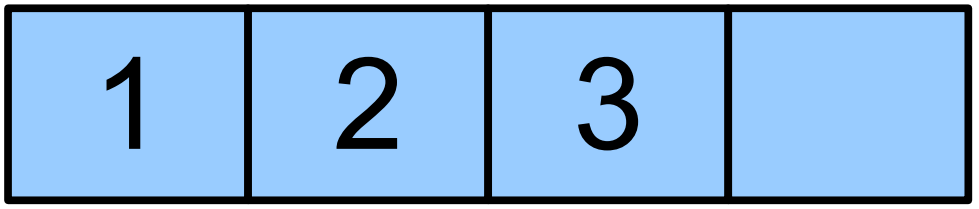
Linked Lists

Part One

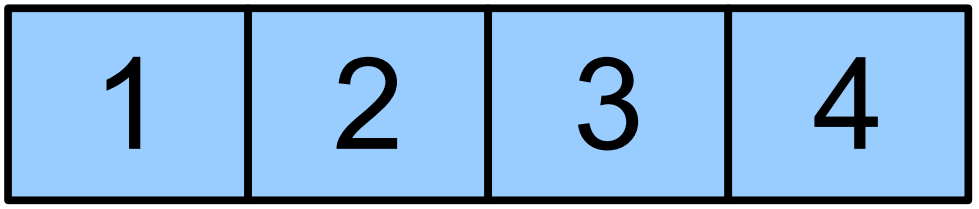
Array-Based Allocation

- Our current implementation of `Stack` uses dynamically-allocated arrays.
- To append an element:
  - If there is free space, put the element into that space.
  - Otherwise, get a *huge* new array and move everything over.



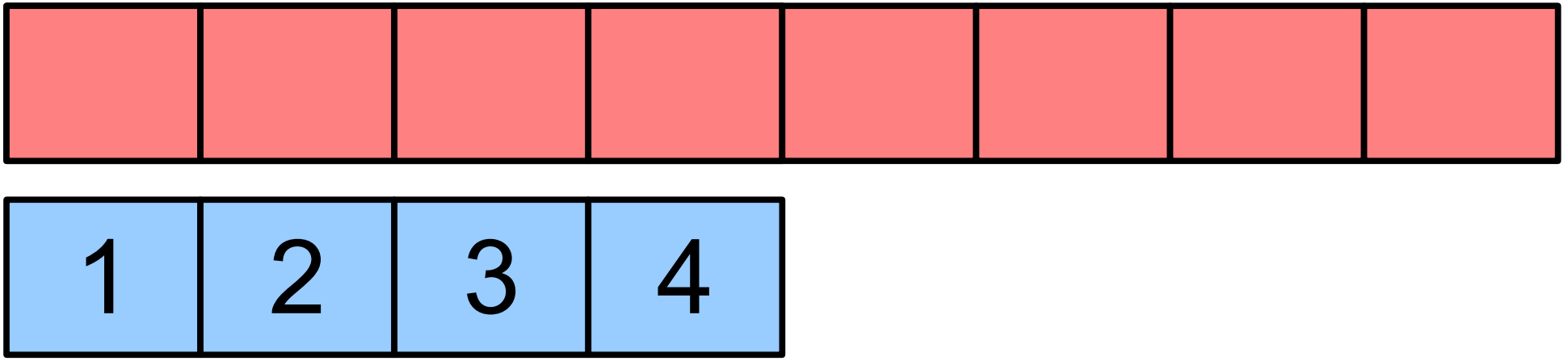
Array-Based Allocation

- Our current implementation of `Stack` uses dynamically-allocated arrays.
- To append an element:
  - If there is free space, put the element into that space.
  - Otherwise, get a *huge* new array and move everything over.



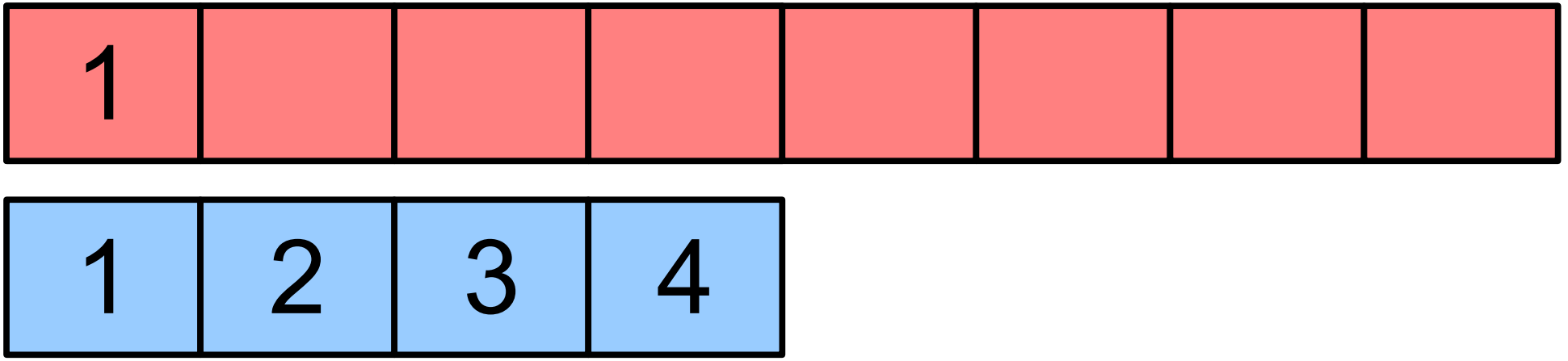
Array-Based Allocation

- Our current implementation of `Stack` uses dynamically-allocated arrays.
- To append an element:
  - If there is free space, put the element into that space.
  - Otherwise, get a *huge* new array and move everything over.



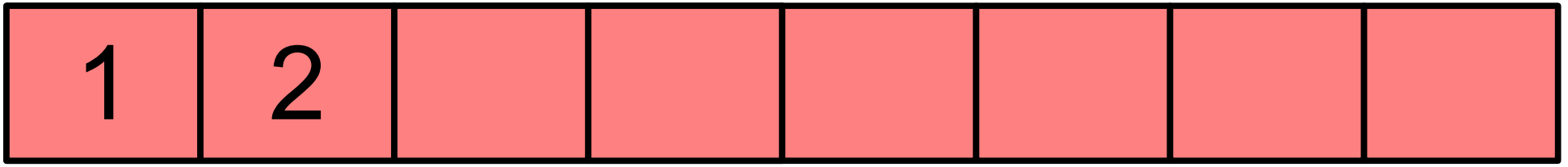
Array-Based Allocation

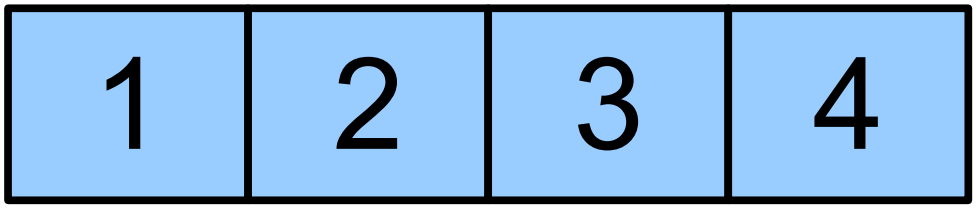
- Our current implementation of `Stack` uses dynamically-allocated arrays.
- To append an element:
  - If there is free space, put the element into that space.
  - Otherwise, get a *huge* new array and move everything over.



Array-Based Allocation

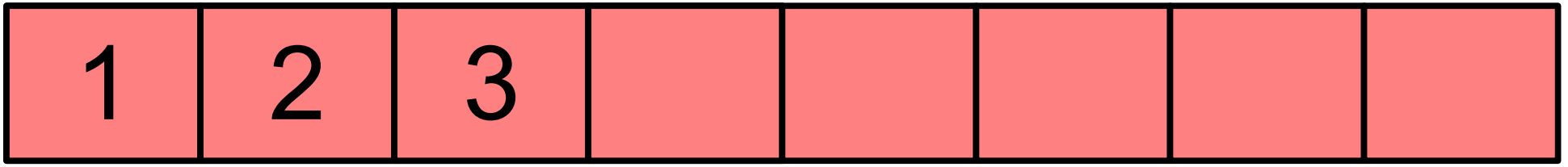
- Our current implementation of `Stack` uses dynamically-allocated arrays.
- To append an element:
  - If there is free space, put the element into that space.
  - Otherwise, get a *huge* new array and move everything over.

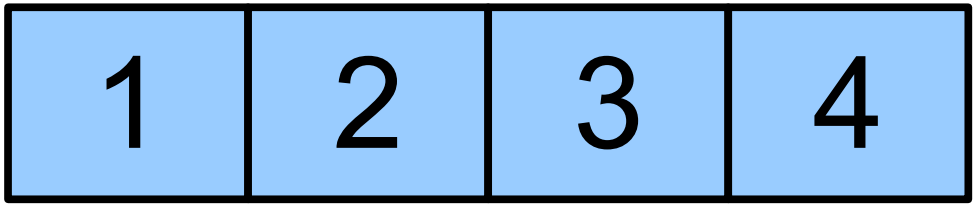




Array-Based Allocation

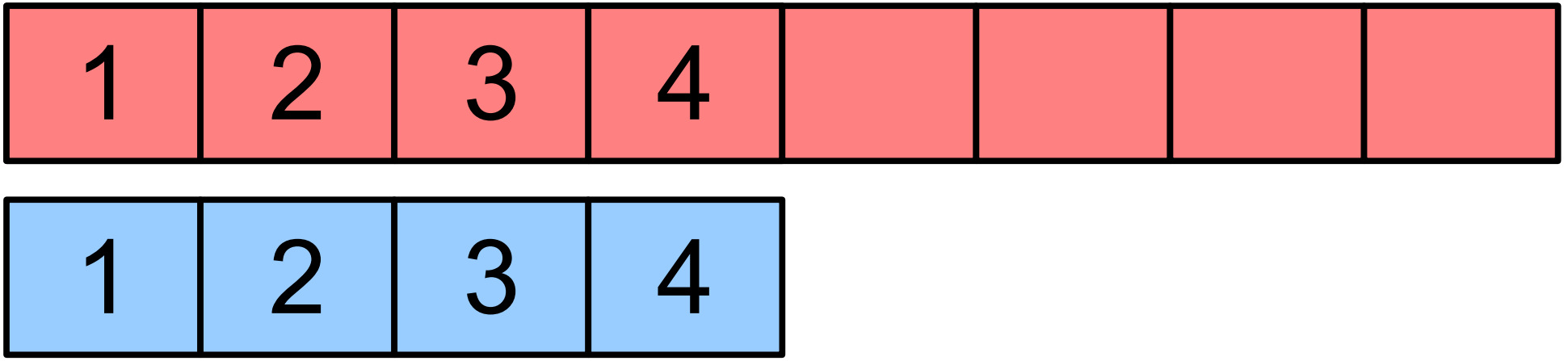
- Our current implementation of `Stack` uses dynamically-allocated arrays.
- To append an element:
  - If there is free space, put the element into that space.
  - Otherwise, get a *huge* new array and move everything over.





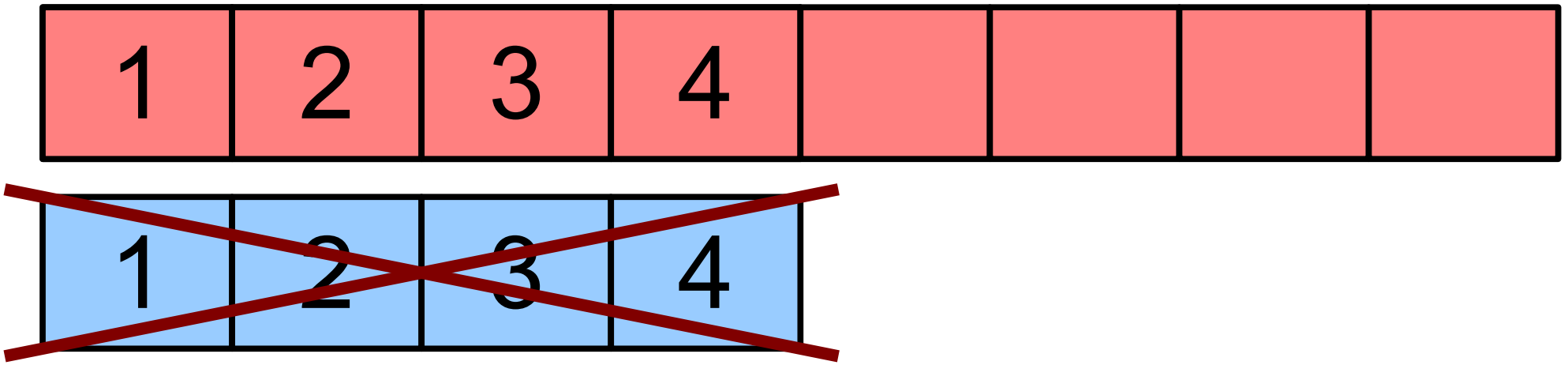
Array-Based Allocation

- Our current implementation of `Stack` uses dynamically-allocated arrays.
- To append an element:
  - If there is free space, put the element into that space.
  - Otherwise, get a *huge* new array and move everything over.



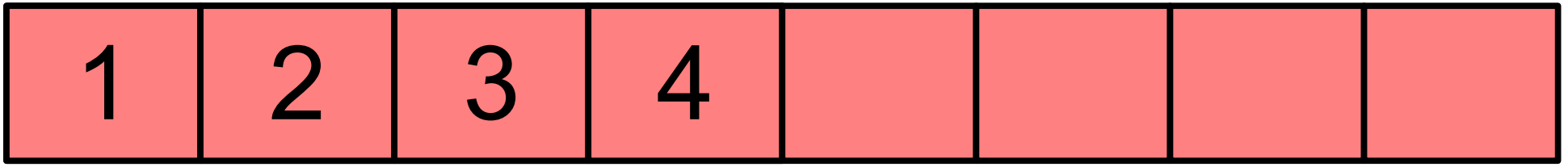
Array-Based Allocation

- Our current implementation of `Stack` uses dynamically-allocated arrays.
- To append an element:
  - If there is free space, put the element into that space.
  - Otherwise, get a *huge* new array and move everything over.



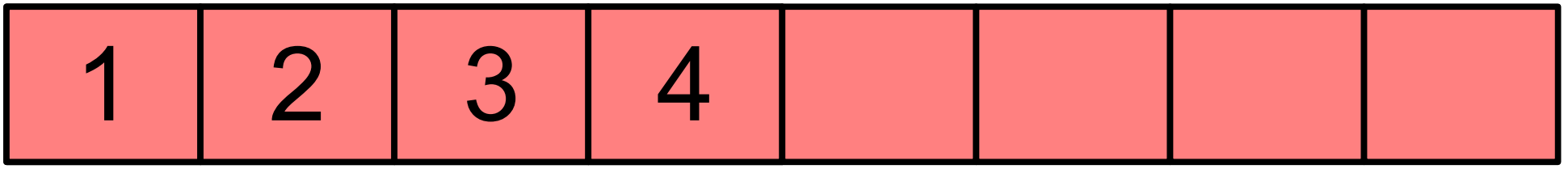
Array-Based Allocation

- Our current implementation of `Stack` uses dynamically-allocated arrays.
- To append an element:
  - If there is free space, put the element into that space.
  - Otherwise, get a *huge* new array and move everything over.



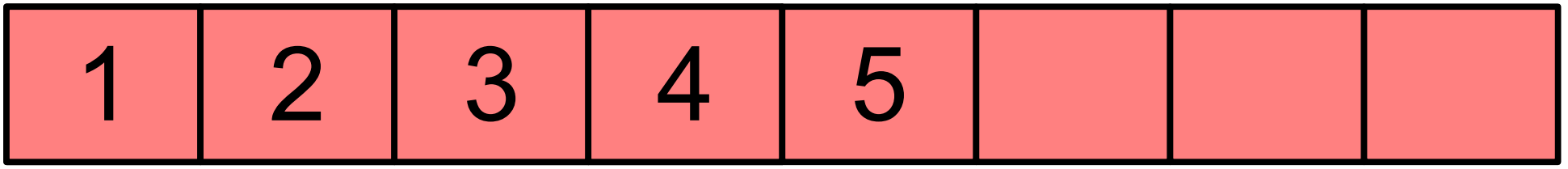
Array-Based Allocation

- Our current implementation of `Stack` uses dynamically-allocated arrays.
- To append an element:
  - If there is free space, put the element into that space.
  - Otherwise, get a *huge* new array and move everything over.



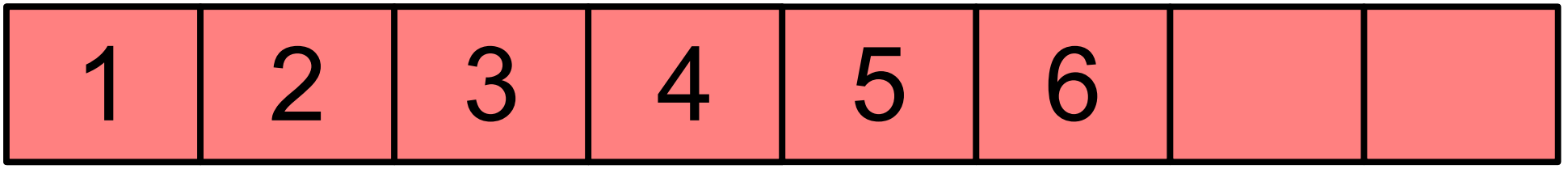
Array-Based Allocation

- Our current implementation of `Stack` uses dynamically-allocated arrays.
- To append an element:
  - If there is free space, put the element into that space.
  - Otherwise, get a *huge* new array and move everything over.



Array-Based Allocation

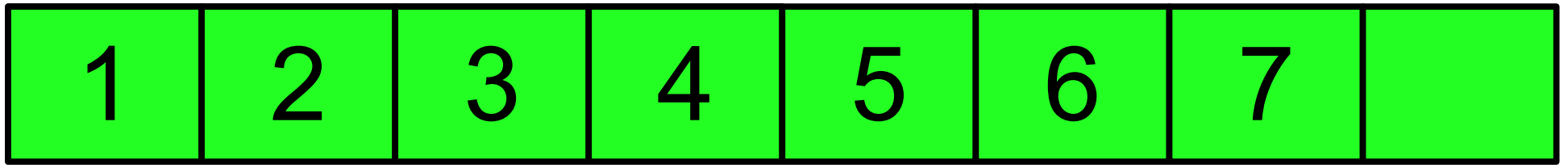
- Our current implementation of `Stack` uses dynamically-allocated arrays.
- To append an element:
  - If there is free space, put the element into that space.
  - Otherwise, get a *huge* new array and move everything over.



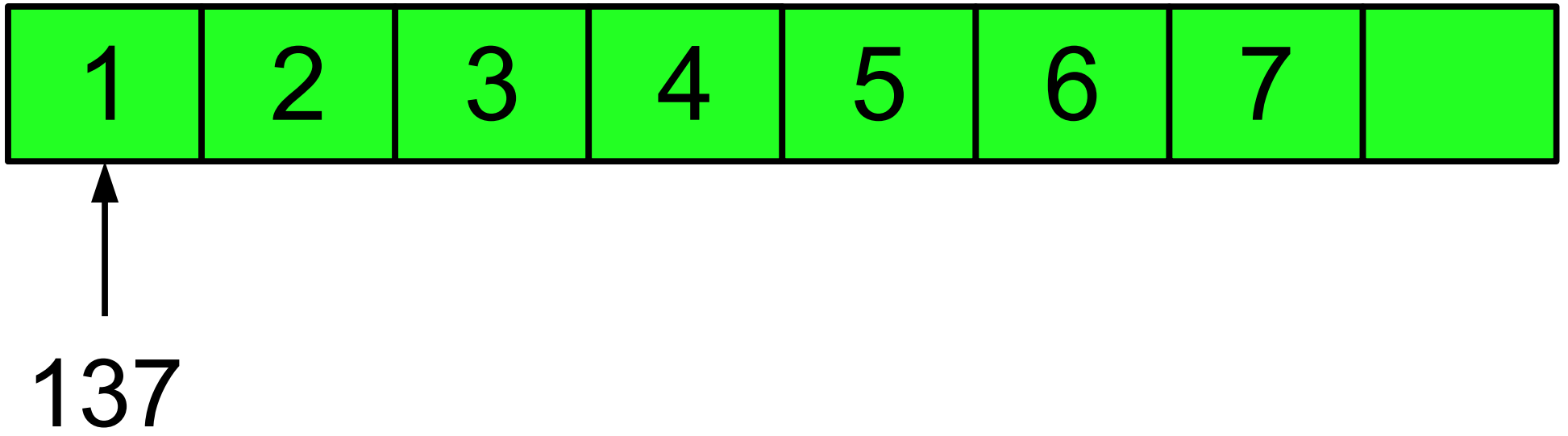
A Different Idea

- Instead of reallocating a huge array to get the space we need, why not just get a tiny amount of extra space for the next element?
- Think about how you take notes: when you run out of space on a page, you just get a new page. You don't copy your entire set of notes onto a longer sheet of paper!

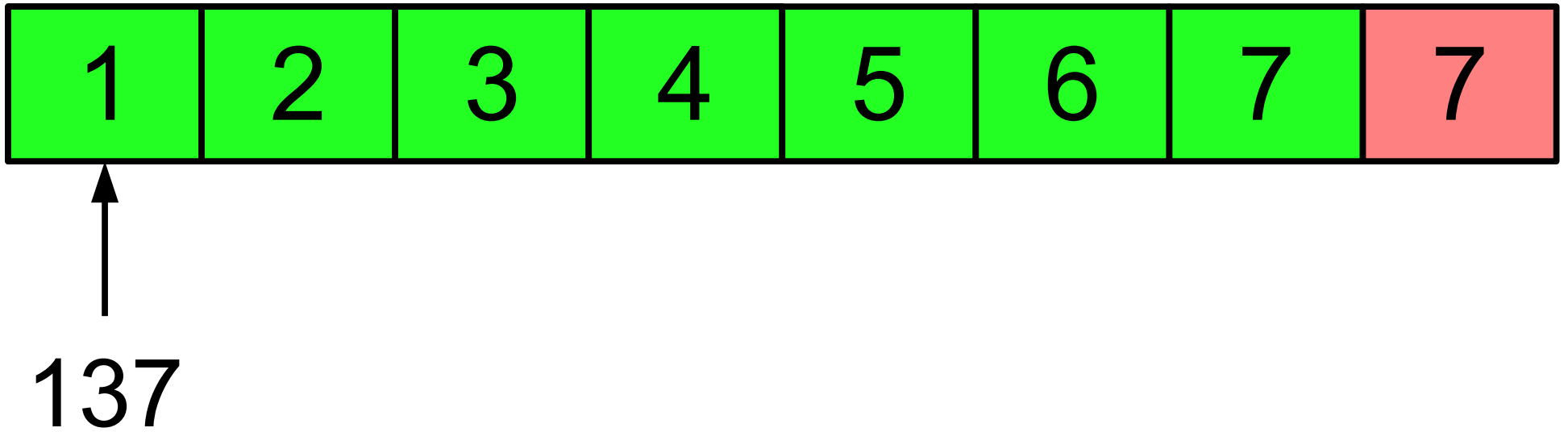
Excuse Me, Coming Through...



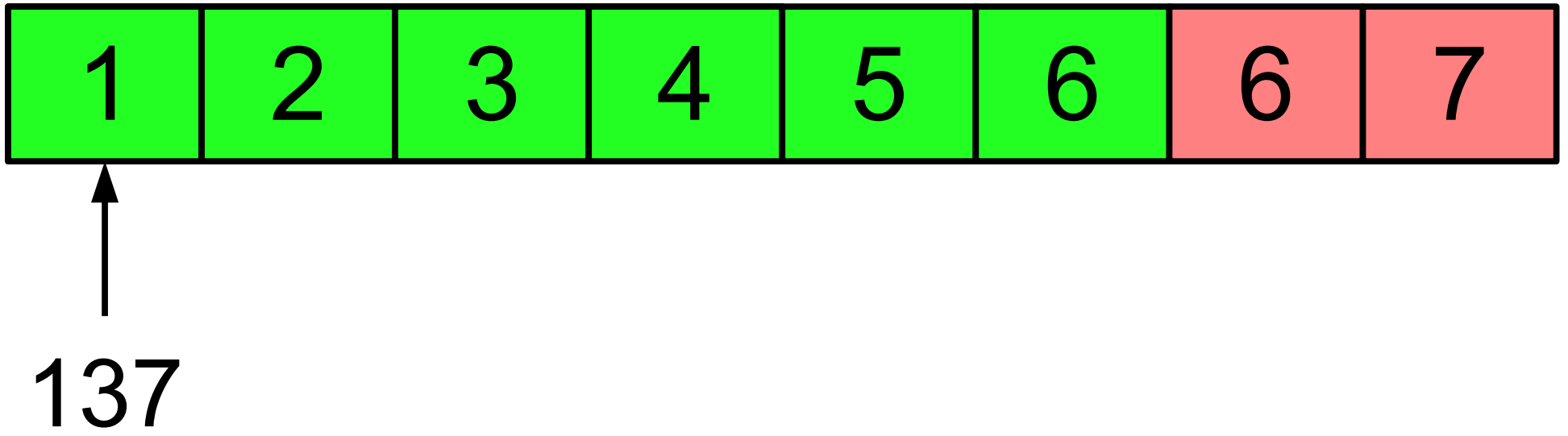
Excuse Me, Coming Through...



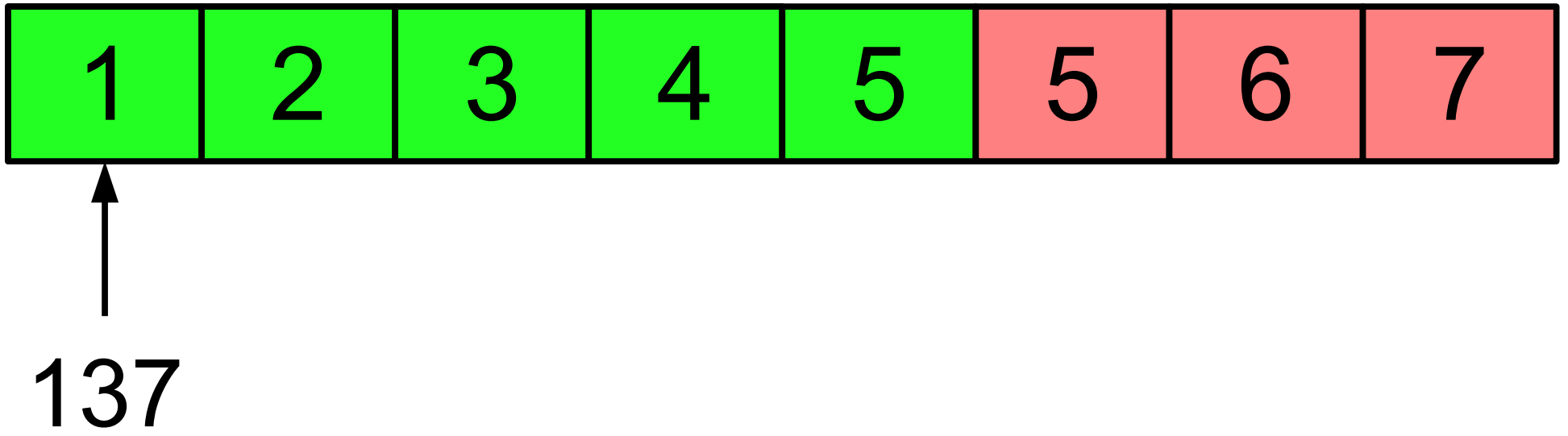
Excuse Me, Coming Through...



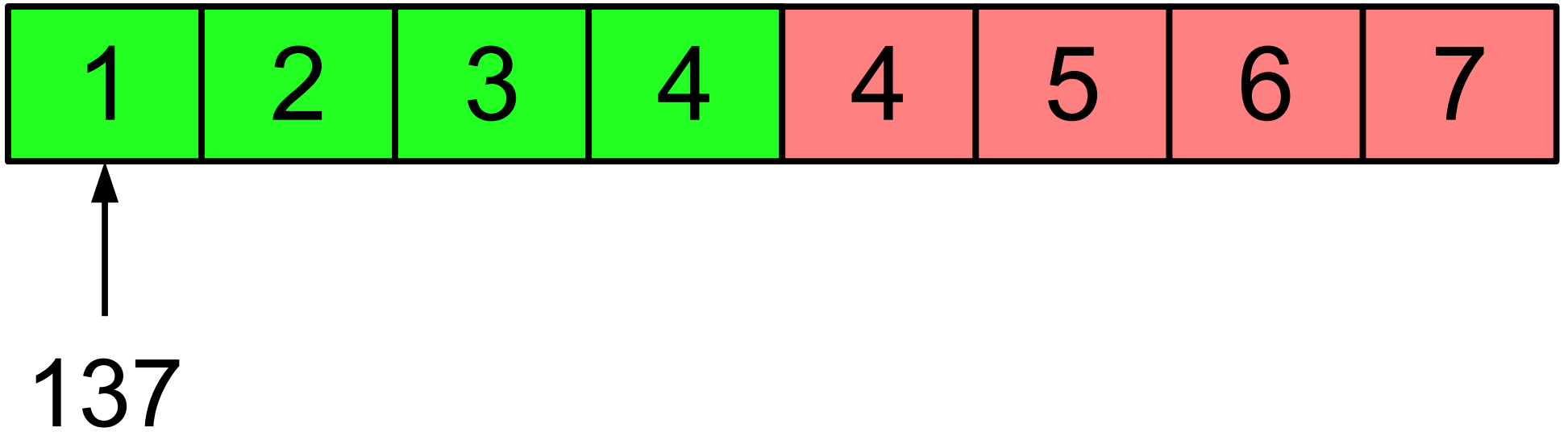
Excuse Me, Coming Through...



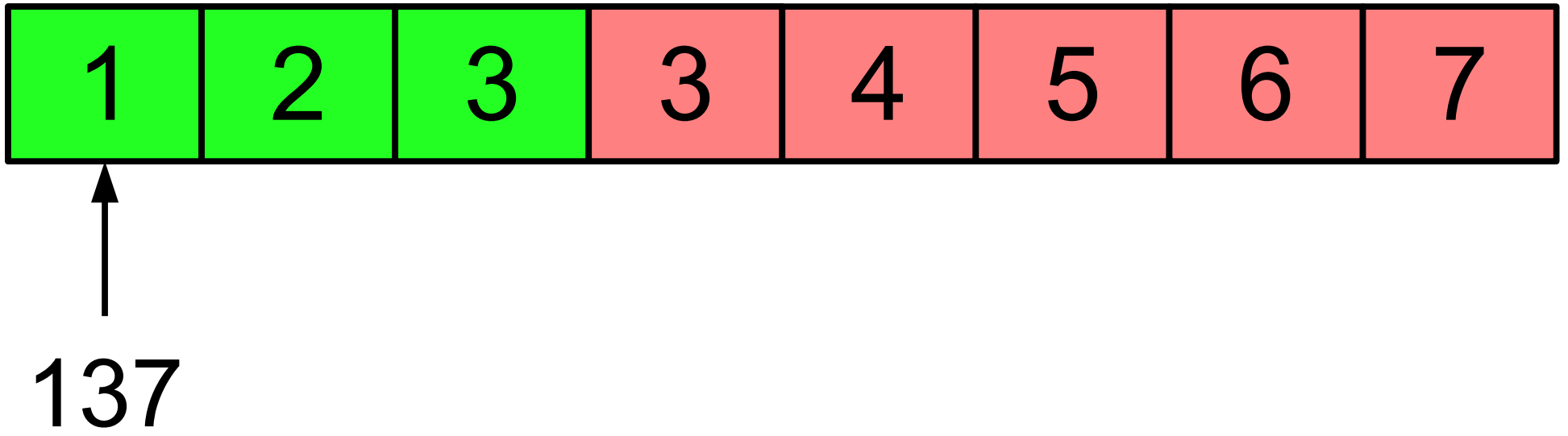
Excuse Me, Coming Through...



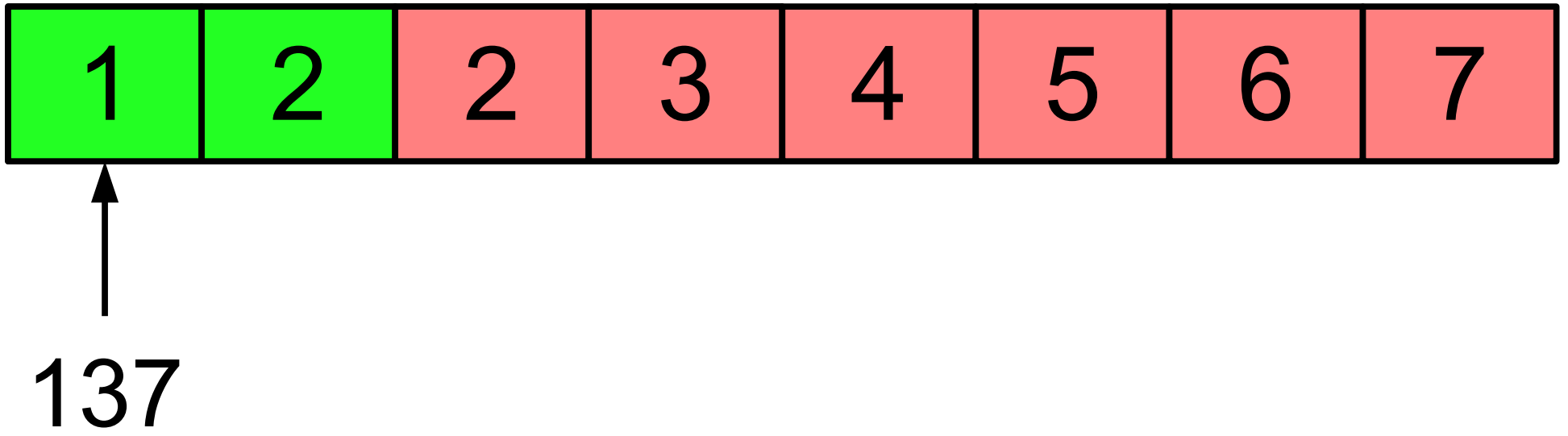
Excuse Me, Coming Through...



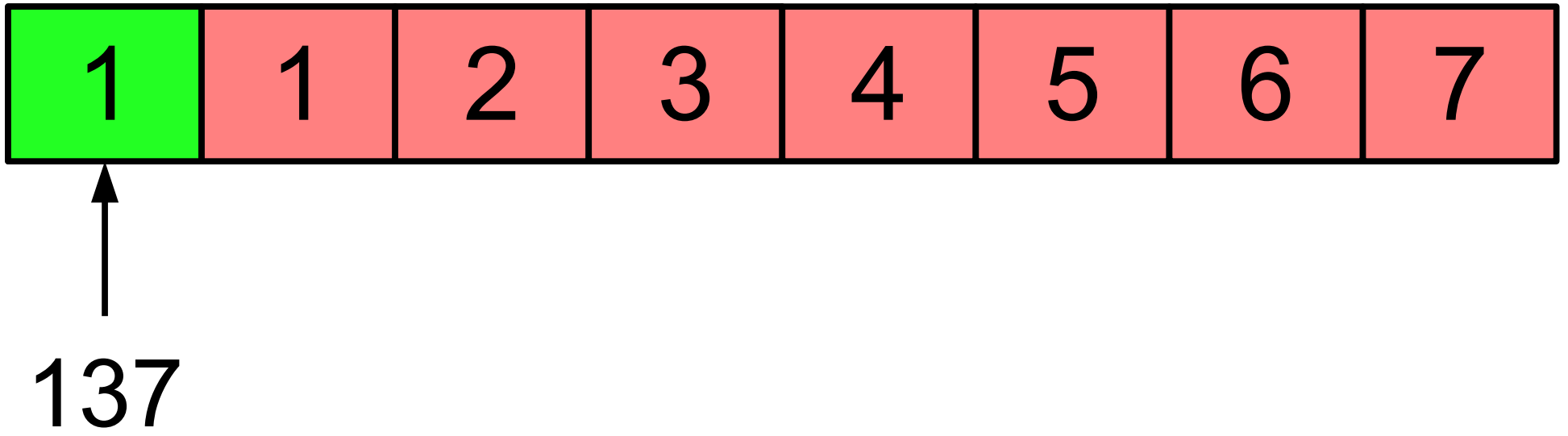
Excuse Me, Coming Through...



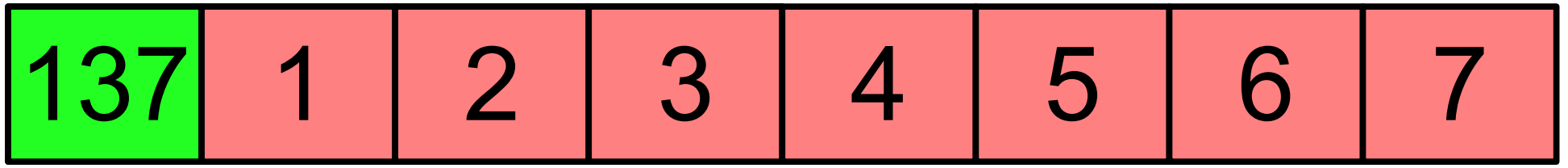
Excuse Me, Coming Through...



Excuse Me, Coming Through...

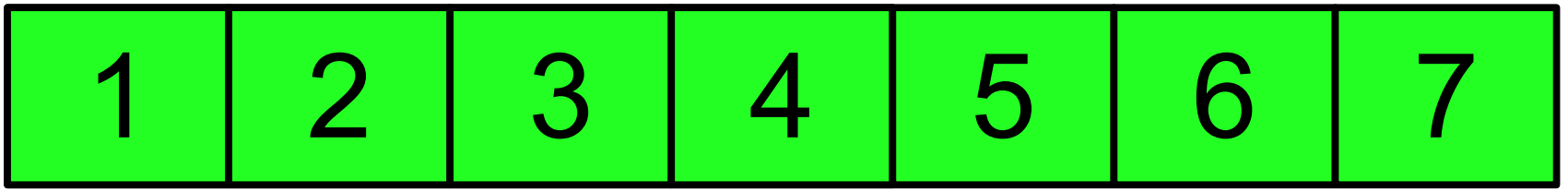


Excuse Me, Coming Through...



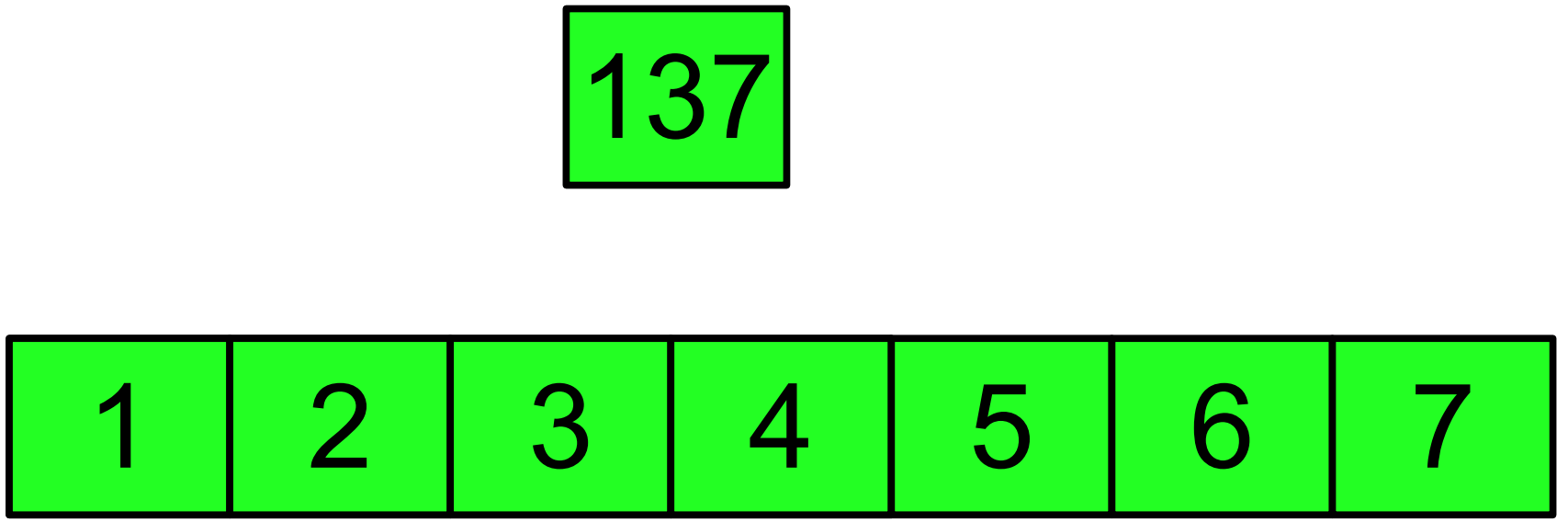
Shoving Things Over

- Right now, inserting an element into a middle of a `Vector` can be very costly.
- Couldn't we just do something like this?



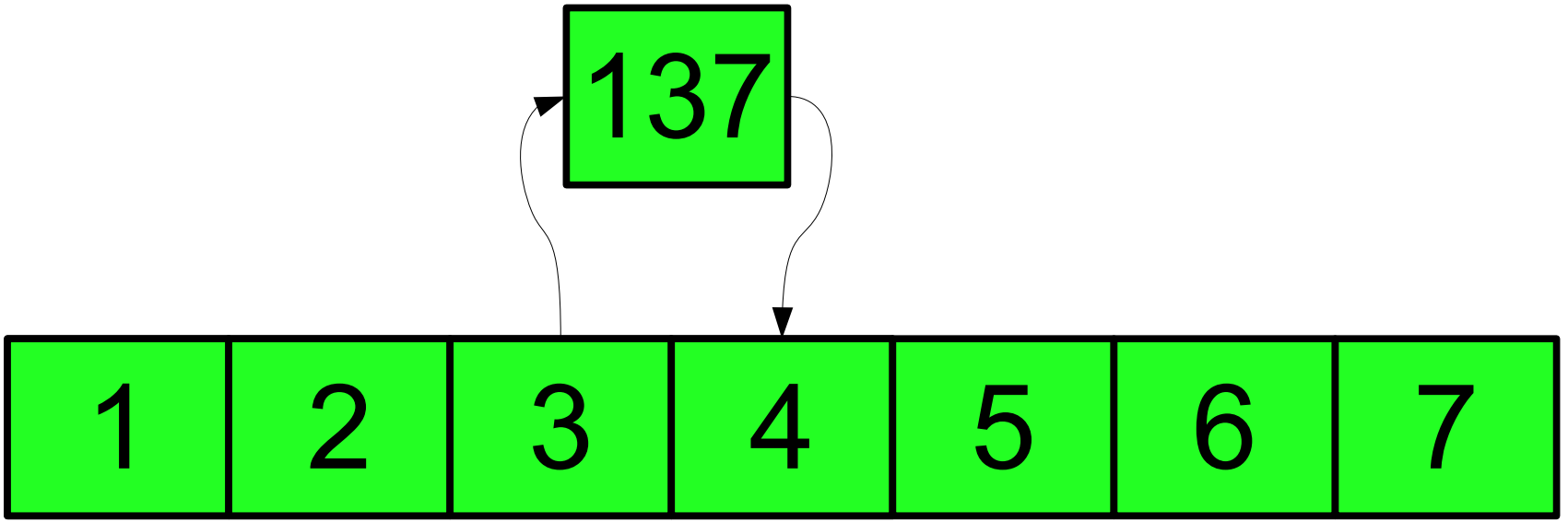
Shoving Things Over

- Right now, inserting an element into a middle of a `Vector` can be very costly.
- Couldn't we just do something like this?



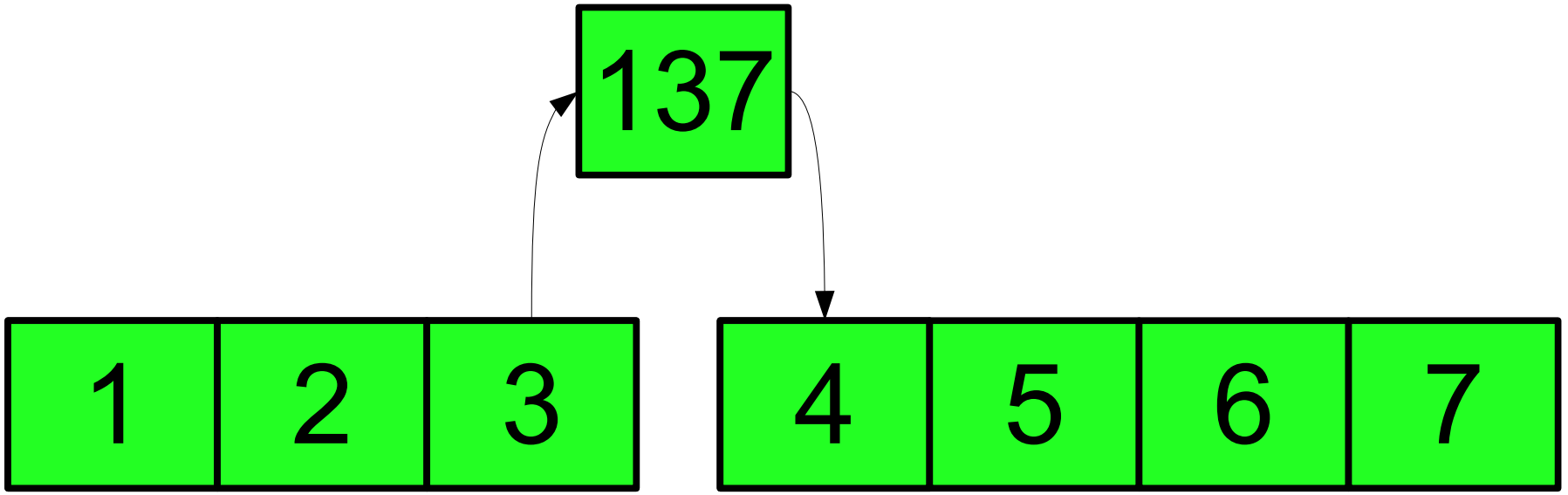
Shoving Things Over

- Right now, inserting an element into a middle of a `Vector` can be very costly.
- Couldn't we just do something like this?



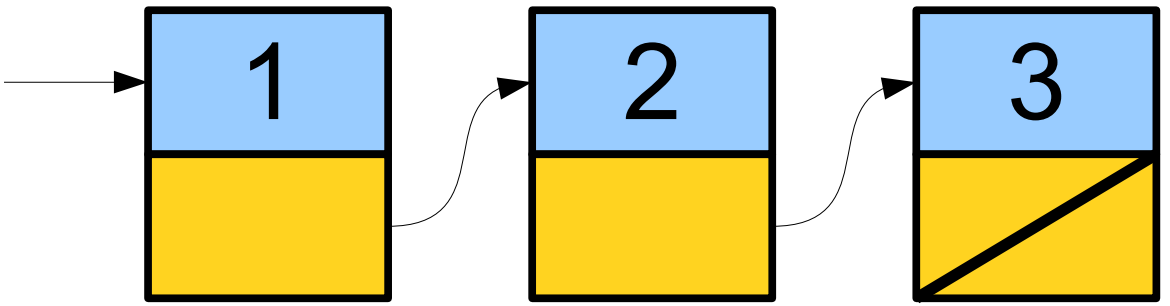
Shoving Things Over

- Right now, inserting an element into a middle of a `Vector` can be very costly.
- Couldn't we just do something like this?



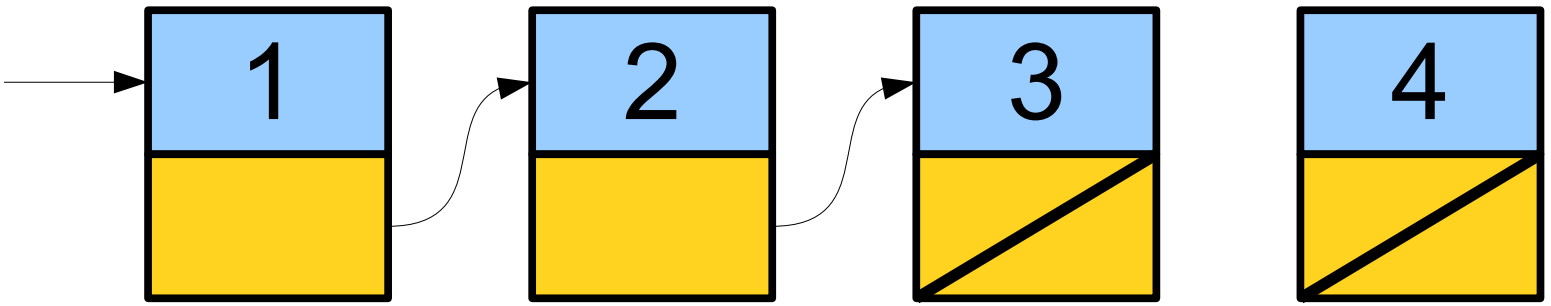
Linked Lists at a Glance

- 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.



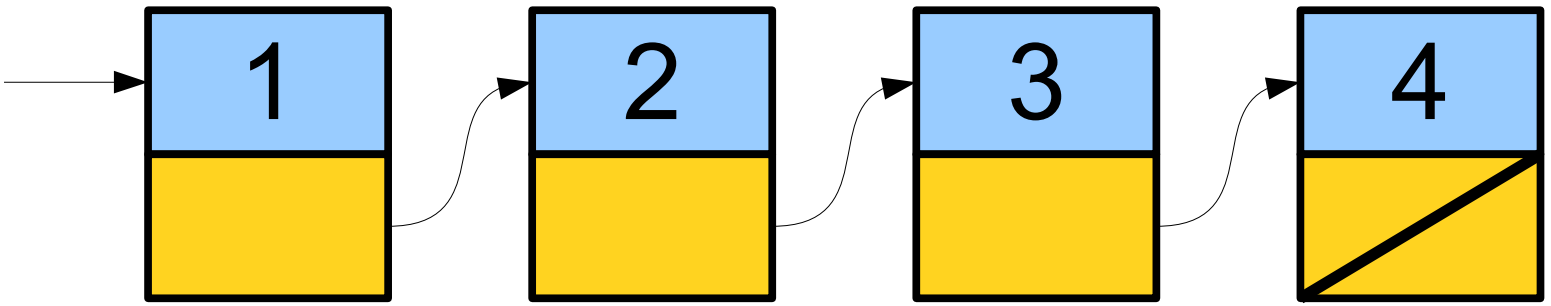
Linked Lists at a Glance

- 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.



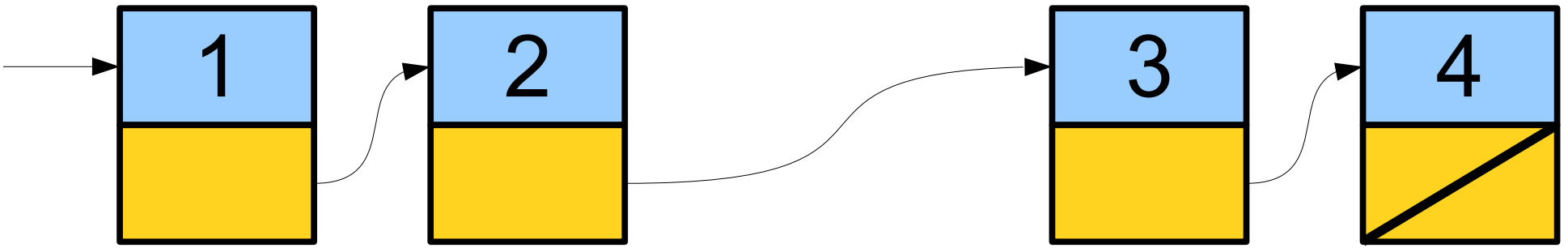
Linked Lists at a Glance

- 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.



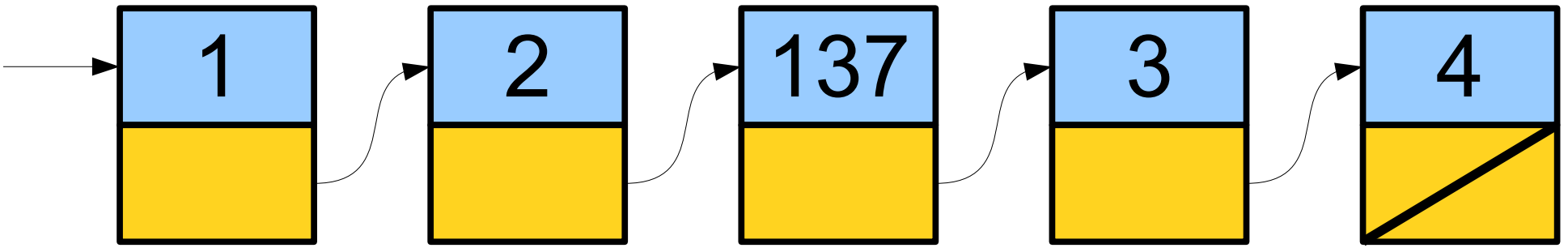
Linked Lists at a Glance

- 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.



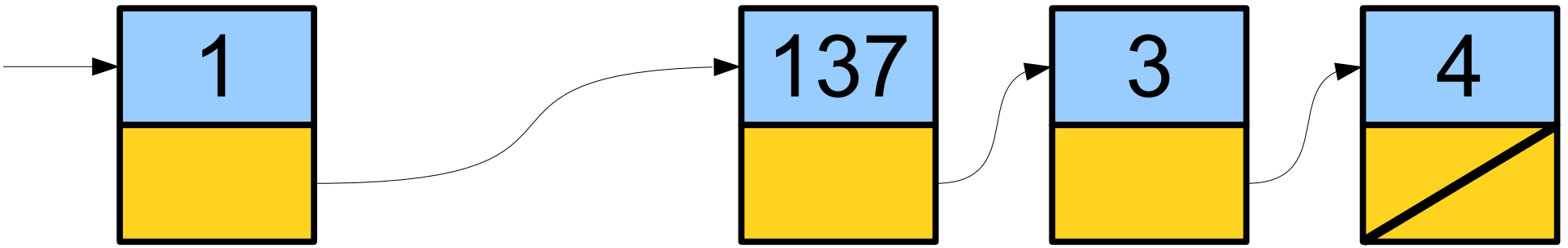
Linked Lists at a Glance

- 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.



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.



Linked Lists at a Glance

- Can efficiently splice new elements into the list or remove existing elements anywhere in the list.
- Never have to do a massive copy step; insertion is efficient in the worst-case.
- Has some tradeoffs; we'll see this later.

Two Technical Prerequisites

Dynamic Memory Allocation

- We have seen the `new` keyword used to allocate arrays, but it can also be used to allocate single objects.
- The syntax

```
new T(args)
```

creates a new object of type *T* passing the appropriate arguments to the constructor, then returns a pointer to it.

Dynamic Memory Allocation

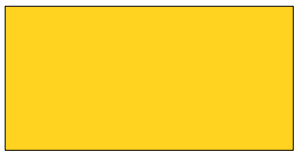
```
struct Tribute {
    string name;
    int districtNumber;
};
```

Dynamic Memory Allocation

```
struct Tribute {
    string name;
    int districtNumber;
};

Tribute* t = new Tribute;
```

Dynamic Memory Allocation

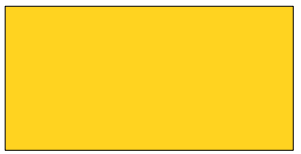
```
struct Tribute {
    string name;
    int districtNumber;
};

Tribute* t = new Tribute;
```



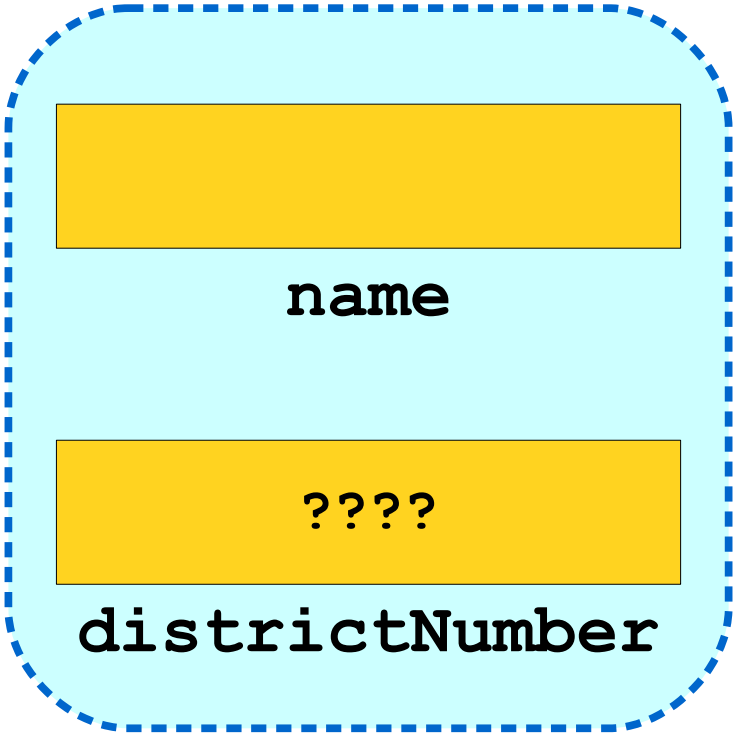
t

Dynamic Memory Allocation

```
struct Tribute {
    string name;
    int districtNumber;
};

Tribute* t = new Tribute;
```

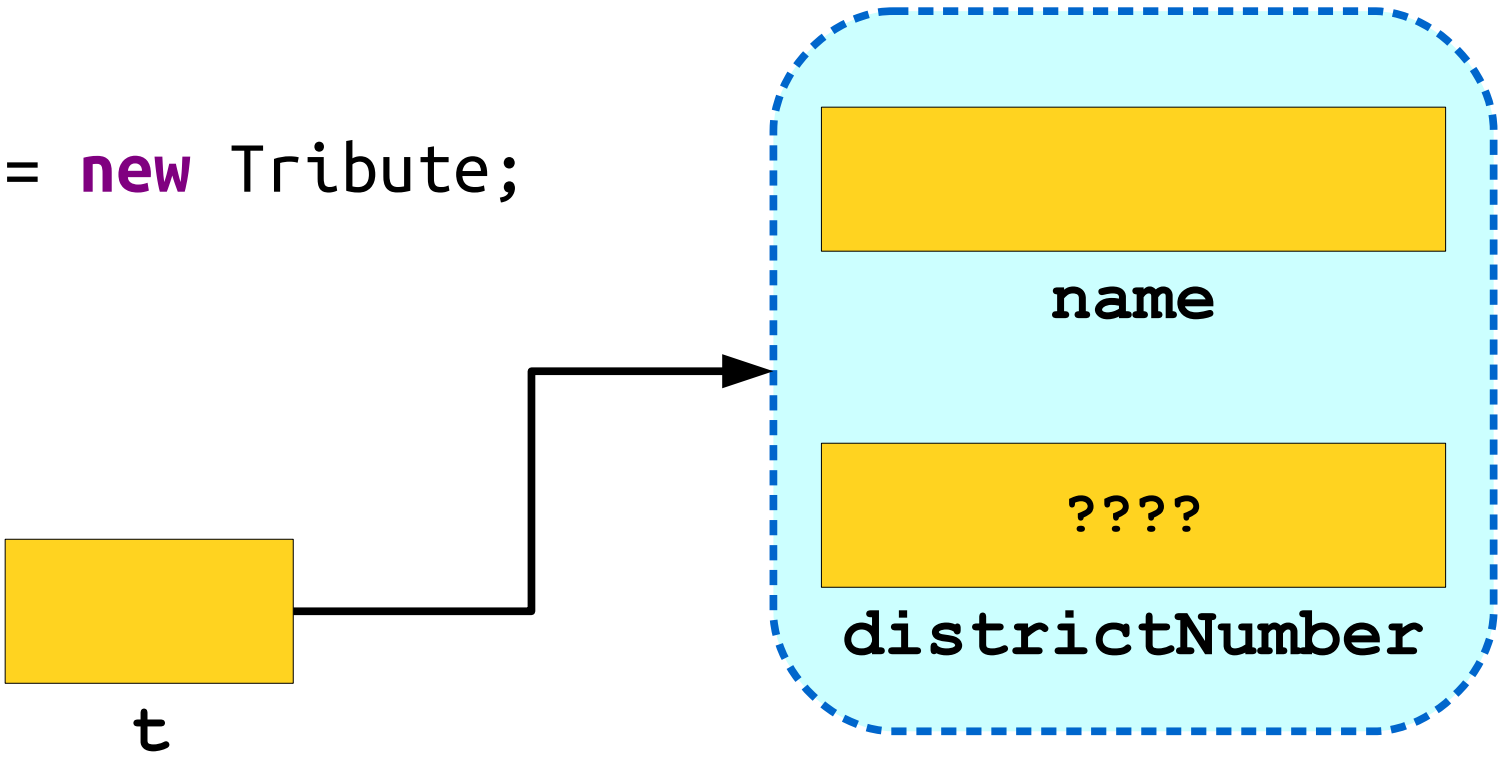


t



Dynamic Memory Allocation

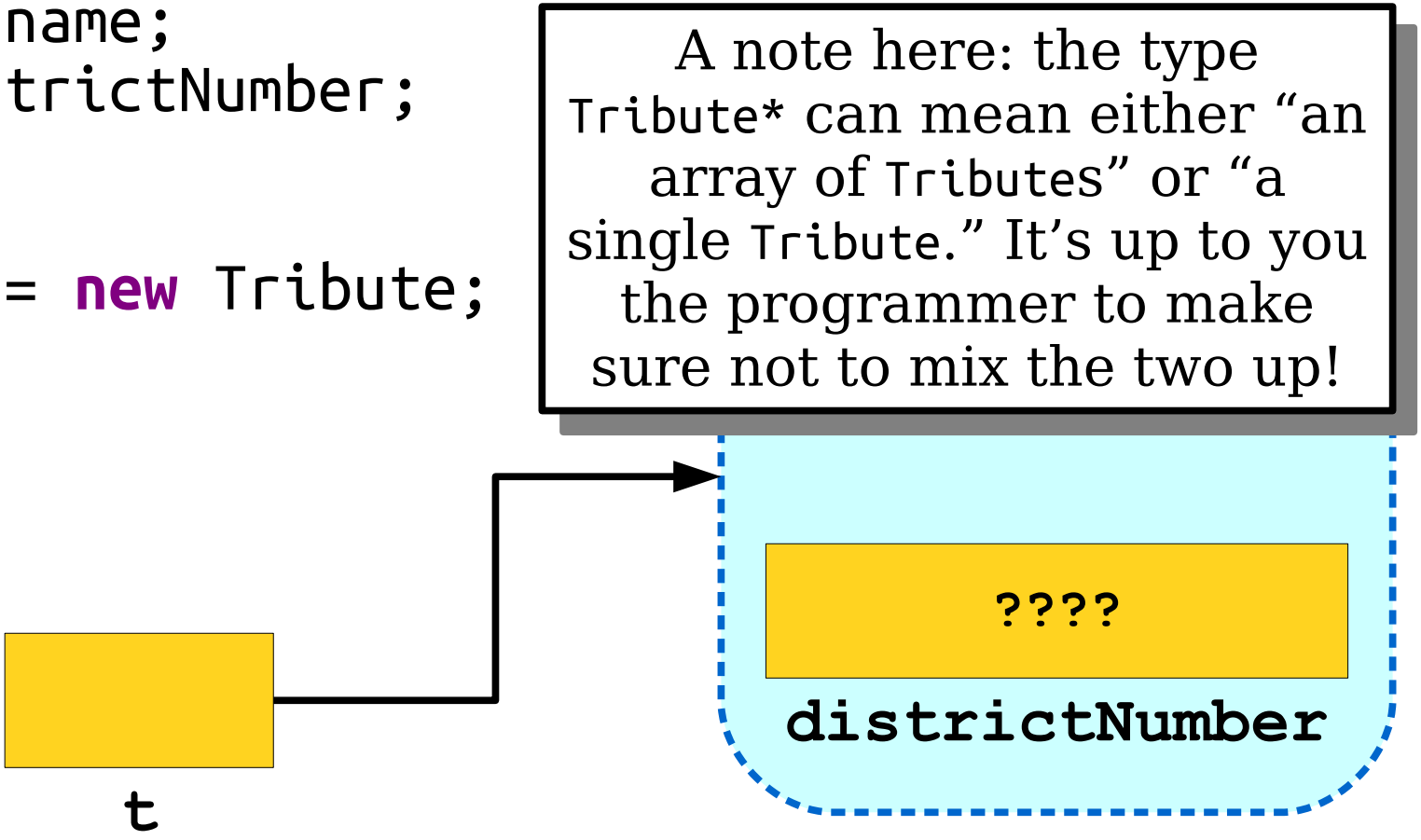
```
struct Tribute {
    string name;
    int districtNumber;
};

Tribute* t = new Tribute;
```



Dynamic Memory Allocation

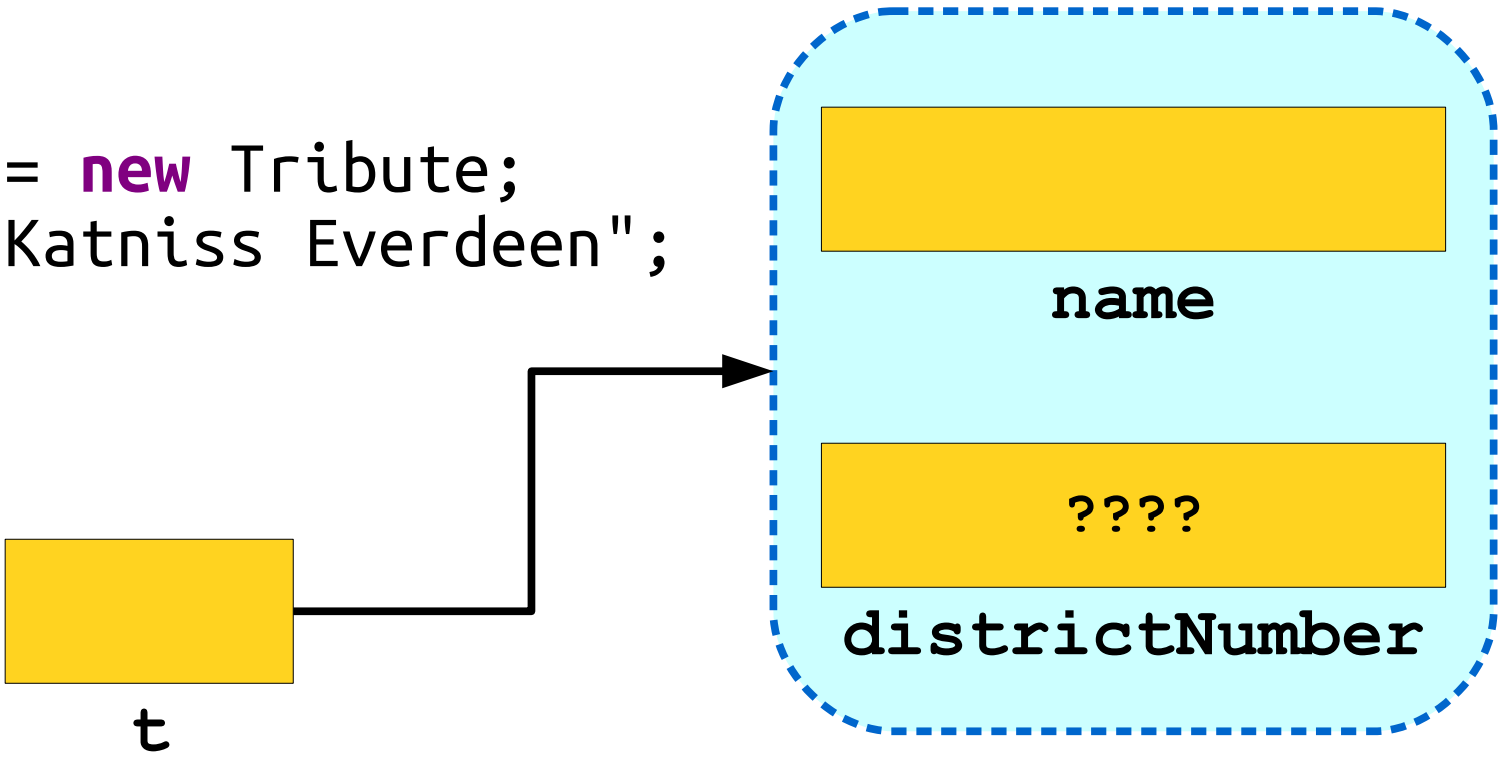
```
struct Tribute {
    string name;
    int districtNumber;
};

Tribute* t = new Tribute;
```

A note here: the type `Tribute*` can mean either "an array of `Tributes`" or "a single `Tribute`." It's up to you the programmer to make sure not to mix the two up!



Dynamic Memory Allocation

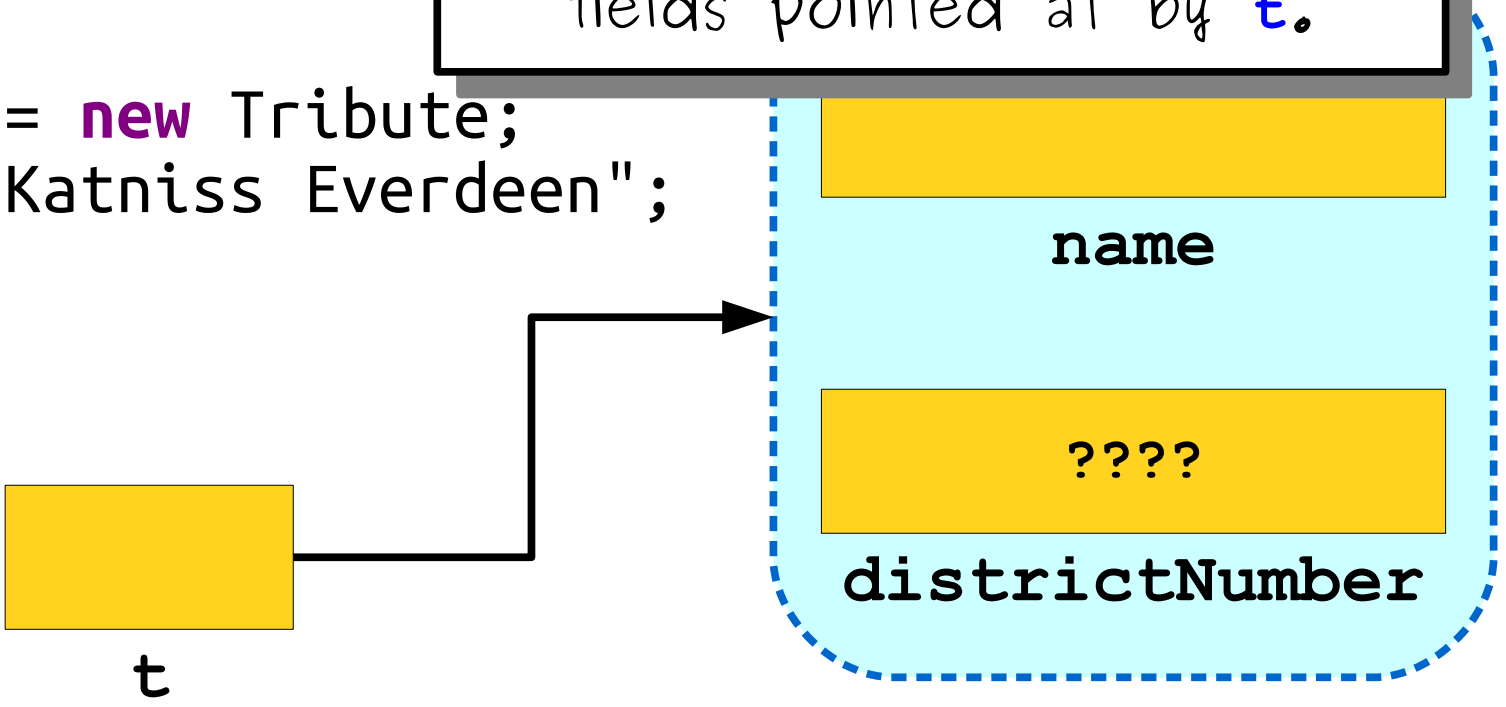
```
struct Tribute {
    string name;
    int districtNumber;
};

Tribute* t = new Tribute;
t->name = "Katniss Everdeen";
```



Dynamic Memory Allocation

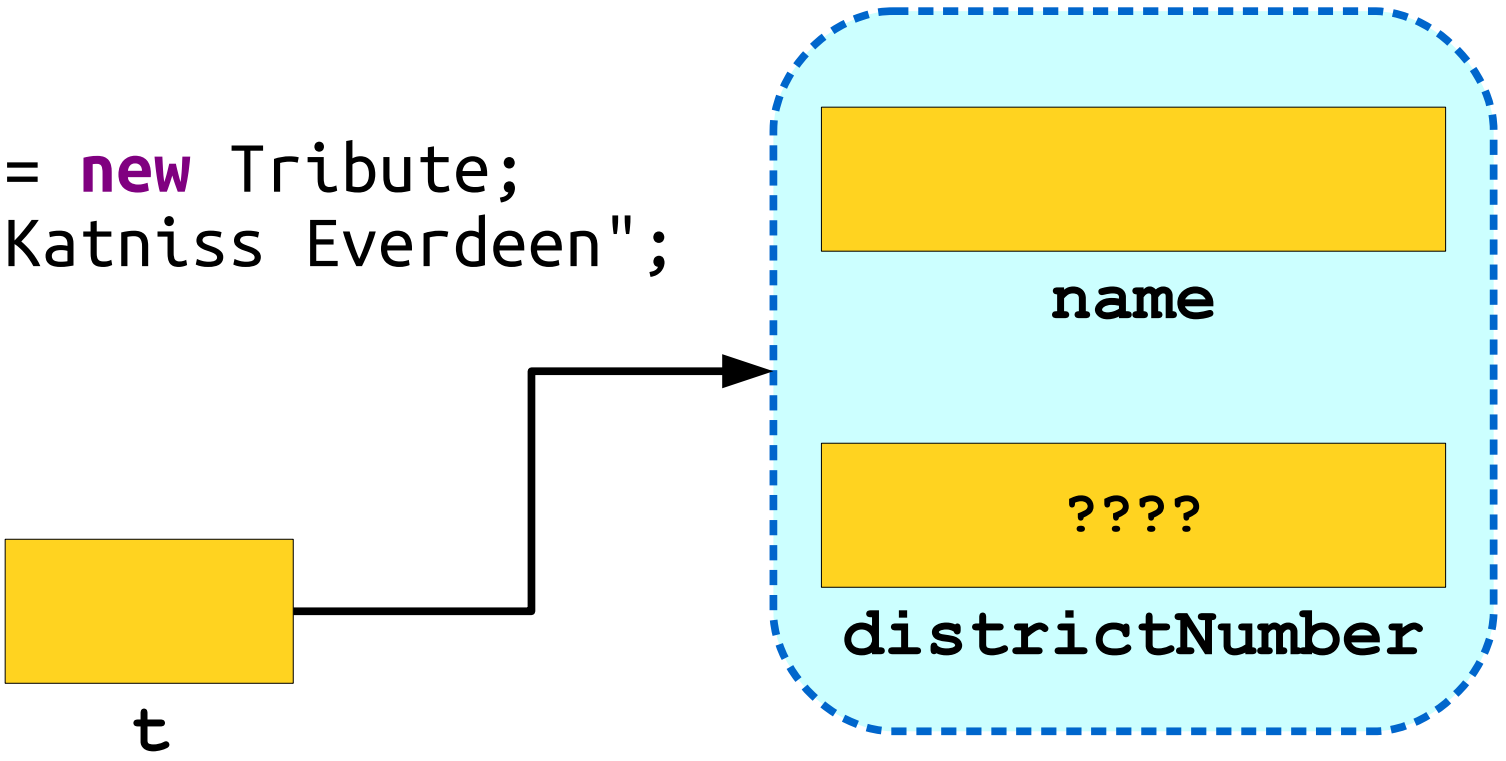
```
struct Tribute {
    string name;
    int districtNumber;
};

Tribute* t = new Tribute;
t->name = "Katniss Everdeen";
```

Because **t** is a pointer to a **Tribute**, not an actual **Tribute**, we have to use the arrow operator to access the fields pointed at by **t**.



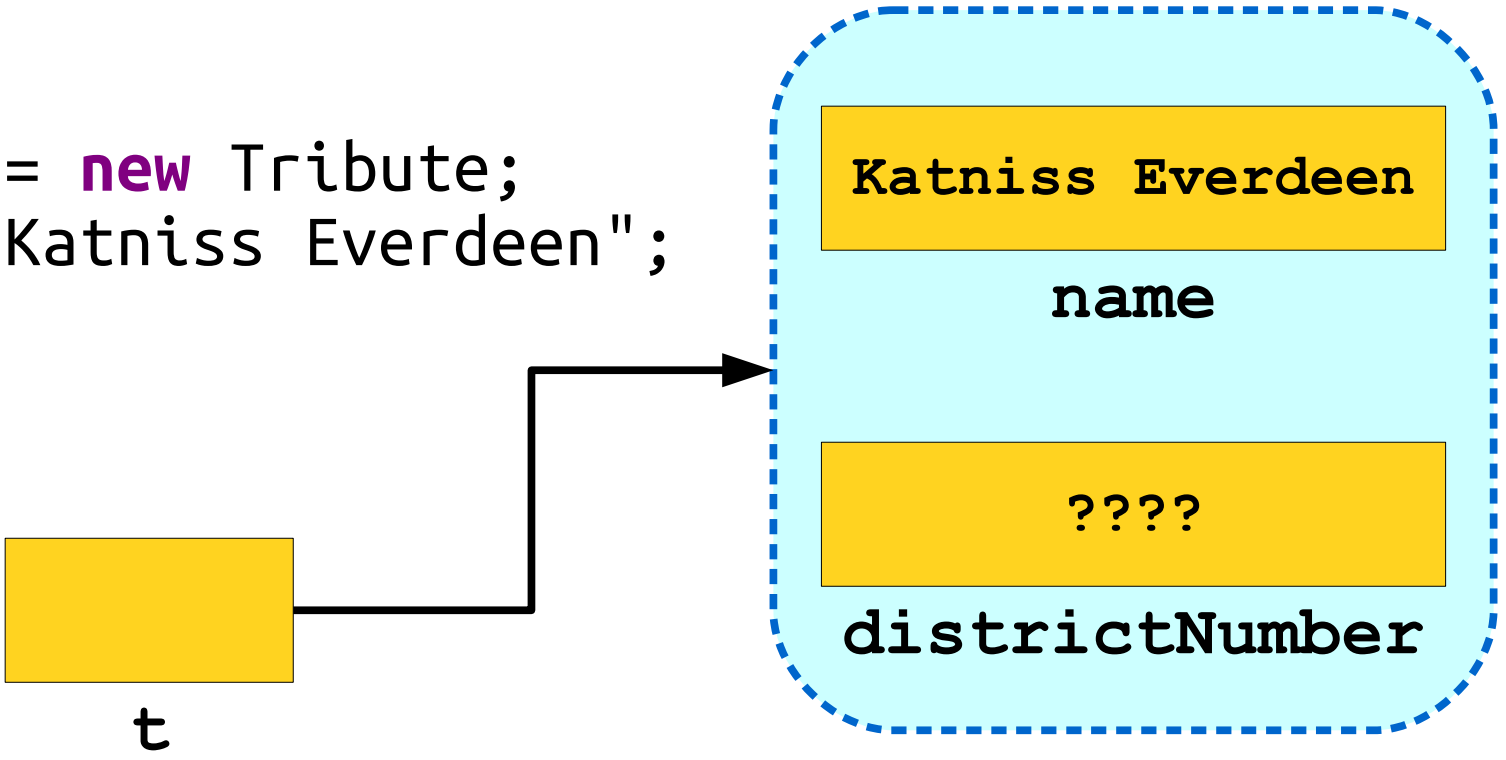
Dynamic Memory Allocation

```
struct Tribute {
    string name;
    int districtNumber;
};

Tribute* t = new Tribute;
t->name = "Katniss Everdeen";
```



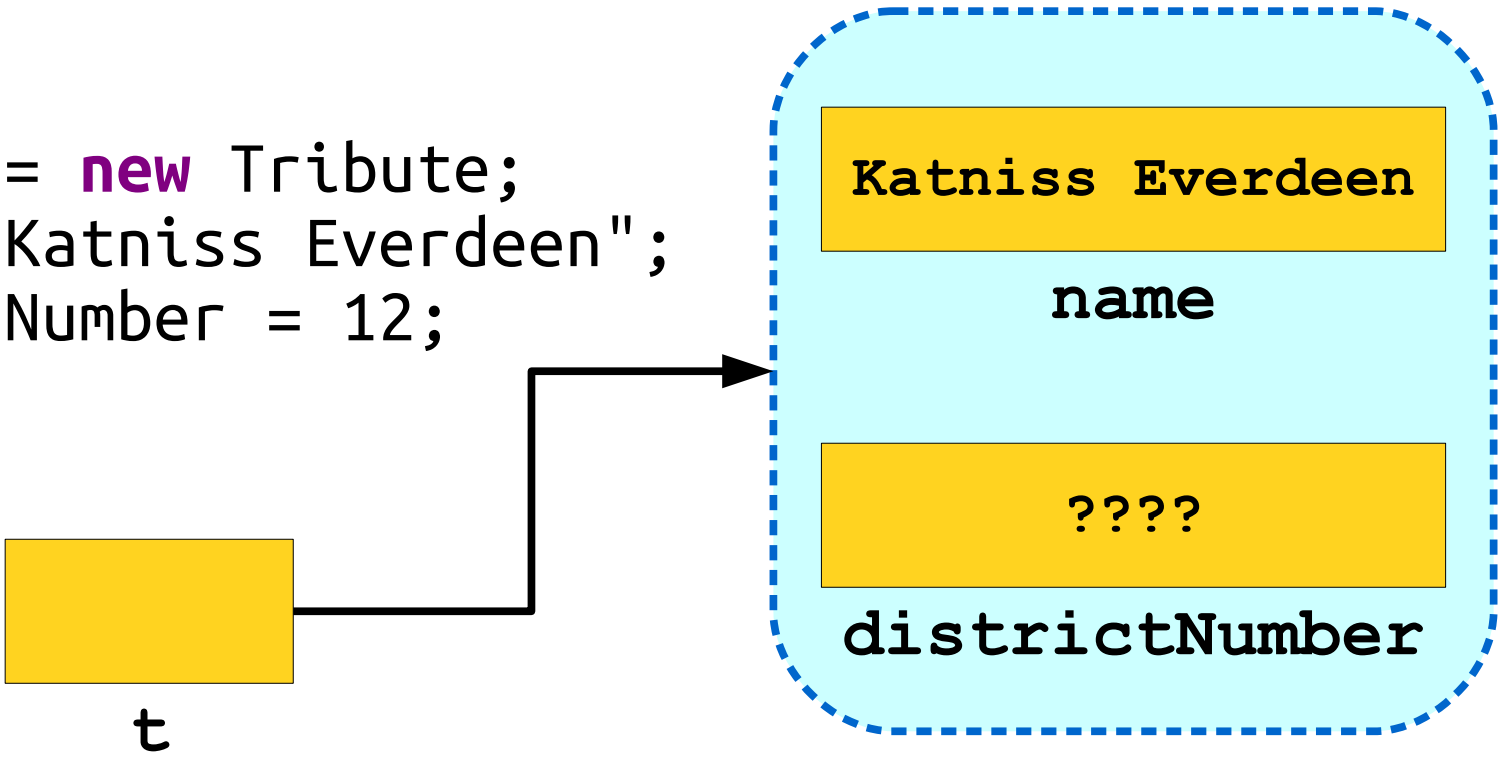
Dynamic Memory Allocation

```
struct Tribute {
    string name;
    int districtNumber;
};

Tribute* t = new Tribute;
t->name = "Katniss Everdeen";
```



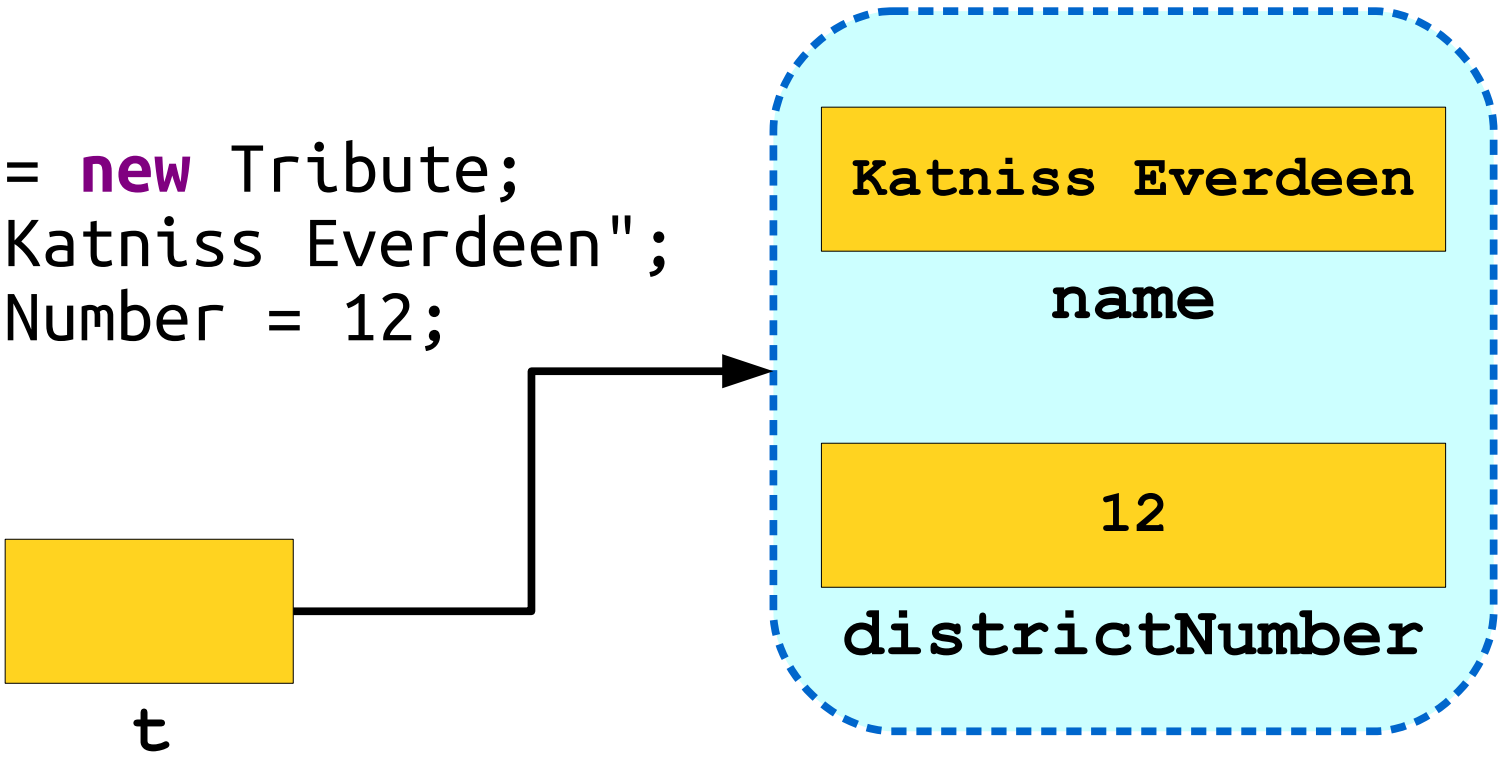
Dynamic Memory Allocation

```
struct Tribute {
    string name;
    int districtNumber;
};

Tribute* t = new Tribute;
t->name = "Katniss Everdeen";
t->districtNumber = 12;
```



Dynamic Memory Allocation

```
struct Tribute {
    string name;
    int districtNumber;
};

Tribute* t = new Tribute;
t->name = "Katniss Everdeen";
t->districtNumber = 12;
```



Cleaning Up

- As with dynamic arrays, you are responsible for cleaning up memory allocated with `new`.
- You can deallocate memory with the `delete` keyword:

```
delete ptr;
```

- This destroys the object pointed at by the given pointer, not the pointer itself.

Cleaning Up

- As with dynamic arrays, you are responsible for cleaning up memory allocated with `new`.
- You can deallocate memory with the `delete` keyword:

```
delete ptr;
```

- This destroys the object pointed at by the given pointer, not the pointer itself.

Cleaning Up

- As with dynamic arrays, you are responsible for cleaning up memory allocated with **`new`**.
- You can deallocate memory with the **`delete`** keyword:

```
delete ptr;
```

- This destroys the object pointed at by the given pointer, not the pointer itself.

Unfortunately…

- In C++, all of the following result in undefined behavior:
  - Deleting an object with `delete[]` that was allocated with `new`.
  - Deleting an object with `delete` that was allocated with `new[]`.
- Although it is not always an error, it is usually a Very Bad Idea to treat an array like a single object or vice-versa.

A Pointless Exercise

- When working with pointers, we sometimes wish to indicate that a pointer is not pointing to anything.
- In C++, you can set a pointer to `nullptr` to indicate that it is not pointing to an object:

```
ptr = nullptr;
```

- This is *not* the default value for pointers; by default, pointers default to a garbage value.
- In older C++ code (and the textbook!), you'll see people use `NULL` instead of `nullptr`. We strongly advise against using `NULL` and recommend you use `nullptr` instead.

And now... linked lists!

But first, some announcements!

Assignment 4

- Assignment 4 was due at the start of class today.
  - Using a late day? You can turn it in by ***Wednesday*** because Monday is a holiday.
  - We ***strongly advise*** against this – the exam expects that you know how to solve all the problems from the assignment and you'll need the time to study.
- Assignment 5 will go out on Wednesday of next week.

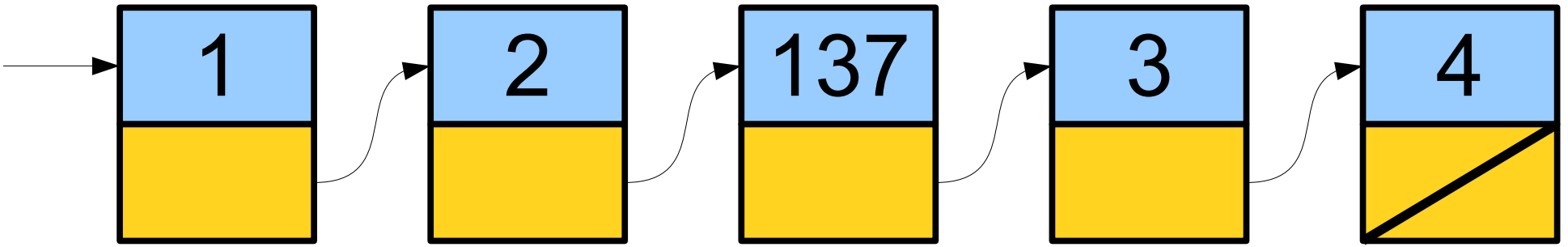
Midterm Logistics

- Midterm is next Tuesday from 7PM – 10PM. Locations are divvied up by last (family) name:
  - **`Abb – Lam`**: Go to Hewlett 200.
  - **`Lee – Nic`**: Go to Hewlett 201.
  - **`Ntu – Zhu`**: Go to Cubberly Auditorium.
- Space is tight, so please go to your assigned exam room.
- You get a double-sided, 8.5” × 11” sheet of notes with you when you take the exam.

Back to CS106B!

Linked List Cells

- A linked list is a chain of ***cells***.
- Each cell contains two pieces of information:
  - Some piece of data that is stored in the sequence, and
  - A ***link*** to the next cell in the list.
- We can traverse the list by starting at the first cell and repeatedly following its link.



Representing a Cell

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

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

Representing a Cell

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

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

Representing a Cell

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

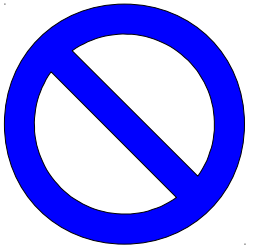
```
struct Cell {
    string value;
    Cell* next;
};
```

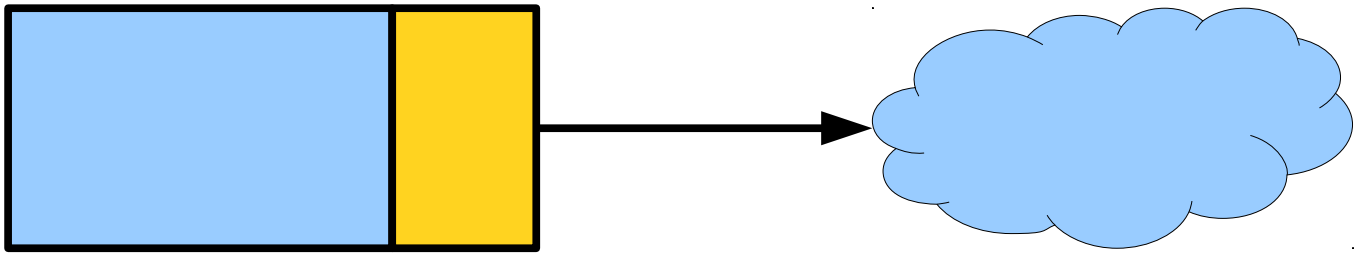
- ***The structure is defined recursively!***

Building Linked Lists

A Linked List is Either…

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



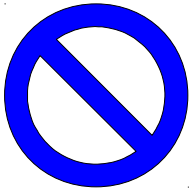


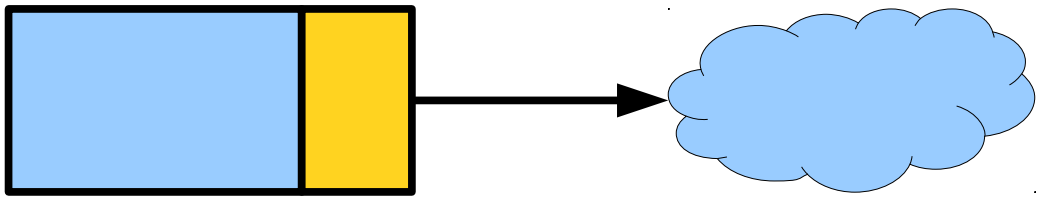
a single linked list cell that points…

… at another linked list.

A Linked List is Either…

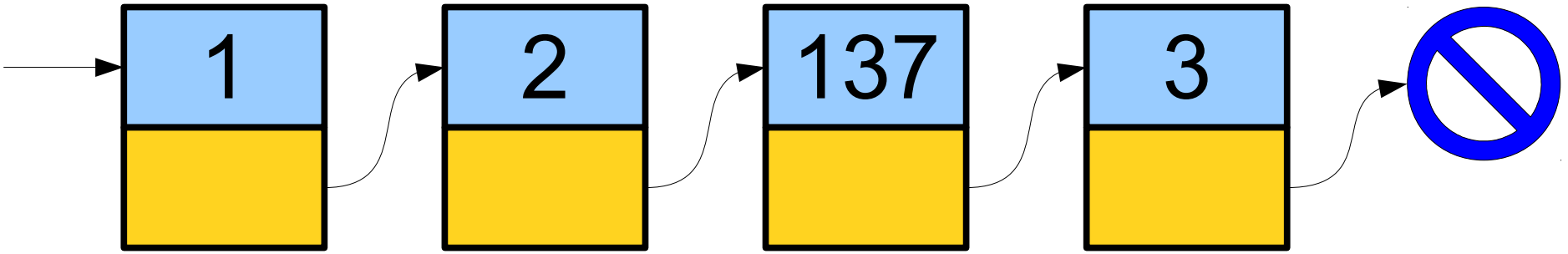
…an empty list, represented by `nullptr`, or…





a single linked list cell that points…

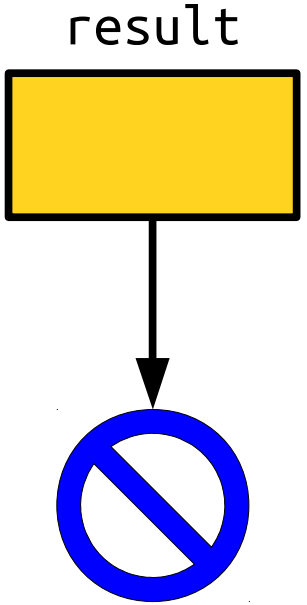
… at another linked list.



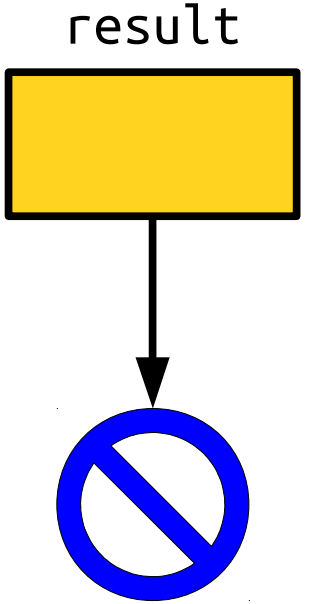
```
Cell* result = nullptr;
while (true) {









}
return result;
```

```
Cell* result = nullptr;
while (true) {




}
return result;
```

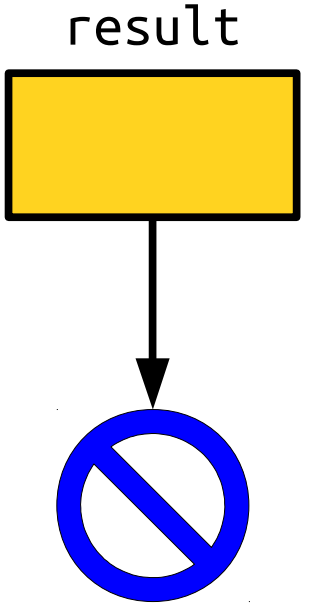
```
Cell* result = nullptr;
while (true) {




}
return result;
```



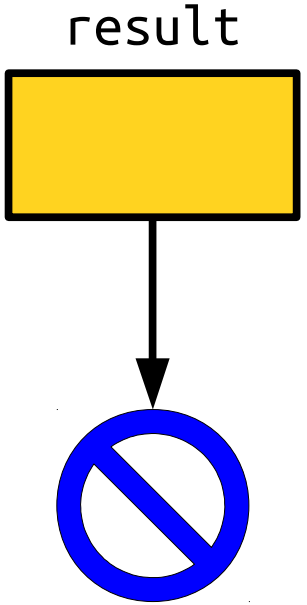
```
Cell* result = nullptr;
while (true) {




}
return result;
```



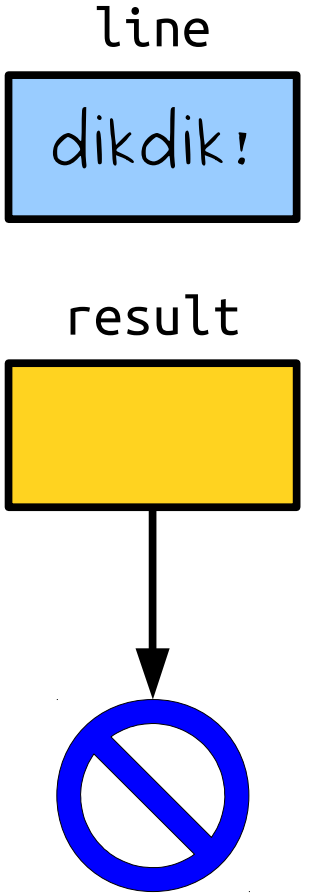
```
Cell* result = nullptr;
while (true) {
    string line = getLine("Next entry? ");
    if (line == "") break;




}
return result;
```



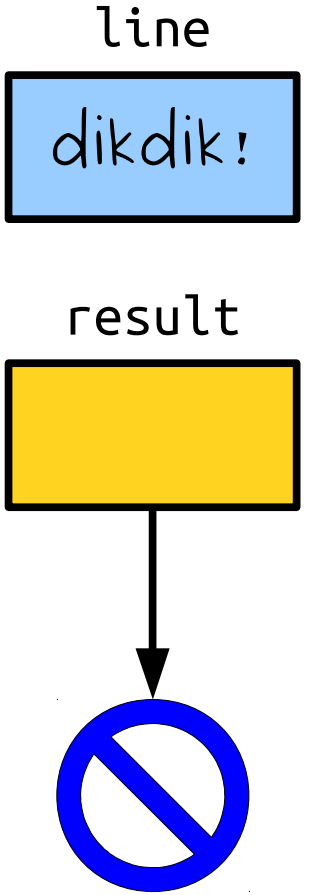
```
Cell* result = nullptr;
while (true) {
    string line = getLine("Next entry? ");
    if (line == "") break;




}
return result;
```



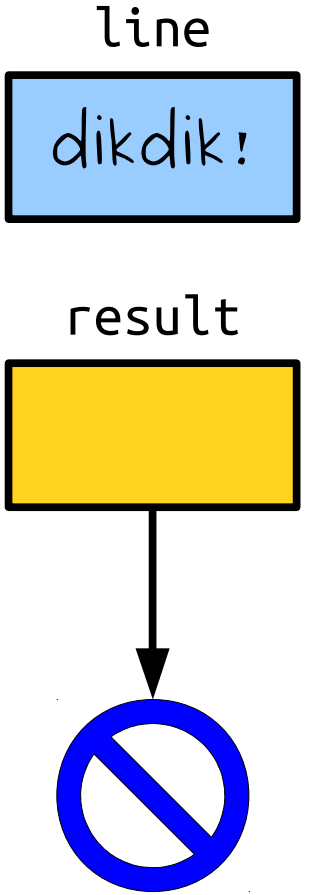
```
Cell* result = nullptr;
while (true) {
    string line = getLine("Next entry? ");
    if (line == "") break;




}
return result;
```



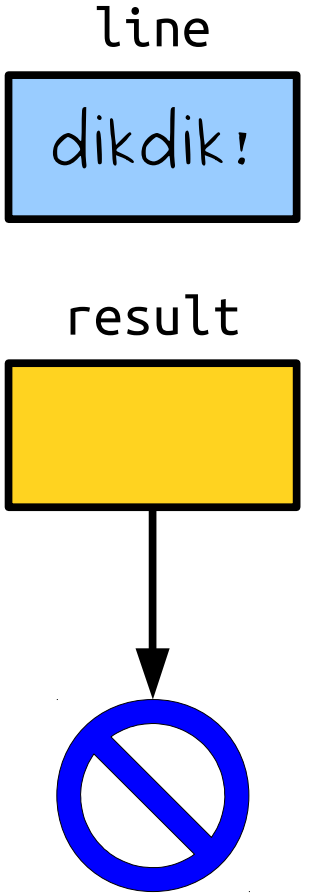
```
Cell* result = nullptr;
while (true) {
    string line = getLine("Next entry? ");
    if (line == "") break;
}
return result;
```



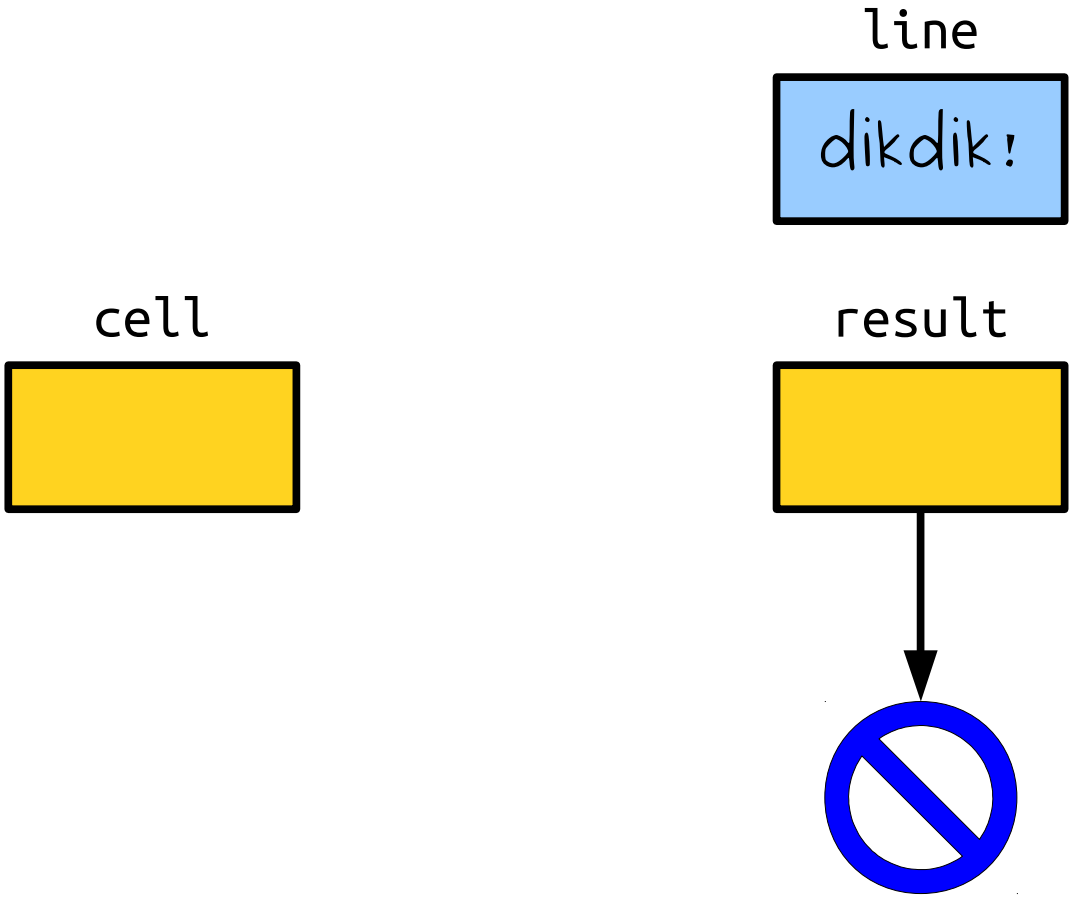
```
Cell* result = nullptr;
while (true) {
    string line = getLine("Next entry? ");
    if (line == "") break;

    Cell* cell = new Cell;




}
return result;
```



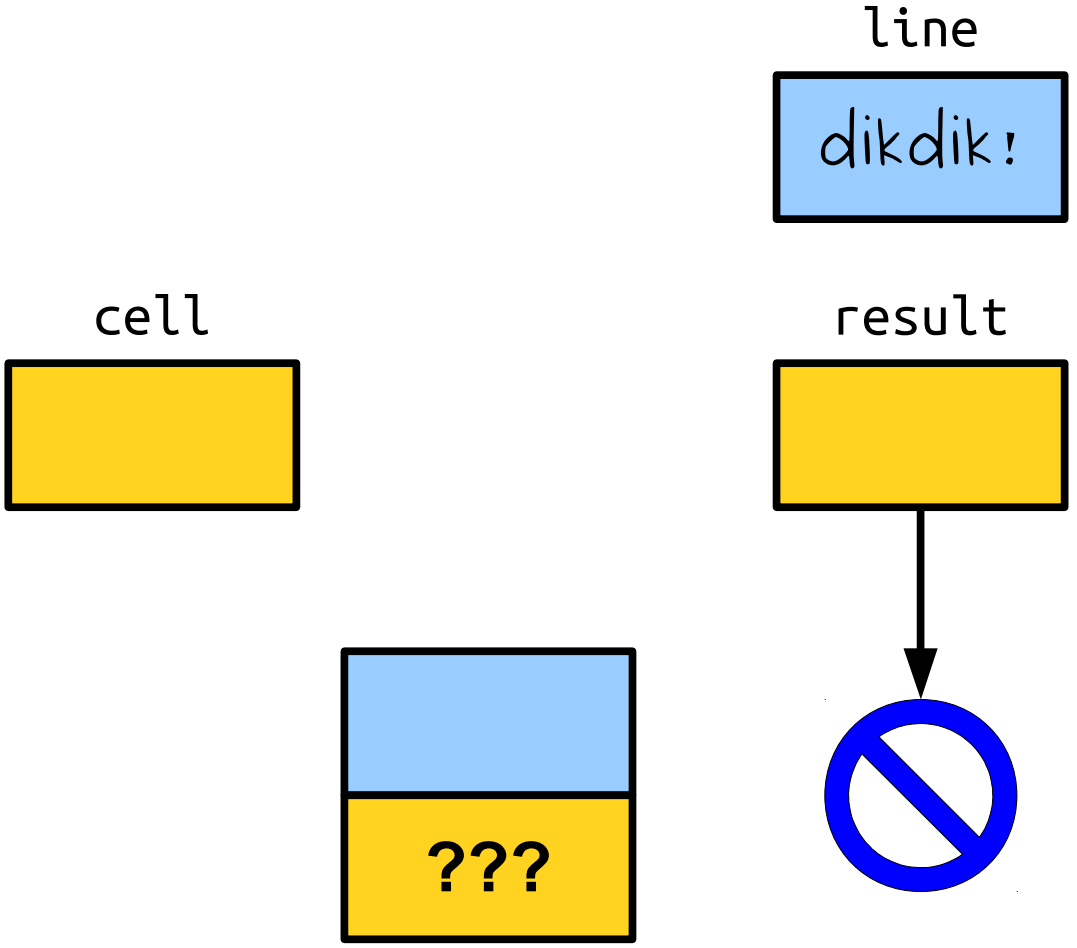
```
Cell* result = nullptr;
while (true) {
    string line = getLine("Next entry? ");
    if (line == "") break;

    Cell* cell = new Cell;



}
return result;
```



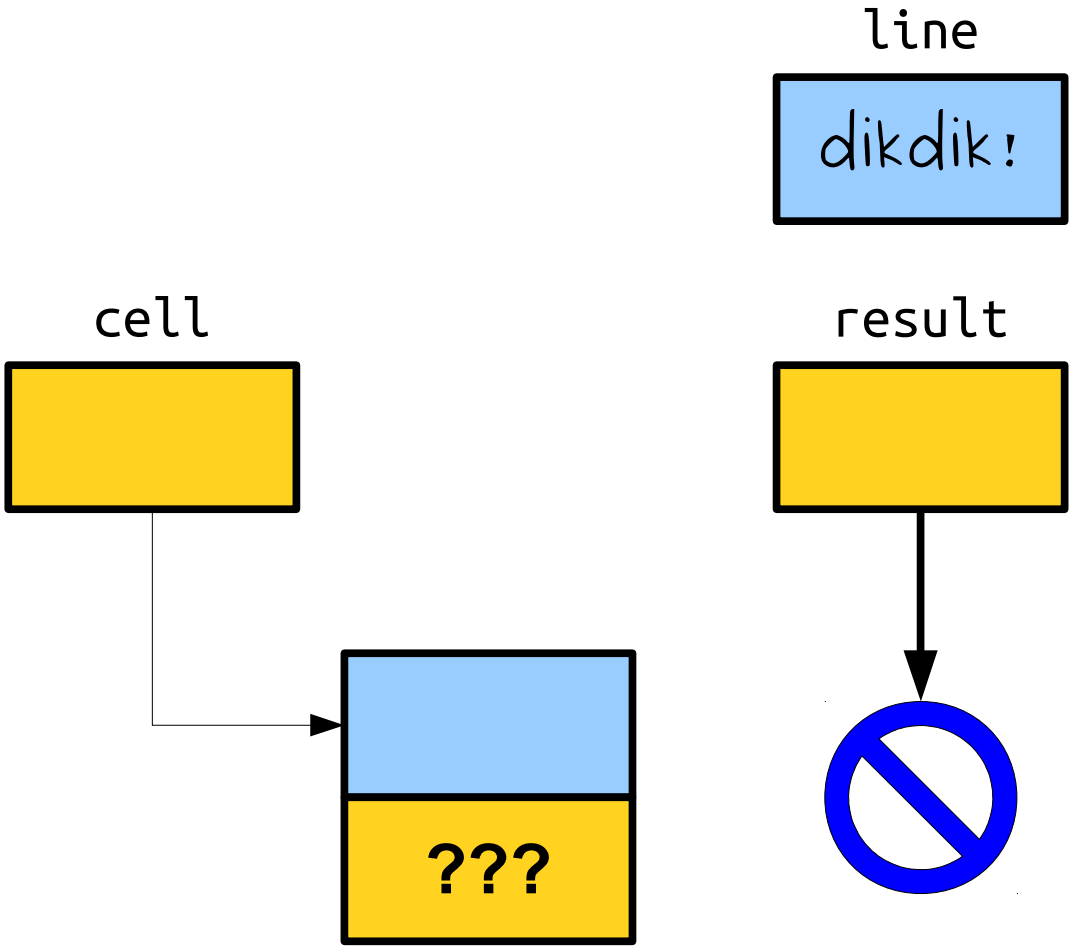
```
Cell* result = nullptr;
while (true) {
    string line = getLine("Next entry? ");
    if (line == "") break;

    Cell* cell = new Cell;



}
return result;
```



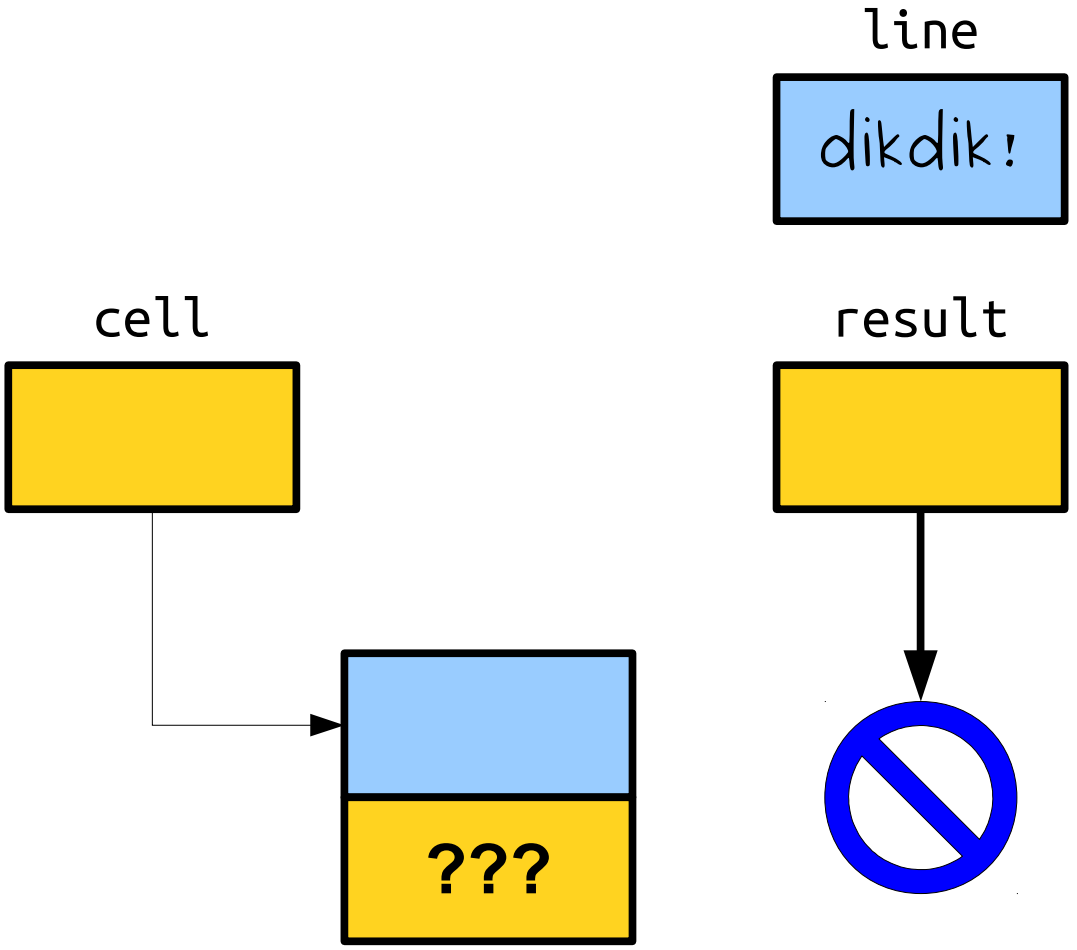
```
Cell* result = nullptr;
while (true) {
    string line = getLine("Next entry? ");
    if (line == "") break;

    Cell* cell = new Cell;



}
return result;
```



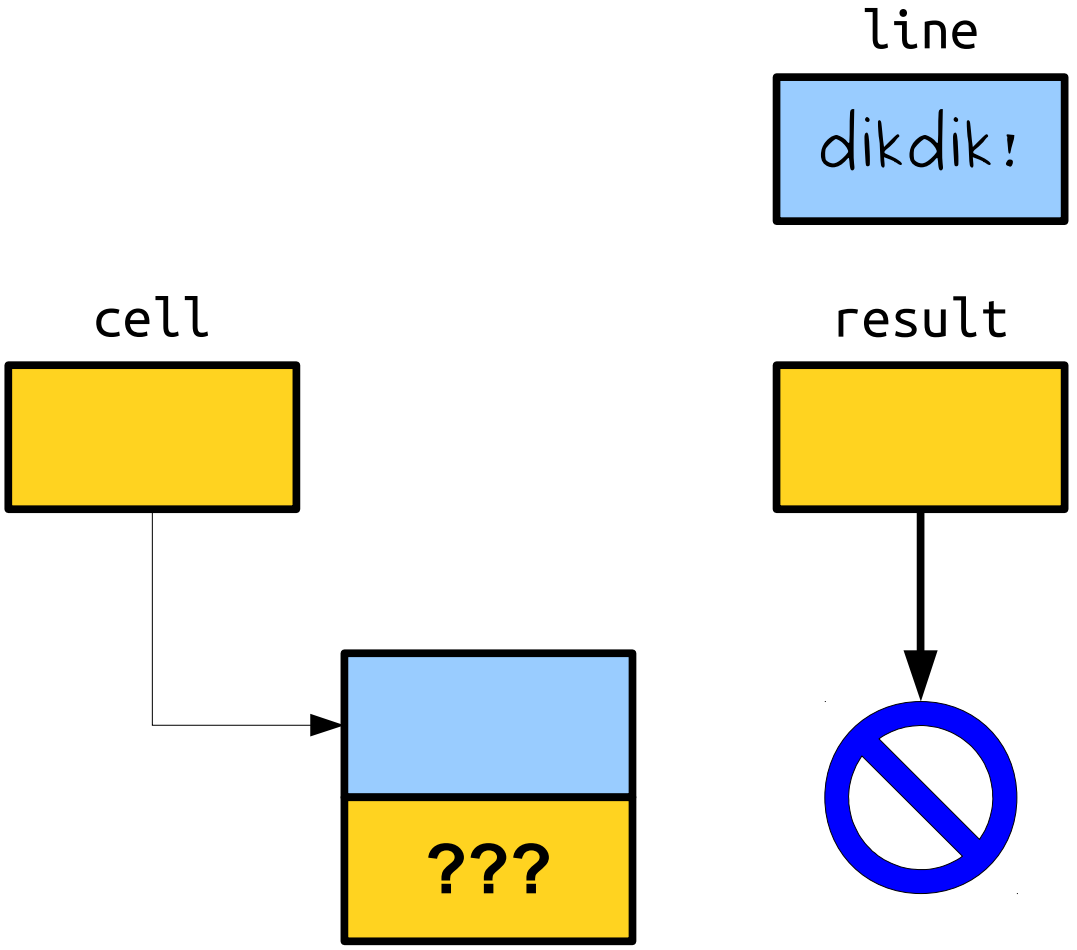
```
Cell* result = nullptr;
while (true) {
    string line = getLine("Next entry? ");
    if (line == "") break;

    Cell* cell = new Cell;



}
return result;
```



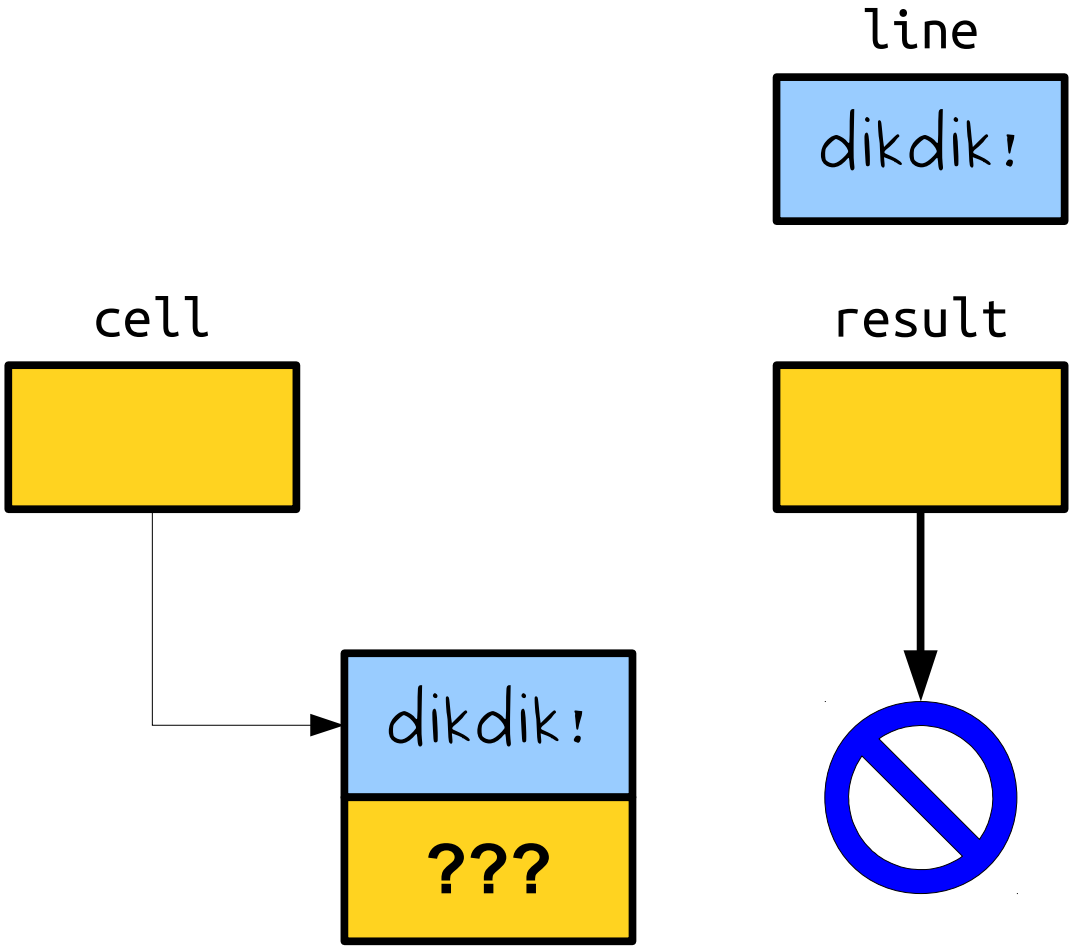
```
Cell* result = nullptr;
while (true) {
    string line = getLine("Next entry? ");
    if (line == "") break;

    Cell* cell = new Cell;
    cell->value = line;



}
return result;
```



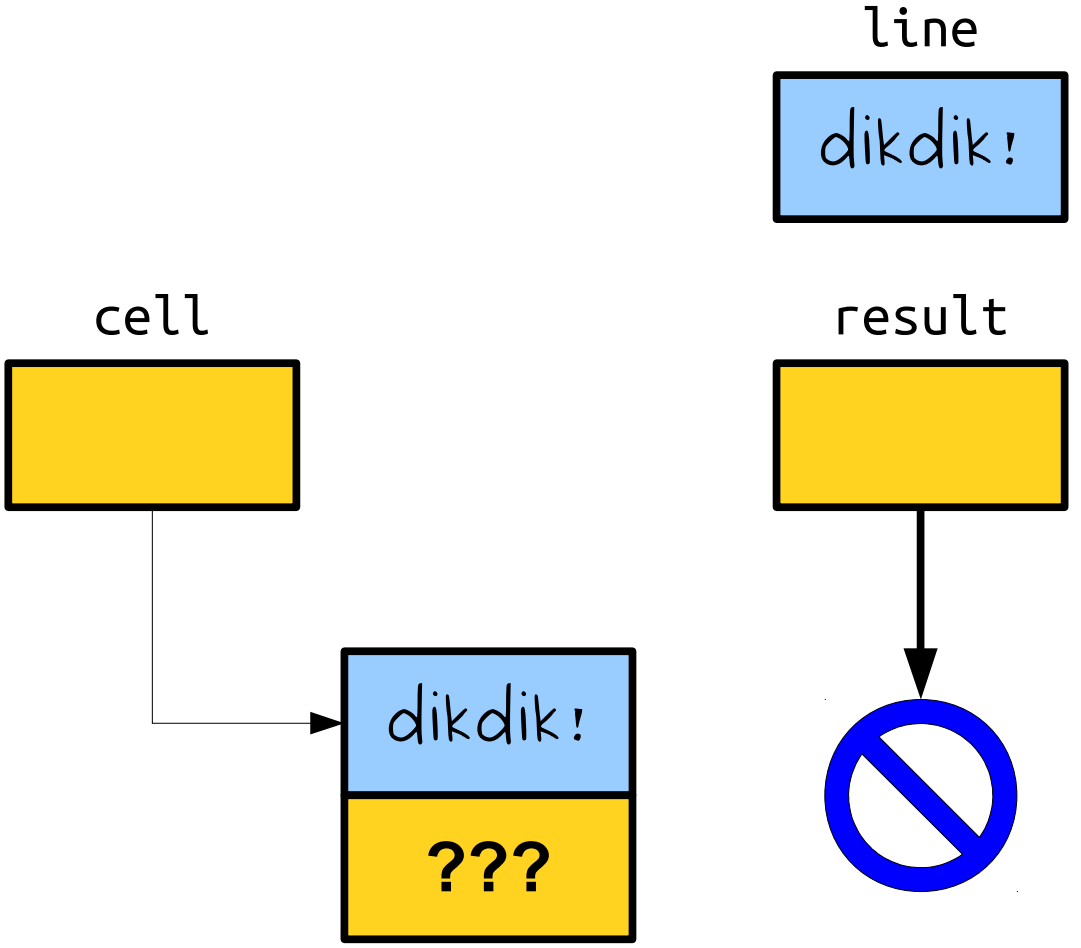
```
Cell* result = nullptr;
while (true) {
    string line = getLine("Next entry? ");
    if (line == "") break;

    Cell* cell = new Cell;
    cell->value = line;



}
return result;
```



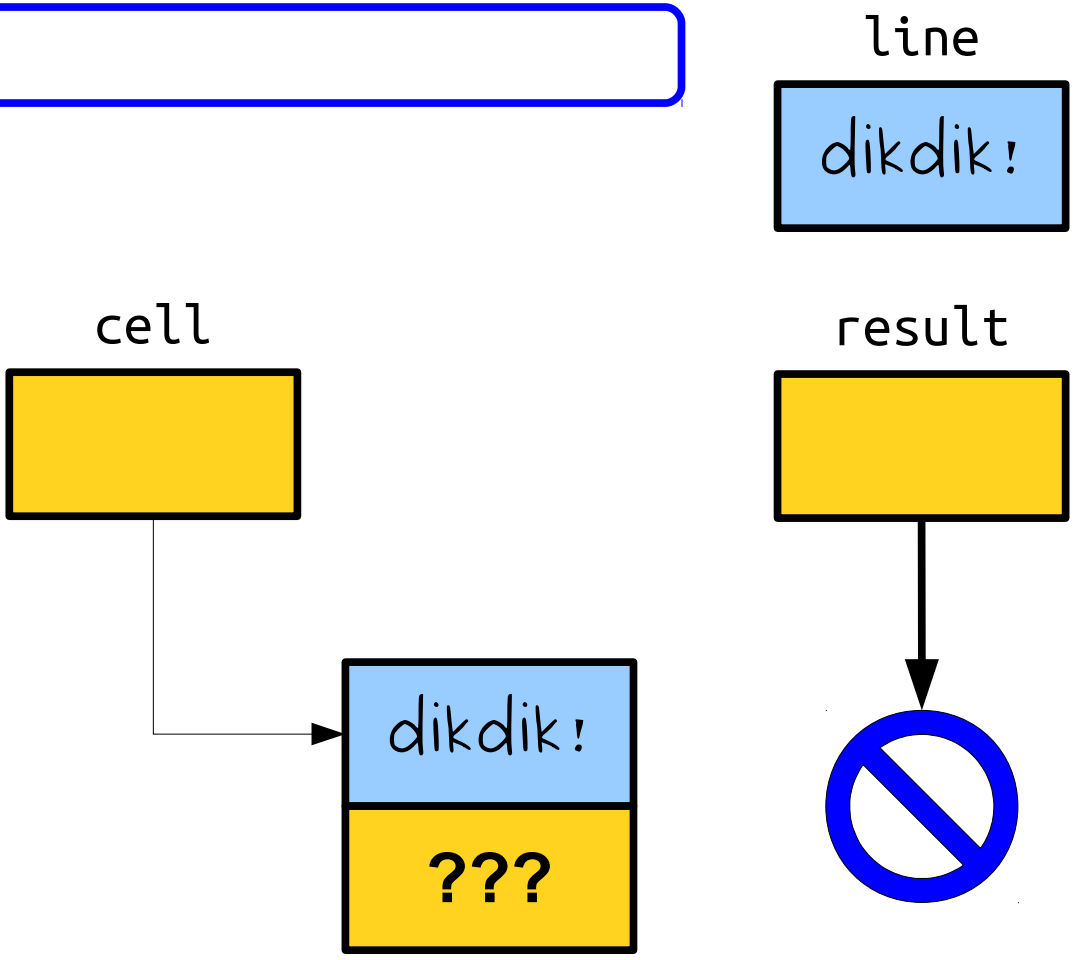
```
Cell* result = nullptr;
while (true) {
    string line = getLine("Next entry? ");
    if (line == "") break;

    Cell* cell = new Cell;
    cell->value = line;



}
return result;
```



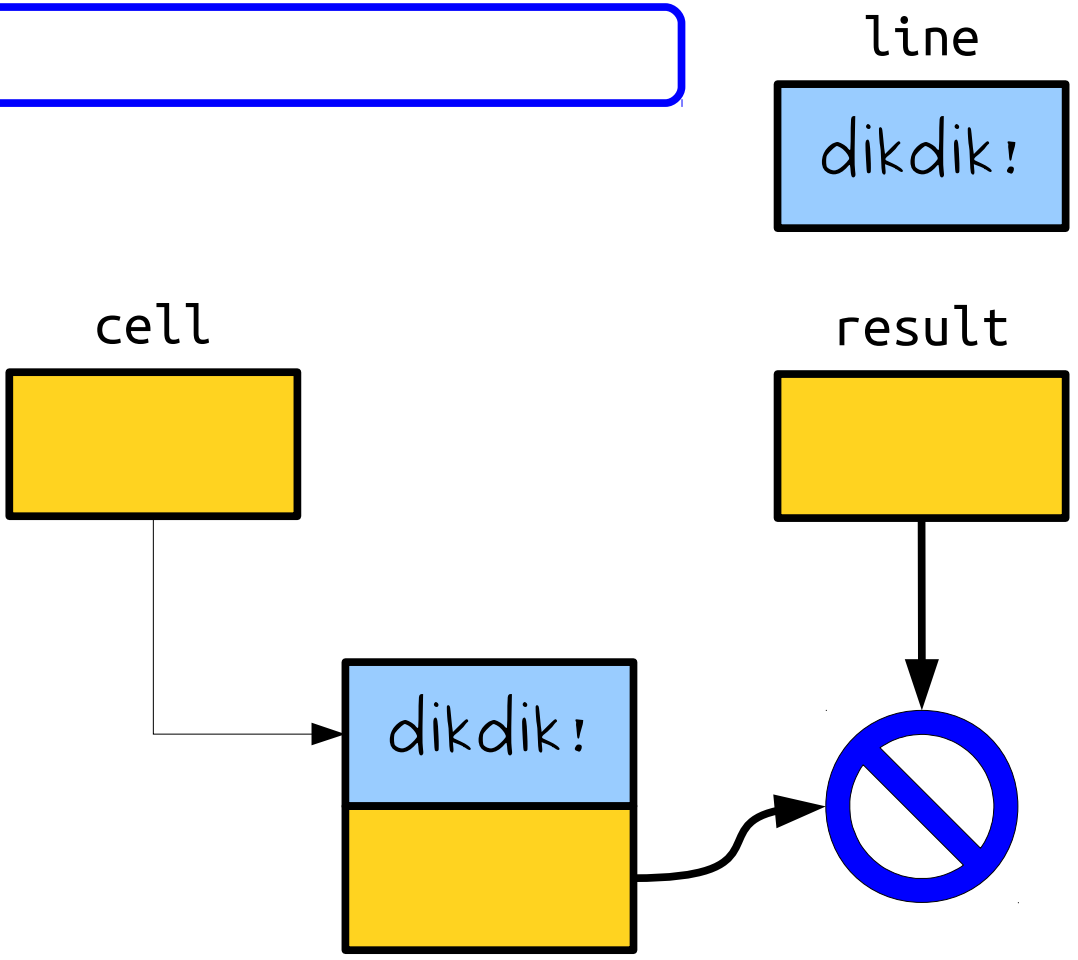
```
Cell* result = nullptr;
while (true) {
    string line = getLine("Next entry? ");
    if (line == "") break;

    Cell* cell = new Cell;
    cell->value = line;

    cell->next = result;

}
return result;
```



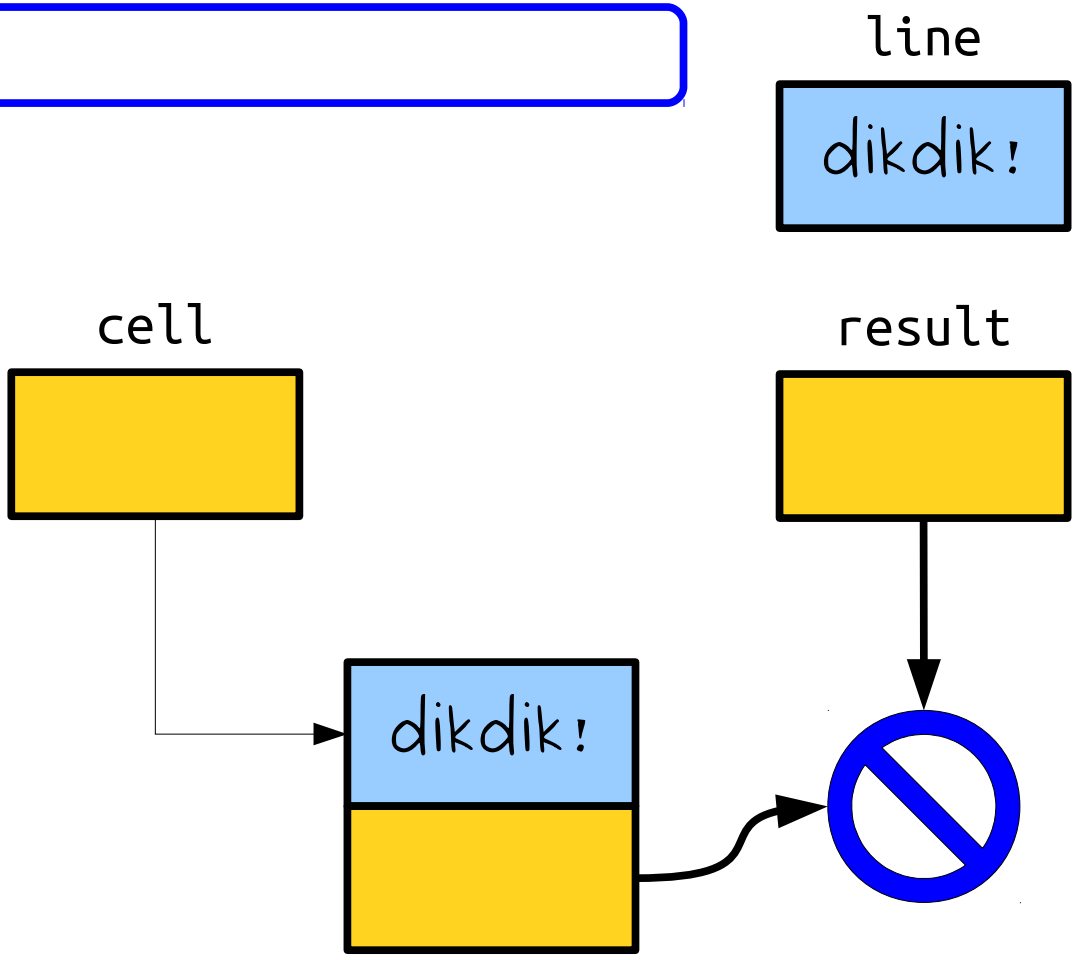
```
Cell* result = nullptr;
while (true) {
    string line = getLine("Next entry? ");
    if (line == "") break;

    Cell* cell = new Cell;
    cell->value = line;

    cell->next = result;

}
return result;
```



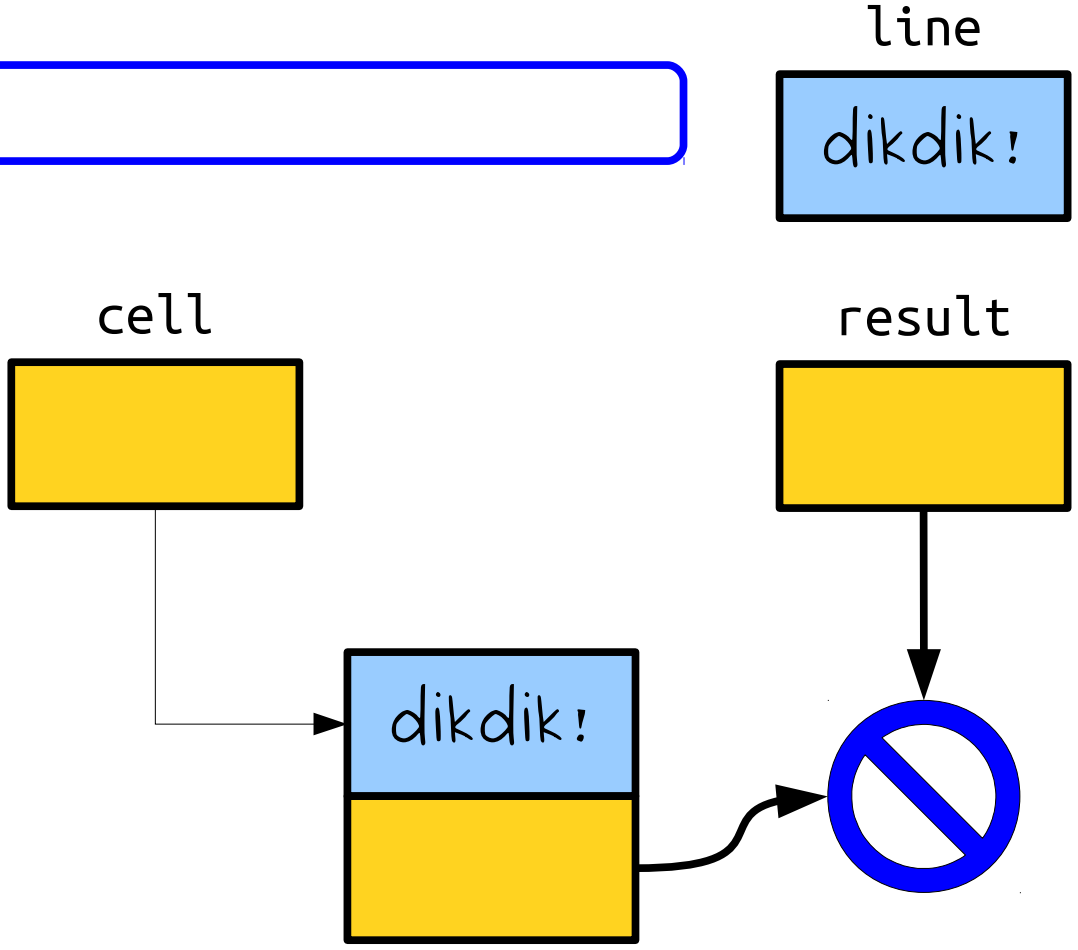
```
Cell* result = nullptr;
while (true) {
    string line = getLine("Next entry? ");
    if (line == "") break;

    Cell* cell = new Cell;
    cell->value = line;

    cell->next = result;

}
return result;
```



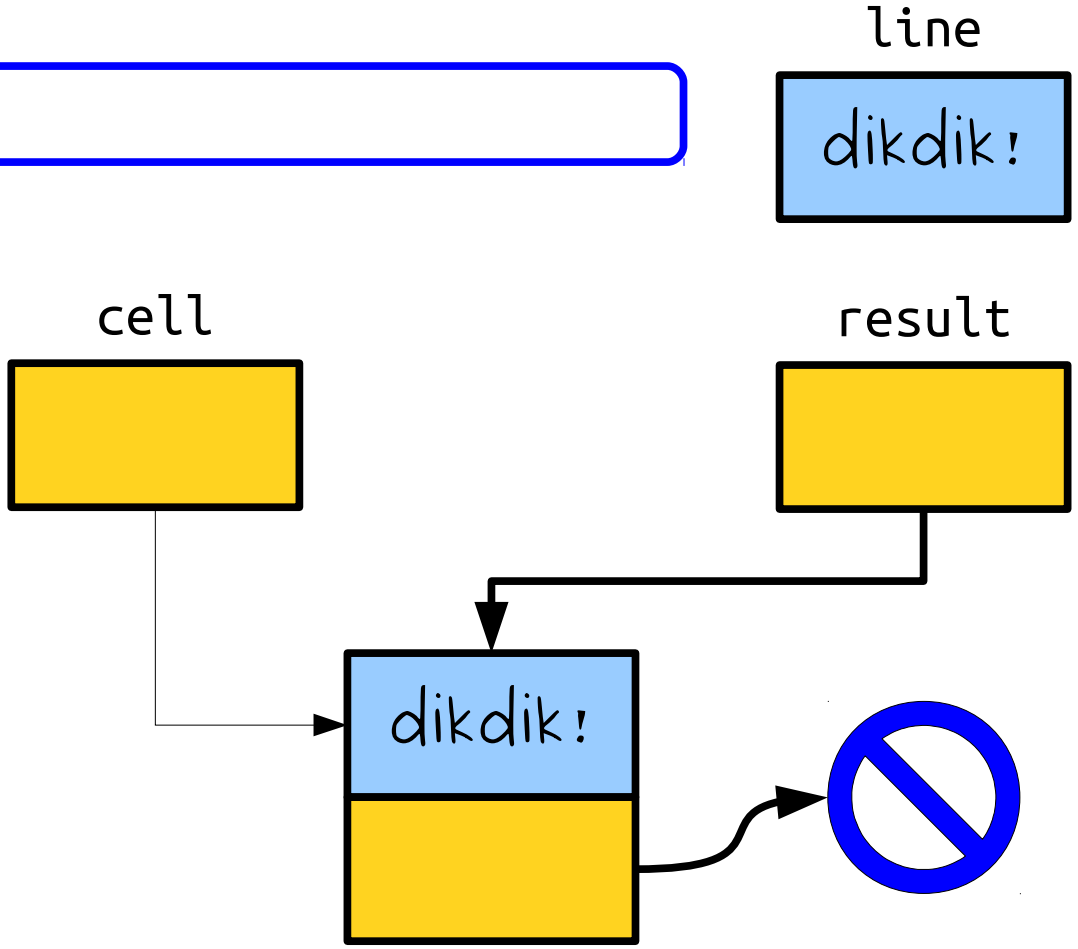
```
Cell* result = nullptr;
while (true) {
    string line = getLine("Next entry? ");
    if (line == "") break;

    Cell* cell = new Cell;
    cell->value = line;

    cell->next = result;
    result = cell;
}
return result;
```



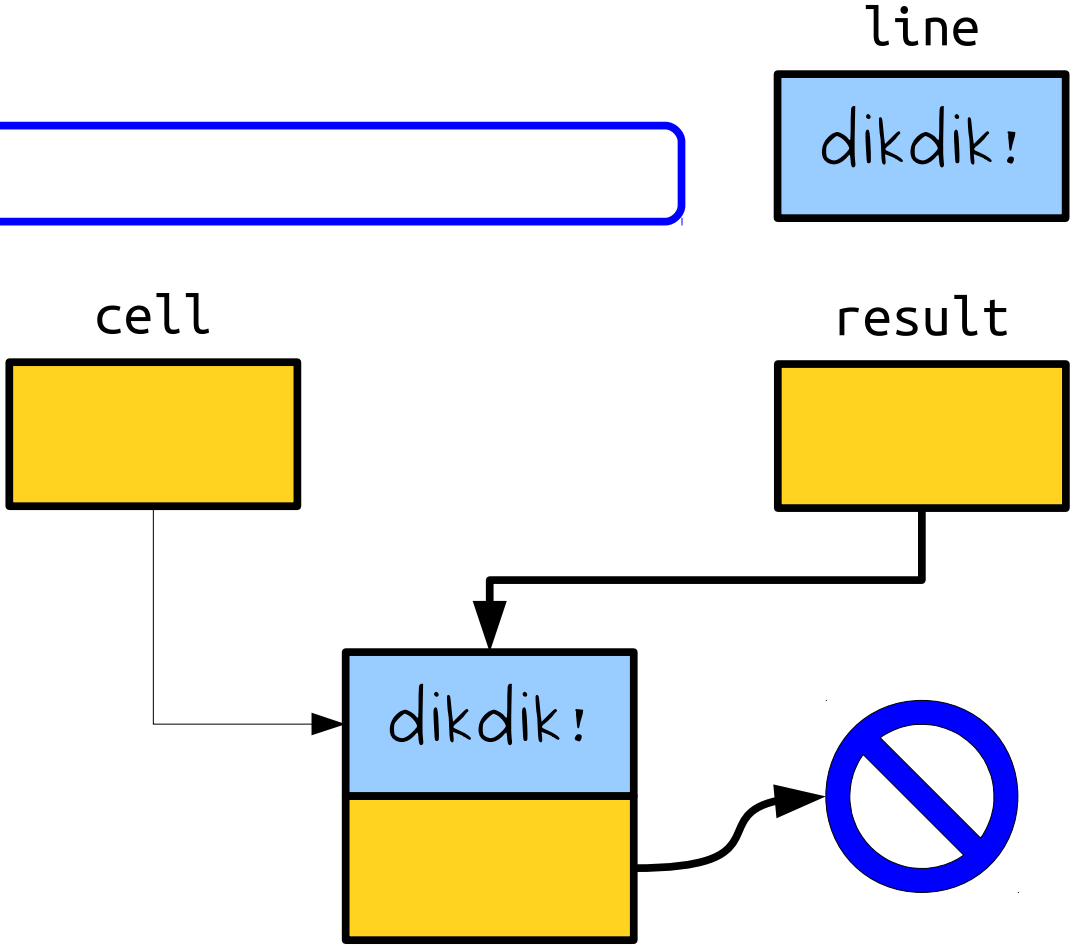
```
Cell* result = nullptr;
while (true) {
    string line = getLine("Next entry? ");
    if (line == "") break;

    Cell* cell = new Cell;
    cell->value = line;

    cell->next = result;
    result = cell;
}
return result;
```



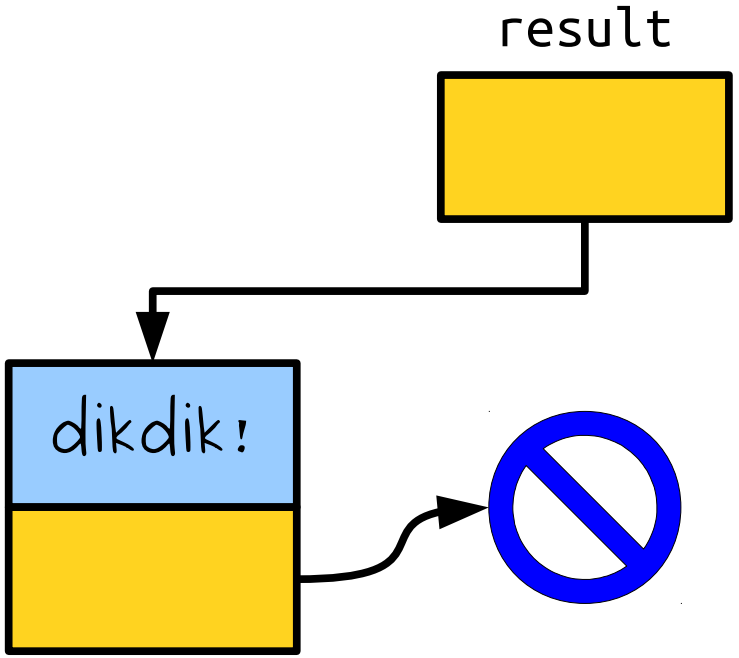
```
Cell* result = nullptr;
while (true) {
    string line = getLine("Next entry? ");
    if (line == "") break;

    Cell* cell = new Cell;
    cell->value = line;

    cell->next = result;
    result = cell;
}
return result;
```



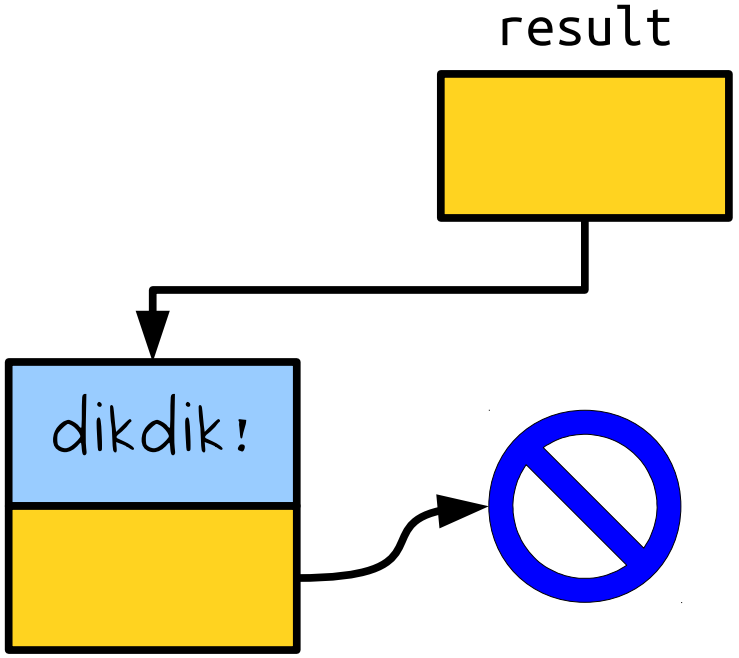
```
Cell* result = nullptr;
while (true) {
    string line = getLine("Next entry? ");
    if (line == "") break;

    Cell* cell = new Cell;
    cell->value = line;

    cell->next = result;
    result = cell;
}
return result;
```



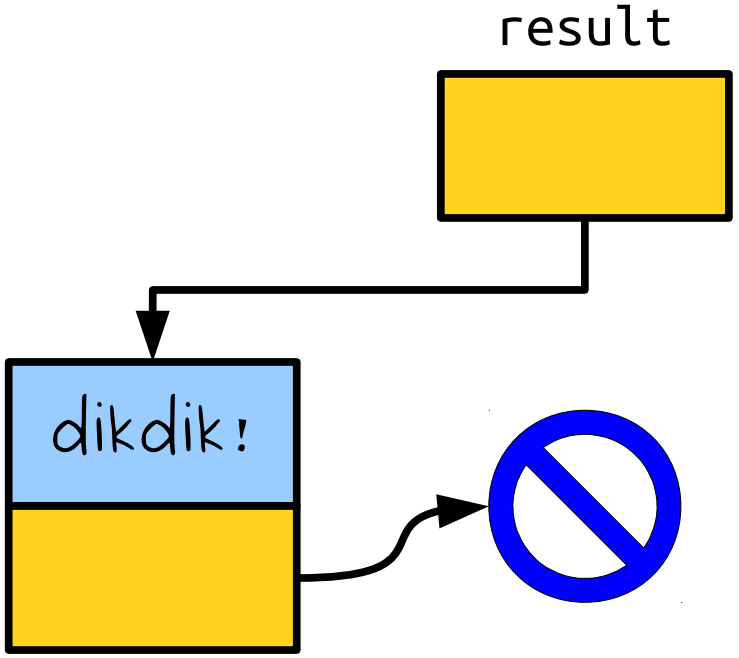
```
Cell* result = nullptr;
while (true) {
    string line = getLine("Next entry? ");
    if (line == "") break;

    Cell* cell = new Cell;
    cell->value = line;

    cell->next = result;
    result = cell;
}
return result;
```



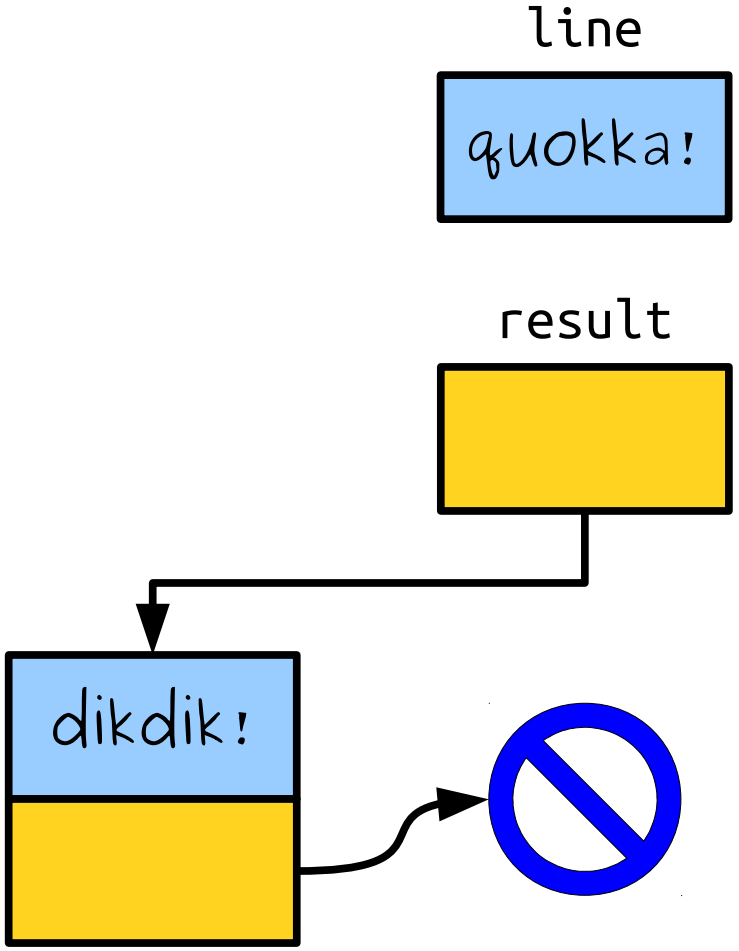
```
Cell* result = nullptr;
while (true) {
    string line = getLine("Next entry? ");
    if (line == "") break;

    Cell* cell = new Cell;
    cell->value = line;

    cell->next = result;
    result = cell;
}
return result;
```



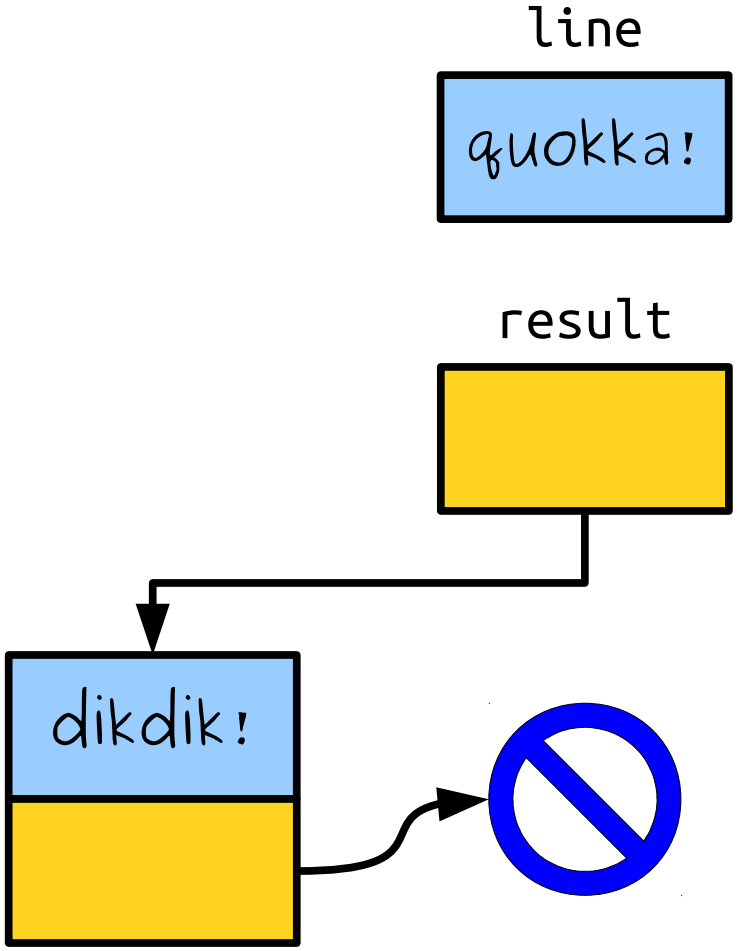
```
Cell* result = nullptr;
while (true) {
    string line = getLine("Next entry? ");
    if (line == "") break;

    Cell* cell = new Cell;
    cell->value = line;

    cell->next = result;
    result = cell;
}
return result;
```



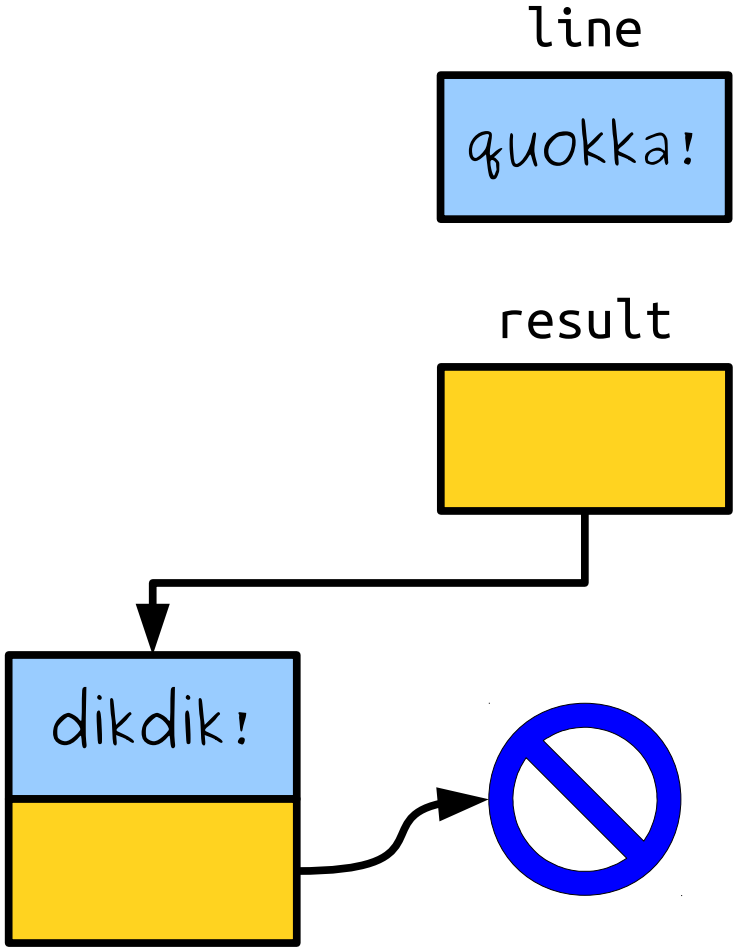
```
Cell* result = nullptr;
while (true) {
    string line = getLine("Next entry? ");
    if (line == "") break;

    Cell* cell = new Cell;
    cell->value = line;

    cell->next = result;
    result = cell;
}
return result;
```



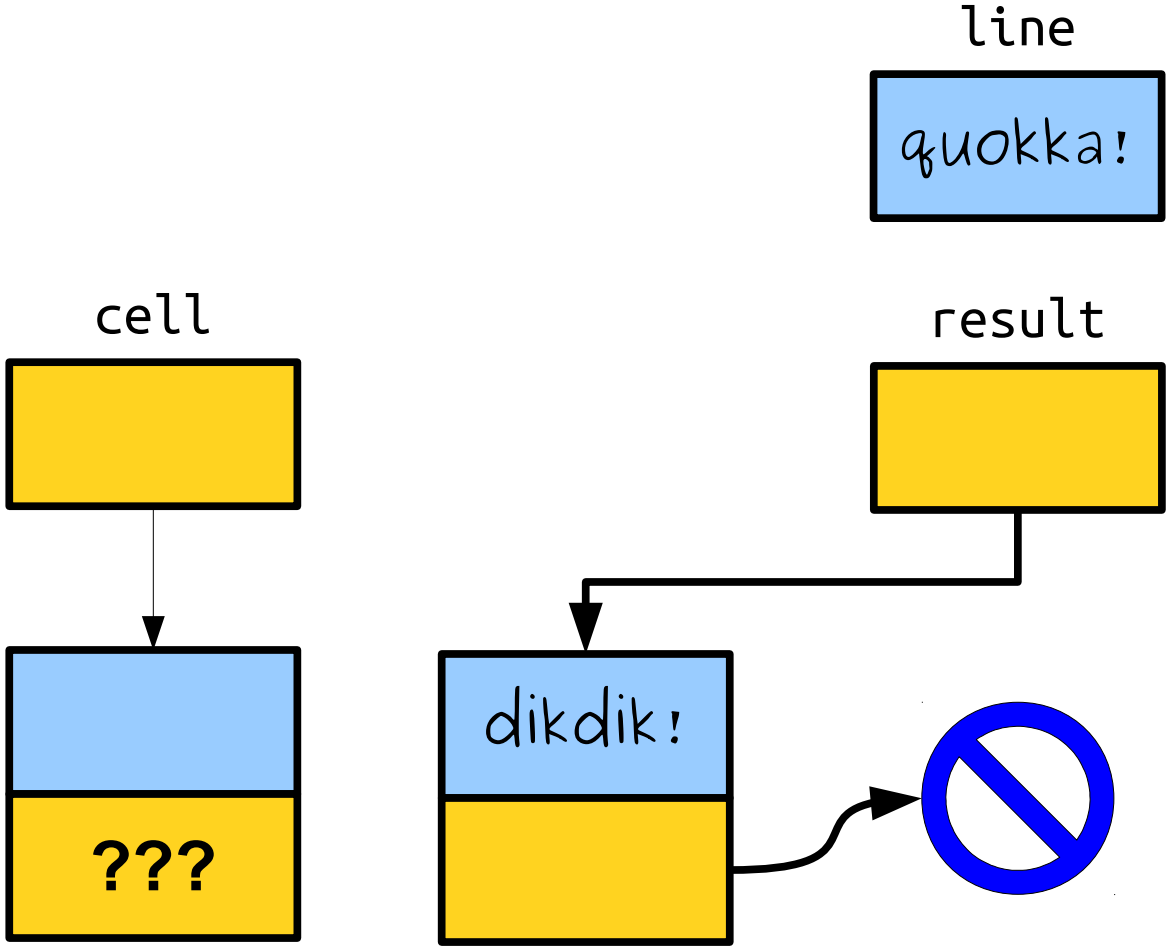
```
Cell* result = nullptr;
while (true) {
    string line = getLine("Next entry? ");
    if (line == "") break;

    Cell* cell = new Cell;
    cell->value = line;

    cell->next = result;
    result = cell;
}
return result;
```



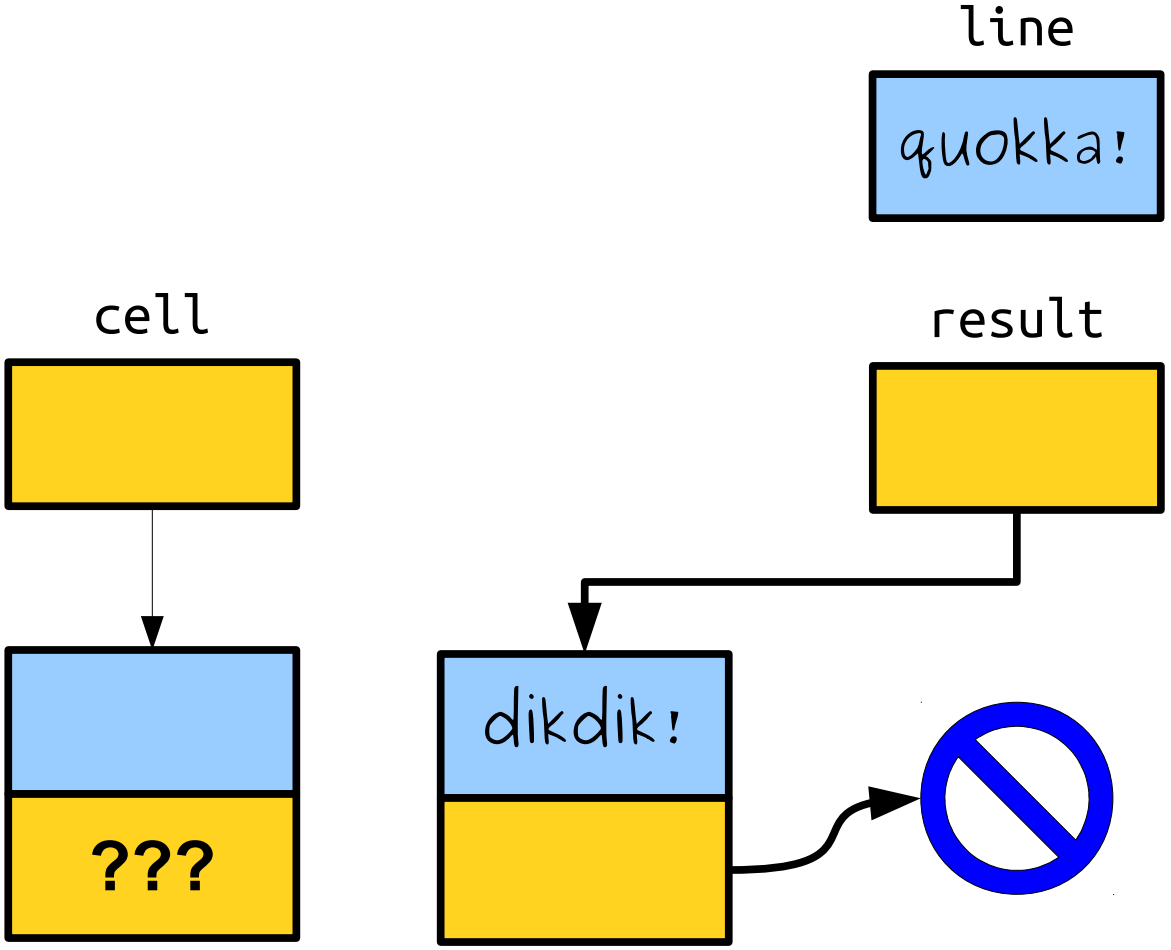
```
Cell* result = nullptr;
while (true) {
    string line = getLine("Next entry? ");
    if (line == "") break;

    Cell* cell = new Cell;
    cell->value = line;

    cell->next = result;
    result = cell;
}
return result;
```



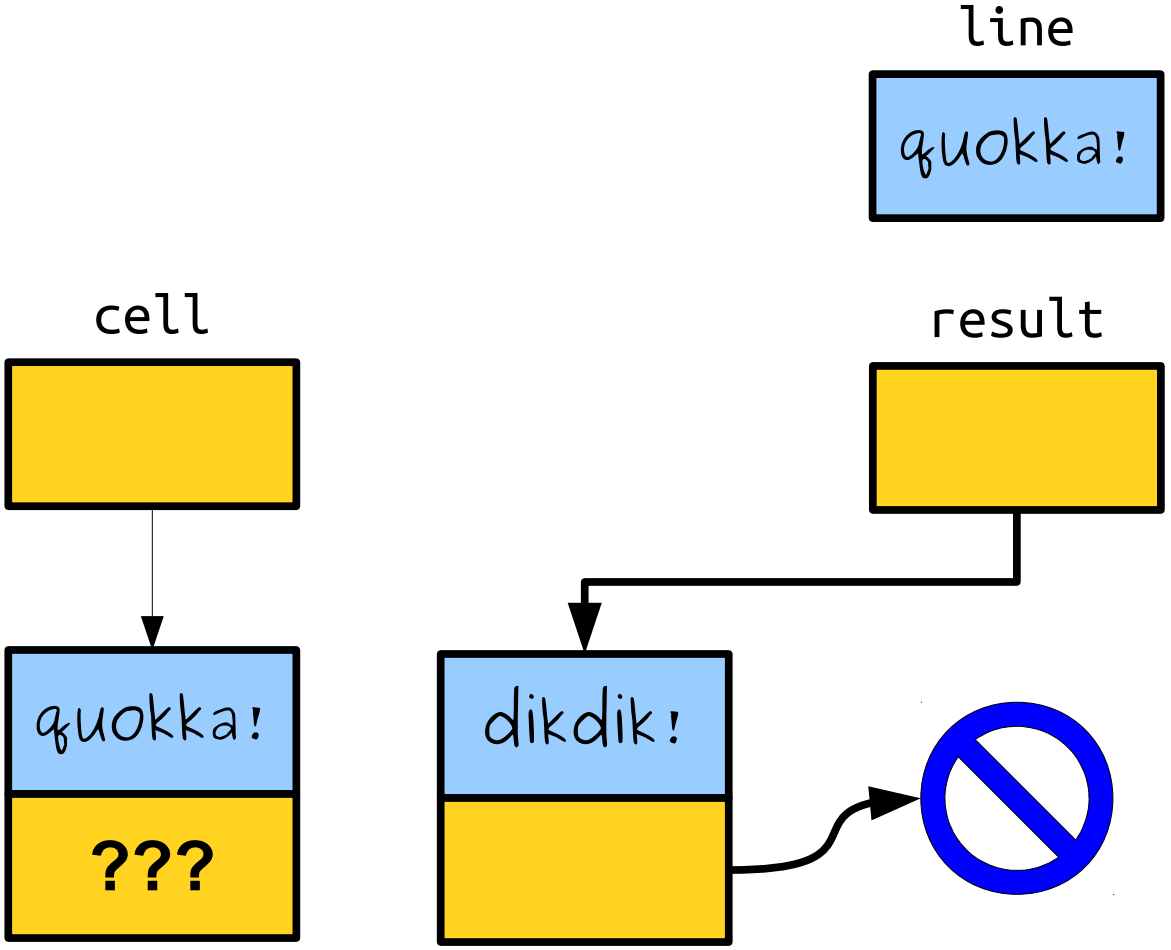
```
Cell* result = nullptr;
while (true) {
    string line = getLine("Next entry? ");
    if (line == "") break;

    Cell* cell = new Cell;
    cell->value = line;

    cell->next = result;
    result = cell;
}
return result;
```



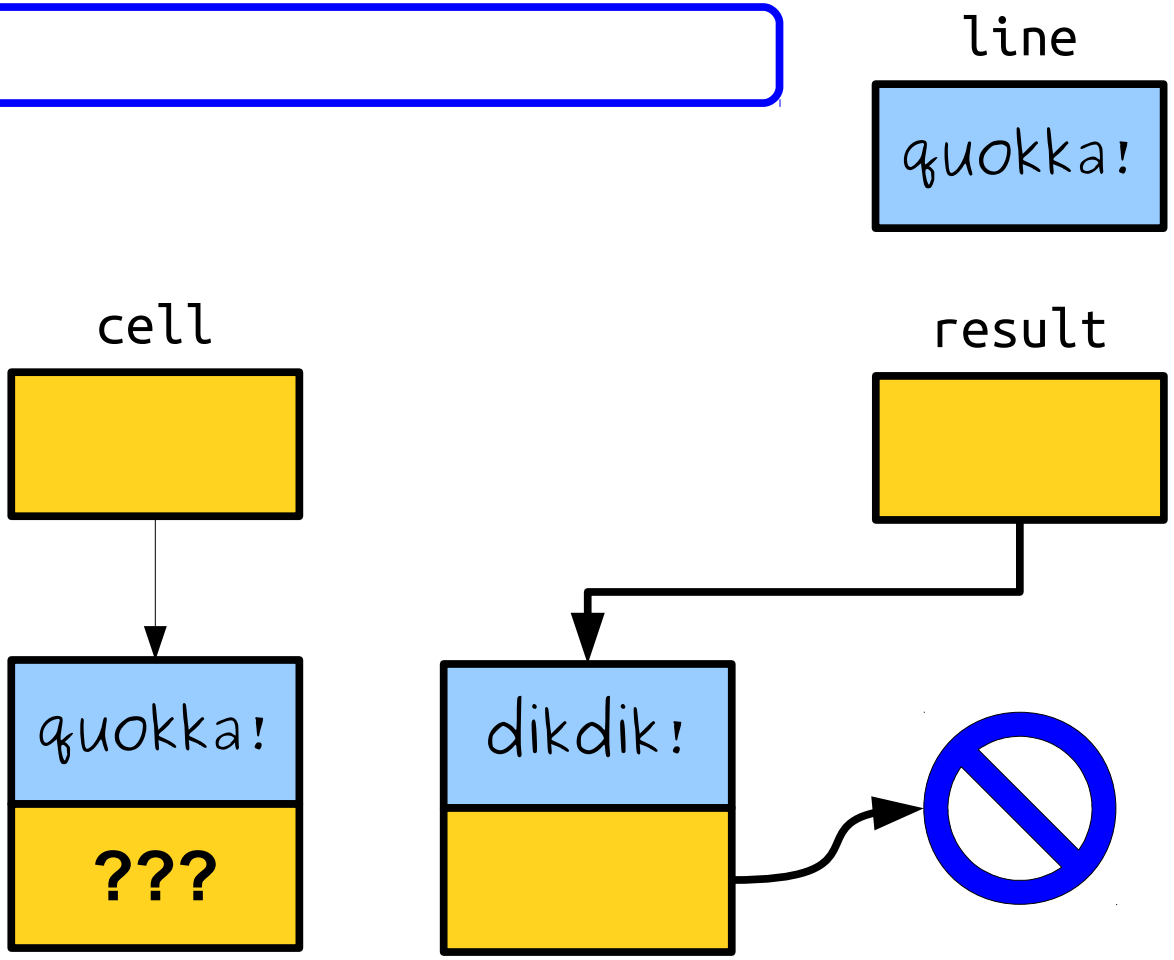
```
Cell* result = nullptr;
while (true) {
    string line = getLine("Next entry? ");
    if (line == "") break;

    Cell* cell = new Cell;
    cell->value = line;

    cell->next = result;
    result = cell;
}
return result;
```



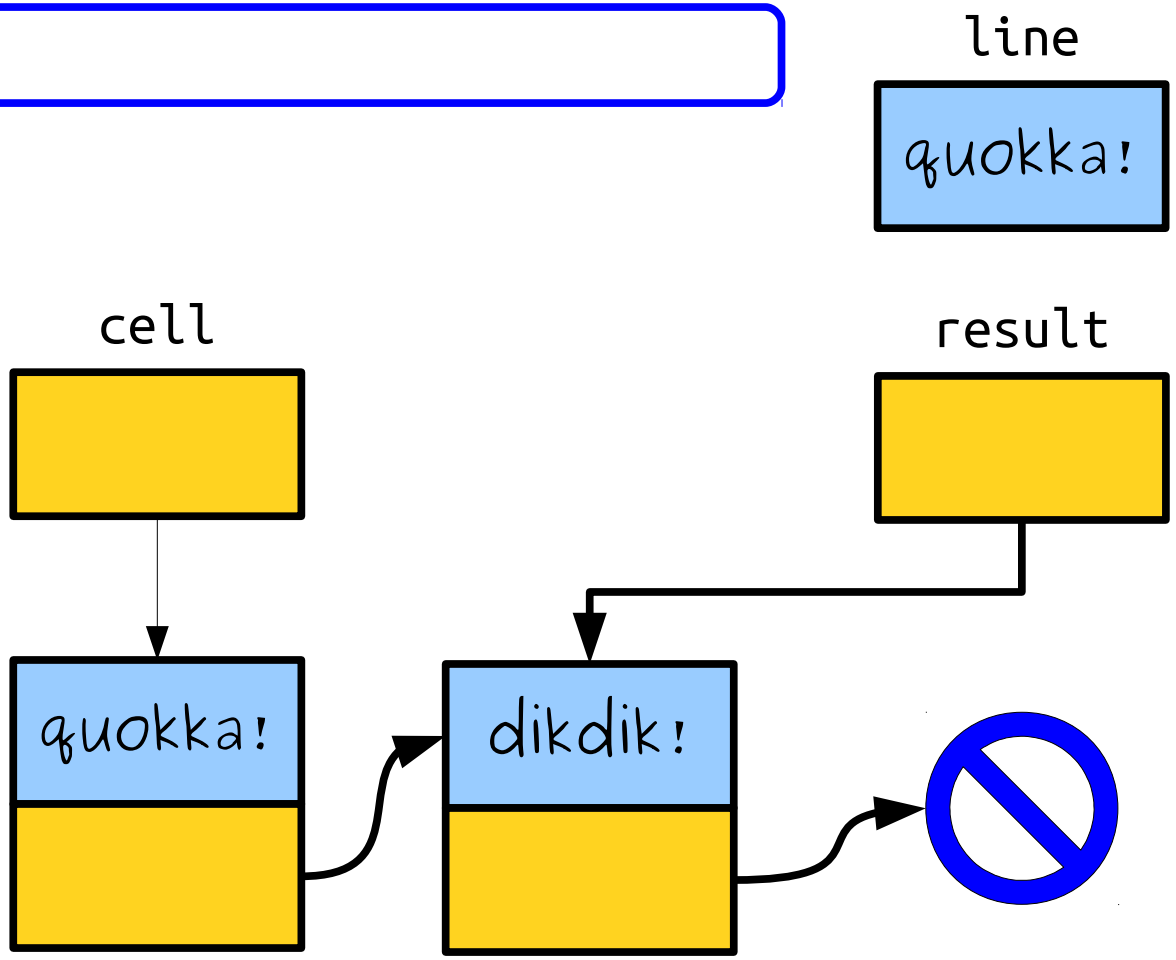
```
Cell* result = nullptr;
while (true) {
    string line = getLine("Next entry? ");
    if (line == "") break;

    Cell* cell = new Cell;
    cell->value = line;

    cell->next = result;
    result = cell;
}
return result;
```



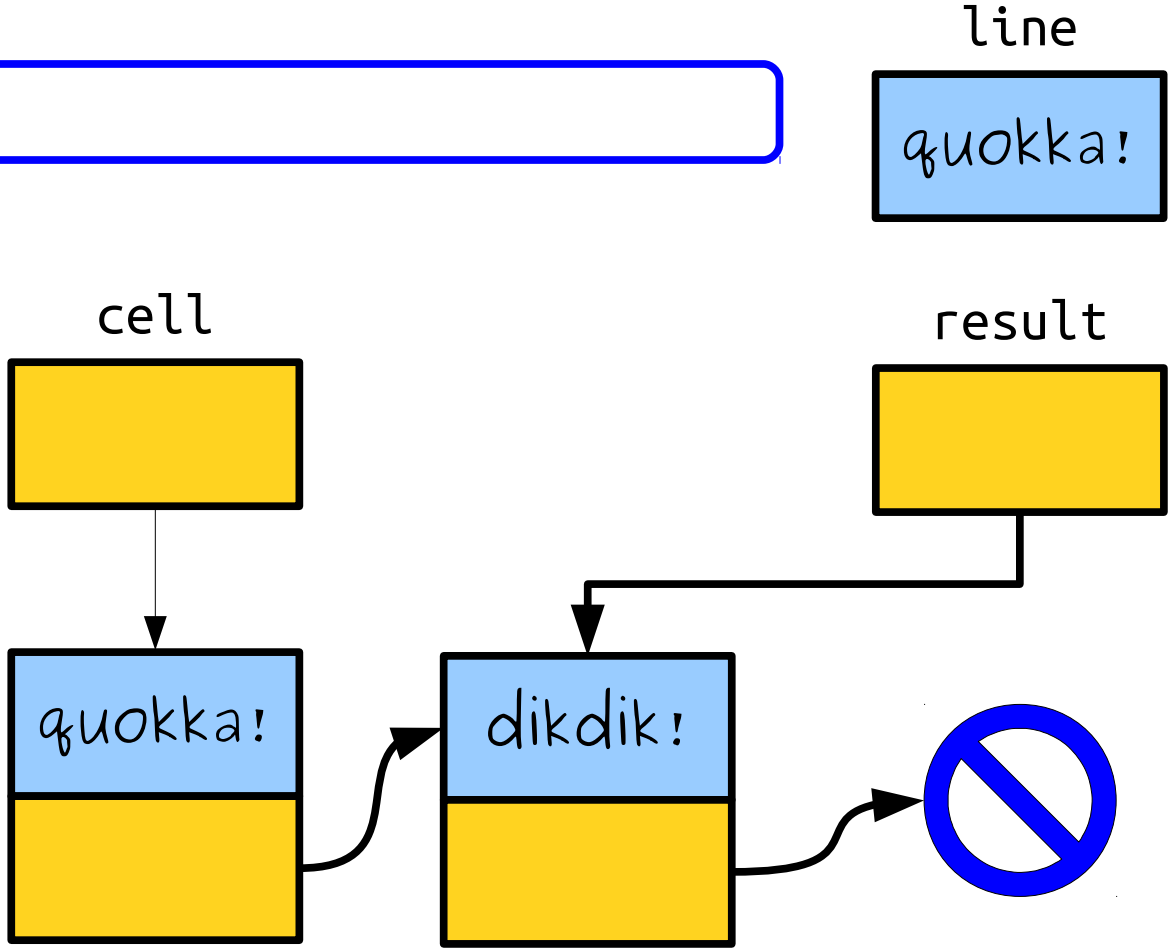
```
Cell* result = nullptr;
while (true) {
    string line = getLine("Next entry? ");
    if (line == "") break;

    Cell* cell = new Cell;
    cell->value = line;

    cell->next = result;
    result = cell;
}
return result;
```



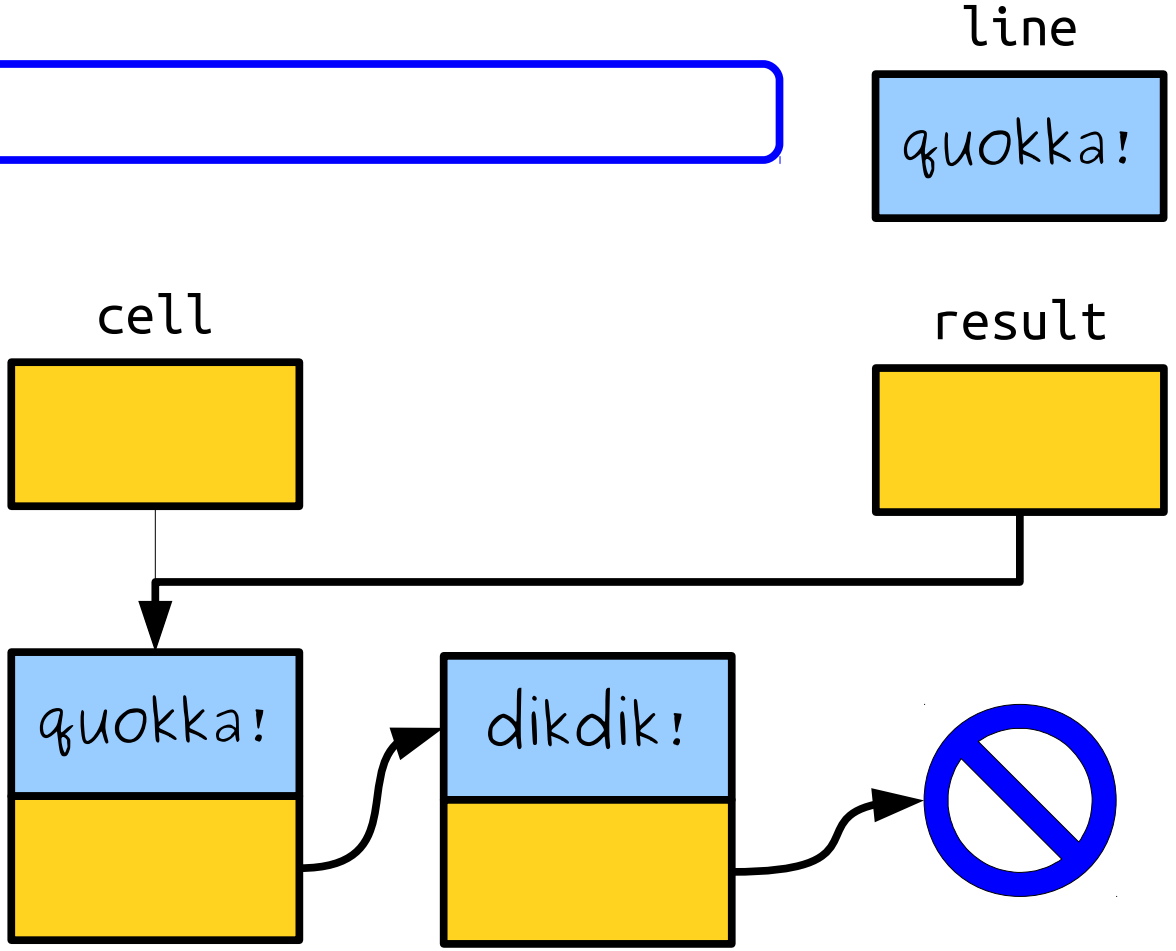
```
Cell* result = nullptr;
while (true) {
    string line = getLine("Next entry? ");
    if (line == "") break;

    Cell* cell = new Cell;
    cell->value = line;

    cell->next = result;
    result = cell;
}
return result;
```



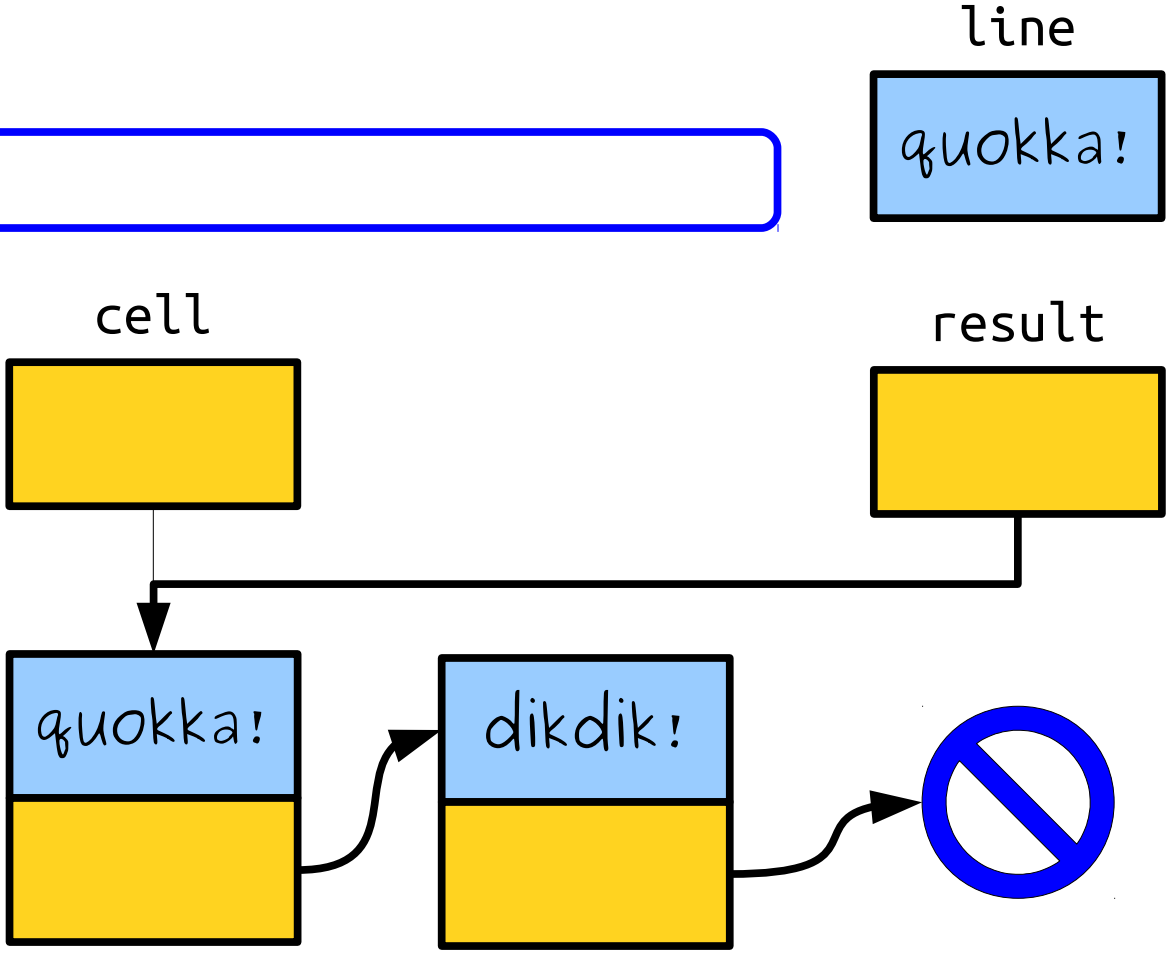
```
Cell* result = nullptr;
while (true) {
    string line = getLine("Next entry? ");
    if (line == "") break;

    Cell* cell = new Cell;
    cell->value = line;

    cell->next = result;
    result = cell;
}
return result;
```



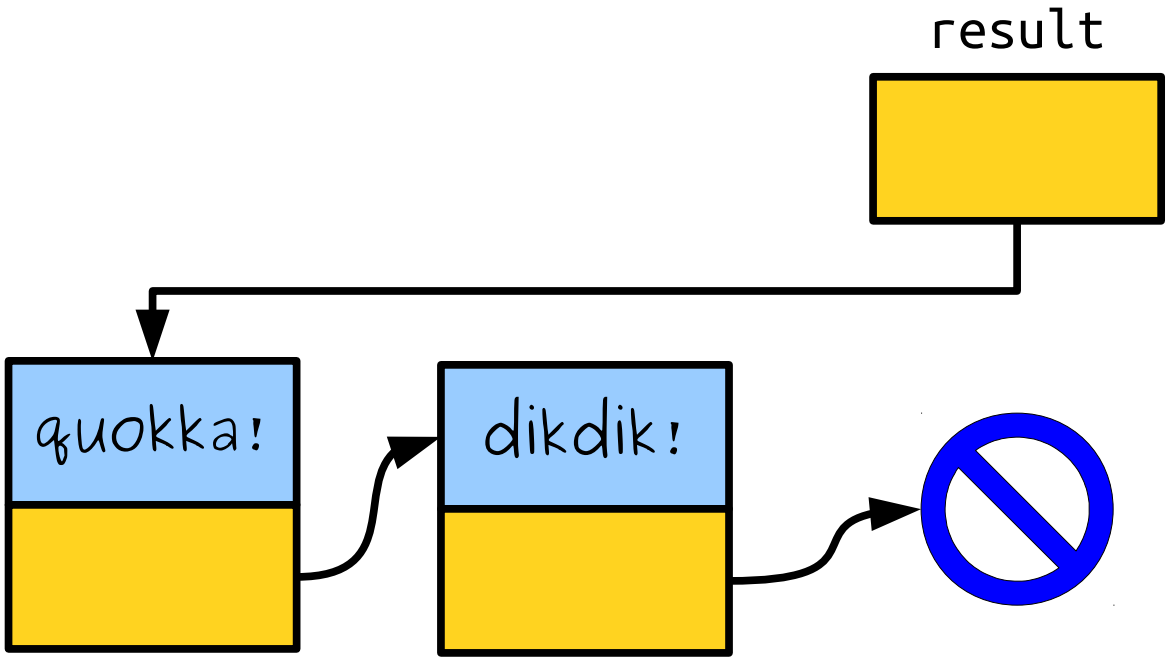
```
Cell* result = nullptr;
while (true) {
    string line = getLine("Next entry? ");
    if (line == "") break;

    Cell* cell = new Cell;
    cell->value = line;

    cell->next = result;
    result = cell;
}
return result;
```



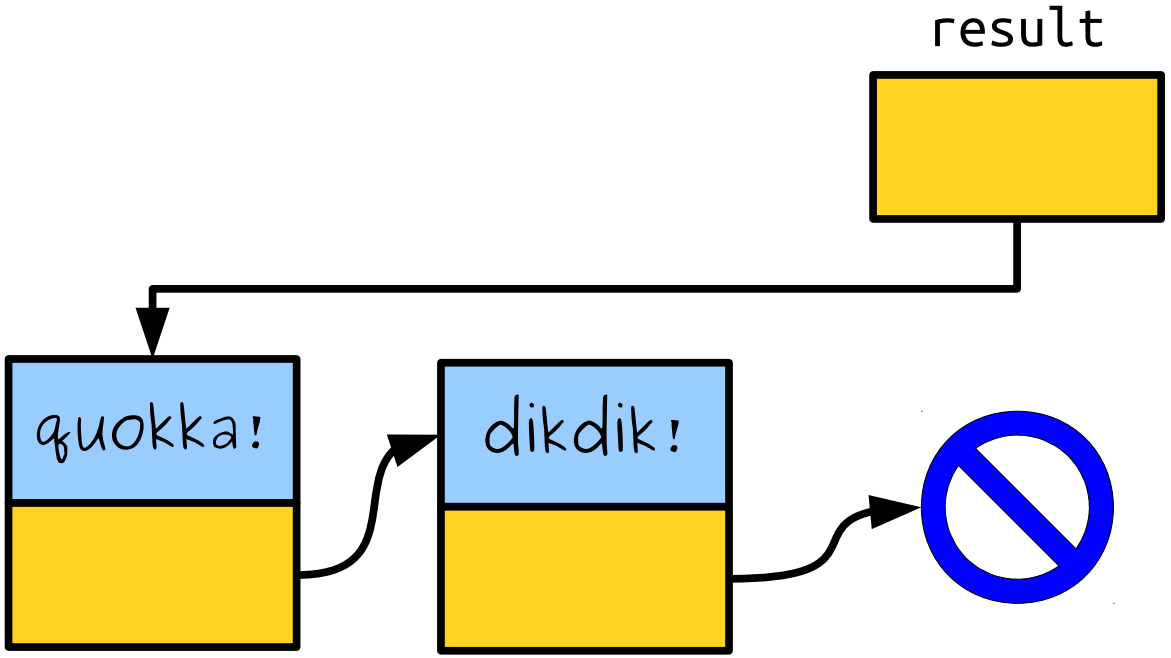
```
Cell* result = nullptr;
while (true) {
    string line = getLine("Next entry? ");
    if (line == "") break;

    Cell* cell = new Cell;
    cell->value = line;

    cell->next = result;
    result = cell;
}
return result;
```



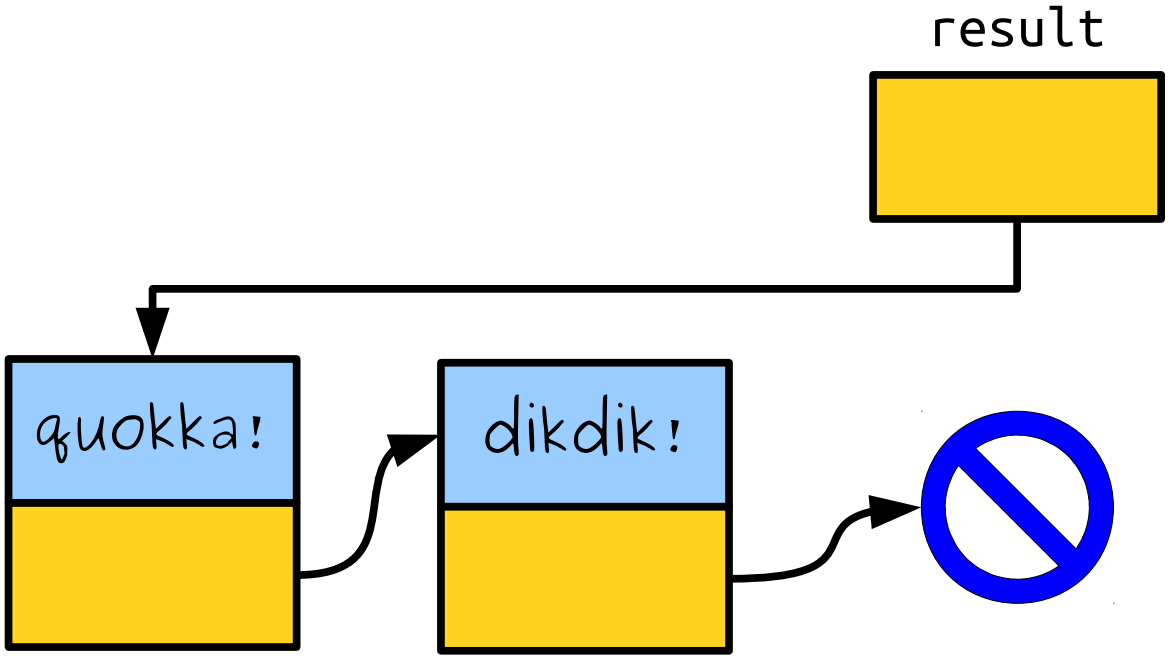
```
Cell* result = nullptr;
while (true) {
    string line = getLine("Next entry? ");
    if (line == "") break;

    Cell* cell = new Cell;
    cell->value = line;

    cell->next = result;
    result = cell;
}
return result;
```



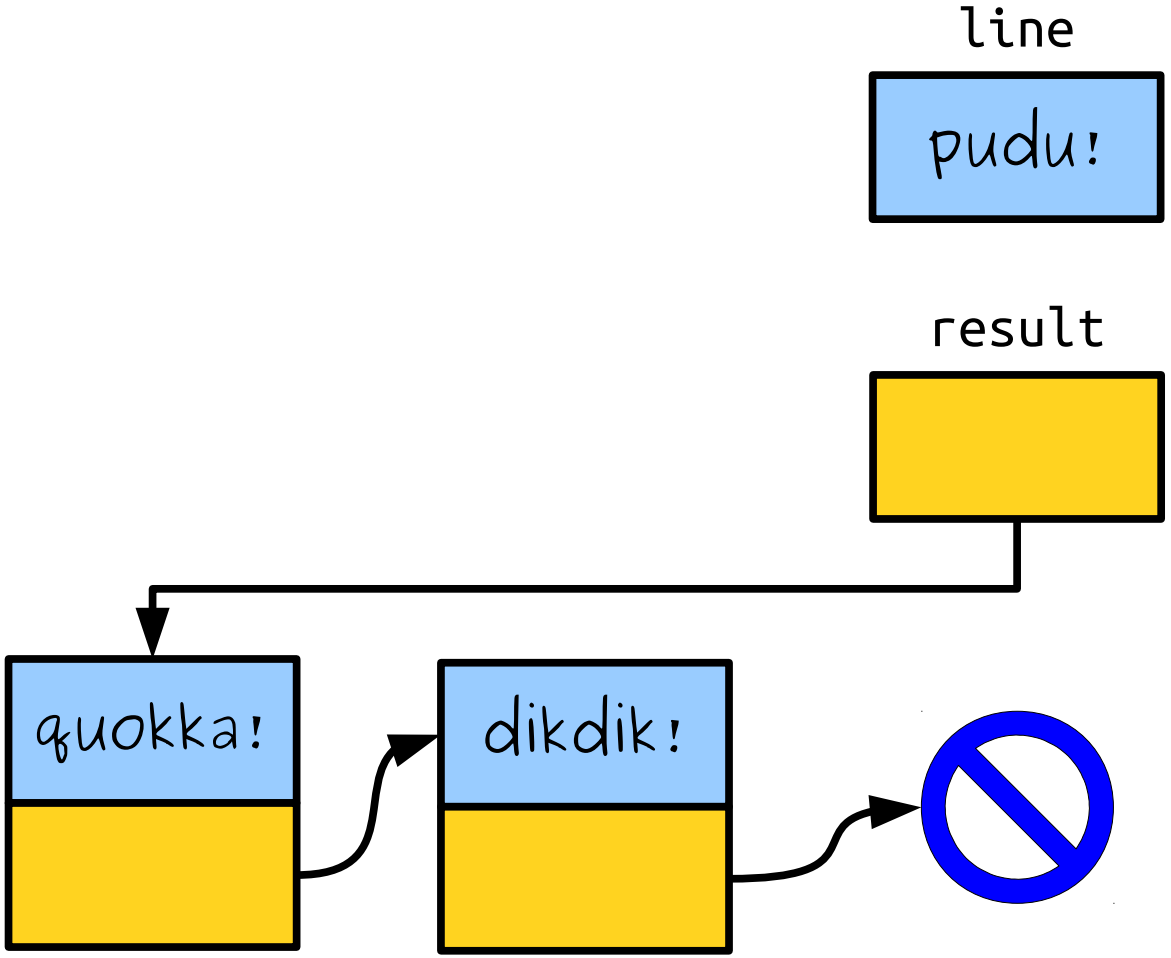
```
Cell* result = nullptr;
while (true) {
    string line = getLine("Next entry? ");
    if (line == "") break;

    Cell* cell = new Cell;
    cell->value = line;

    cell->next = result;
    result = cell;
}
return result;
```



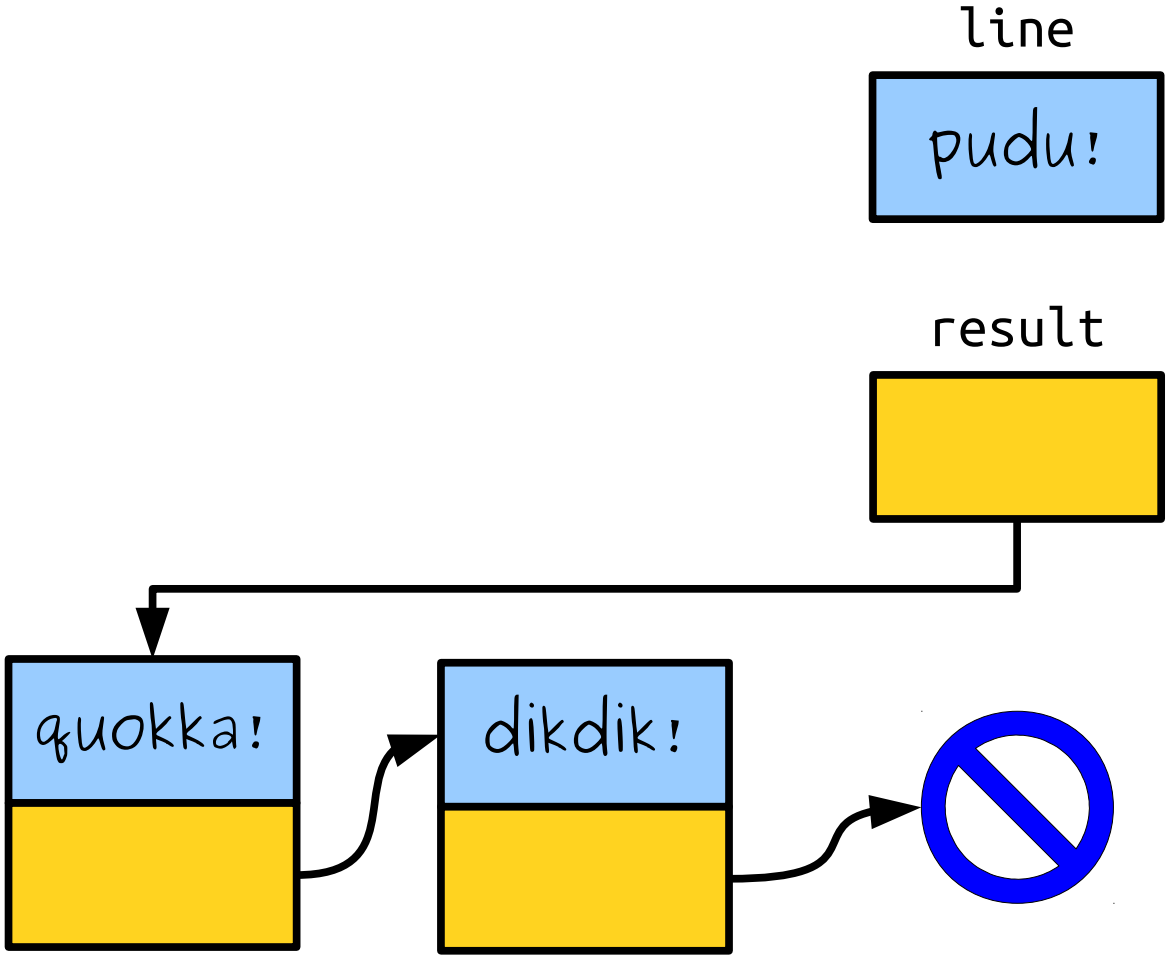
```
Cell* result = nullptr;
while (true) {
    string line = getLine("Next entry? ");
    if (line == "") break;

    Cell* cell = new Cell;
    cell->value = line;

    cell->next = result;
    result = cell;
}
return result;
```



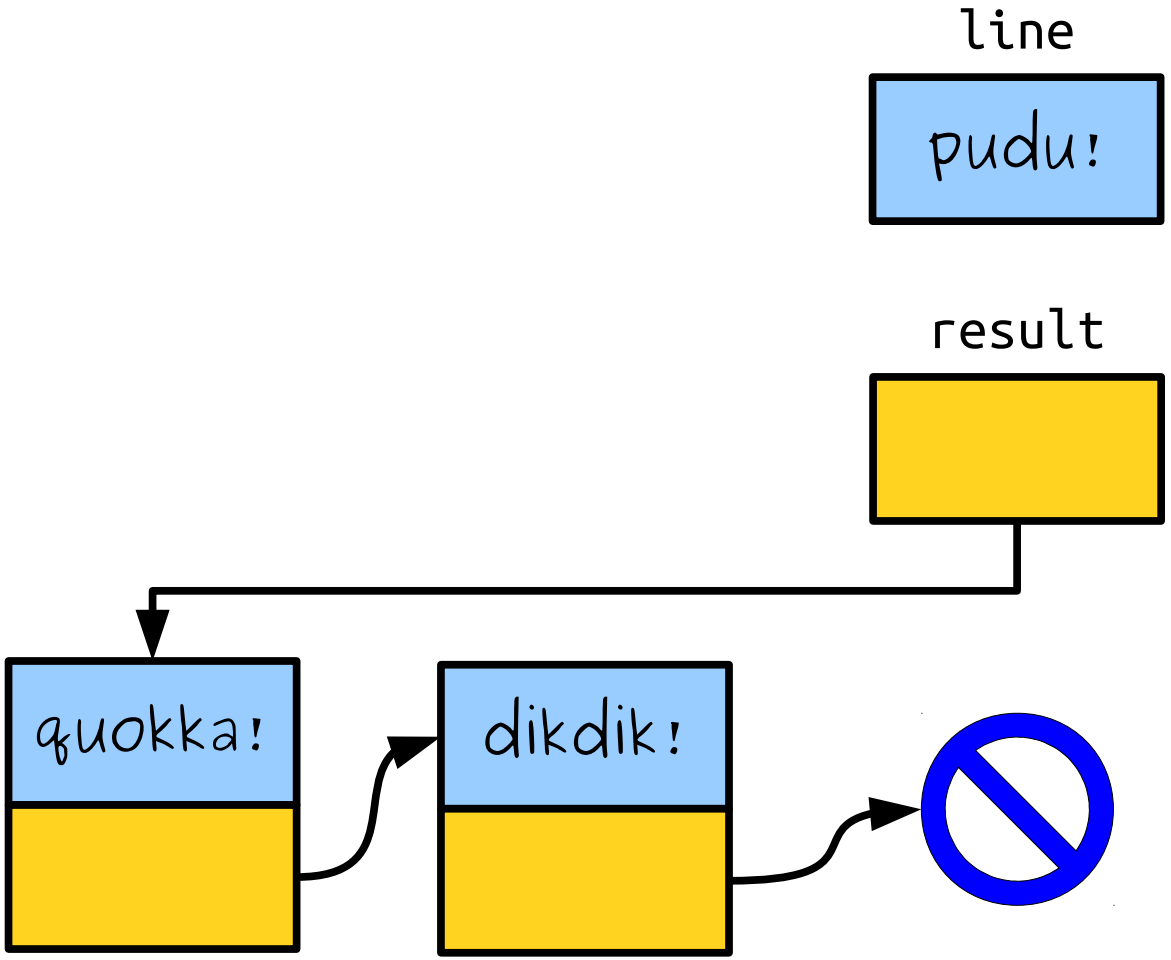
```
Cell* result = nullptr;
while (true) {
    string line = getLine("Next entry? ");
    if (line == "") break;

    Cell* cell = new Cell;
    cell->value = line;

    cell->next = result;
    result = cell;
}
return result;
```



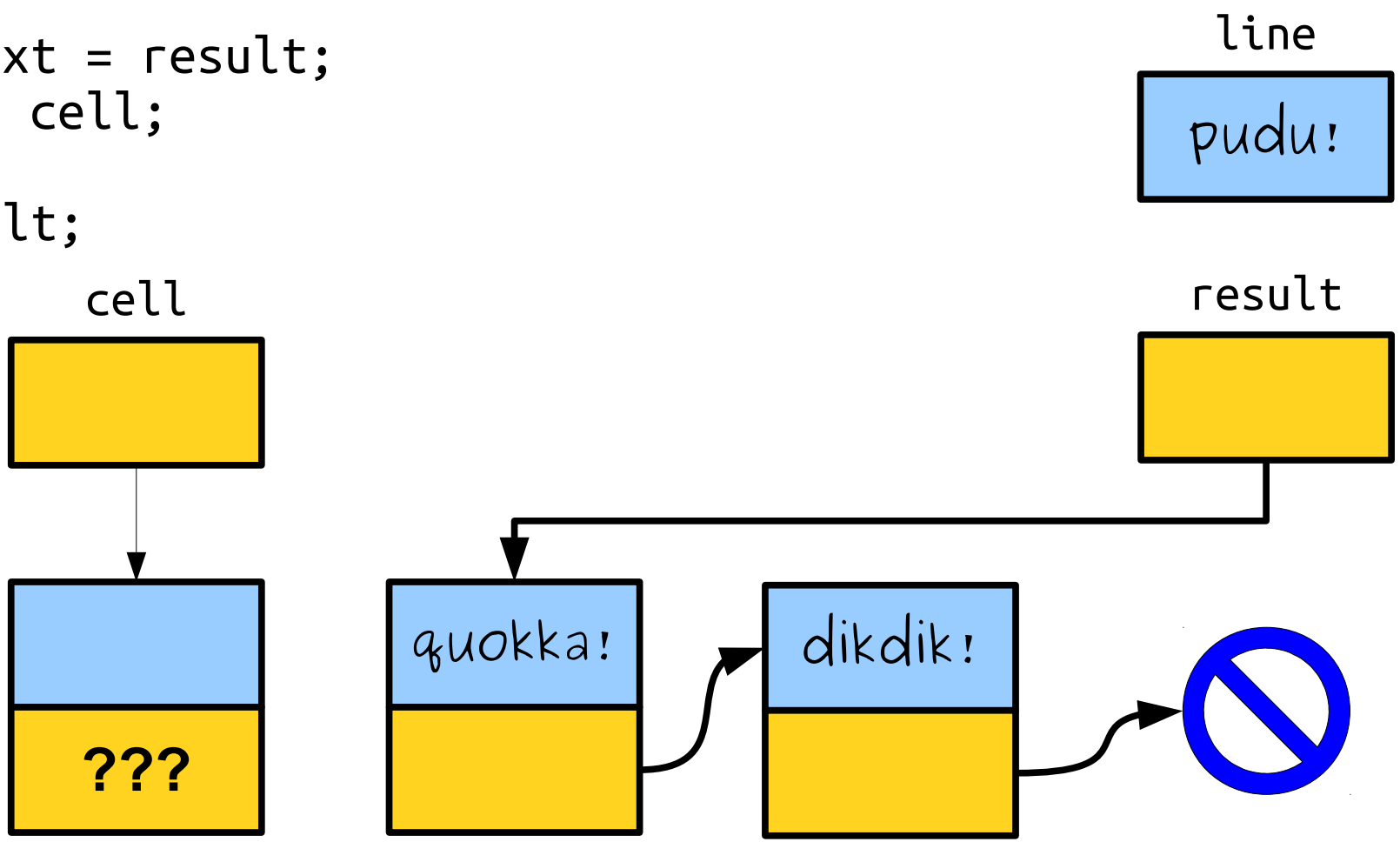
```
Cell* result = nullptr;
while (true) {
    string line = getLine("Next entry? ");
    if (line == "") break;

    Cell* cell = new Cell;
    cell->value = line;

    cell->next = result;
    result = cell;
}
return result;
```



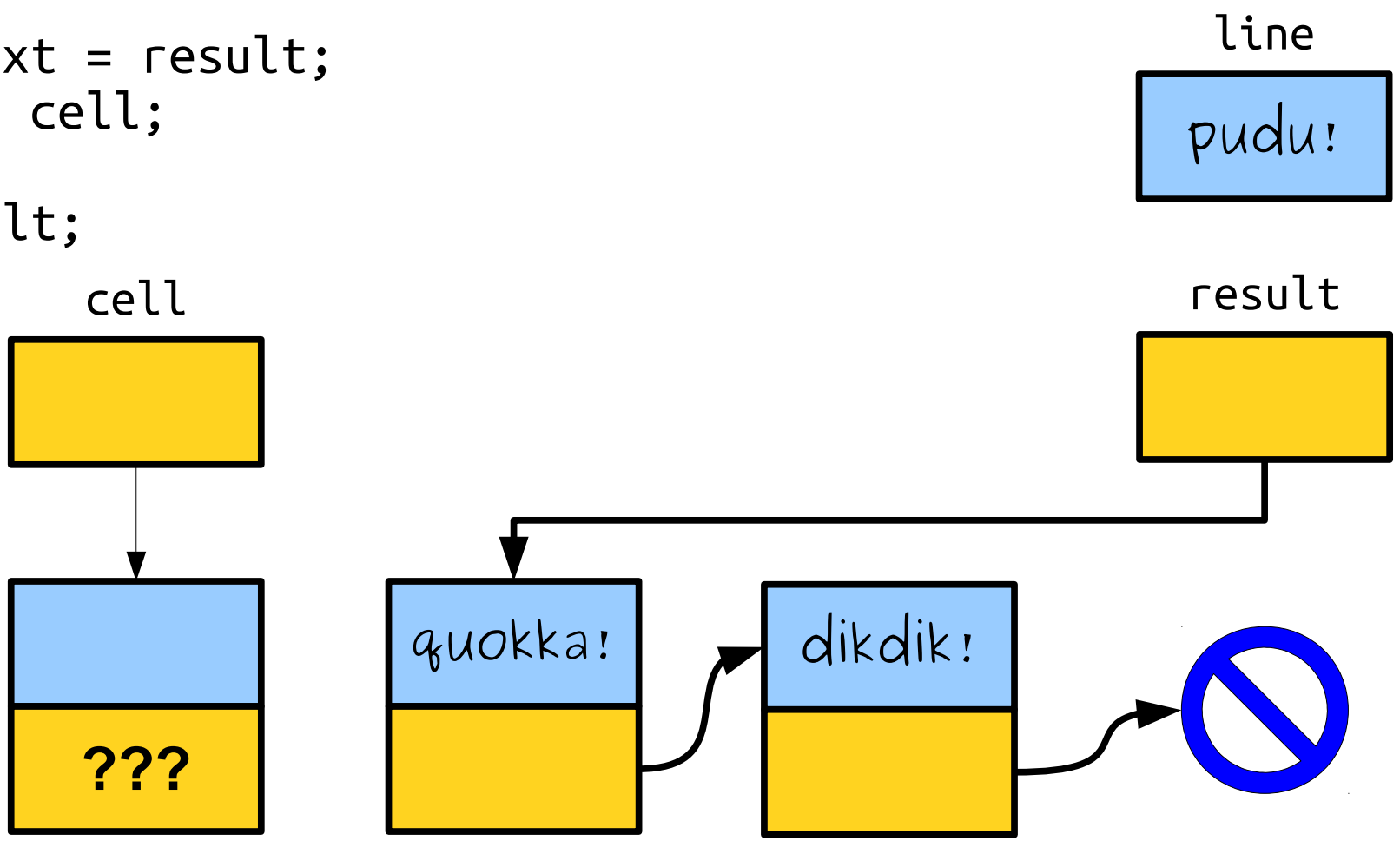
```
Cell* result = nullptr;
while (true) {
    string line = getLine("Next entry? ");
    if (line == "") break;

    Cell* cell = new Cell;
    cell->value = line;

    cell->next = result;
    result = cell;
}
return result;
```



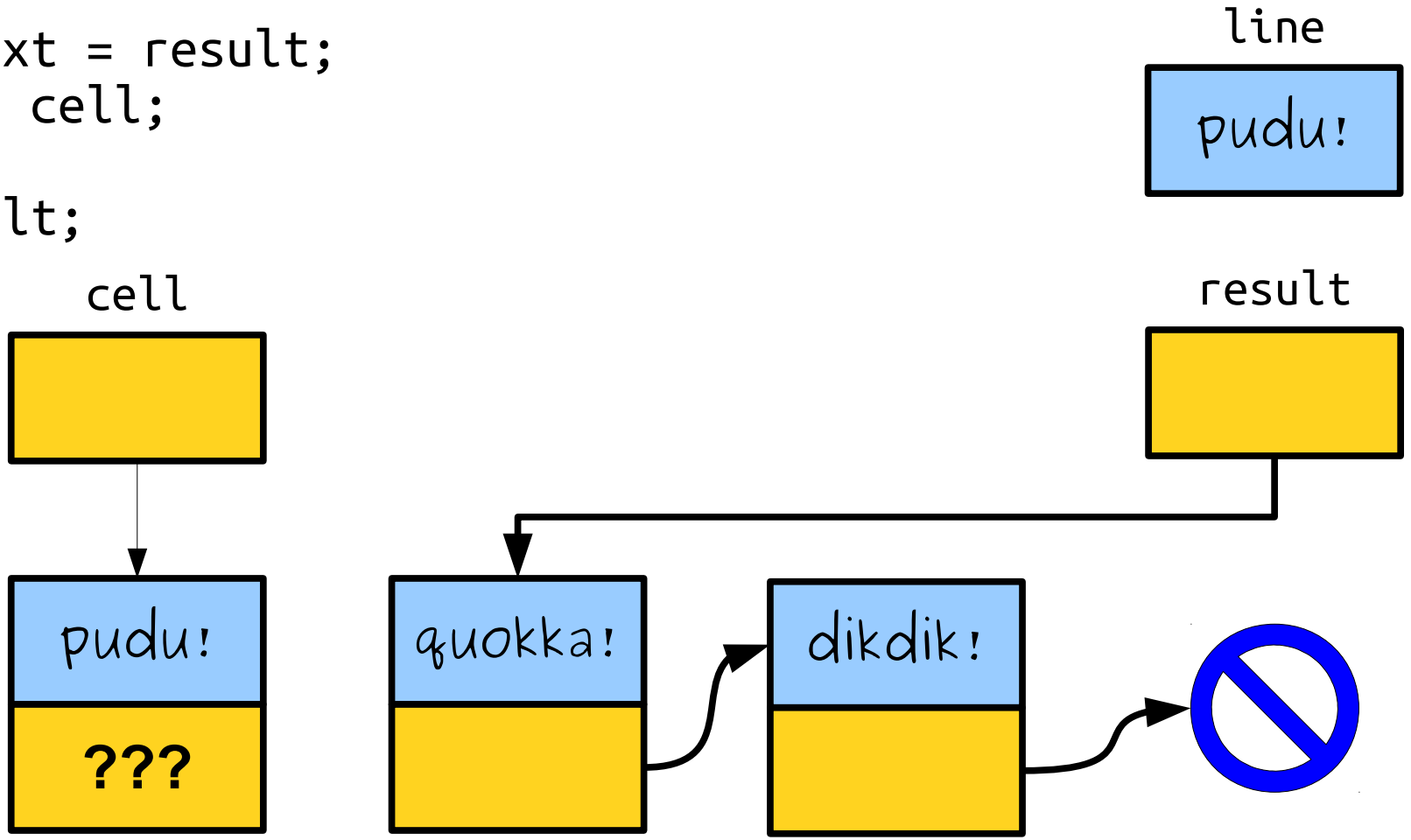
```
Cell* result = nullptr;
while (true) {
    string line = getLine("Next entry? ");
    if (line == "") break;

    Cell* cell = new Cell;
    cell->value = line;

    cell->next = result;
    result = cell;
}
return result;
```



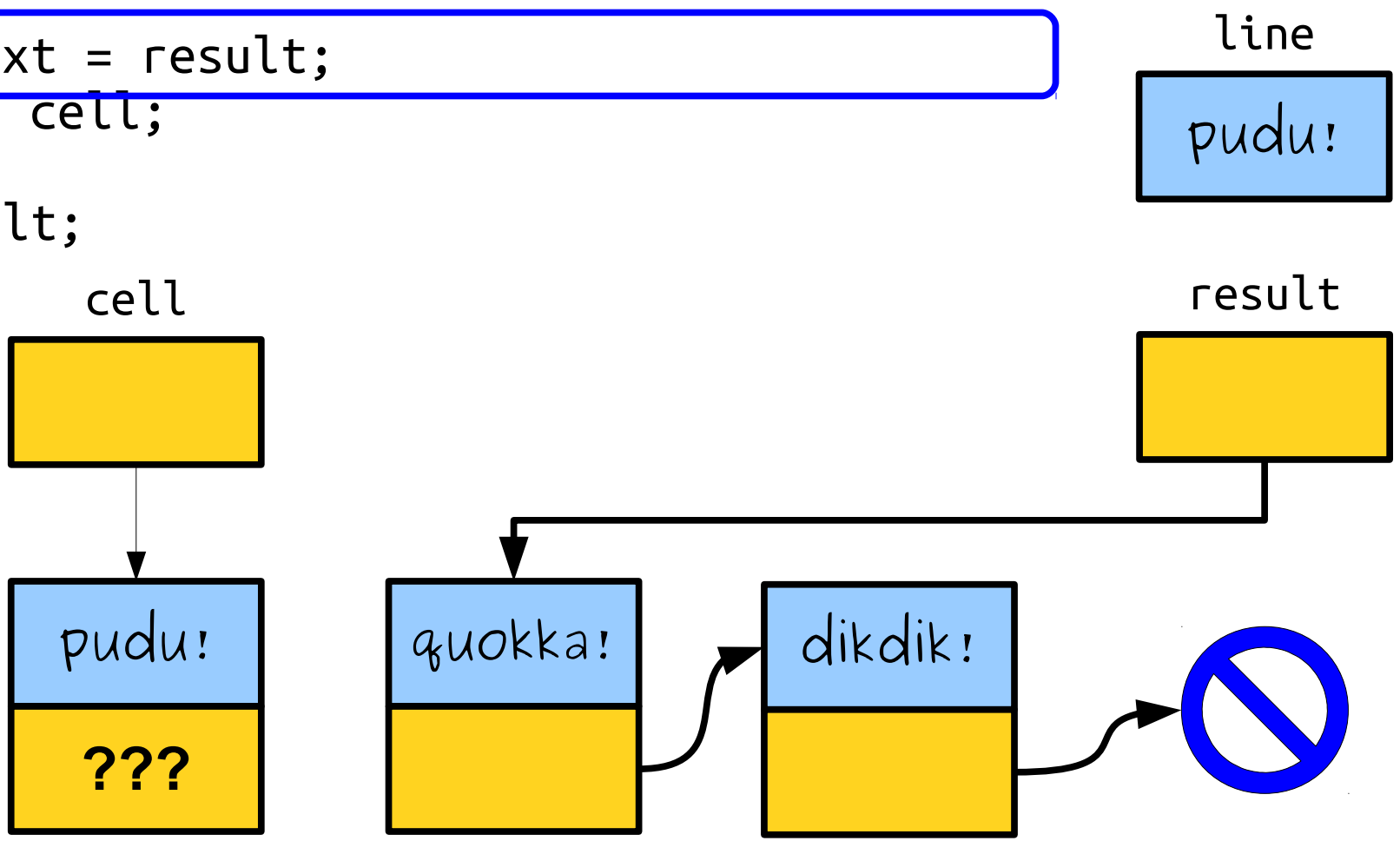
```
Cell* result = nullptr;
while (true) {
    string line = getLine("Next entry? ");
    if (line == "") break;

    Cell* cell = new Cell;
    cell->value = line;

    cell->next = result;
    result = cell;
}
return result;
```



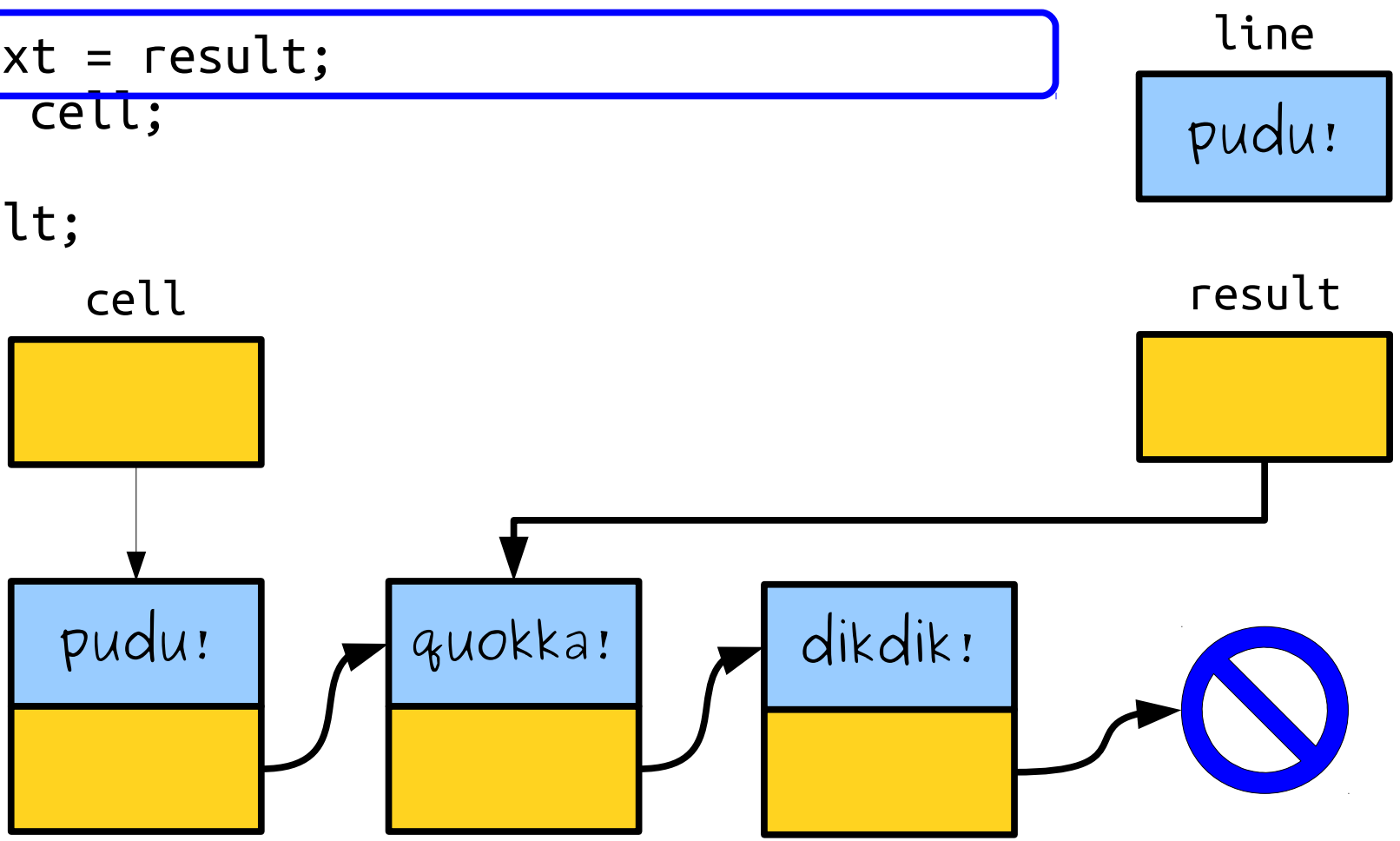
```
Cell* result = nullptr;
while (true) {
    string line = getLine("Next entry? ");
    if (line == "") break;

    Cell* cell = new Cell;
    cell->value = line;

    cell->next = result;
    result = cell;
}
return result;
```



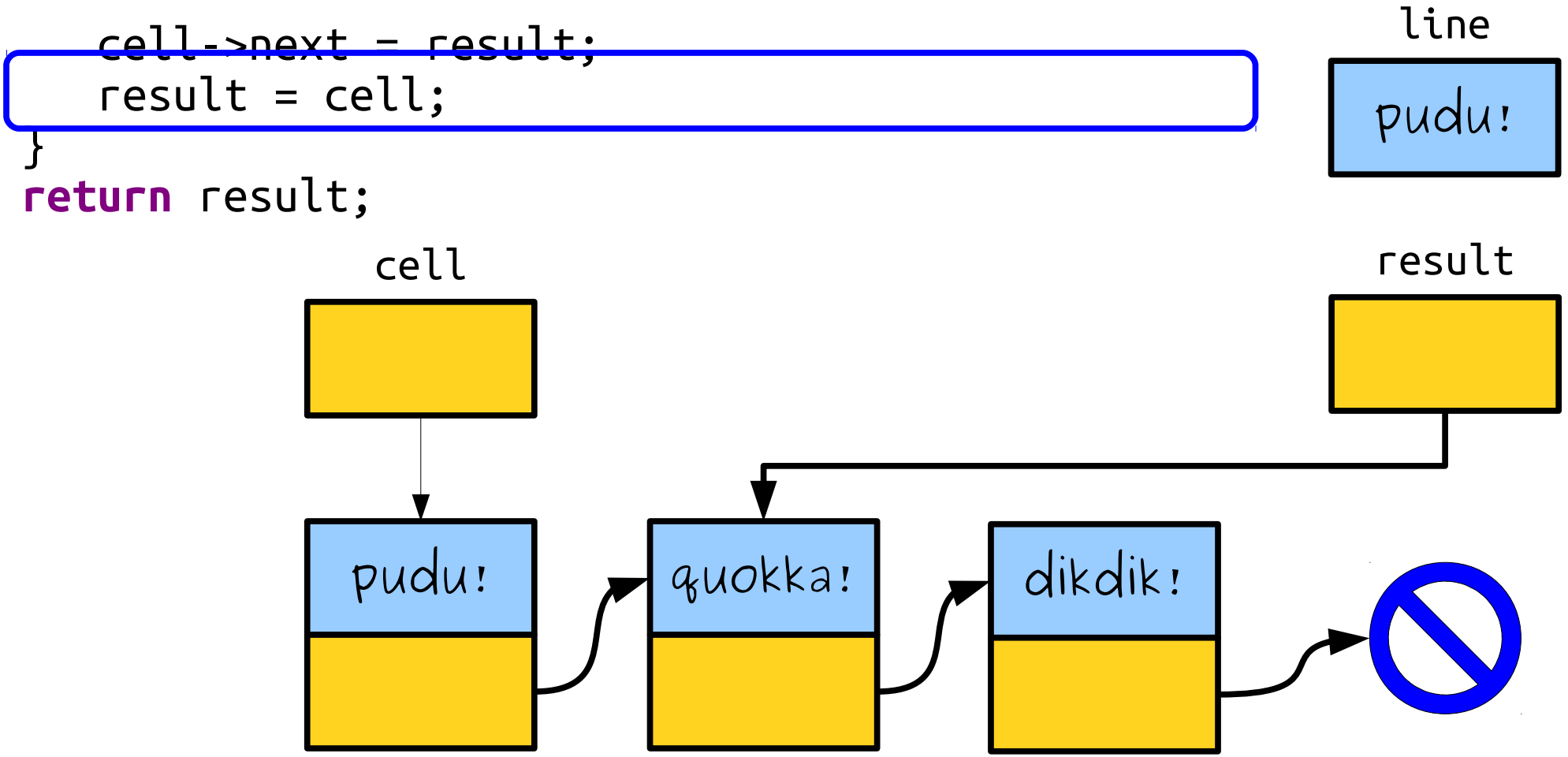
```
Cell* result = nullptr;
while (true) {
    string line = getLine("Next entry? ");
    if (line == "") break;

    Cell* cell = new Cell;
    cell->value = line;

    cell->next = result;
    result = cell;
}
return result;
```



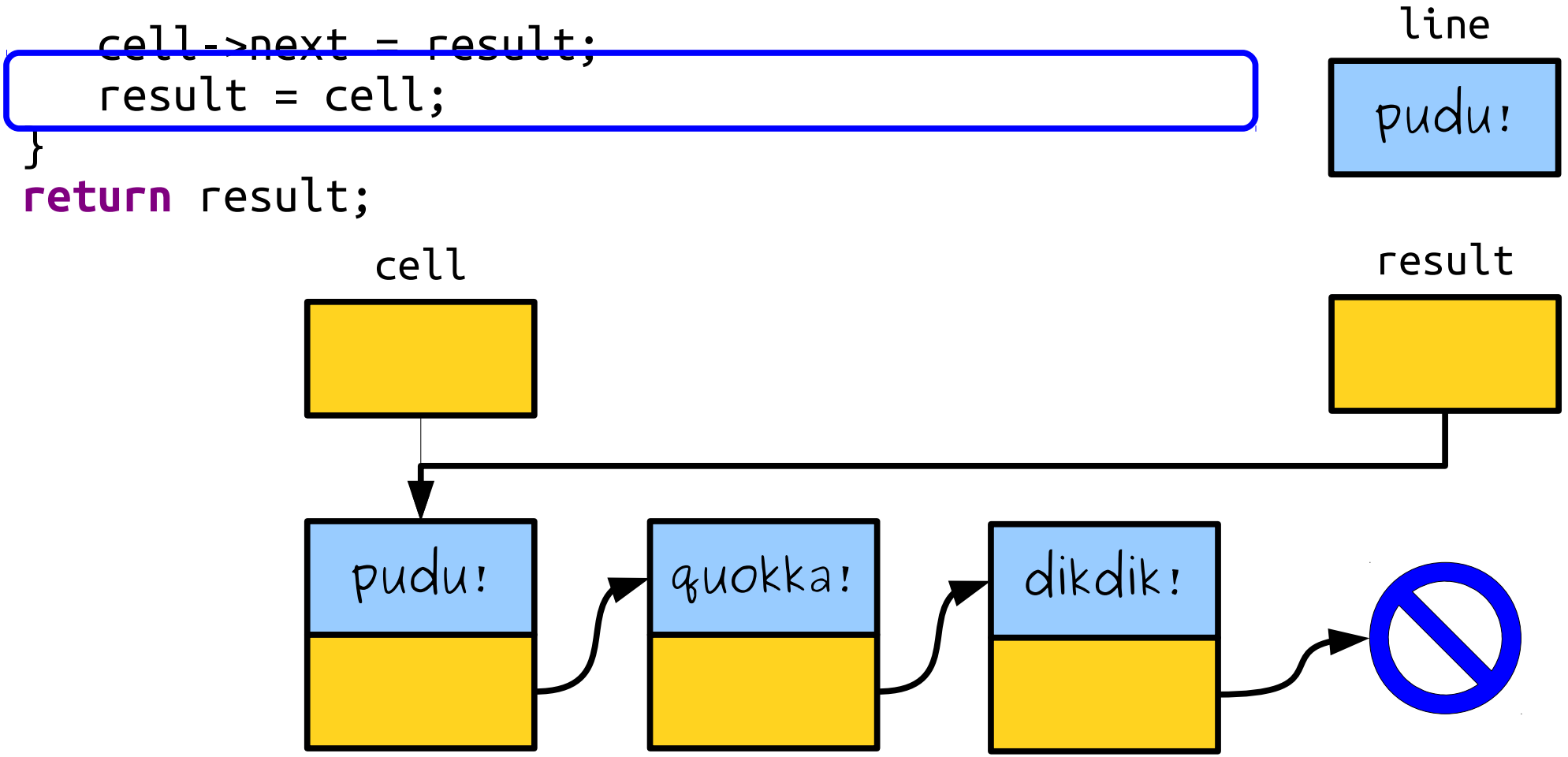
```
Cell* result = nullptr;
while (true) {
    string line = getLine("Next entry? ");
    if (line == "") break;

    Cell* cell = new Cell;
    cell->value = line;

    cell->next = result;
    result = cell;
}
return result;
```



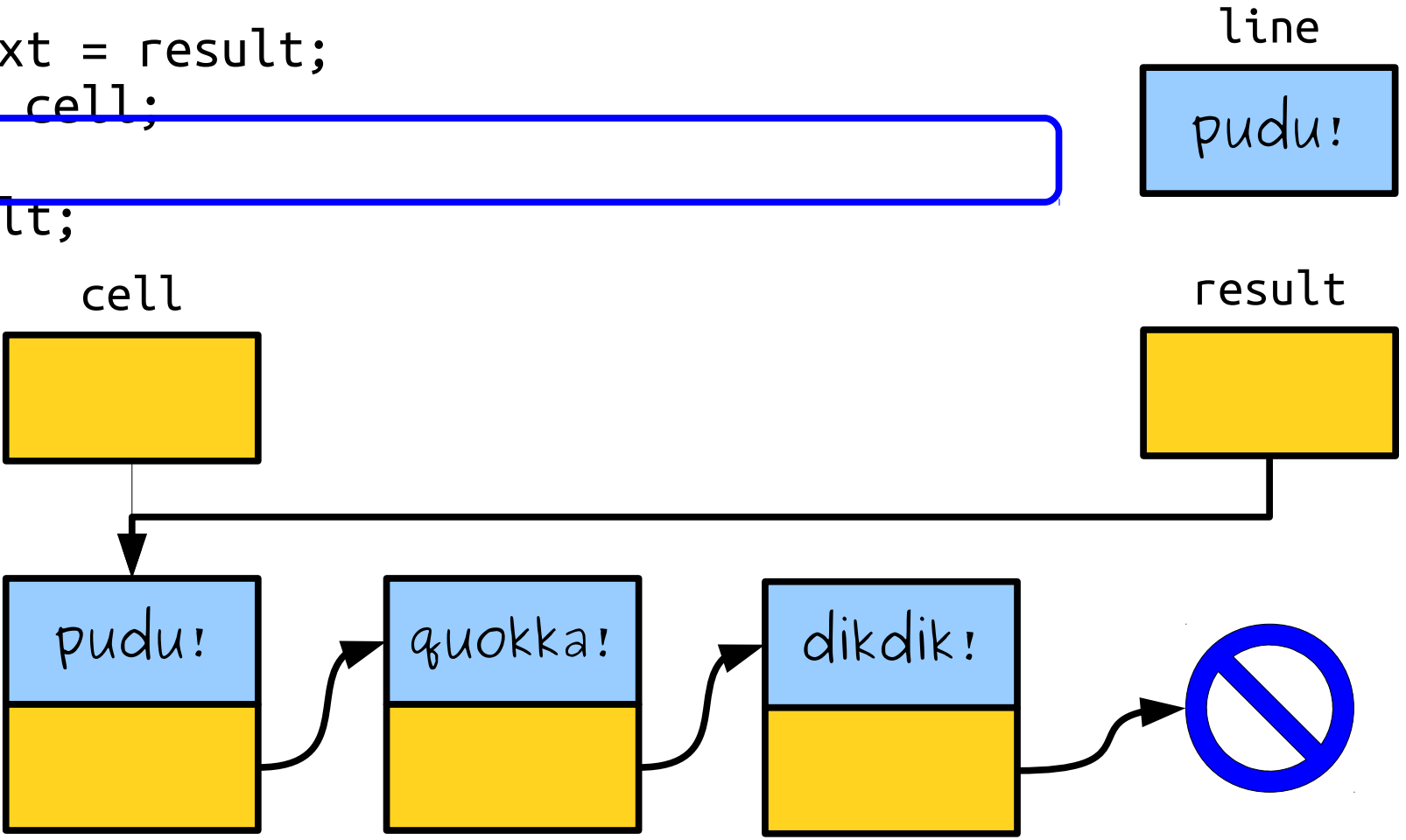
```
Cell* result = nullptr;
while (true) {
    string line = getLine("Next entry? ");
    if (line == "") break;

    Cell* cell = new Cell;
    cell->value = line;

    cell->next = result;
    result = cell;
}
return result;
```



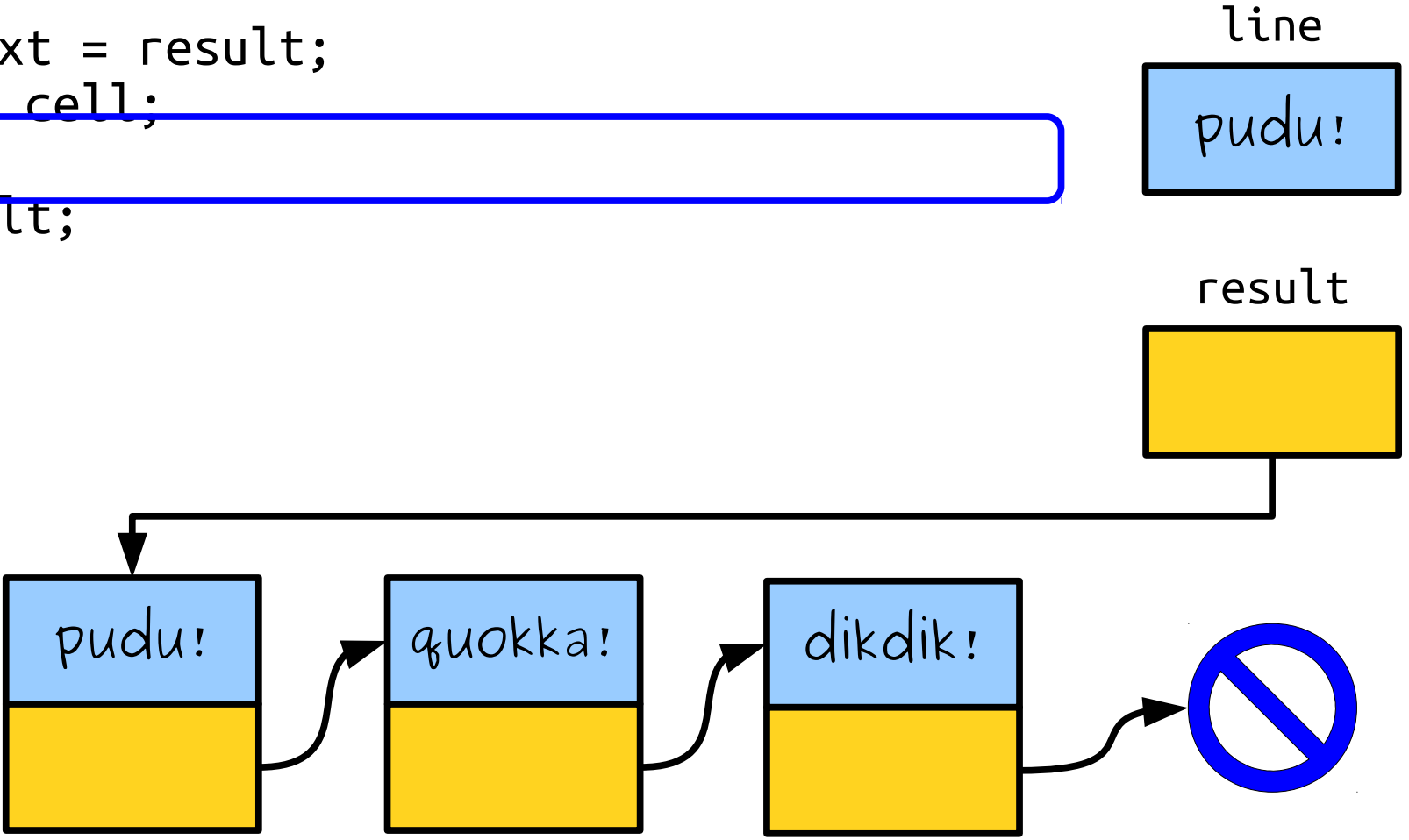
```
Cell* result = nullptr;
while (true) {
    string line = getLine("Next entry? ");
    if (line == "") break;

    Cell* cell = new Cell;
    cell->value = line;

    cell->next = result;
    result = cell;
}
return result;
```



```
Cell* result = nullptr;
while (true) {
    string line = getLine("Next entry? ");
    if (line == "") break;

    Cell* cell = new Cell;
    cell->value = line;

    cell->next = result;
    result = cell;
}
return result;
```



```
Cell* result = nullptr;
while (true) {
    string line = getLine("Next entry? ");
    if (line == "") break;

    Cell* cell = new Cell;
    cell->value = line;

    cell->next = result;
    result = cell;
}
return result;
```

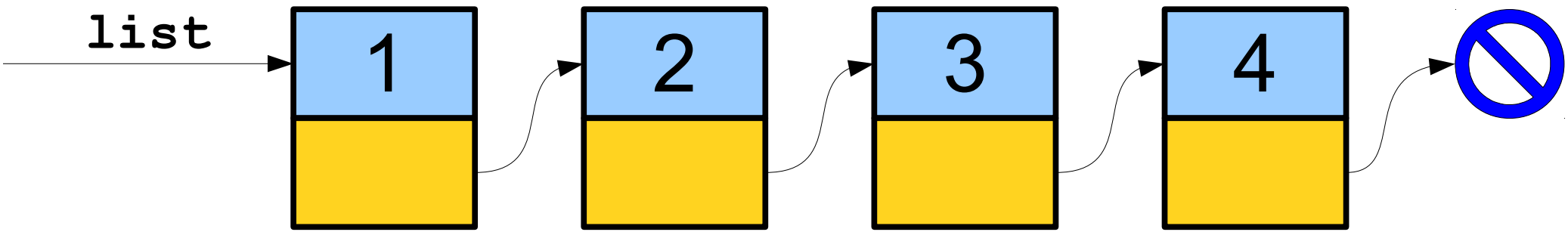


Now that we’ve got the list,
what can we do with it?

Traversing a Linked List

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

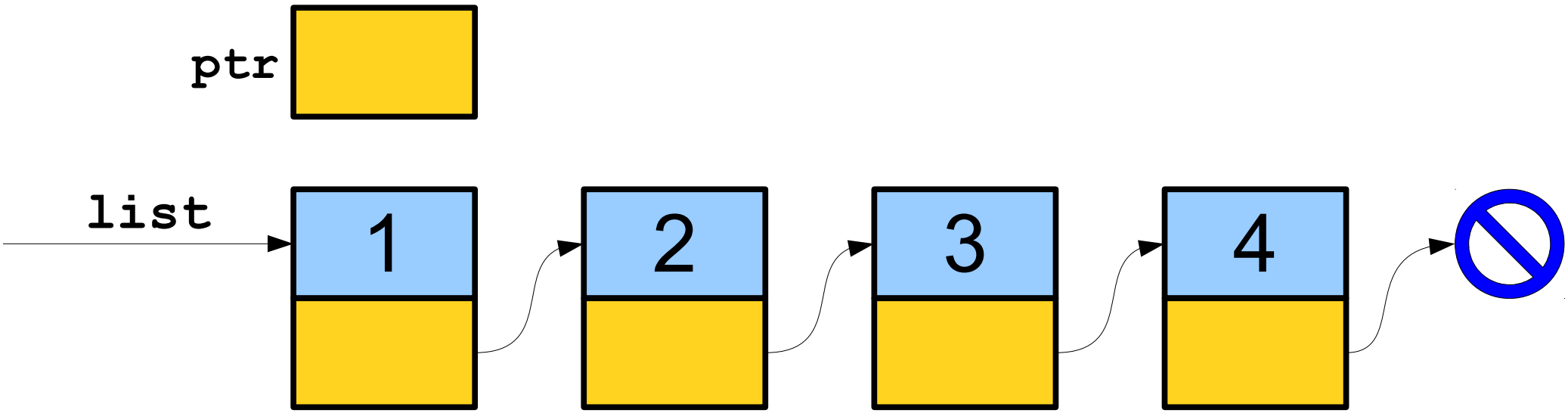
```
for (Cell* ptr = list; ptr != nullptr; ptr = ptr->next) {
    /* … use ptr … */
}
```



Traversing a Linked List

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

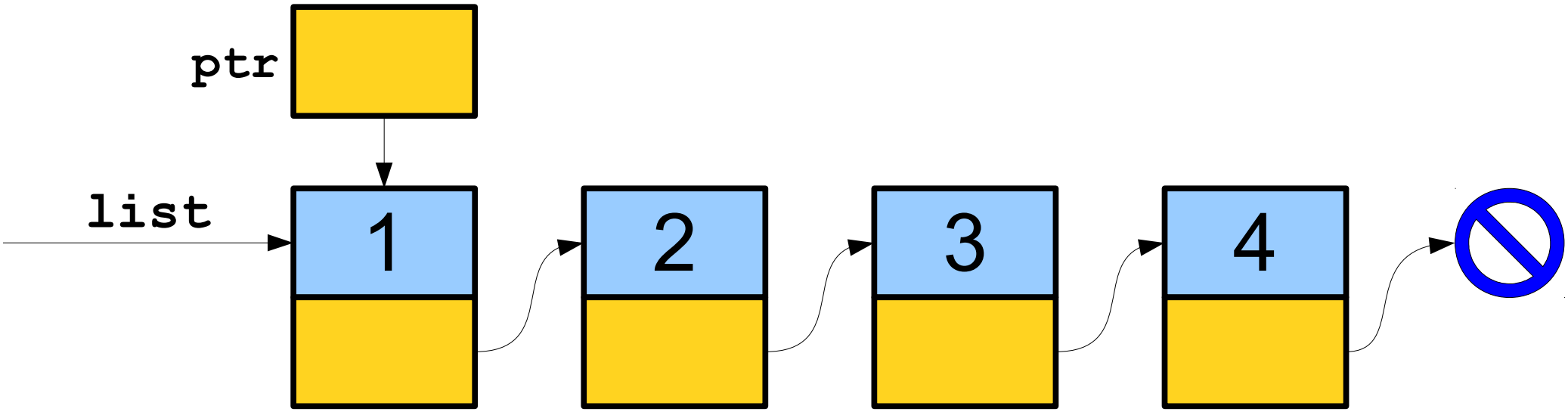
```
for (Cell* ptr = list; ptr != nullptr; ptr = ptr->next) {
    /* … use ptr … */
}
```



Traversing a Linked List

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

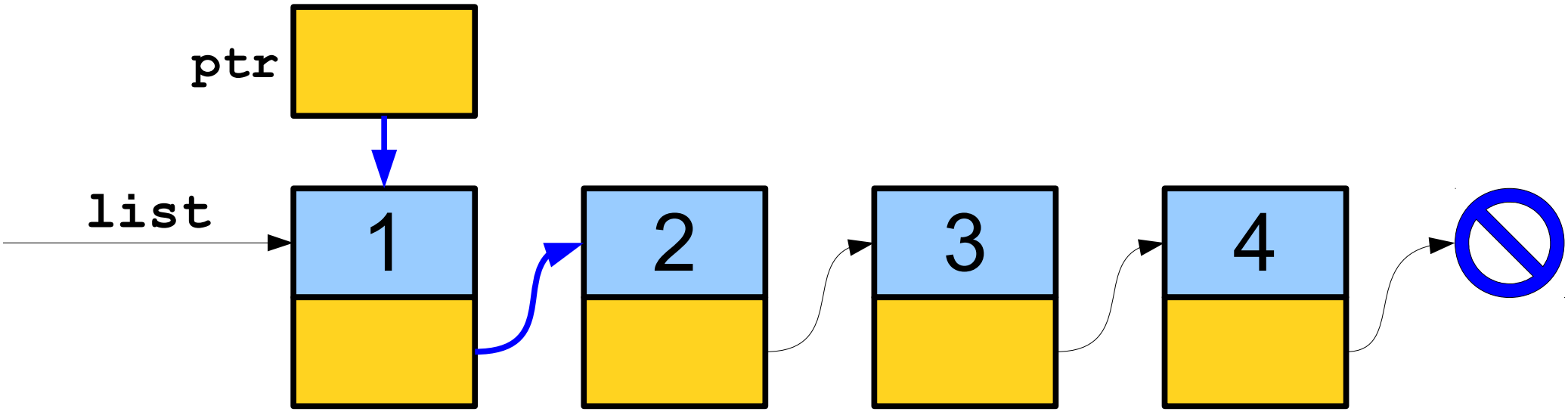
```
for (Cell* ptr = list; ptr != nullptr; ptr = ptr->next) {
    /* … use ptr … */
}
```



Traversing a Linked List

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

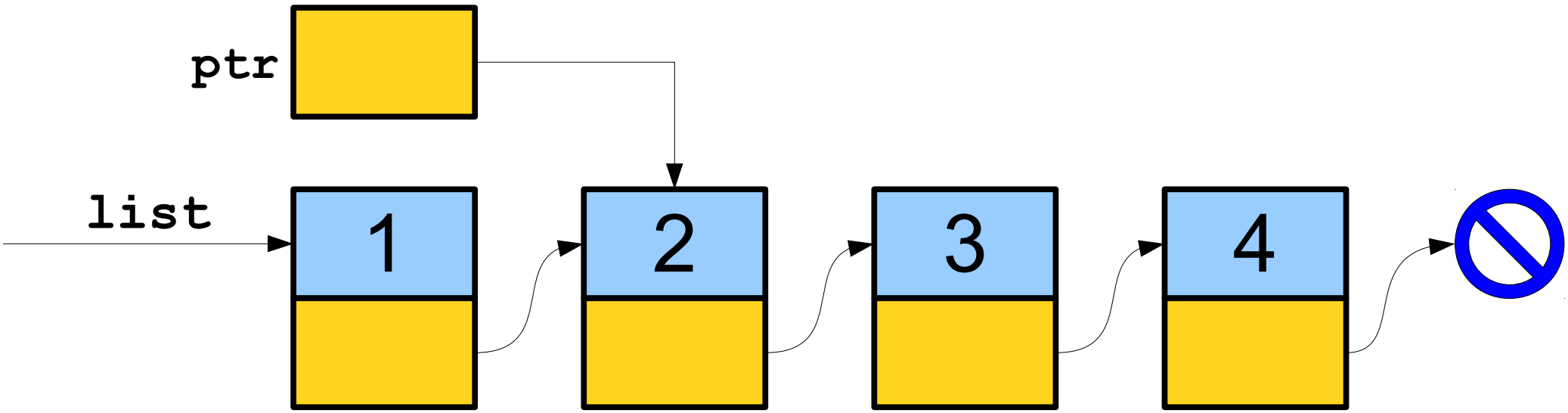
```
for (Cell* ptr = list; ptr != nullptr; ptr = ptr->next) {
    /* … use ptr … */
}
```



Traversing a Linked List

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

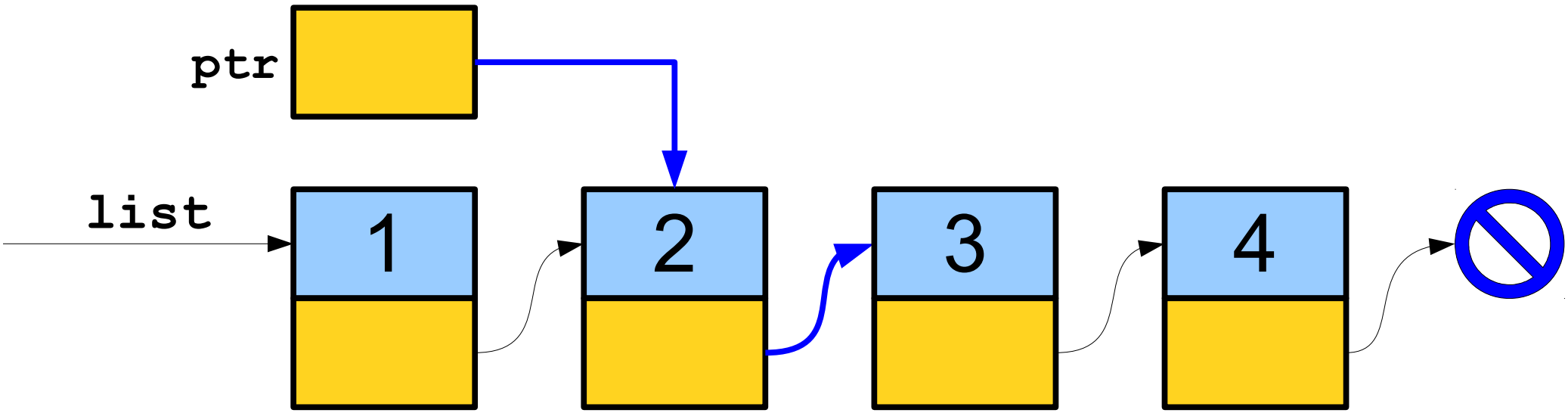
```
for (Cell* ptr = list; ptr != nullptr; ptr = ptr->next) {
    /* … use ptr … */
}
```



Traversing a Linked List

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

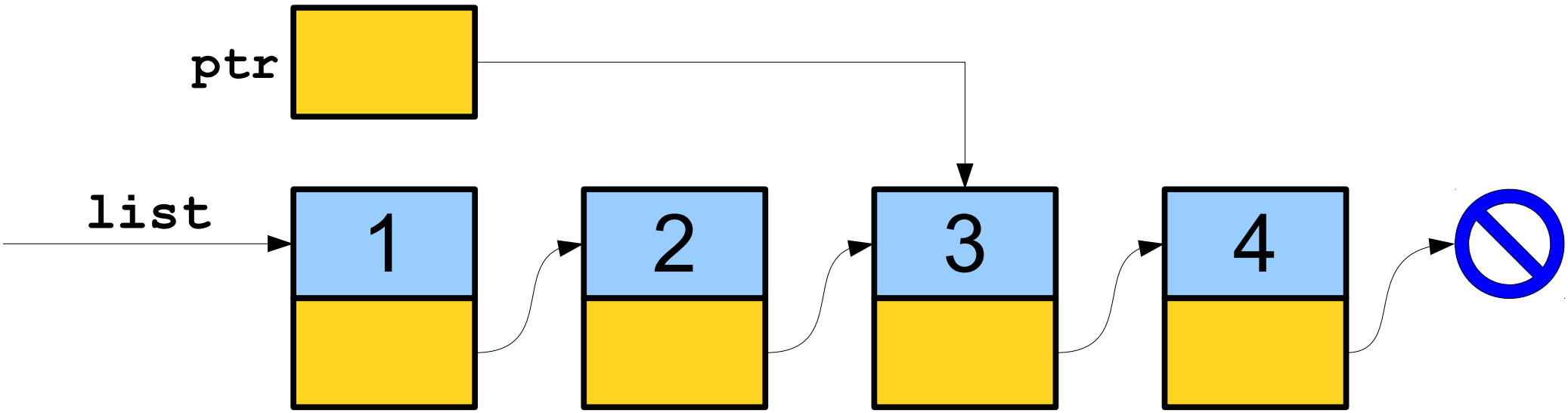
```
for (Cell* ptr = list; ptr != nullptr; ptr = ptr->next) {
    /* … use ptr … */
}
```



Traversing a Linked List

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

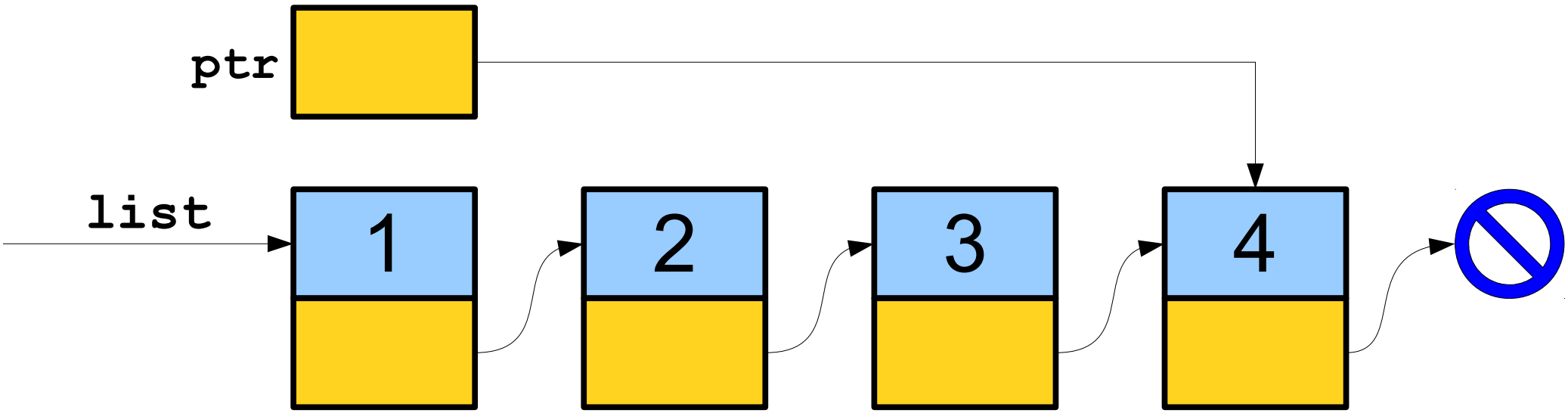
```
for (Cell* ptr = list; ptr != nullptr; ptr = ptr->next) {
    /* … use ptr … */
}
```



Traversing a Linked List

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

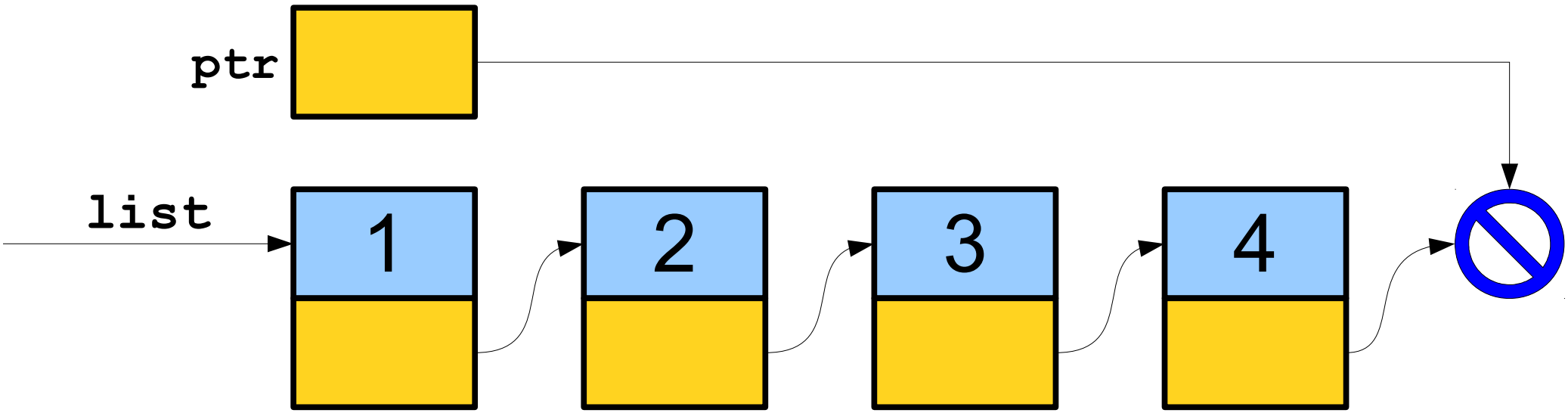
```
for (Cell* ptr = list; ptr != nullptr; ptr = ptr->next) {
    /* … use ptr … */
}
```



Traversing a Linked List

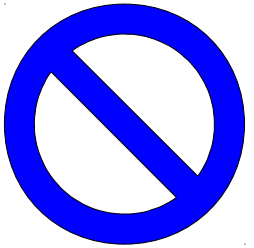
- Once we have a linked list, we can traverse it by following the links one at a time.

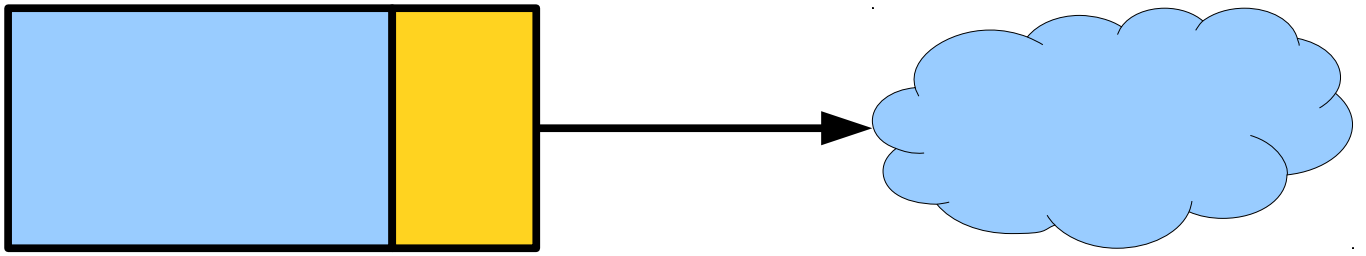
```
for (Cell* ptr = list; ptr != nullptr; ptr = ptr->next) {
    /* … use ptr … */
}
```



A Linked List is Either…

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





a single linked list cell that points…

… at another linked list.

Next Time

- ***Pointers by Reference***
  - Fun for the whole linked list family!
- ***Reimplementing Stacks and Queues***
  - Worst-case efficiency, at a price!