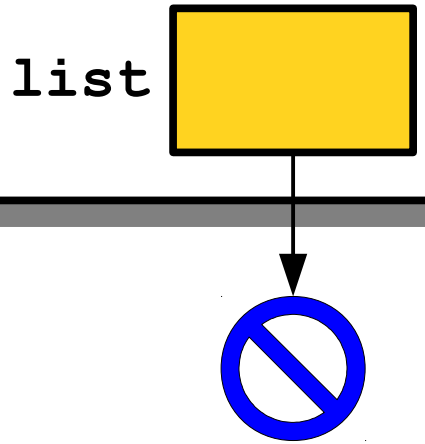


What went wrong?

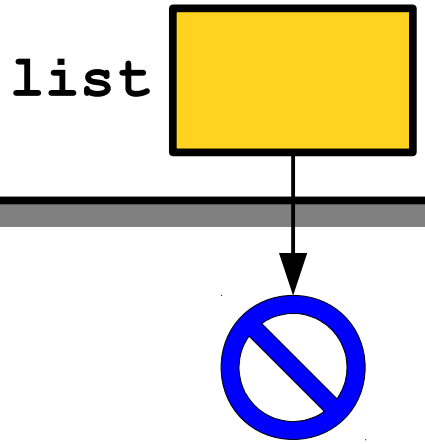
```
int main() {
    Cell* list = nullptr;
    listInsert(list, "Sartre");
    listInsert(list, "Camus");
    listInsert(list, "Nietzsche");

    return 0;
}
```

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

    return 0;
}
```

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

    return 0;
}
```

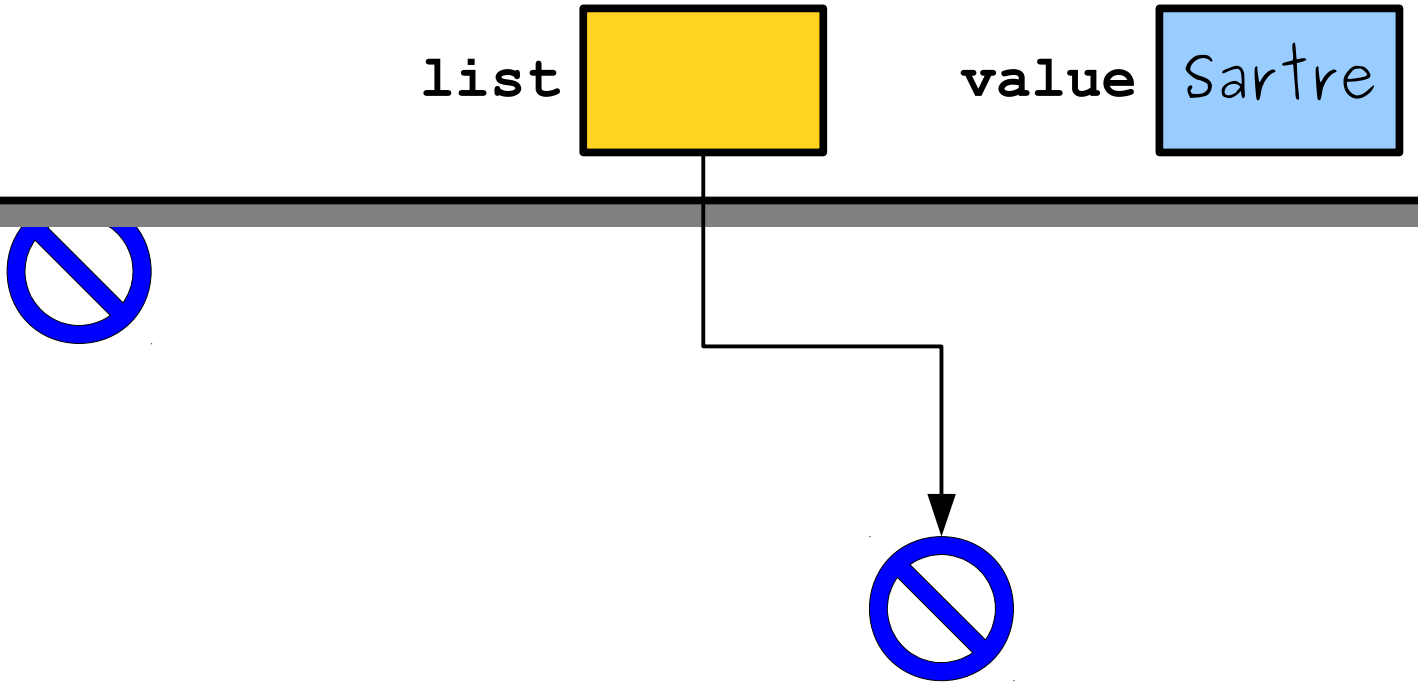


```
int main() {
    Cell* list = nullptr;
    listInsert(list, "Sartre");
    listInsert(list, "Camus");
    listInsert(list, "Nietzsche");

    return 0;
}
```



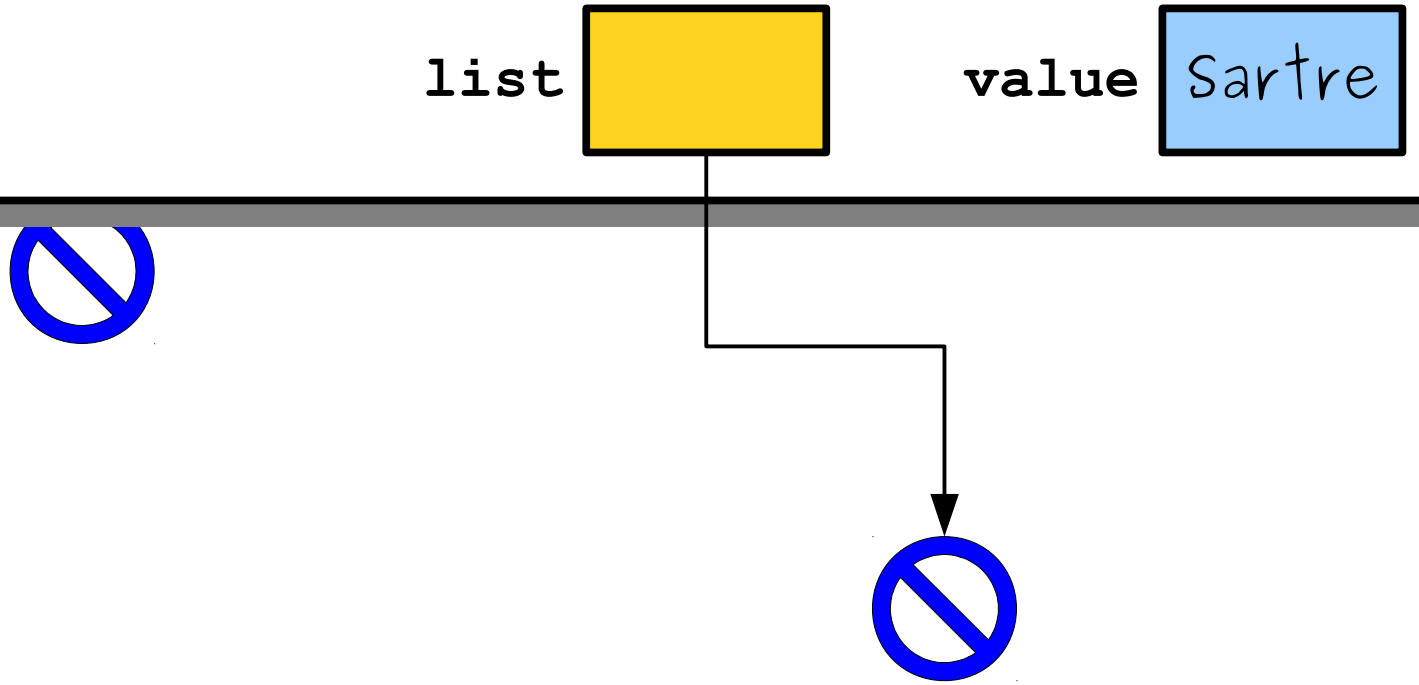
```
int main() {
}
```

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



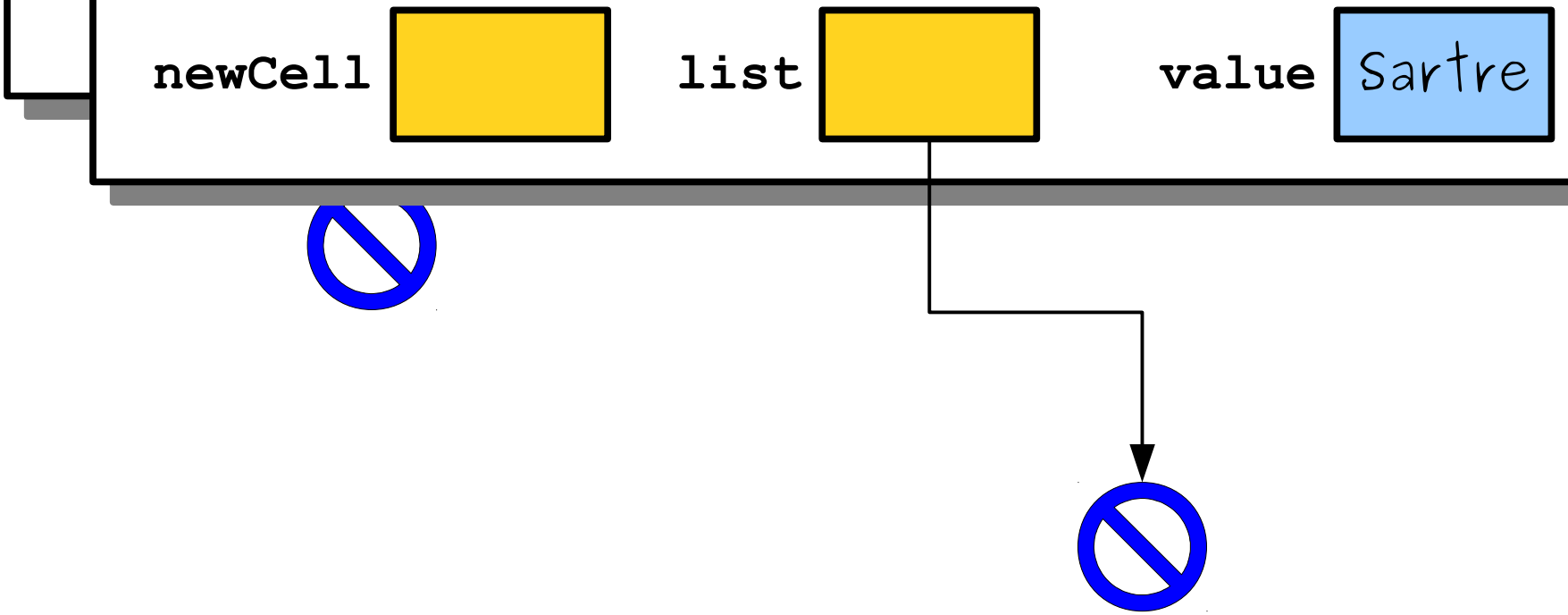
```
int main() {
}
```

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

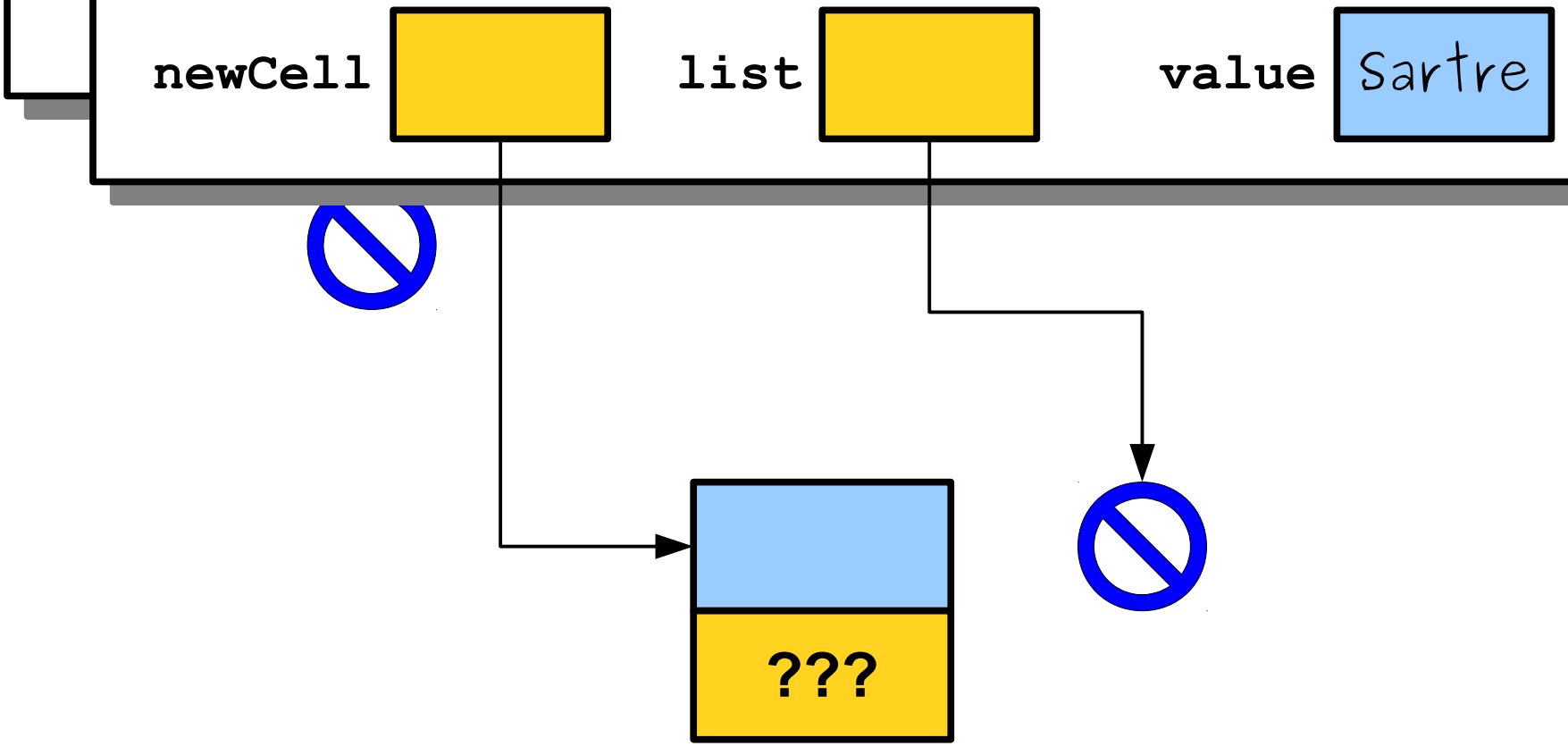


```
int main() {
}
```

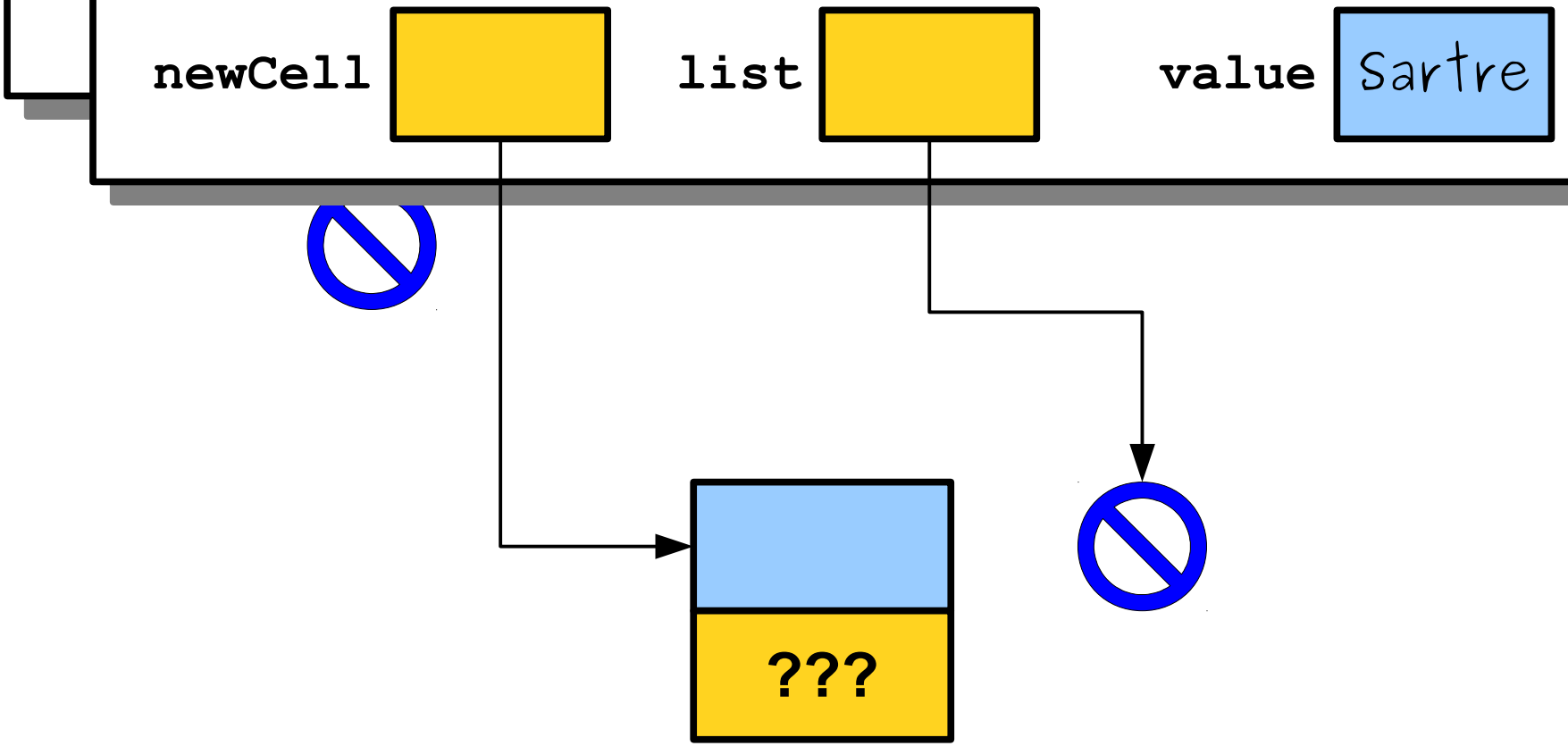
```
void listInsert(Cell* list, const string& value) {
    Cell* newCell = new Cell;
    newCell->value = value;
    newCell->next = list;
    list = newCell;
}
```



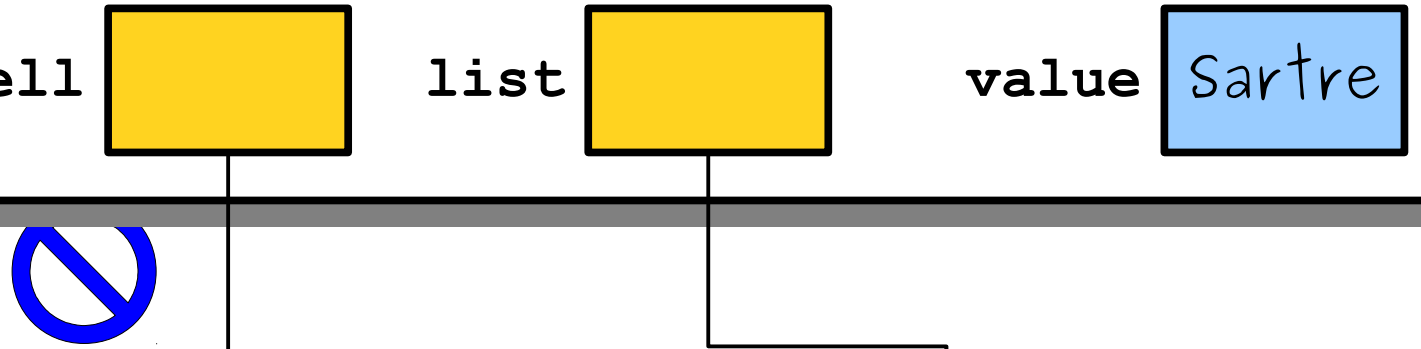
```
int main() {
    void listInsert(Cell* list, const string& value) {
        Cell* newCell = new Cell;
        newCell->value = value;
        newCell->next = list;
        list = newCell;
    }
}
```

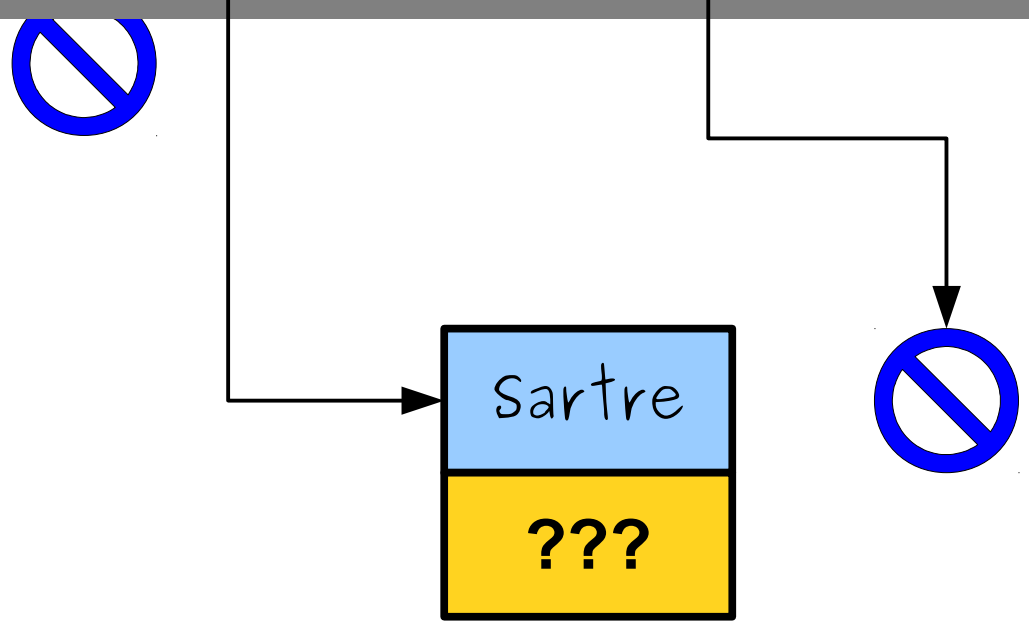


```
int main() {
    void listInsert(Cell* list, const string& value) {
        Cell* newCell = new Cell;
        newCell->value = value;
        newCell->next = list;
        list = newCell;
    }
}
```



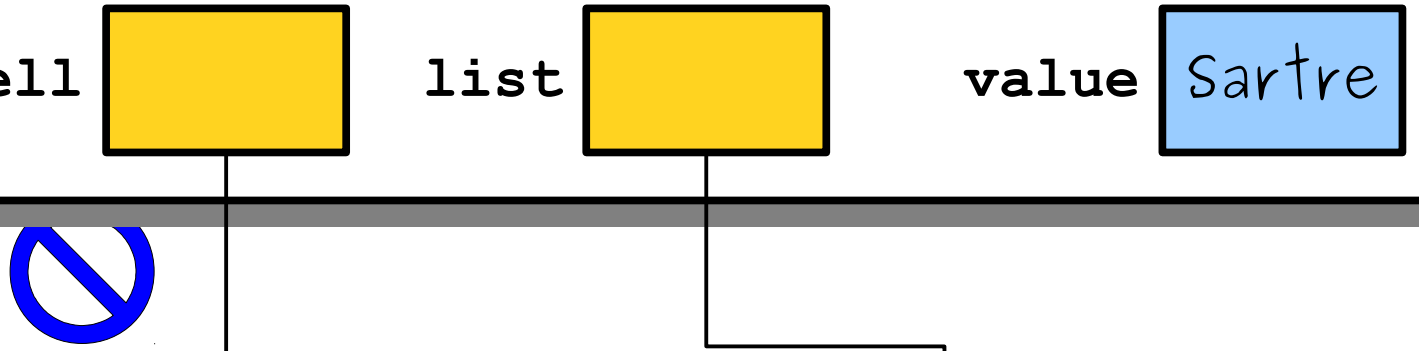
```
int main() {
    void listInsert(Cell* list, const string& value) {
        Cell* newCell = new Cell;
        newCell->value = value;
        newCell->next = list;
        list = newCell;
    }
}
```

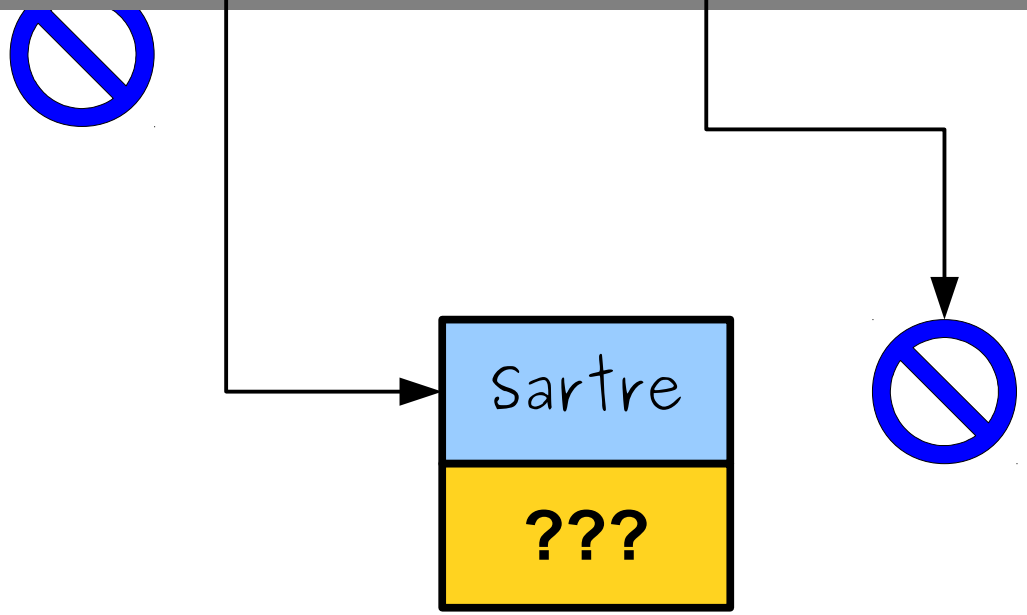




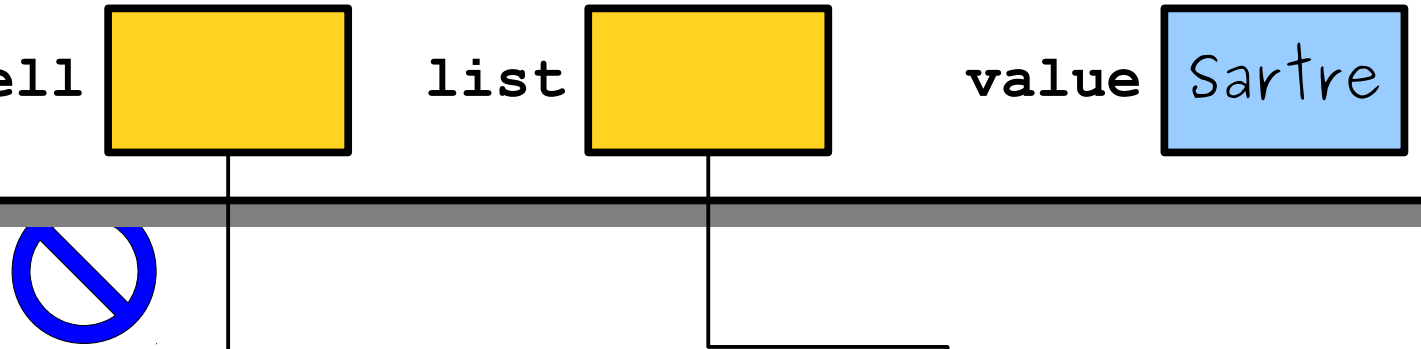
```
int main() {
}
```

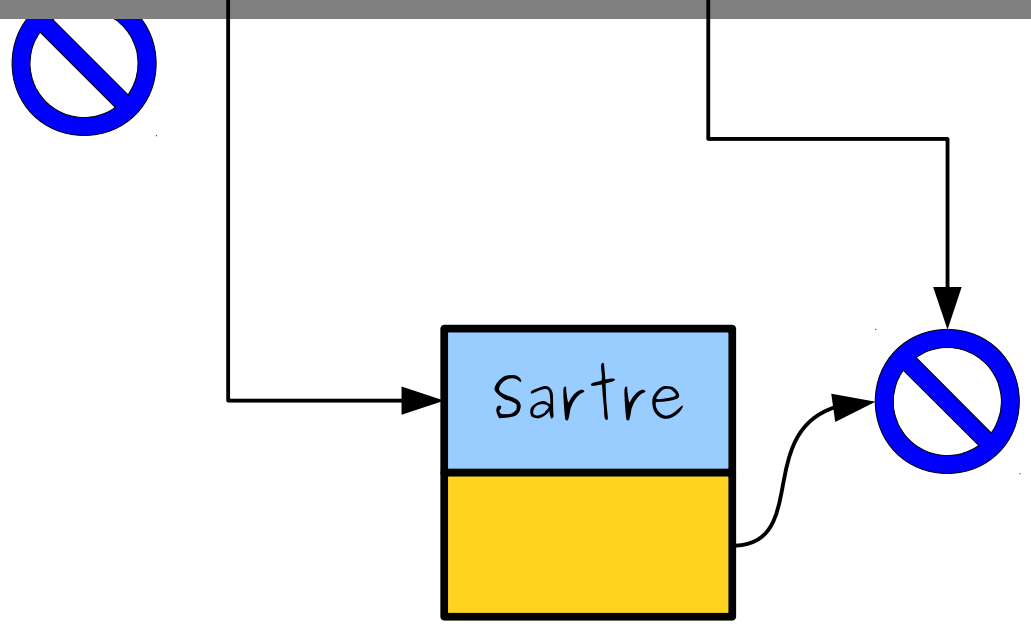
```
void listInsert(Cell* list, const string& value) {
    Cell* newCell = new Cell;
    newCell->value = value;
    newCell->next = list;
    list = newCell;
}
```





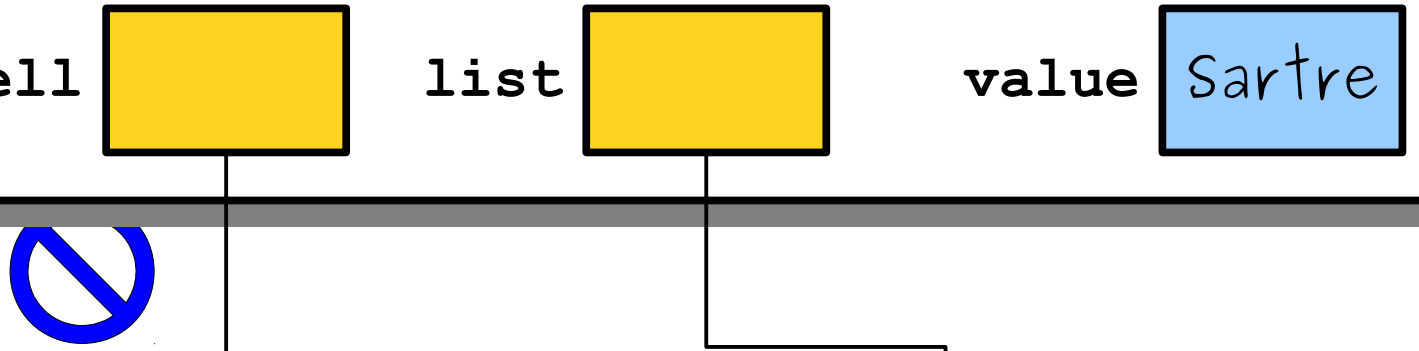
```
int main() {
    void listInsert(Cell* list, const string& value) {
        Cell* newCell = new Cell;
        newCell->value = value;
        newCell->next = list;
        list = newCell;
    }
}
```

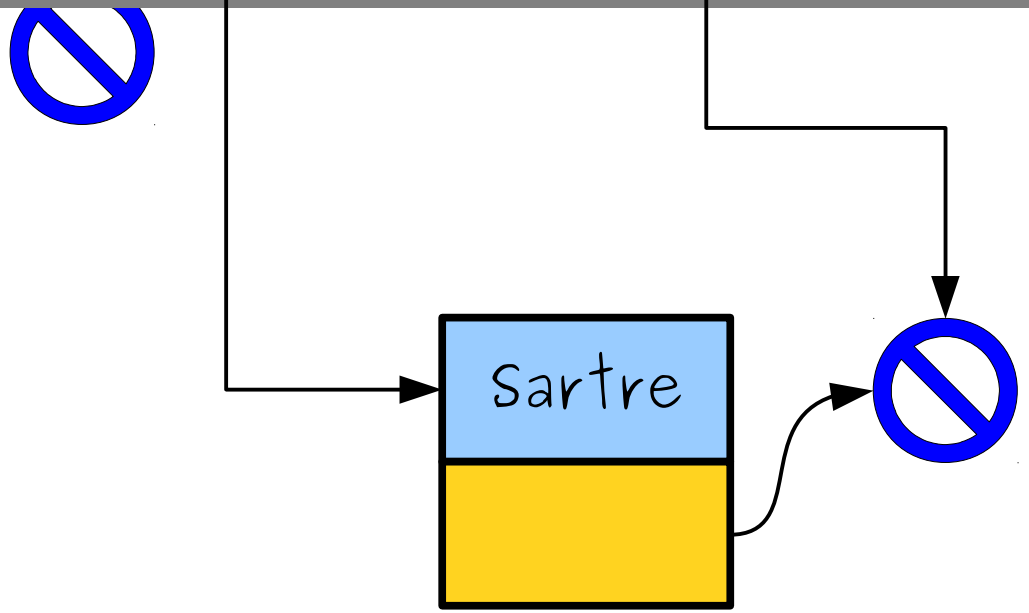




```
int main() {
}
```

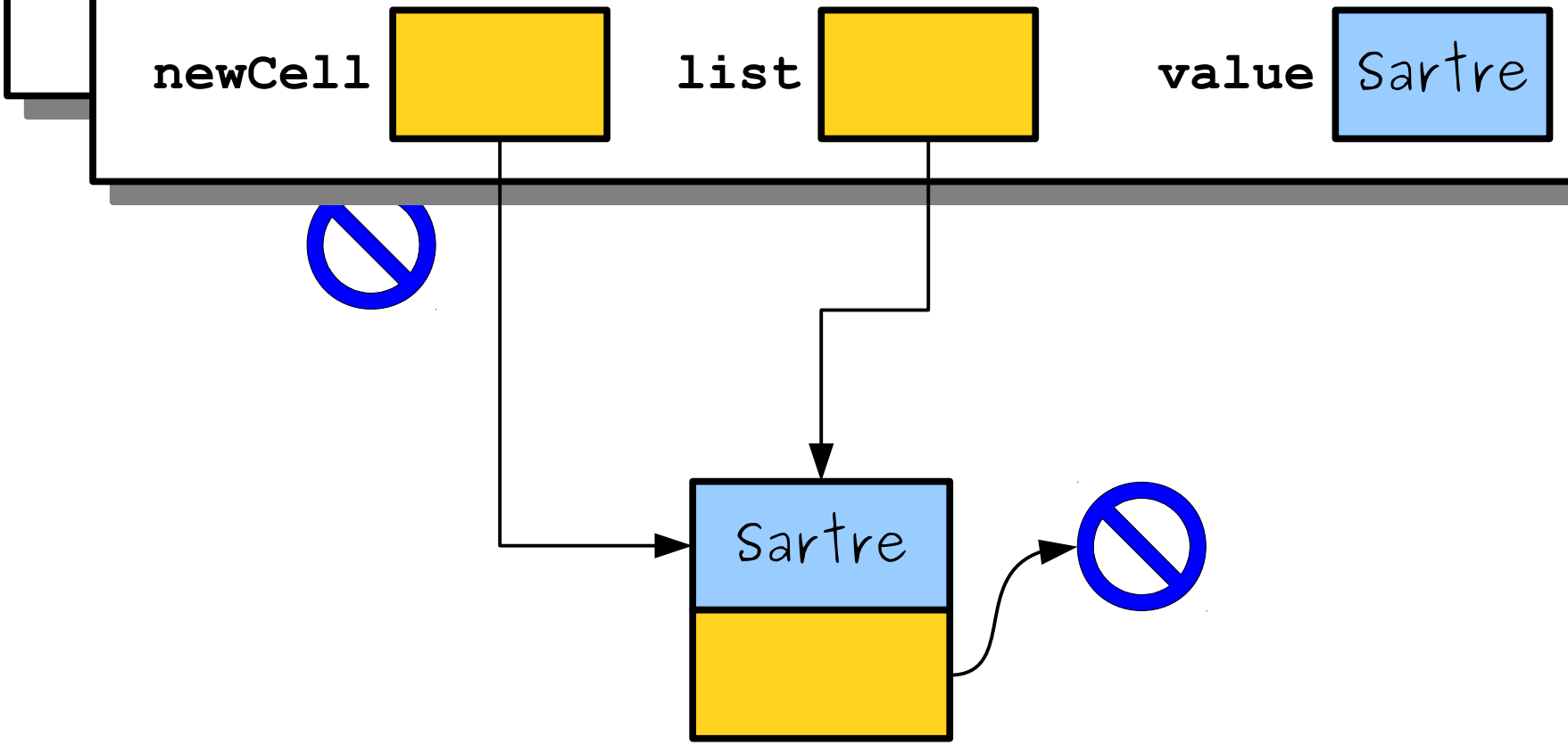
```
void listInsert(Cell* list, const string& value) {
    Cell* newCell = new Cell;
    newCell->value = value;
    newCell->next = list;
    list = newCell;
}
```



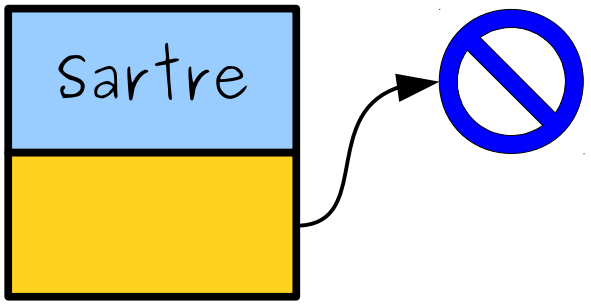


```
int main() {
}
```

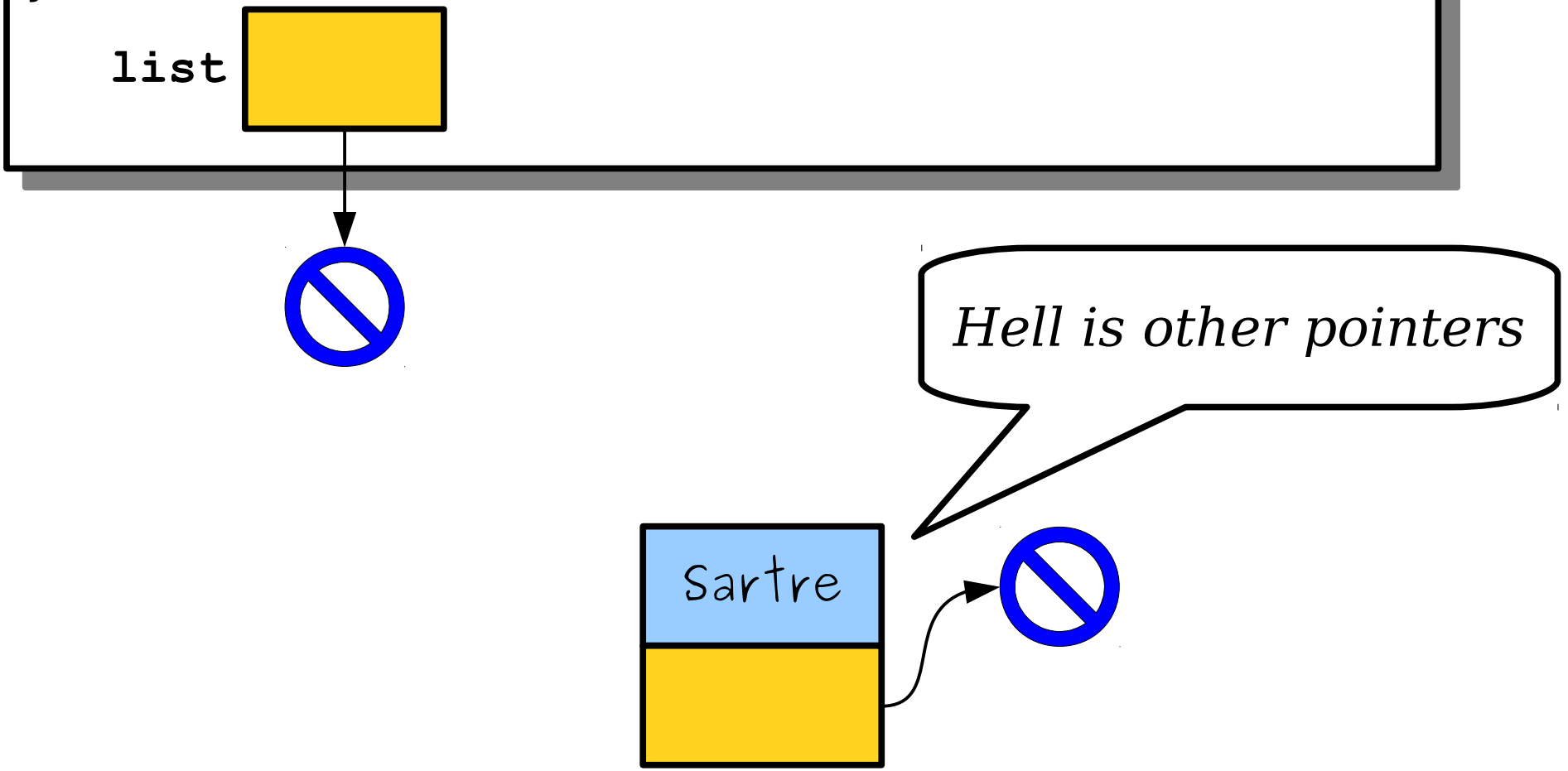
```
void listInsert(Cell* list, const string& value) {
    Cell* newCell = new Cell;
    newCell->value = value;
    newCell->next = list;
    list = newCell;
}
```



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

    return 0;
}
```



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

    return 0;
}
```



Pointers by Reference

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

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

Pointers by Reference

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

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

Pointers by Reference

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

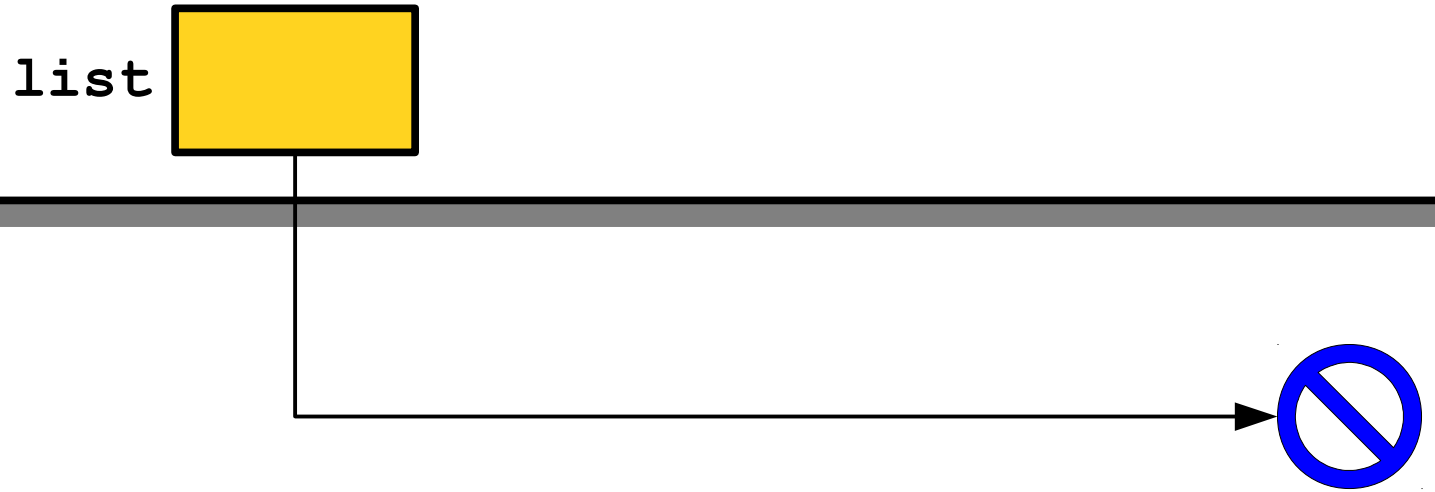
```
void listInsert(Cell*& list, const string& value) {
    Cell* newCell = new Cell;
    newCell->value = value;
    newCell->next = list;
    list = newCell;
}
```

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

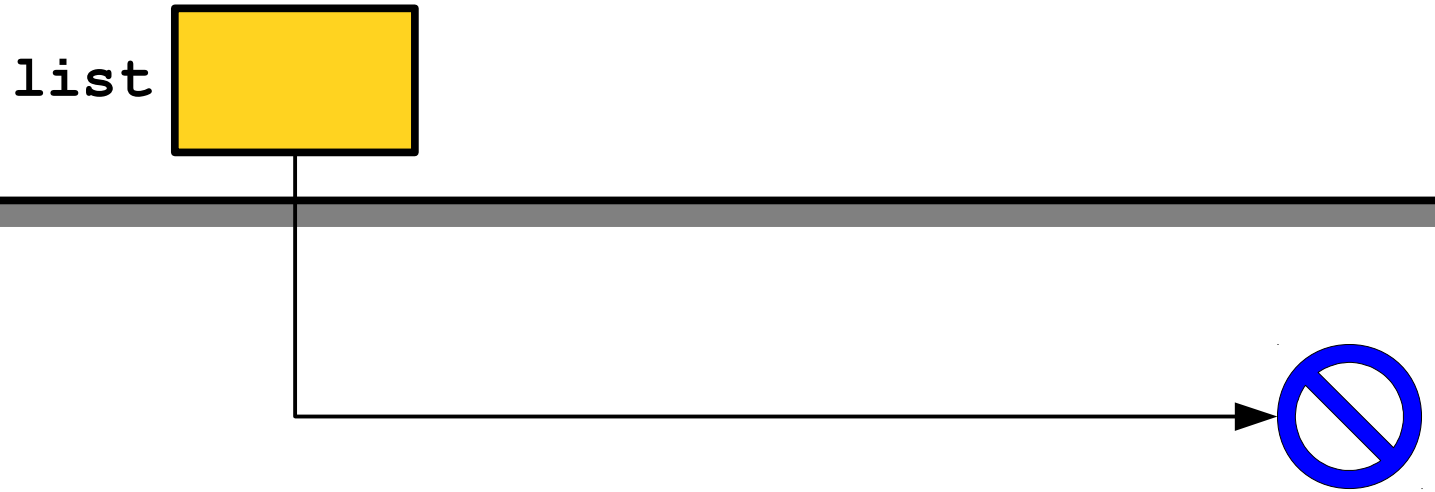
```
int main() {
    Cell* list = nullptr;
    listInsert(list, "The Turtles");
    listInsert(list, "The Beatles");
    listInsert(list, "A Flock of Seagulls");

    return 0;
}
```

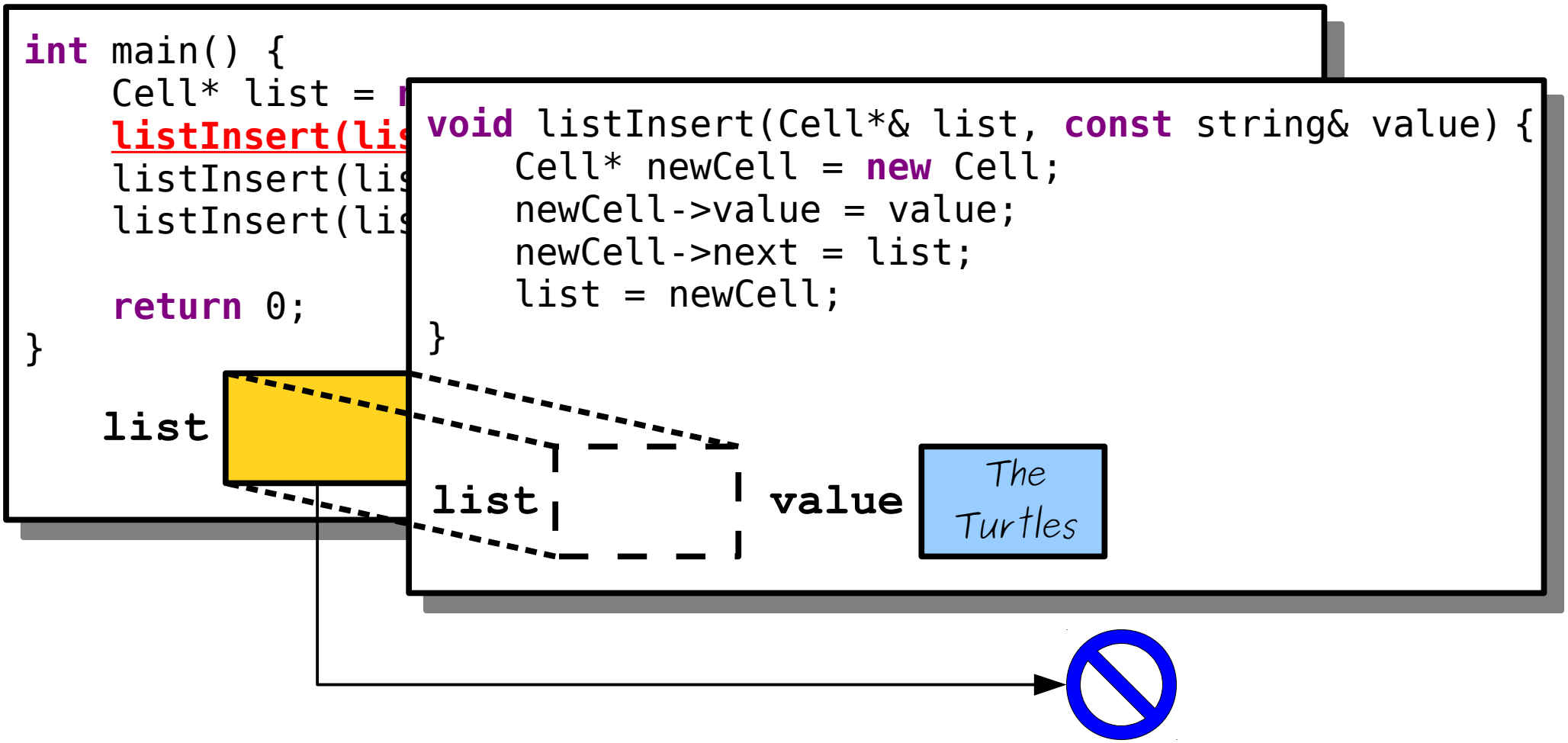
```
int main() {
    Cell* list = nullptr;
    listInsert(list, "The Turtles");
    listInsert(list, "The Beatles");
    listInsert(list, "A Flock of Seagulls");

    return 0;
}
```

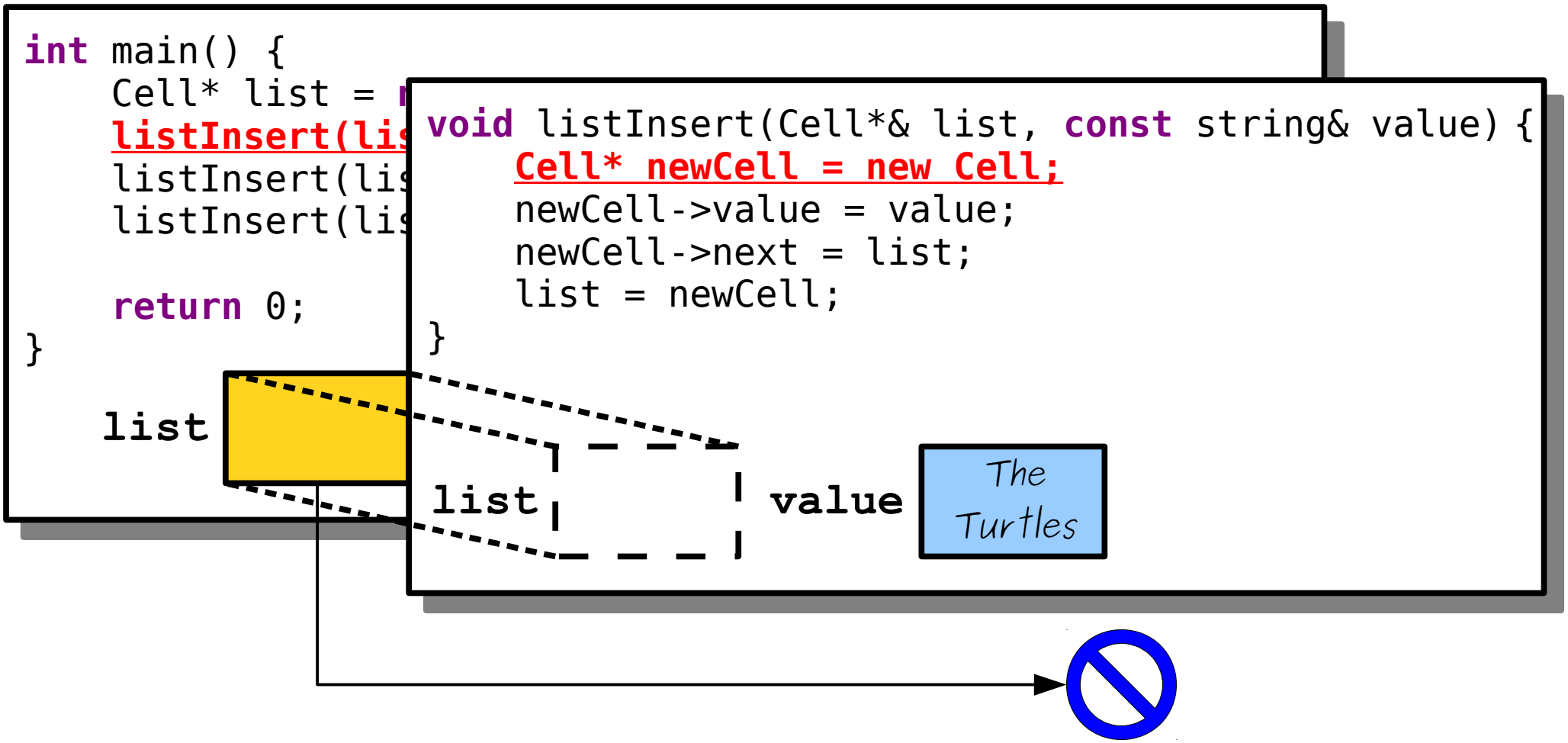
```
int main() {
    Cell* list = nullptr;
    listInsert(list, "The Turtles");
    listInsert(list, "The Beatles");
    listInsert(list, "A Flock of Seagulls");

    return 0;
}
```



```
int main() {
    Cell* list = nullptr;
    listInsert(list, "The Turtles");
    listInsert(list, "The Beatles");
    listInsert(list, "A Flock of Seagulls");

    return 0;
}
```

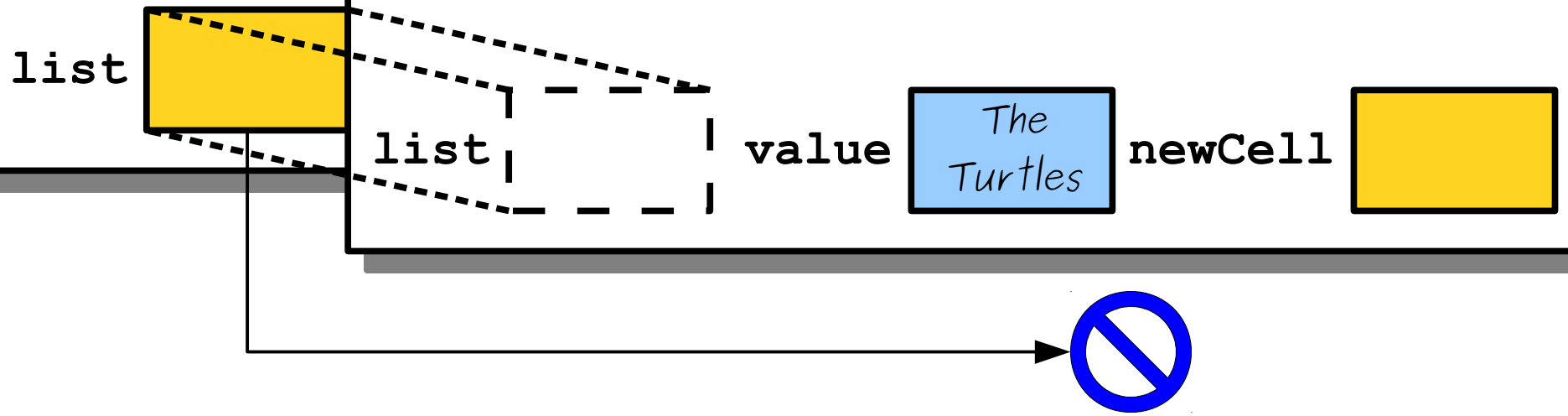


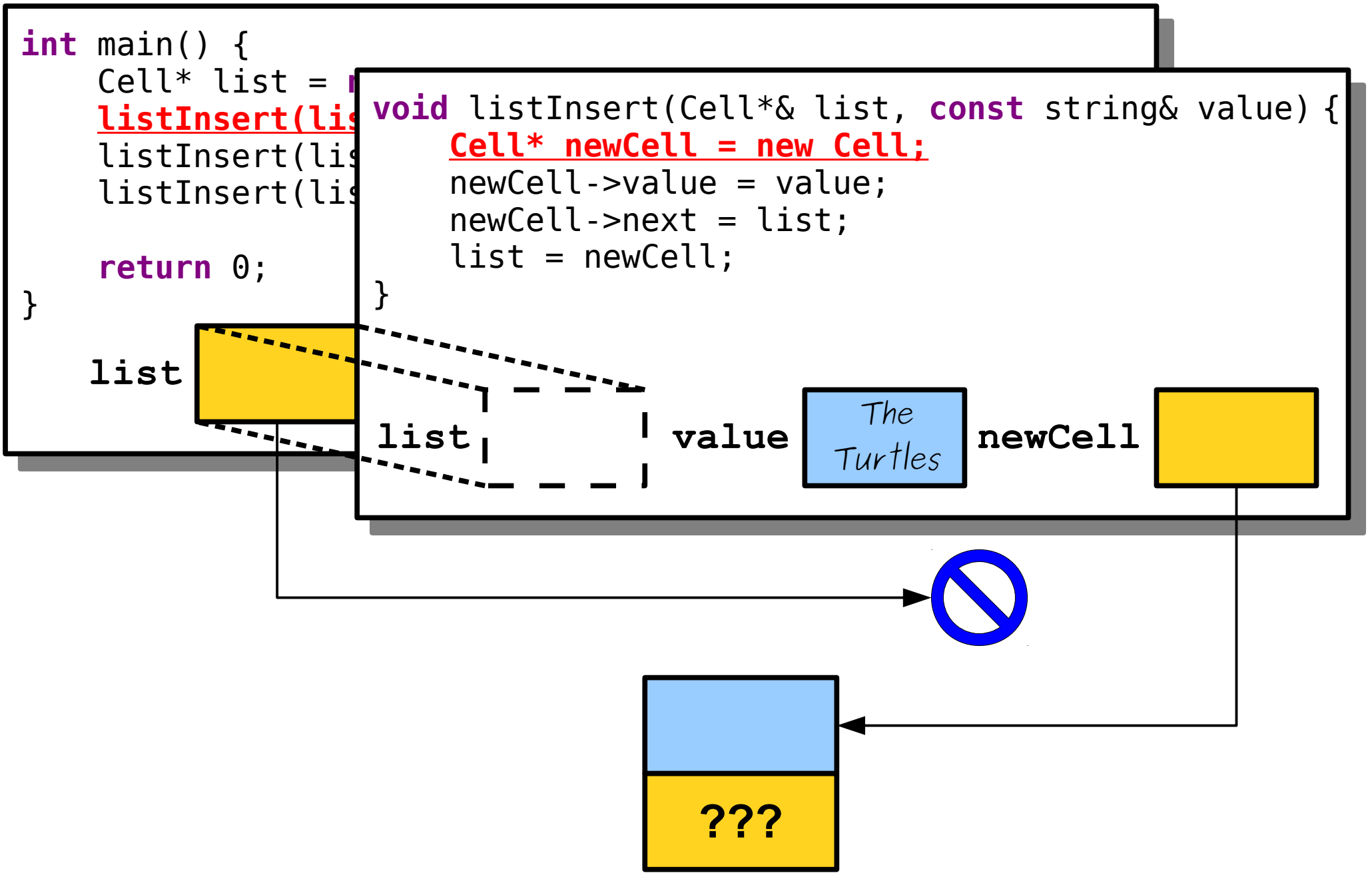

int main() {
Cell* list = n
listInsert(lis
listInsert(lis
listInsert(lis
return 0;
}
void listInsert(Cell*& list, const string& value) {
Cell* newCell = new Cell;
newCell->value = value;
newCell->next = list;
list = newCell;
}
list
list
value
The Turtles

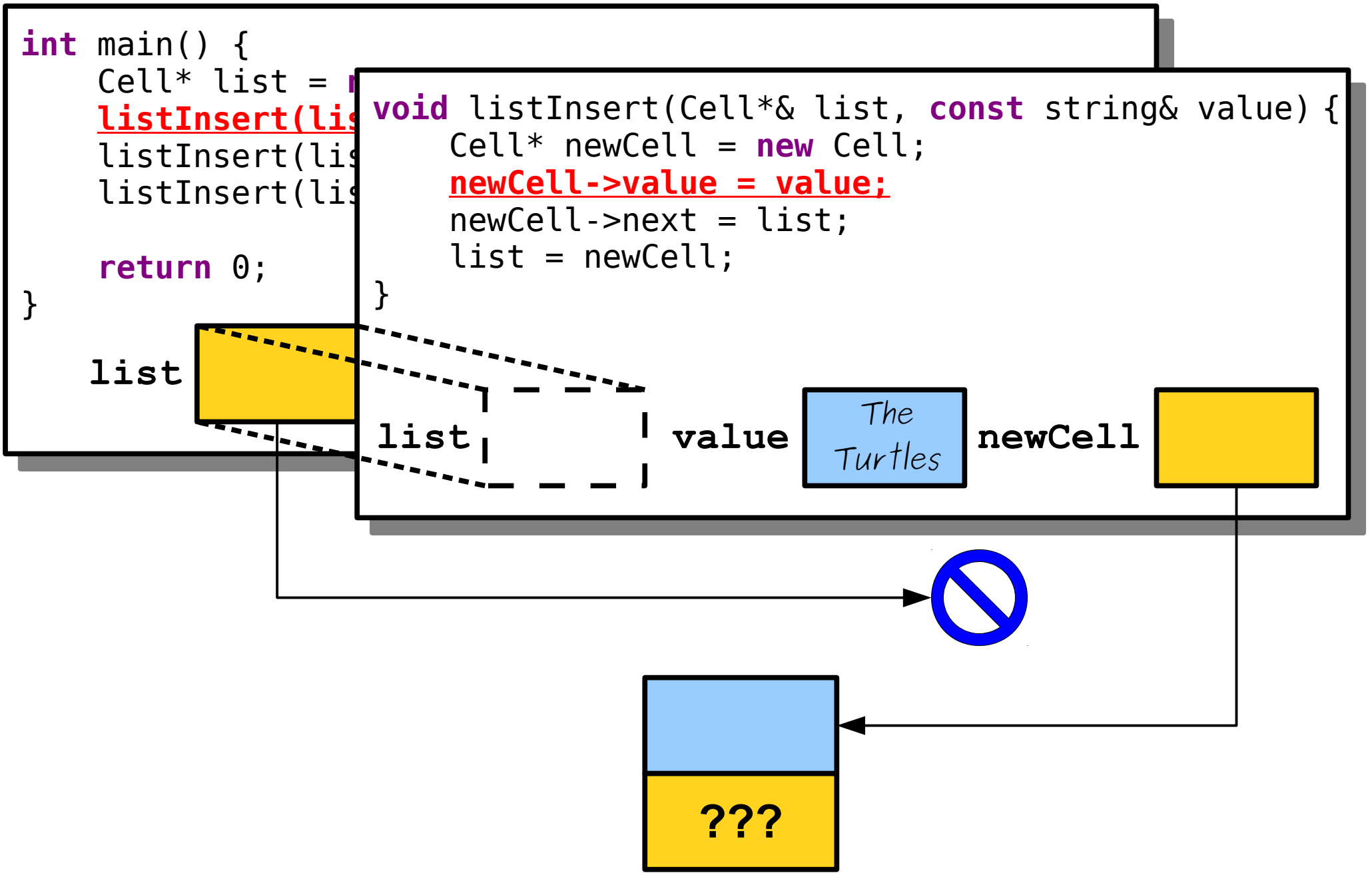

int main() {
Cell* list = n
listInsert(lis
listInsert(lis
listInsert(lis
return 0;
}
void listInsert(Cell*& list, const string& value) {
Cell* newCell = new Cell;
newCell->value = value;
newCell->next = list;
list = newCell;
}
list
list
value
The Turtles

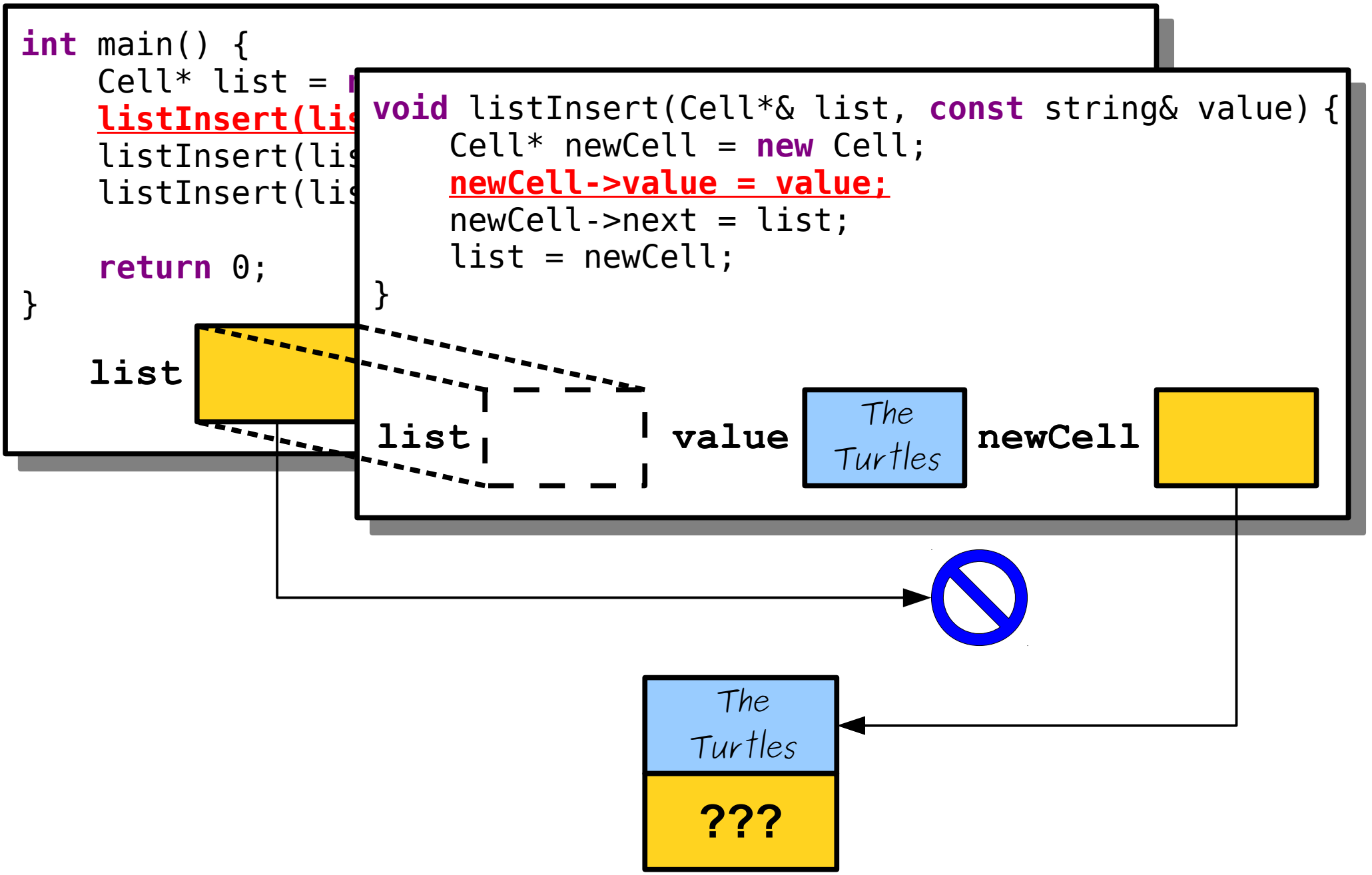
```
int main() {
    Cell* list = n
    listInsert(lis
    listInsert(lis
    listInsert(lis

    return 0;
}
```

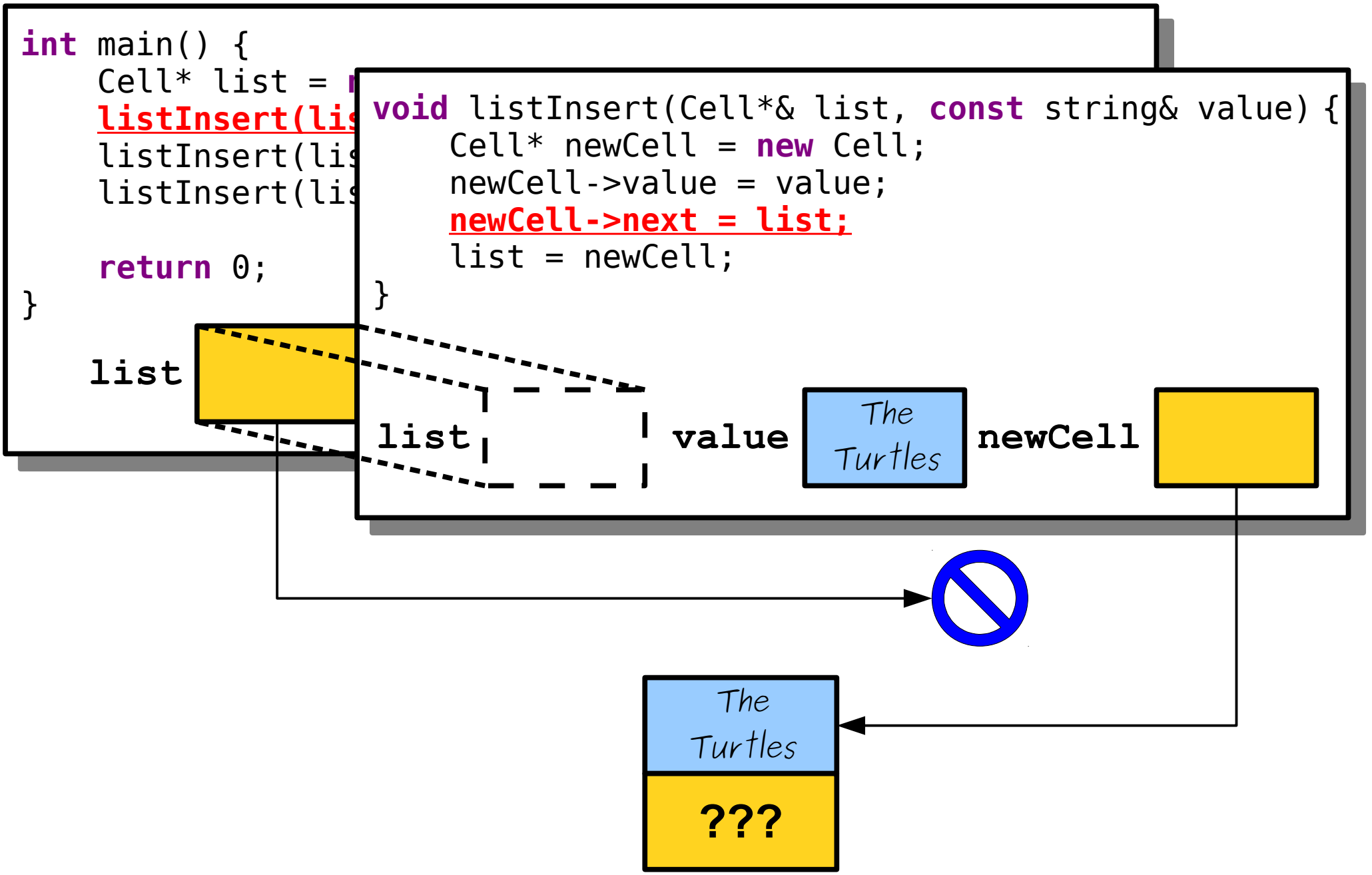
```
void listInsert(Cell*& list, const string& value) {
    Cell* newCell = new Cell;
    newCell->value = value;
    newCell->next = list;
    list = newCell;
}
```

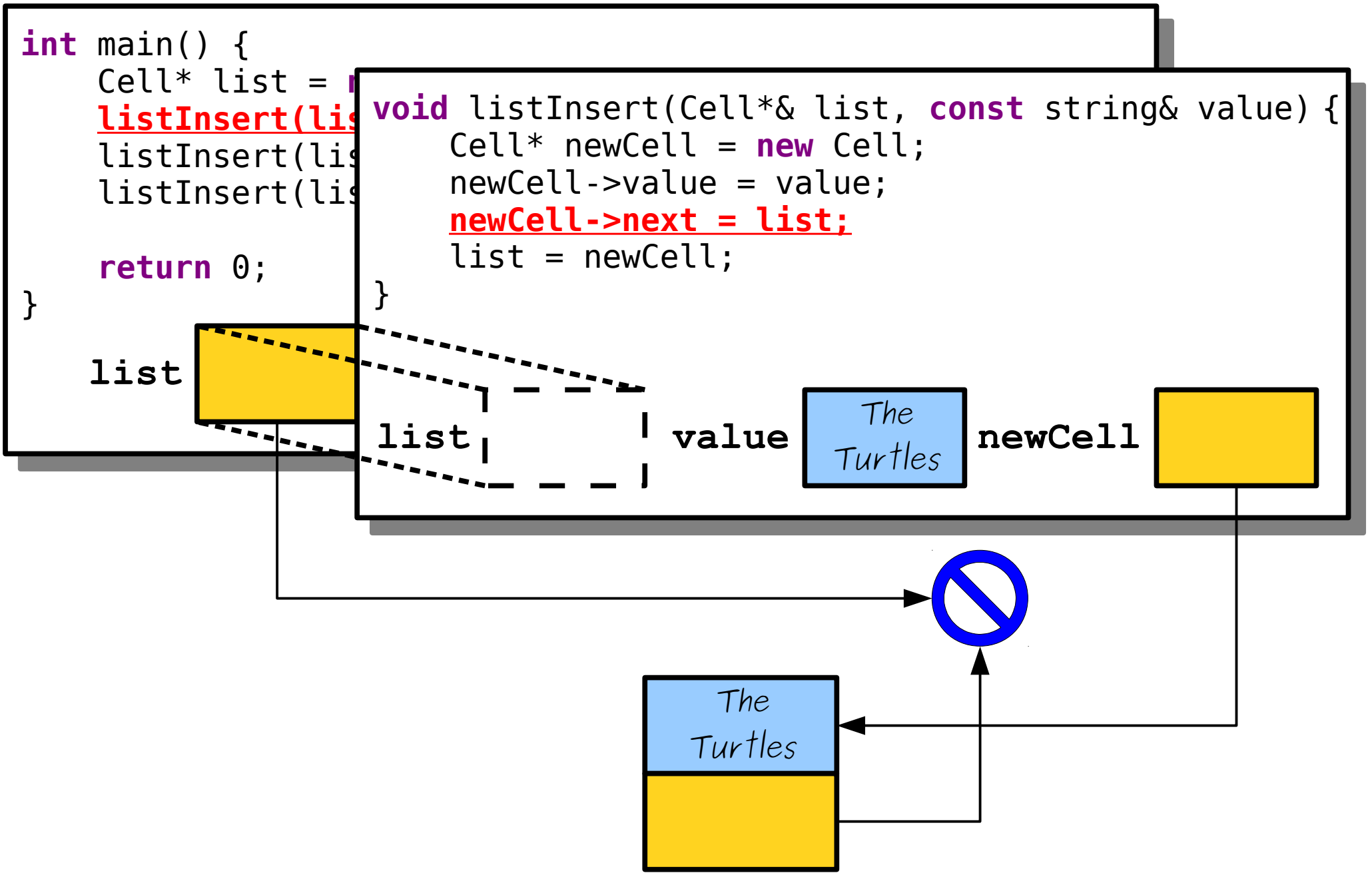


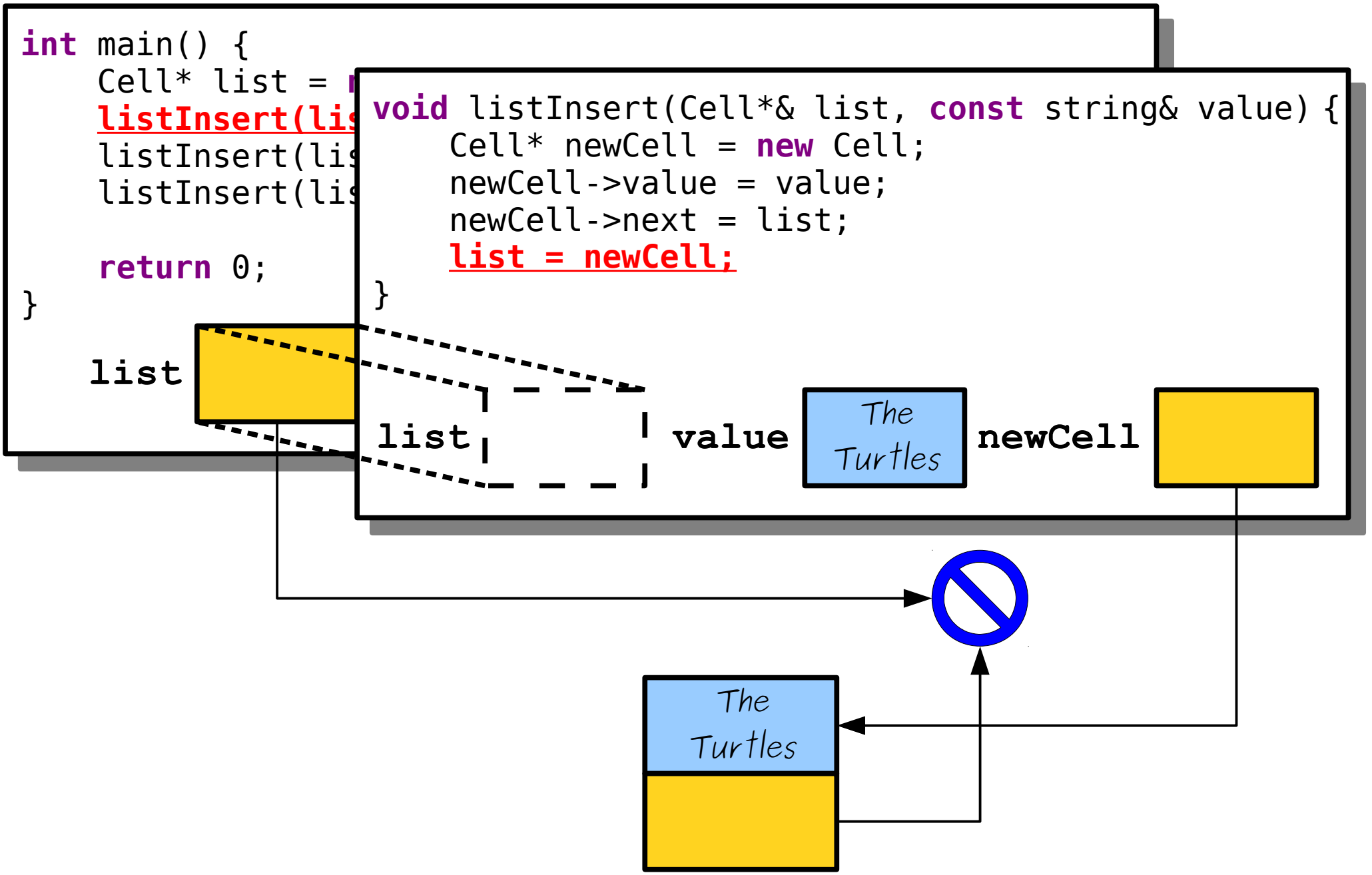

int main() {
Cell* list =
listInsert(lis
listInsert(lis
listInsert(lis
return 0;
}
void listInsert(Cell*& list, const string& value) {
Cell* newCell = new Cell;
newCell->value = value;
newCell->next = list;
list = newCell;
}
list
list
value
The Turtles
newCell
???

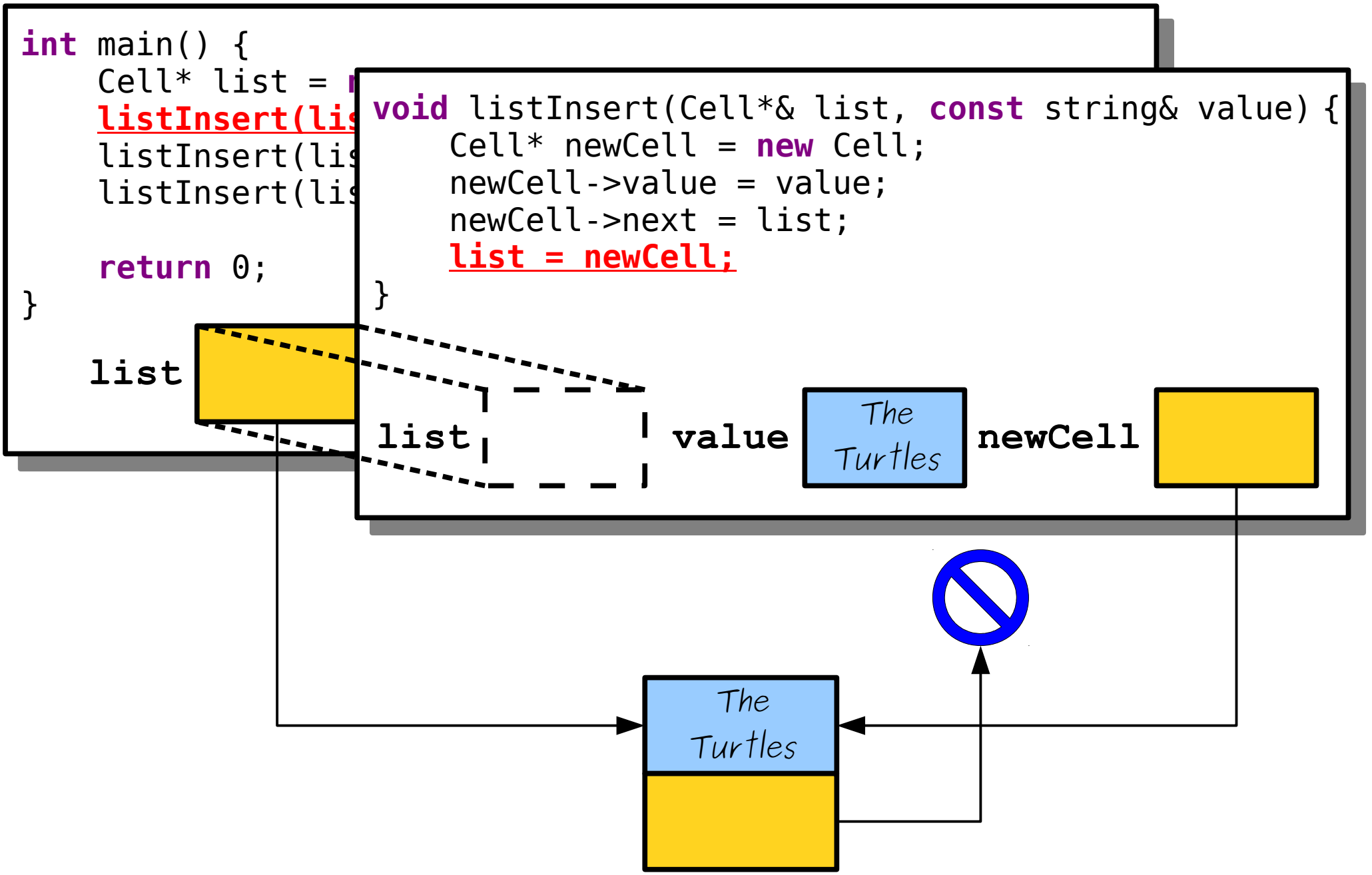

int main() {
Cell* list = n
listInsert(lis
listInsert(lis
listInsert(lis
return 0;
}
list
void listInsert(Cell*& list, const string& value) {
Cell* newCell = new Cell;
newCell->value = value;
newCell->next = list;
list = newCell;
}
list
value
The Turtles
newCell
???


int main() {
Cell* list = n
listInsert(lis
listInsert(lis
listInsert(lis
return 0;
}
list
void listInsert(Cell*& list, const string& value) {
Cell* newCell = new Cell;
newCell->value = value;
newCell->next = list;
list = newCell;
}
list
value
The Turtles
newCell
The Turtles
???

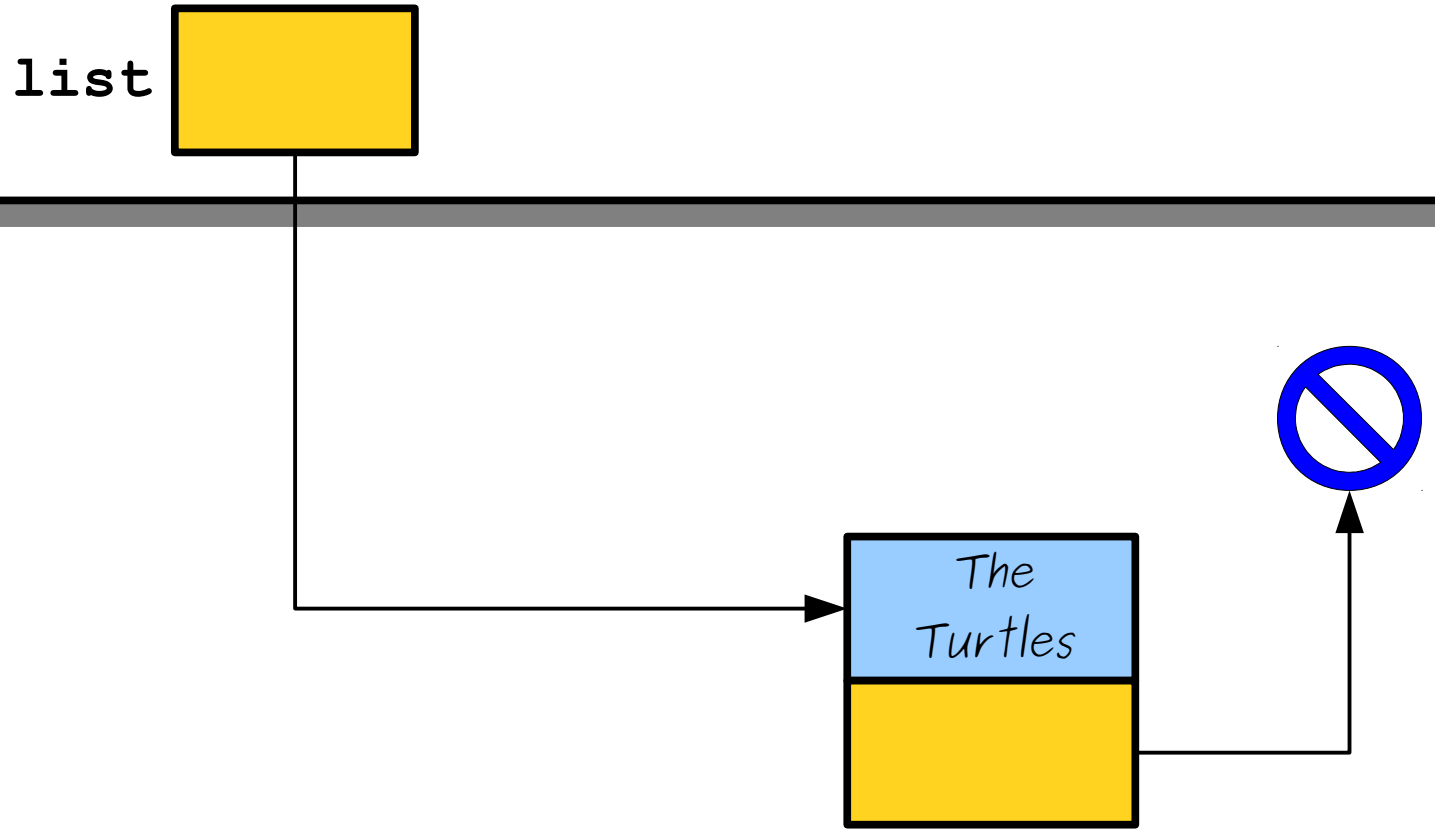

int main() {
Cell* list = n
listInsert(lis
listInsert(lis
listInsert(lis
return 0;
}
list
void listInsert(Cell*& list, const string& value) {
Cell* newCell = new Cell;
newCell->value = value;
newCell->next = list;
list = newCell;
}
list
value
The Turtles
newCell
The Turtles
???


int main() {
Cell* list = n
listInsert(lis
listInsert(lis
listInsert(lis
return 0;
}
void listInsert(Cell*& list, const string& value) {
Cell* newCell = new Cell;
newCell->value = value;
newCell->next = list;
list = newCell;
}
list
list
value
The Turtles
newCell
The Turtles

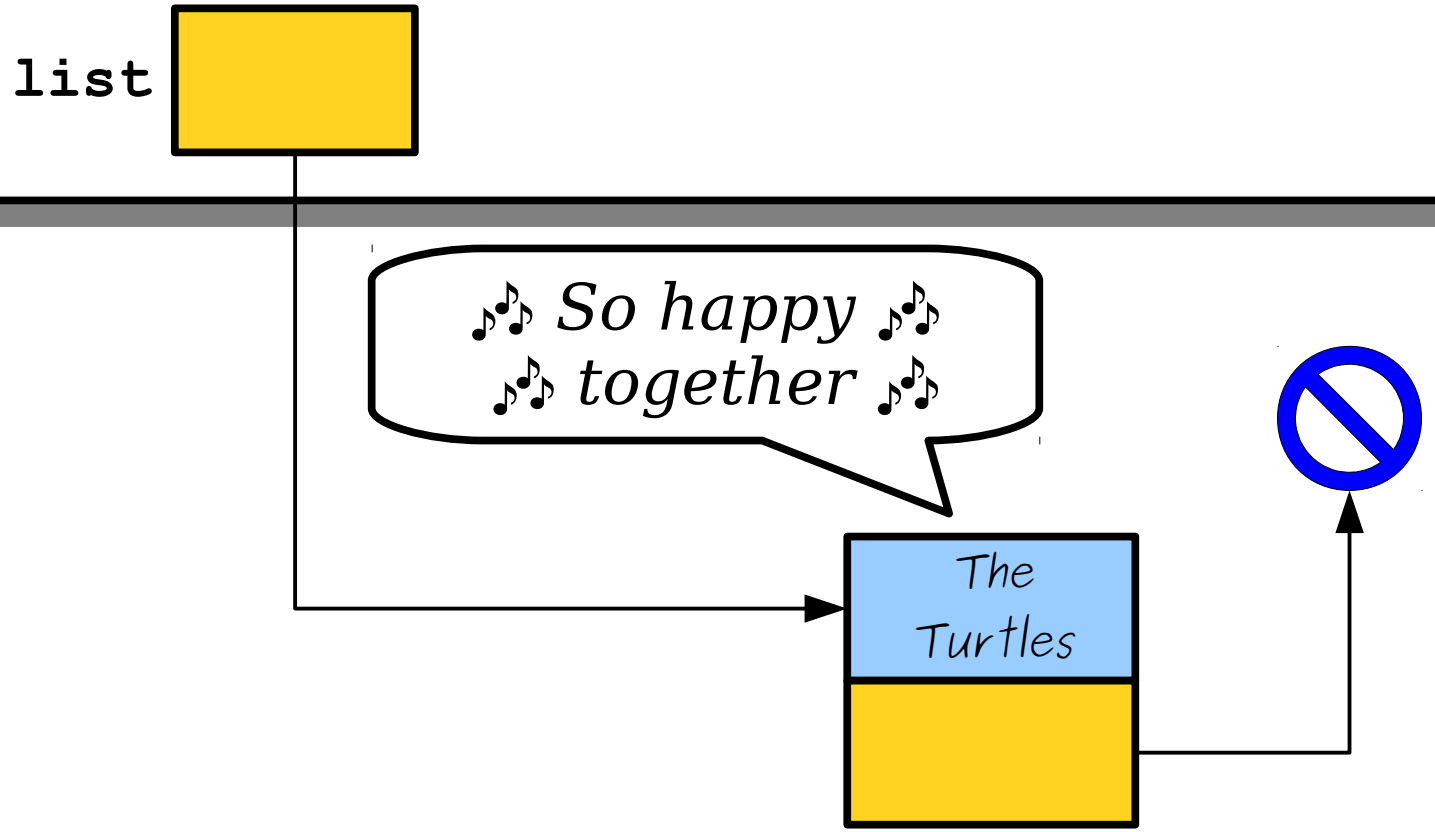

int main() {
Cell* list = n
listInsert(lis
listInsert(lis
listInsert(lis
return 0;
}
void listInsert(Cell*& list, const string& value) {
Cell* newCell = new Cell;
newCell->value = value;
newCell->next = list;
list = newCell;
}
list
list
value
The Turtles
newCell
The Turtles


int main() {
Cell* list = n
listInsert(lis
listInsert(lis
listInsert(lis
return 0;
}
void listInsert(Cell*& list, const string& value) {
Cell* newCell = new Cell;
newCell->value = value;
newCell->next = list;
list = newCell;
}
list
list
value
The Turtles
newCell
The Turtles

```
int main() {
    Cell* list = nullptr;
    listInsert(list, "The Turtles");
    listInsert(list, "The Beatles");
    listInsert(list, "A Flock of Seagulls");

    return 0;
}
```



```
int main() {
    Cell* list = nullptr;
    listInsert(list, "The Turtles");
    listInsert(list, "The Beatles");
    listInsert(list, "A Flock of Seagulls");

    return 0;
}
```



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.

Time-Out for Announcements!

Midterm Timetable

- You’re done with the midterm exam! Woohoo!
- We’ll be grading it over the weekend and returning graded exams on Monday along with stats and solutions.
- Have any questions in the meantime? Just ask!

Assignment 5

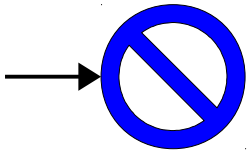
- Assignment 5 (***Data Sagas***) is due one week from today.
- We assume most of you have not yet started, and that's fine. Start working through that assignment this evening and make slow and steady progress.
- Have questions? Stop by the LaIR or CLaIR!

```
lecture = announcements->next;
```

Implementing the `Queue`

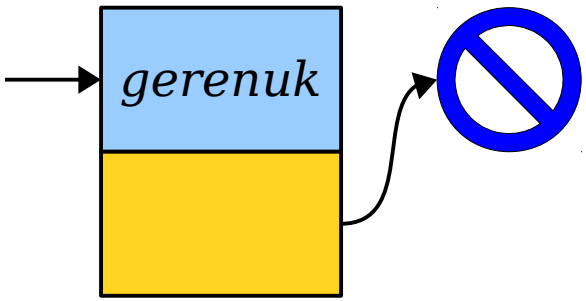
Implementing the `Queue`

- There are many ways to implement the `Queue`, and a common one is to use linked lists.
  - New elements get added to the back of the list.
  - Dequeued elements are taken off the front of the list.
- ***Question:*** How efficient is this?



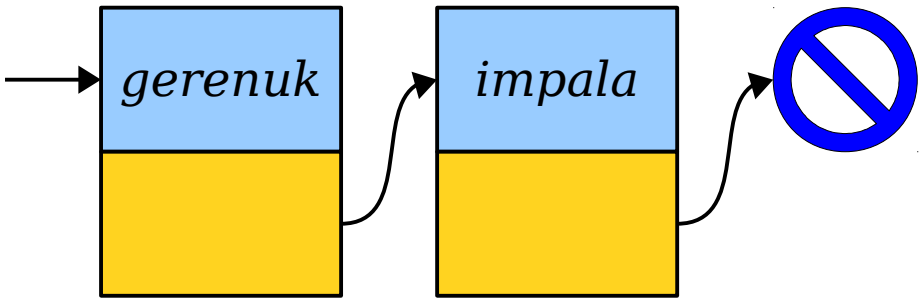
Implementing the Queue

- There are many ways to implement the Queue, and a common one is to use linked lists.
  - New elements get added to the back of the list.
  - Dequeued elements are taken off the front of the list.
- ***Question:*** How efficient is this?



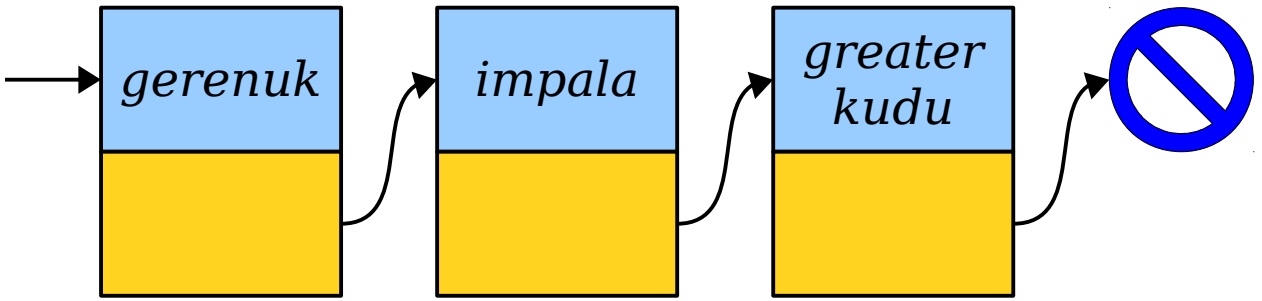
Implementing the Queue

- There are many ways to implement the Queue, and a common one is to use linked lists.
  - New elements get added to the back of the list.
  - Dequeued elements are taken off the front of the list.
- ***Question:*** How efficient is this?



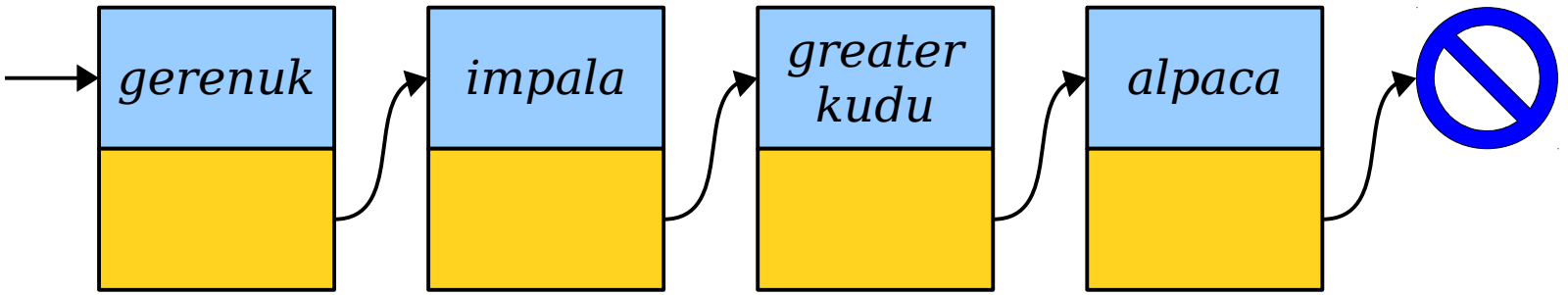
Implementing the Queue

- There are many ways to implement the Queue, and a common one is to use linked lists.
  - New elements get added to the back of the list.
  - Dequeued elements are taken off the front of the list.
- ***Question:*** How efficient is this?



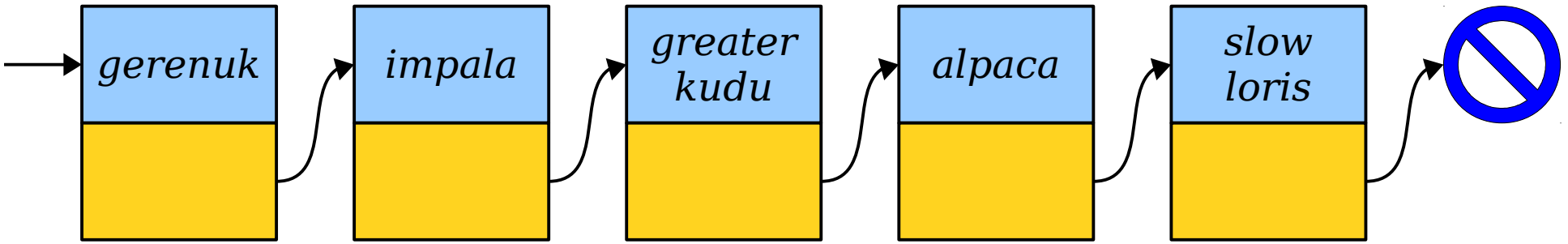
Implementing the Queue

- There are many ways to implement the Queue, and a common one is to use linked lists.
  - New elements get added to the back of the list.
  - Dequeued elements are taken off the front of the list.
- ***Question:*** How efficient is this?



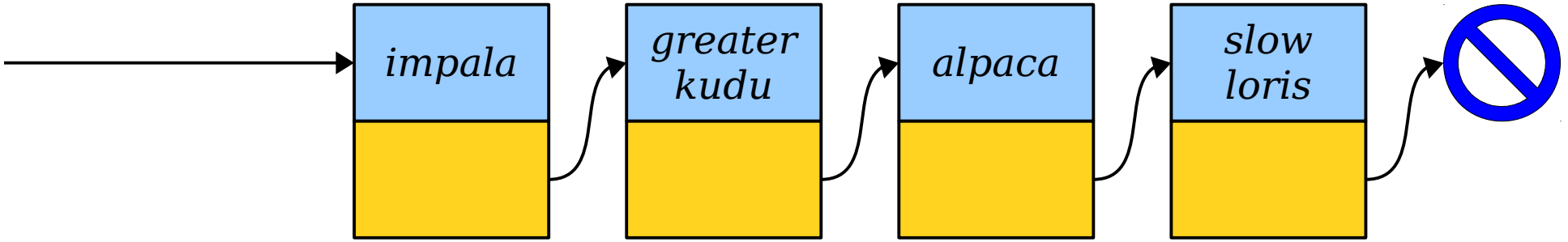
Implementing the Queue

- There are many ways to implement the Queue, and a common one is to use linked lists.
  - New elements get added to the back of the list.
  - Dequeued elements are taken off the front of the list.
- ***Question:*** How efficient is this?



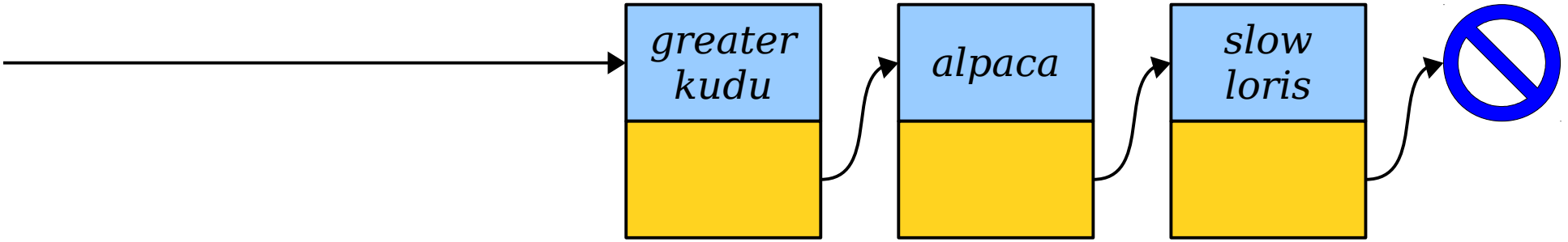
Implementing the Queue

- There are many ways to implement the Queue, and a common one is to use linked lists.
  - New elements get added to the back of the list.
  - Dequeued elements are taken off the front of the list.
- ***Question:*** How efficient is this?



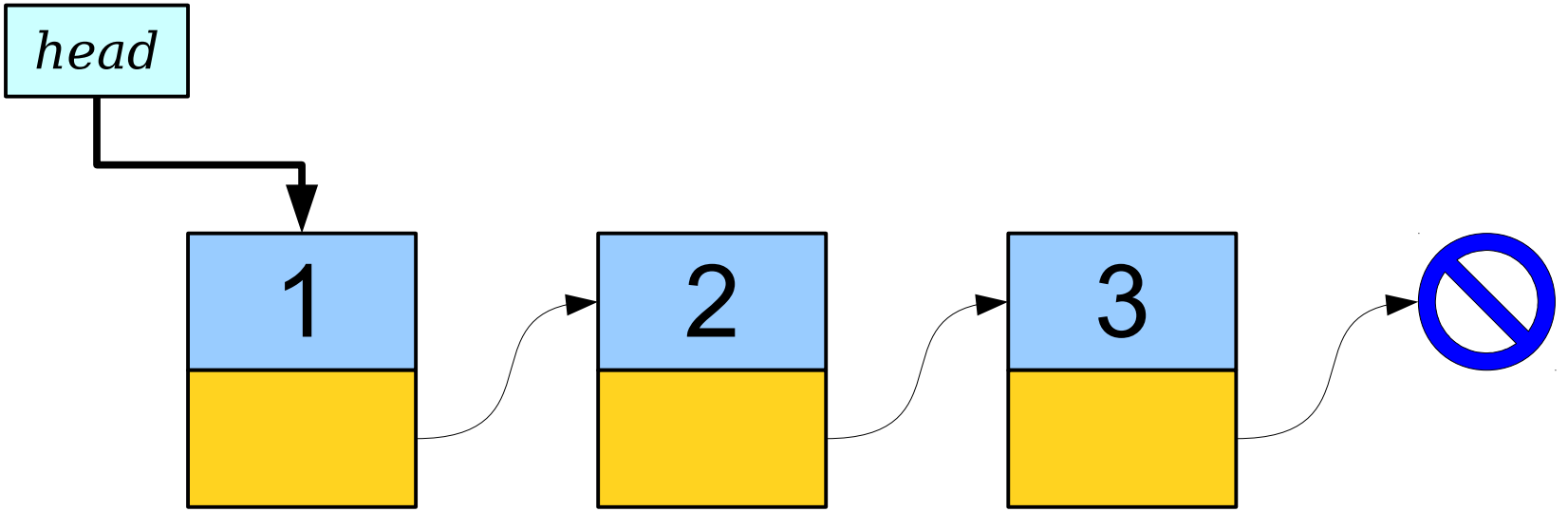
Implementing the Queue

- There are many ways to implement the Queue, and a common one is to use linked lists.
  - New elements get added to the back of the list.
  - Dequeued elements are taken off the front of the list.
- ***Question:*** How efficient is this?



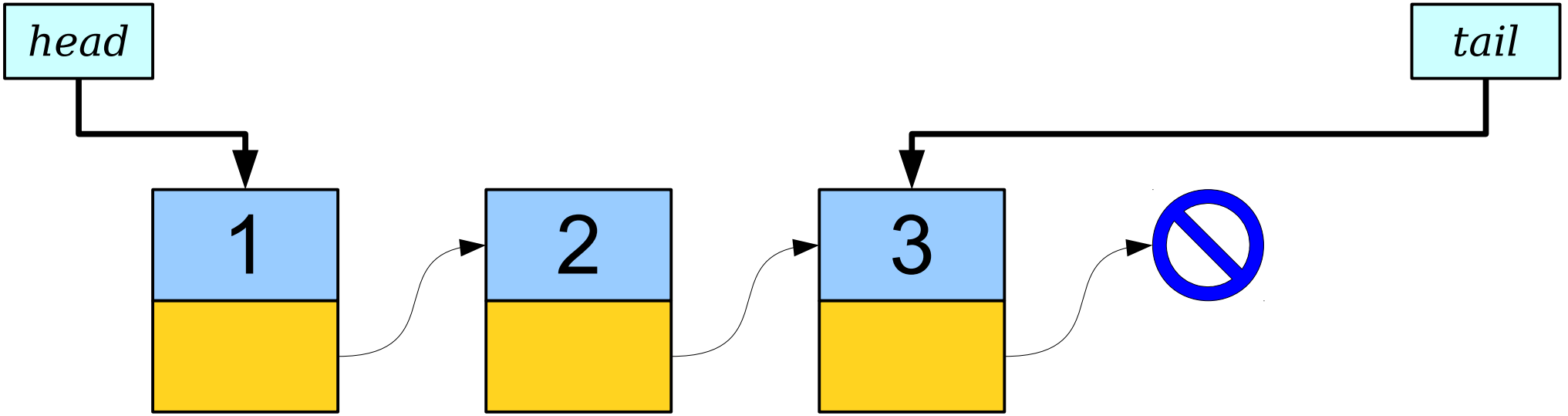
Tail Pointers

- 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.
- We can use tail pointers to implement an efficient `Queue` using a linked list.



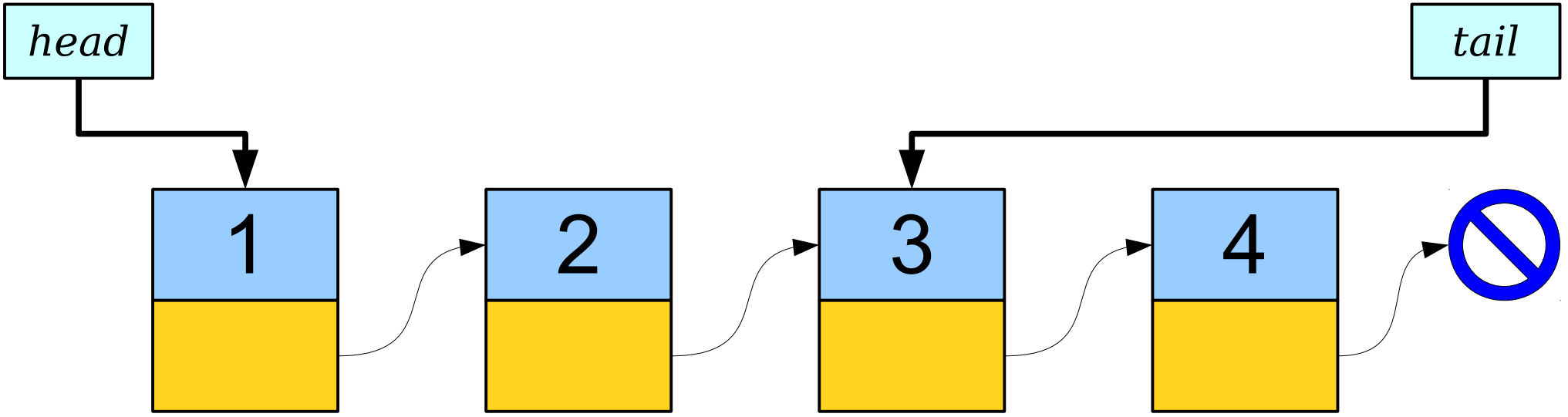
Tail Pointers

- 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.
- We can use tail pointers to implement an efficient `Queue` using a linked list.



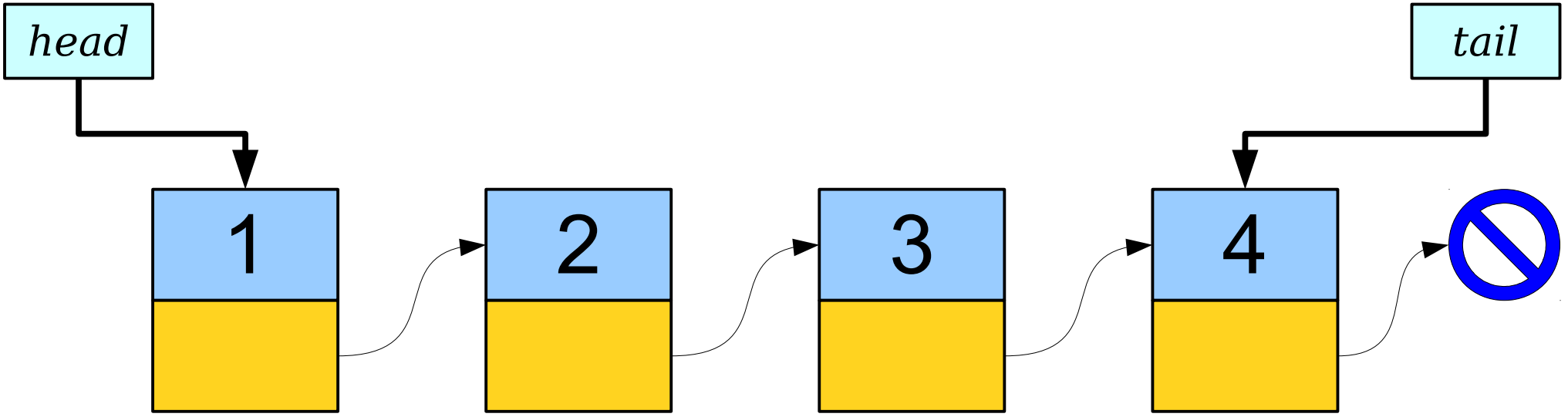
Tail Pointers

- 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.
- We can use tail pointers to implement an efficient `Queue` using a linked list.



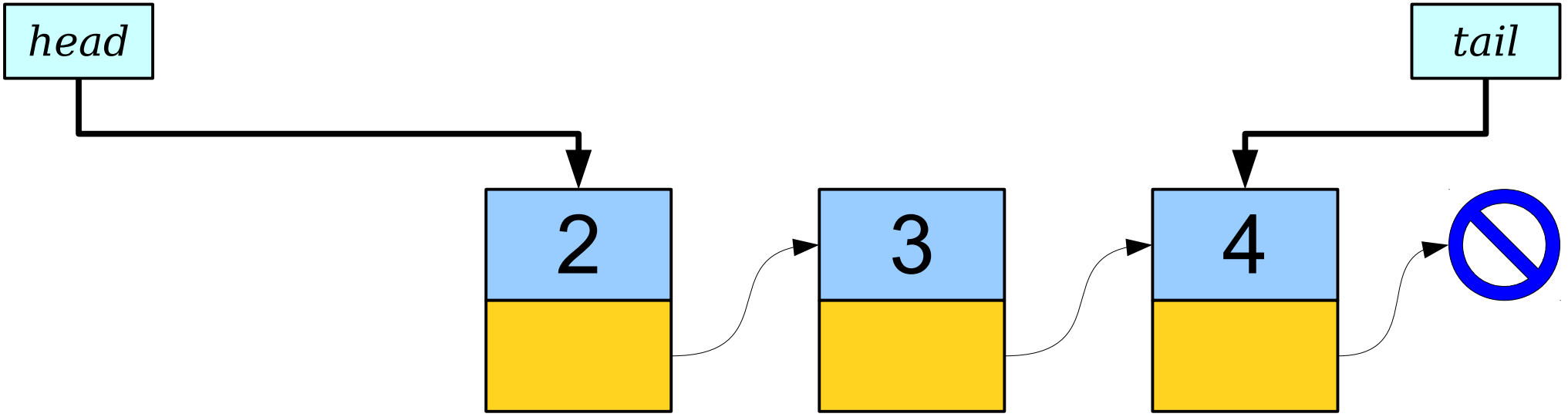
Tail Pointers

- 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.
- We can use tail pointers to implement an efficient `Queue` using a linked list.



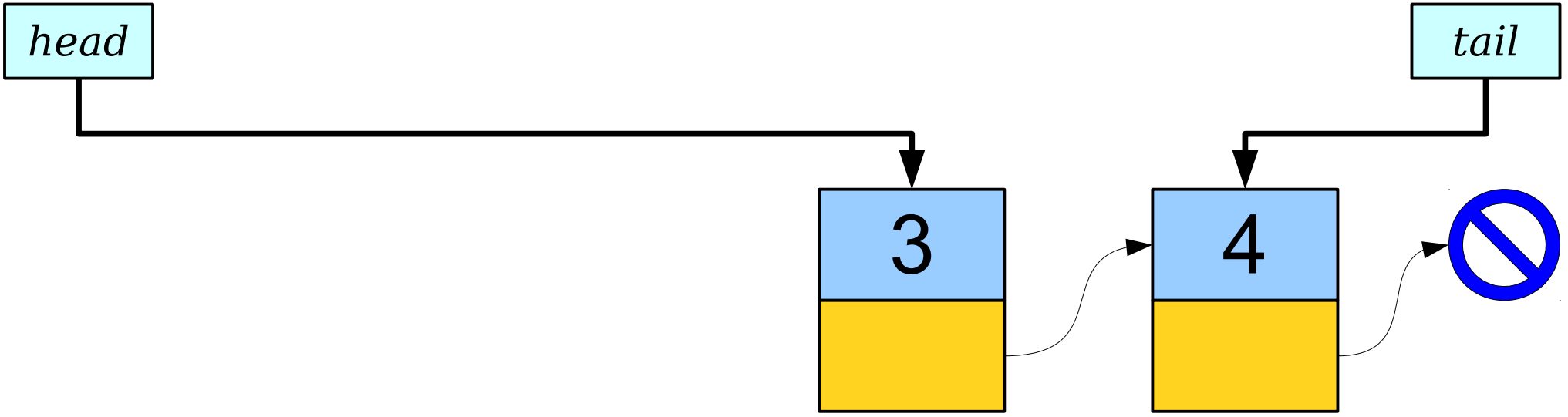
Tail Pointers

- 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.
- We can use tail pointers to implement an efficient `Queue` using a linked list.



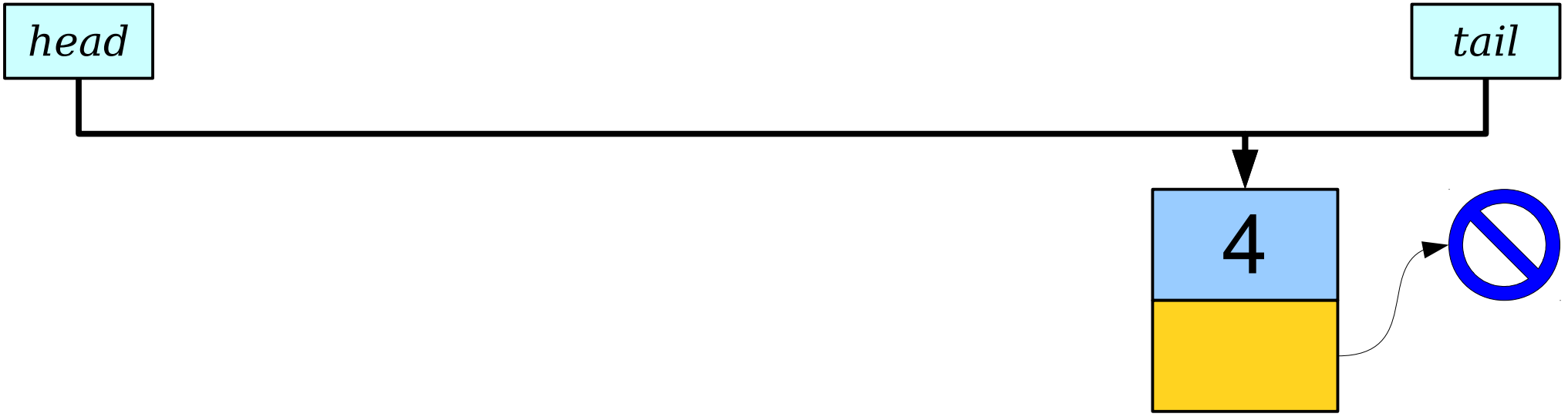
Tail Pointers

- 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.
- We can use tail pointers to implement an efficient `Queue` using a linked list.



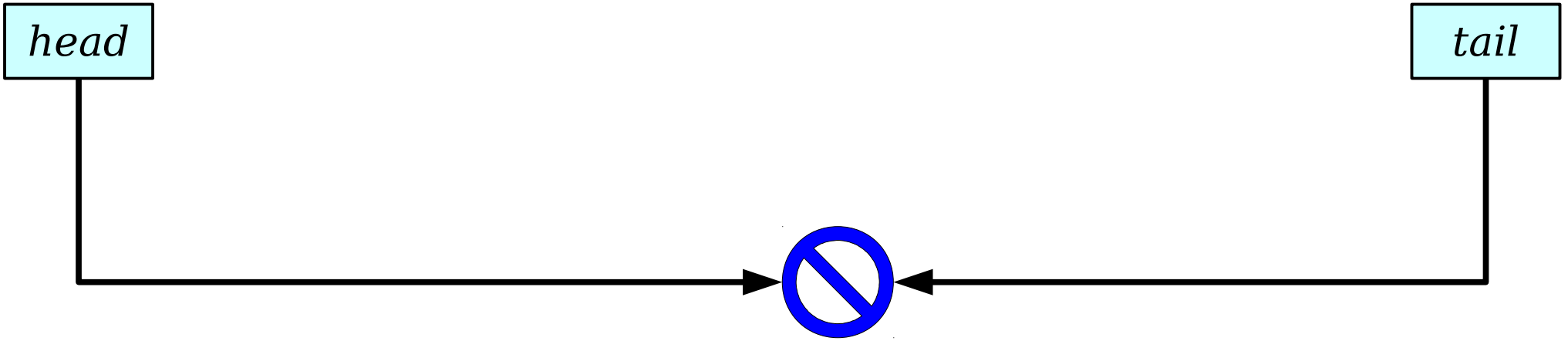
Tail Pointers

- 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.
- We can use tail pointers to implement an efficient `Queue` using 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.
- We can use tail pointers to implement an efficient `Queue` using a linked list.



```
class OurQueue {
public:
    OurQueue();
    ~OurQueue();

    int peek() const;
    void enqueue(int value);
    int dequeue();

    int size() const;
    bool isEmpty() const;

private:
    struct Cell {
        int value;
        Cell* next;
    };

    Cell* head;
    Cell* tail;
};
```

Enqueuing Things

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

Enqueuing Things

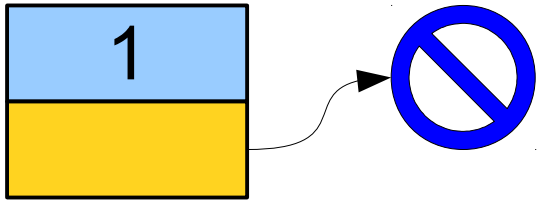
- ***Case 1:*** The queue is empty.

Enqueuing Things

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

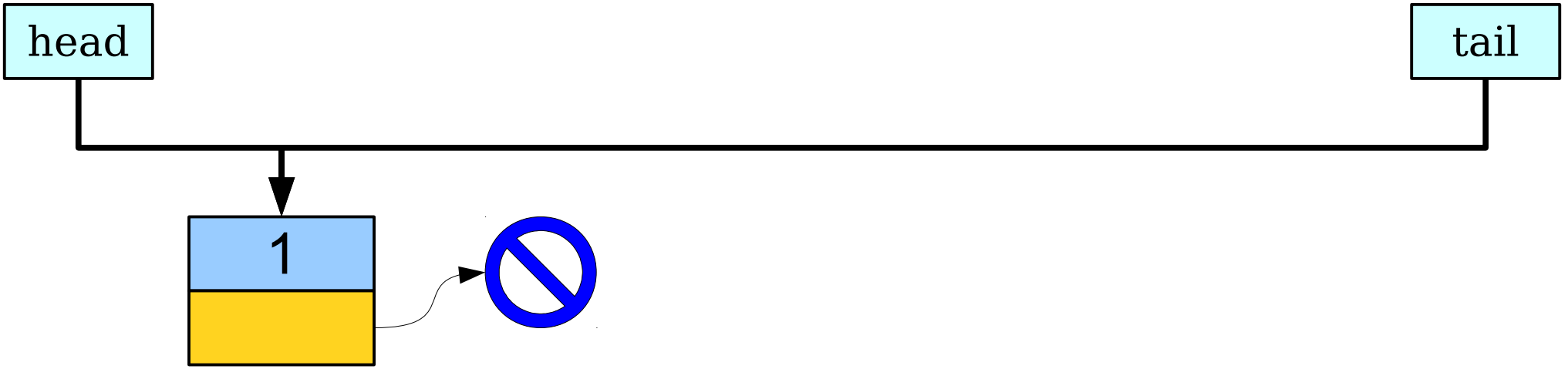
head

tail



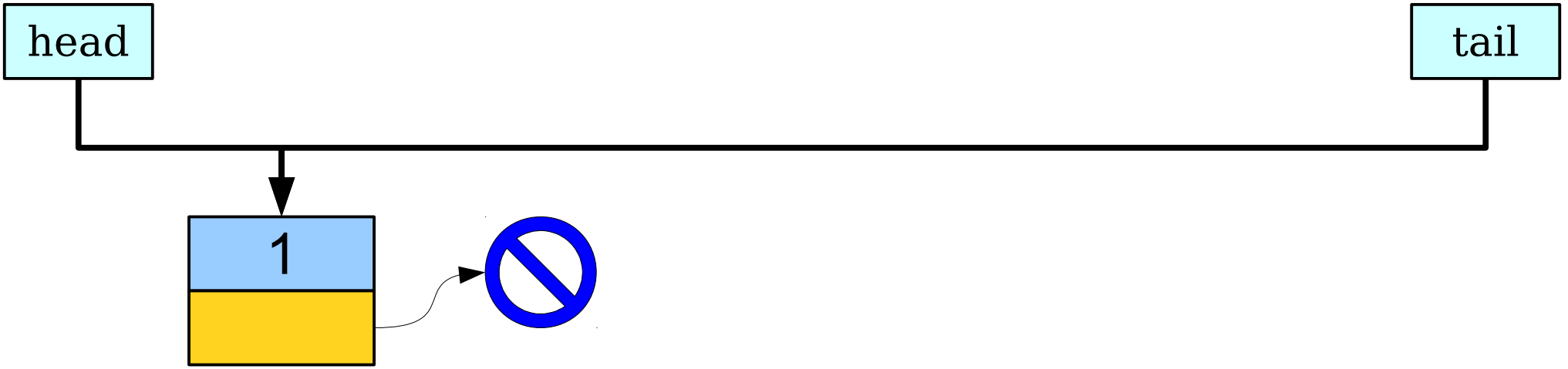
Enqueuing Things

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

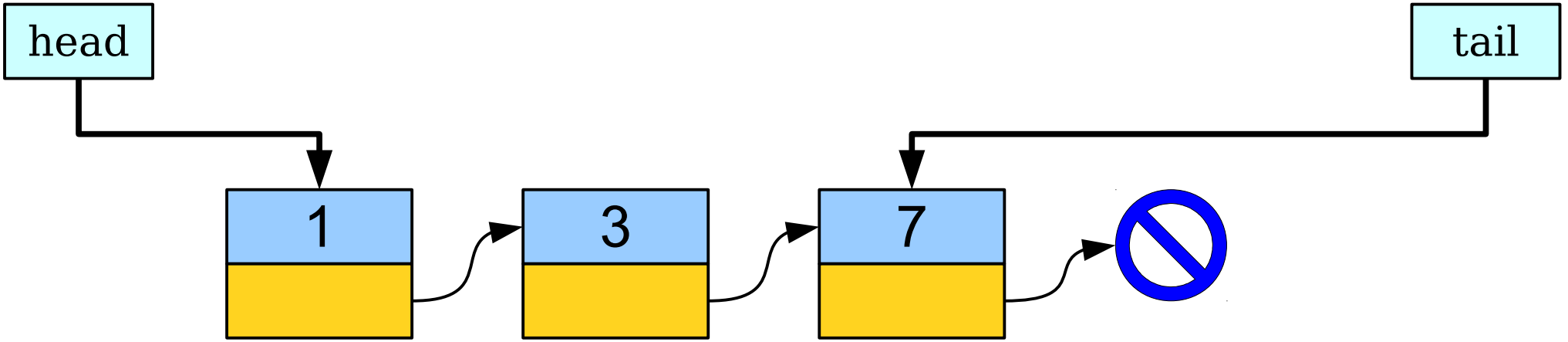


Enqueuing Things

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

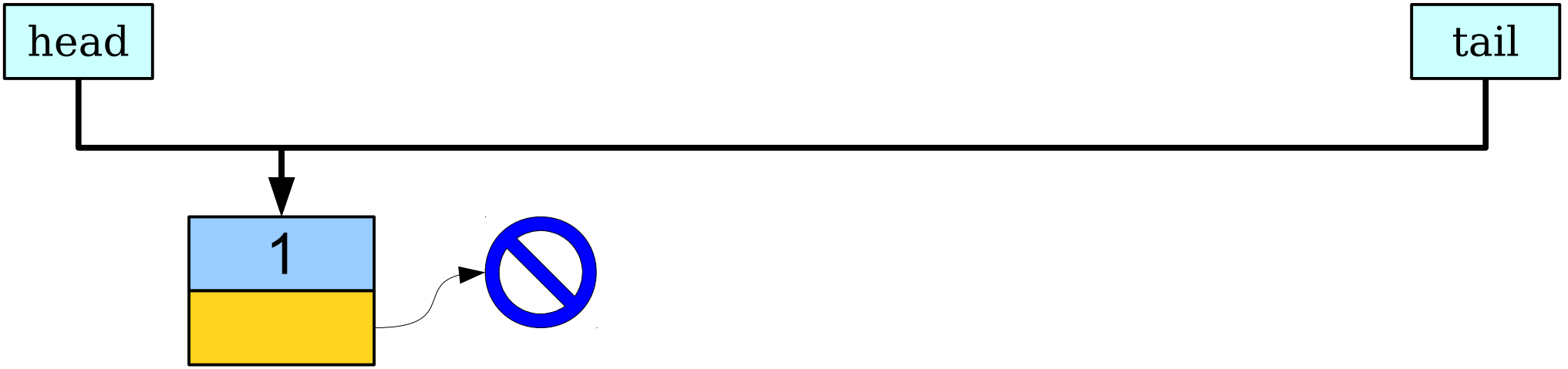


- ***Case 2:*** The queue is not empty.

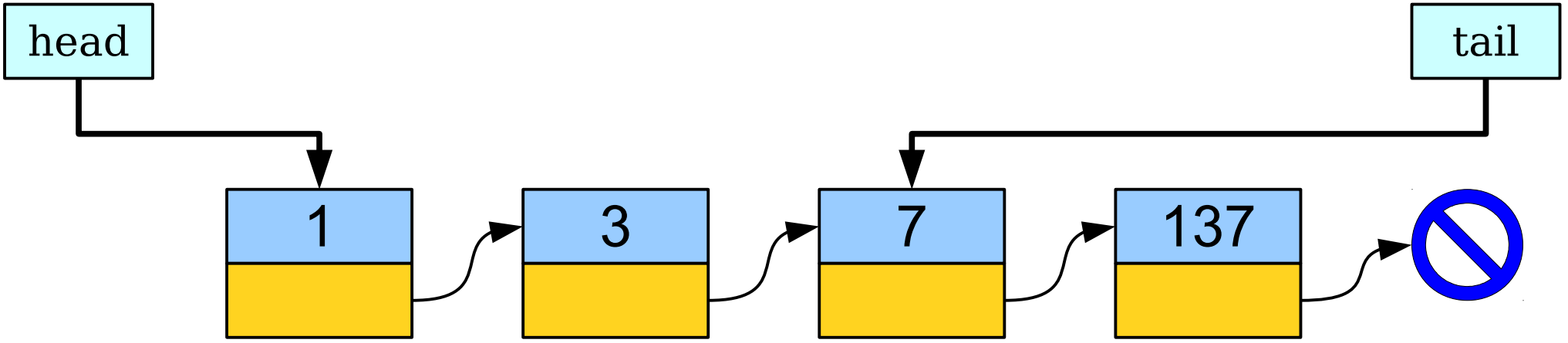


Enqueuing Things

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

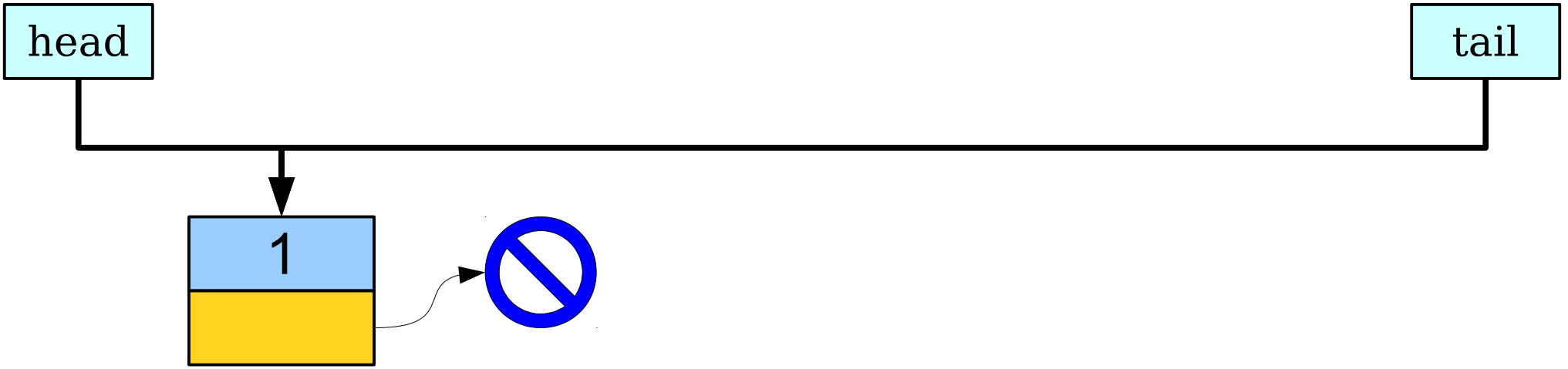


- ***Case 2:*** The queue is not empty.

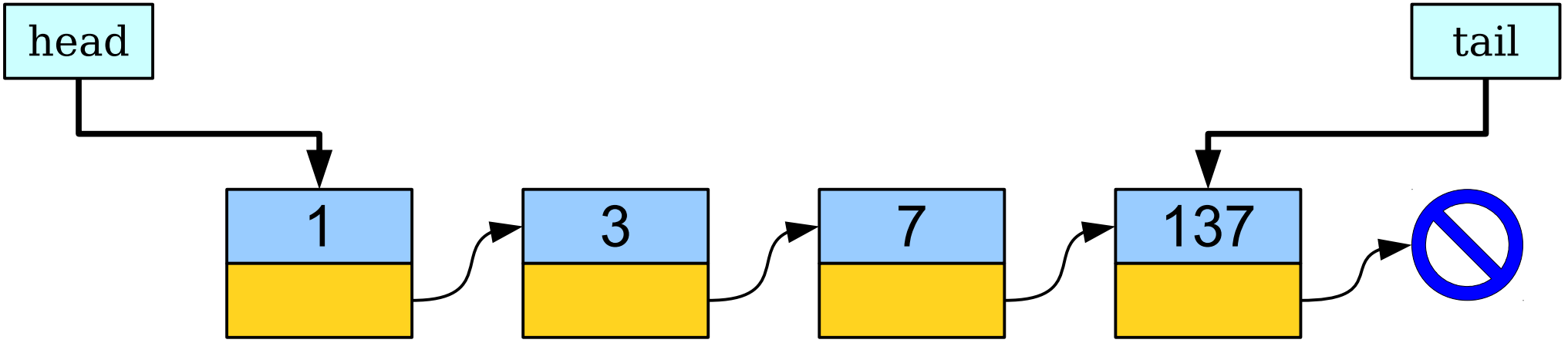


Enqueuing Things

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



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

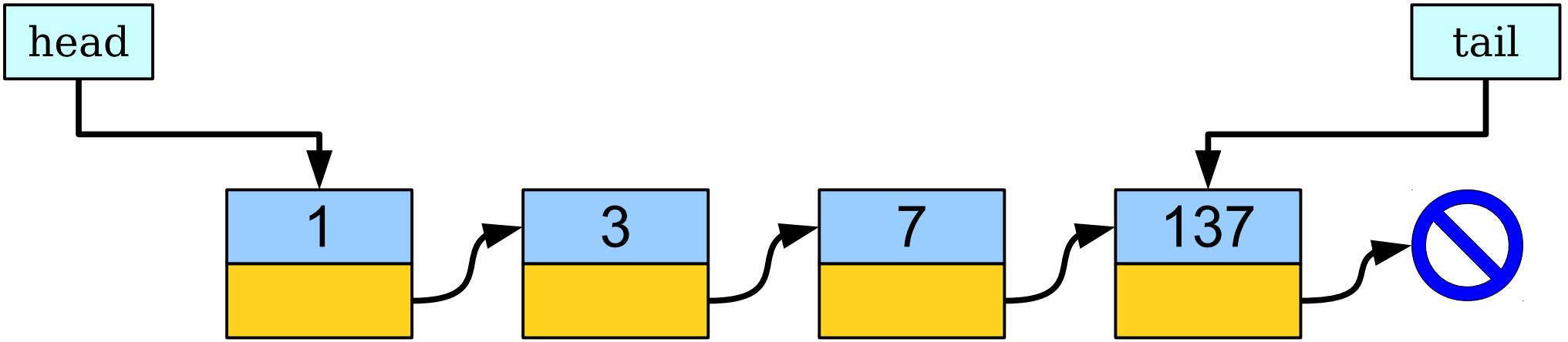


Dequeuing Things

- ***Case 1:*** Dequeuing when there are 2+ elements.

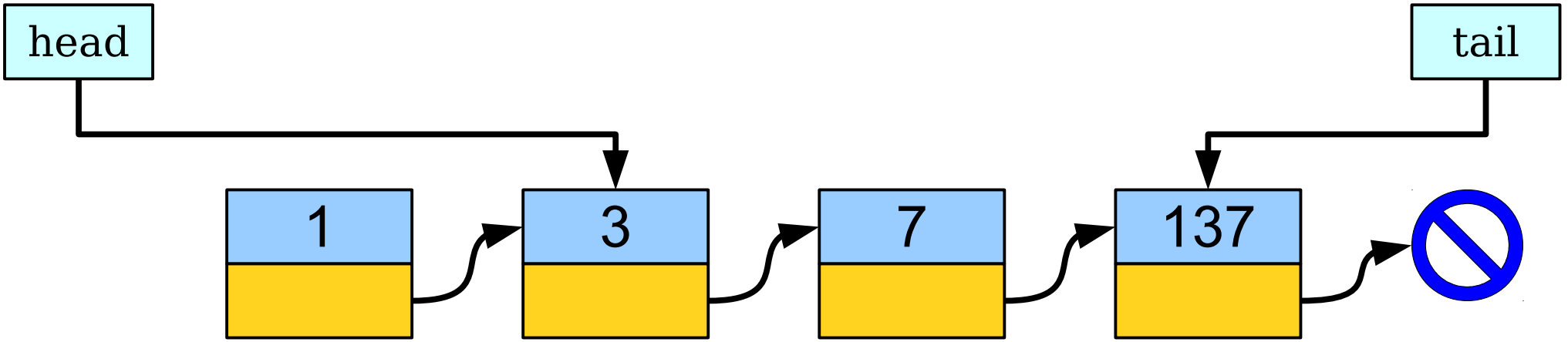
Dequeuing Things

- ***Case 1:*** Dequeuing when there are 2+ elements.



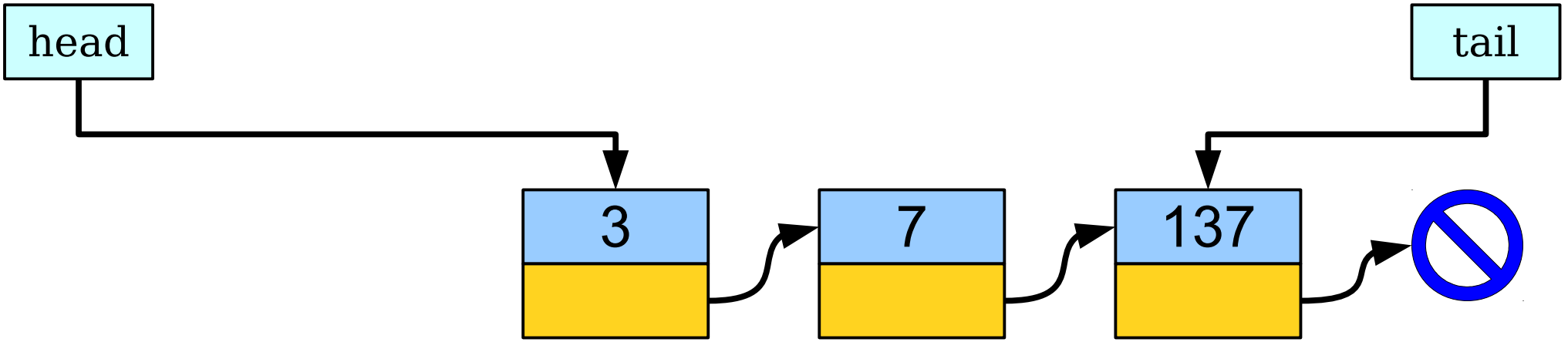
Dequeuing Things

- ***Case 1:*** Dequeuing when there are 2+ elements.



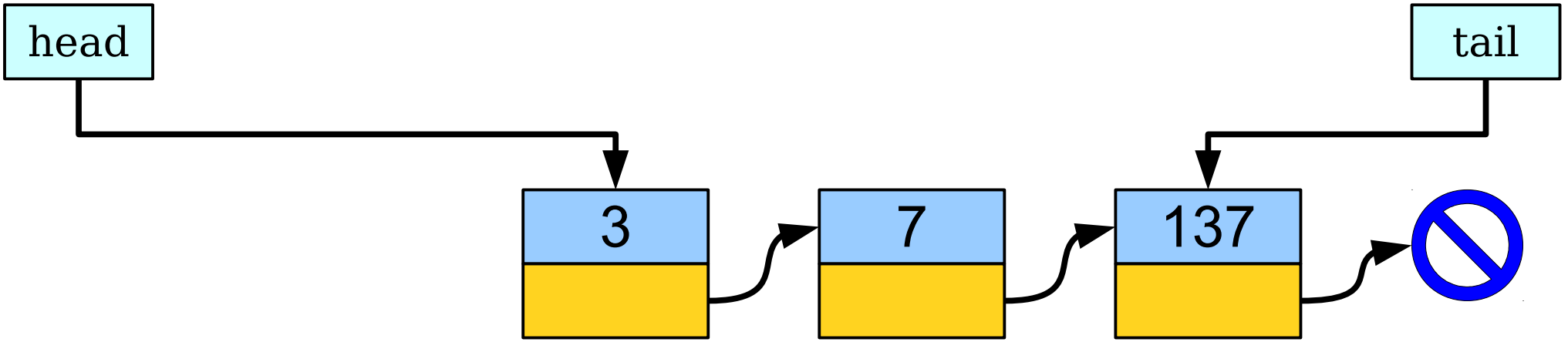
Dequeuing Things

- ***Case 1:*** Dequeuing when there are 2+ elements.

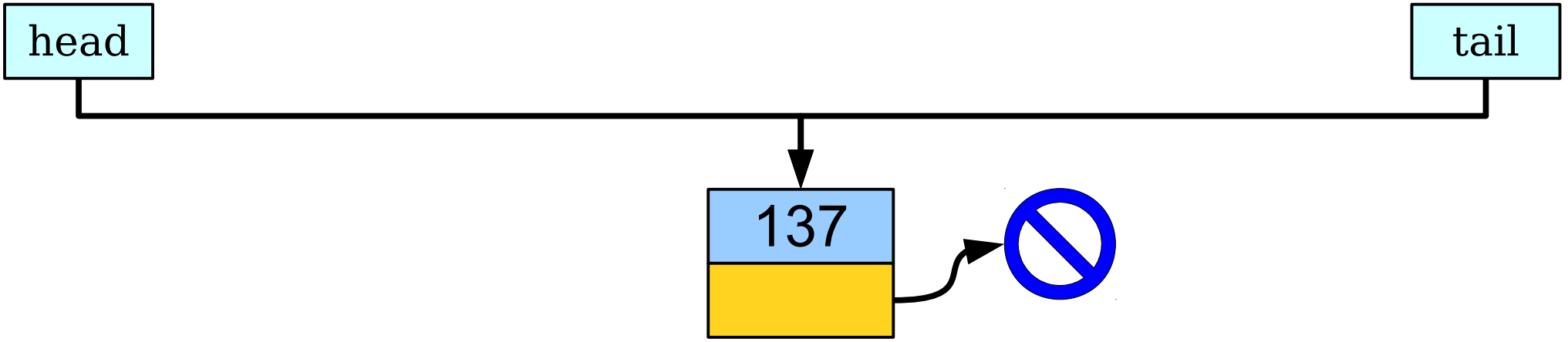


Dequeuing Things

- ***Case 1:*** Dequeuing when there are 2+ elements.

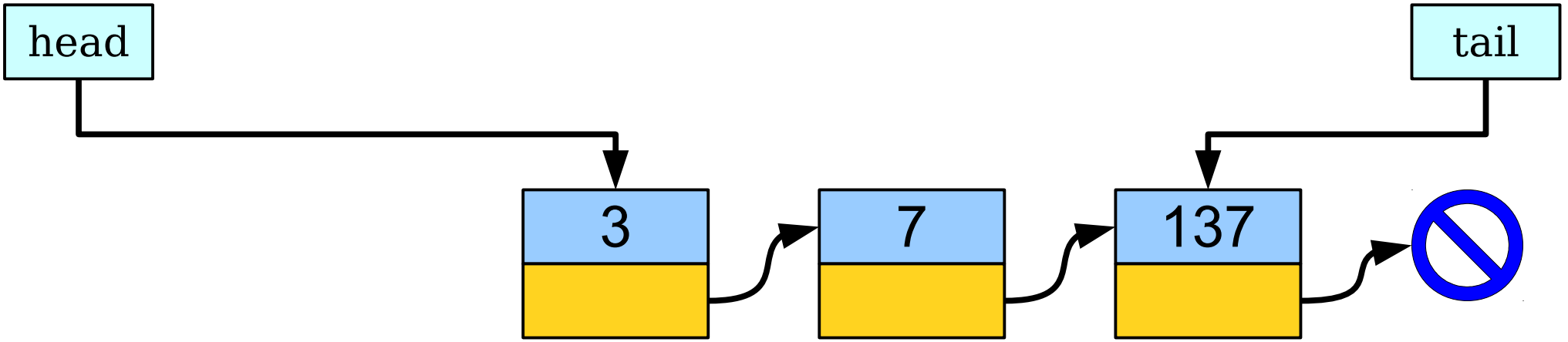


Case 2: Dequeuing the last element.

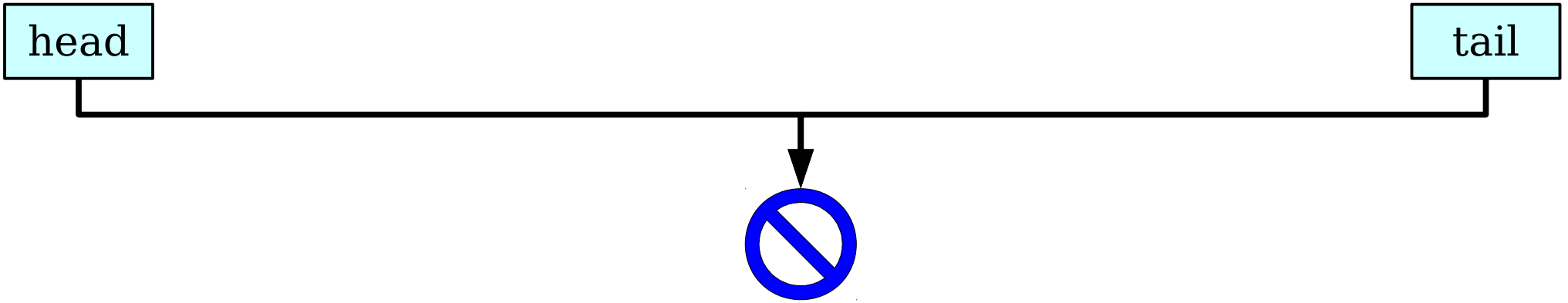


Dequeuing Things

- ***Case 1:*** Dequeuing when there are 2+ elements.



Case 2: Dequeuing the last element.



The Overall Analysis

- Implementing a queue using a linked list without a tail pointer:
  - Cost of an `enqueue`: **O(*n*)**
  - Cost of a `dequeue`: **O(1)**
- Implementing a queue using a linked list with a tail pointer:
  - Cost of an `enqueue`: **O(1)**
  - Cost of a `dequeue`: **O(1)**
- This is really, really fast!

Your Action Items

- ***Read Chapter 12 of the textbook (and, optionally, Chapter 13).***
  - It'll provide more information about linked lists, data structure implementation, and runtime efficiency.
- ***Work on Assignment 5.***
  - At a bare minimum, read through the handout and make sure you know what's asked of you.
  - Recommendation: Complete multiway merge and start lower bound searching by next time.

Next Time

- ***Tree Structures***
  - Encoding trees directly in software!
- ***Binary Search Trees***
  - A fast, flexible, powerful data structure.