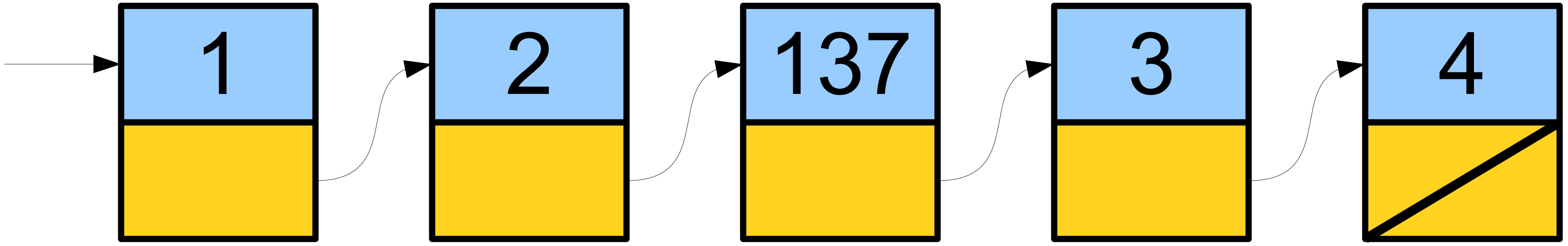
Linked Lists

Part Two

Recap from Last Time

Linked Lists at a Glance

- 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.



Representing a Cell

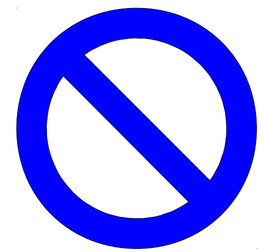
- For simplicity, let's assume we're building a linked list of `string`s.
- We can represent a cell in the linked list as a structure:

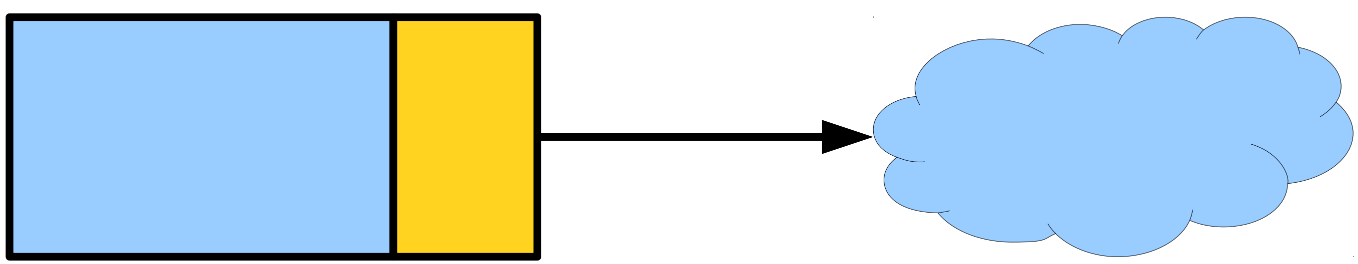
```
struct Cell {
    string value;
    Cell* next;
};
```

- ***The structure is defined recursively!***

A Linked List is Either…

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





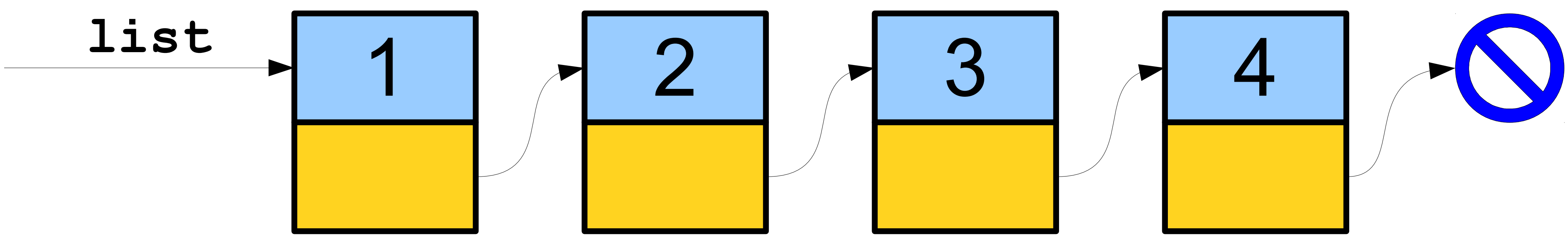
a single linked list cell that points…

… at another linked list.

Traversing a Linked List

- Once we have a linked list, we can traverse it by following the links one at a time.

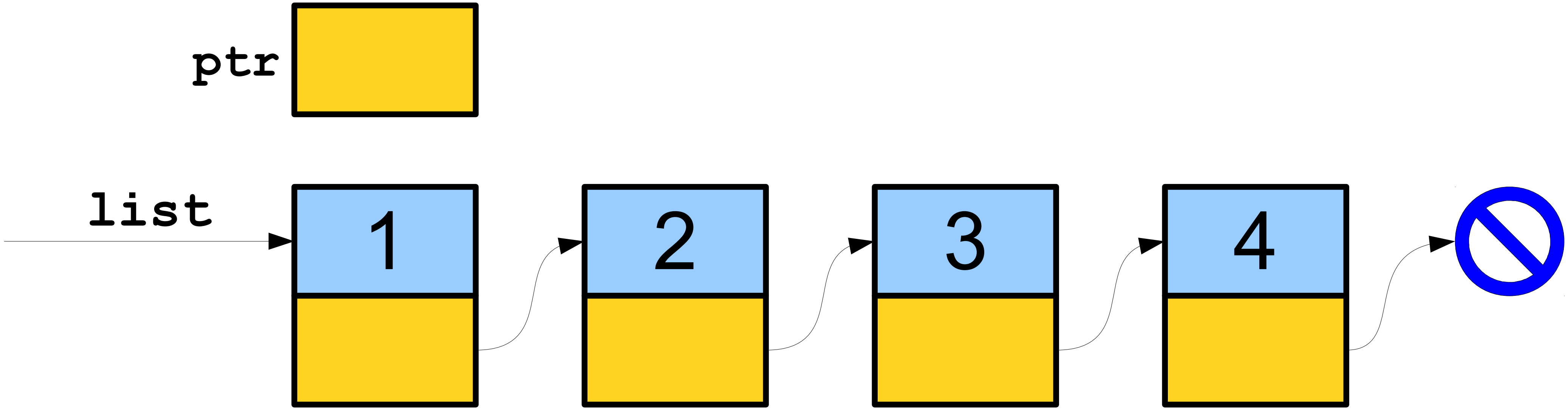
```
for (Cell* ptr = list; ptr != nullptr; ptr = ptr->next) {
    /* … use ptr … */
}
```



Traversing a Linked List

- Once we have a linked list, we can traverse it by following the links one at a time.

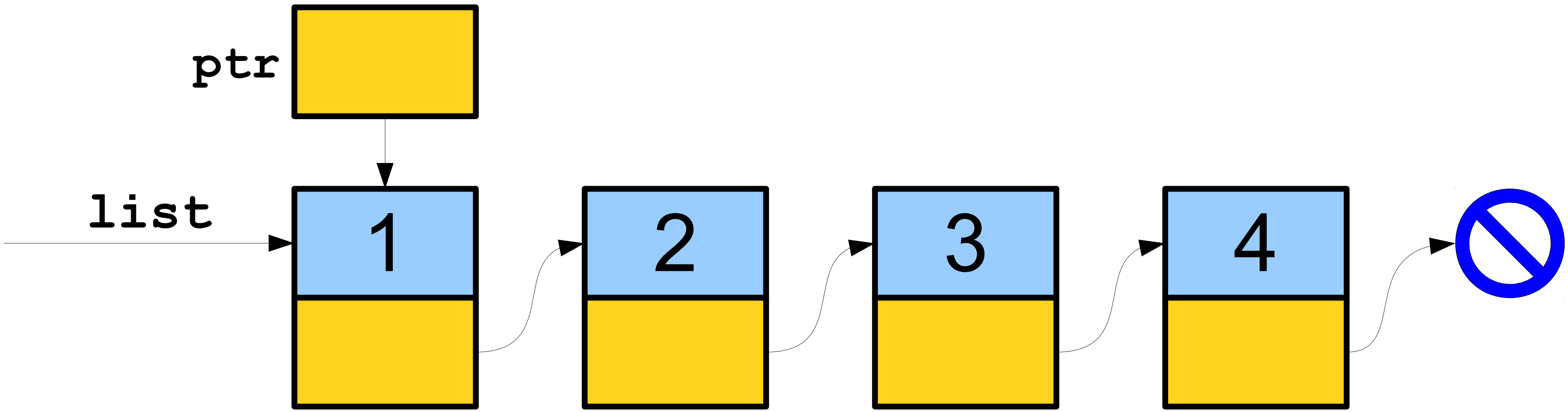
```
for (Cell* ptr = list; ptr != nullptr; ptr = ptr->next) {
    /* … use ptr … */
}
```



Traversing a Linked List

- Once we have a linked list, we can traverse it by following the links one at a time.

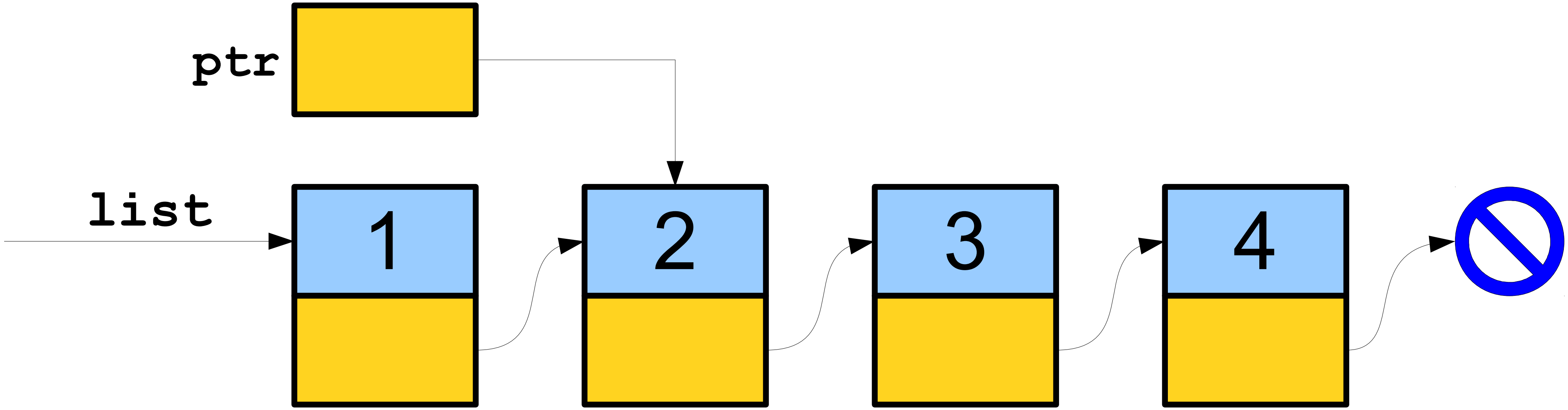
```
for (Cell* ptr = list; ptr != nullptr; ptr = ptr->next) {
    /* … use ptr … */
}
```



Traversing a Linked List

- Once we have a linked list, we can traverse it by following the links one at a time.

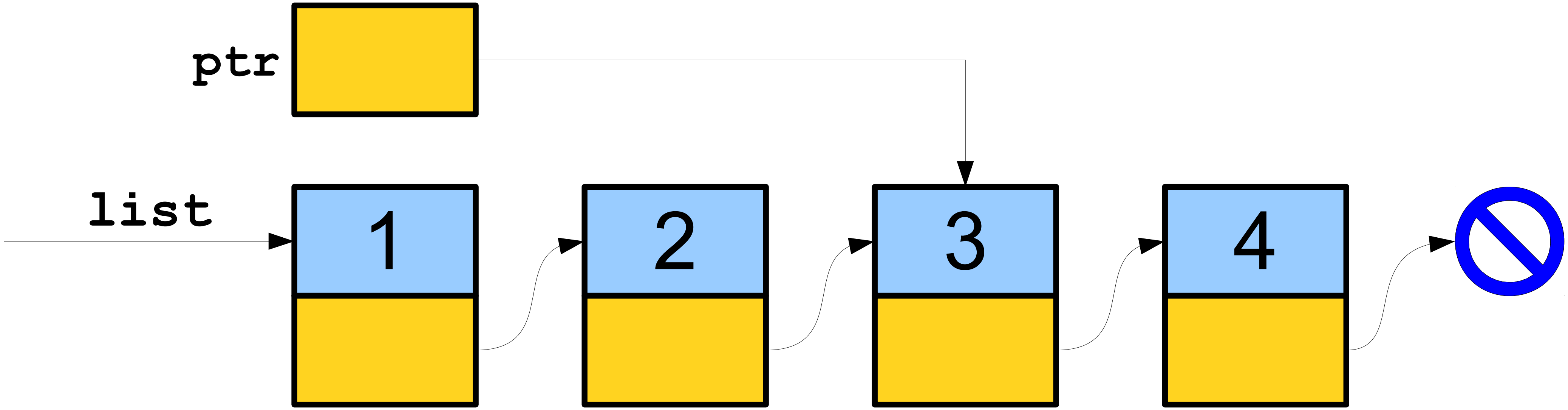
```
for (Cell* ptr = list; ptr != nullptr; ptr = ptr->next) {
    /* … use ptr … */
}
```



Traversing a Linked List

- Once we have a linked list, we can traverse it by following the links one at a time.

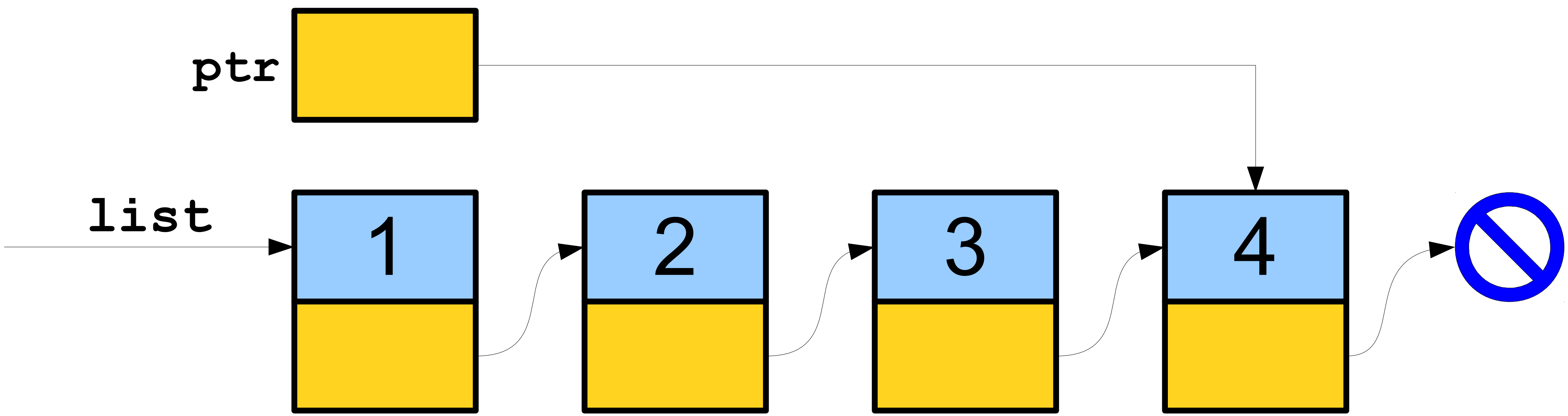
```
for (Cell* ptr = list; ptr != nullptr; ptr = ptr->next) {
    /* … use ptr … */
}
```



Traversing a Linked List

- Once we have a linked list, we can traverse it by following the links one at a time.

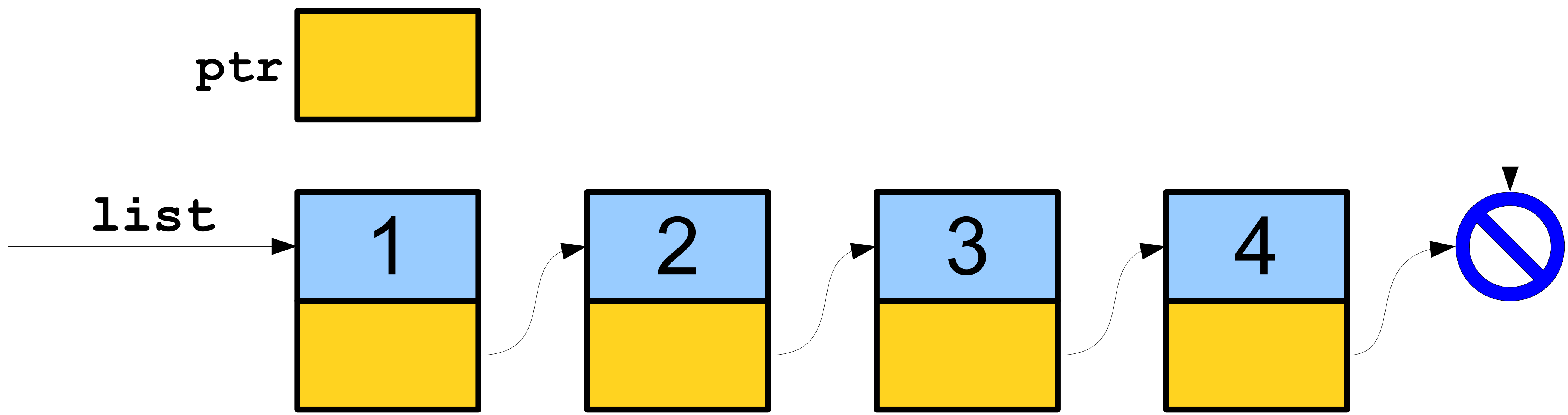
```
for (Cell* ptr = list; ptr != nullptr; ptr = ptr->next) {
    /* … use ptr … */
}
```



Traversing a Linked List

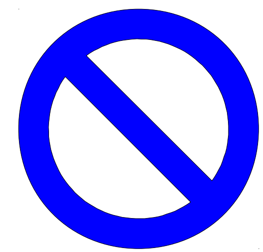
- Once we have a linked list, we can traverse it by following the links one at a time.

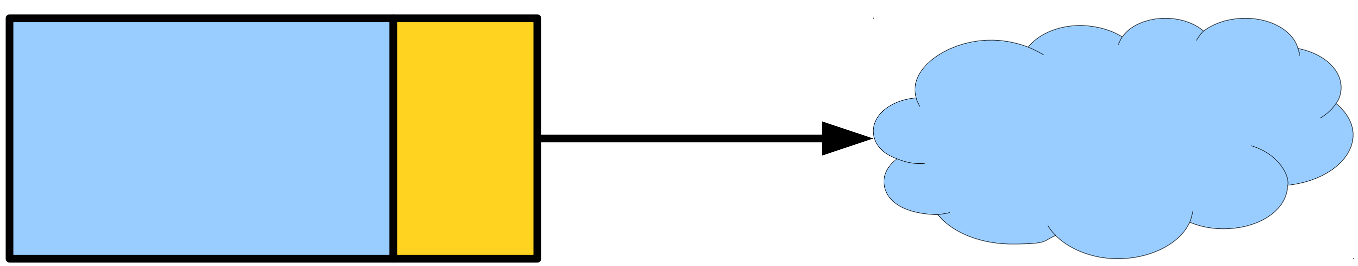
```
for (Cell* ptr = list; ptr != nullptr; ptr = ptr->next) {
    /* … use ptr … */
}
```



A Linked List is Either…

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





a single linked list cell that points…

… at another linked list.

New Stuff!

Cleaning Up Our Messes

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

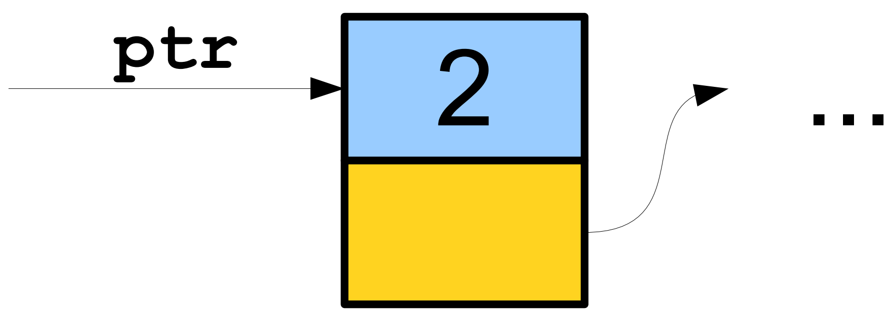
- All good things must come to an end, and we eventually need to reclaim the memory for a linked list.
- The following code triggers *undefined behavior*. ***Don't do this!***

```
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!***

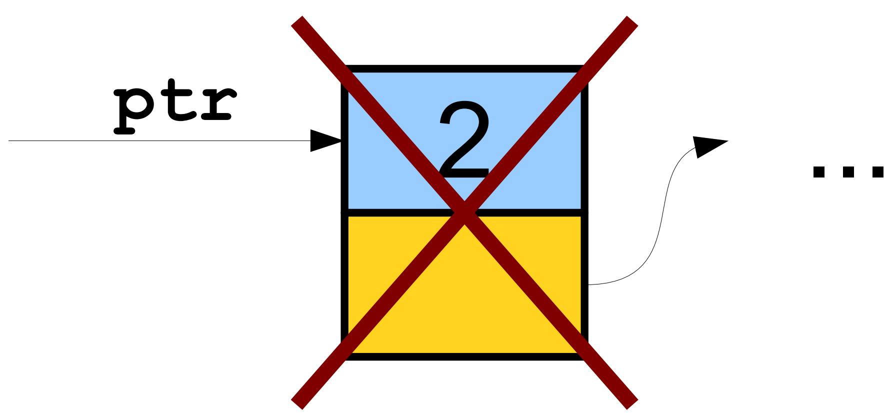
```
for (Cell* ptr = list; ptr != nullptr; ptr = ptr->next) {
    delete ptr;
}
```



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!***

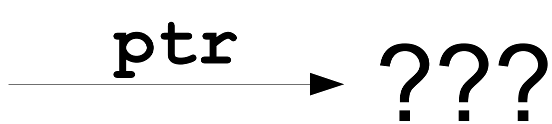
```
for (Cell* ptr = list; ptr != nullptr; ptr = ptr->next) {
    delete ptr;
}
```



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;
}
```

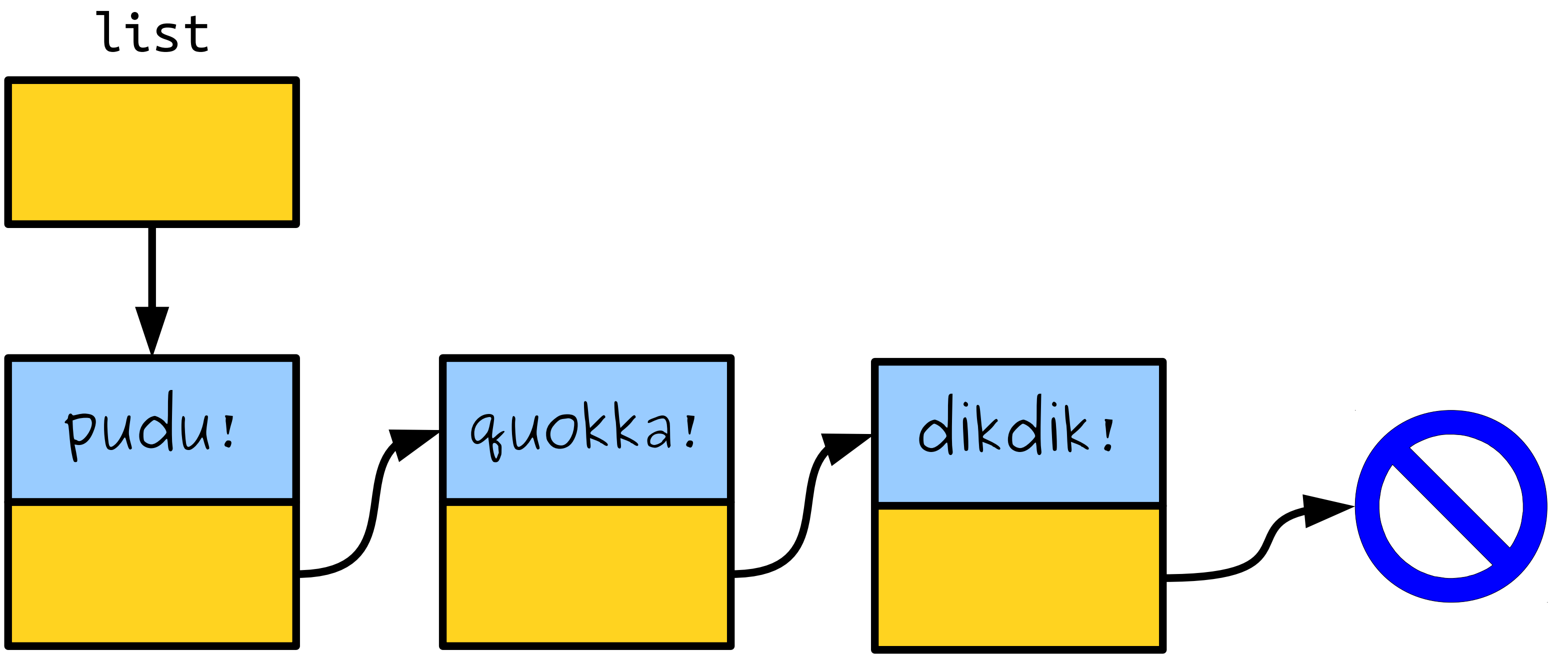


Freeing a Linked List Properly

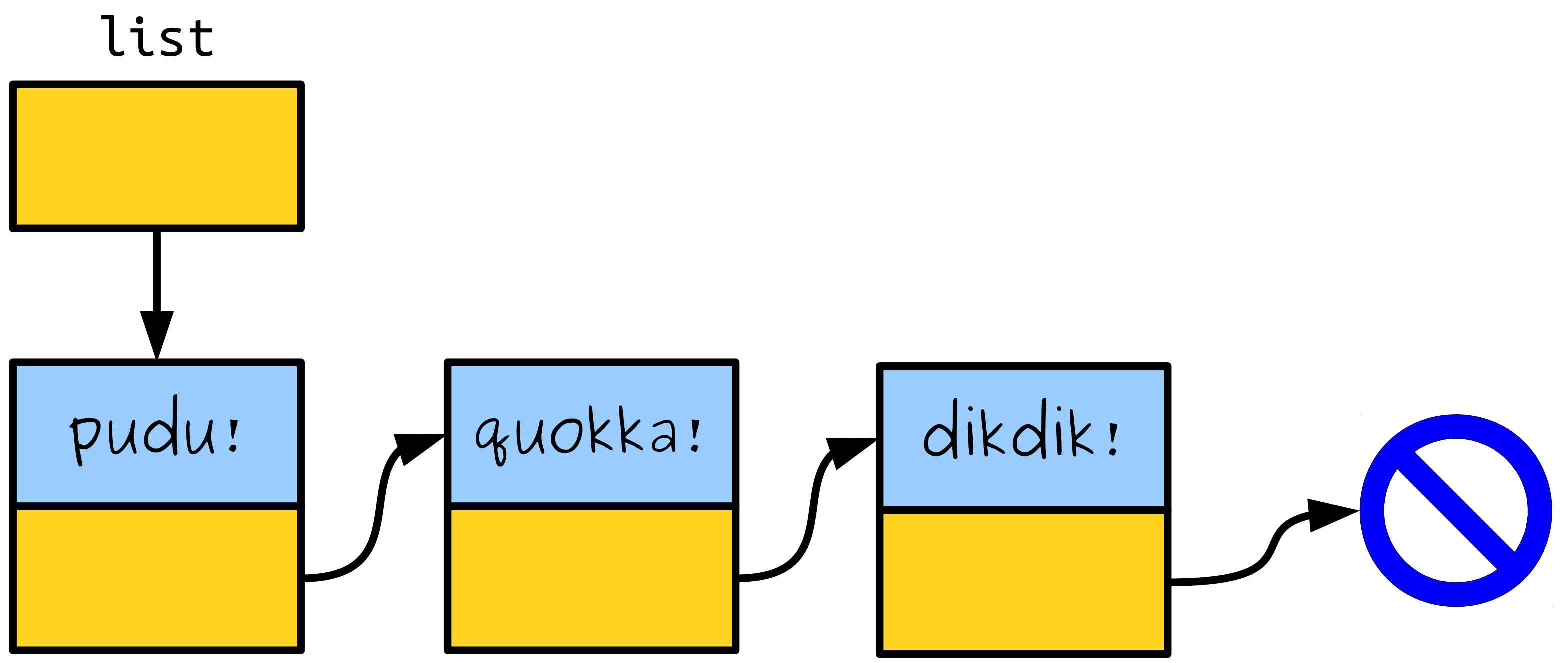
- To properly free a linked list, we have to be able to
  - Destroy a cell, and
  - Advance to the cell after it.
- How might we accomplish this?

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

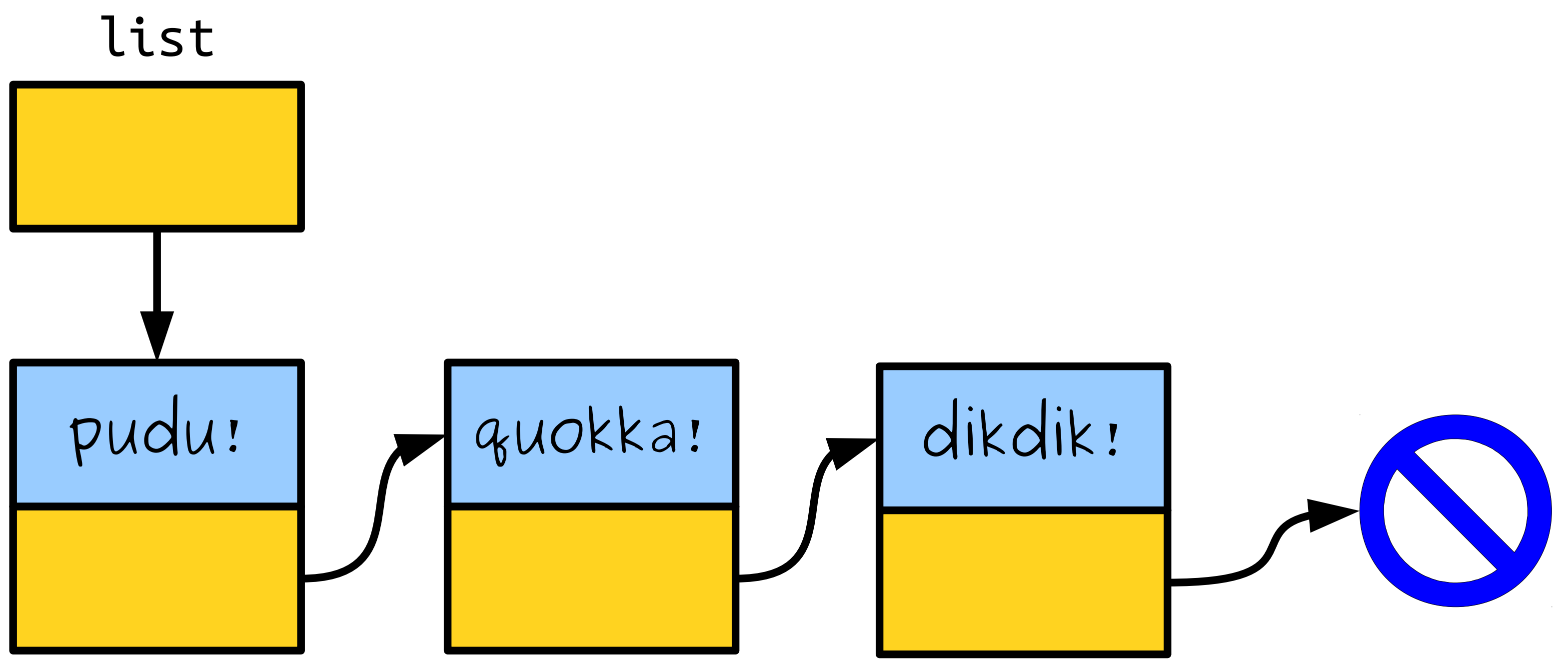
```
while (list != nullptr) {
   Cell* next = list->next;
   delete list;
   list = next;
}
```



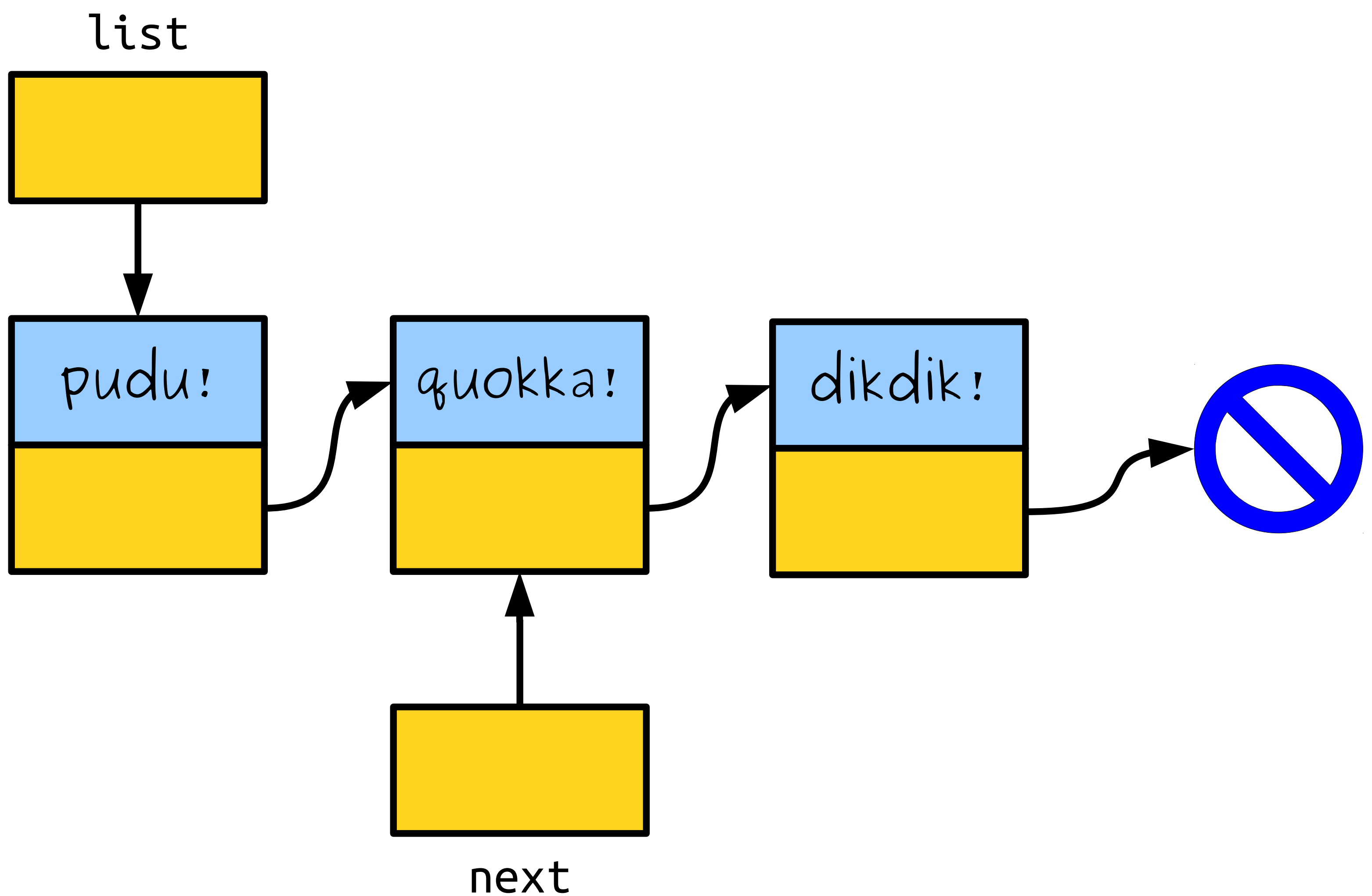
```
while (list != nullptr) {
    Cell* next = list->next;
    delete list;
    list = next;
}
```



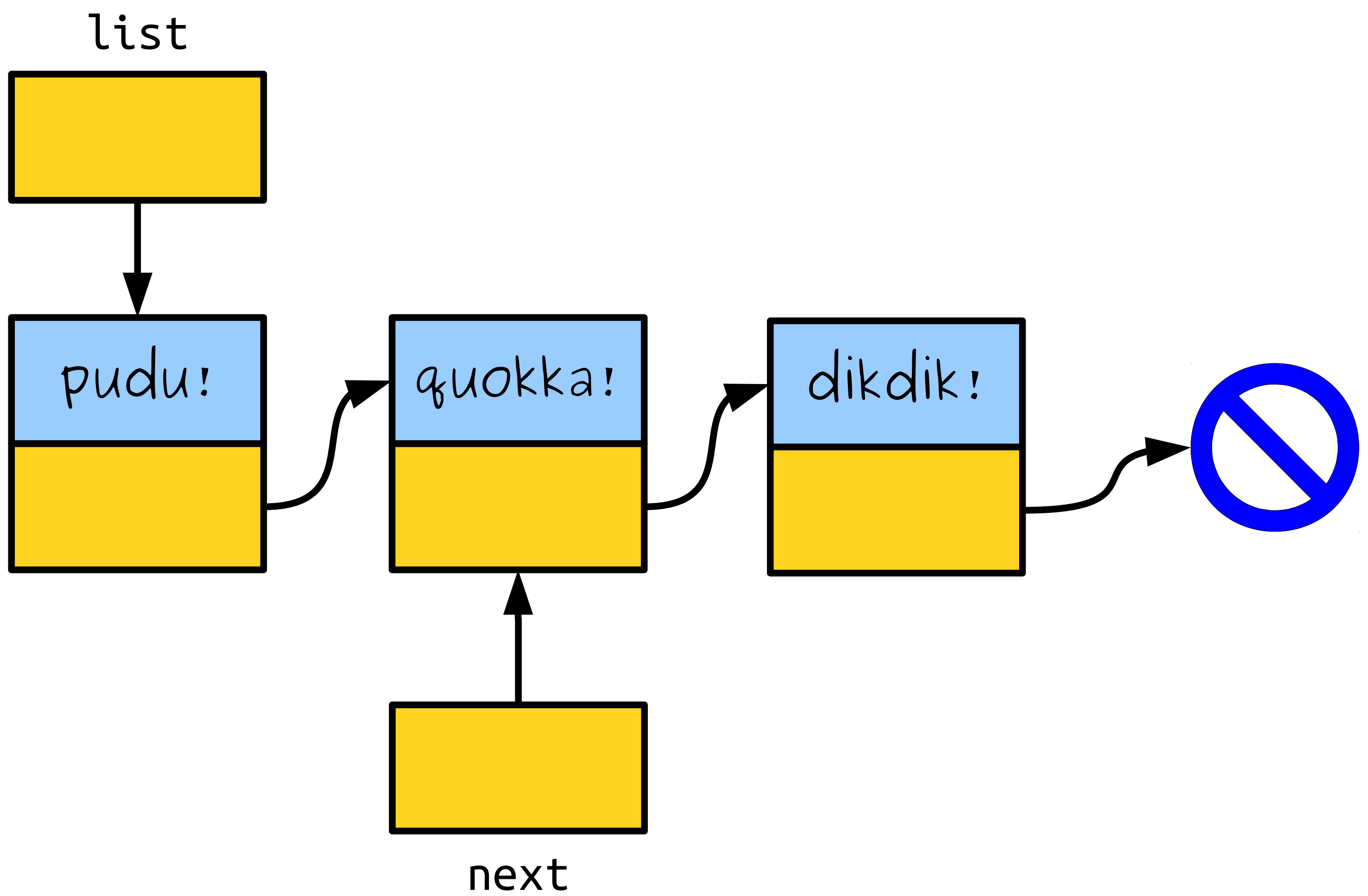
```
while (list != nullptr) {
    Cell* next = list->next;
    delete list;
    list = next;
}
```



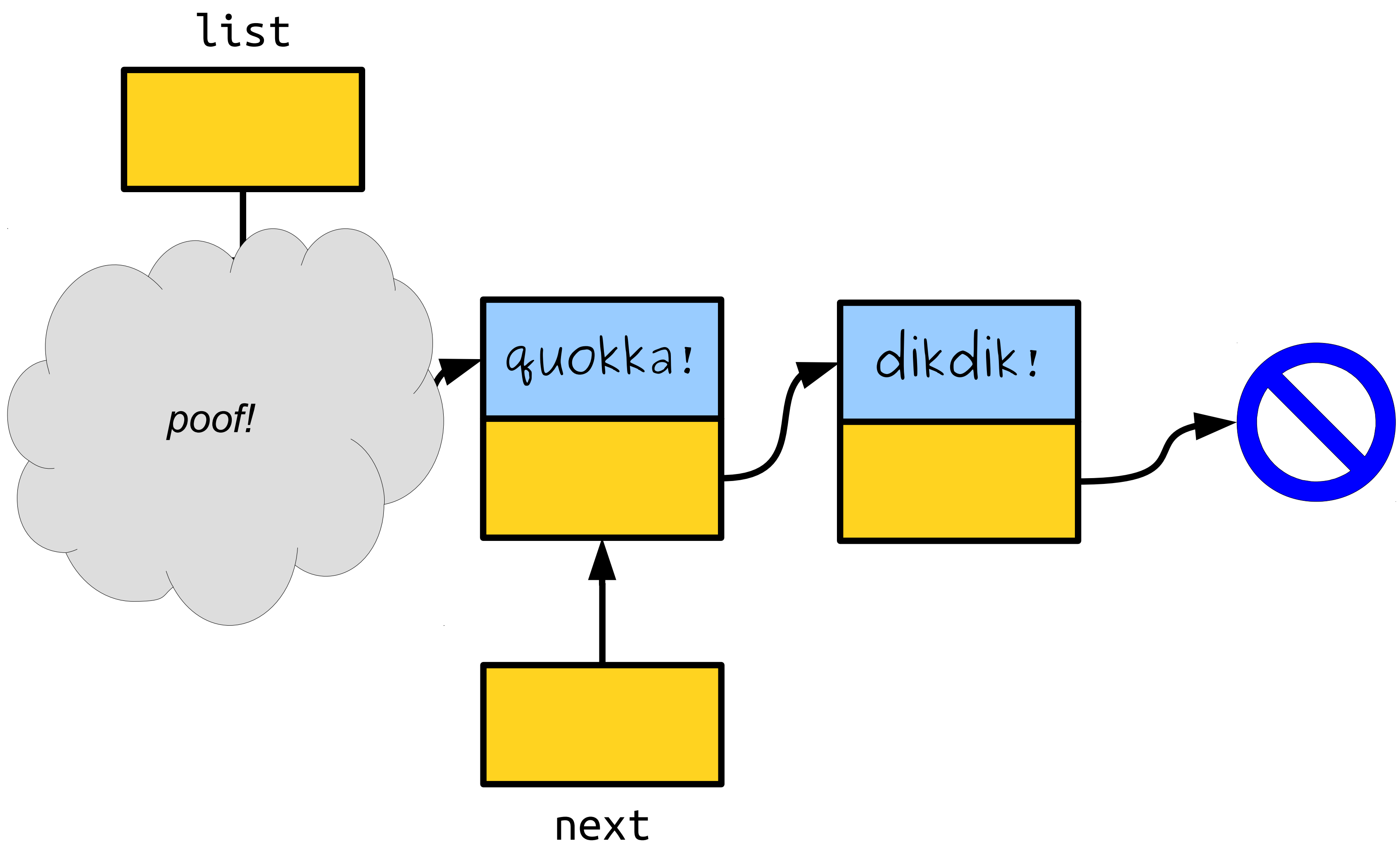
```
while (list != nullptr) {
    Cell* next = list->next;
    delete list;
    list = next;
}
```



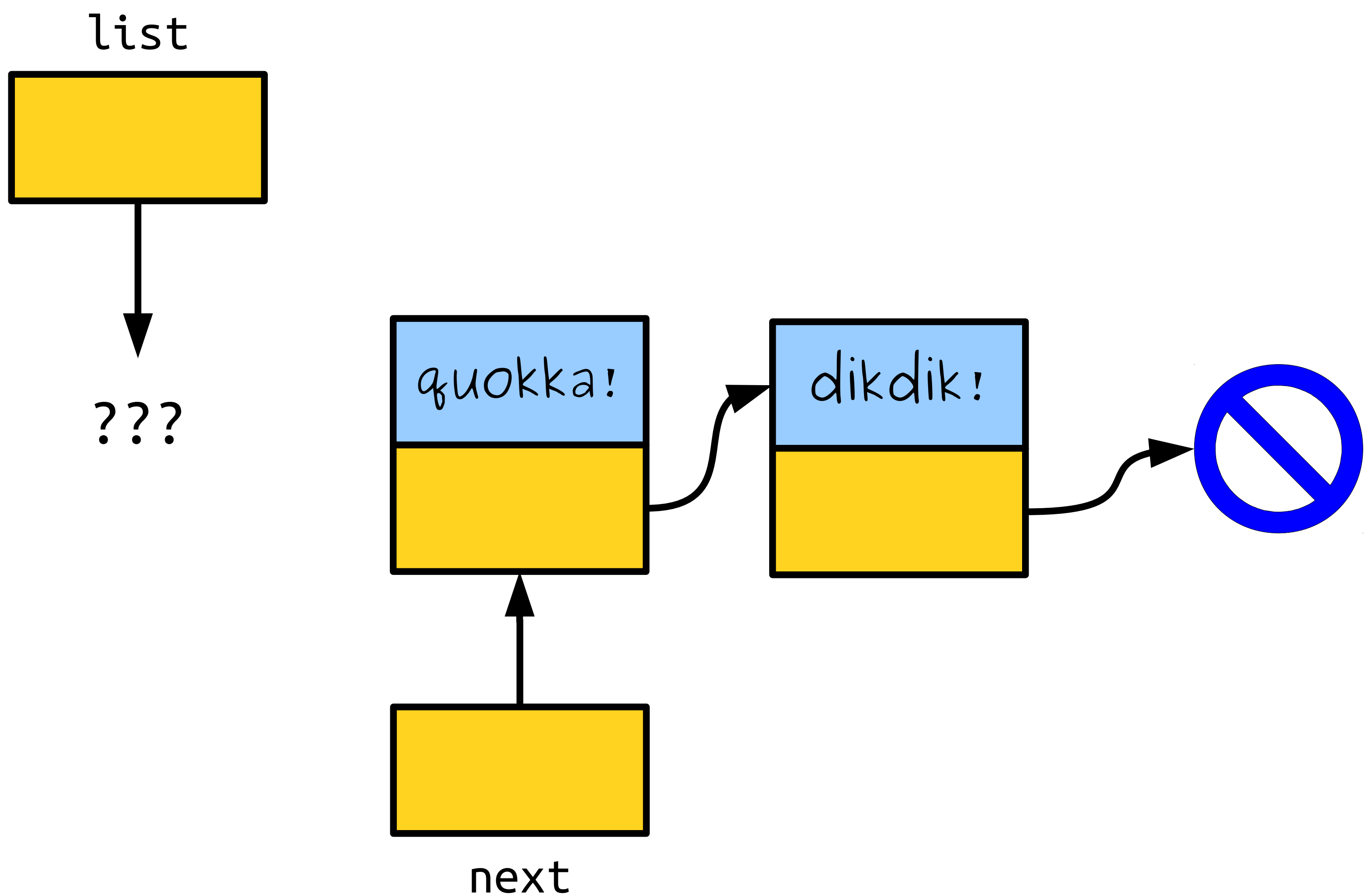
```
while (list != nullptr) {
    Cell* next = list->next;
    delete list;
    list = next;
}
```



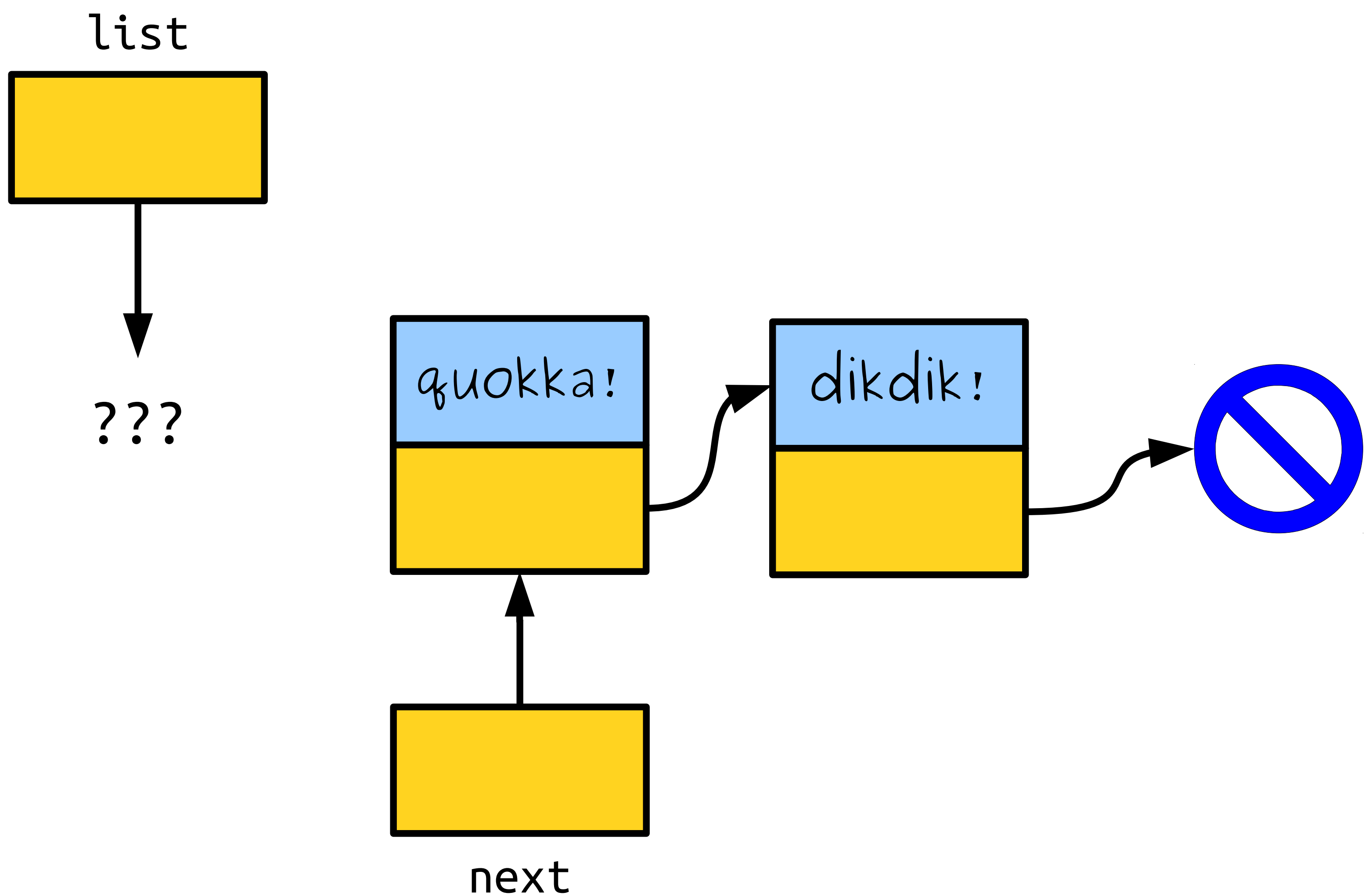
```
while (list != nullptr) {
    Cell* next = list->next;
    delete list;
    list = next;
}
```



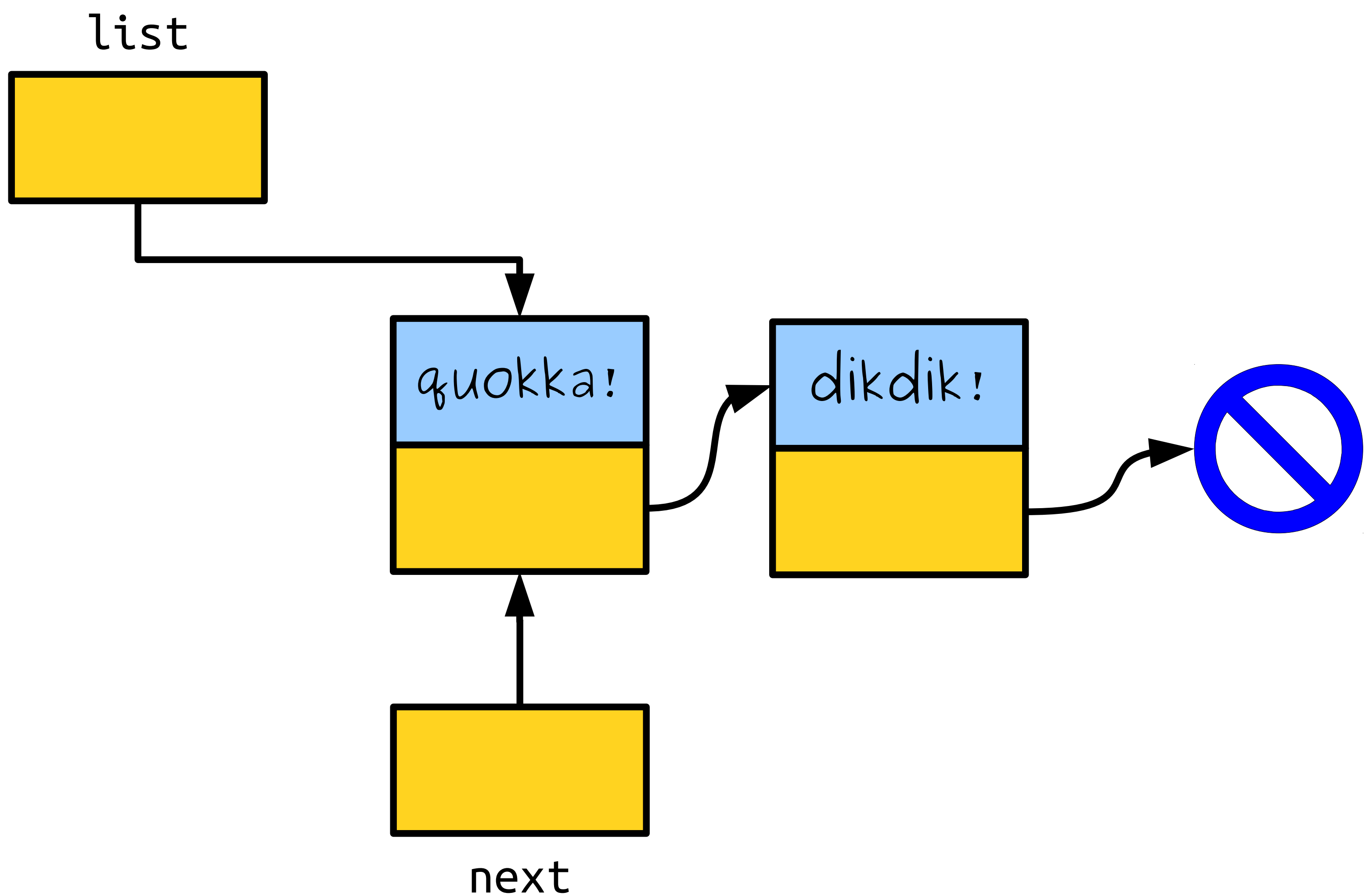
```
while (list != nullptr) {
    Cell* next = list->next;
    delete list;
    list = next;
}
```



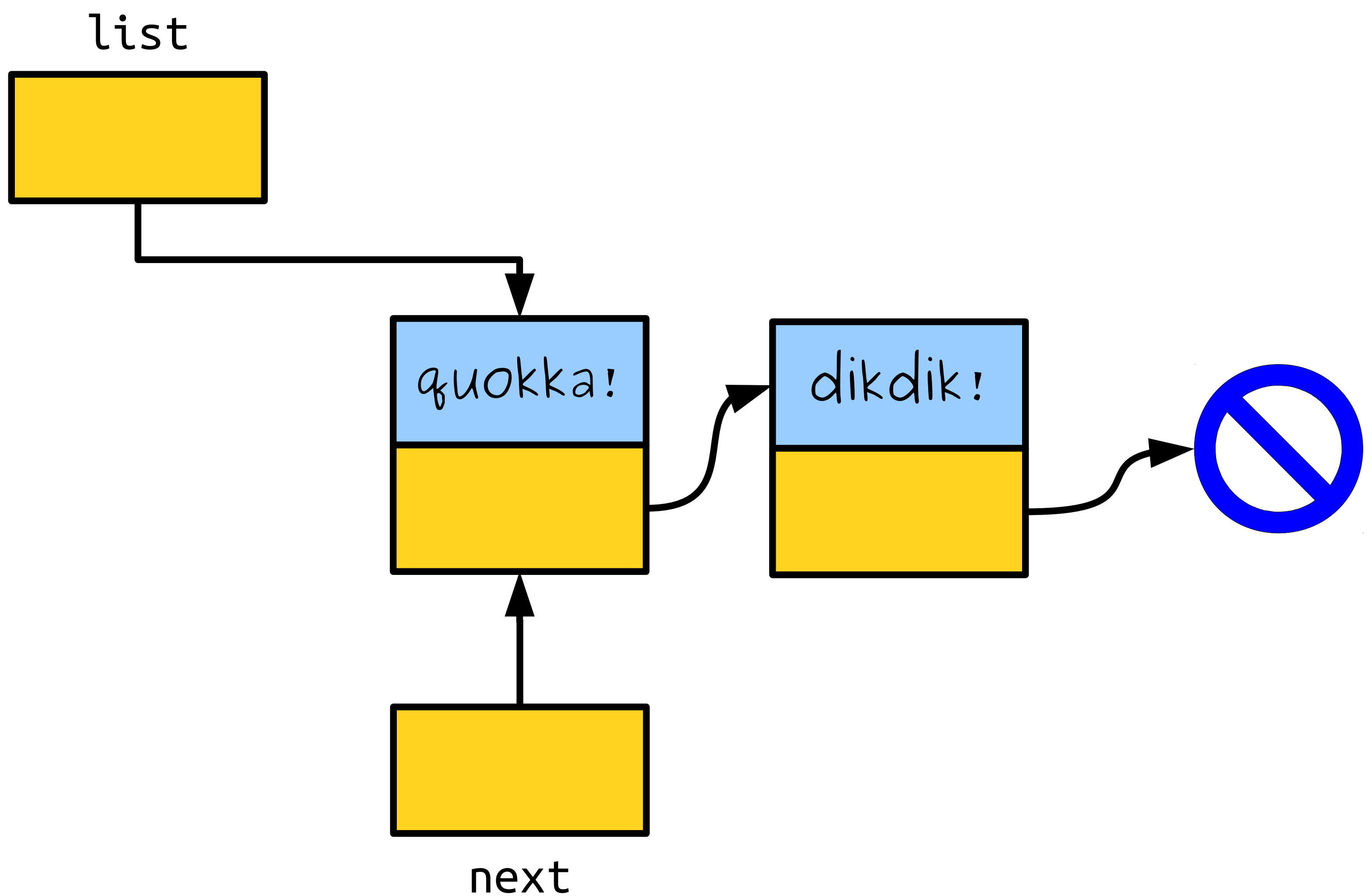
```
while (list != nullptr) {
    Cell* next = list->next;
    delete list;
    list = next;
}
```



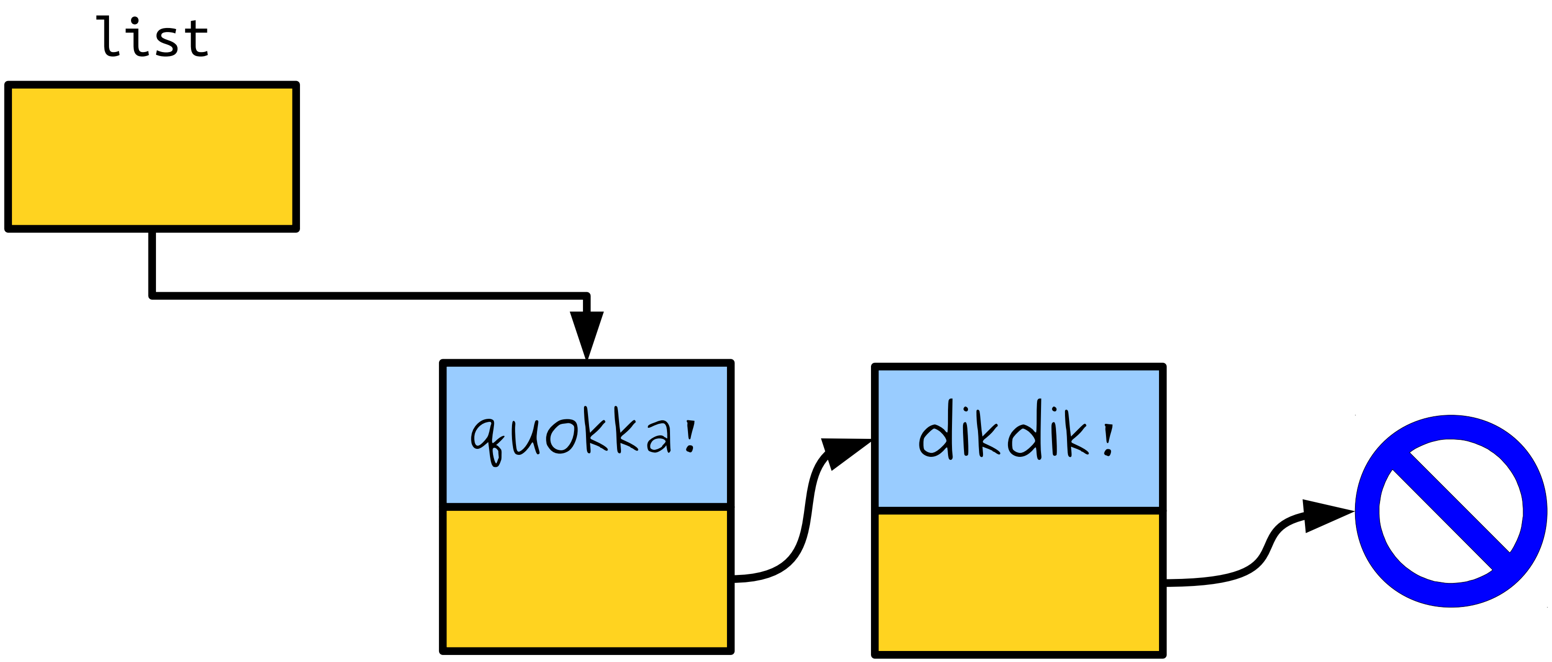
```
while (list != nullptr) {
    Cell* next = list->next;
    delete list;
    list = next;
}
```



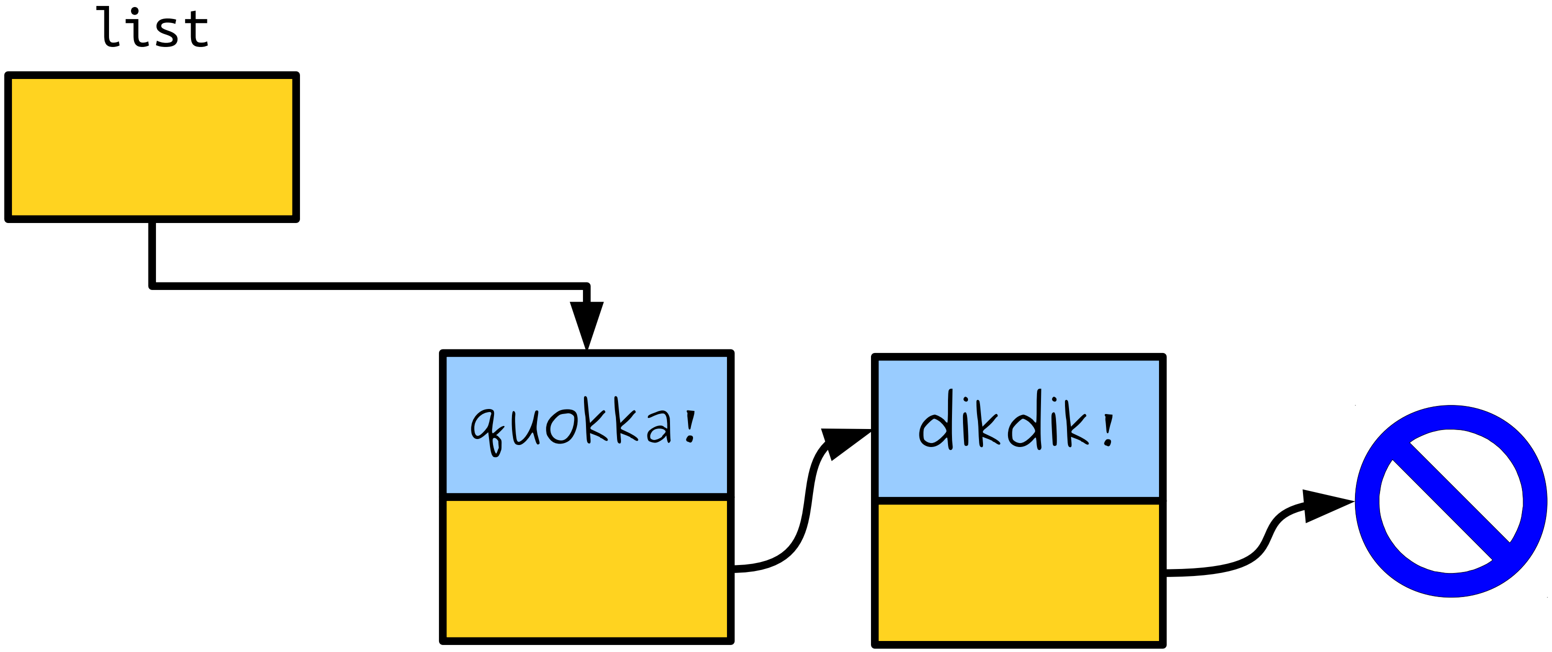
```
while (list != nullptr) {
    Cell* next = list->next;
    delete list;
    list = next;
}
```



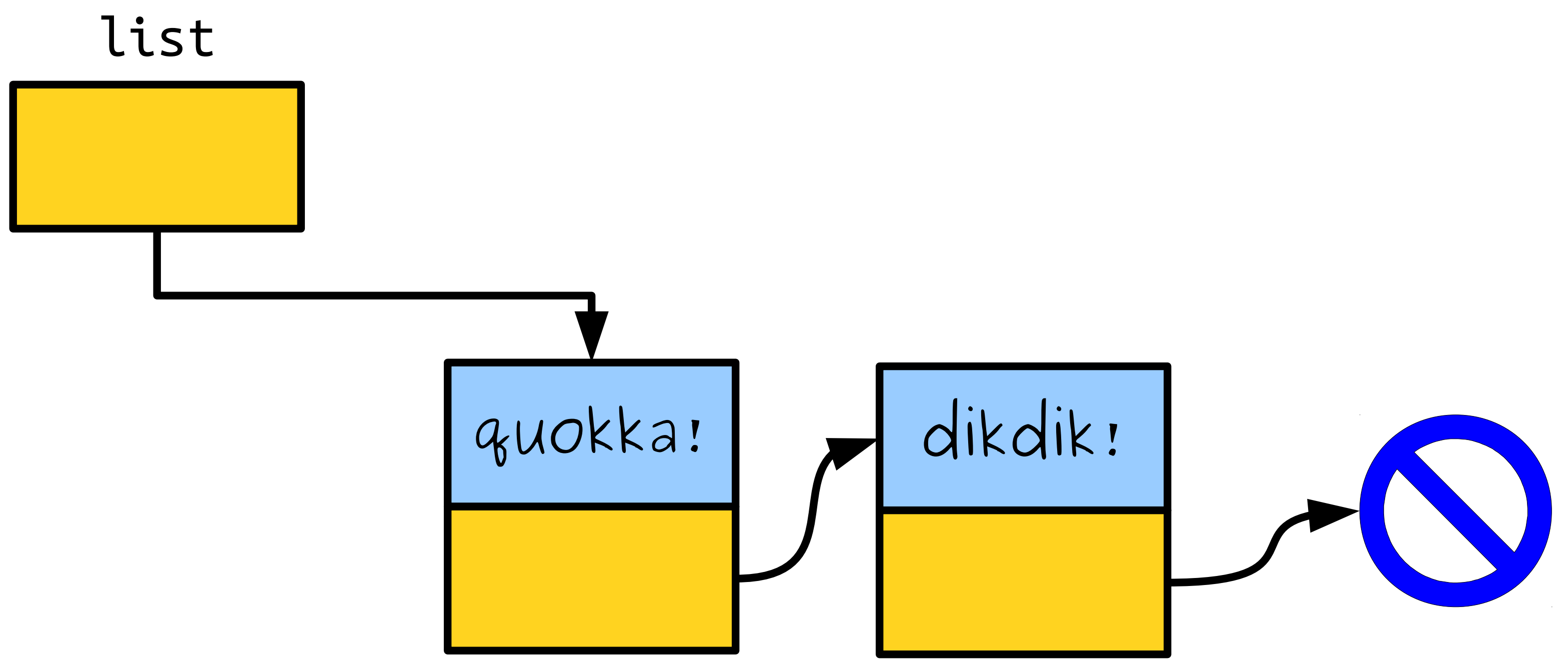
```
while (list != nullptr) {
    Cell* next = list->next;
    delete list;
    list = next;
}
```



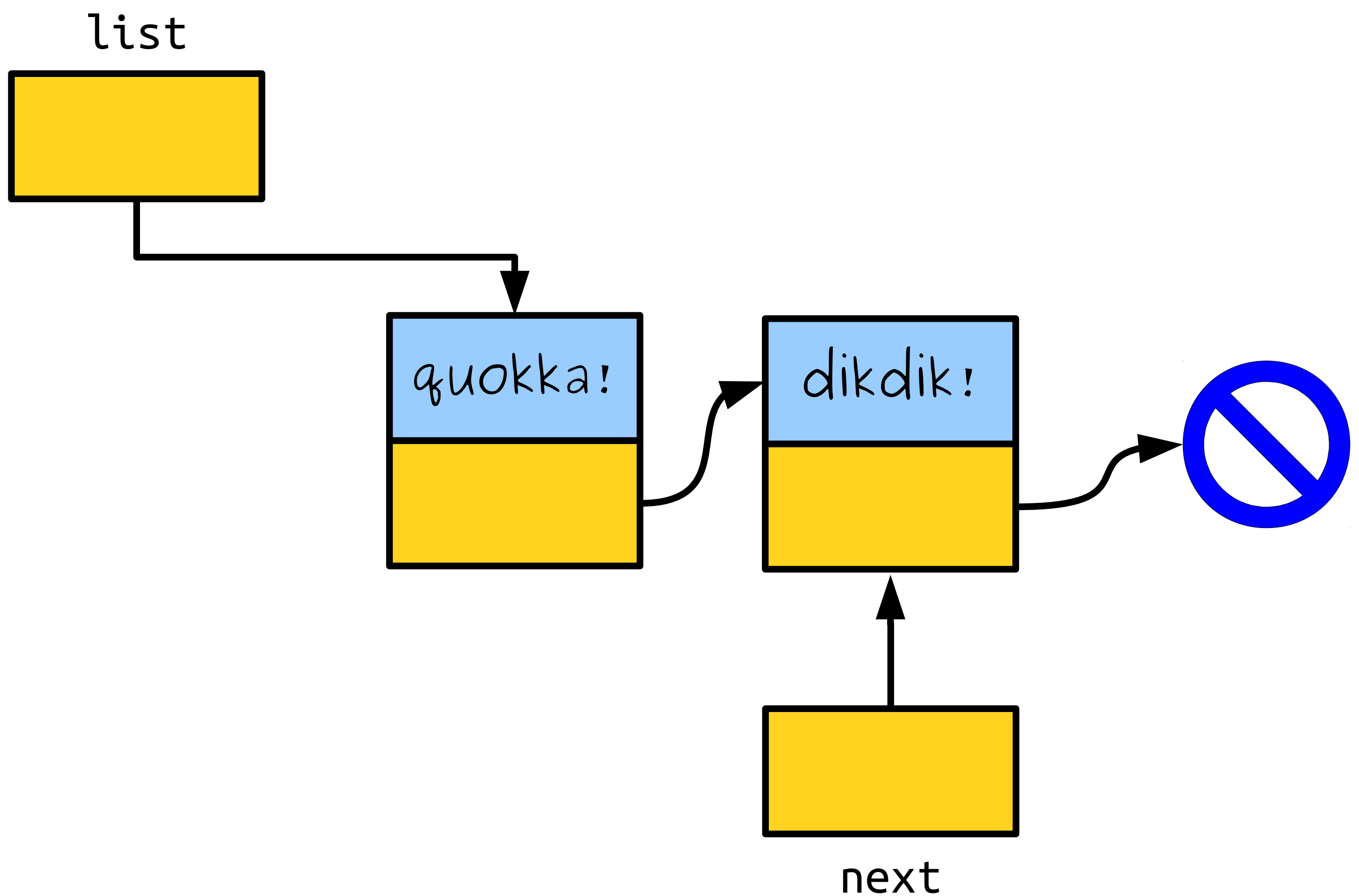
```
while (list != nullptr) {
    Cell* next = list->next;
    delete list;
    list = next;
}
```



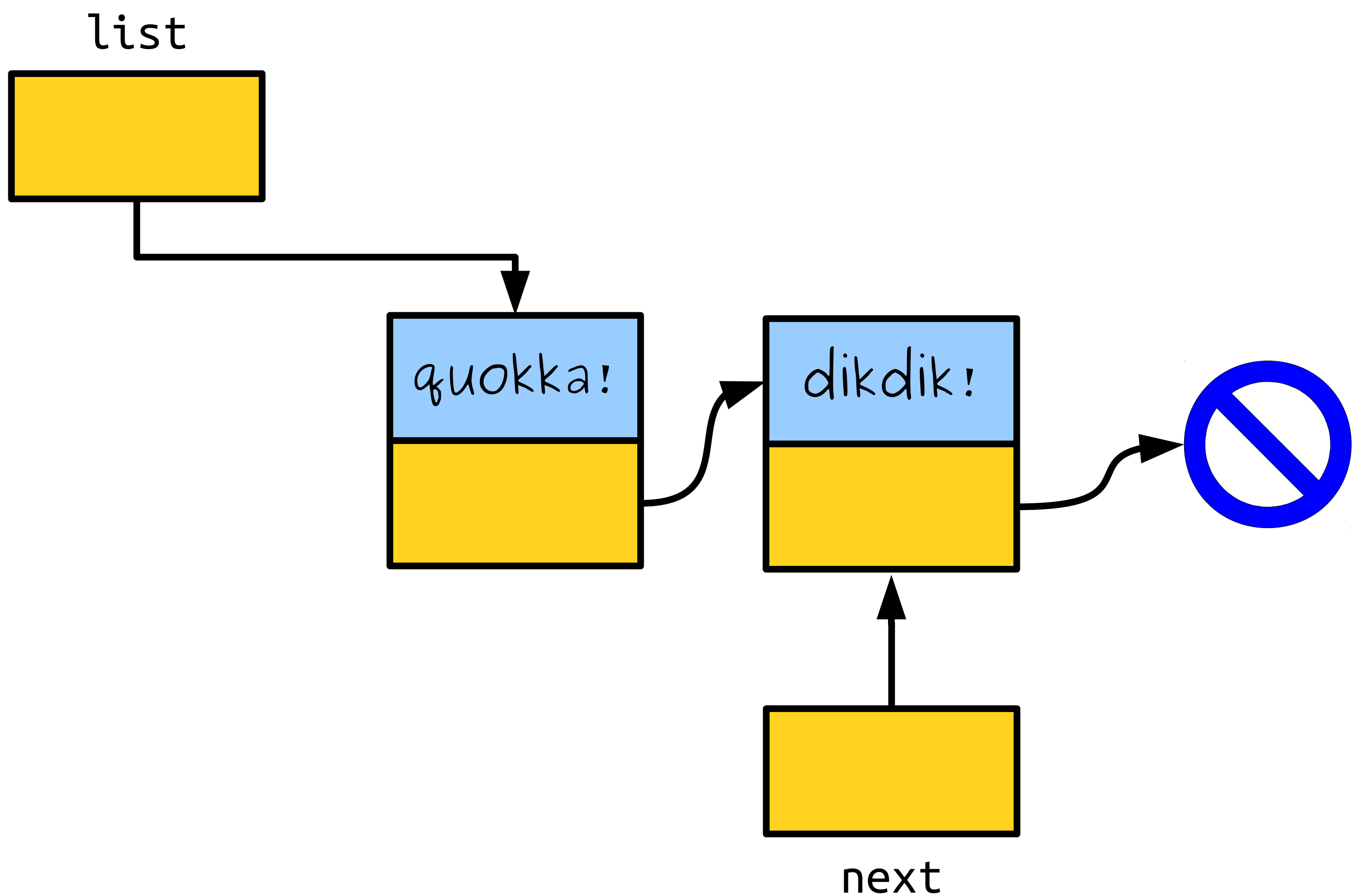
```
while (list != nullptr) {
    Cell* next = list->next;
    delete list;
    list = next;
}
```



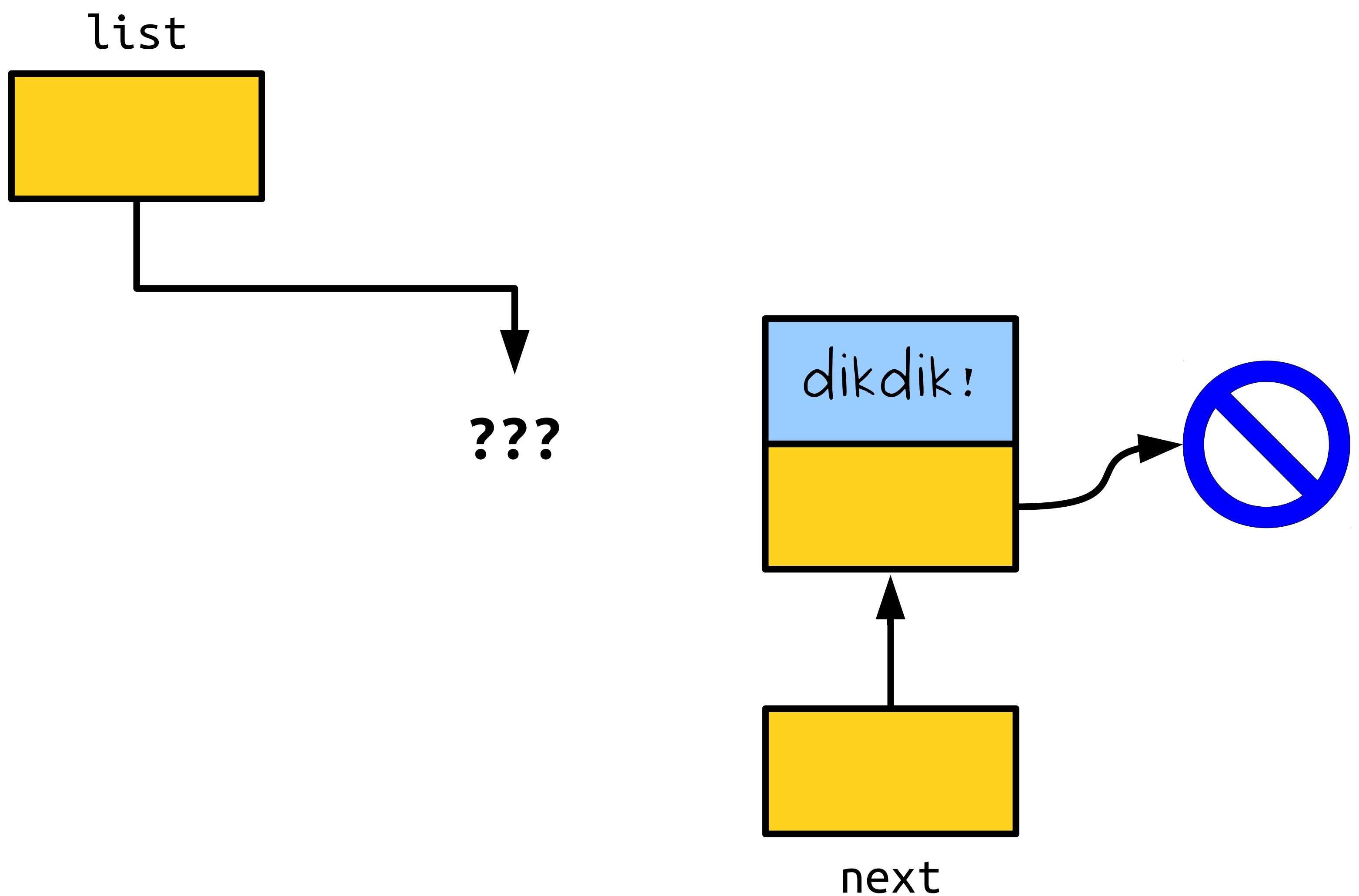
```
while (list != nullptr) {
    Cell* next = list->next;
    delete list;
    list = next;
}
```



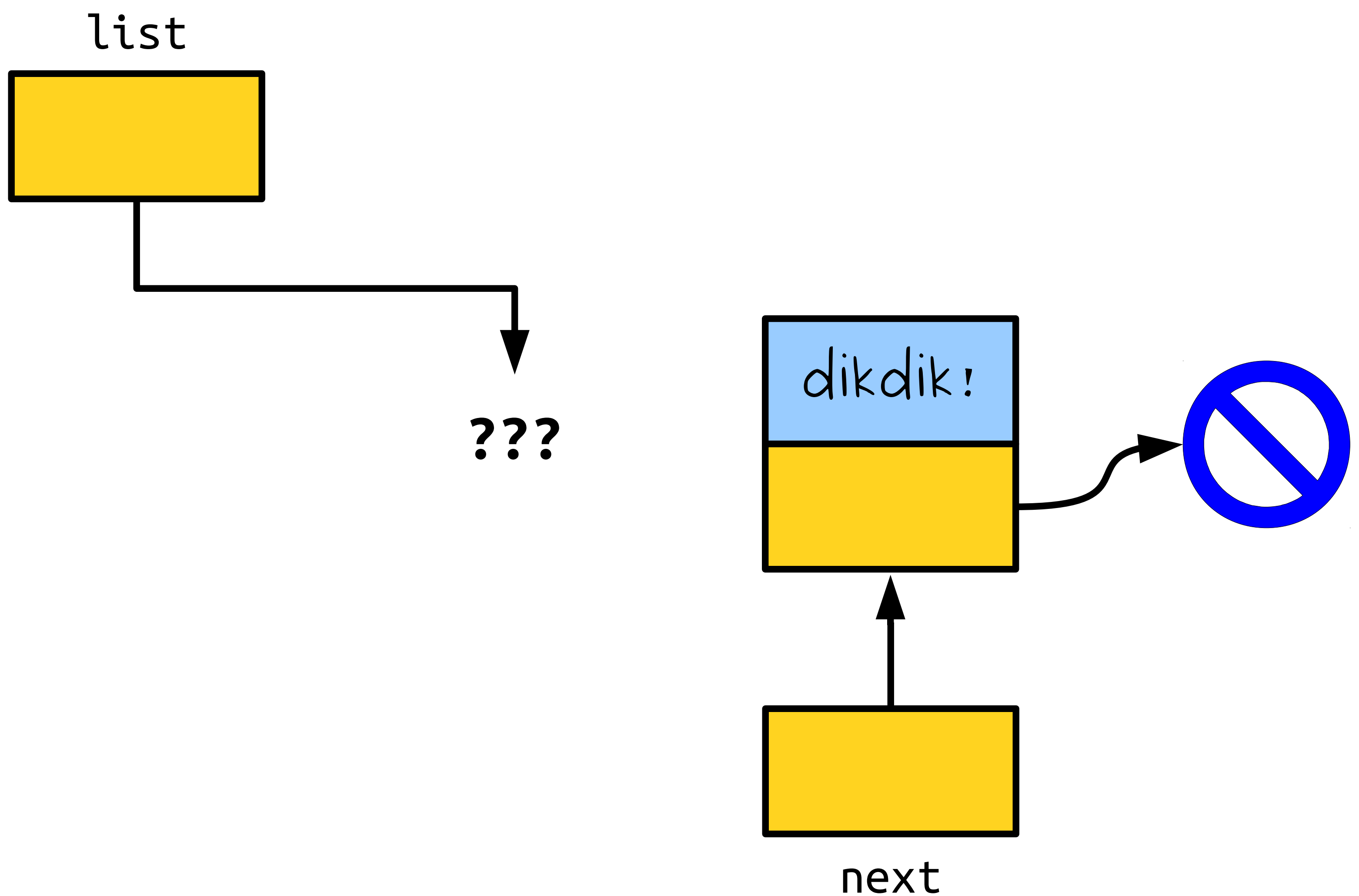
```
while (list != nullptr) {
    Cell* next = list->next;
    delete list;
    list = next;
}
```



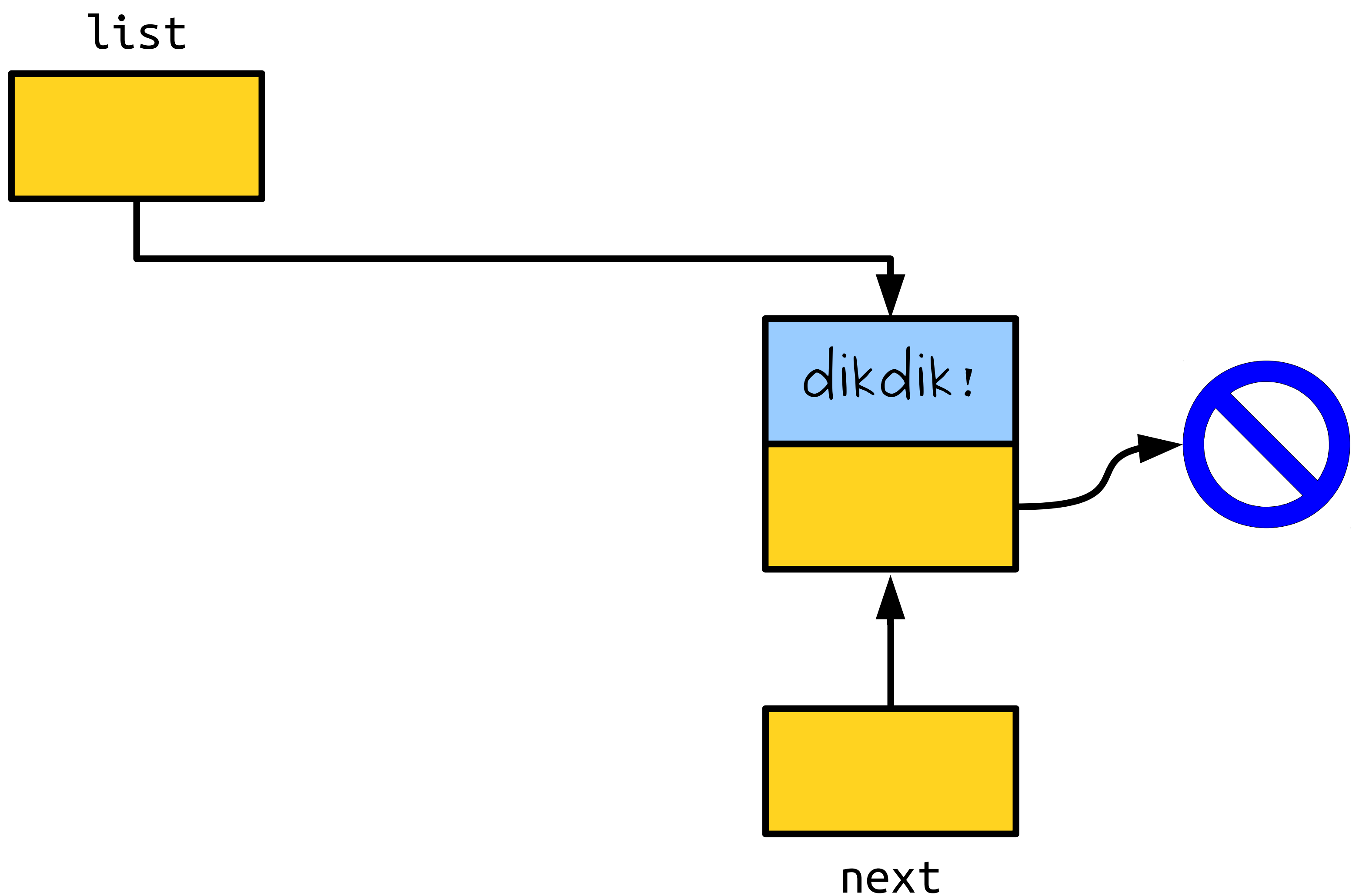
```
while (list != nullptr) {
    Cell* next = list->next;
    delete list;
    list = next;
}
```



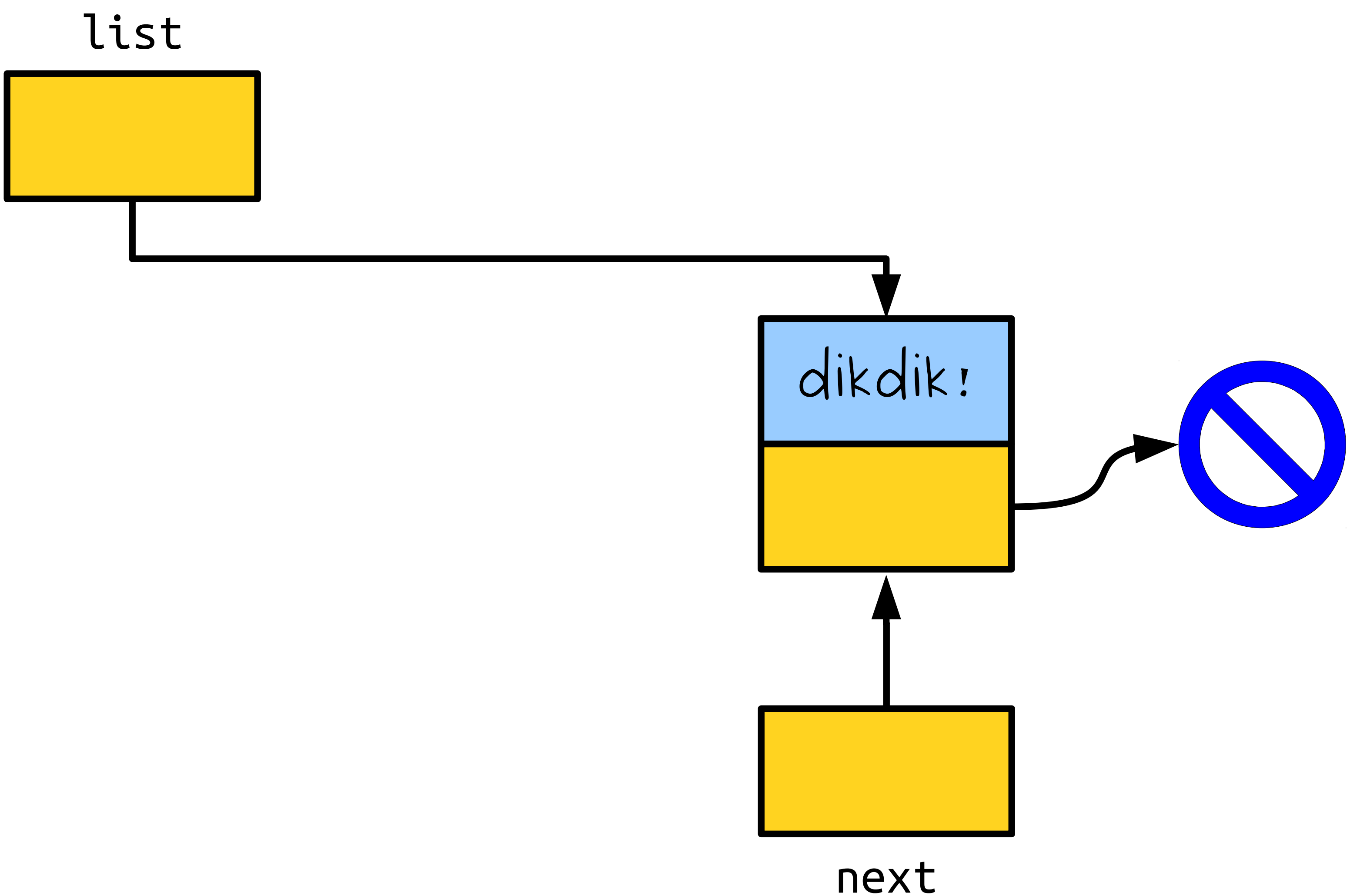
```
while (list != nullptr) {
    Cell* next = list->next;
    delete list;
    list = next;
}
```



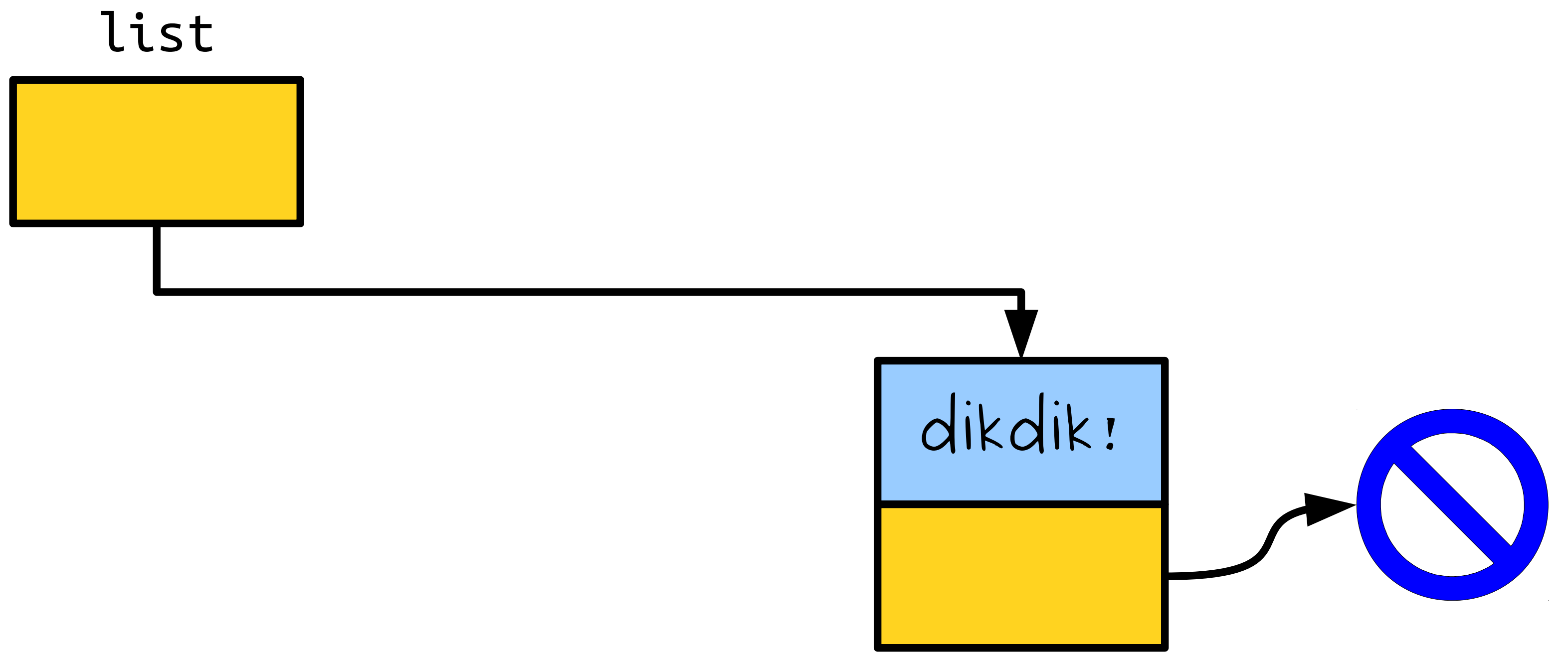
```
while (list != nullptr) {
    Cell* next = list->next;
    delete list;
    list = next;
}
```



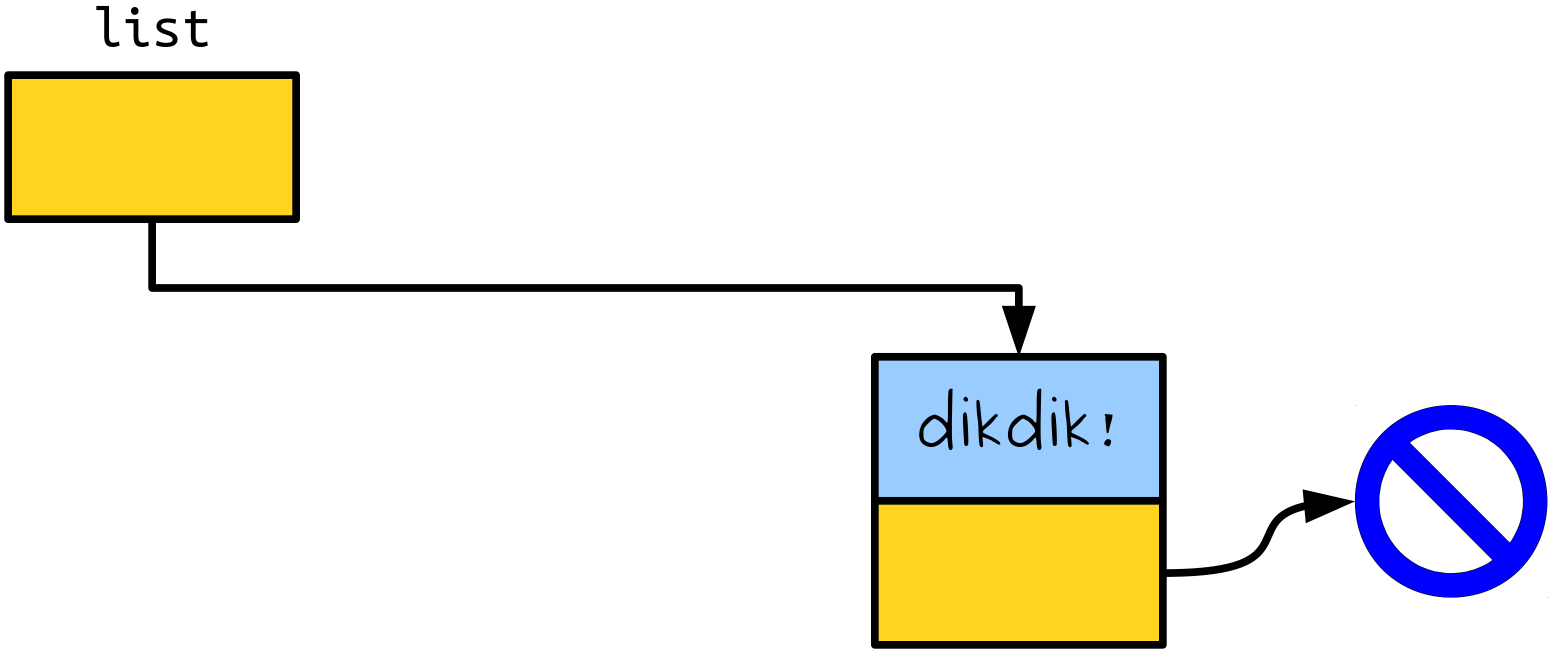
```
while (list != nullptr) {
    Cell* next = list->next;
    delete list;
    list = next;
}
```



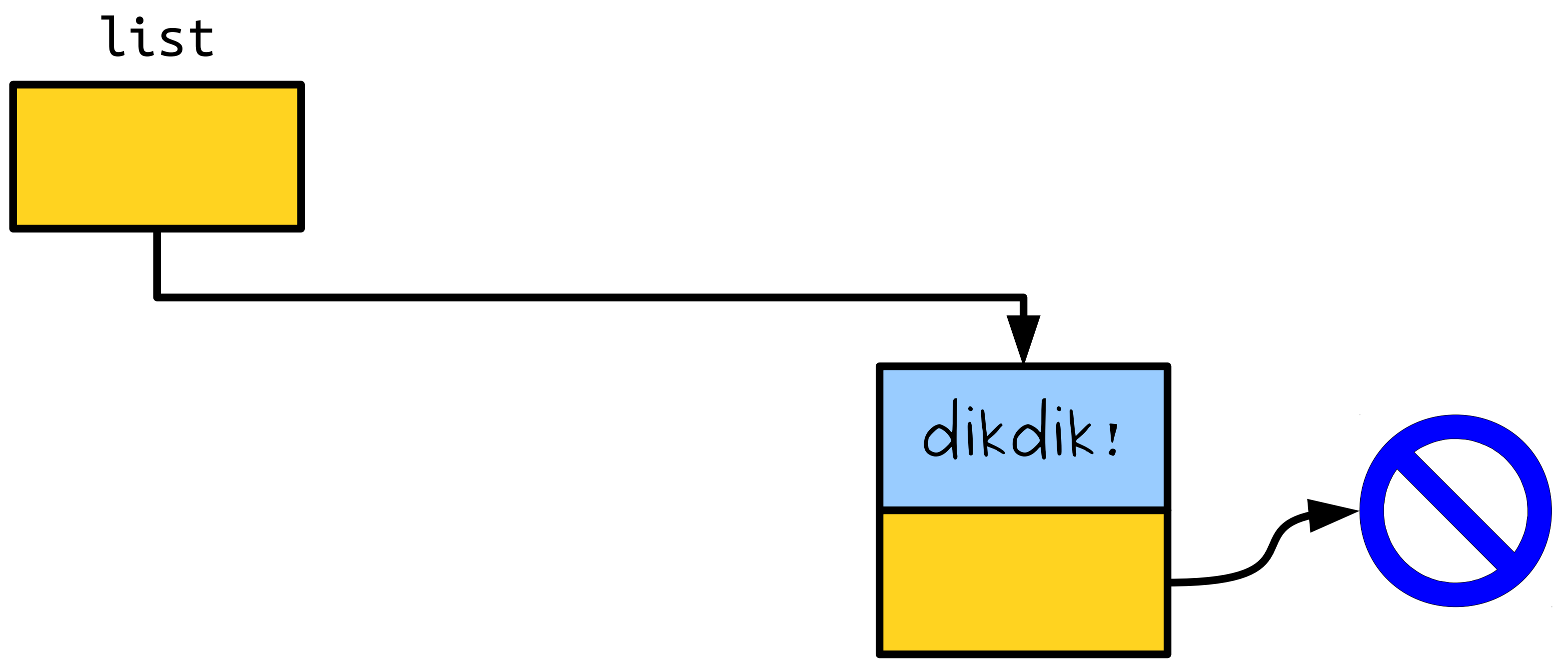
```
while (list != nullptr) {
    Cell* next = list->next;
    delete list;
    list = next;
}
```



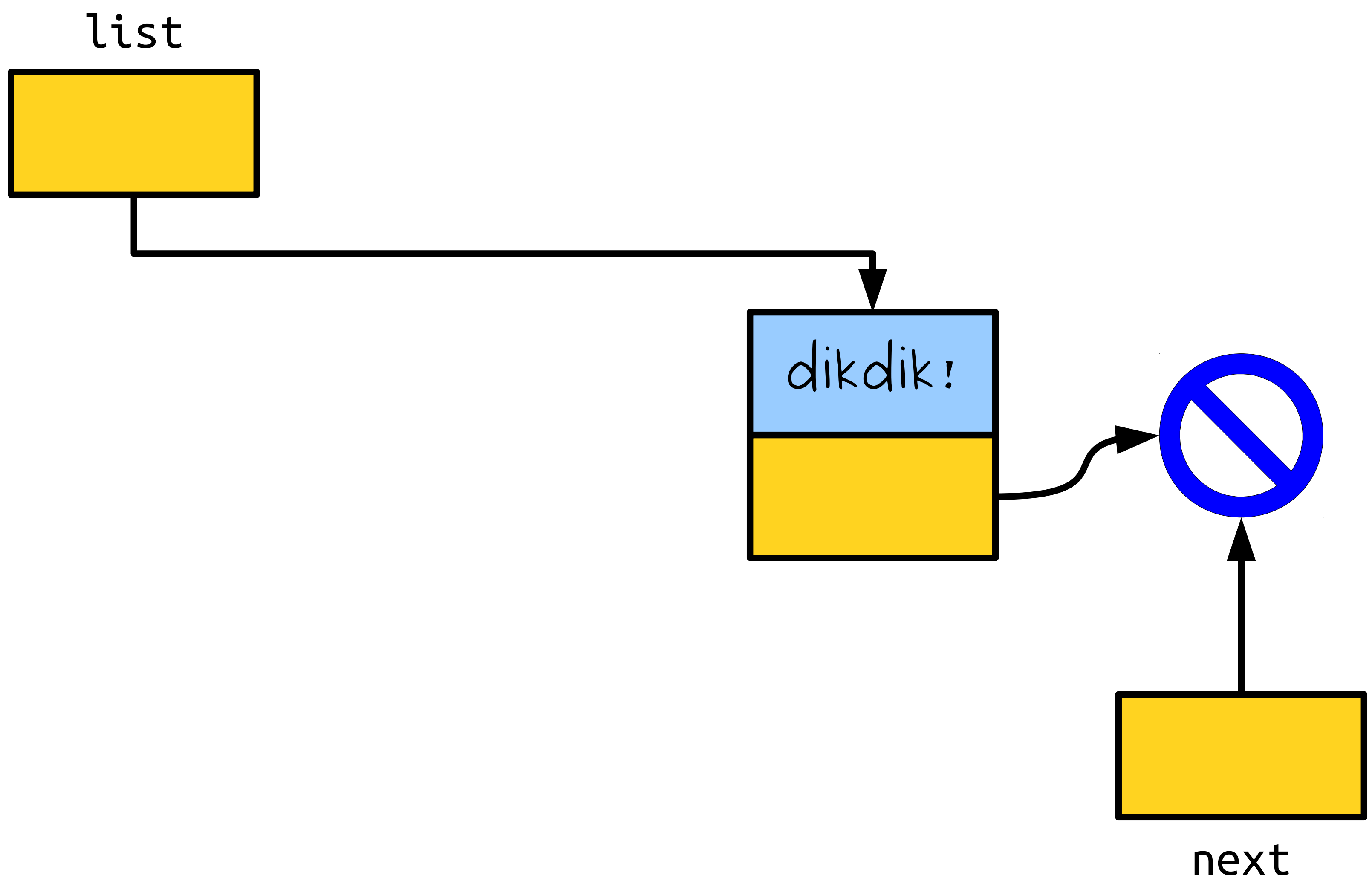
```
while (list != nullptr) {
    Cell* next = list->next;
    delete list;
    list = next;
}
```



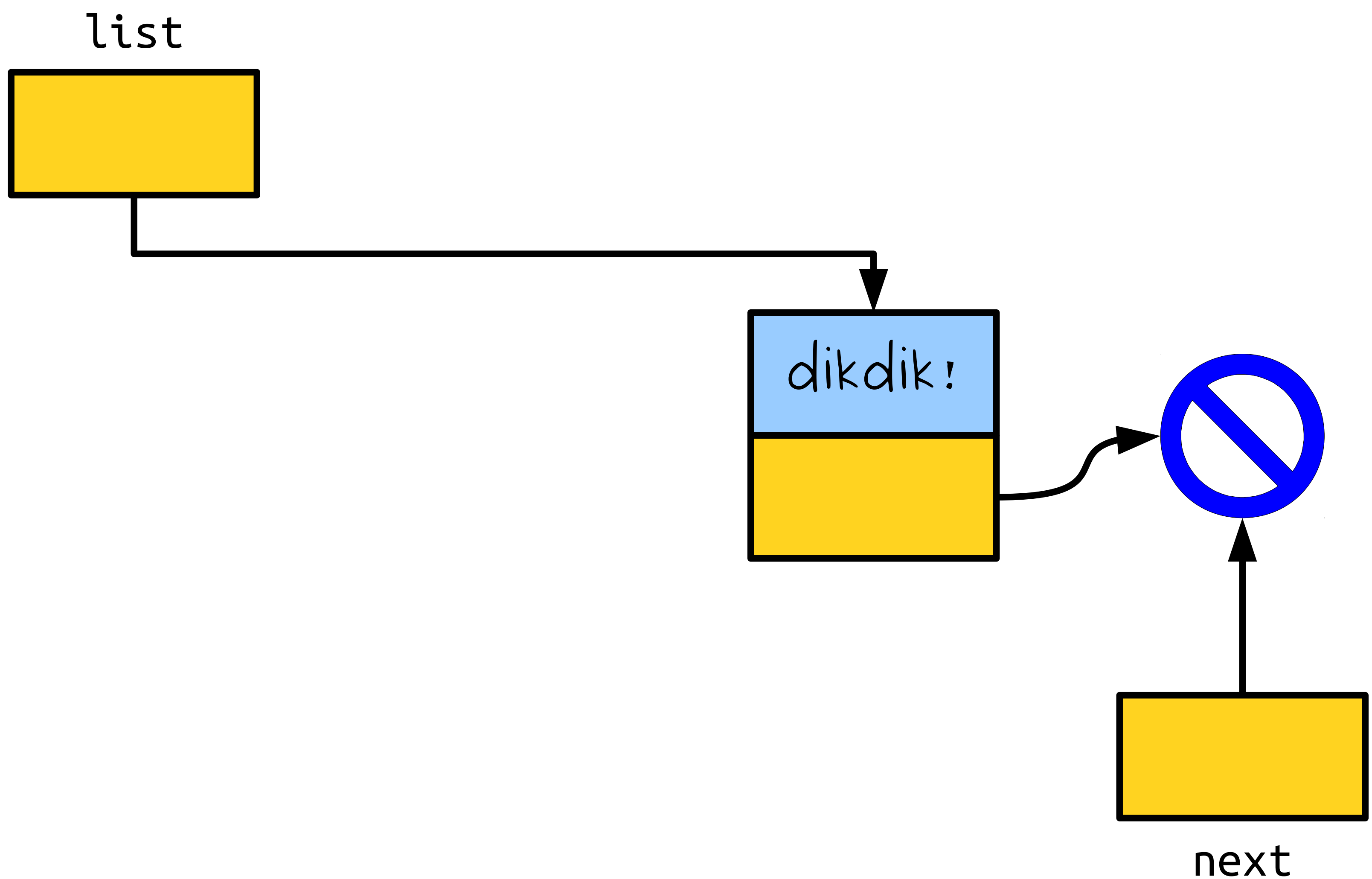
```
while (list != nullptr) {
    Cell* next = list->next;
    delete list;
    list = next;
}
```



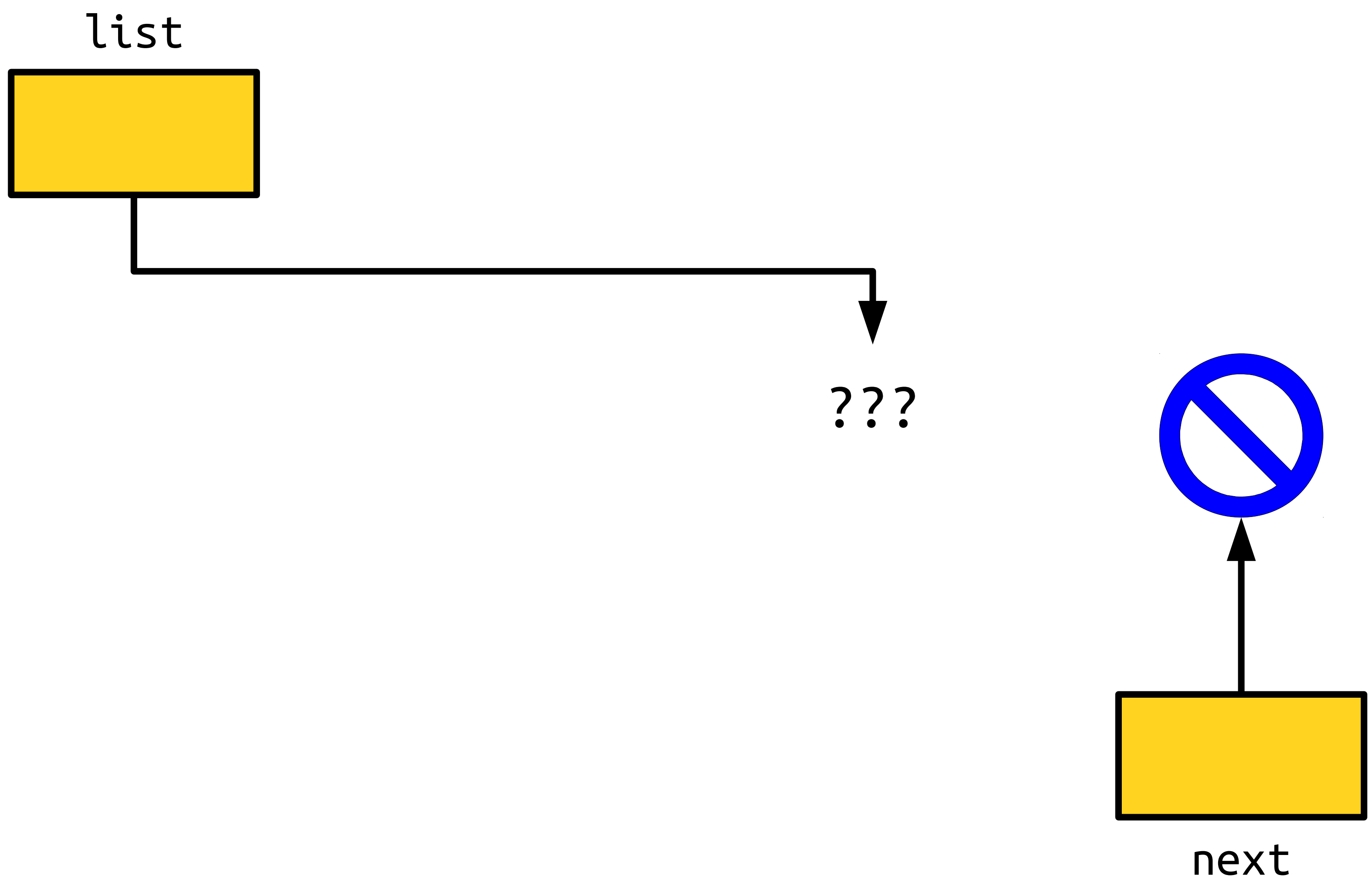
```
while (list != nullptr) {
    Cell* next = list->next;
    delete list;
    list = next;
}
```



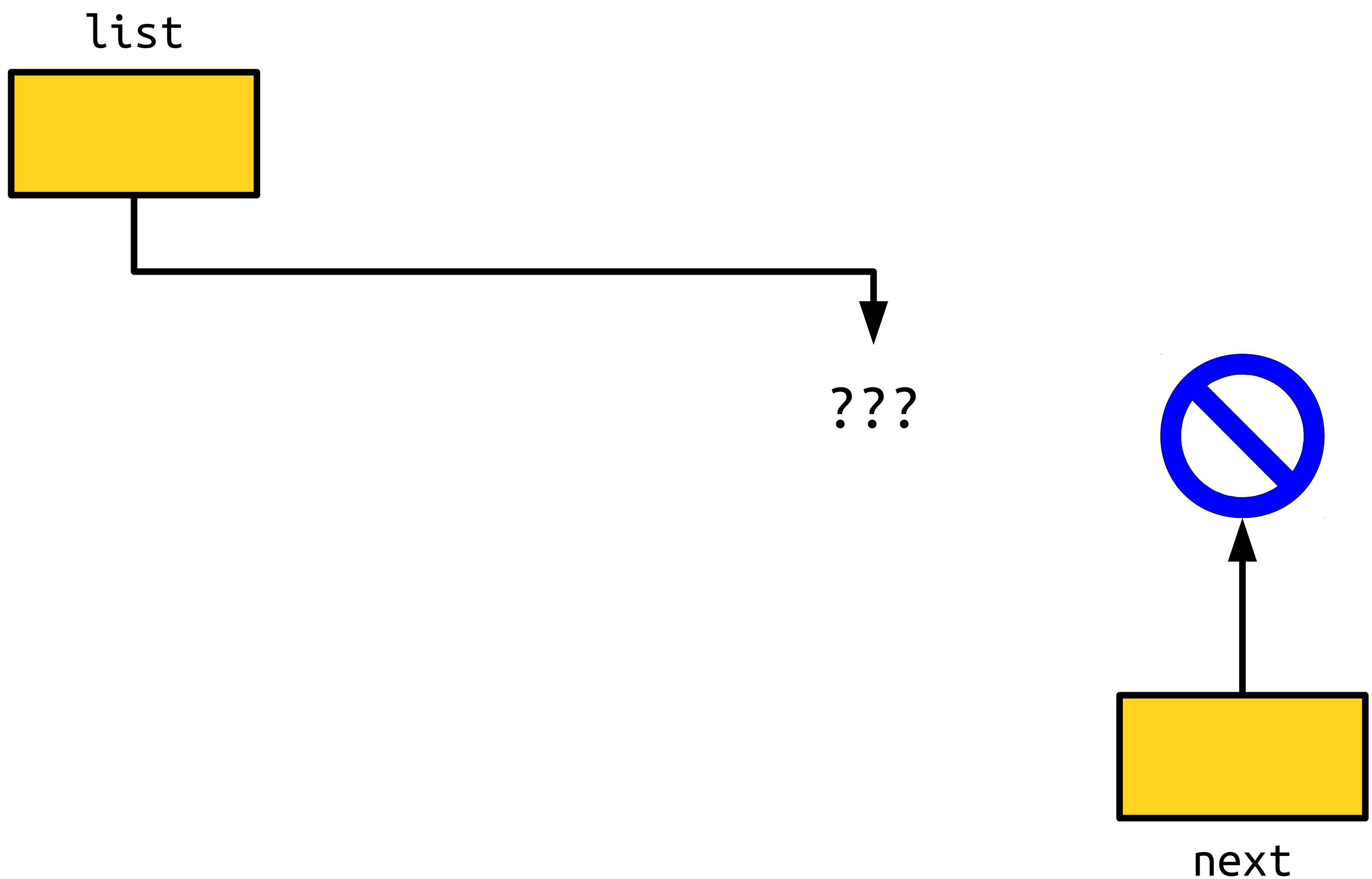
```
while (list != nullptr) {
    Cell* next = list->next;
    delete list;
    list = next;
}
```



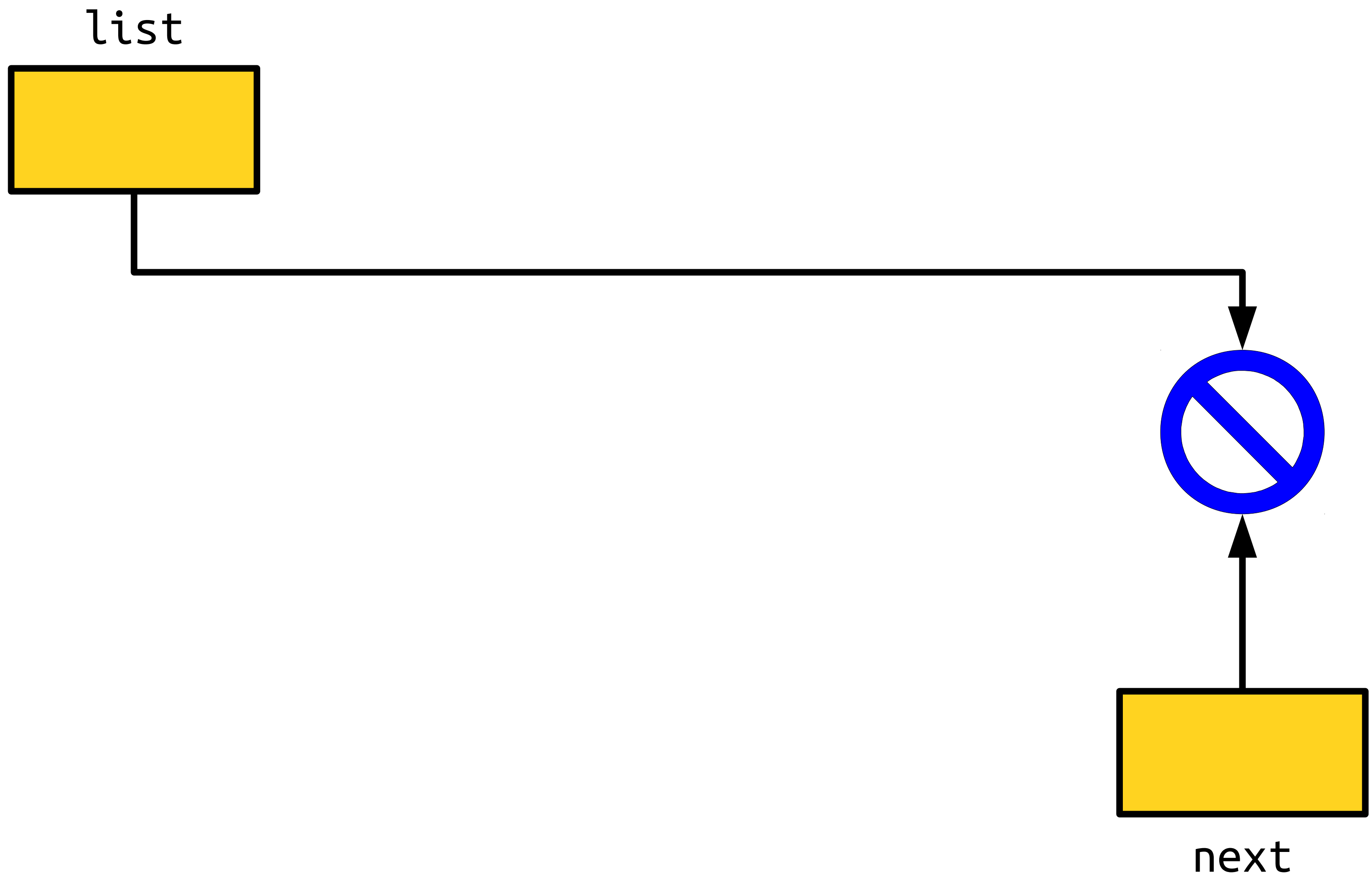
```
while (list != nullptr) {
    Cell* next = list->next;
    delete list;
    list = next;
}
```



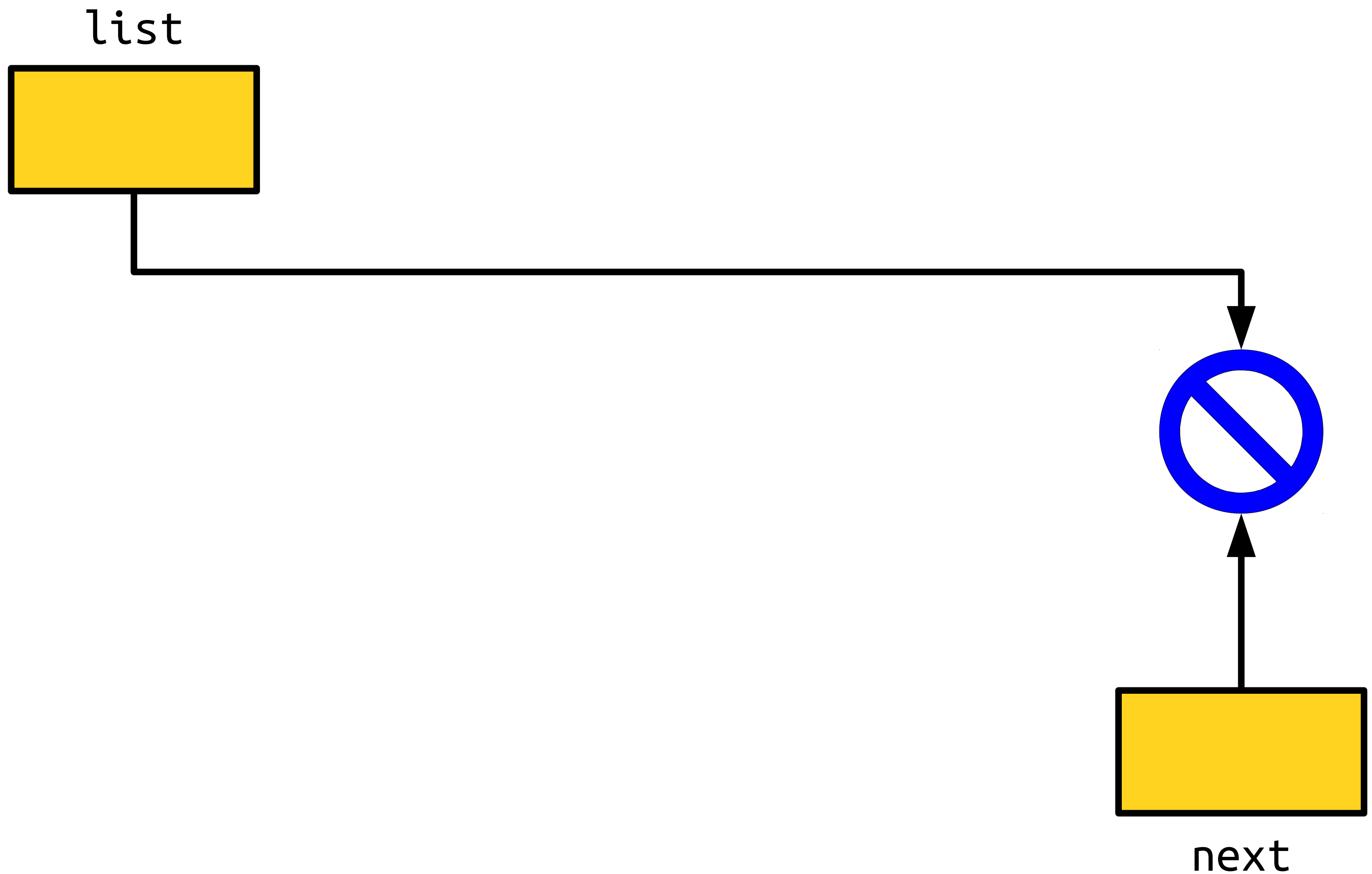
```
while (list != nullptr) {
    Cell* next = list->next;
    delete list;
    list = next;
}
```



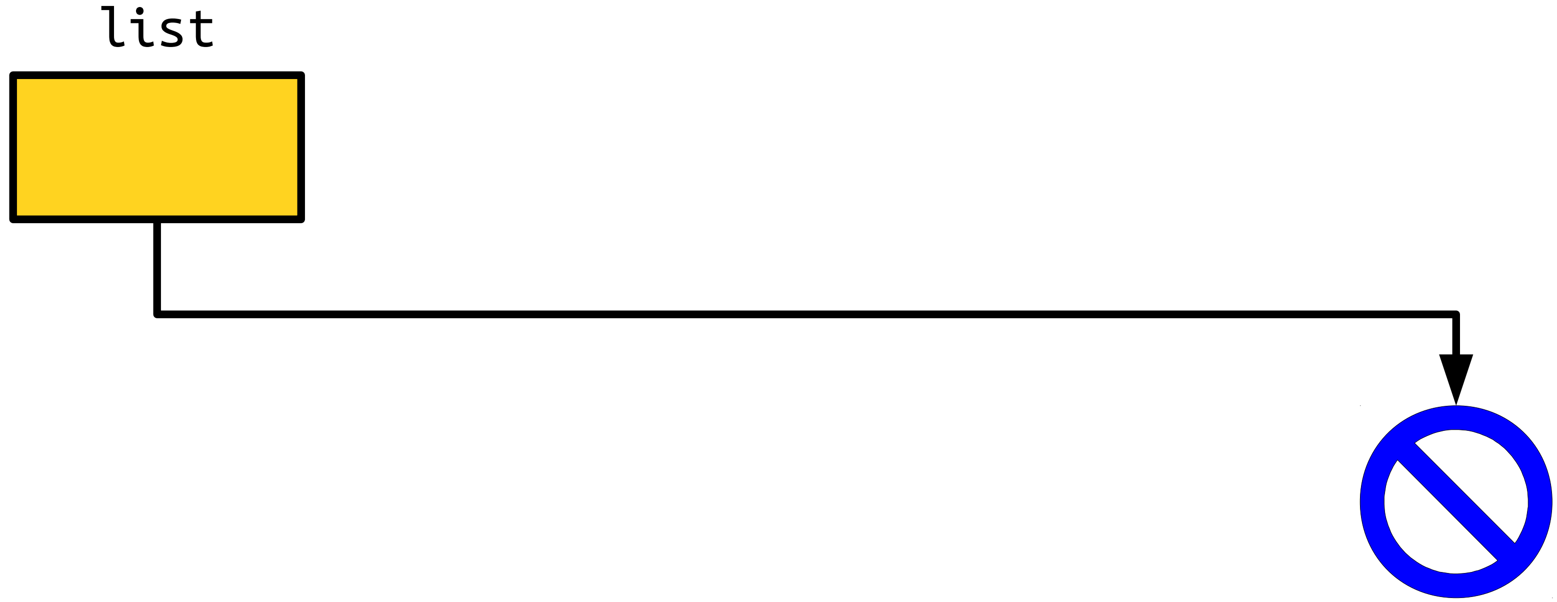
```
while (list != nullptr) {
    Cell* next = list->next;
    delete list;
    list = next;
}
```



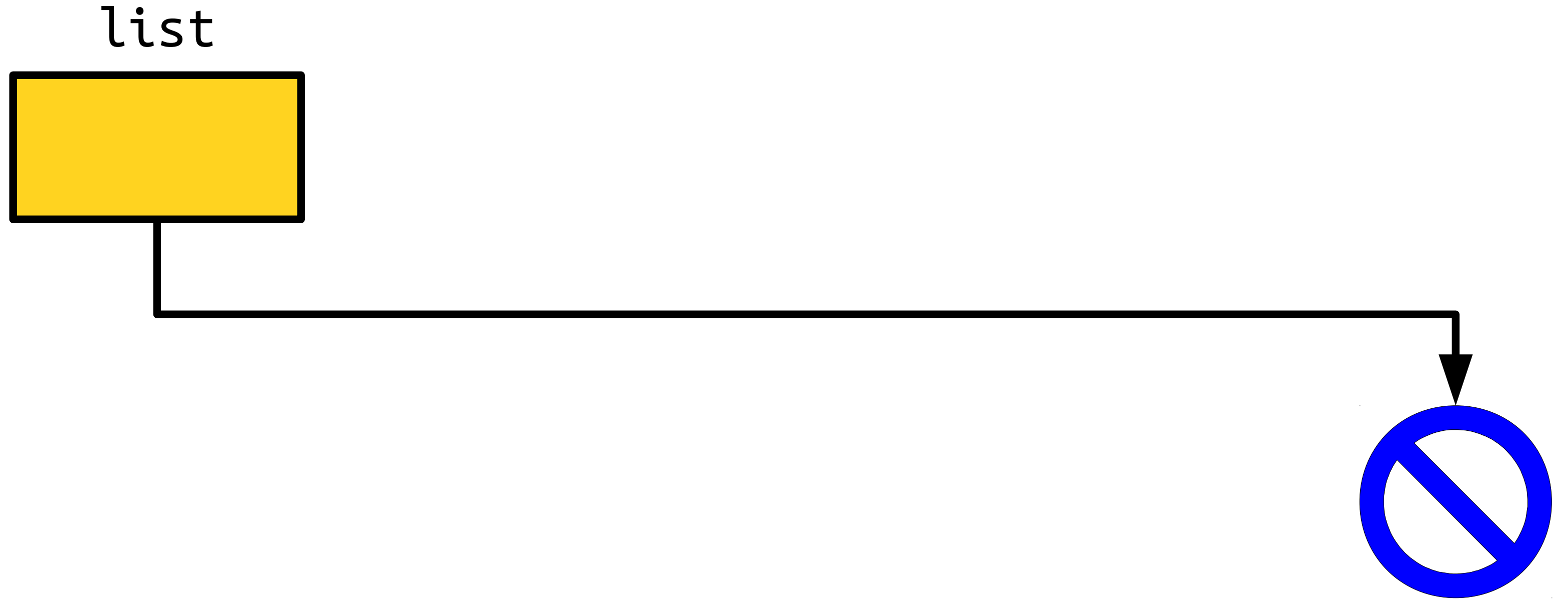
```
while (list != nullptr) {
    Cell* next = list->next;
    delete list;
    list = next;
}
```



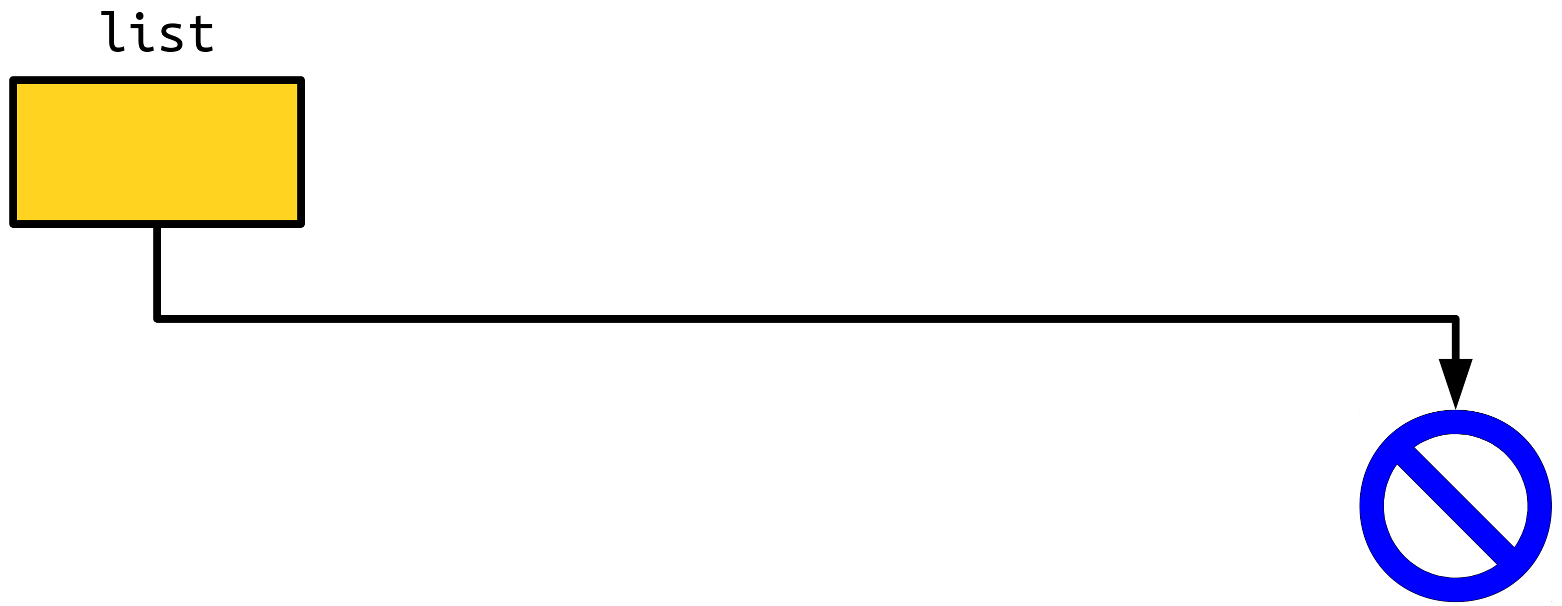
```
while (list != nullptr) {
    Cell* next = list->next;
    delete list;
    list = next;
}
```



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

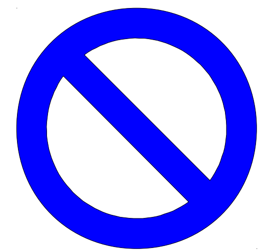


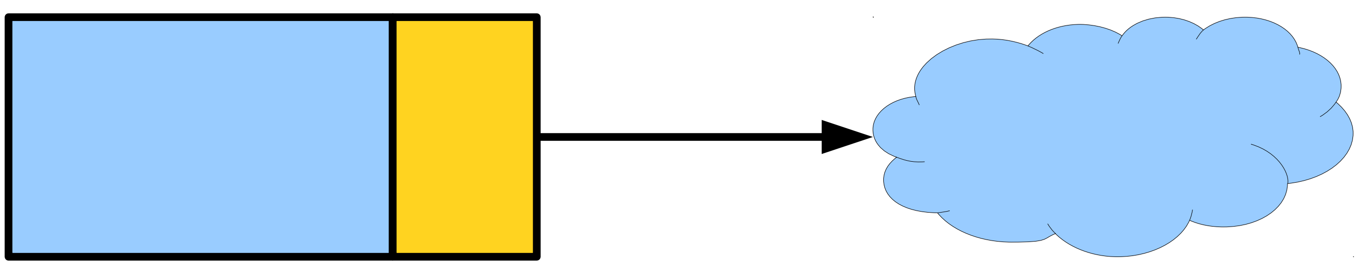
```
while (list != nullptr) {
    Cell* next = list->next;
    delete list;
    list = next;
}
```



A Linked List is Either…

…an empty list,
represented by
nullptr, or…





a single linked list
cell that points…

… at another linked
list.

Linked Lists: The Tricky Parts

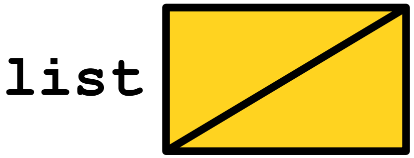
- Suppose that we want to write a function that will add an element to the front of a linked list.
- What might this function look like?

What went wrong?

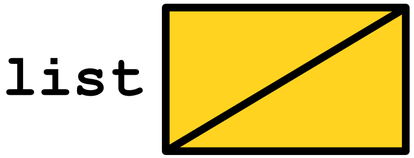
```
int main() {
    Cell* list = nullptr;
    listInsert(list, "A");
    listInsert(list, "B");
    listInsert(list, "C");

    return 0;
}
```

```
int main() {
    Cell* list = nullptr;
    listInsert(list, "A");
    listInsert(list, "B");
    listInsert(list, "C");

    return 0;
}
```

```
int main() {
    Cell* list = nullptr;
    listInsert(list, "A");
    listInsert(list, "B");
    listInsert(list, "C");

    return 0;
}
```



```
int main() {
    Cell* list = nullptr;
    listInsert(list, "A");
    listInsert(list, "B");
    listInsert(list, "C");

    return 0;
}
```



```
int main() {

}
```

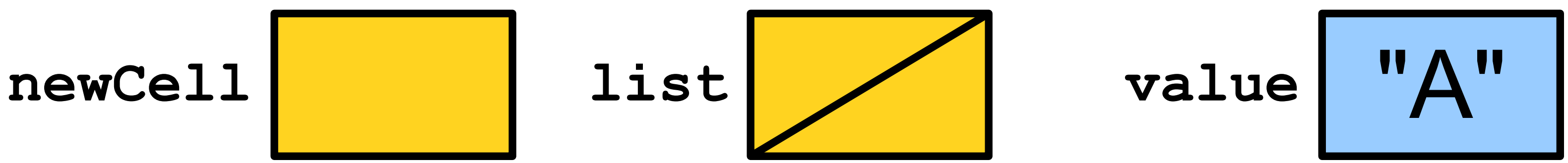
```
void listInsert(Cell* list, const string& value) {
    Cell* newCell = new Cell;
    newCell->value = value;
    newCell->next = list;
    list = newCell;
}
```

```
int main() {

}
```

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

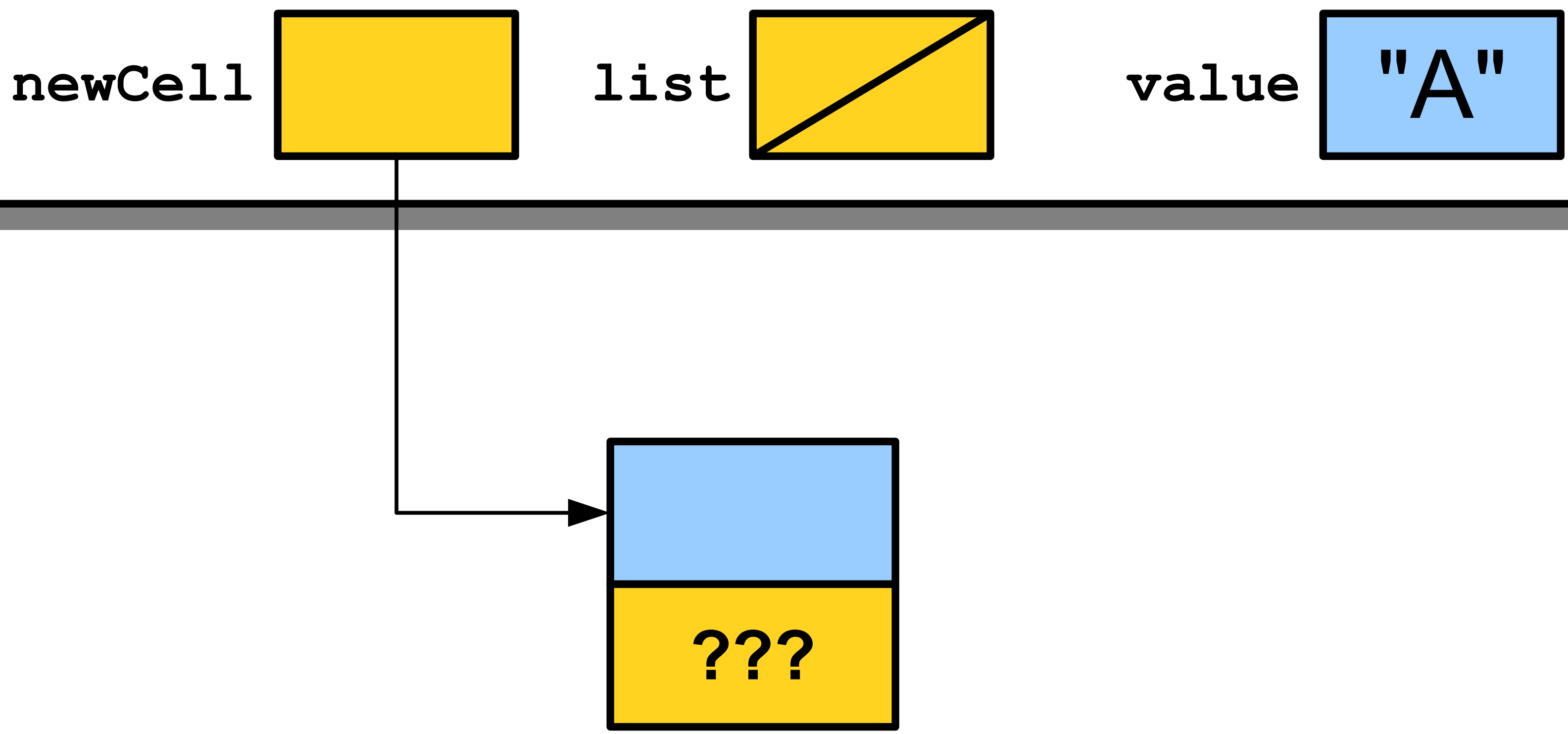
```
int main() {

}
```

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



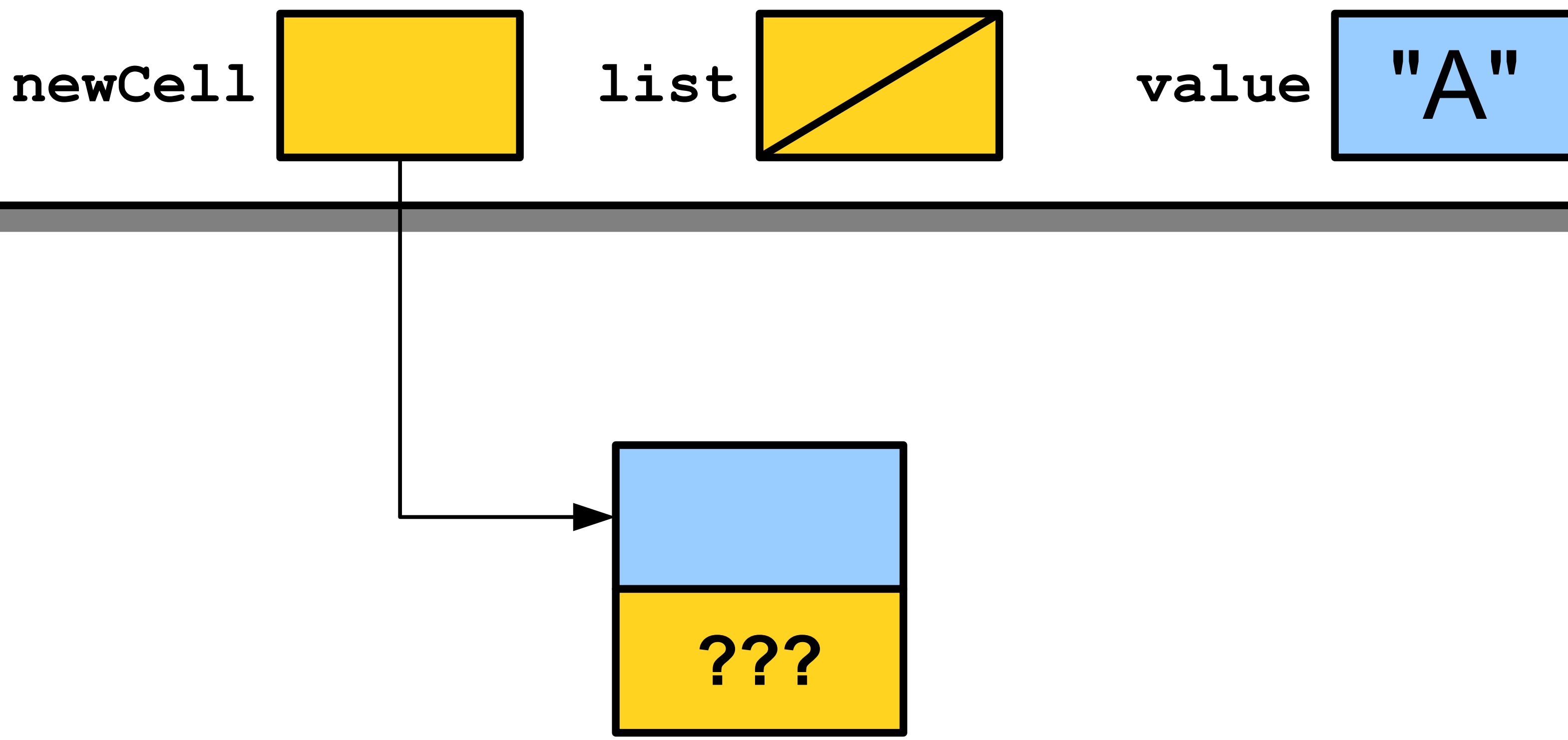
```
int main() {

}
```

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



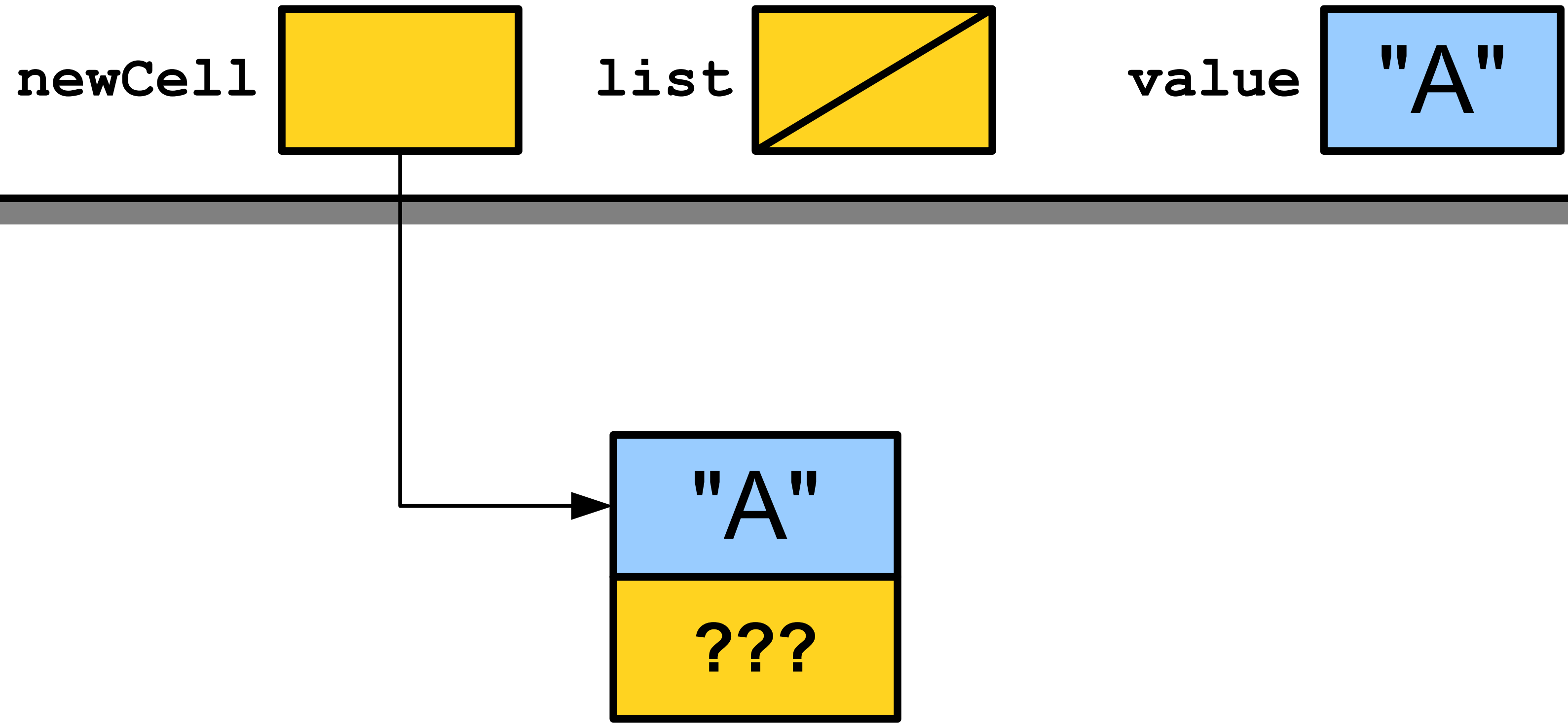
```
int main() {

}
```

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



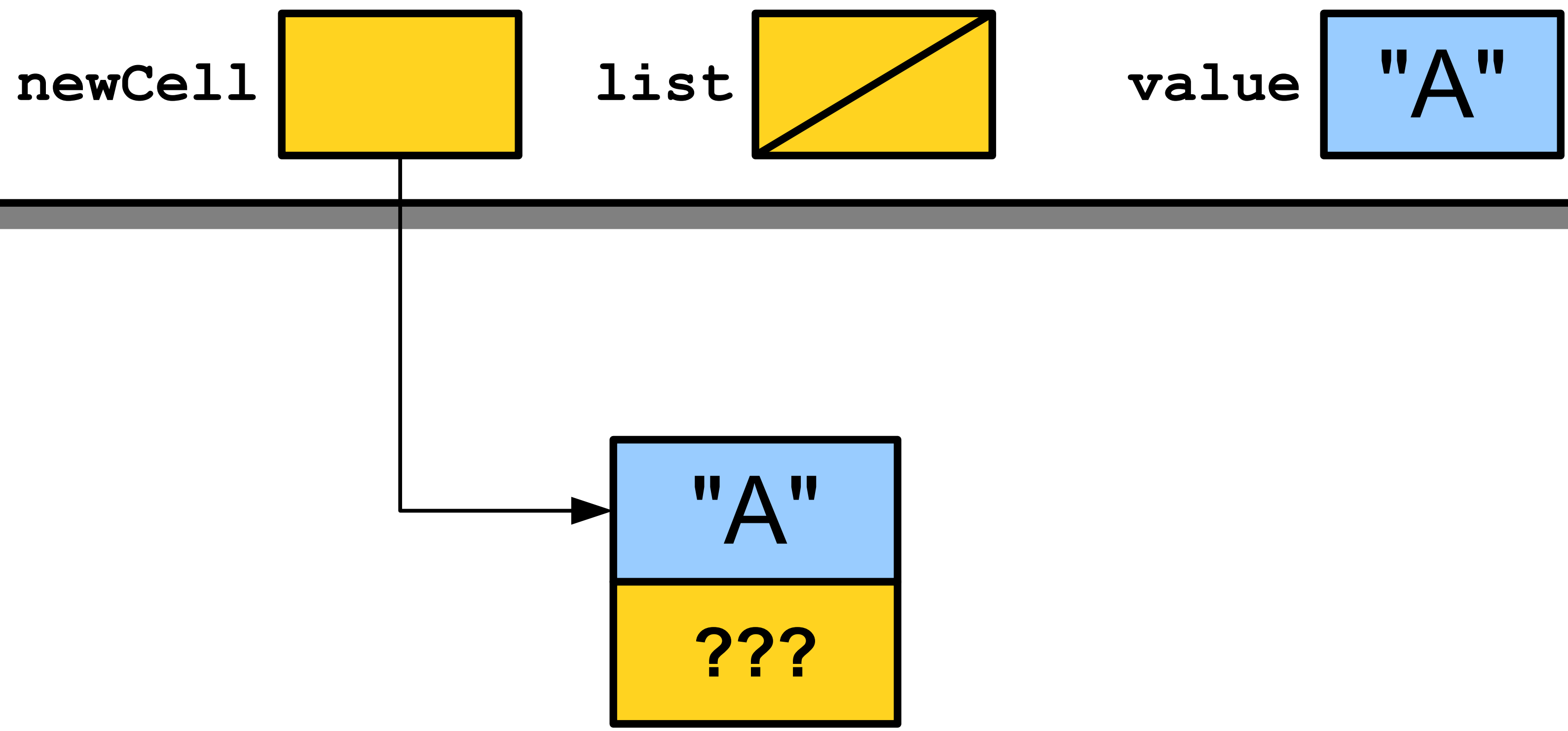
```
int main() {

}
```

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



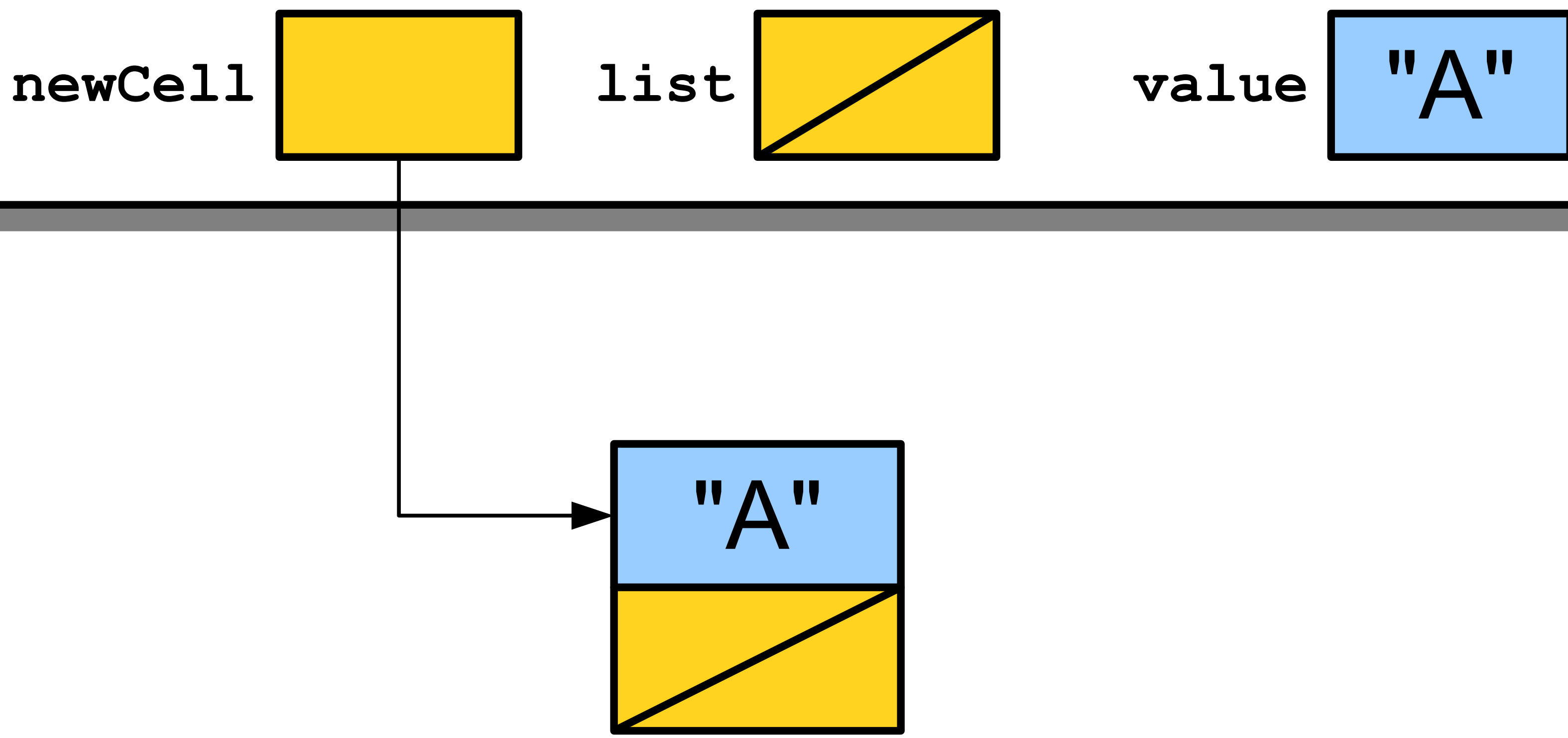
```
int main() {
}
```

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



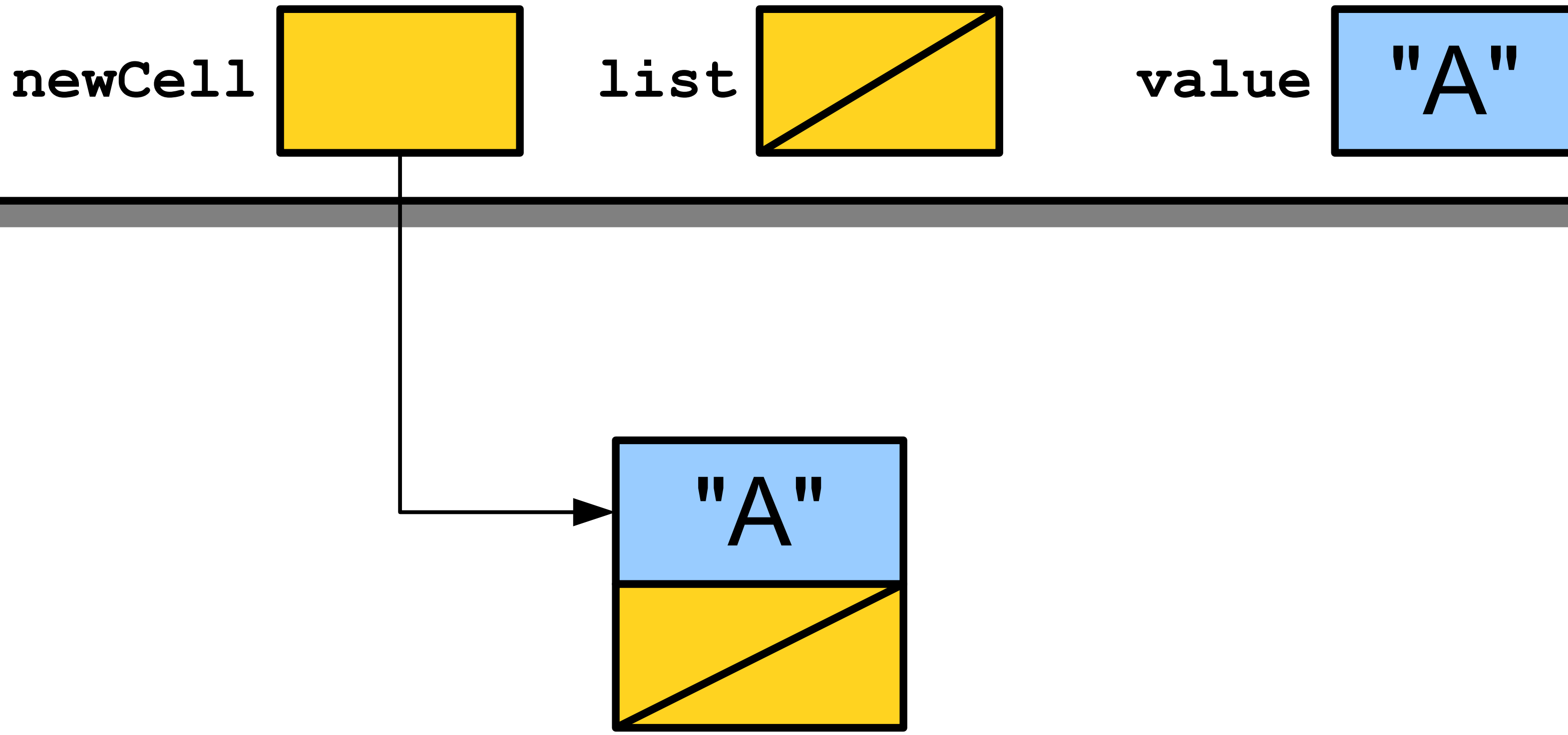
```
int main() {

}
```

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



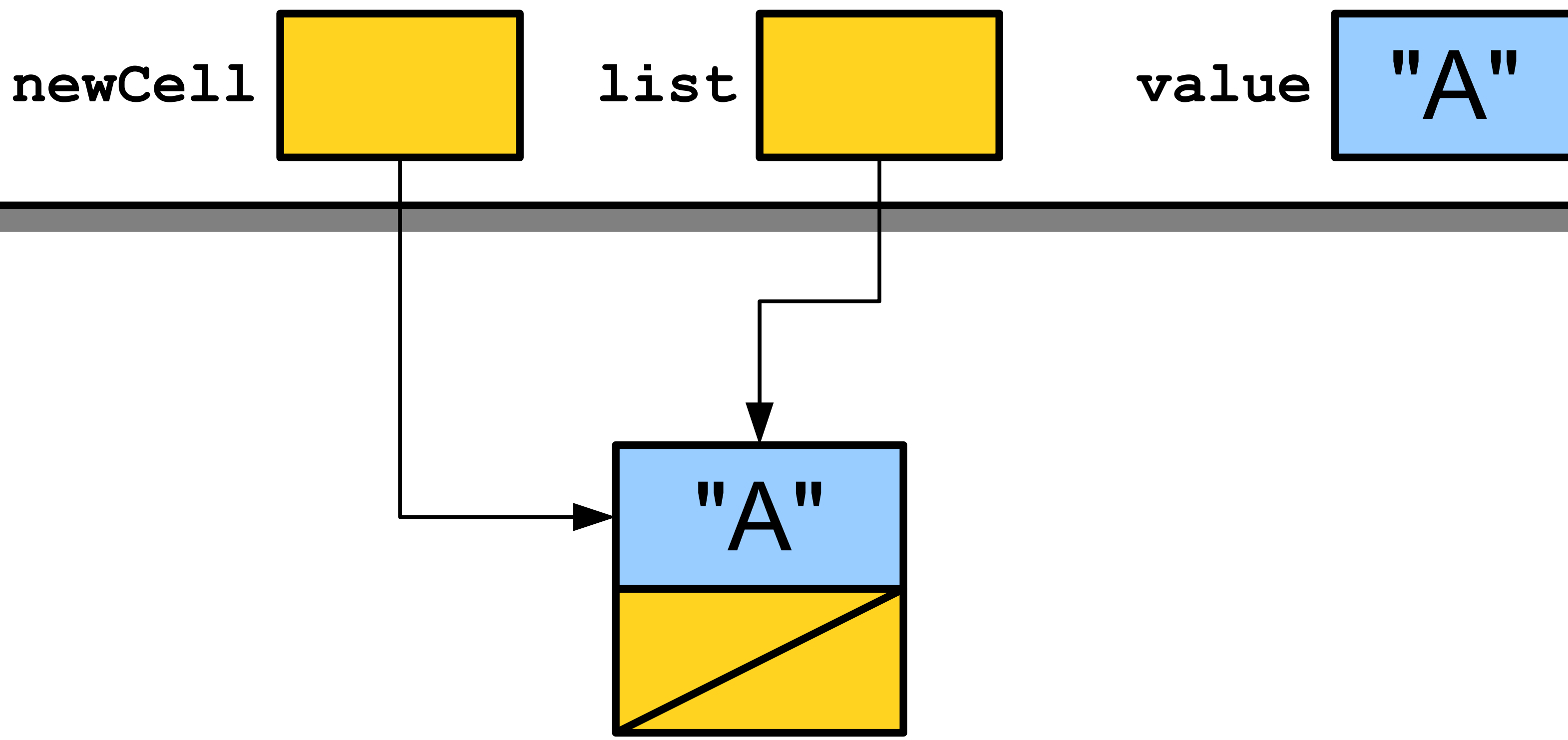
```
int main() {

}
```

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

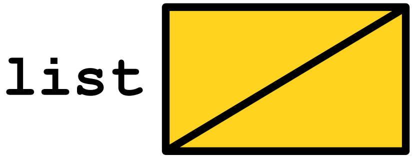


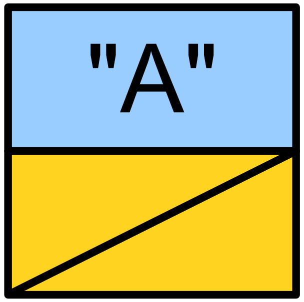
```
int main() {

}
```

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

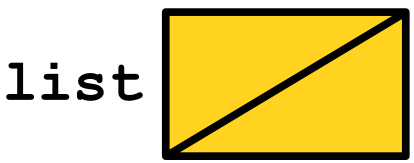


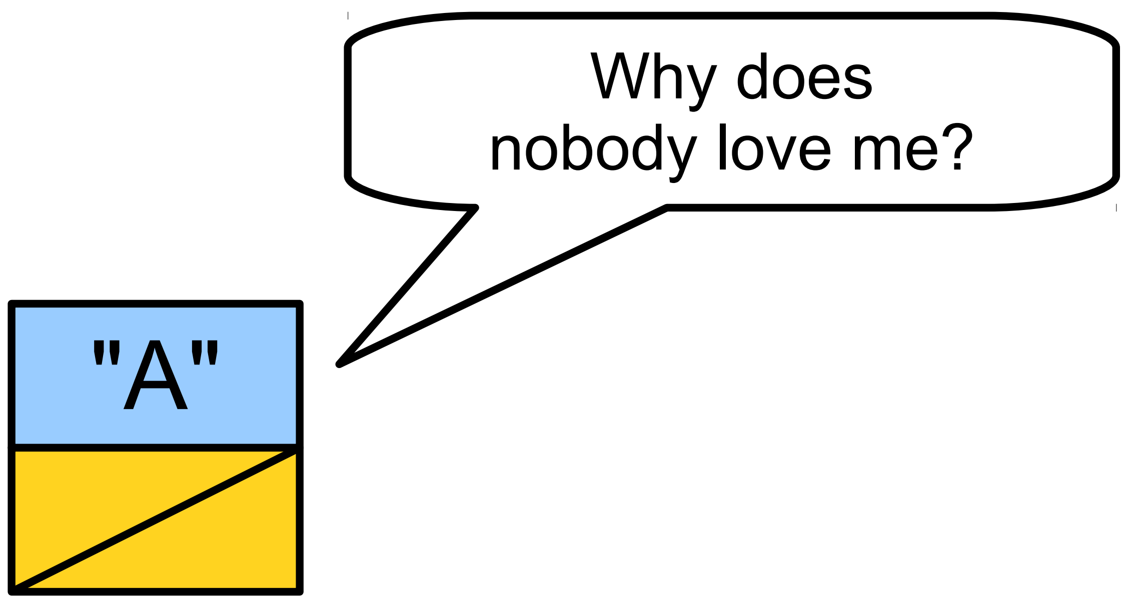
```
int main() {
    Cell* list = nullptr;
    listInsert(list, "A");
    listInsert(list, "B");
    listInsert(list, "C");

    return 0;
}
```





```
int main() {
    Cell* list = nullptr;
    listInsert(list, "A");
    listInsert(list, "B");
    listInsert(list, "C");

    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:

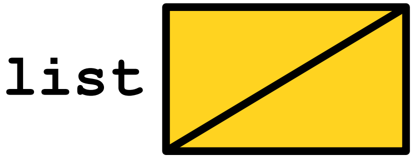
```
void listInsert(Cell*& list, const string& value) {
    Cell* newCell = new Cell;
    newCell->value = value;
    newCell->next = list;
    list = newCell;
}
```

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

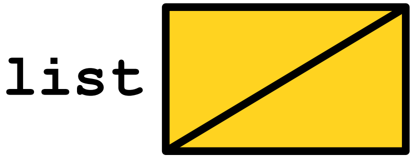
```
int main() {
    Cell* list = nullptr;
    listInsert(list, "A");
    listInsert(list, "B");
    listInsert(list, "C");

    return 0;
}
```

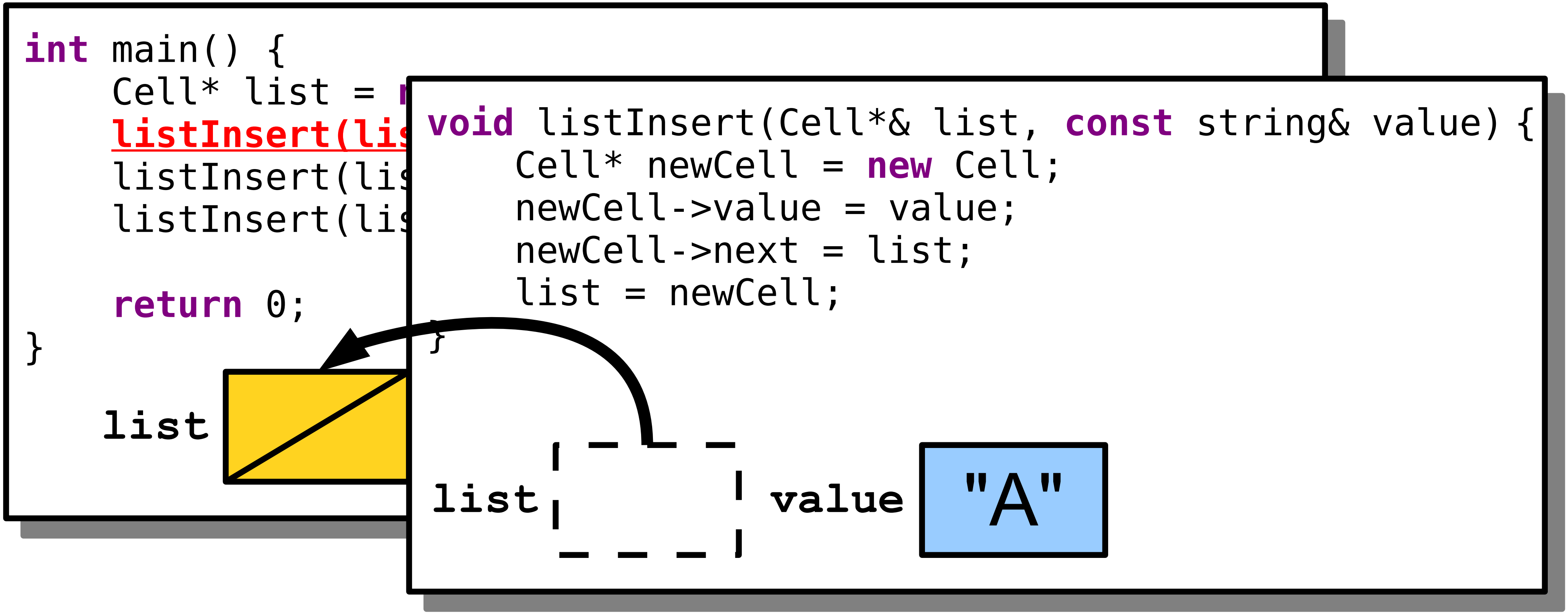
```
int main() {
    Cell* list = nullptr;
    listInsert(list, "A");
    listInsert(list, "B");
    listInsert(list, "C");

    return 0;
}
```

```
int main() {
    Cell* list = nullptr;
    listInsert(list, "A");
    listInsert(list, "B");
    listInsert(list, "C");

    return 0;
}
```



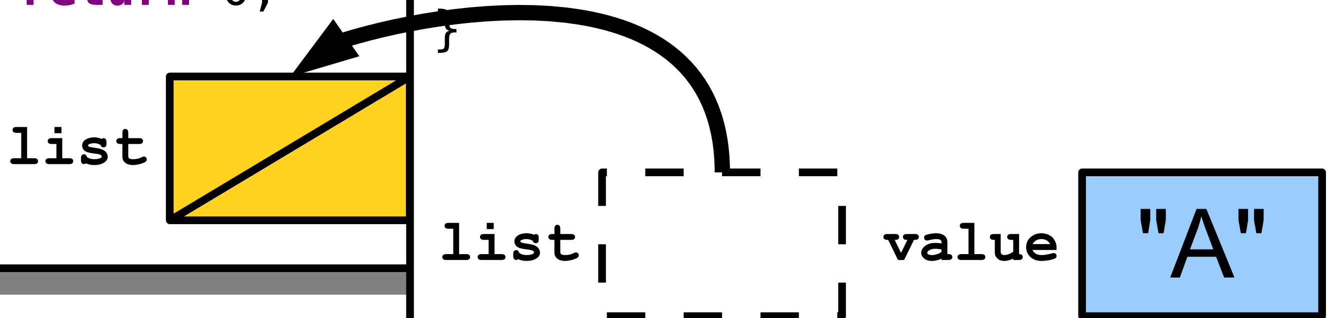
```
int main() {
    Cell* list = nullptr;
    listInsert(list, "A");
    listInsert(list, "B");
    listInsert(list, "C");

    return 0;
}
```

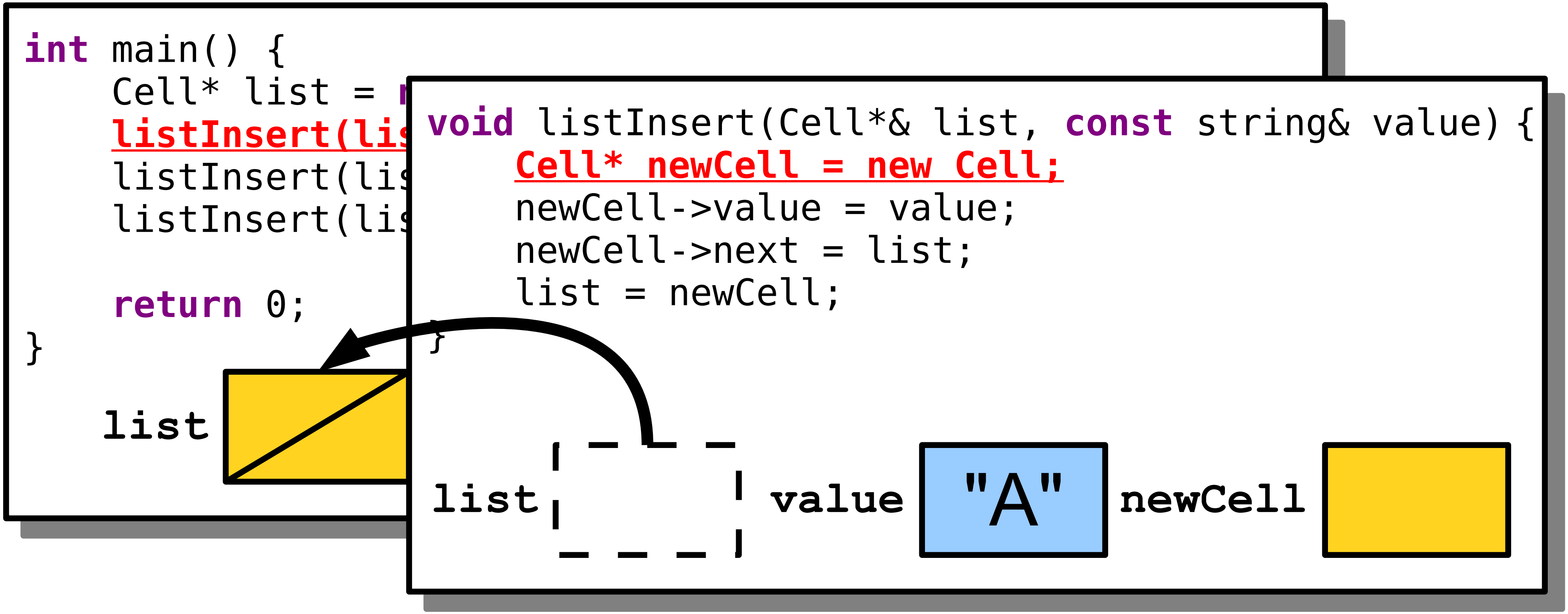



int main() {
Cell* list = n
listInsert(lis
listInsert(lis
listInsert(lis
return 0;
}
list
void listInsert(Cell*& list, const string& value) {
Cell* newCell = new Cell;
newCell->value = value;
newCell->next = list;
list = newCell;
}
list
value
"A"

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

    return 0;
}
```

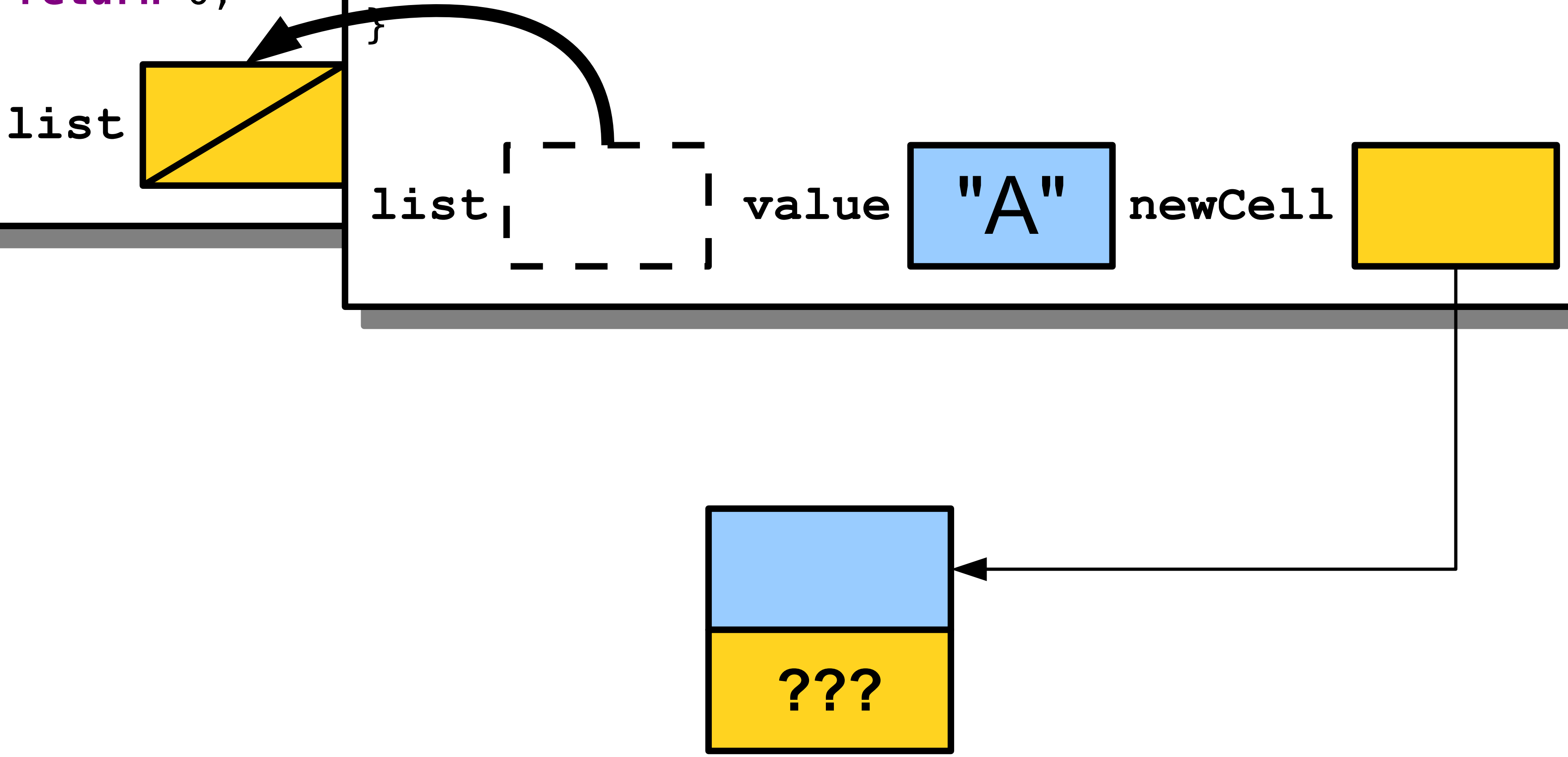
```
void listInsert(Cell*& list, const string& value) {
    Cell* newCell = new Cell;
    newCell->value = value;
    newCell->next = list;
    list = newCell;
}
```

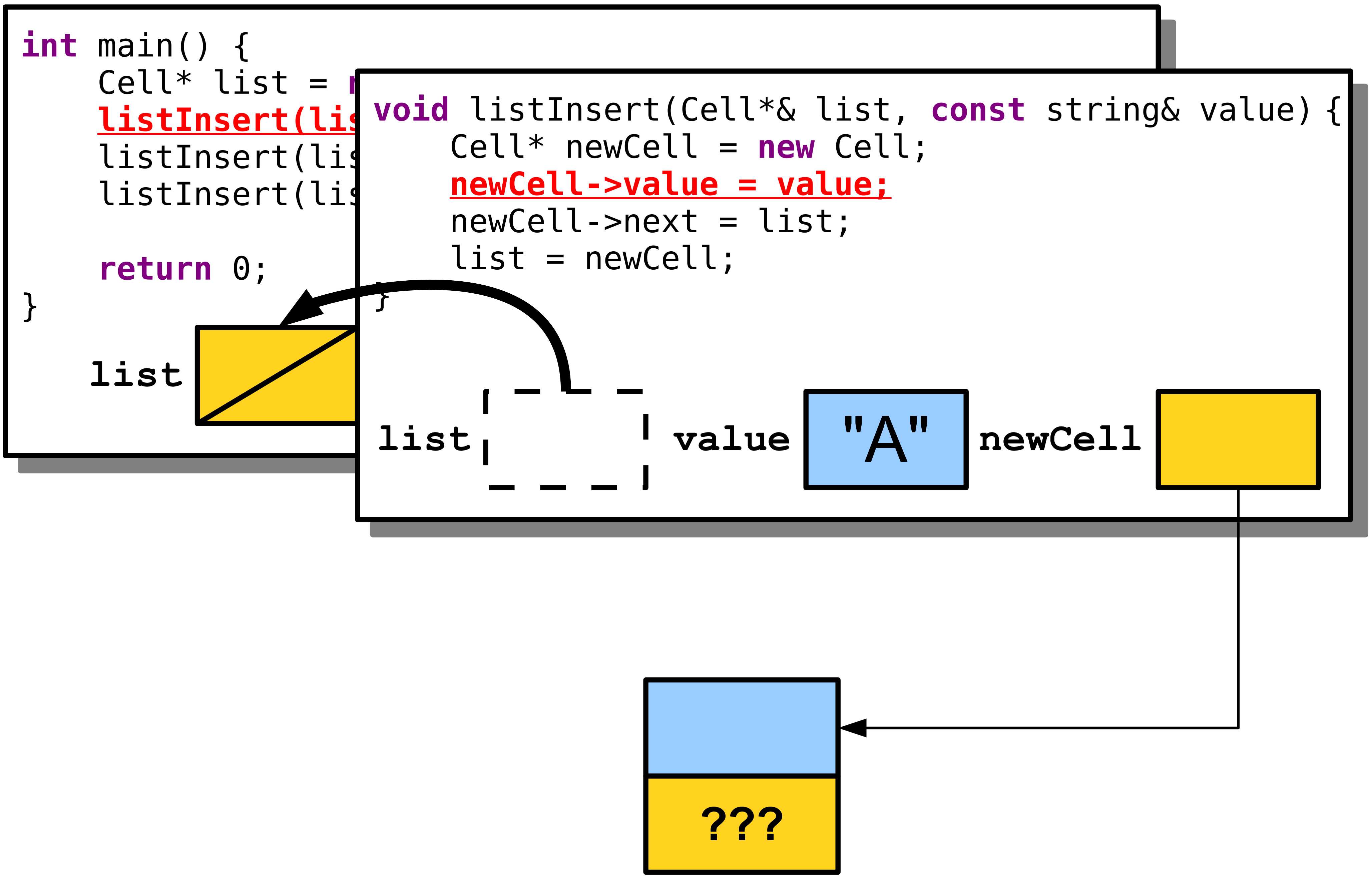


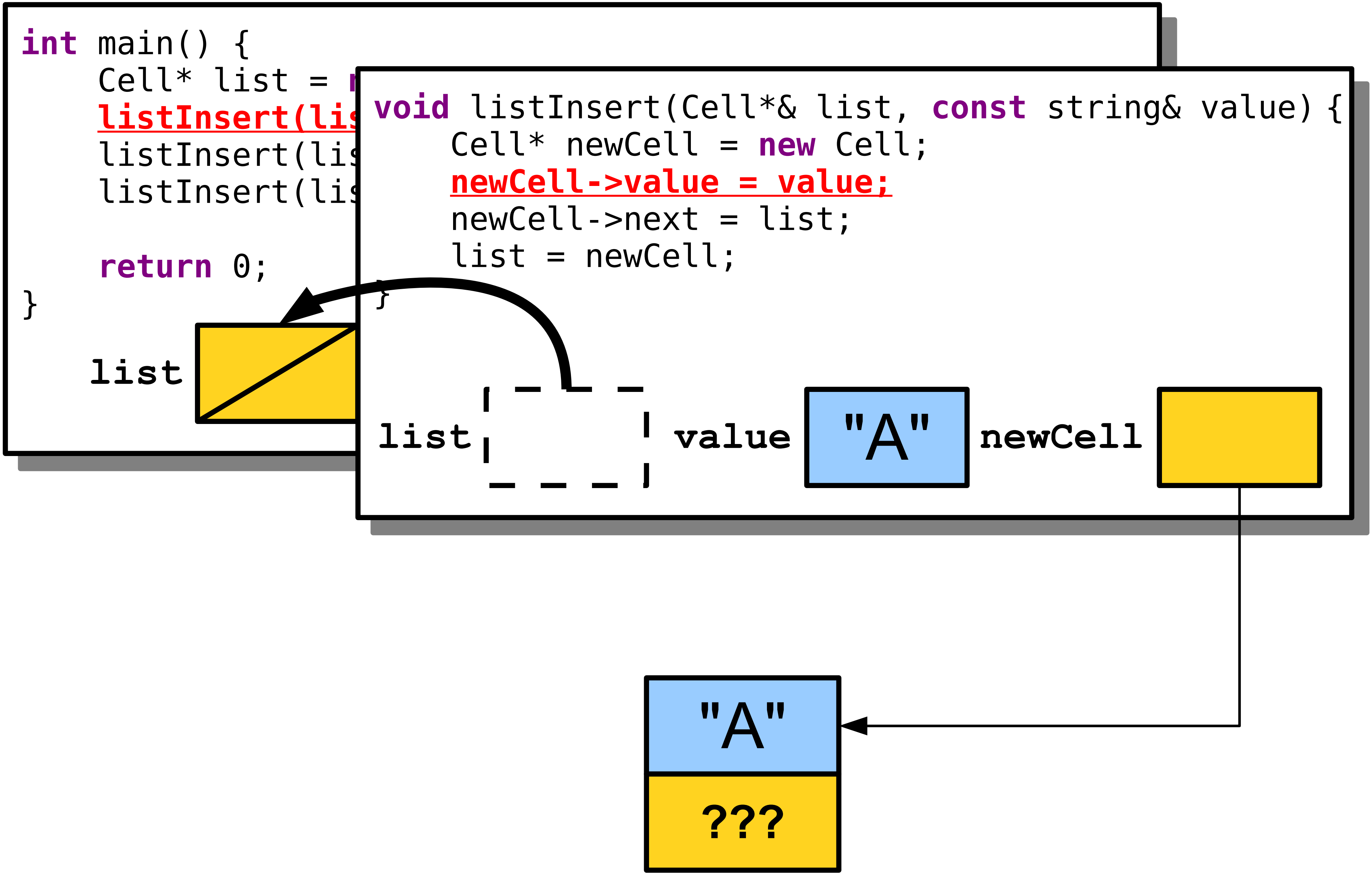

int main() {
Cell* list = n
listInsert(lis
listInsert(lis
listInsert(lis
return 0;
}
list
void listInsert(Cell*& list, const string& value) {
Cell* newCell = new Cell;
newCell->value = value;
newCell->next = list;
list = newCell;
}
list
value
"A"
newCell

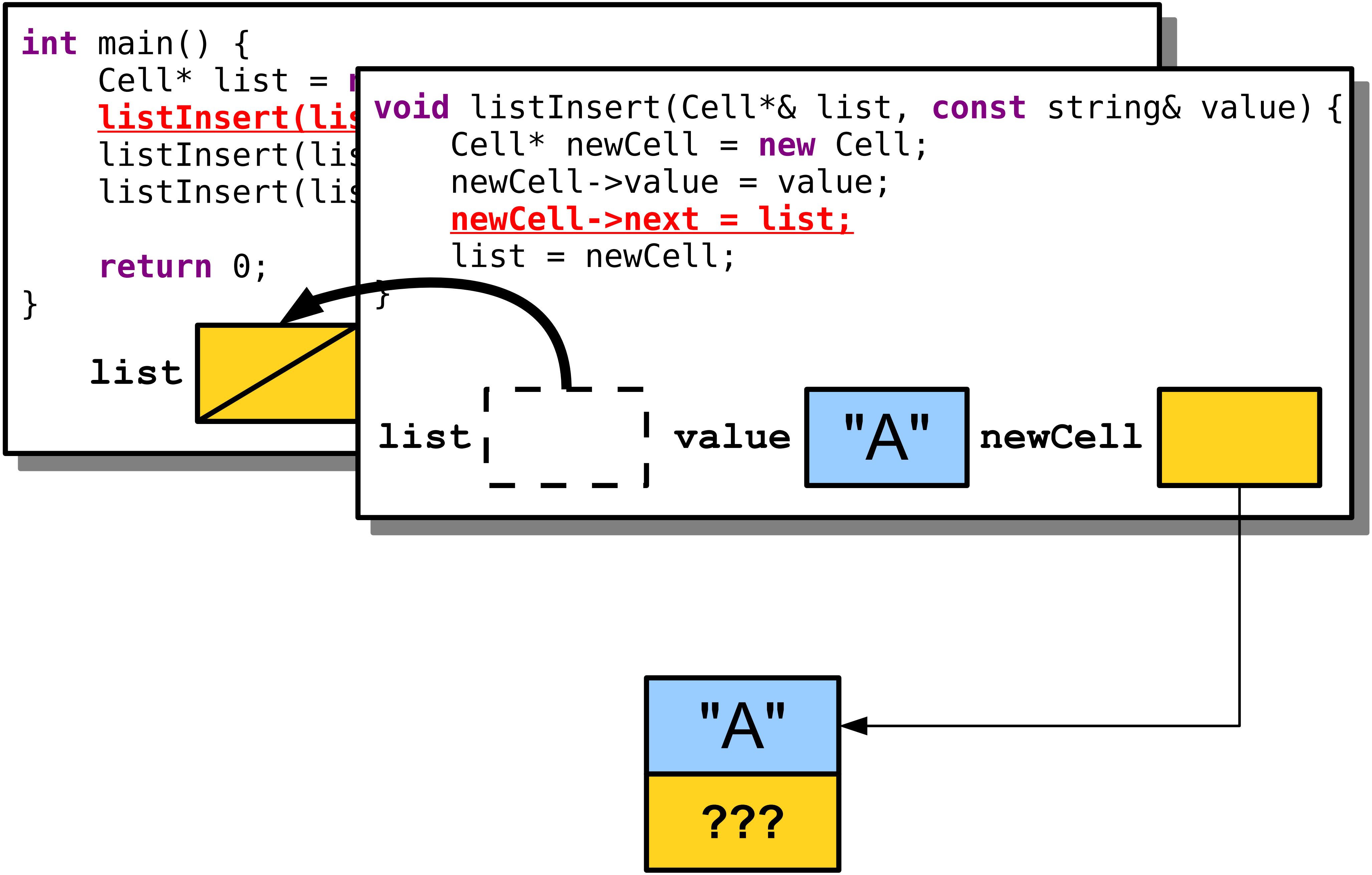
```
int main() {
    Cell* list = n
    listInsert(lis
    listInsert(lis
    listInsert(lis

    return 0;
}
```

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

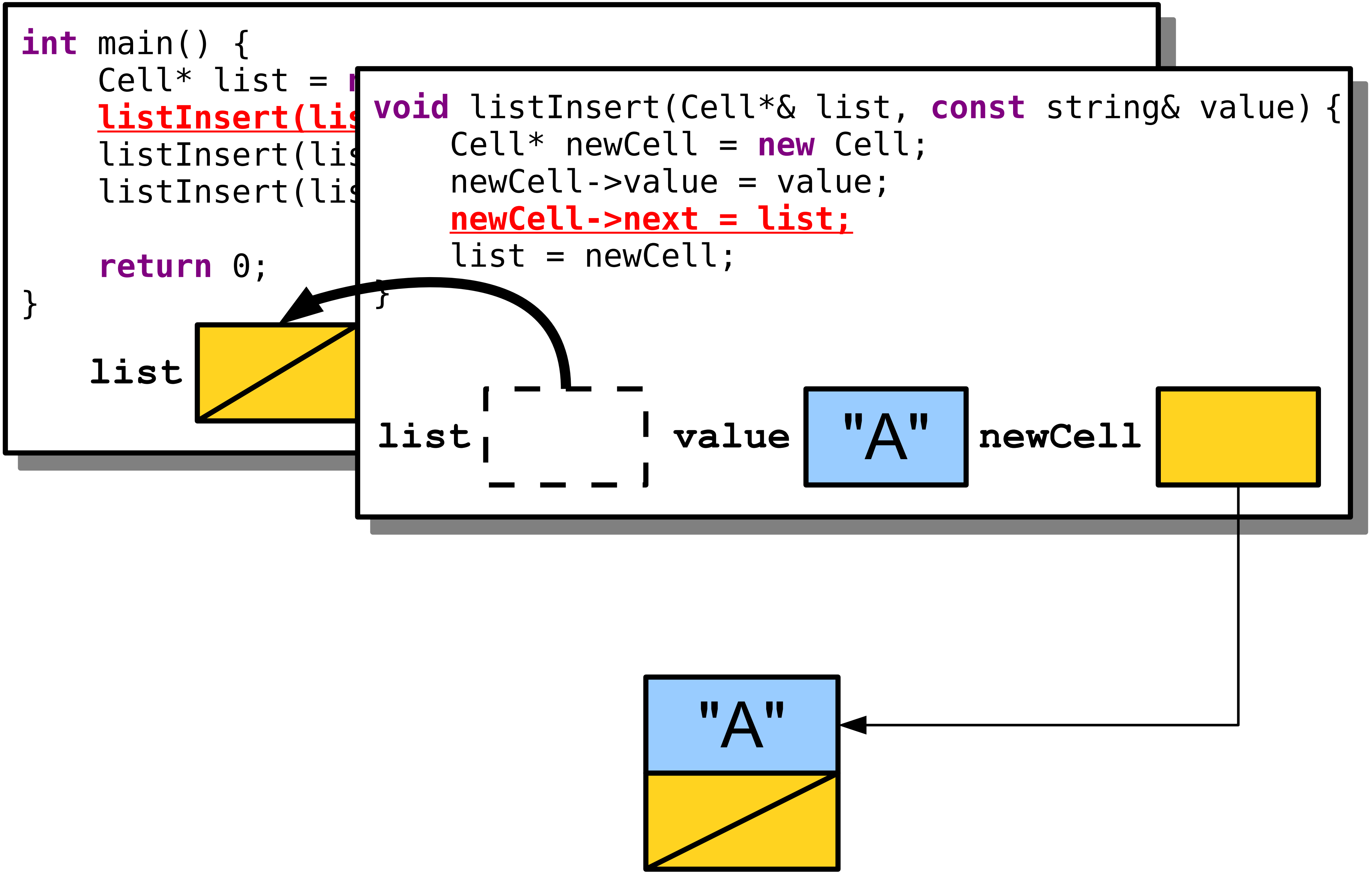


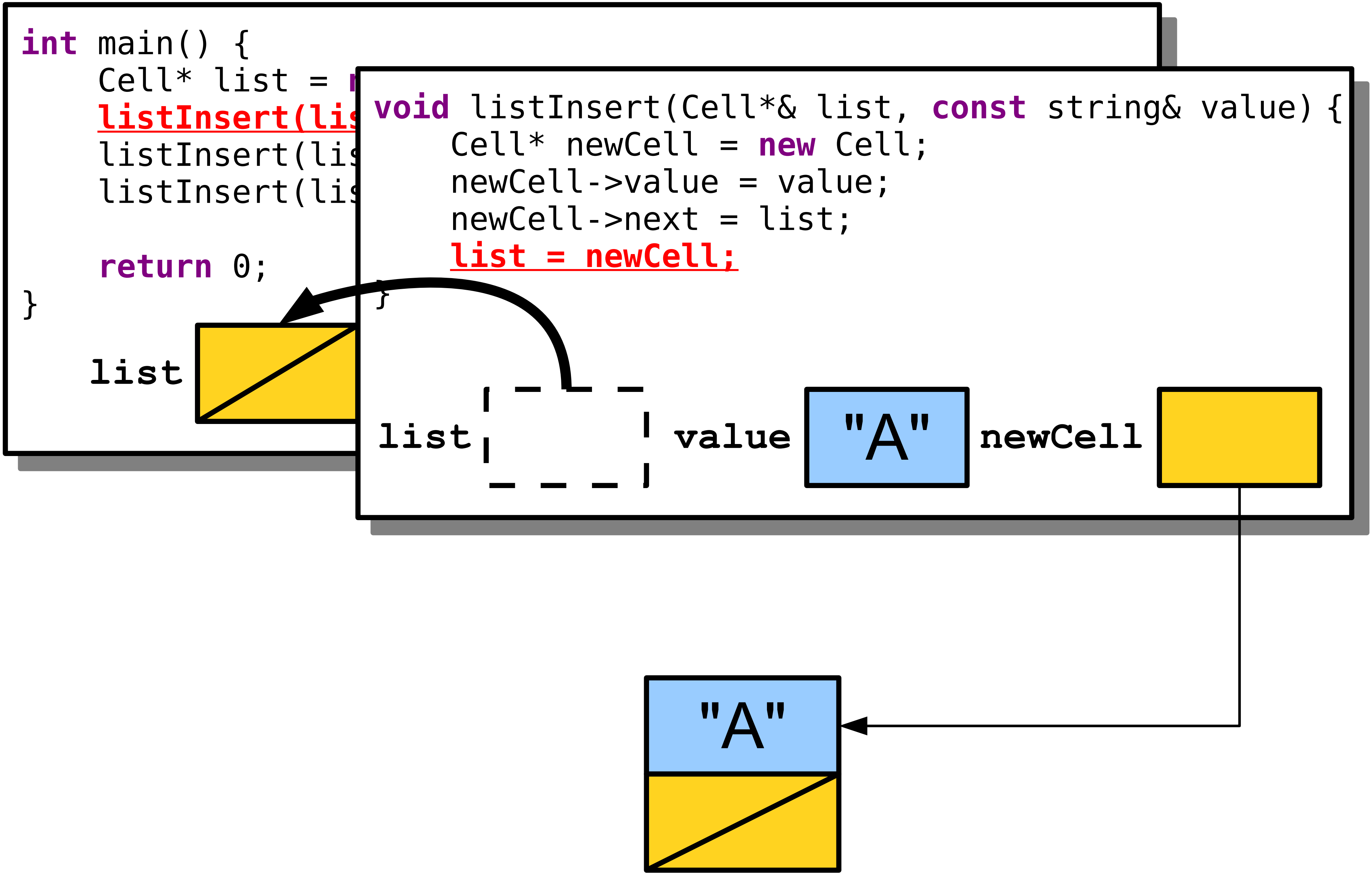

int main() {
Cell* list = n
listInsert(lis
listInsert(lis
listInsert(lis
return 0;
}
list
void listInsert(Cell*& list, const string& value) {
Cell* newCell = new Cell;
newCell->value = value;
newCell->next = list;
list = newCell;
}
list
value
"A"
newCell
???


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
"A"
newCell
"A"
???

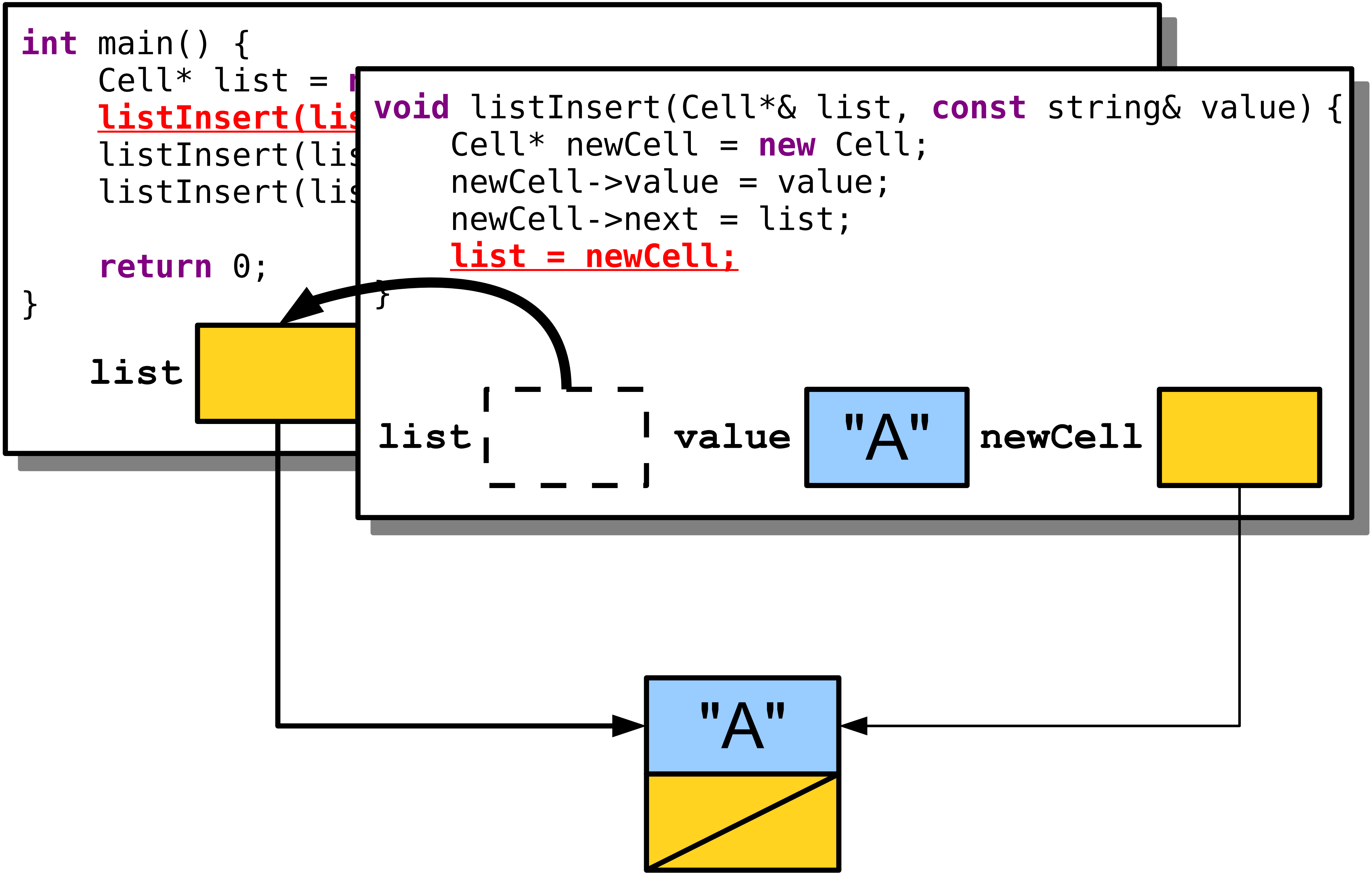


int main() {
Cell* list = n
listInsert(lis
listInsert(lis
listInsert(lis
return 0;
}
list
void listInsert(Cell*& list, const string& value) {
Cell* newCell = new Cell;
newCell->value = value;
newCell->next = list;
list = newCell;
}
list
value
"A"
newCell
"A"
???

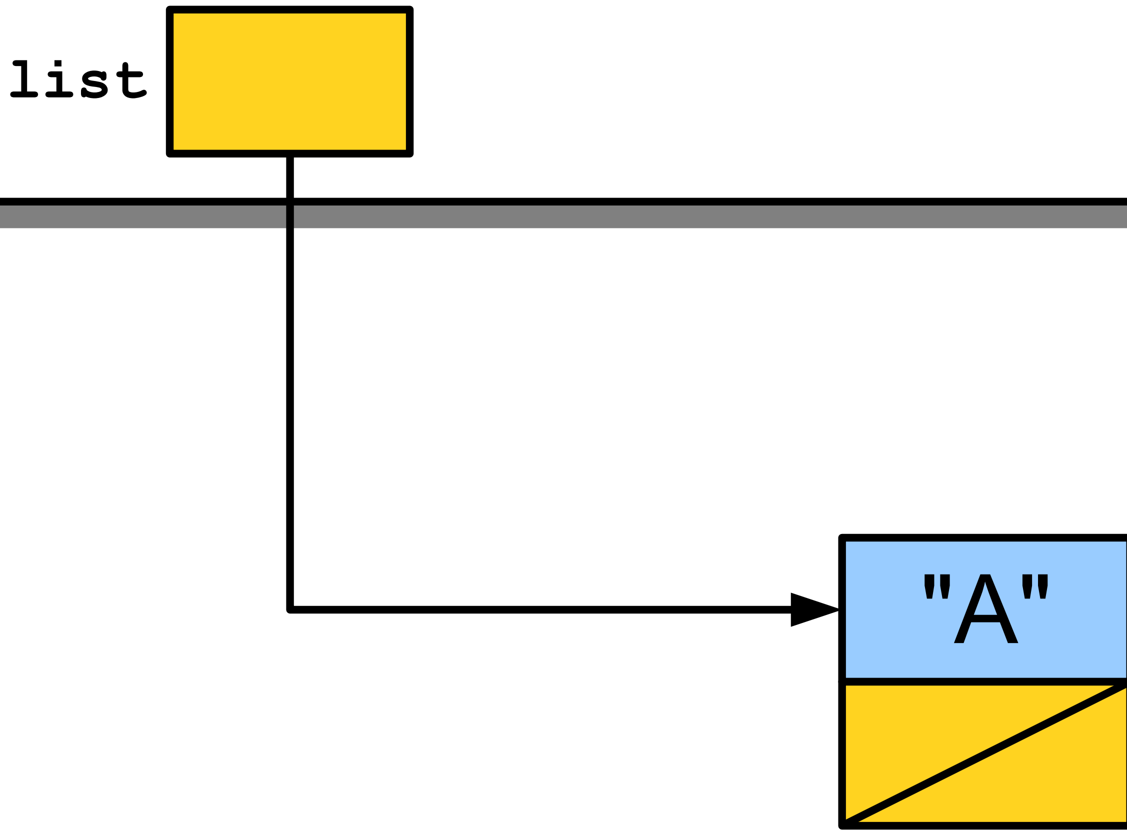

int main() {
Cell* list = n
listInsert(lis
listInsert(lis
listInsert(lis
return 0;
}
list
void listInsert(Cell*& list, const string& value) {
Cell* newCell = new Cell;
newCell->value = value;
newCell->next = list;
list = newCell;
}
list
value
"A"
newCell
"A"



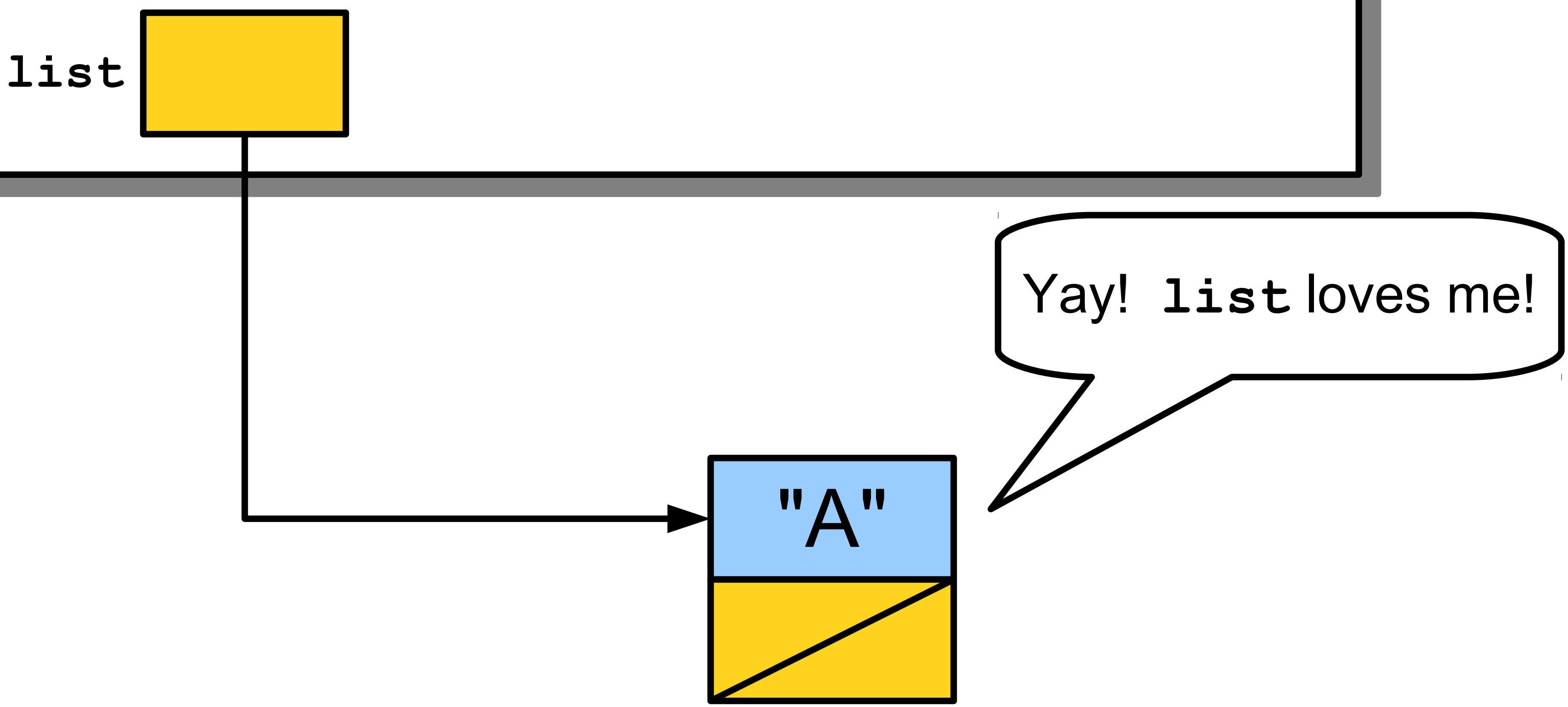
int main() {
Cell* list = n
listInsert(lis
listInsert(lis
listInsert(lis
return 0;
}
list
void listInsert(Cell*& list, const string& value) {
Cell* newCell = new Cell;
newCell->value = value;
newCell->next = list;
list = newCell;
}
list
value
"A"
newCell
"A"


int main() {
Cell* list =
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
"A"
newCell
"A"

```
int main() {
    Cell* list = nullptr;
    listInsert(list, "A");
    listInsert(list, "B");
    listInsert(list, "C");

    return 0;
}
```



```
int main() {
    Cell* list = nullptr;
    listInsert(list, "A");
    listInsert(list, "B");
    listInsert(list, "C");

    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!

Assignment 5

- Assignment 5 (***Priority Queue***) goes out today. It's due next Friday at the start of class.
- It's a four-parter, and we've included a timetable on the front of the assignment.
  - ***Start this assignment as soon as you get it!*** You'll have plenty of time to finish everything, but not if you put it off to the last minute.
- Working in pairs is permitted – and encouraged! – on this assignment.
- Anton will be holding YEAH hours tomorrow evening. We'll announce the time and location on Piazza and over email.

Stanford Women
in Computer Science

CASUAL CS DINNER

Monday, February 27 from 6-7 PM at the WCC
RSVP link **here!**

**Come have dinner with CS students and faculty.
Everyone is welcome, especially students
just starting out in CS!**

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!

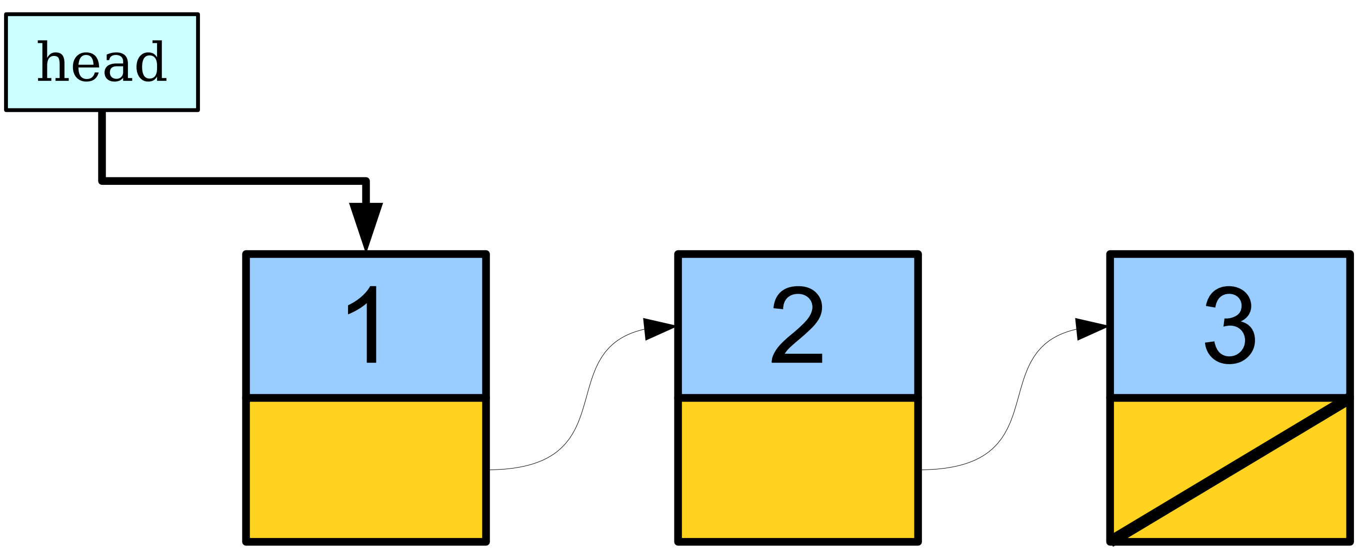
Back to Linked Lists!

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.

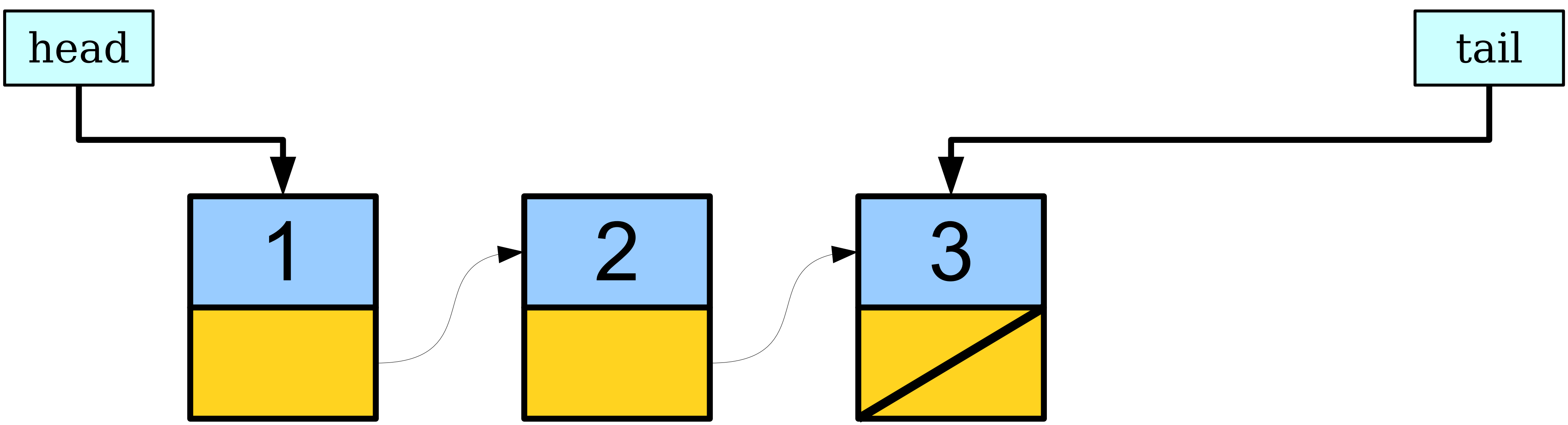
Tail Pointers

- 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.



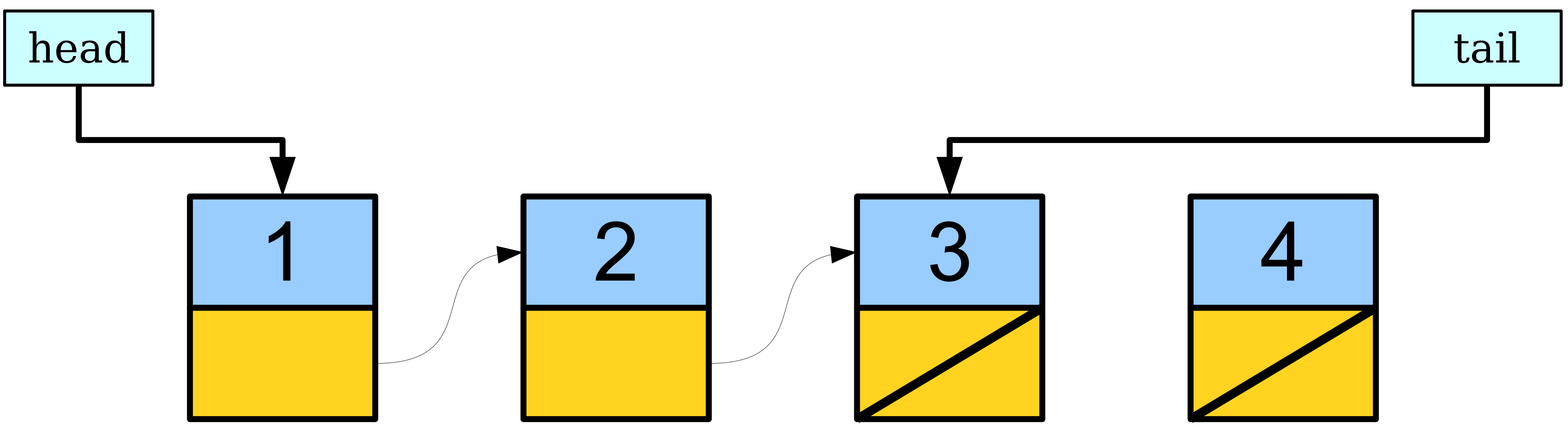
Tail Pointers

- 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.



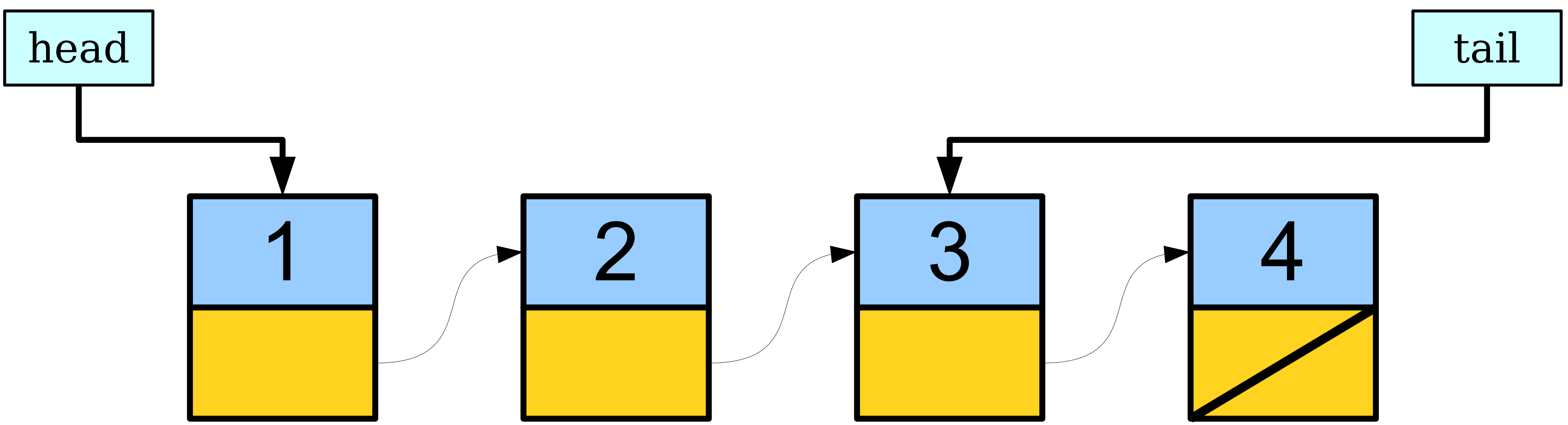
Tail Pointers

- 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.



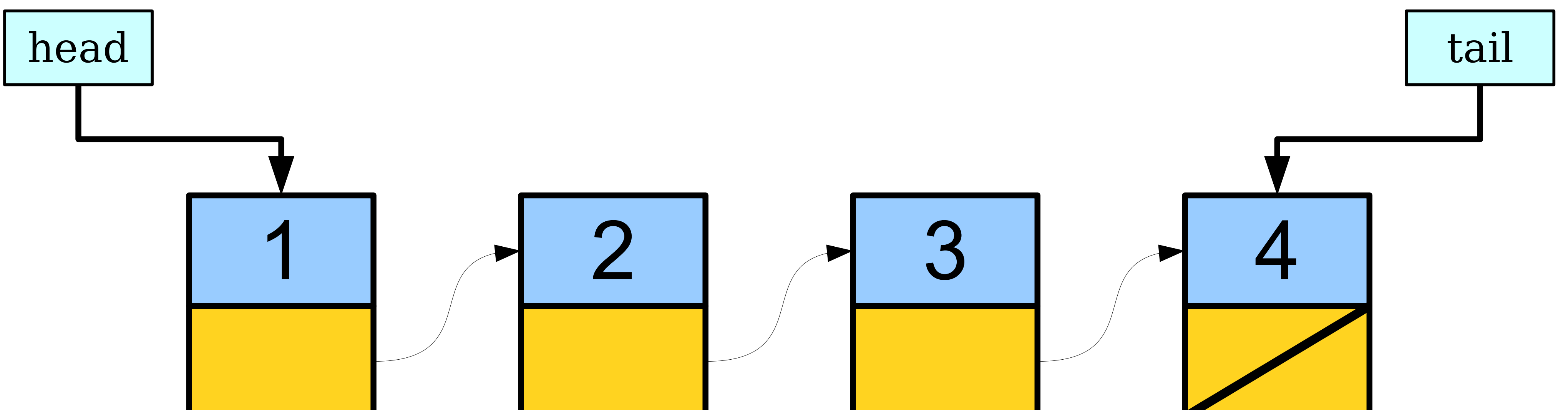
Tail Pointers

- 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.



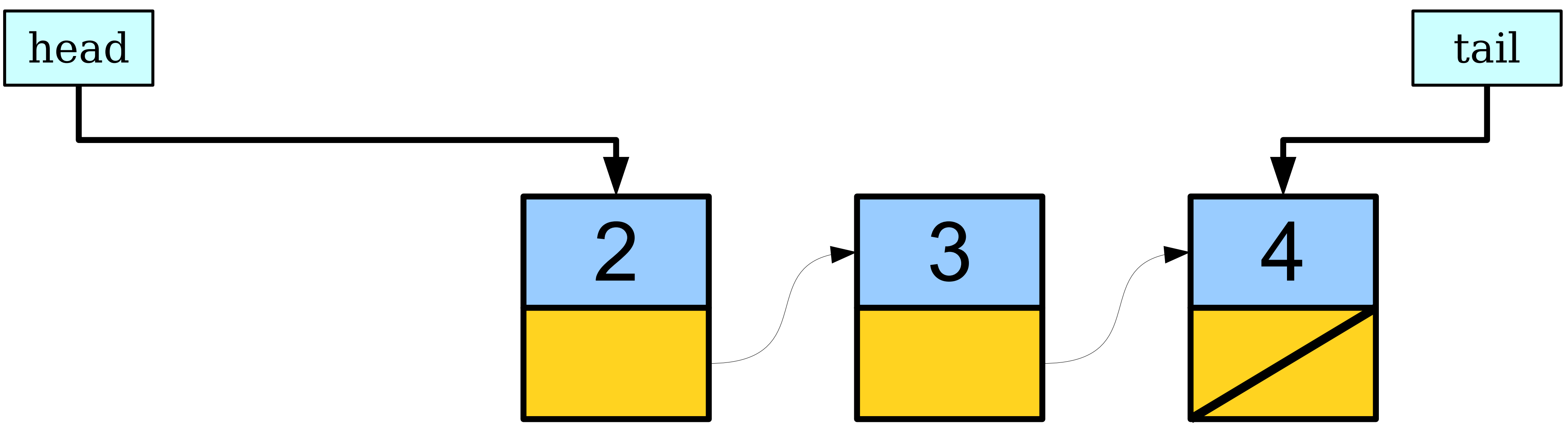
Tail Pointers

- 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.



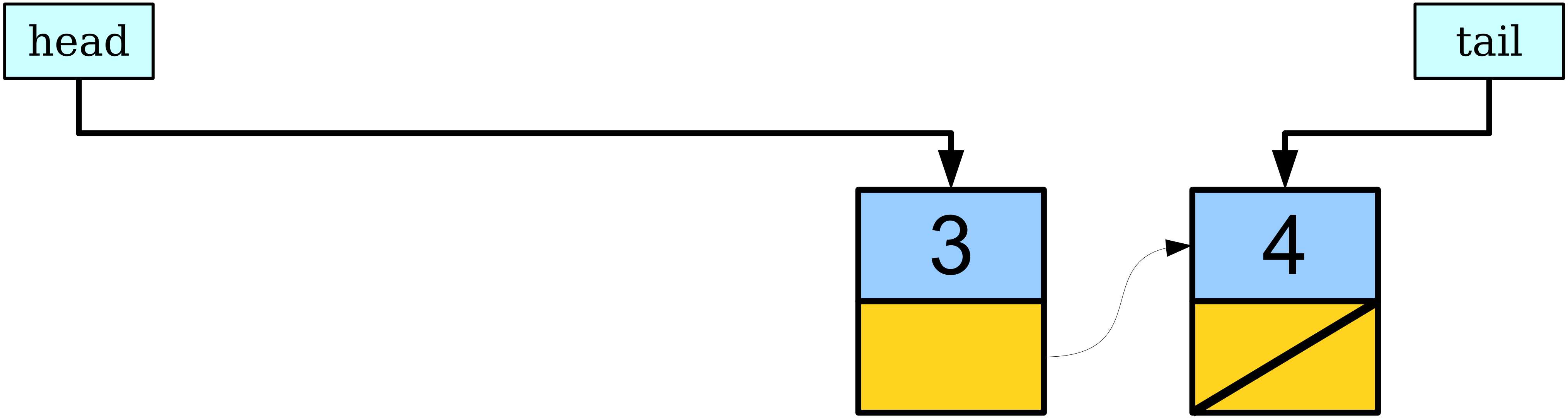
Tail Pointers

- 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.



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.

head

tail

Enqueuing Things

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

Enqueuing Things

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

head

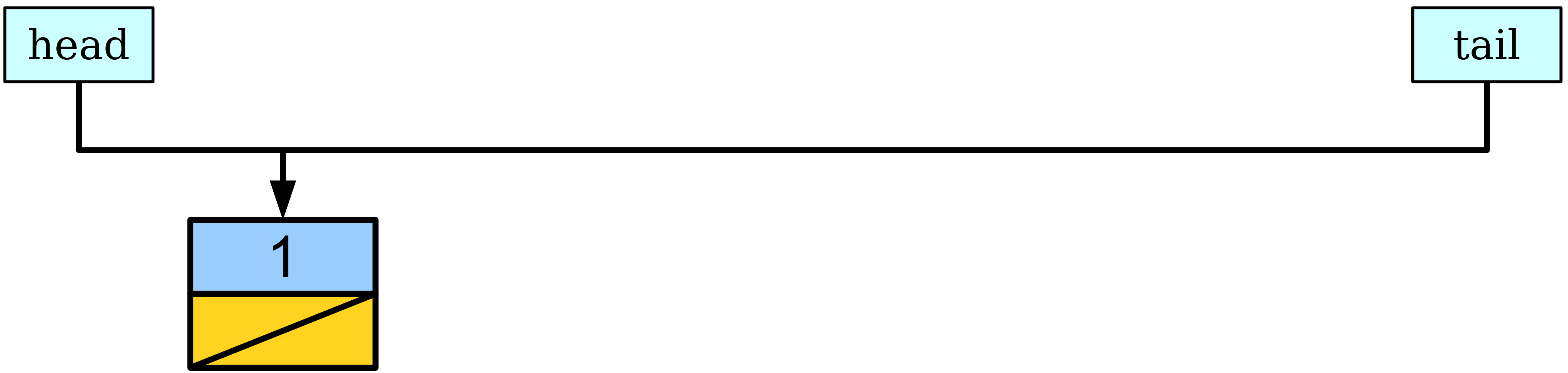
tail

Enqueuing Things

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

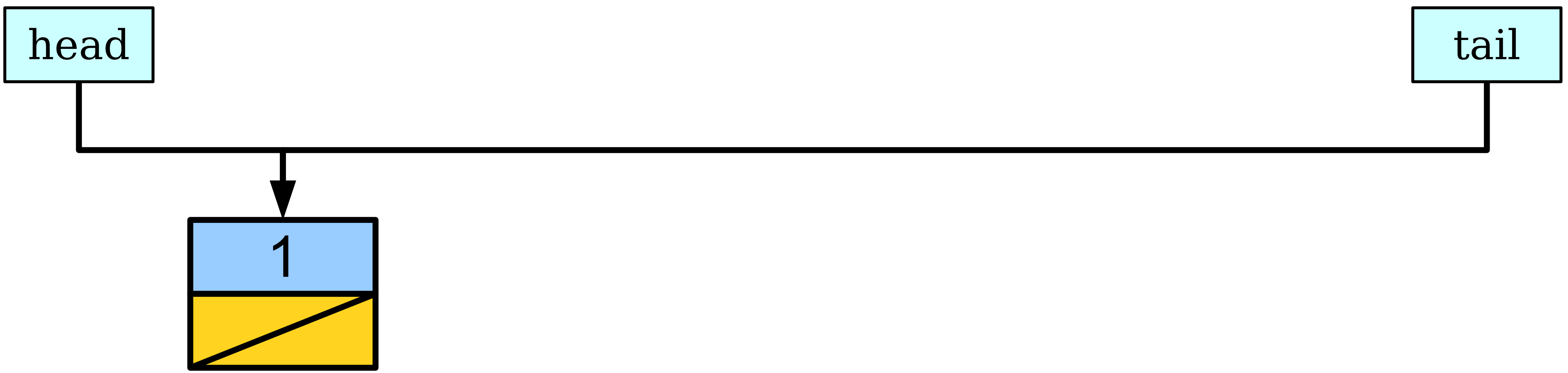
Enqueuing Things

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

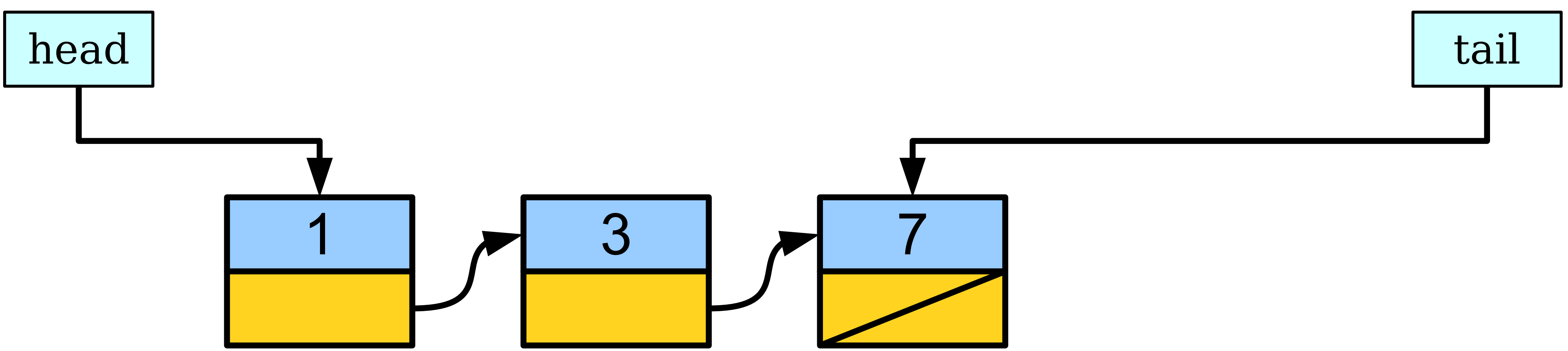


Enqueuing Things

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

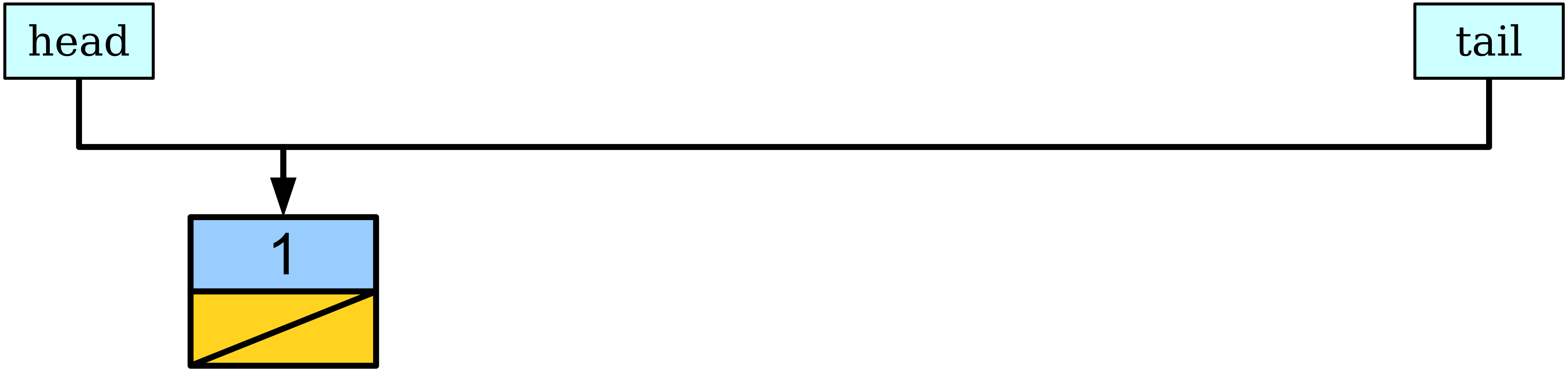


- ***Case 2:*** The queue is not empty.

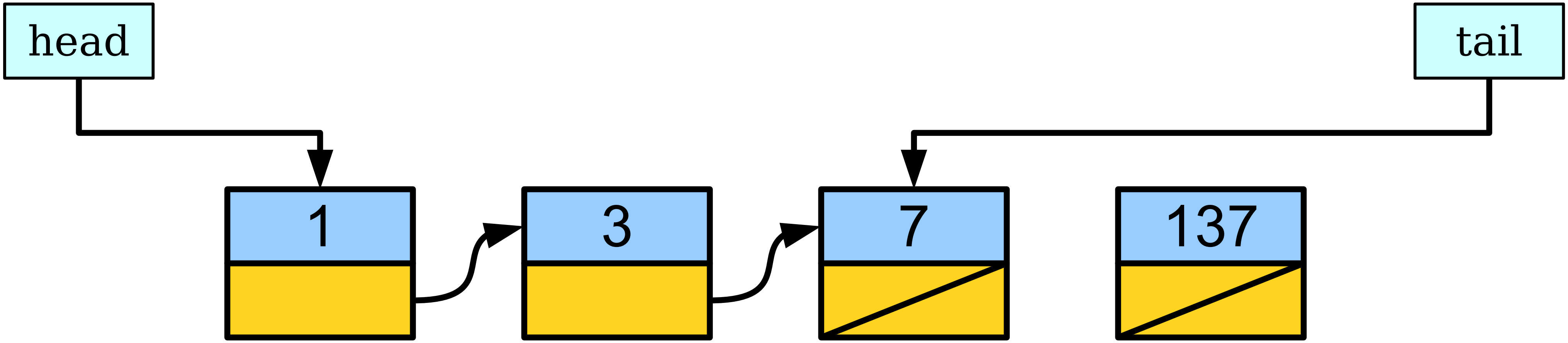


Enqueuing Things

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

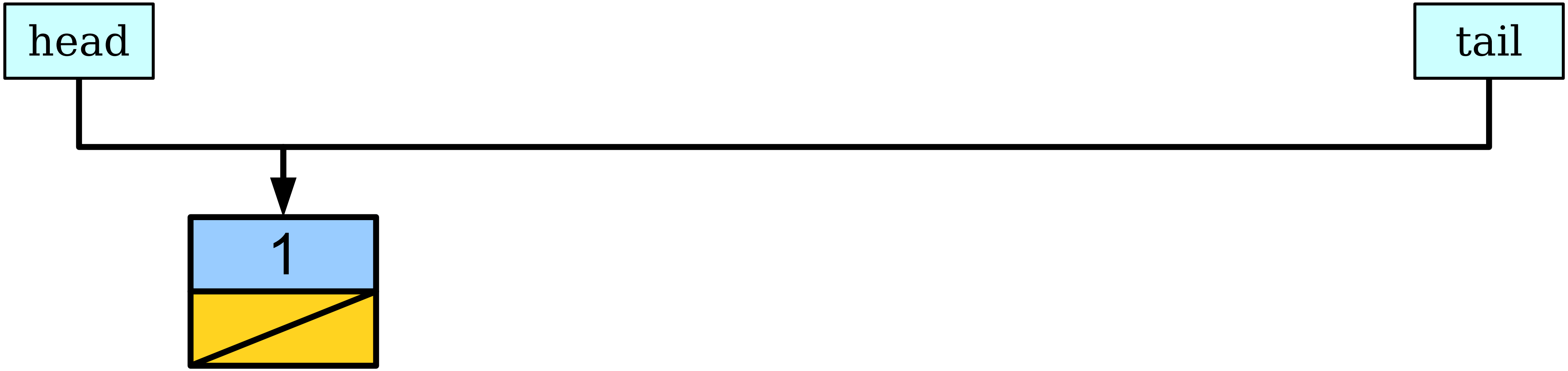


- ***Case 2:*** The queue is not empty.

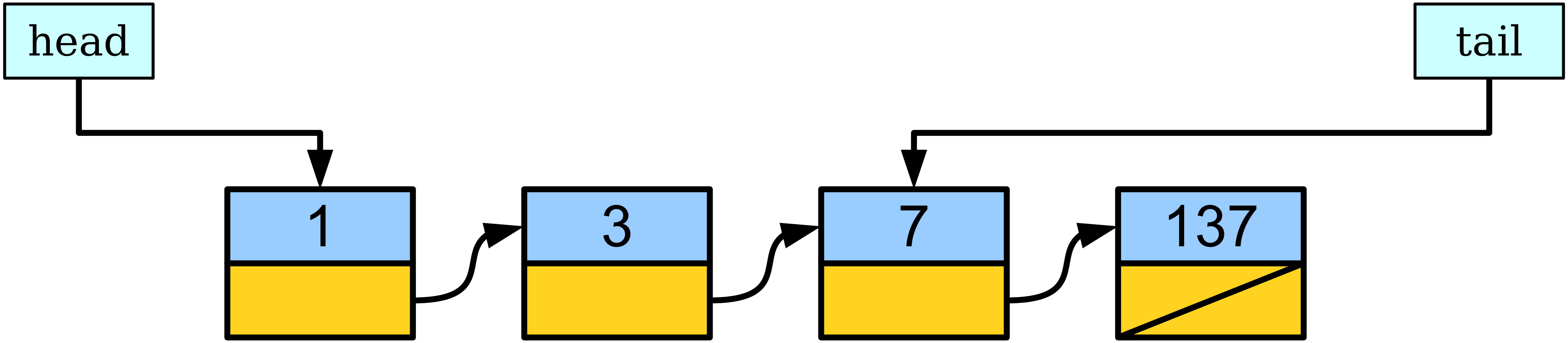


Enqueuing Things

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

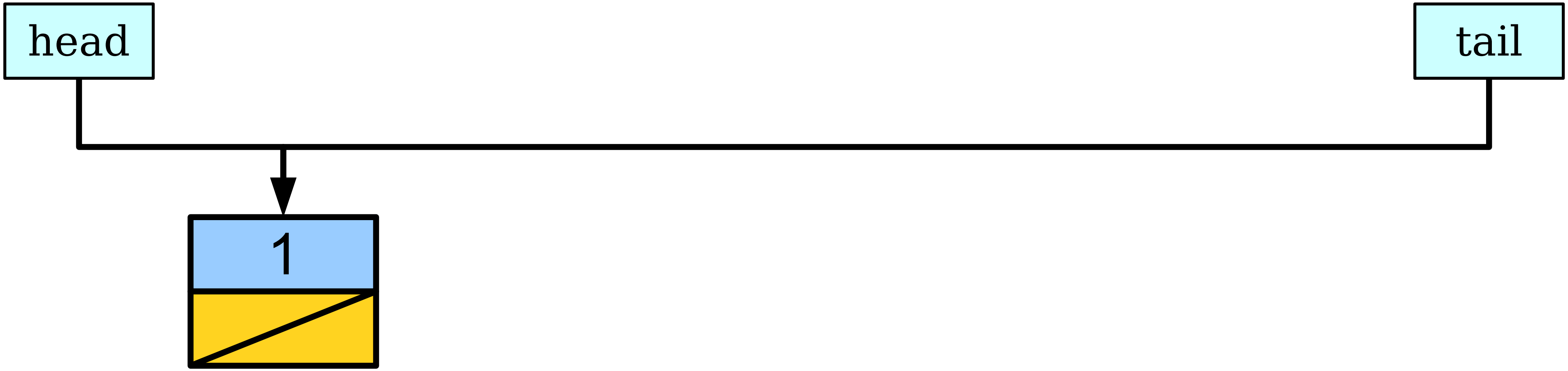


- ***Case 2:*** The queue is not empty.

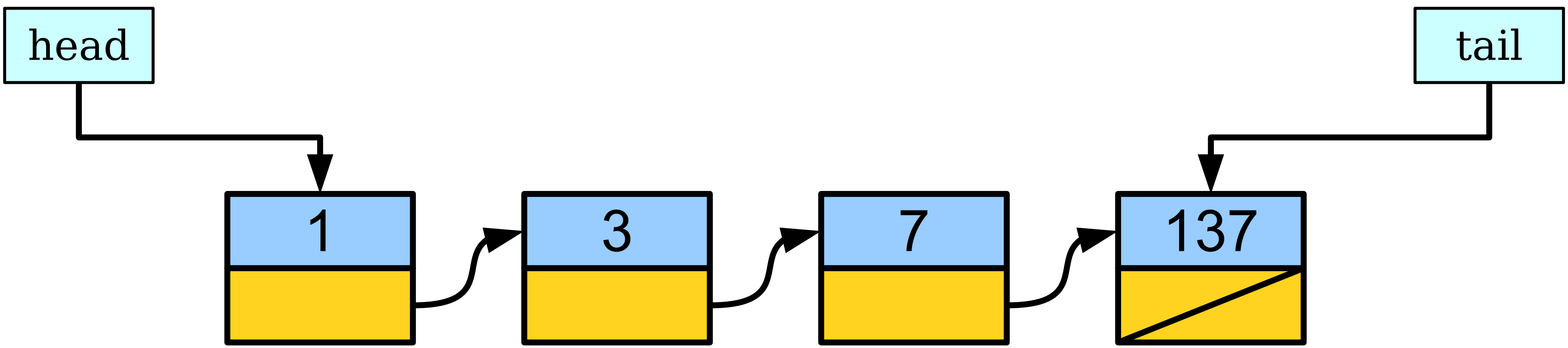


Enqueuing Things

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



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

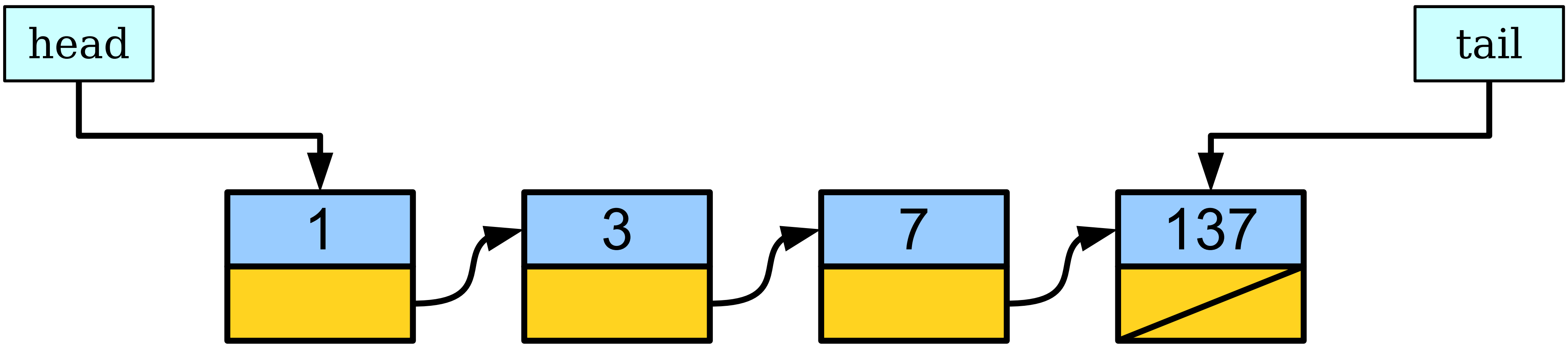


Dequeuing Things

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

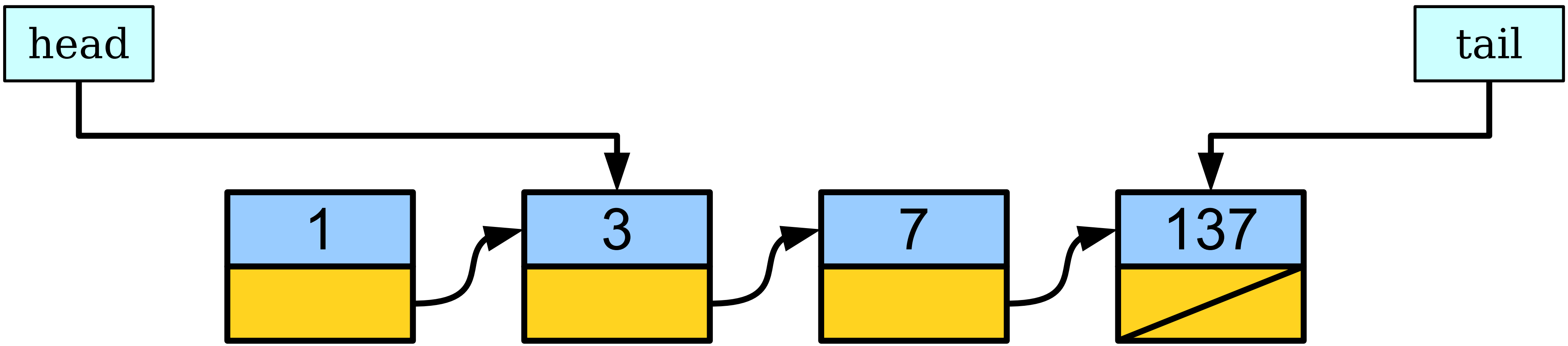
Dequeuing Things

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



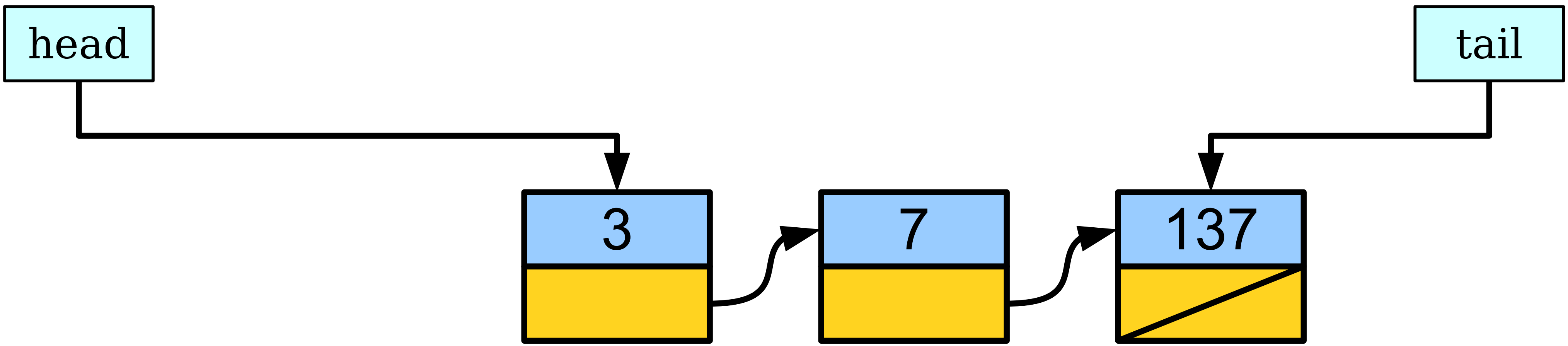
Dequeuing Things

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



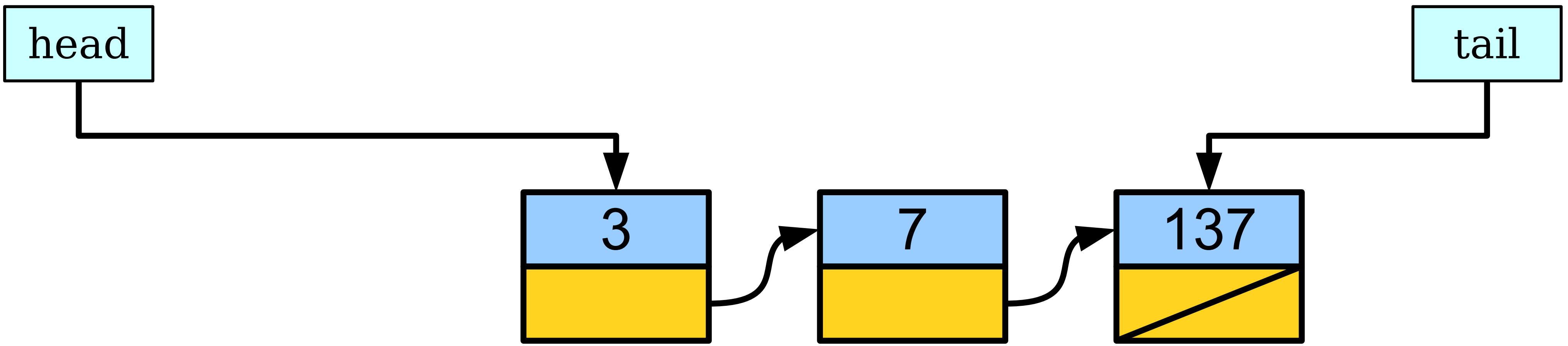
Dequeuing Things

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

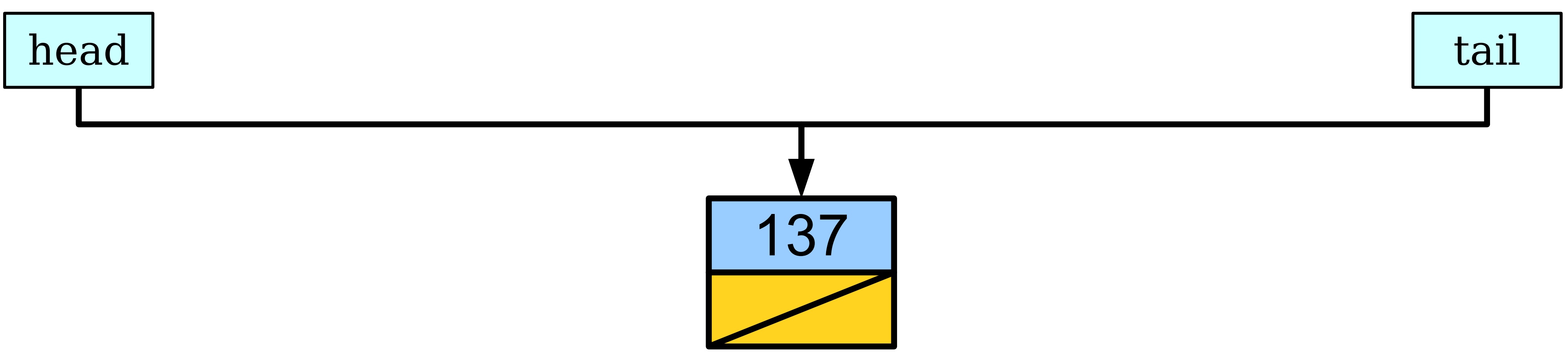


Dequeuing Things

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

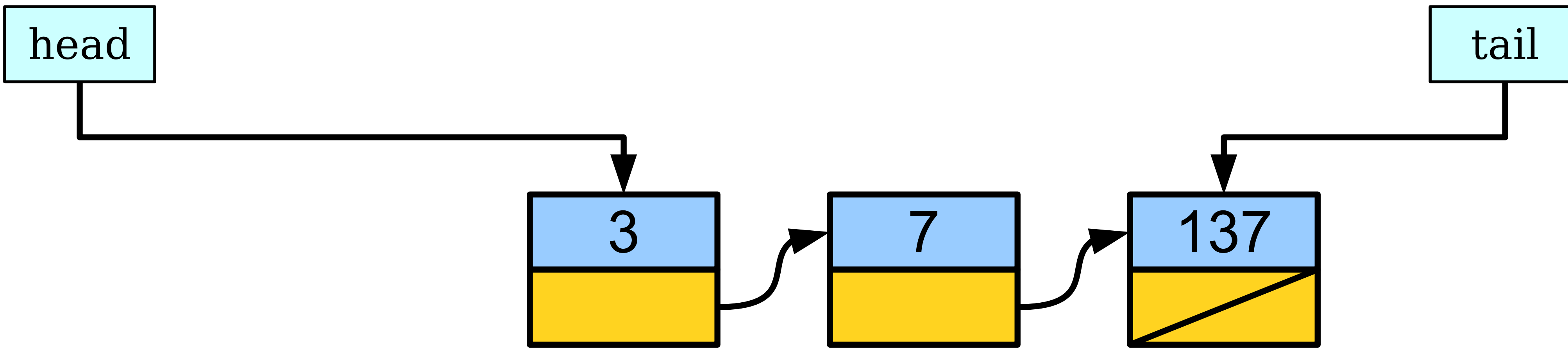


- ***Case 2:*** Dequeuing the last element.



Dequeuing Things

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



- ***Case 2:*** Dequeuing the last element.

Analyzing Efficiency

- What is the big-O complexity of a dequeue?
- Answer: **O(1)**.
- What is the big-O complexity of an enqueue?
- Answer: **O(1)**.