Linked Lists

Part Three

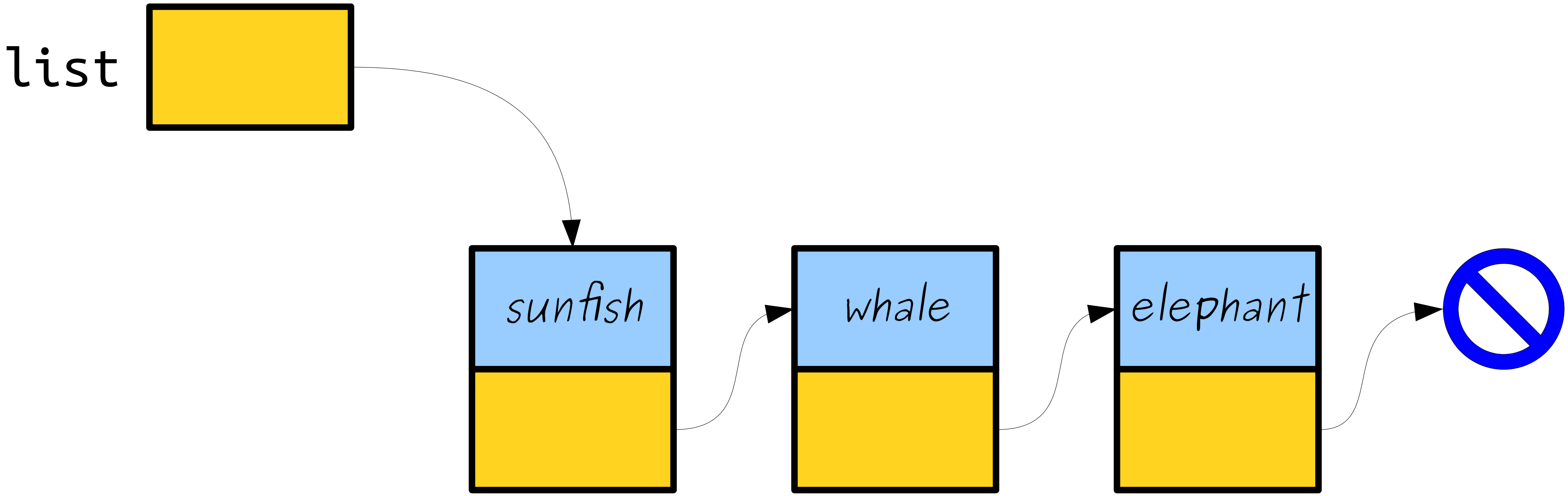
Outline for Today

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

Pointers and References

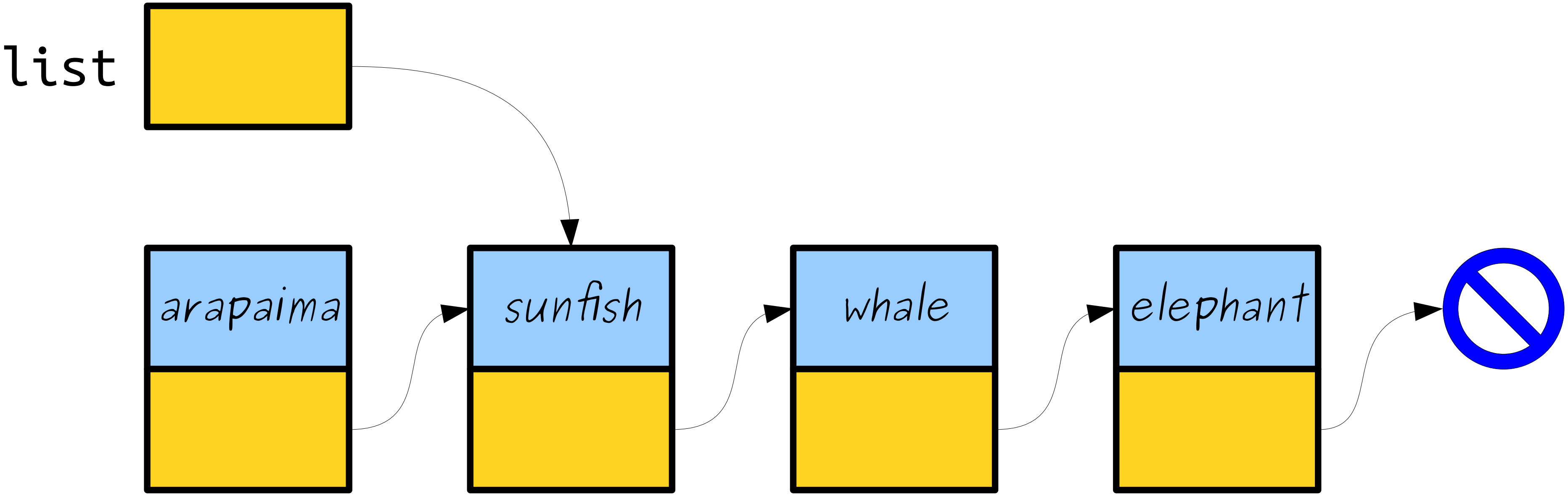
Prepending an Element

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



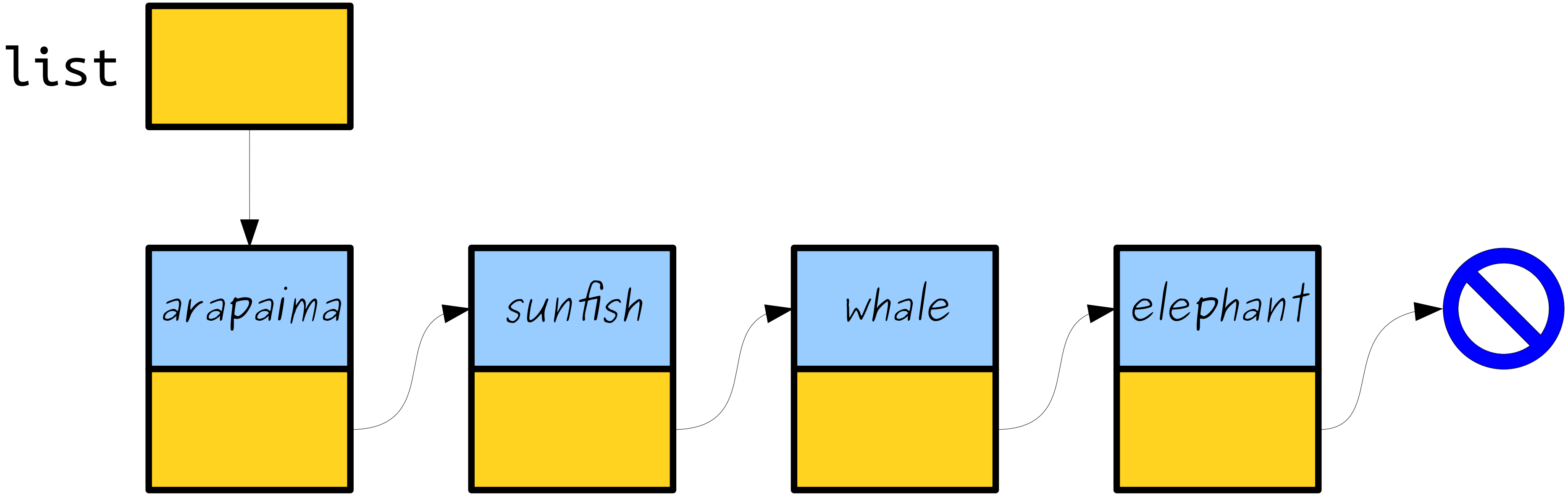
Prepending an Element

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



Prepending an Element

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

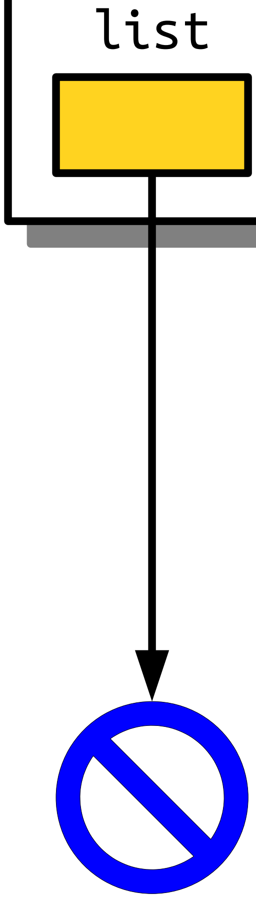


What went wrong?

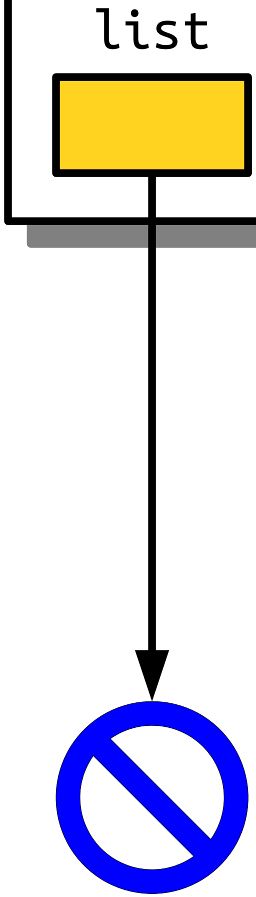
```cpp
int main() {
    Cell* list = nullptr;
    prependTo(list, "Sartre");
    prependTo(list, "Camus");
    prependTo(list, "Nietzsche");

    return 0;
}
```

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

    return 0;
}
```

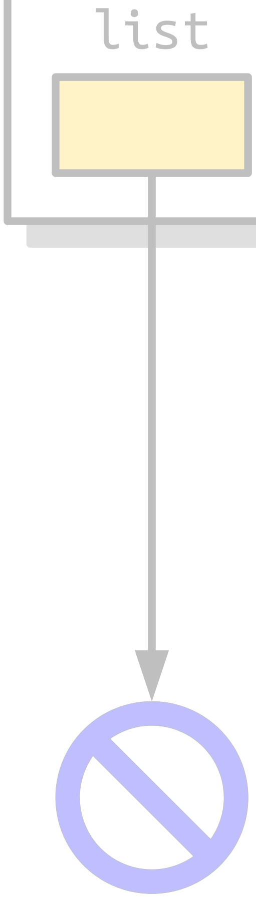
```
int main() {
    Cell* list = nullptr;
    prependTo(list, "Sartre");
    prependTo(list, "Camus");
    prependTo(list, "Nietzsche");

    return 0;
}
```

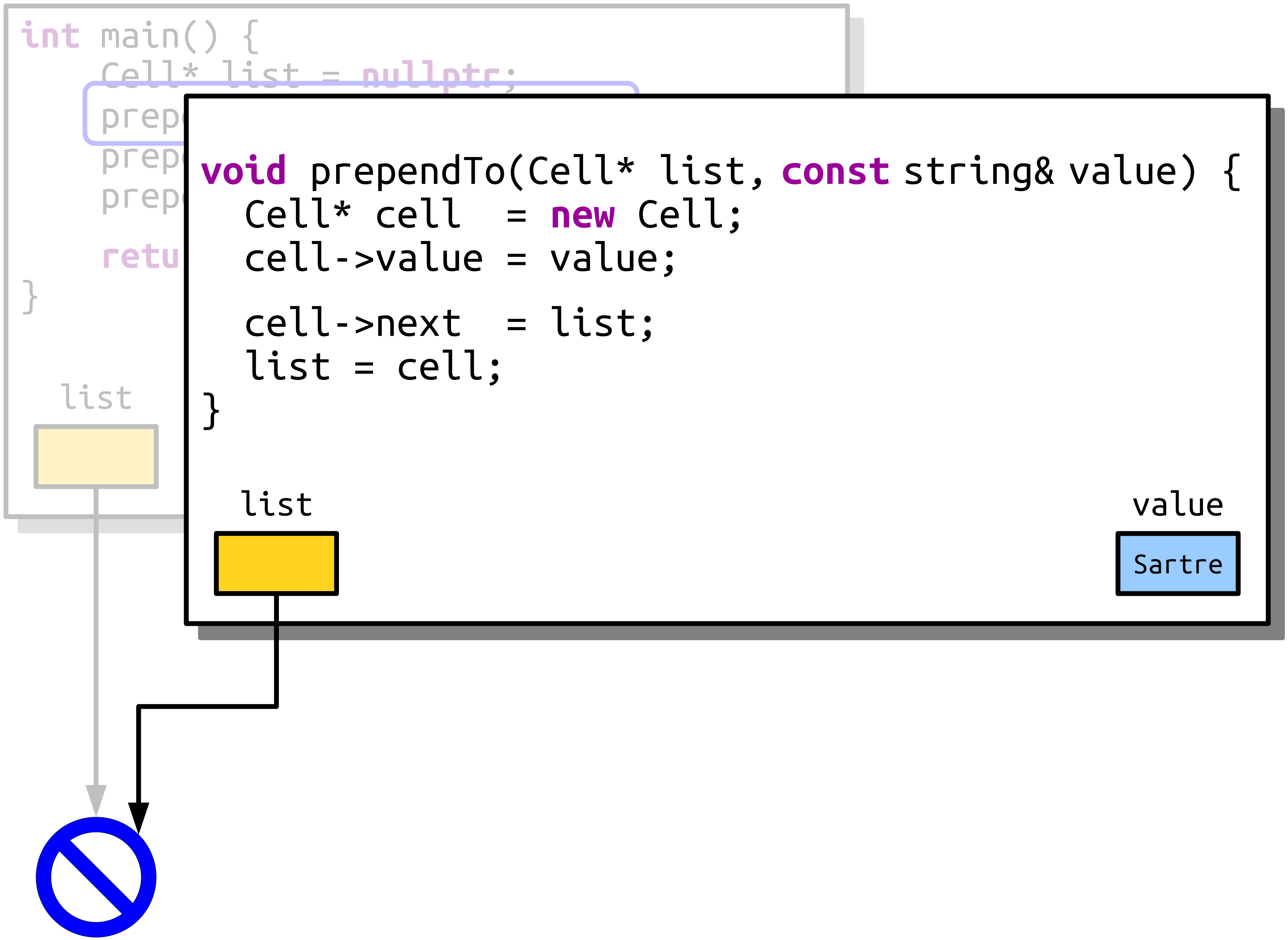


```
int main() {
    Cell* list = nullptr;
    prependTo(list, "Sartre");
    prependTo(list, "Camus");
    prependTo(list, "Nietzsche");

    return 0;
}
```



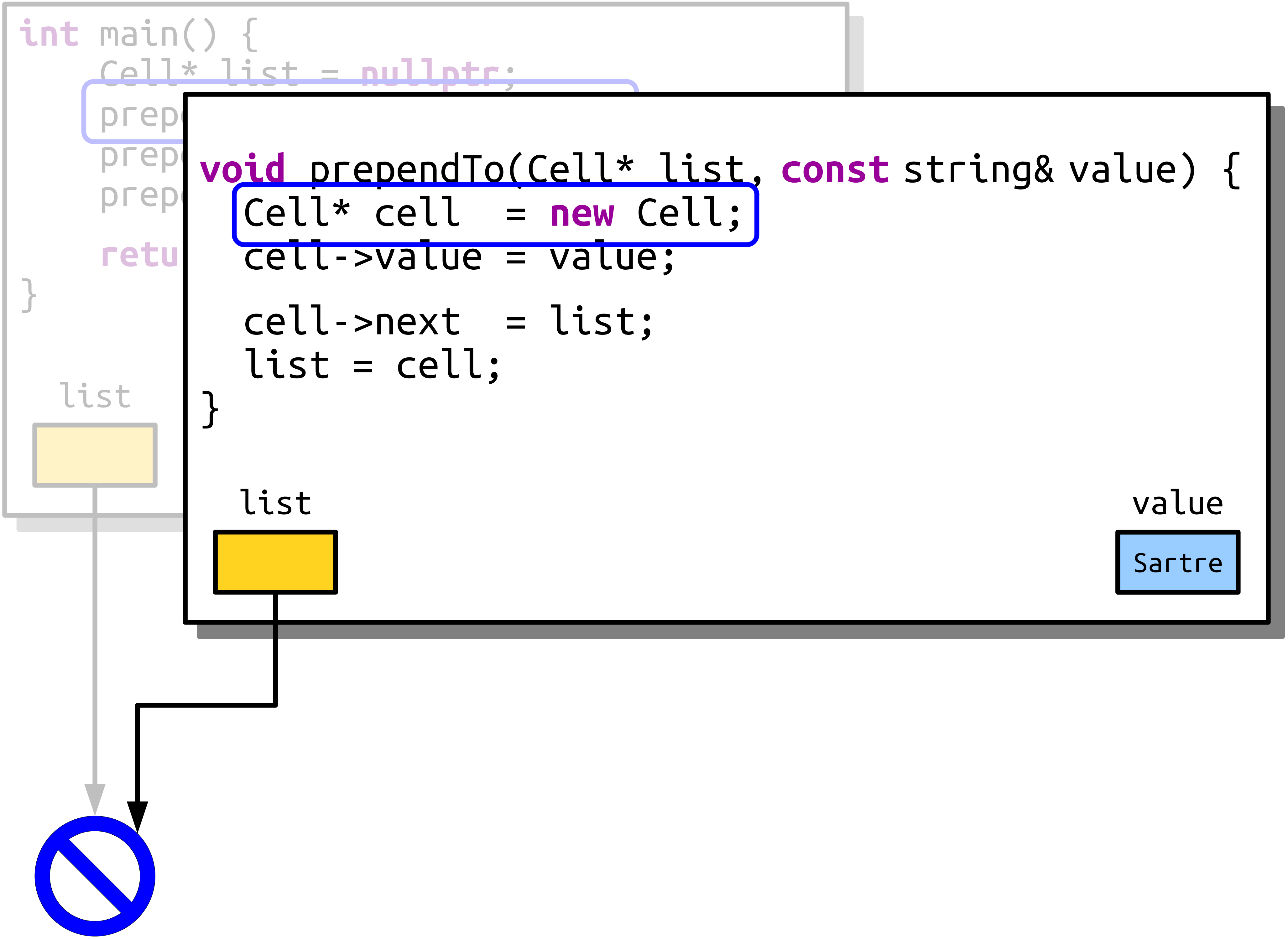
```
int main() {
    Cell* list = nullptr;
    prependTo(list, "Sartre");
    prependTo(list, "Camus");
    prependTo(list, "Nietzsche");

    return 0;
}
```





```
int main() {
    Cell* list = nullptr;
    prepe
    prepe
    prepe

    retu
}
```

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

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

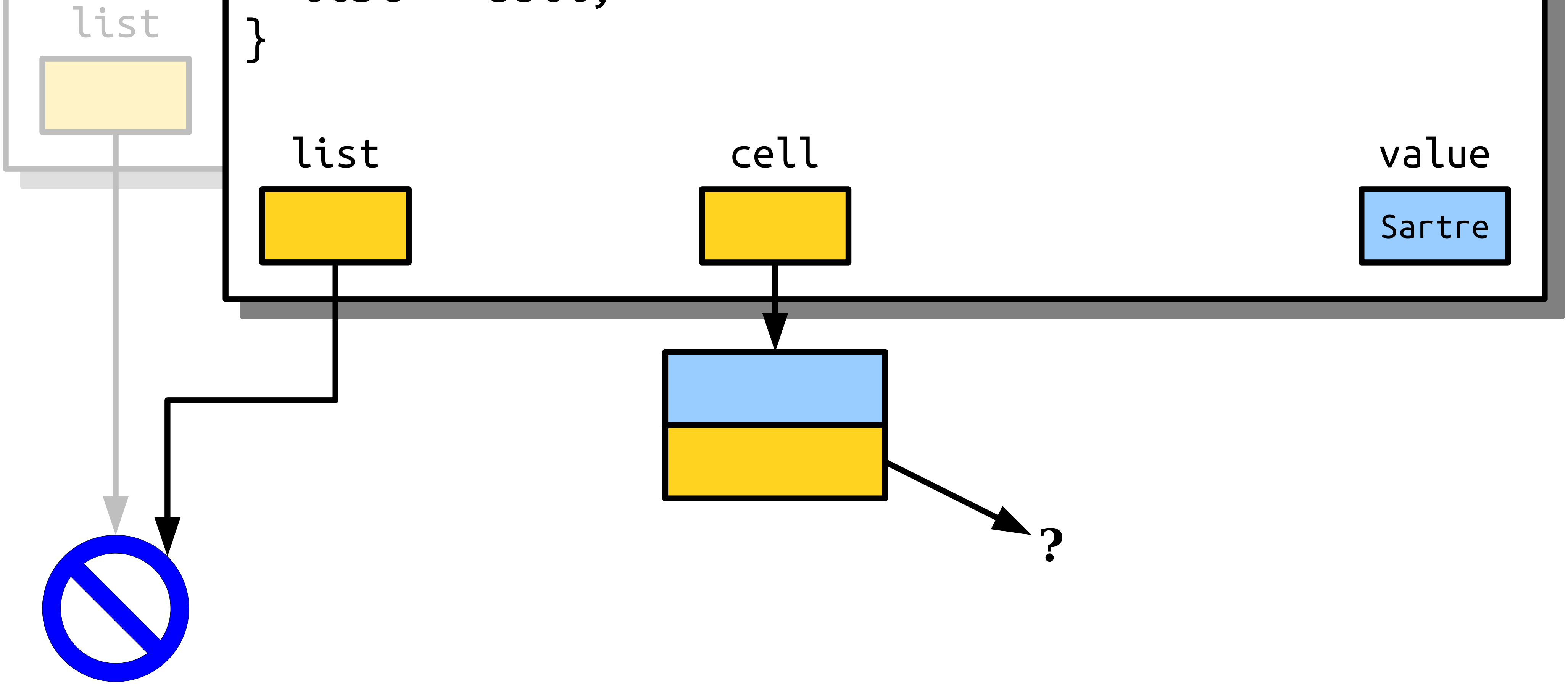


```
int main() {
    Cell* list = nullptr;
    prepe
    prepe
    prepe

    retu
}
```

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

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

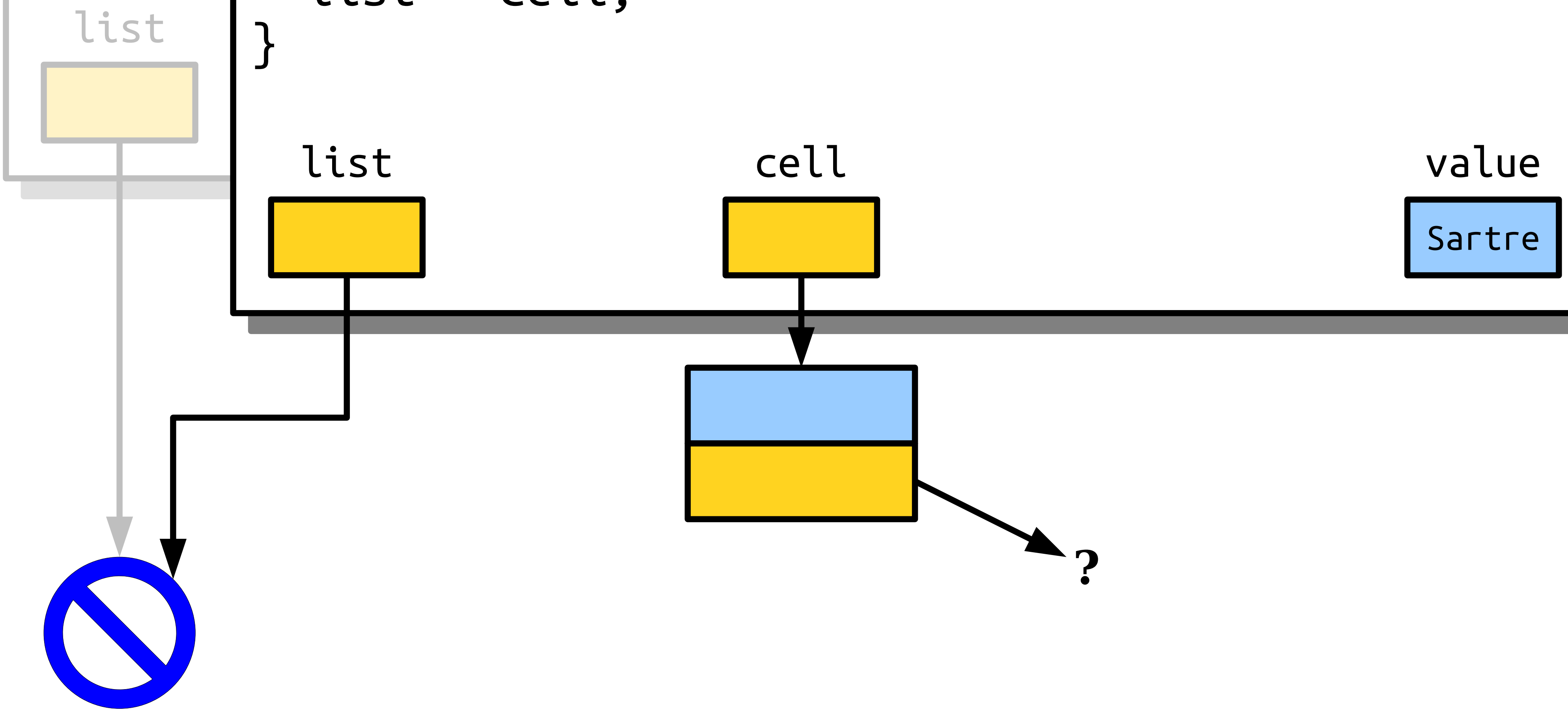
```
int main() {
    Cell* list = nullptr;
    prepe
    prepe
    prepe

    retu
}
```

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

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



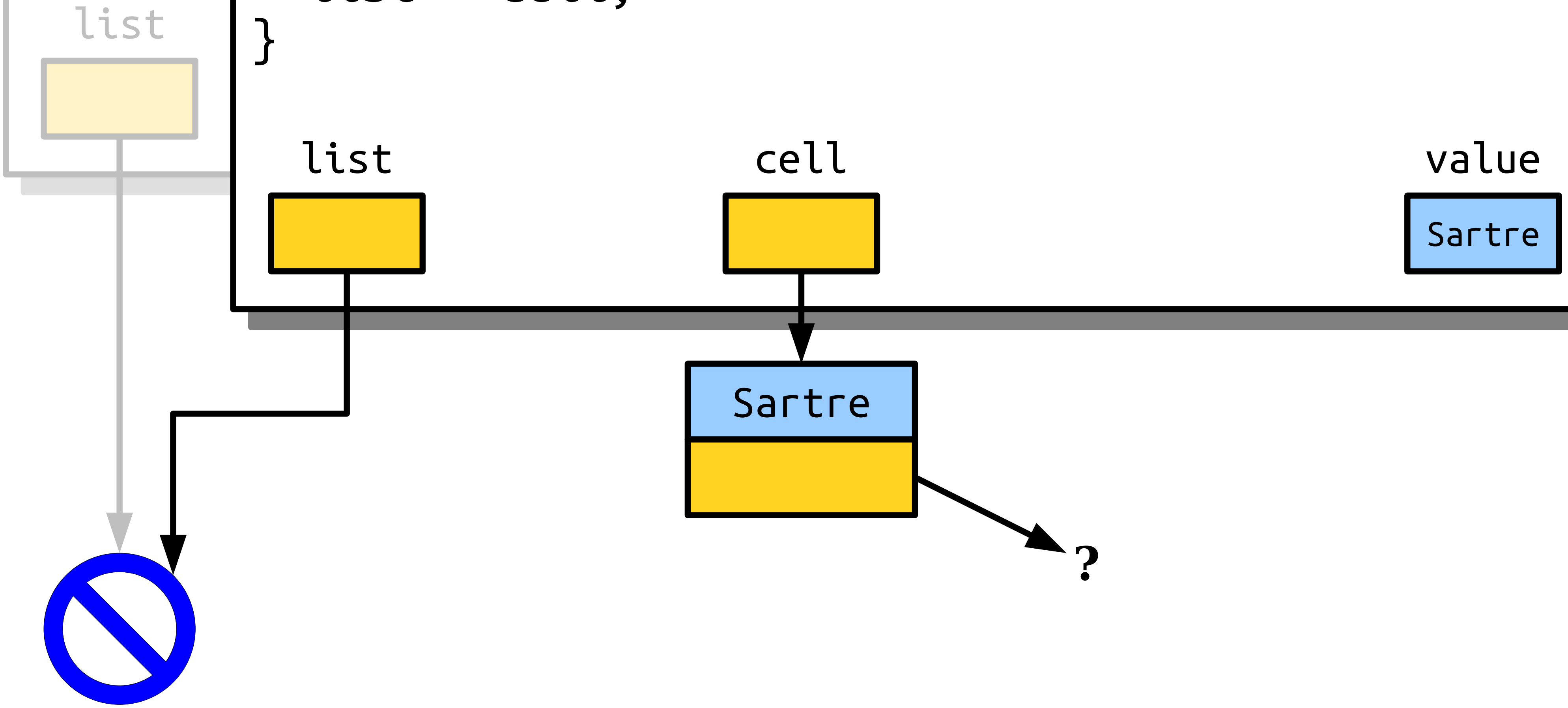
```
int main() {
    Cell* list = nullptr;
    prepe
    prepe
    prepe

    retu
}
```

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

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



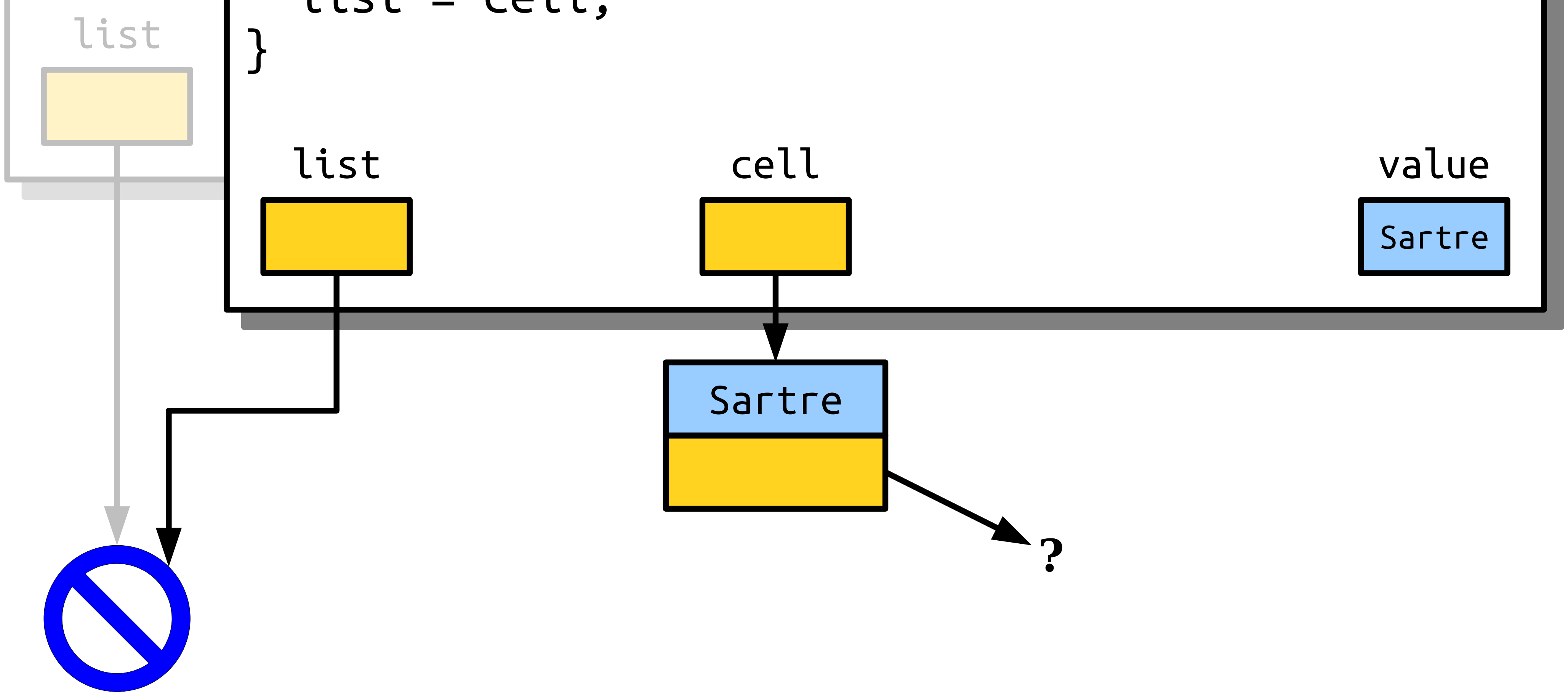
```
int main() {
    Cell* list = nullptr;
    prepe
    prepe
    prepe

    retu
}
```

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

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



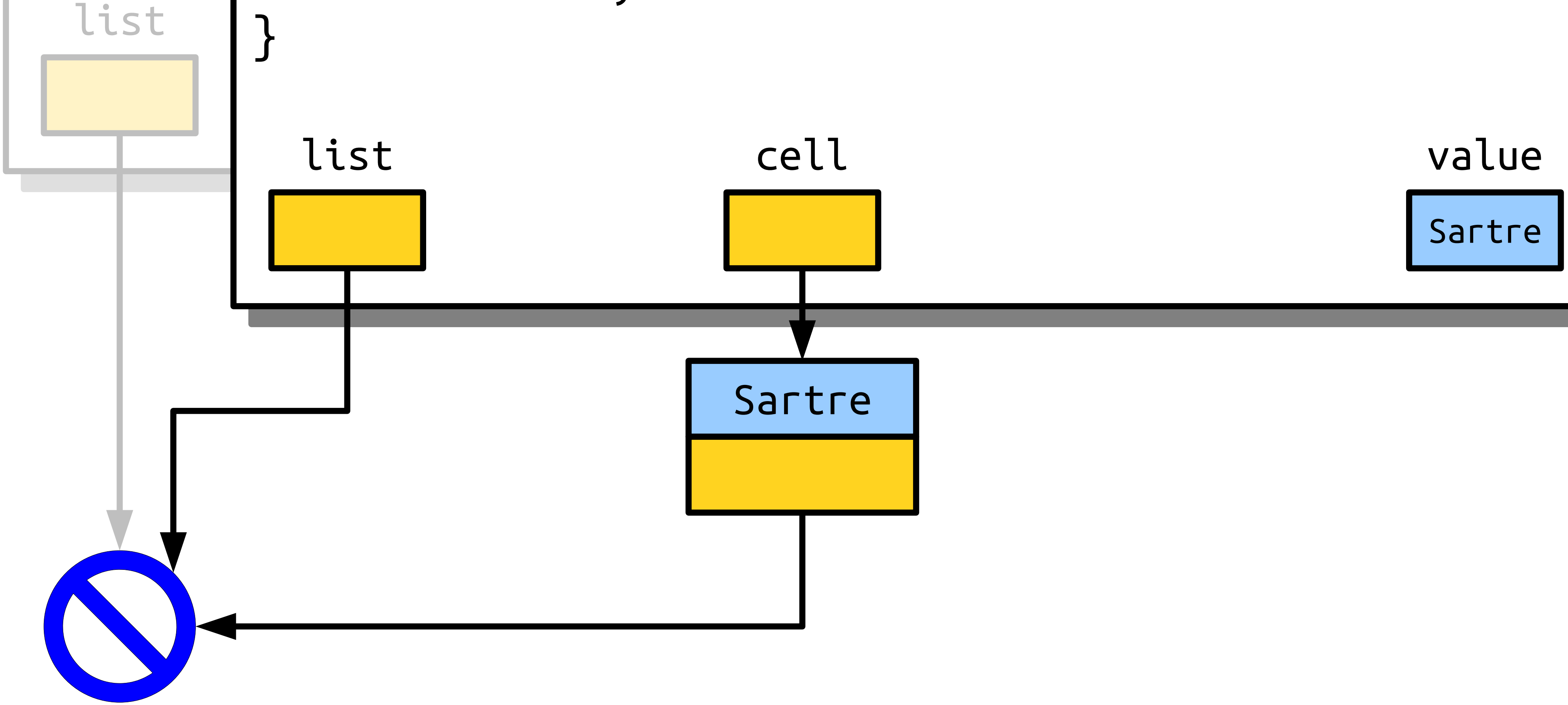
```
int main() {
    Cell* list = nullptr;
    prepe
    prepe
    prepe

    retu
}
```

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

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



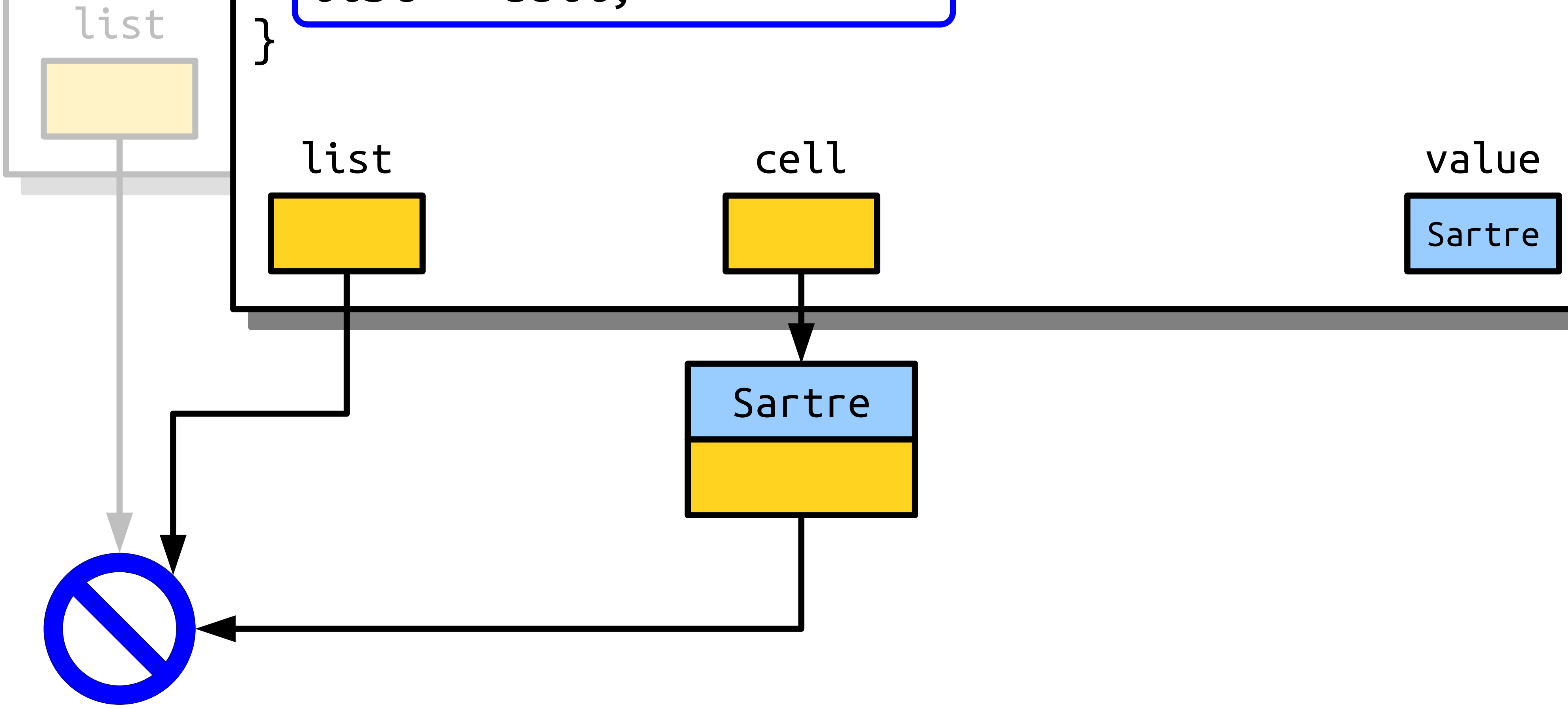
```
int main() {
    Cell* list = nullptr;
    prepe
    prepe
    prepe

    retu
}
```

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

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



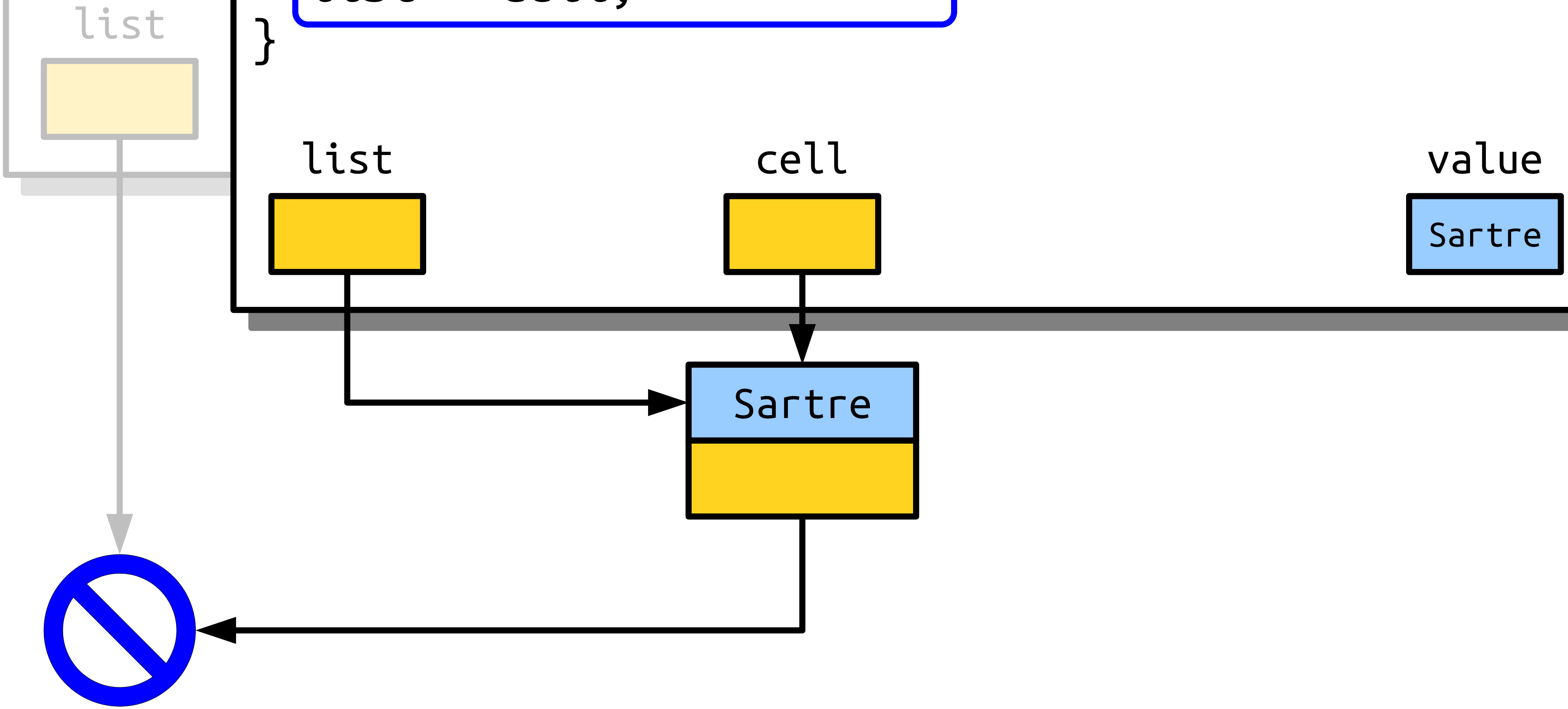
```
int main() {
    Cell* list = nullptr;
    prepe
    prepe
    prepe

    retu
}
```

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

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

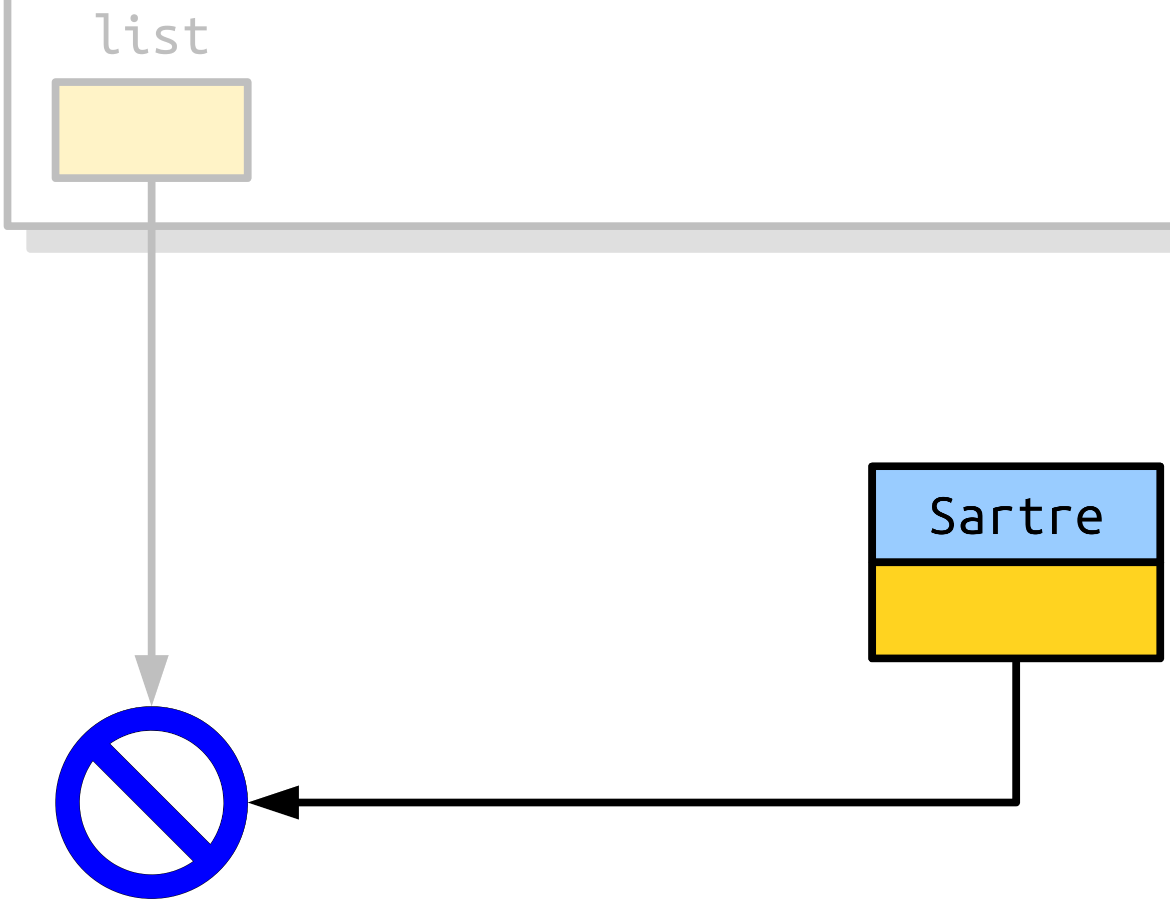


```
int main() {
    Cell* list = nullptr;
    prepe
    prepe
    prepe

    retu
}
```

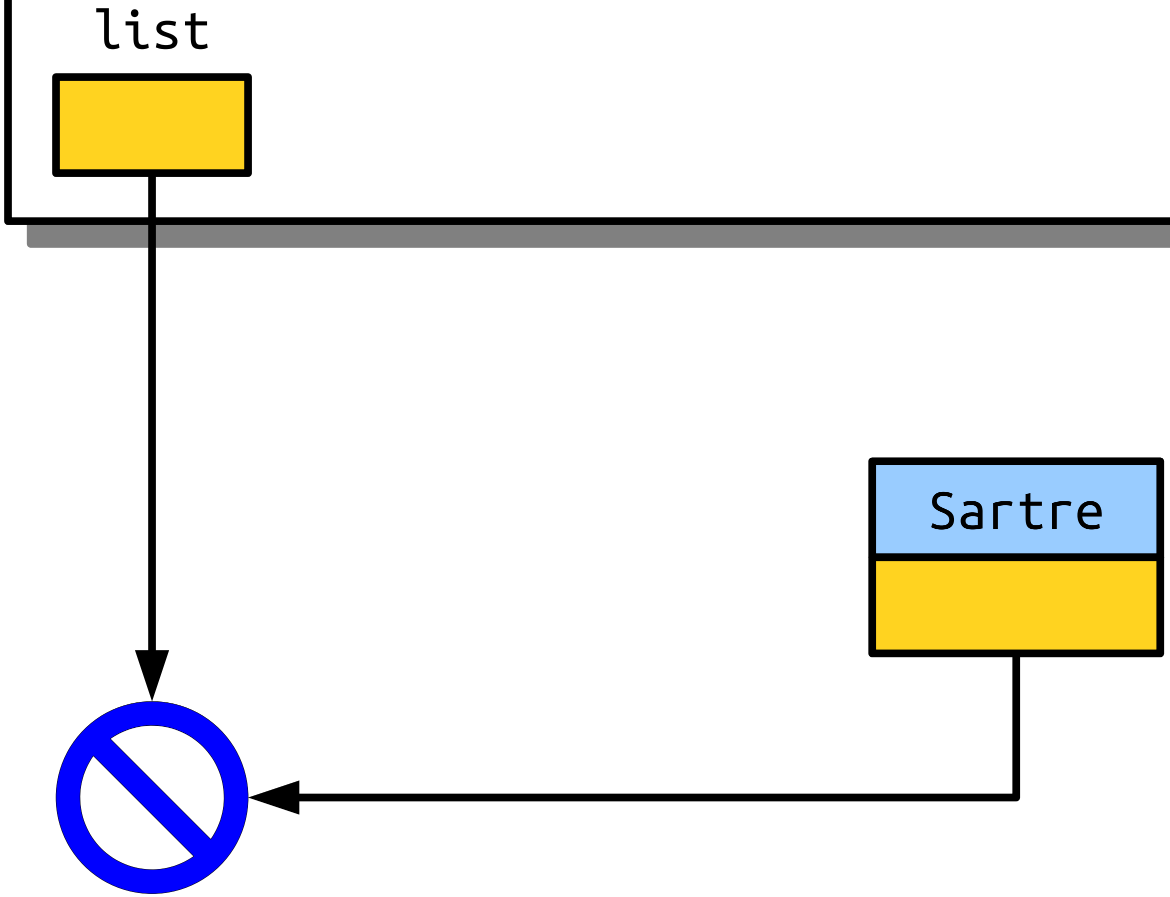
```
void prependTo(Cell* list, const string& value) {
  Cell* cell  = new Cell;
  cell->value = value;

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

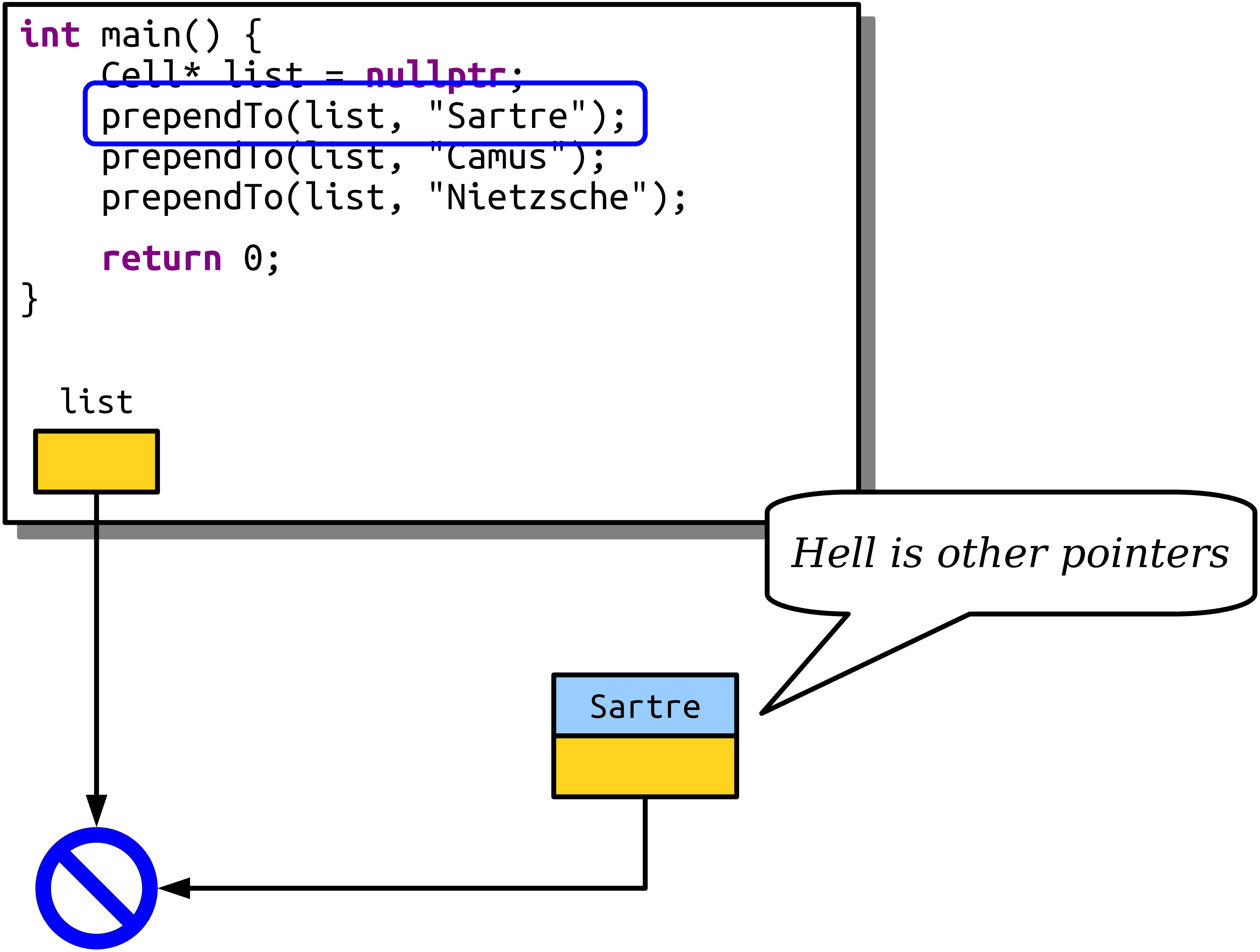


```
int main() {
    Cell* list = nullptr;
    prependTo(list, "Sartre");
    prependTo(list, "Camus");
    prependTo(list, "Nietzsche");

    return 0;
}
```



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

    return 0;
}
```

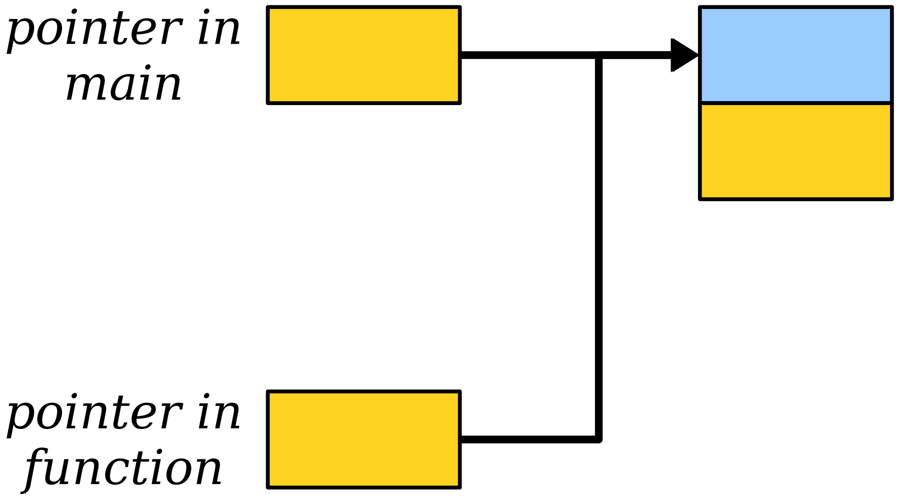




```
int main() {
    Cell* list = nullptr;
    prependTo(list, "Sartre");
    prependTo(list, "Camus");
    prependTo(list, "Nietzsche");

    return 0;
}
```

Pointers By Value

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



Pointers by Reference

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

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

Pointers by Reference

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

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

Pointers by Reference

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

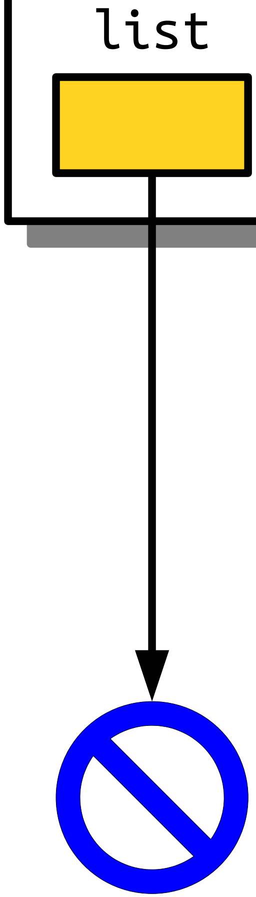
```
void prependTo(Cell*& list, const string& value) {
    Cell* cell = new Cell;
    cell->value = value;
    cell->next = list;
    list = cell;
}
```

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

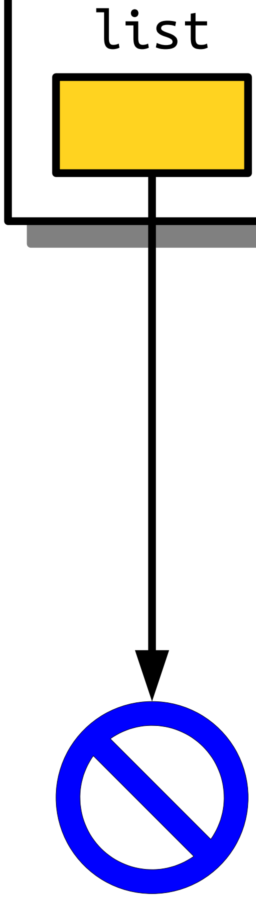
```
int main() {
    Cell* list = nullptr;
    prependTo(list, "Descartes");
    prependTo(list, "Kant");
    prependTo(list, "Bentham");

    return 0;
}
```

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

    return 0;
}
```

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

    return 0;
}
```

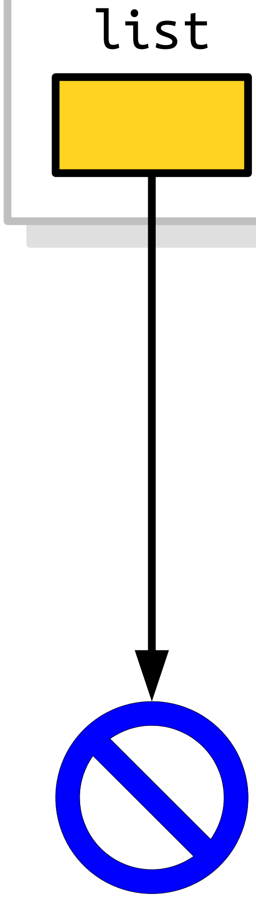


```
int main() {
    Cell* list = nullptr;
    prependTo(list, "Descartes");
    prependTo(list, "Kant");
    prependTo(list, "Bentham");

    return 0;
}
```



```
int main() {
    Cell* list = nullptr;
    prepe
    prepe
    prepe

    retu
}
```

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

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

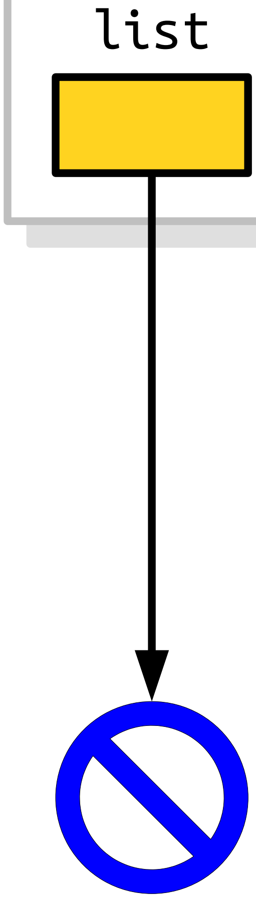


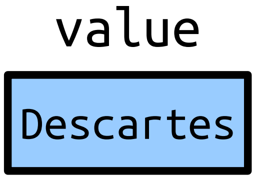
value

Descartes

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

    retu
}
```

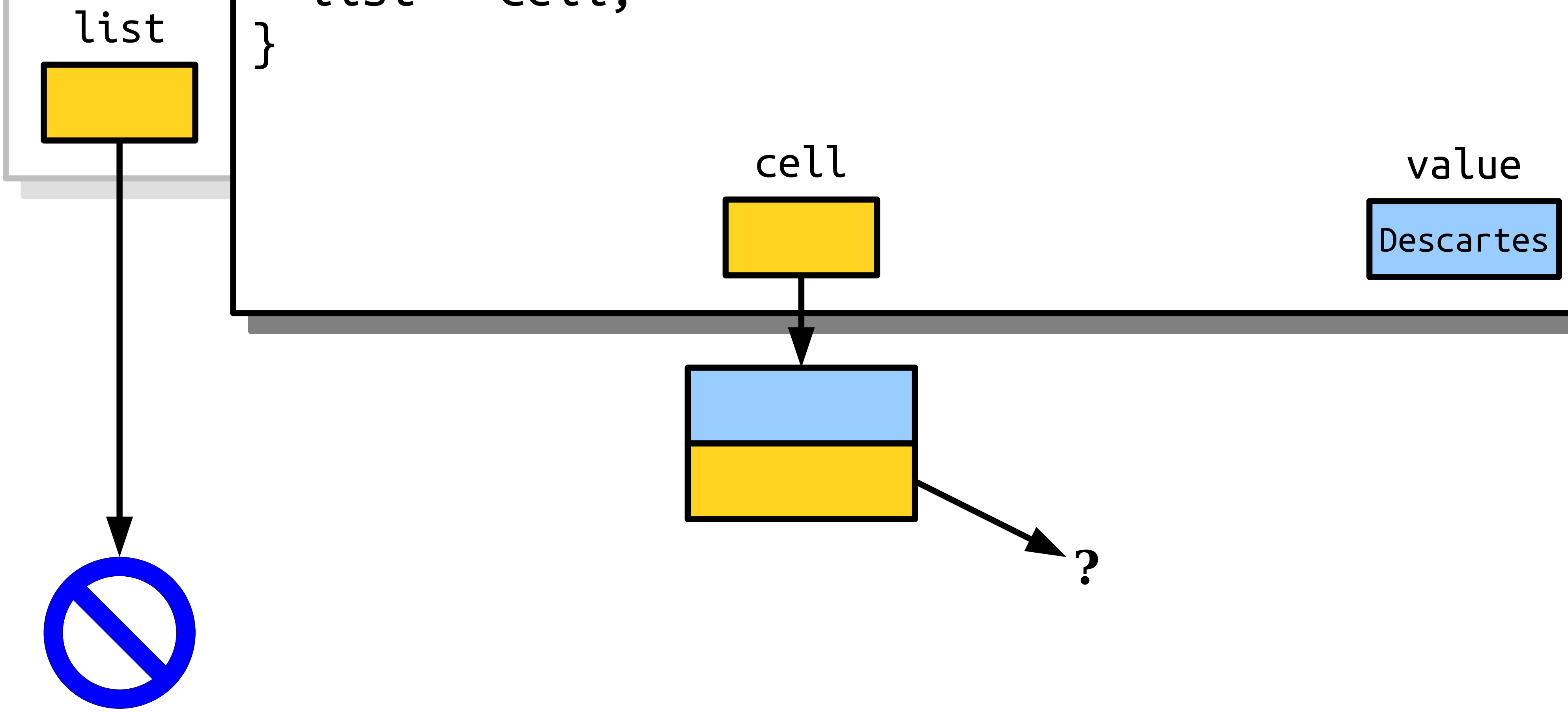
```
void prependTo(Cell*& list, const string& value) {
  Cell* cell  = new Cell;
  cell->value = value;

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





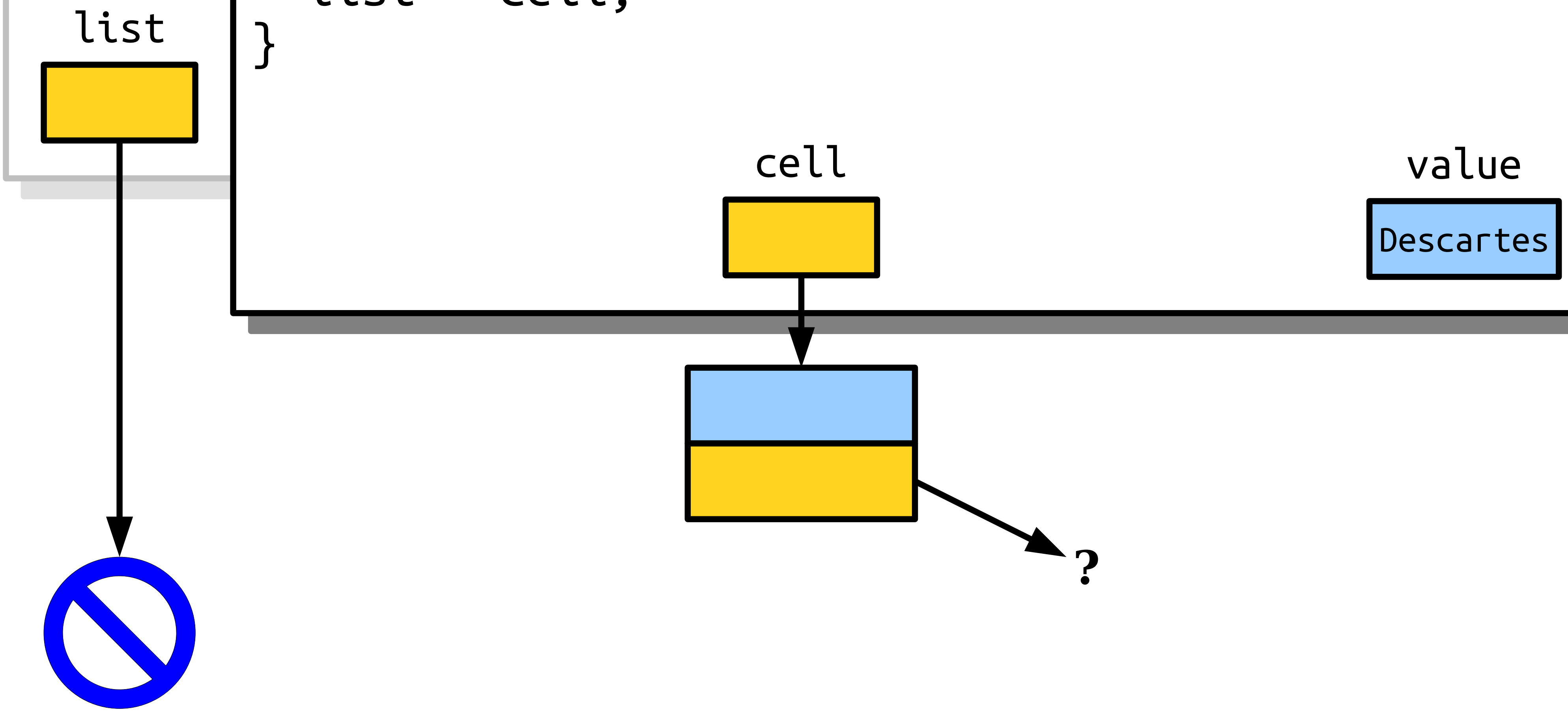
```
int main() {
    Cell* list = nullptr;
    prepe
    prepe
    prepe

    retu
}
```

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

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



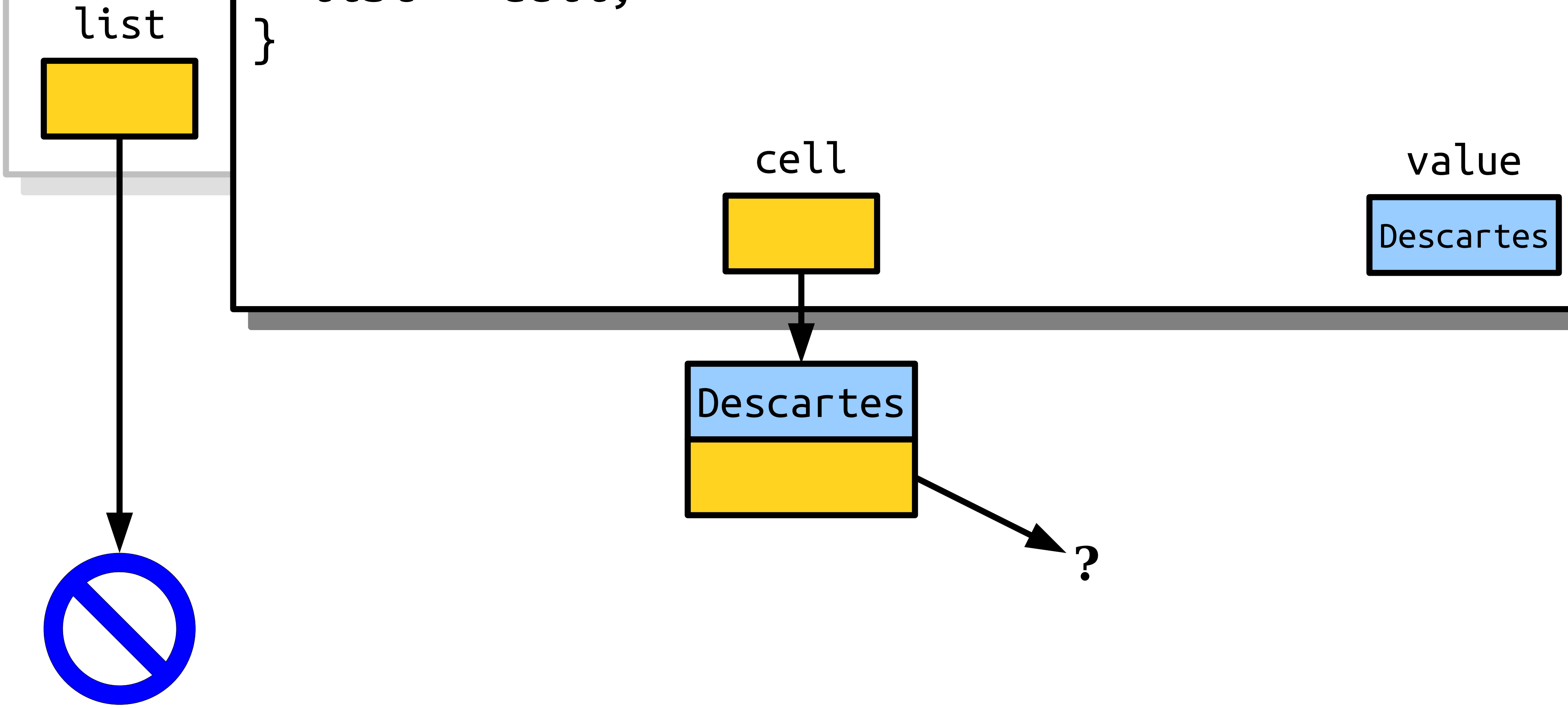
```
int main() {
    Cell* list = nullptr;
    prepe
    prepe
    prepe

    retu
}
```

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

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



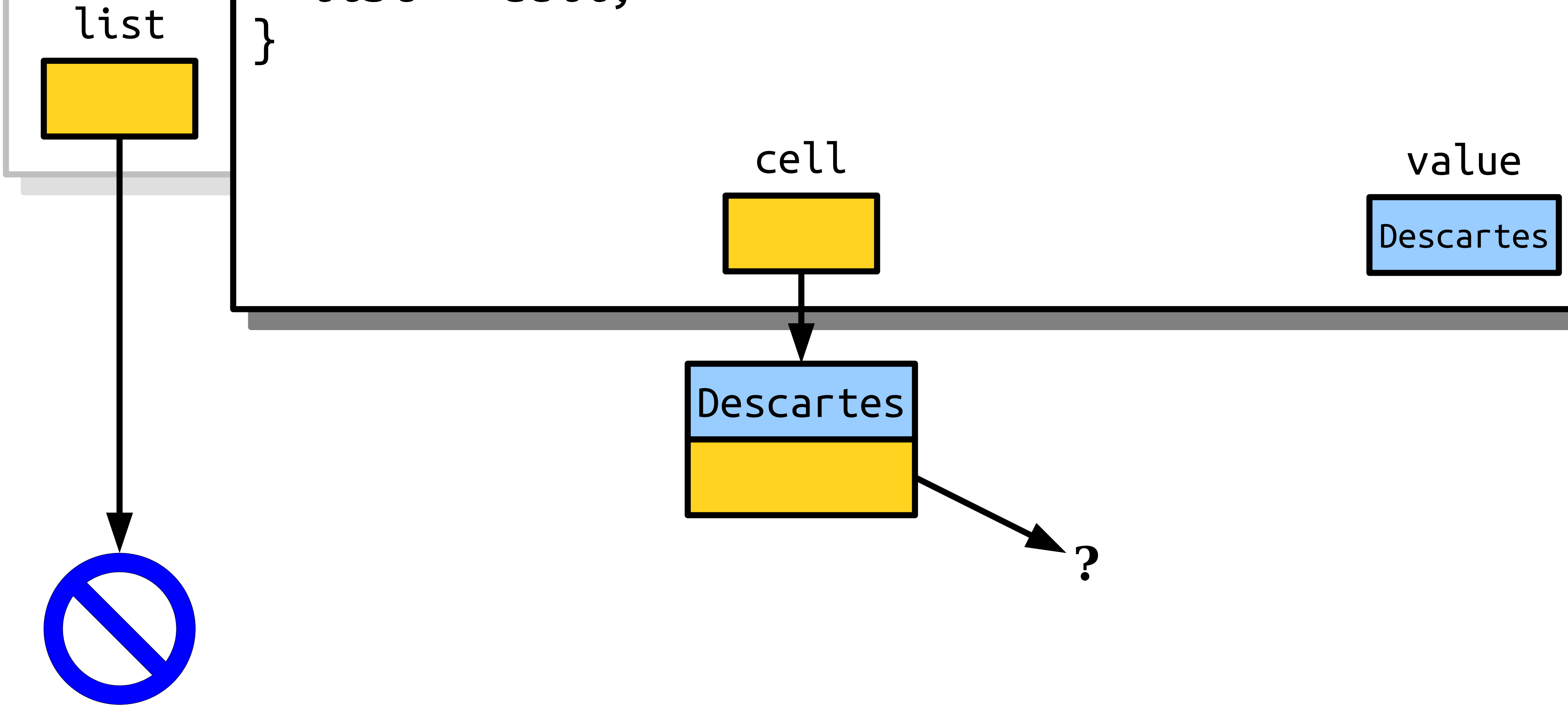
```
int main() {
    Cell* list = nullptr;
    prep
    prep
    prep

    retu
}
```

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

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



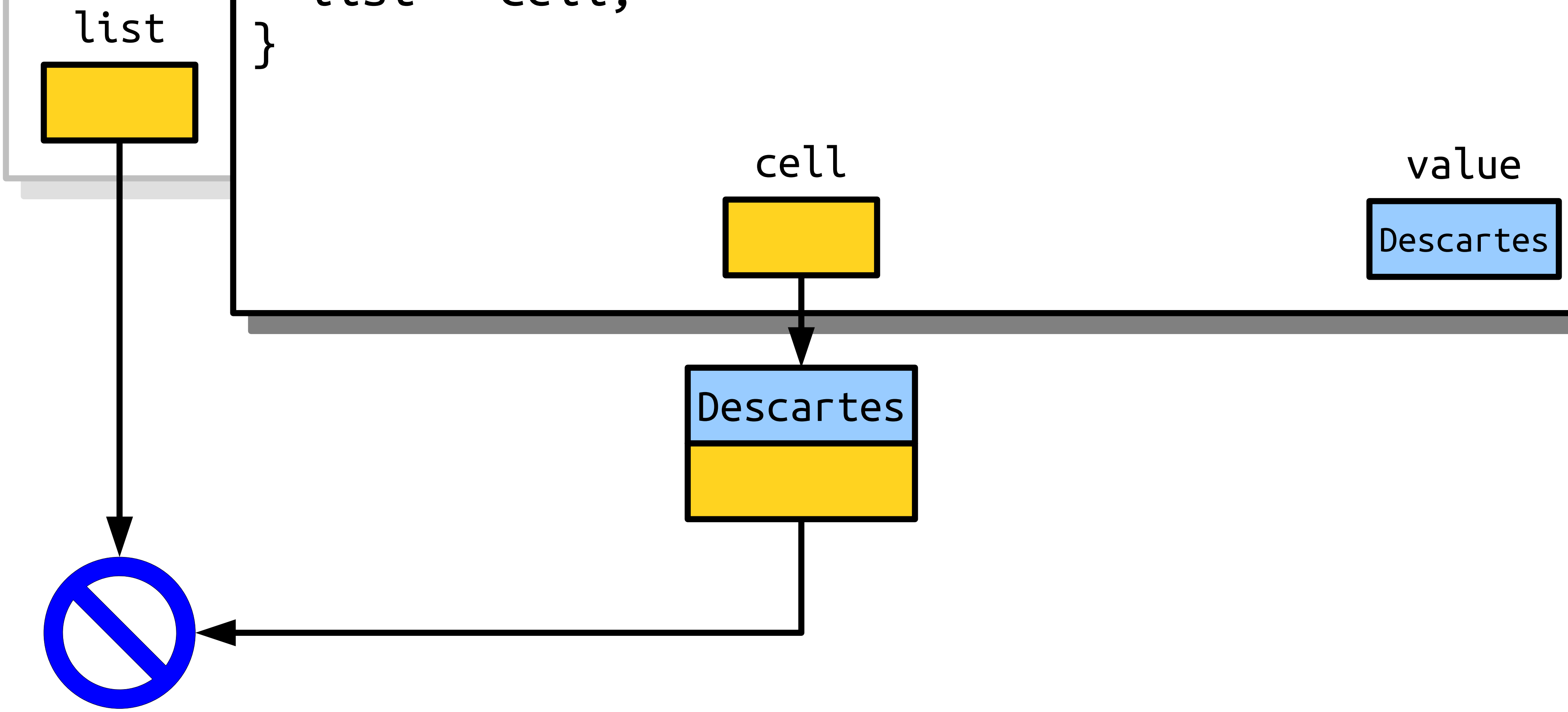
```
int main() {
    Cell* list = nullptr;
    prepe
    prepe
    prepe

    retu
}
```

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

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



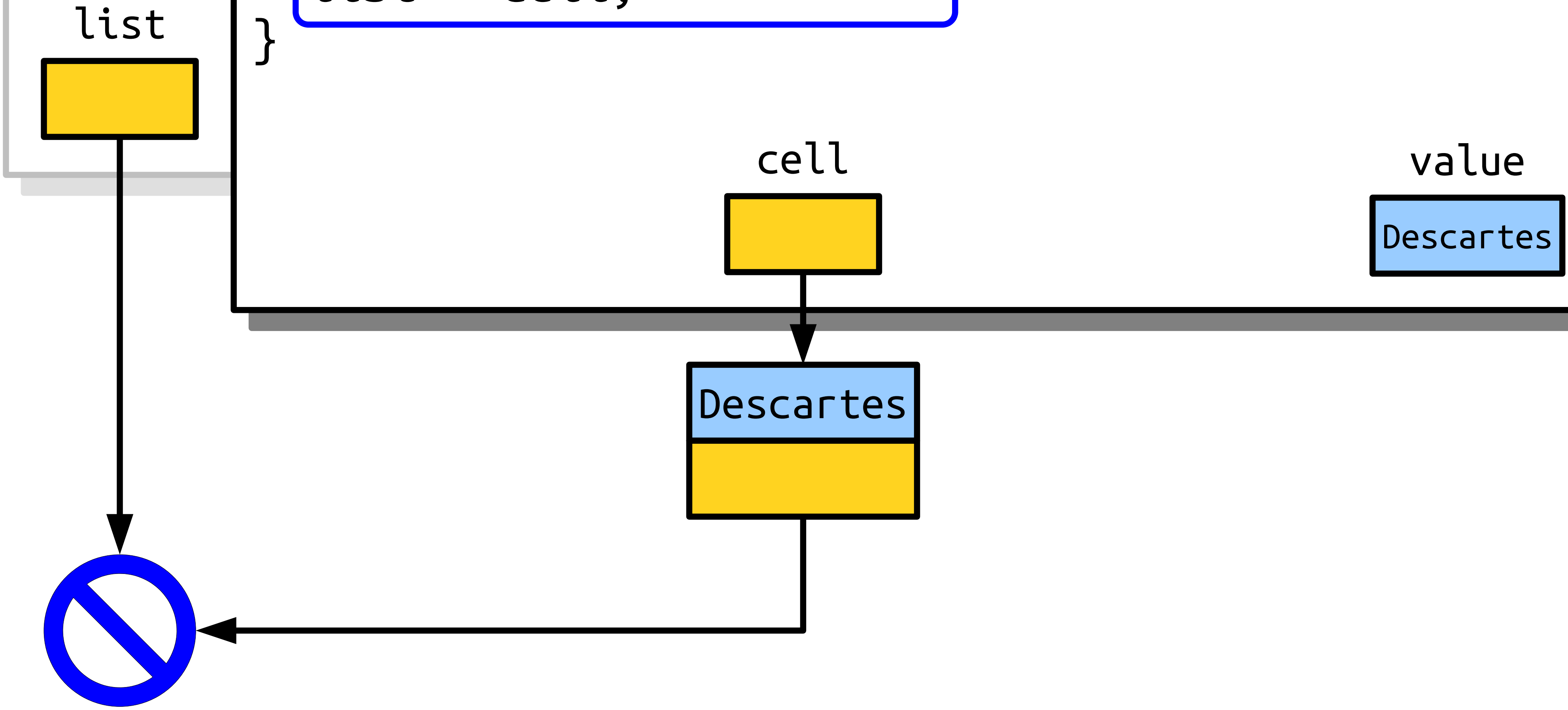
```
int main() {
    Cell* list = nullptr;
    prepe
    prepe
    prepe

    retu
}
```

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

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



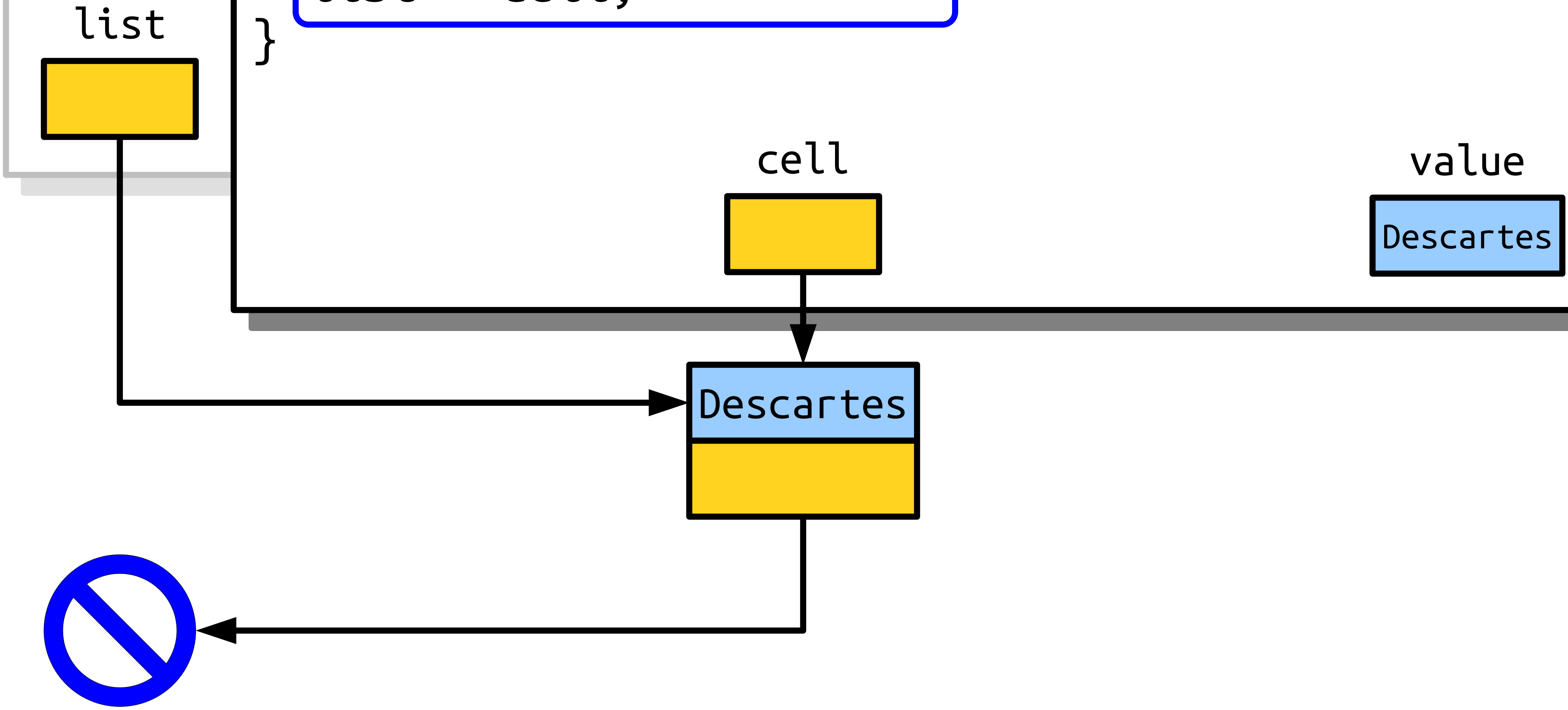
```
int main() {
    Cell* list = nullptr;
    prepe
    prepe
    prepe

    retu
}
```

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

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

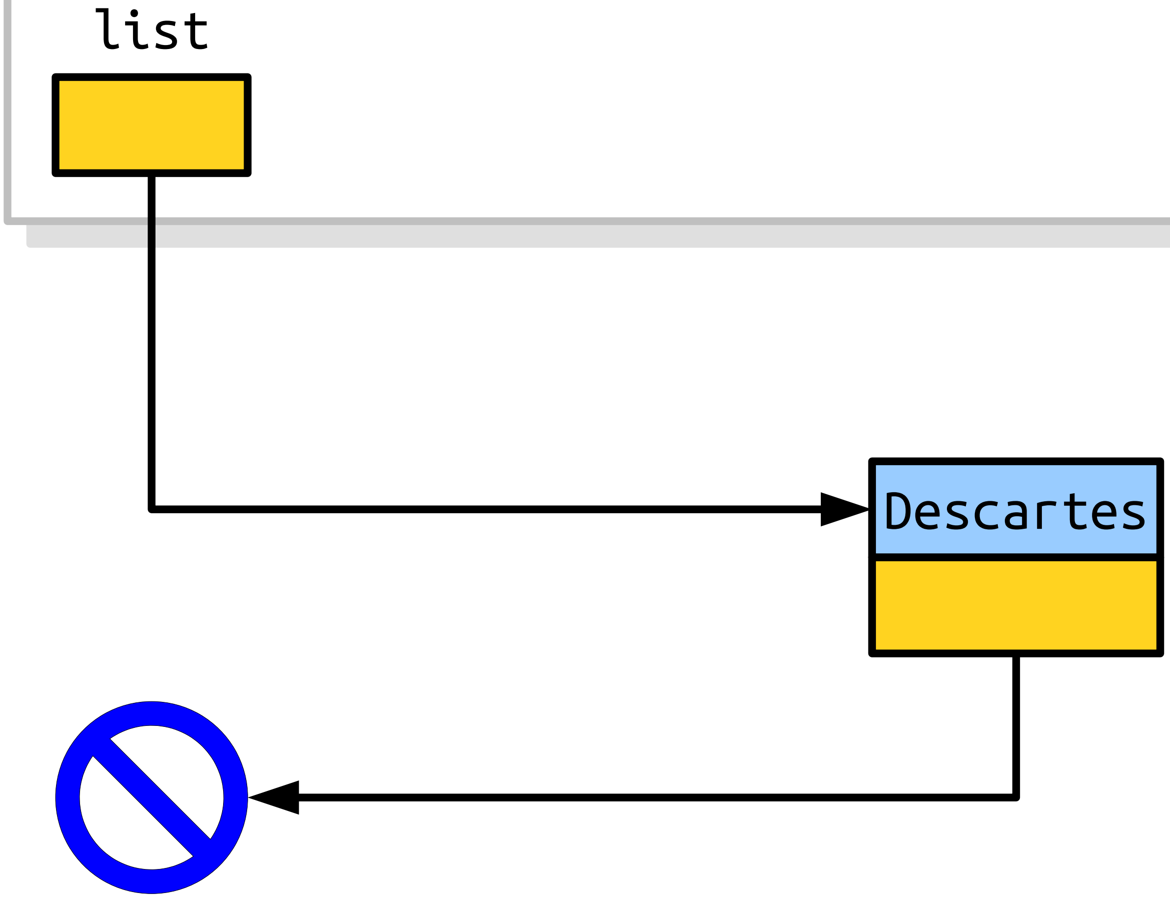


```
int main() {
    Cell* list = nullptr;
    prep
    prep
    prep

    retu
}
```

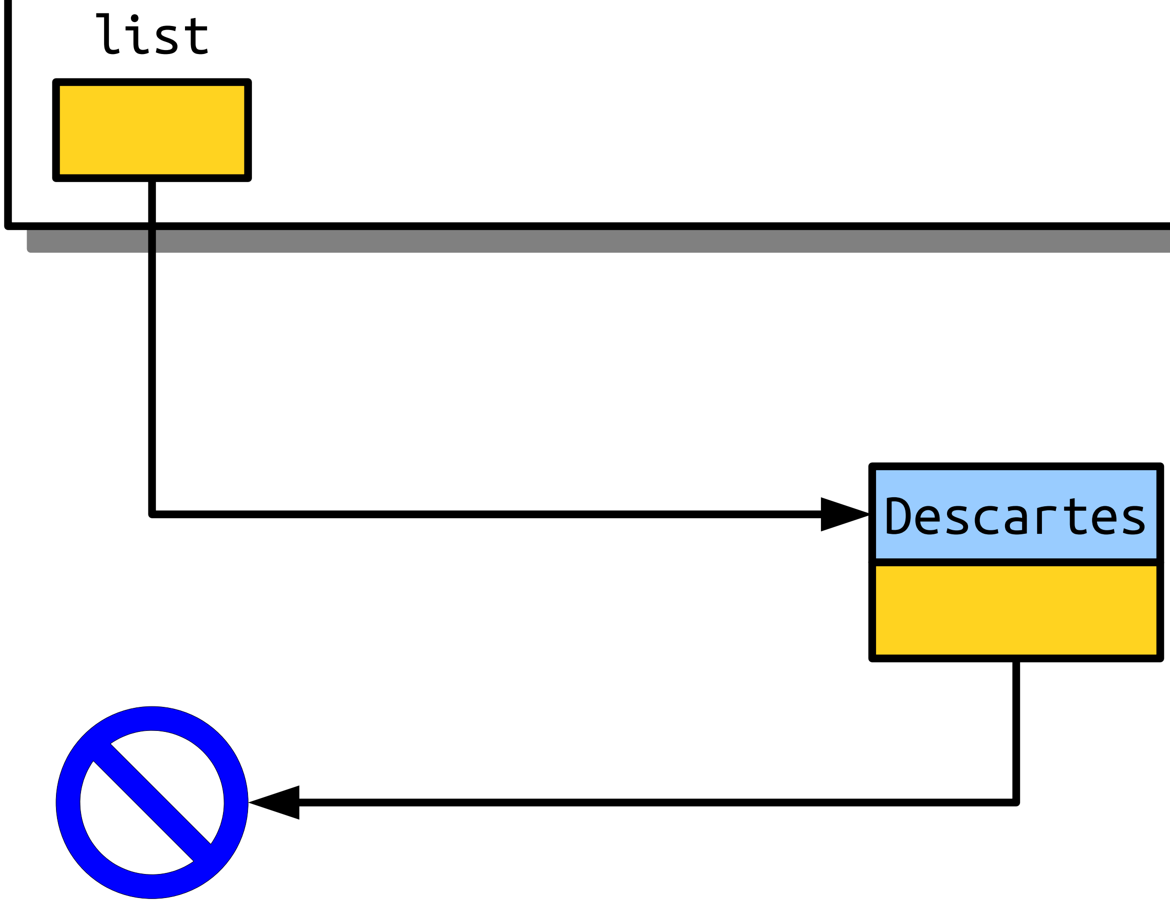
```
void prependTo(Cell*& list, const string& value) {
  Cell* cell  = new Cell;
  cell->value = value;

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



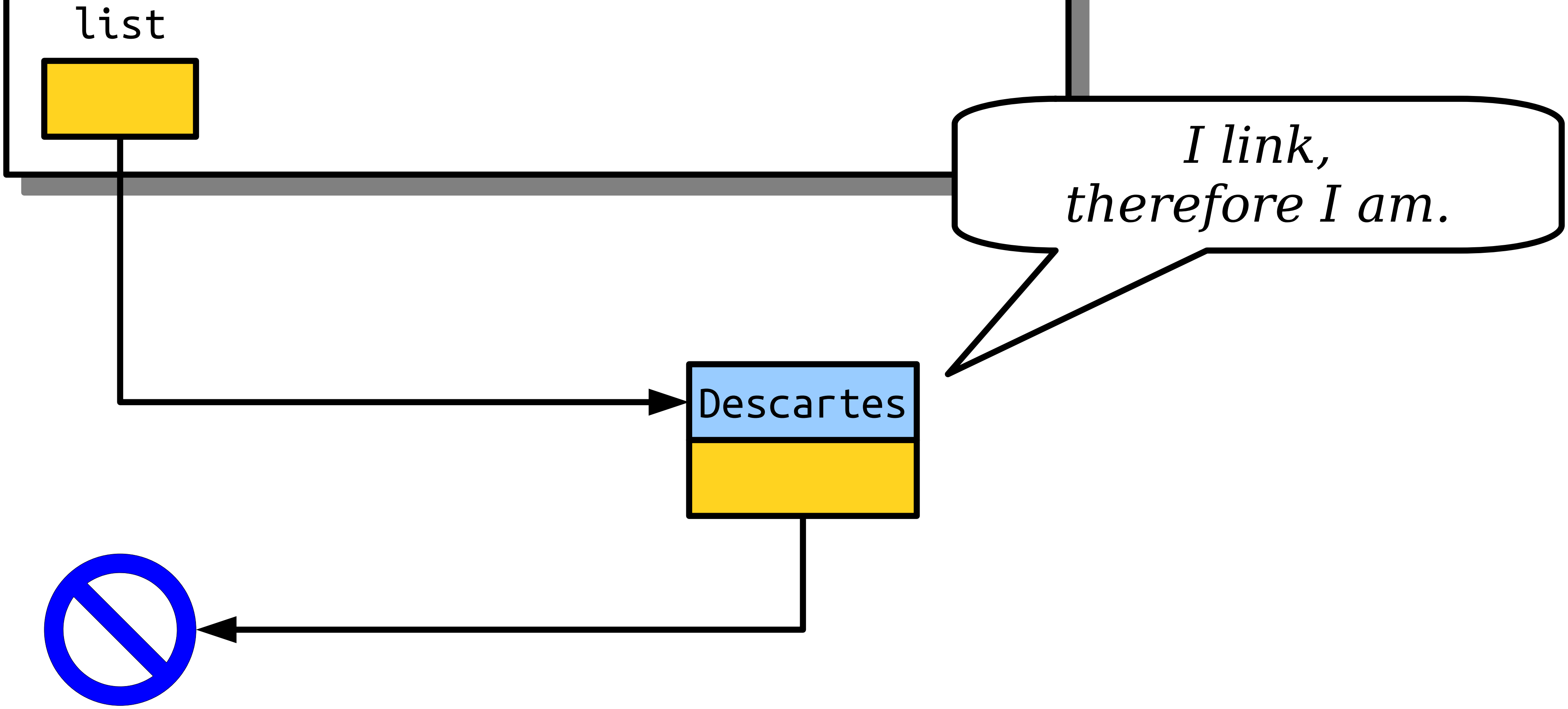
```
int main() {
    Cell* list = nullptr;
    prependTo(list, "Descartes");
    prependTo(list, "Kant");
    prependTo(list, "Bentham");

    return 0;
}
```



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

    return 0;
}
```



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

    return 0;
}
```



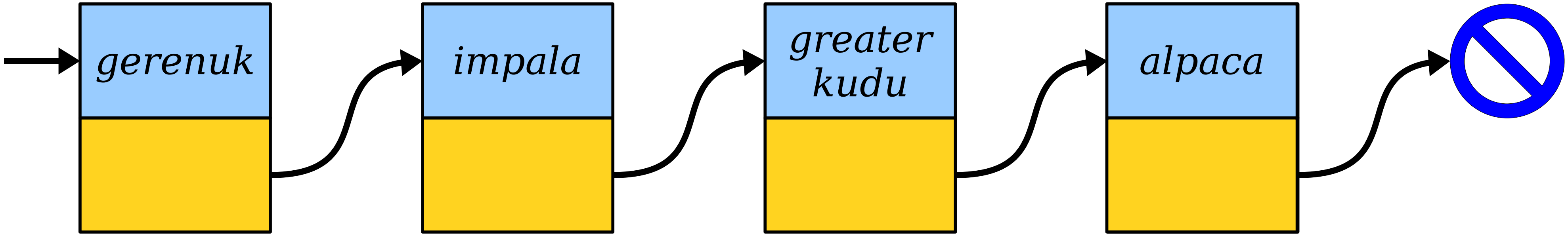
Pointers by Reference

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

Appending to a List

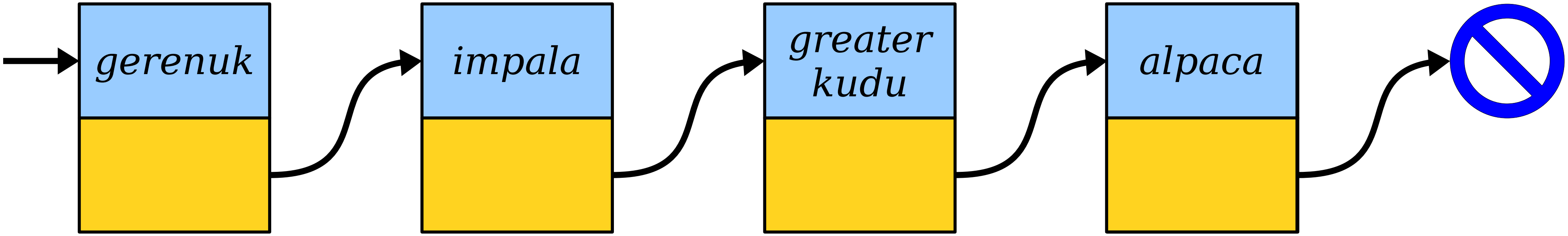
Appending to a List

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



Appending to a List

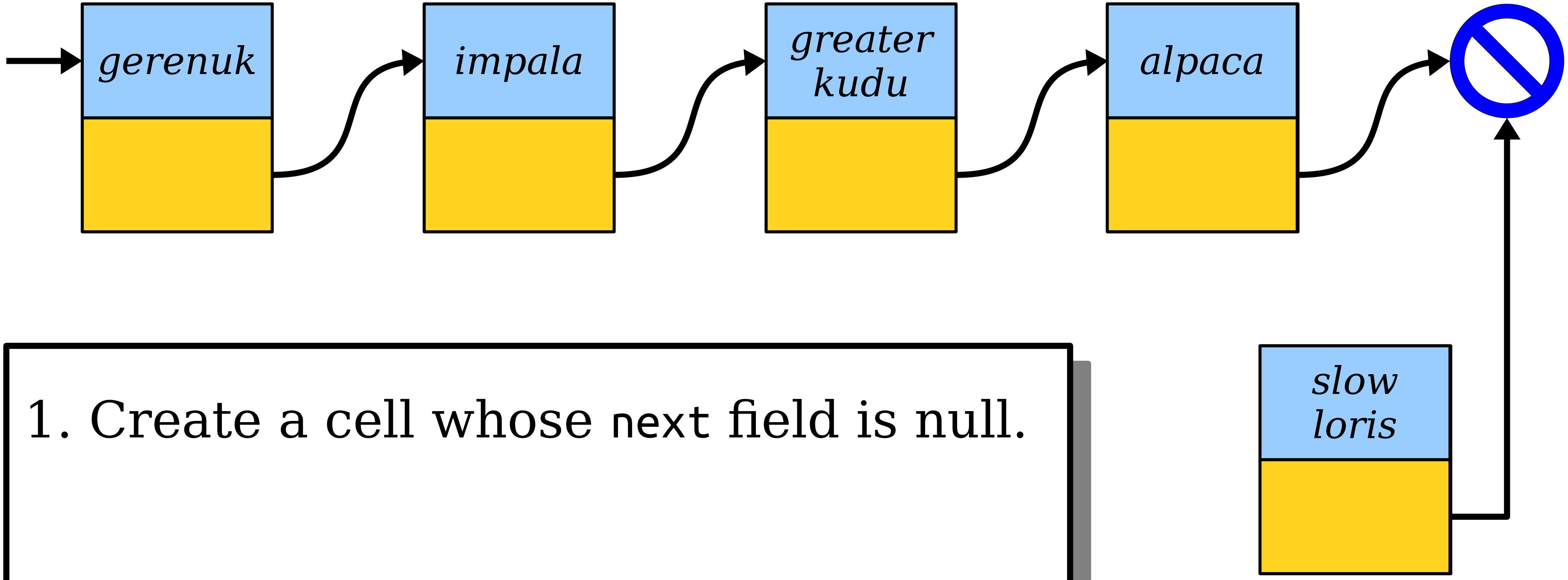
- Think about which link needs to get changed to append something to this list:



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

Appending to a List

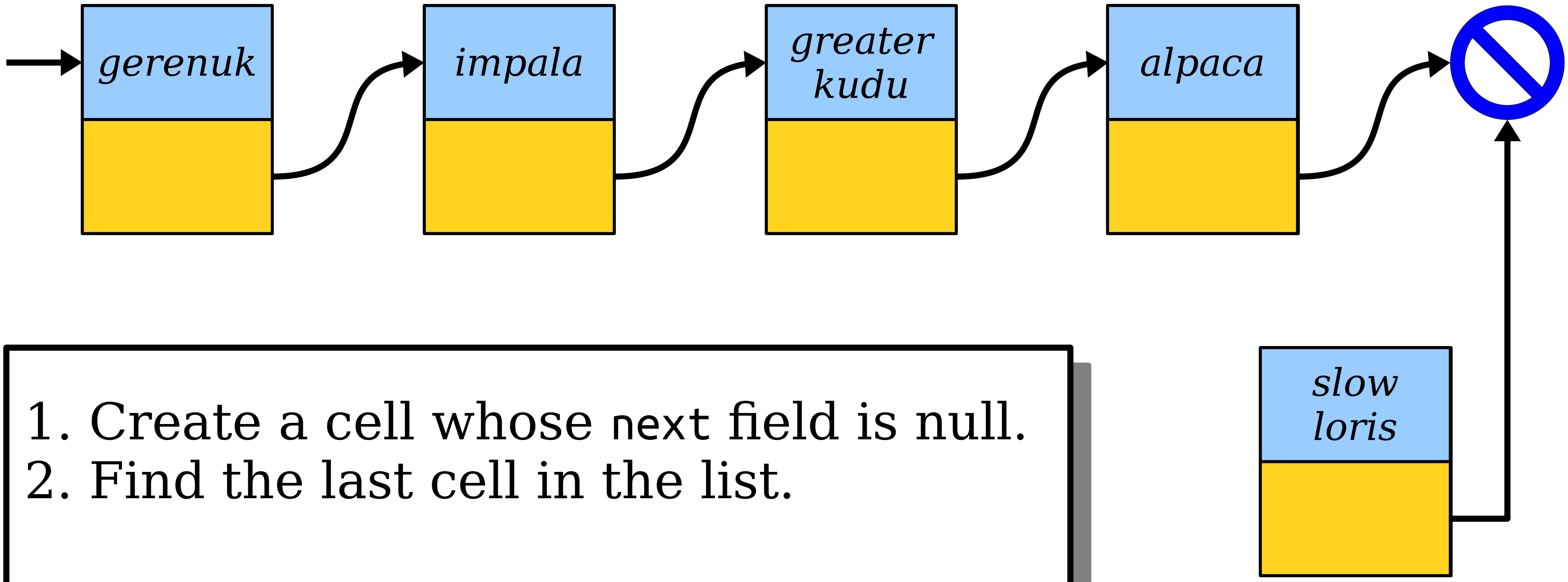
- Think about which link needs to get changed to append something to this list:



1. Create a cell whose next field is null.

Appending to a List

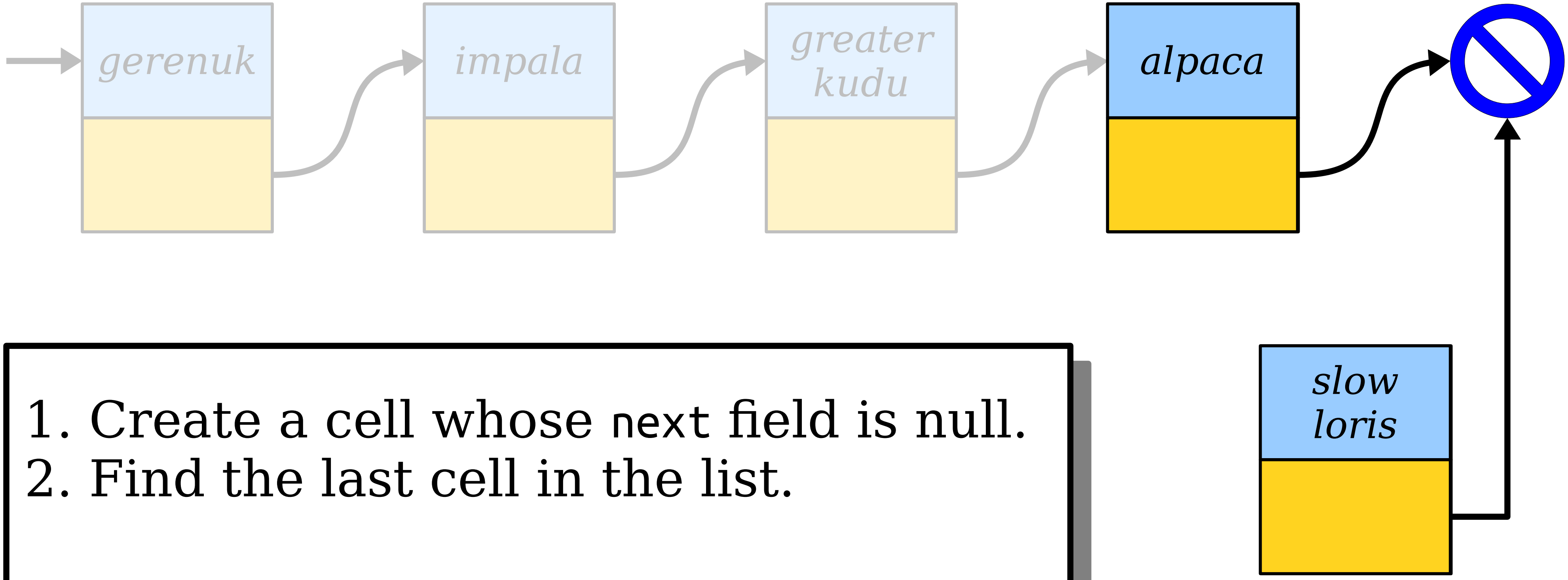
- Think about which link needs to get changed to append something to this list:



1. Create a cell whose next field is null.
2. Find the last cell in the list.

Appending to a List

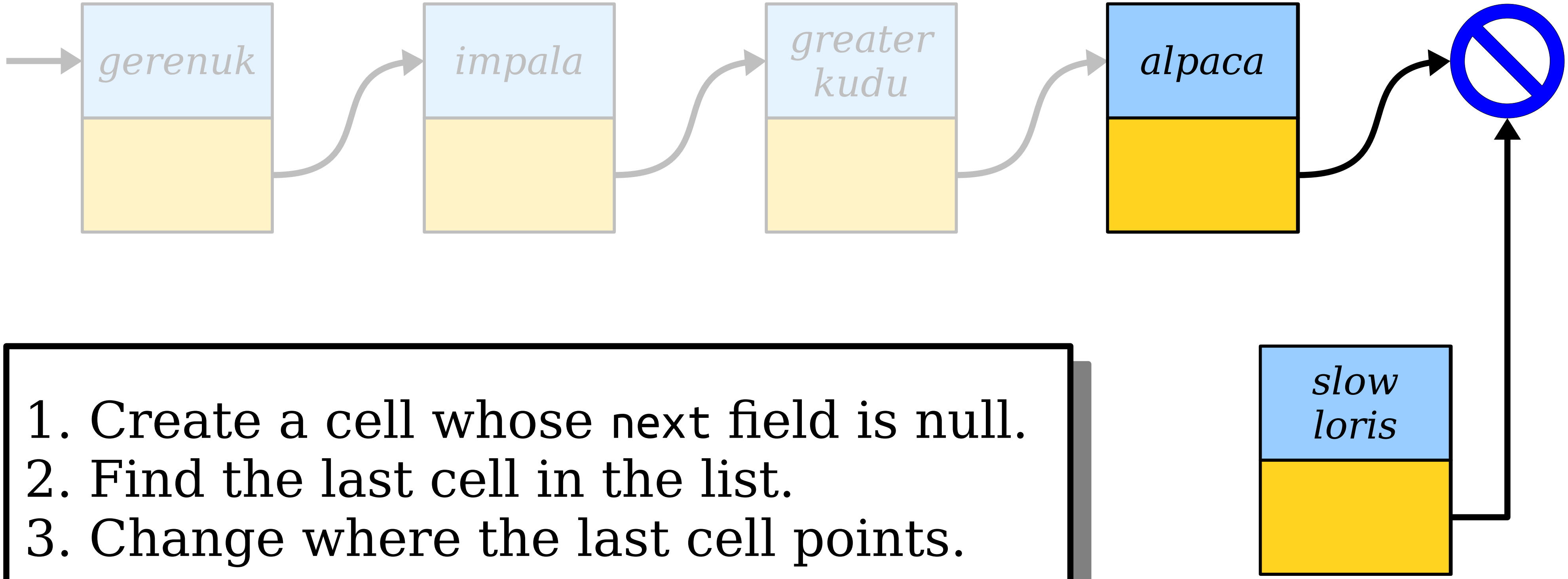
- Think about which link needs to get changed to append something to this list:



1. Create a cell whose next field is null.
2. Find the last cell in the list.

Appending to a List

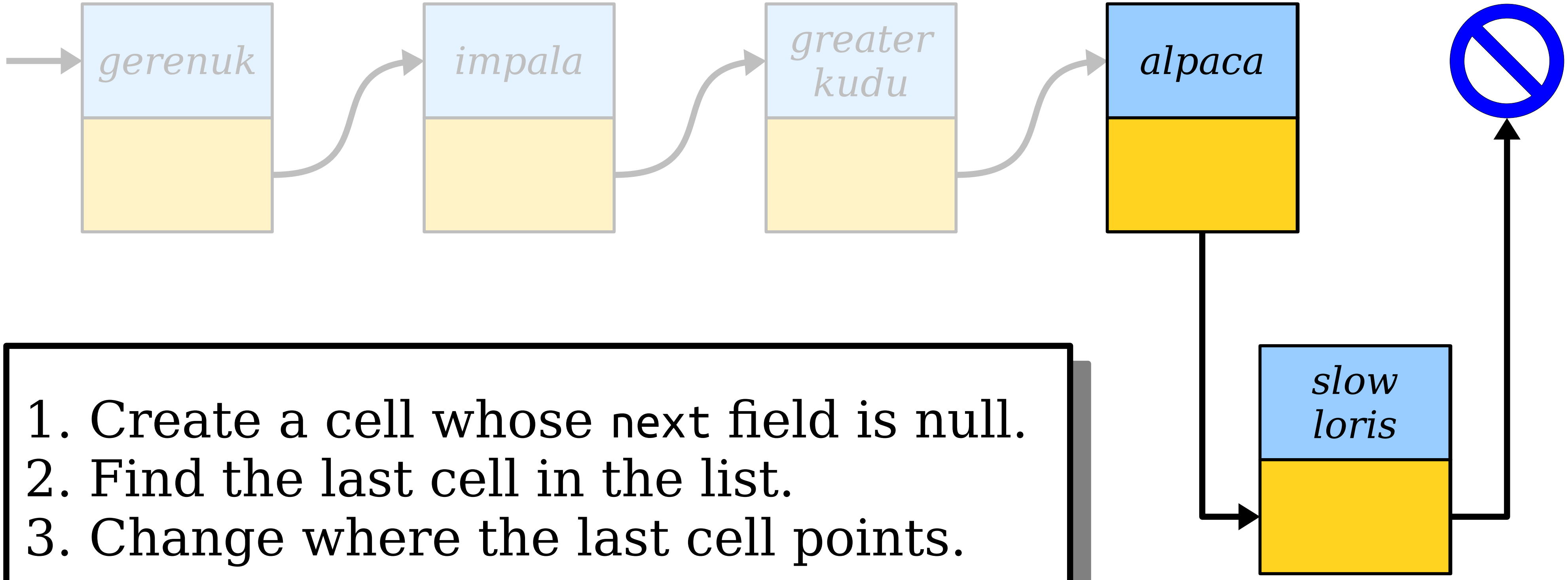
- Think about which link needs to get changed to append something to this list:



1. Create a cell whose next field is null.
2. Find the last cell in the list.
3. Change where the last cell points.

Appending to a List

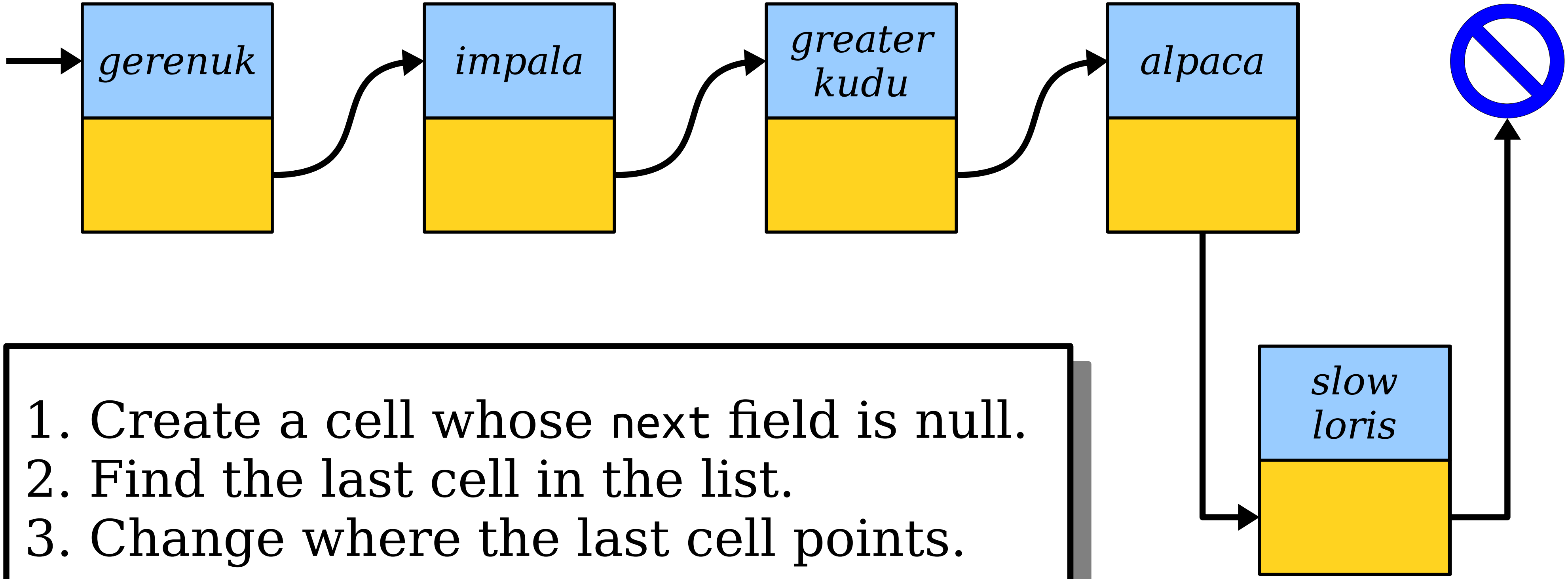
- Think about which link needs to get changed to append something to this list:



1. Create a cell whose next field is null.
2. Find the last cell in the list.
3. Change where the last cell points.

Appending to a List

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



1. Create a cell whose next field is null.
2. Find the last cell in the list.
3. Change where the last cell points.

Why did this code crash?

Formulate a hypothesis, but ***don’t post it in chat yet***.

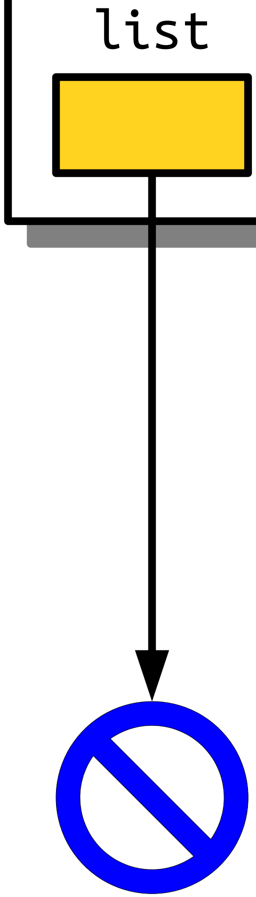
Why did this code crash?

Now, ***post your best guess in chat***. Not sure? Just answer “??”

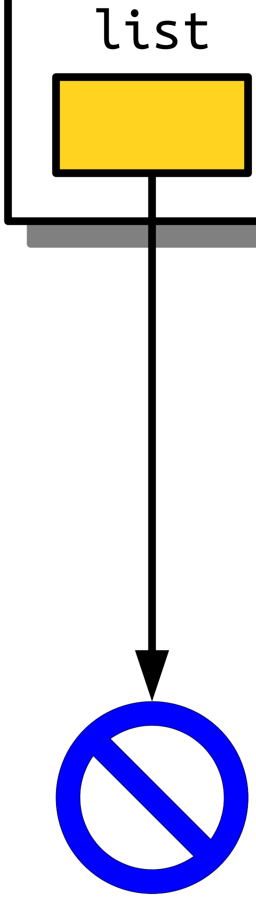
```
int main() {
   Cell* list = nullptr;
   appendTo(list, "Last");
   appendTo(list, "Final");
   appendTo(list, "Ultimate");
   appendTo(list, "Terminal");

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

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

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

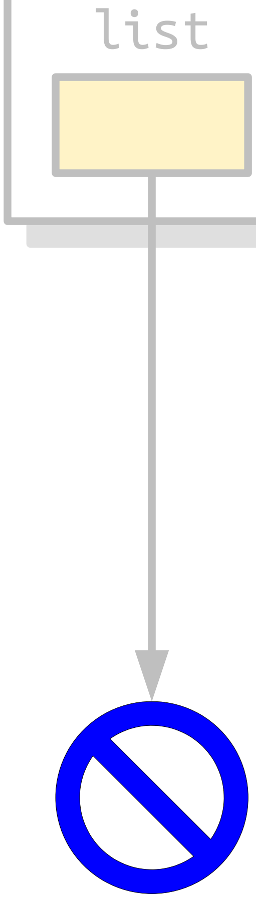
```
int main() {
   Cell* list = nullptr;
   appendTo(list, "Last");
   appendTo(list, "Final");
   appendTo(list, "Ultimate");
   appendTo(list, "Terminal");

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



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

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

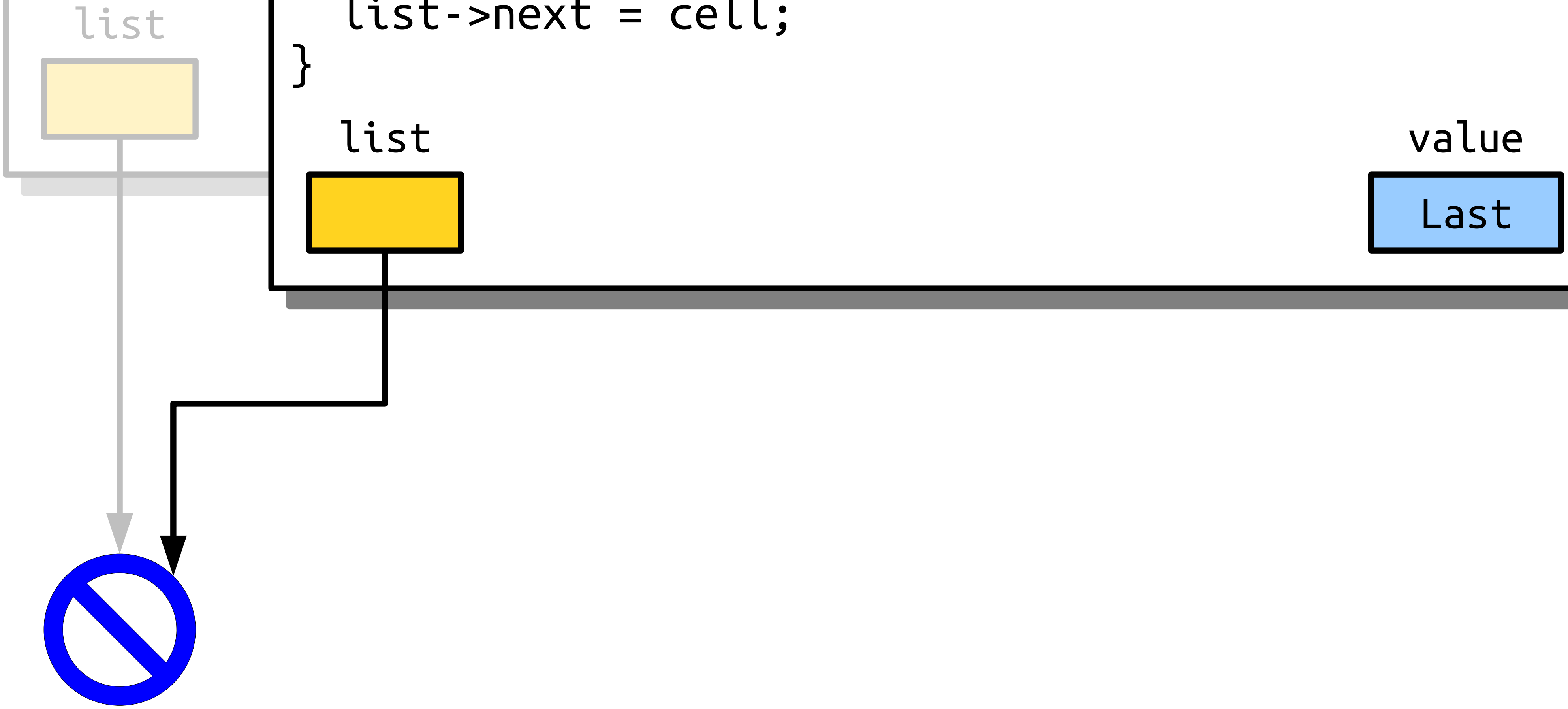


```
int main() {
    Cell* list = nullptr;
    appendTo(list, "Last");
    appendTo(list, "Final");
    appendTo(list, "Ultimate");
    appendTo(list, "Terminal");

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



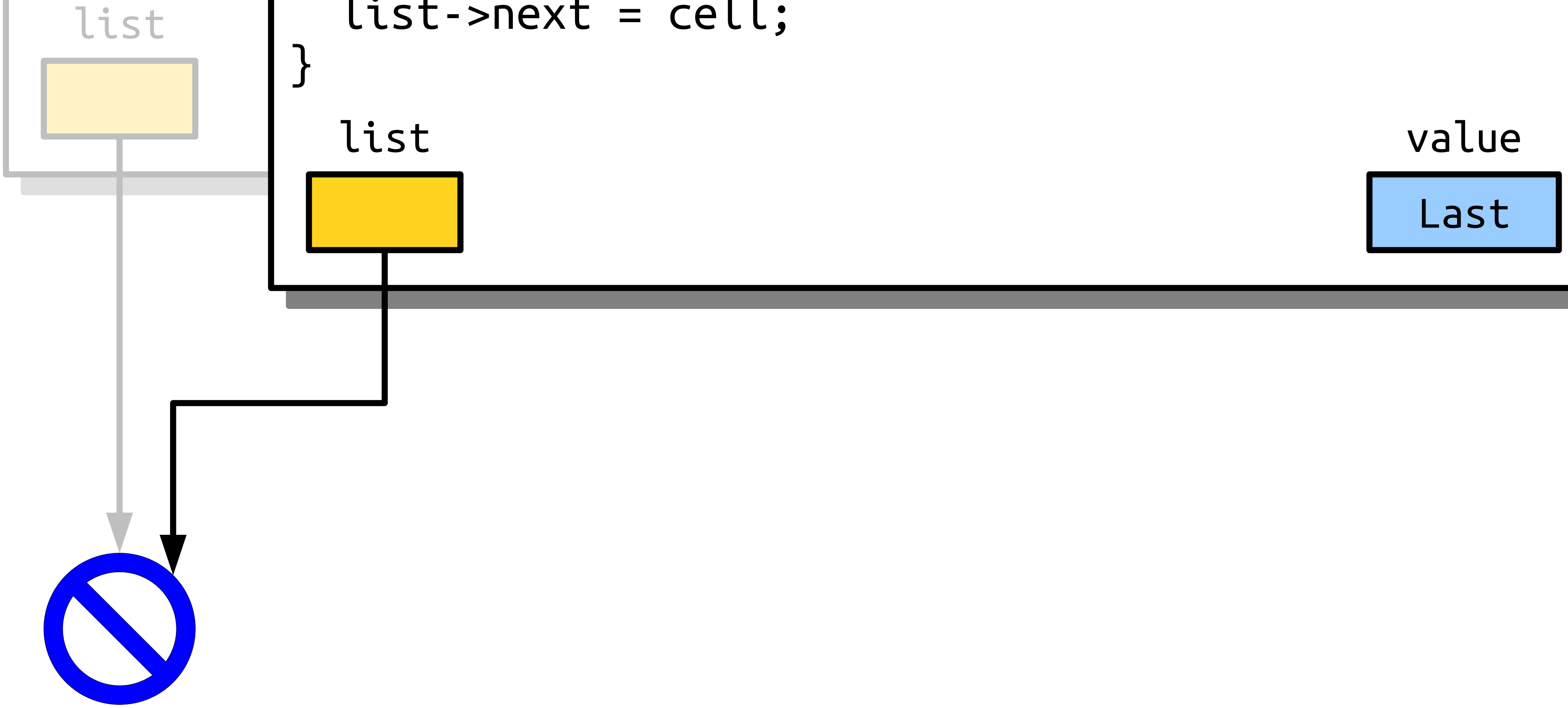
```
int main() {
   Cell* l
   appendT
   appendT
   appendT
   appendT

   /* … ot
}
```

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

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

  list->next = cell;
}
```



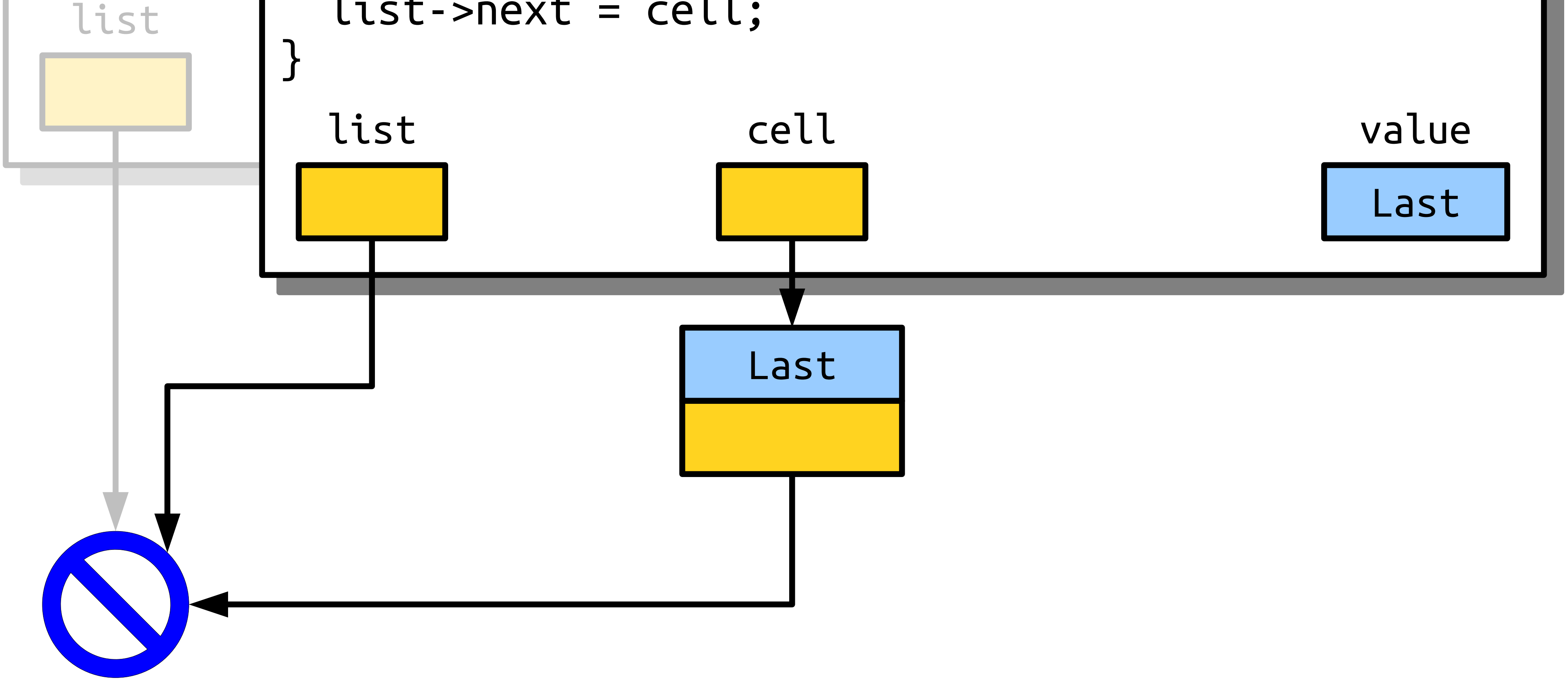
```
int main() {
  Cell* l
  appendT
  appendT
  appendT
  appendT

  /* … ot
}
```

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

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

  list->next = cell;
}
```



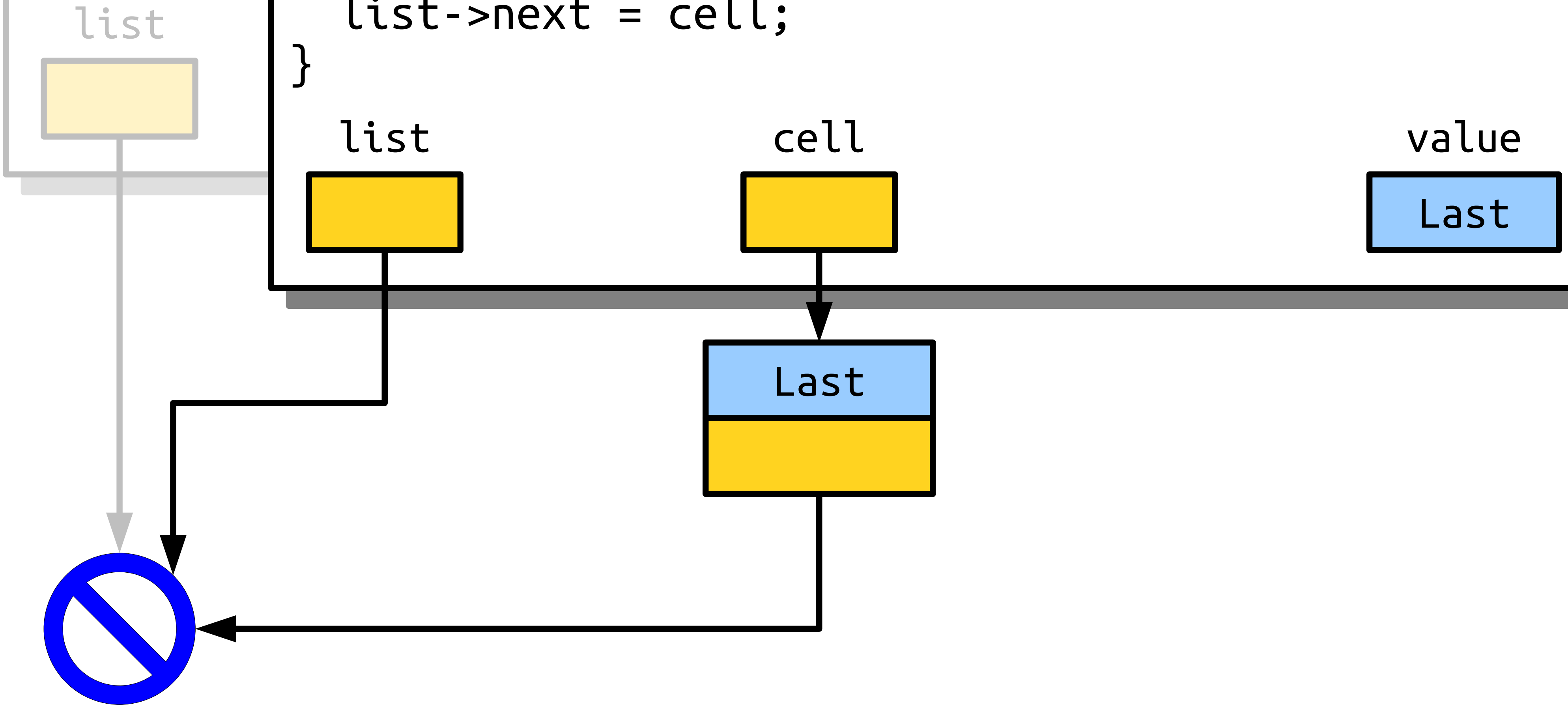
```
int main() {
  Cell* l
  appendT
  appendT
  appendT
  appendT

  /* … ot
}
```

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

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

  list->next = cell;
}
```



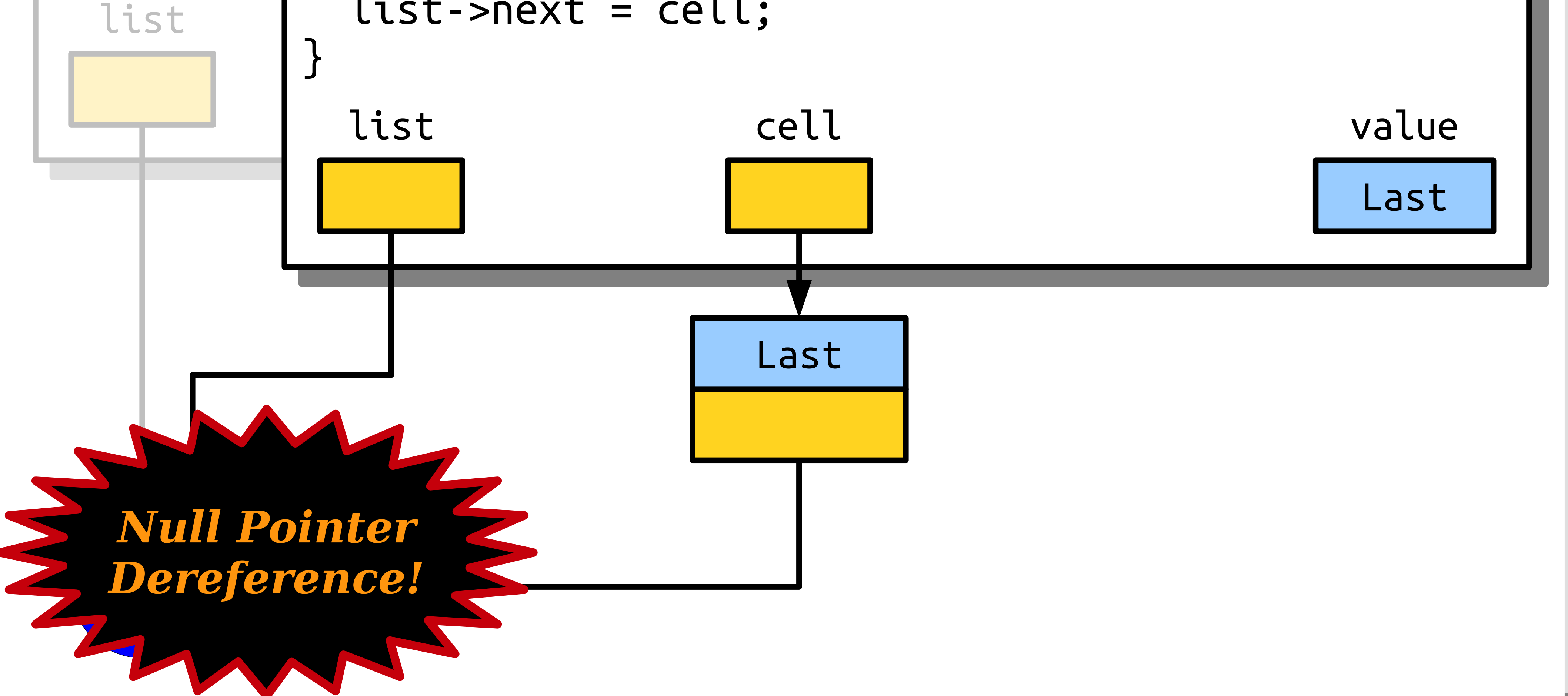
```
int main() {
   Cell* l
   appendT
   appendT
   appendT
   appendT

   /* … ot
}
```

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

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

  list->next = cell;
}
```



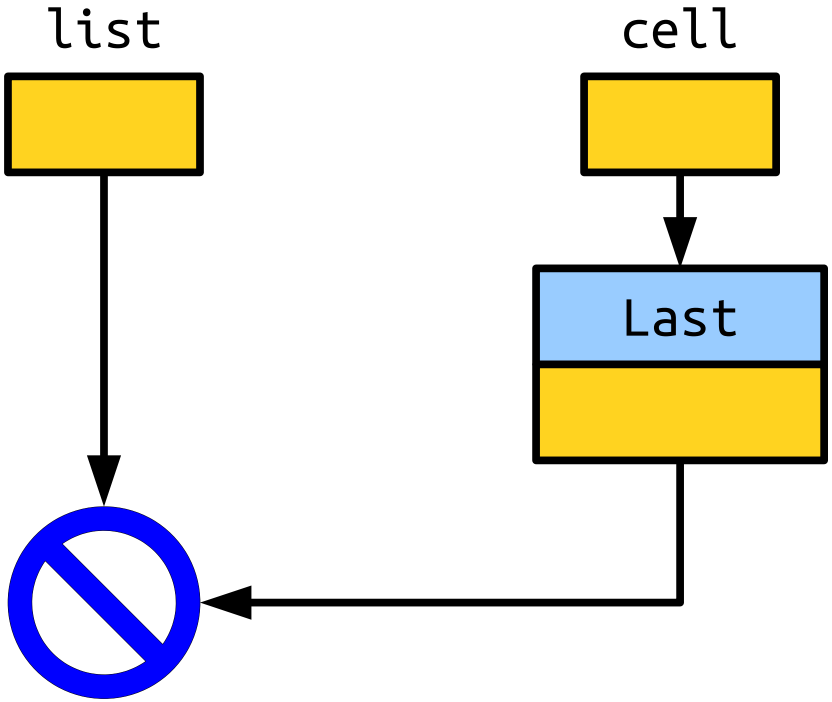
```
int main() {
   Cell* l
   appendT
   appendT
   appendT
   appendT

   /* … ot
}
```

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

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

  list->next = cell;
}
```



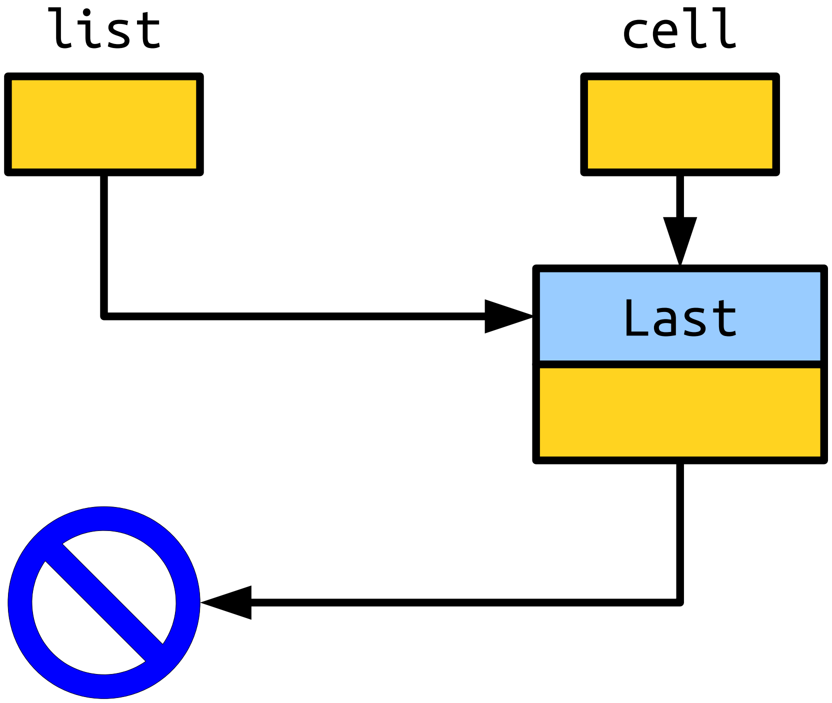
Appending to a List

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



Appending to a List

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



Why didn’t this code work?

Formulate a hypothesis, but ***don’t post it in chat yet***.

Why didn’t this code work?

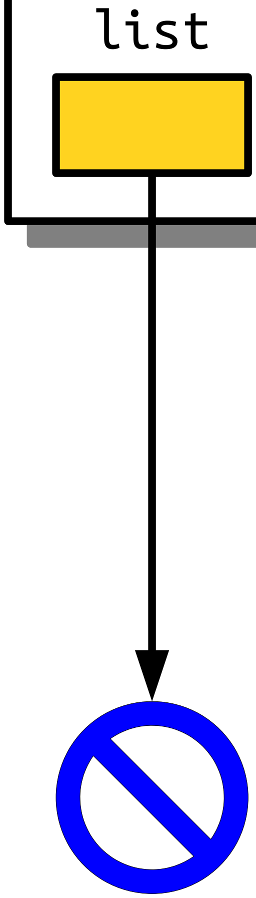
Now, ***post your best guess in chat***. Not sure? Just answer “??”

What Went Wrong (This Other Time)?

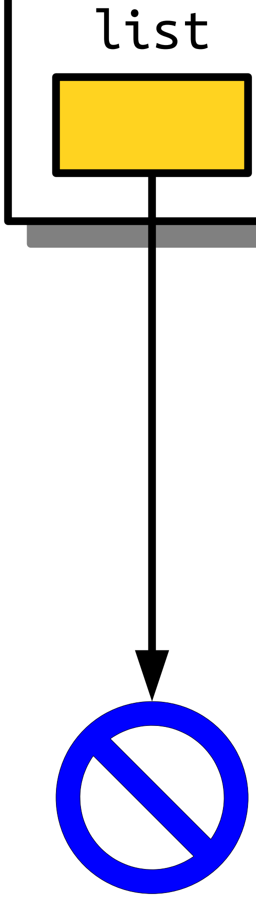
```
int main() {
   Cell* list = nullptr;
   appendTo(list, "Last");
   appendTo(list, "Final");
   appendTo(list, "Ultimate");
   appendTo(list, "Terminal");

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

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

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

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

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

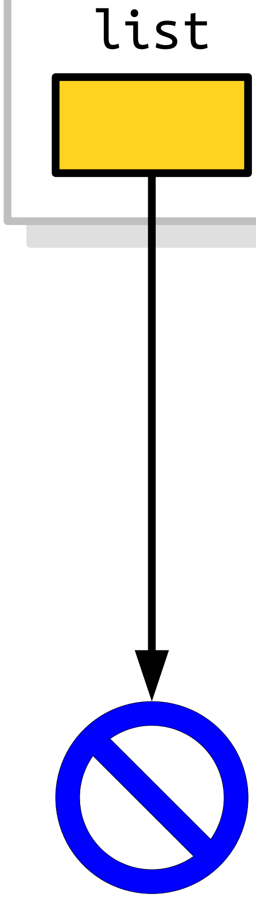


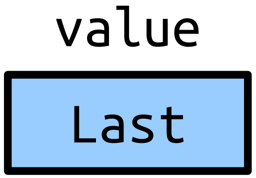
```
int main() {
    Cell* list = nullptr;
    appendTo(list, "Last");
    appendTo(list, "Final");
    appendTo(list, "Ultimate");
    appendTo(list, "Terminal");

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



```
int main() {
  Cell* l
  appendT
  appendT
  appendT
  appendT

  /* … ot
}
```

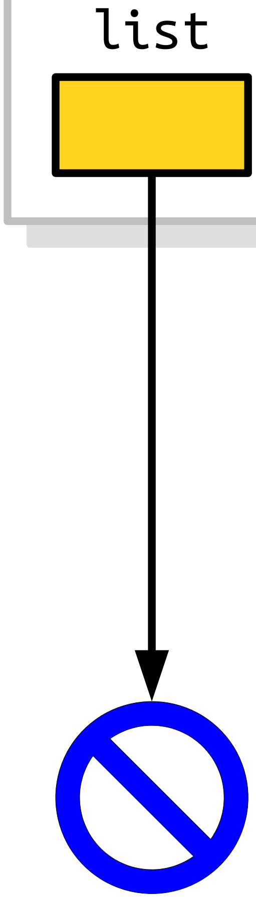
```
void appendTo(Cell*& list, const string& value) {
  Cell* cell  = new Cell;
  cell->value = value;
  cell->next  = nullptr;

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

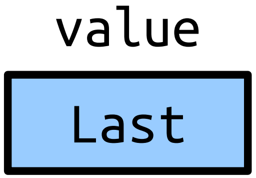


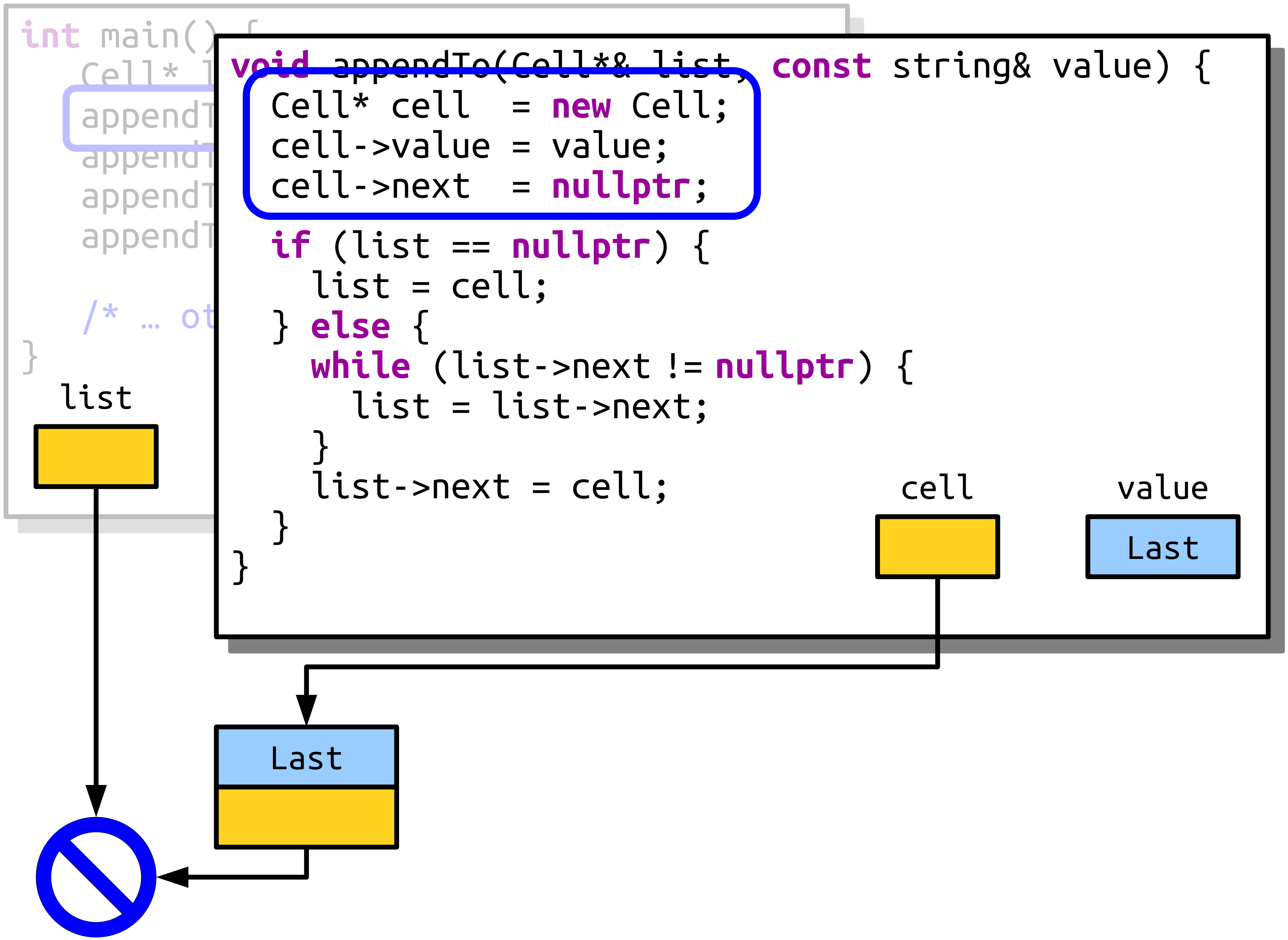


```
int main() {
   Cell* l
   appendT
   appendT
   appendT
   appendT

   /* … ot
}
```

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





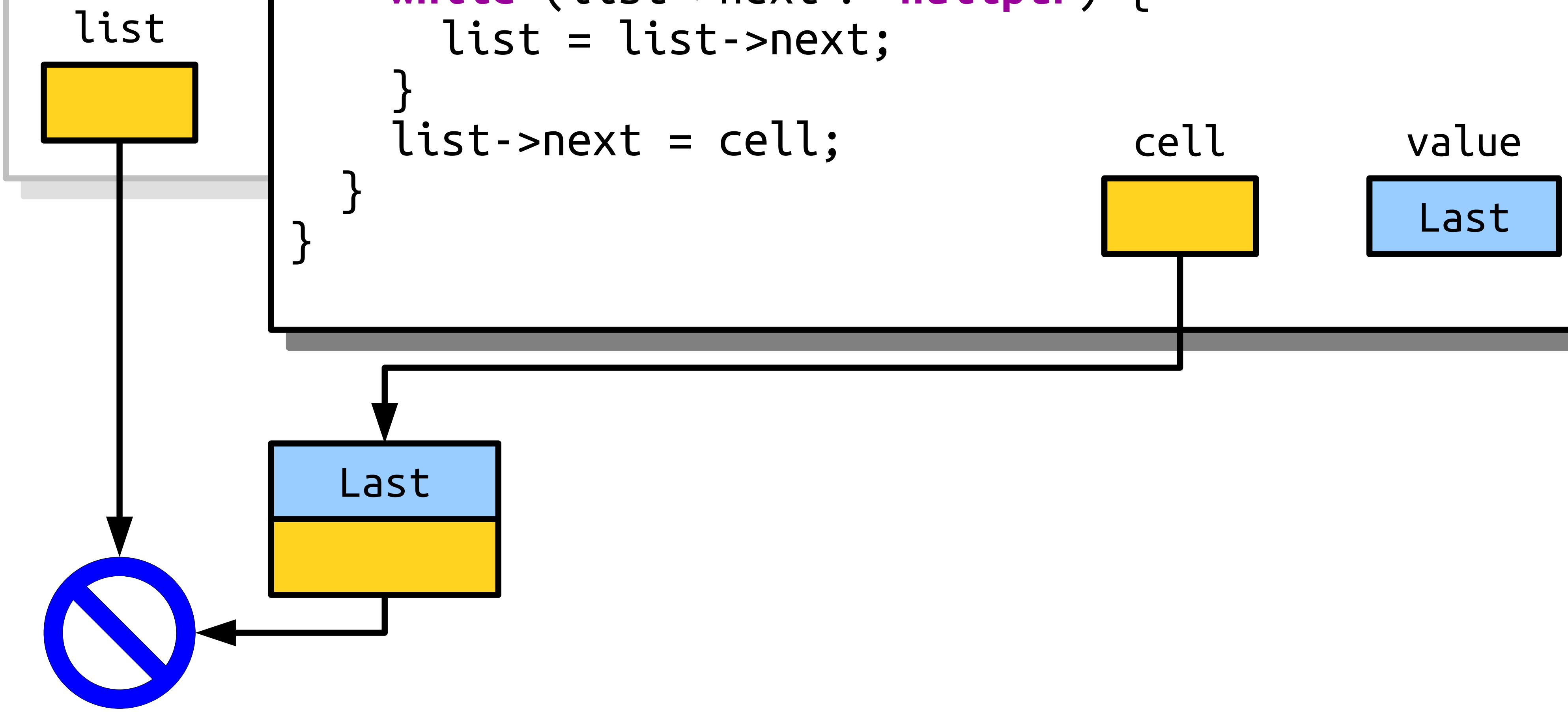


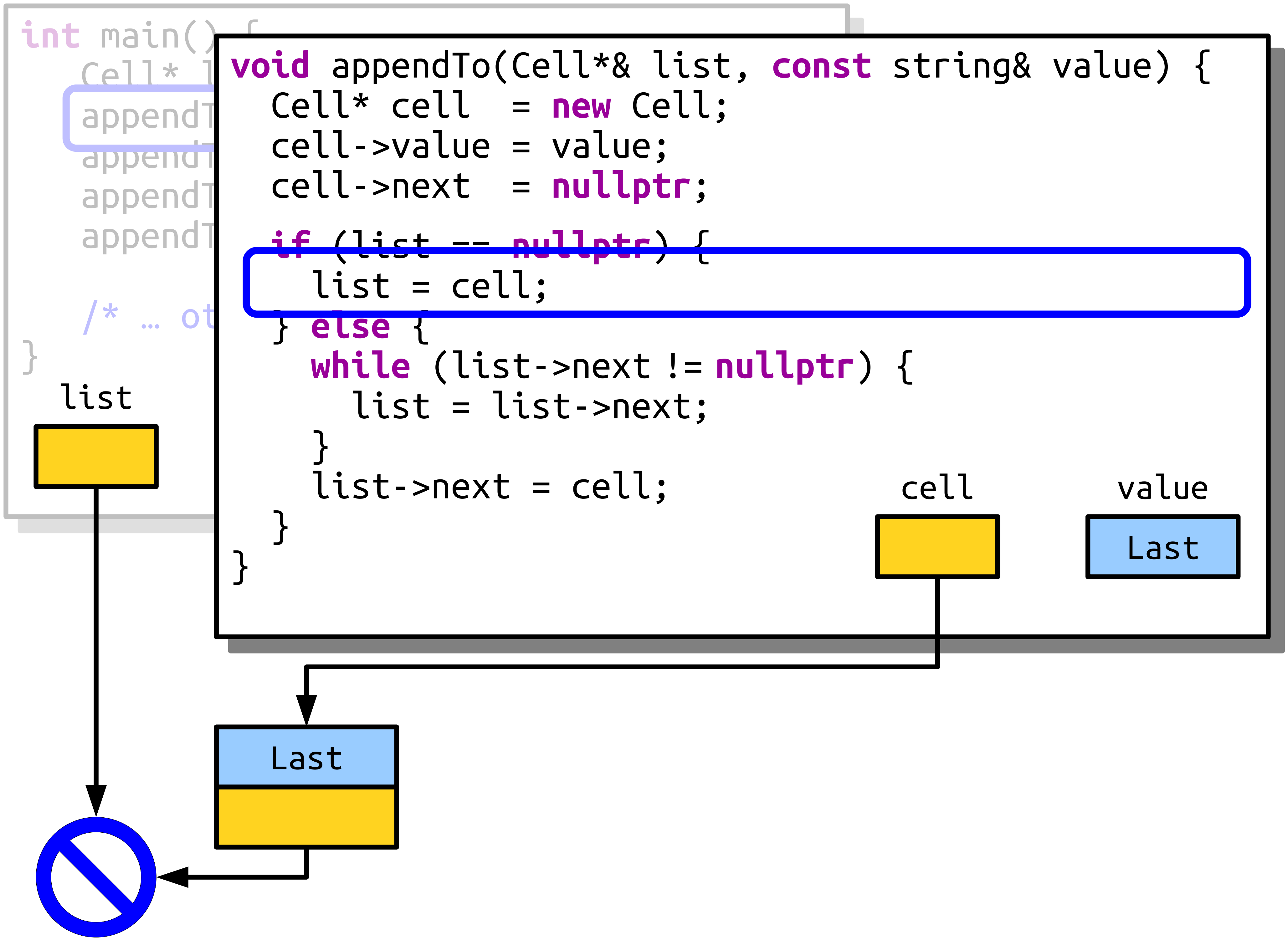
```
int main() {
    Cell* l
    appendT
    appendT
    appendT
    appendT

    /* … ot
}
```

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

```
int main() {
   Cell* l
   appendT
   appendT
   appendT
   appendT

   /* … ot
}
```

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

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



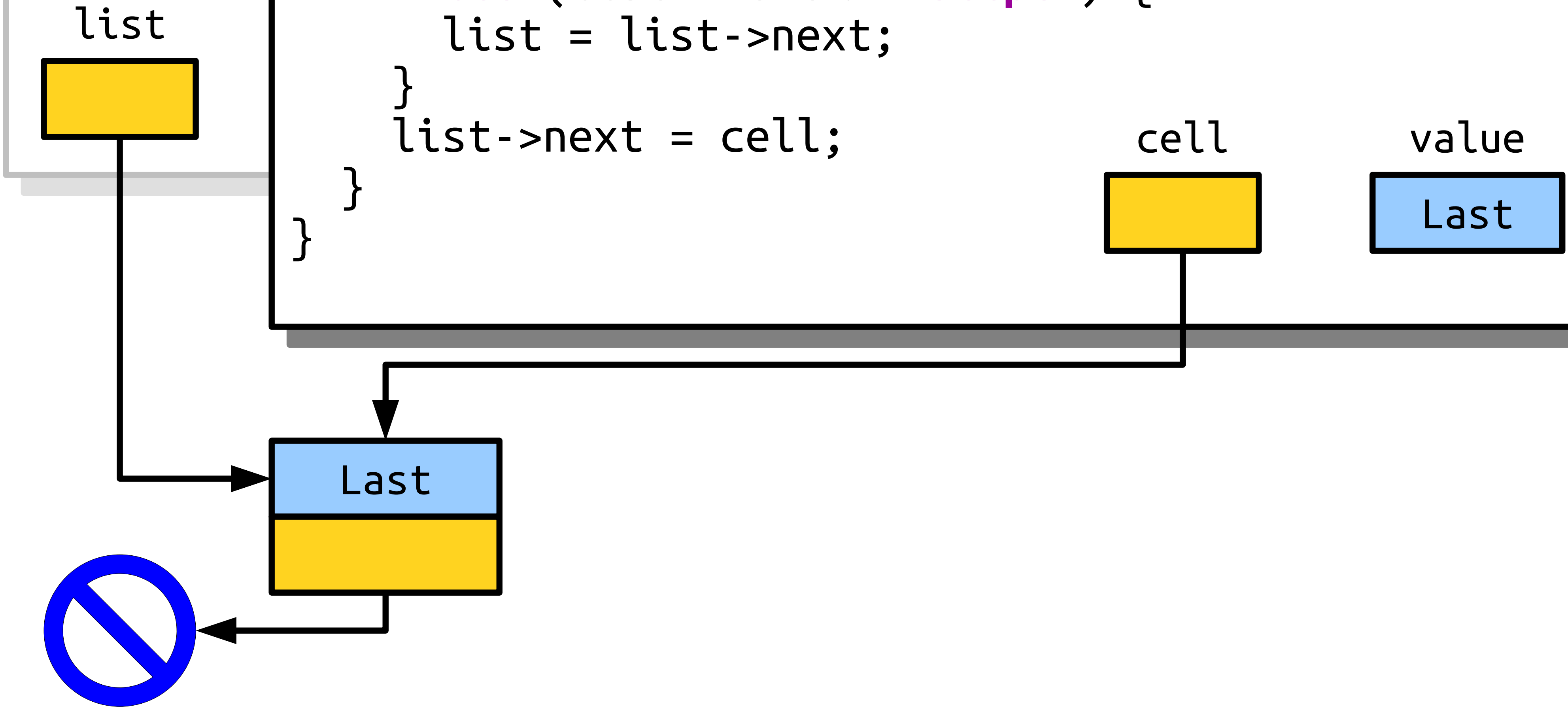


```
int main() {
   Cell* l
   appendT
   appendT
   appendT
   appendT

   /* … ot
}
```

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

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

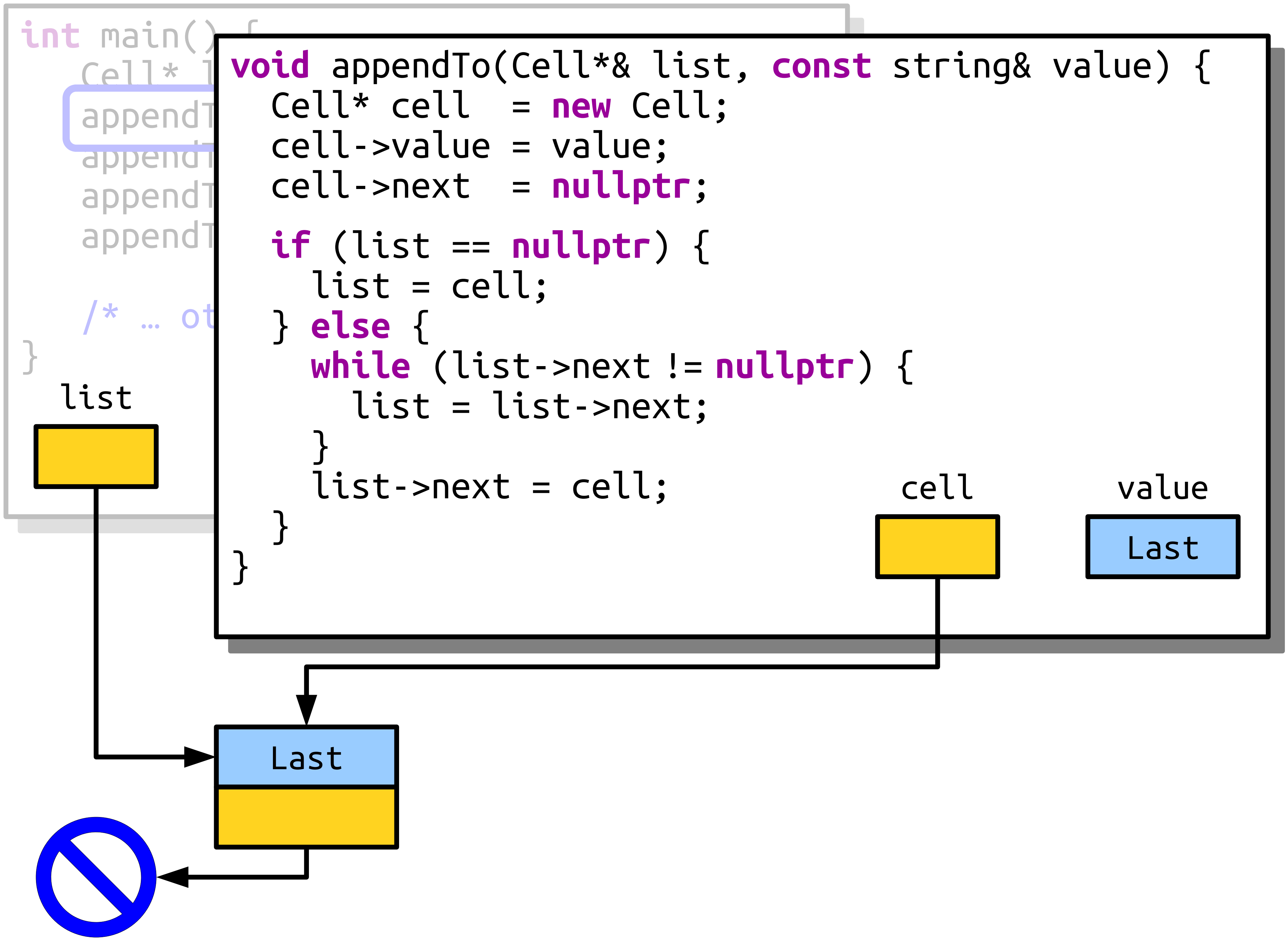
```
int main() {
   Cell* l
   appendT
   appendT
   appendT
   appendT

   /* … ot
}
```

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

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

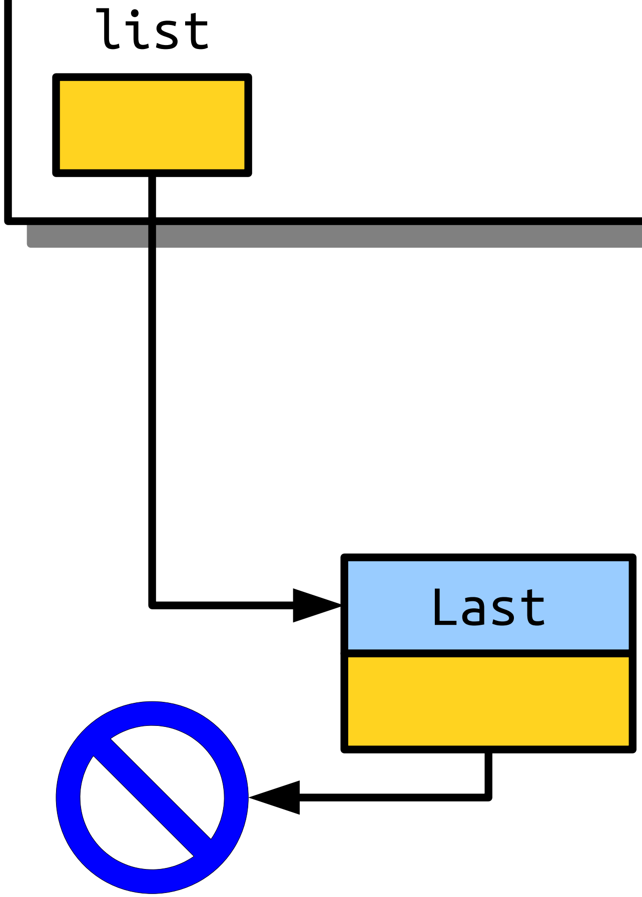


```
int main() {
   Cell* l
   appendT
   appendT
   appendT
   appendT

   /* … ot
}
```

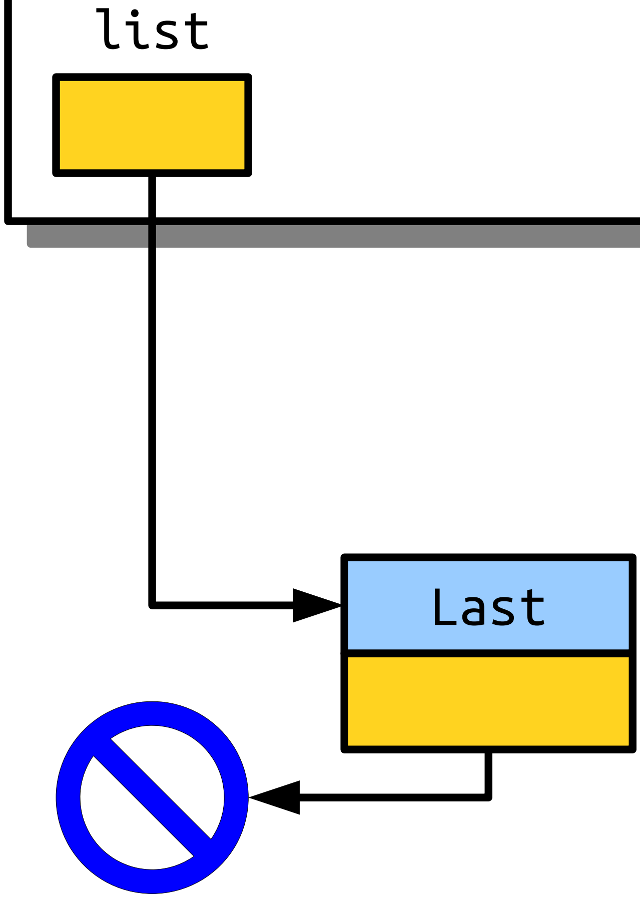
```
void appendTo(Cell*& list, const string& value) {
  Cell* cell  = new Cell;
  cell->value = value;
  cell->next  = nullptr;

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



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

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

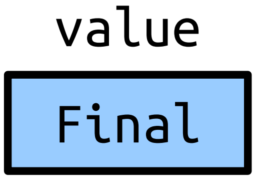


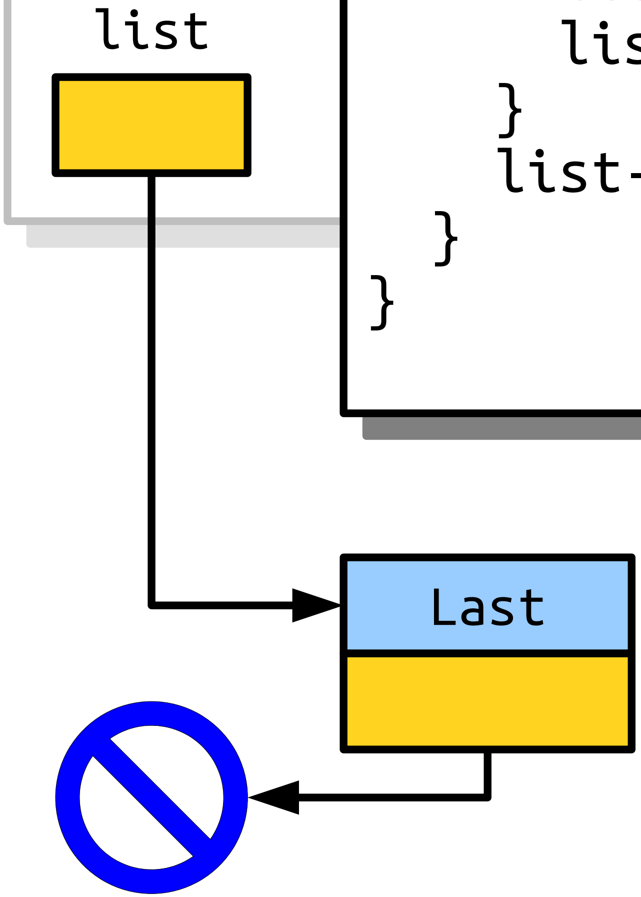
```
int main() {
    Cell* list = nullptr;
    appendTo(list, "Last");
    appendTo(list, "Final");
    appendTo(list, "Ultimate");
    appendTo(list, "Terminal");

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



```
int main() {
   Cell* l
   appendT
   appendT
   appendT
   appendT

   /* … ot
}
```

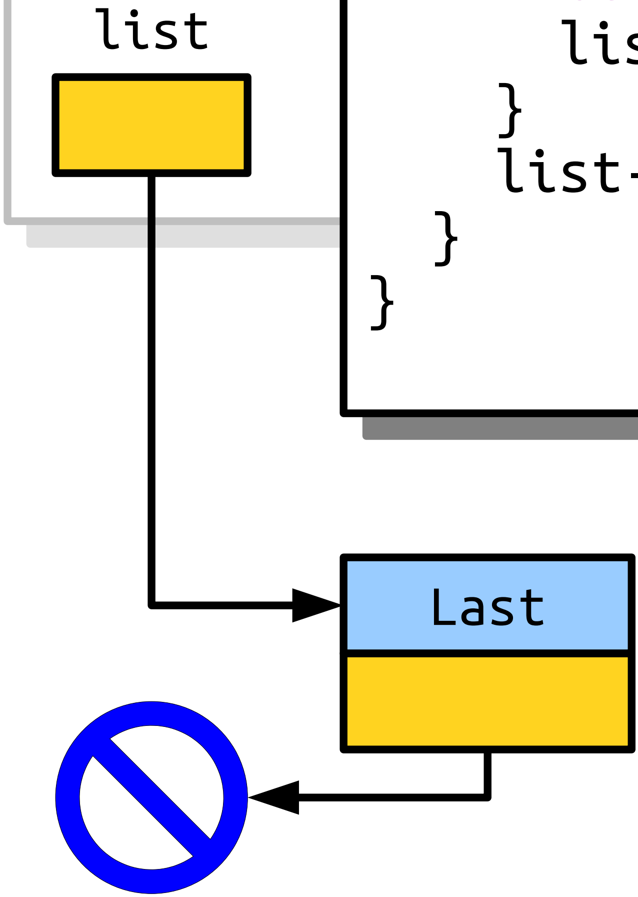
```
void appendTo(Cell*& list, const string& value) {
  Cell* cell  = new Cell;
  cell->value = value;
  cell->next  = nullptr;

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

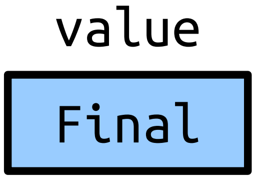




```
int main() {
   Cell* l
   appendT
   appendT
   appendT
   appendT

   /* … ot
}
```

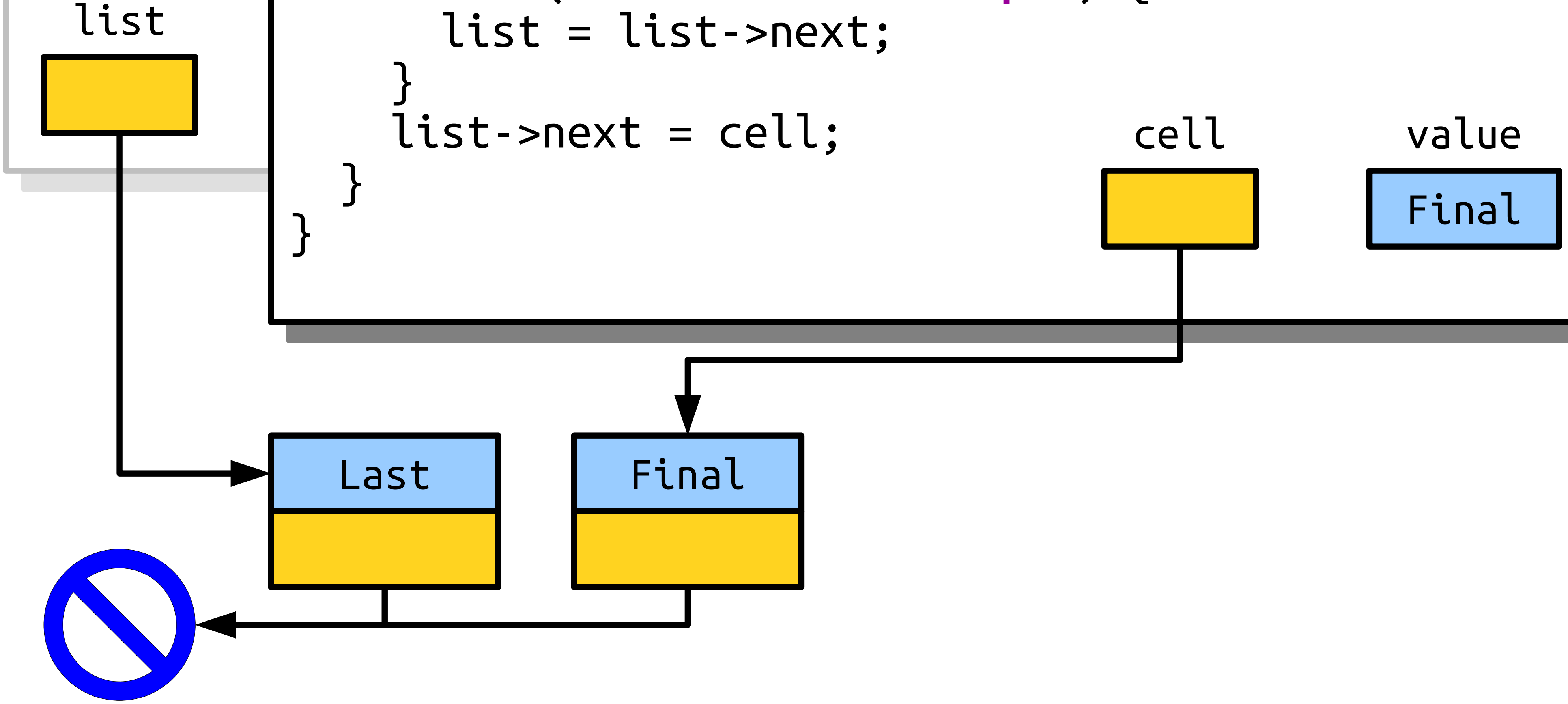
```
void appendTo(Cell*& list, const string& value) {
  Cell* cell  = new Cell;
  cell->value = value;
  cell->next  = nullptr;
  if (list == nullptr) {
    list = cell;
  } else {
    while (list->next != nullptr) {
      list = list->next;
    }
    list->next = cell;
  }
}
```





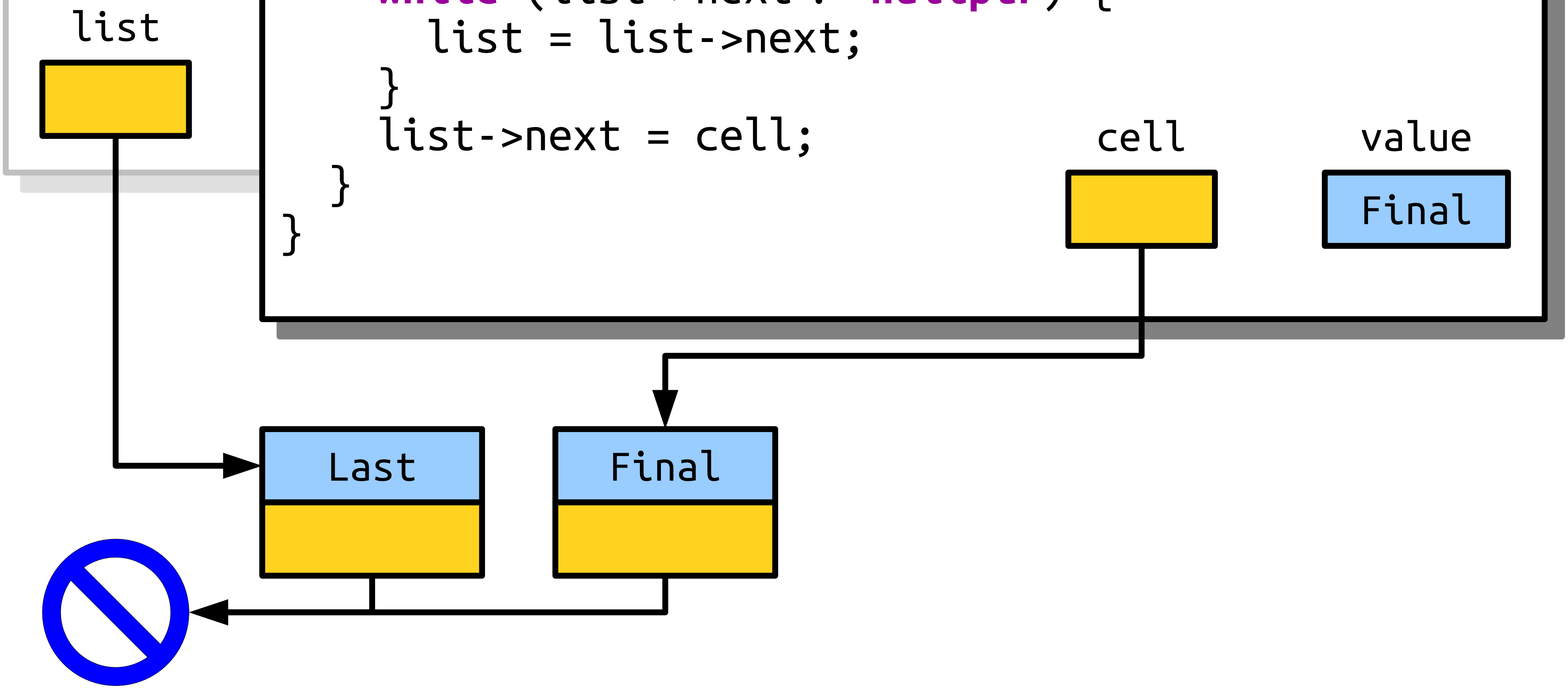
```
int main() {
   Cell* l
   appendT
   appendT
   appendT
   appendT

   /* … ot
}
```

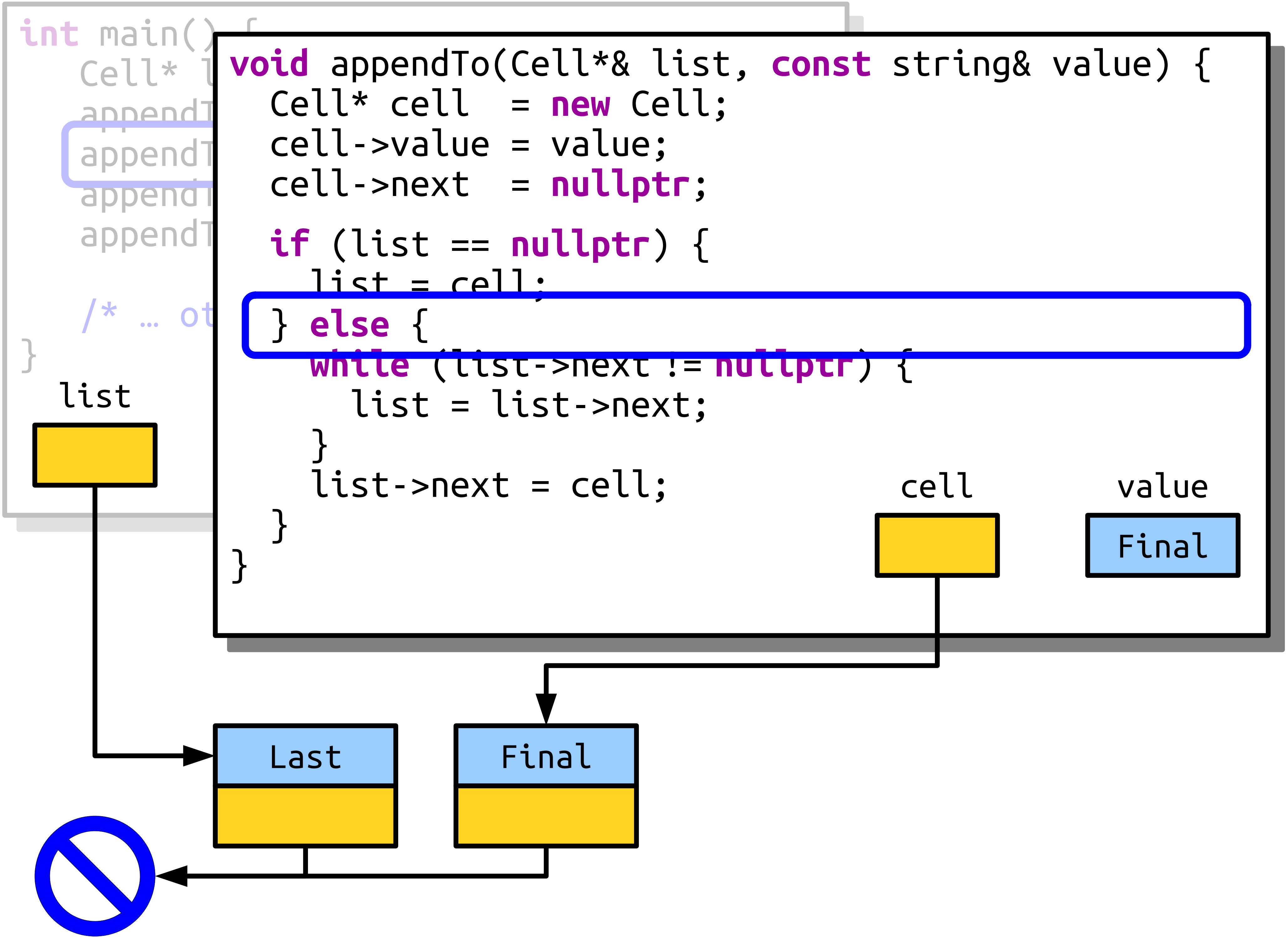
```
void appendTo(Cell*& list, const string& value) {
  Cell* cell  = new Cell;
  cell->value = value;
  cell->next  = nullptr;
  if (list == nullptr) {
    list = cell;
  } else {
    while (list->next != nullptr) {
      list = list->next;
    }
    list->next = cell;
  }
}
```



```
int main() {
   Cell* l
   appendT
   appendT
   appendT
   appendT

   /* … ot
}
```

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

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



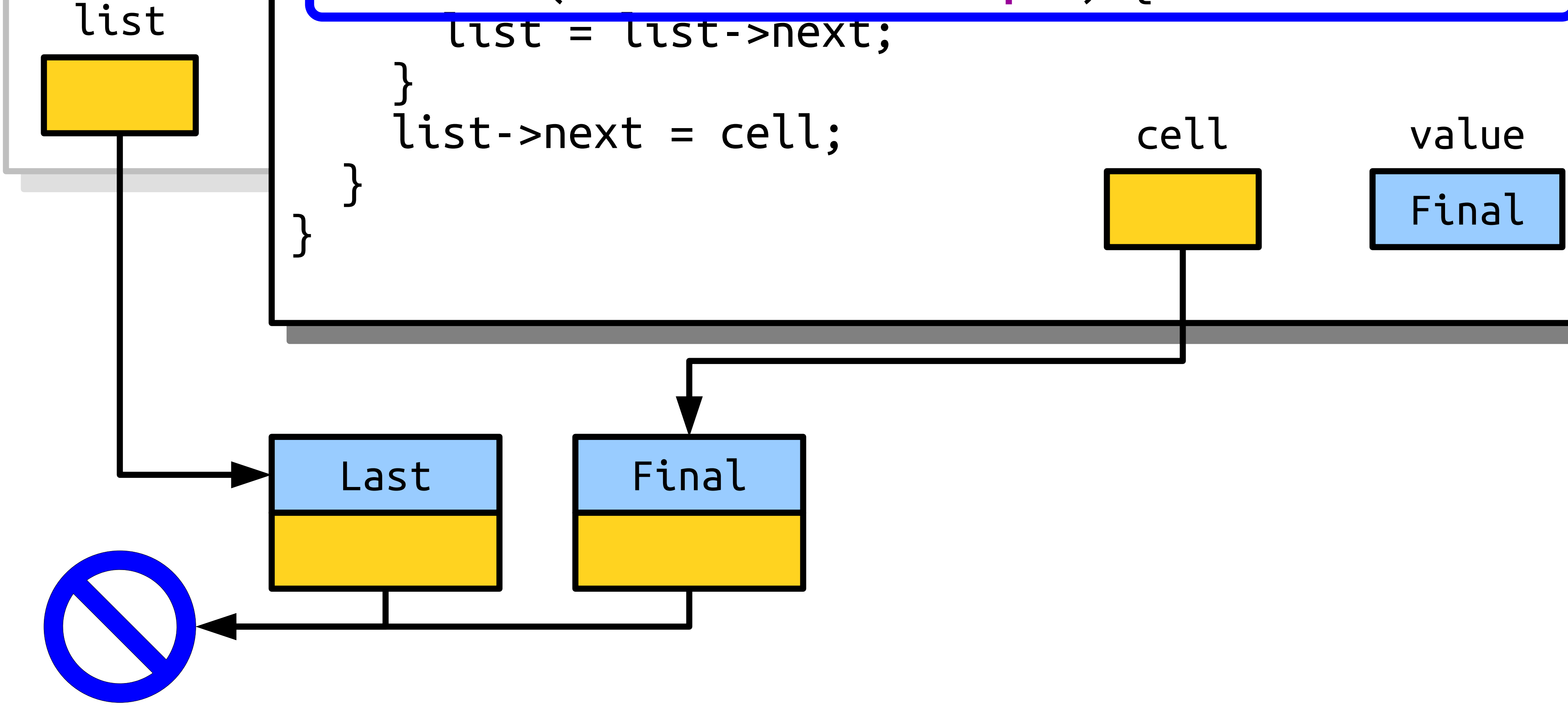


```
int main() {
   Cell* l
   appendT
   appendT
   appendT
   appendT

   /* … ot
}
```

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

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

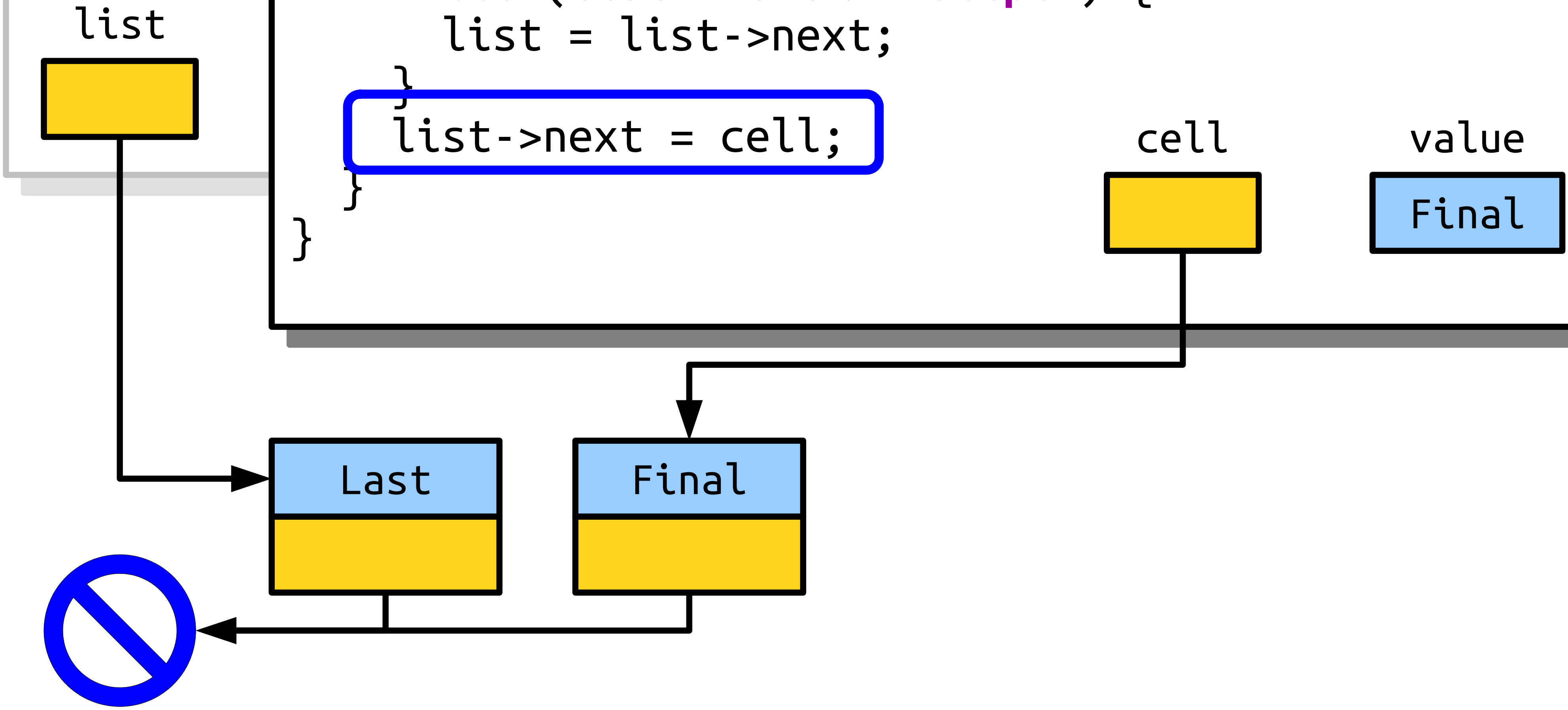
```
int main() {
   Cell* l
   appendT
   appendT
   appendT
   appendT

   /* … ot
}
```

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

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



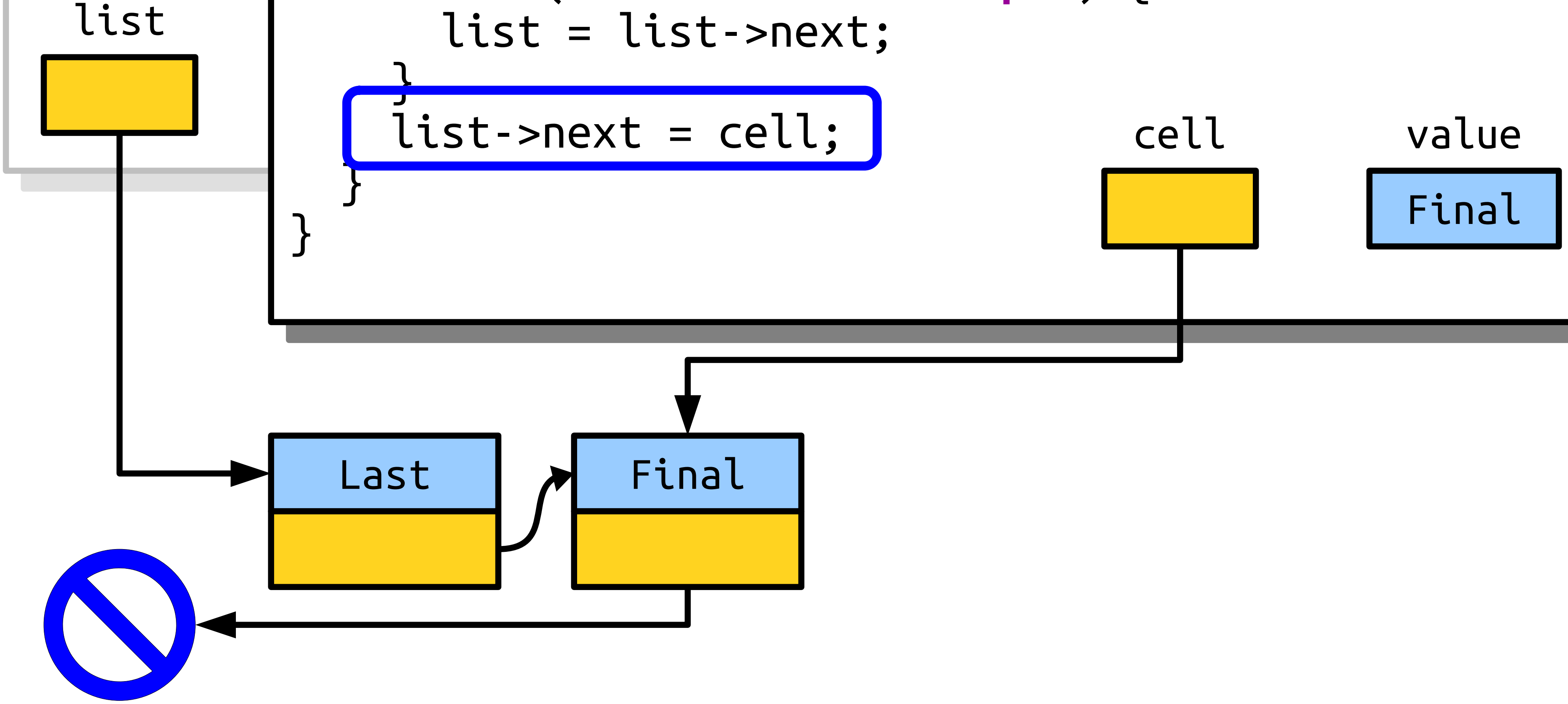
```
int main() {
   Cell* l
   appendT
   appendT
   appendT
   appendT

   /* … ot
}
```

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

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



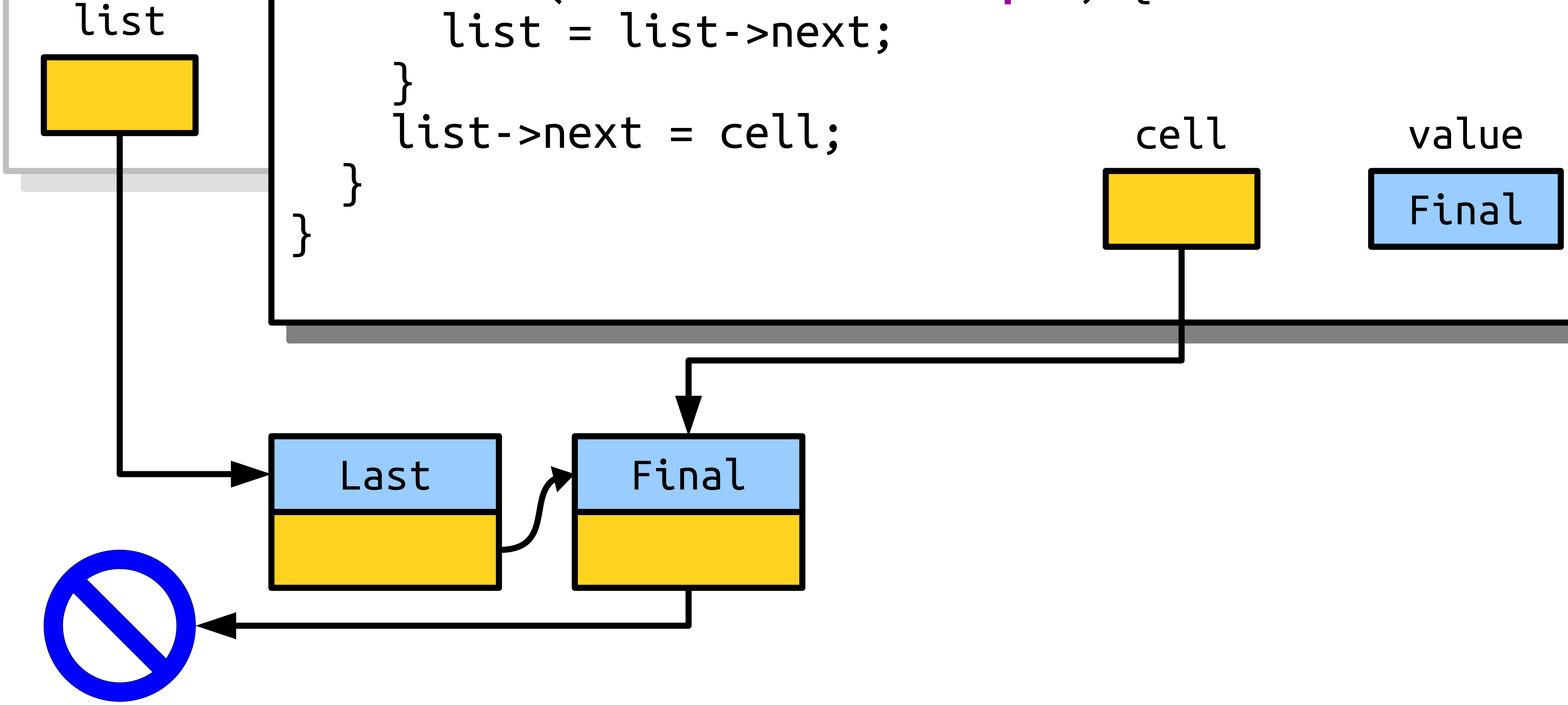
```
int main() {
  Cell* l
  appendT
  appendT
  appendT
  appendT

  /* … ot
}
```

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

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

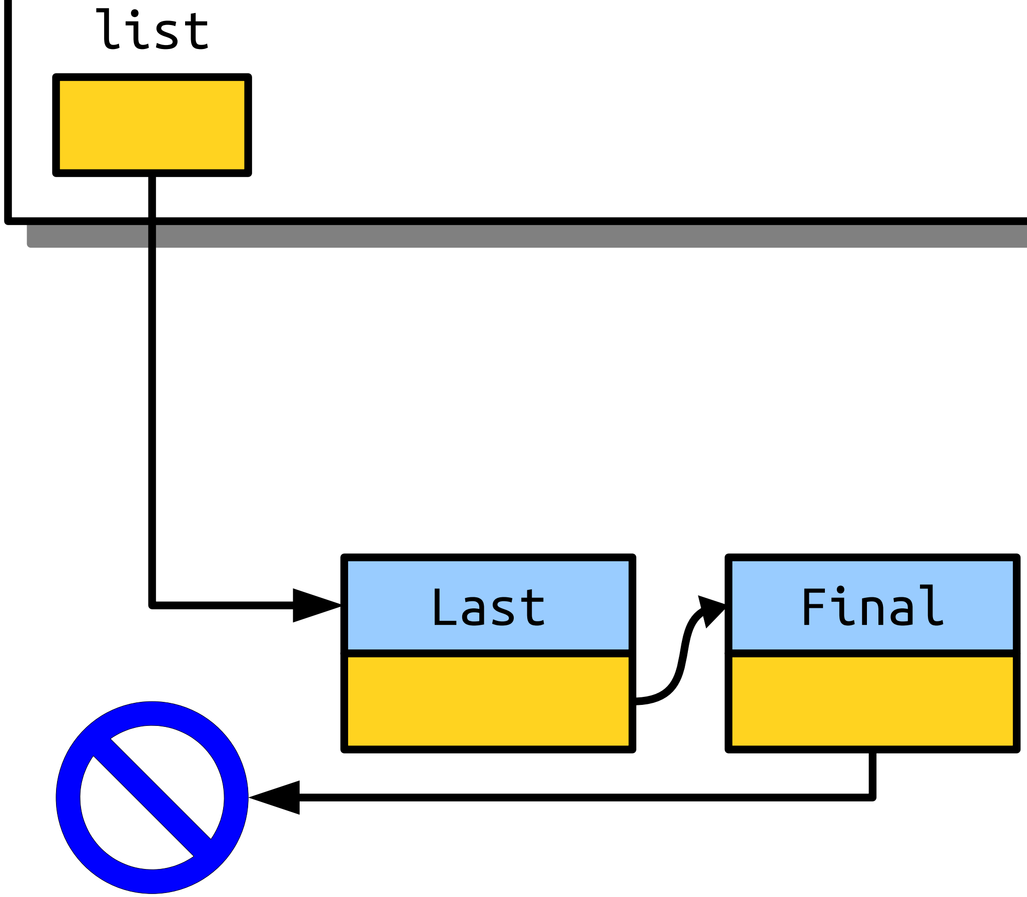


```
int main() {
   Cell* l
   appendT
   appendT
   appendT
   appendT

   /* … ot
}
```

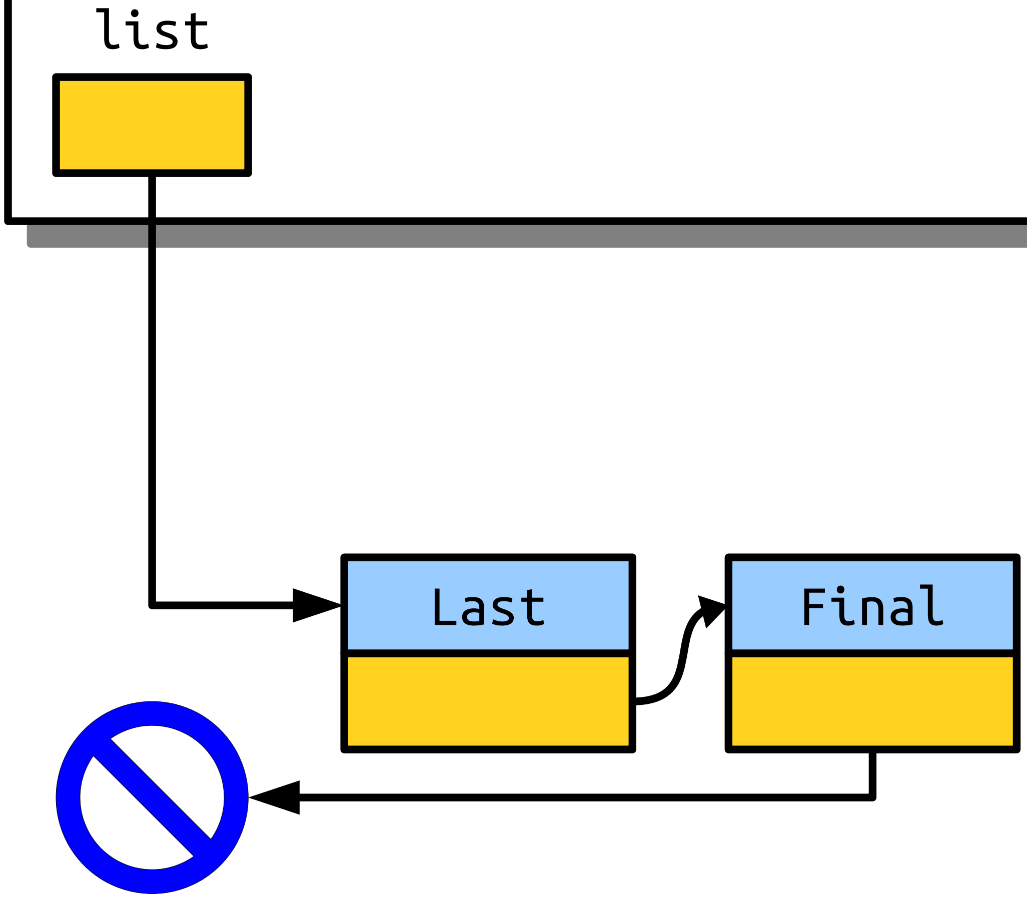
```
void appendTo(Cell*& list, const string& value) {
  Cell* cell  = new Cell;
  cell->value = value;
  cell->next  = nullptr;

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



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

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

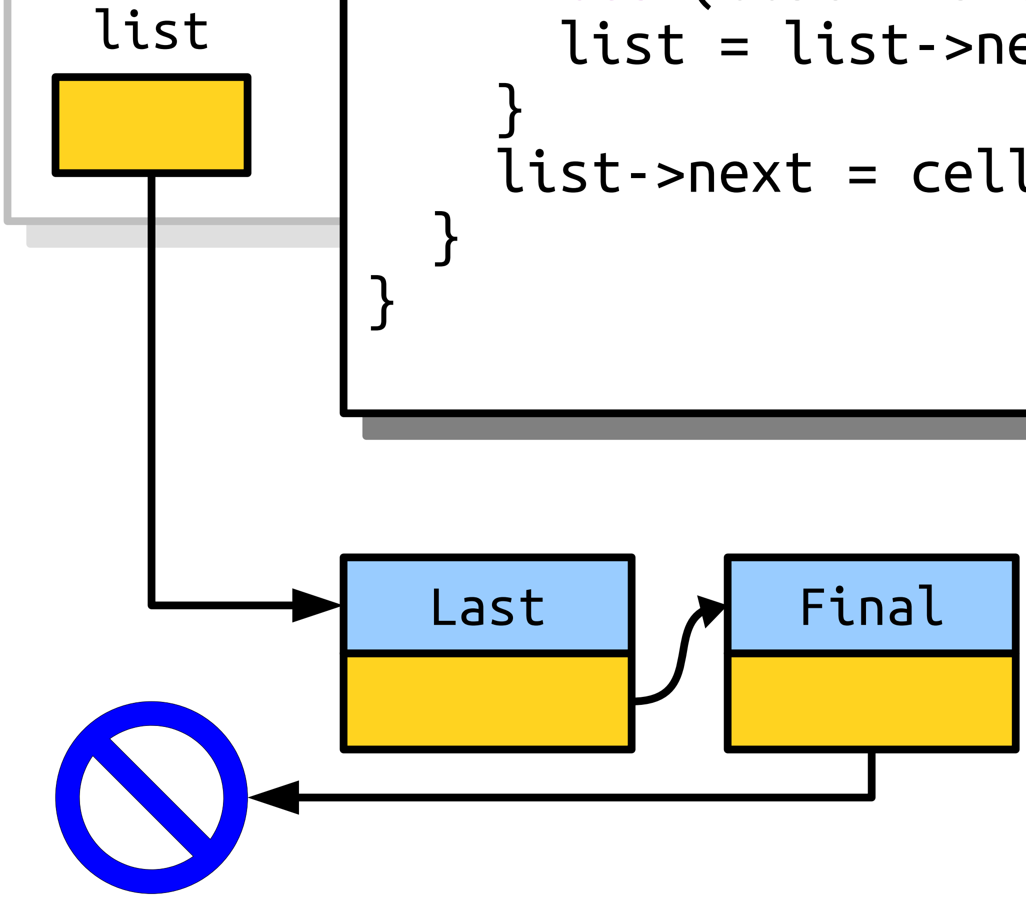


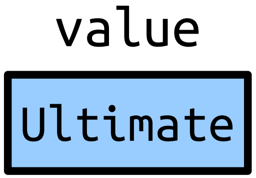
```
int main() {
   Cell* list = nullptr;
   appendTo(list, "Last");
   appendTo(list, "Final");
   appendTo(list, "Ultimate");
   appendTo(list, "Terminal");

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



```
int main() {
   Cell* l
   appendT
   appendT
   appendT
   appendT

   /* … ot
}
```

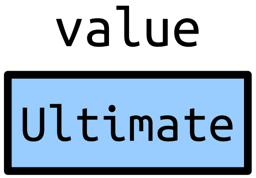
```
void appendTo(Cell*& list, const string& value) {
  Cell* cell  = new Cell;
  cell->value = value;
  cell->next  = nullptr;

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

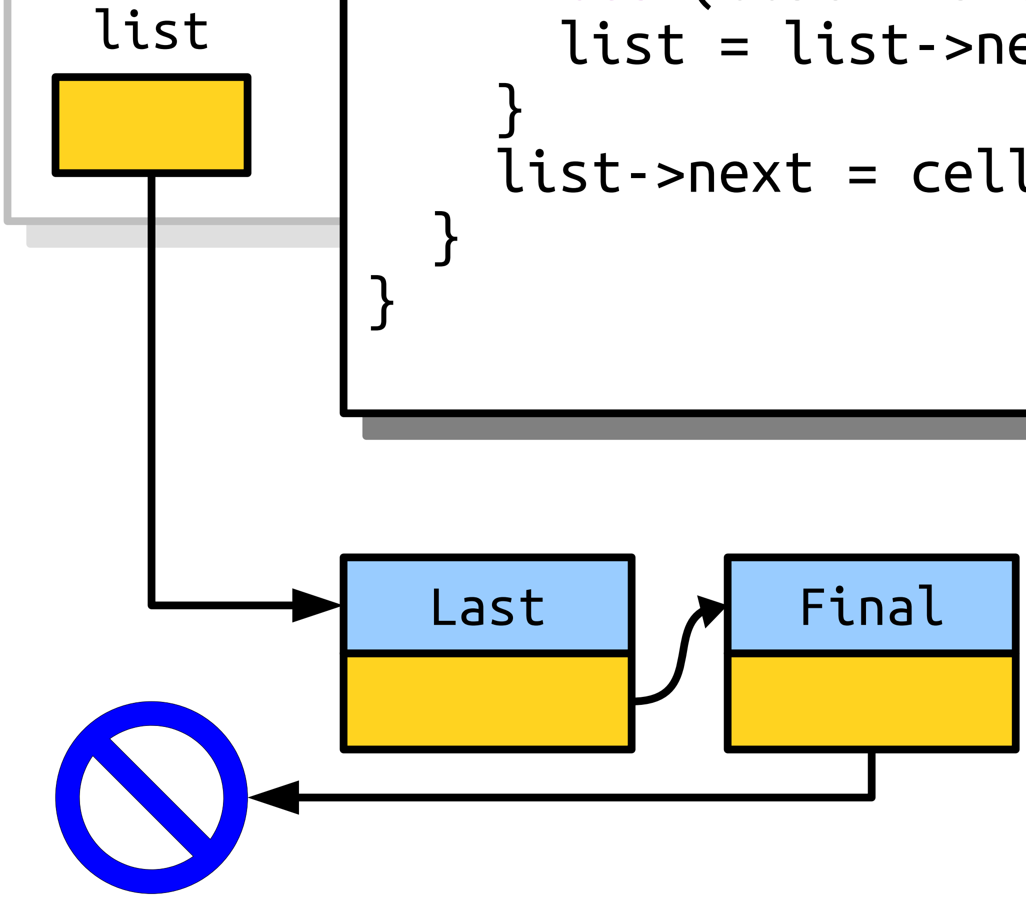


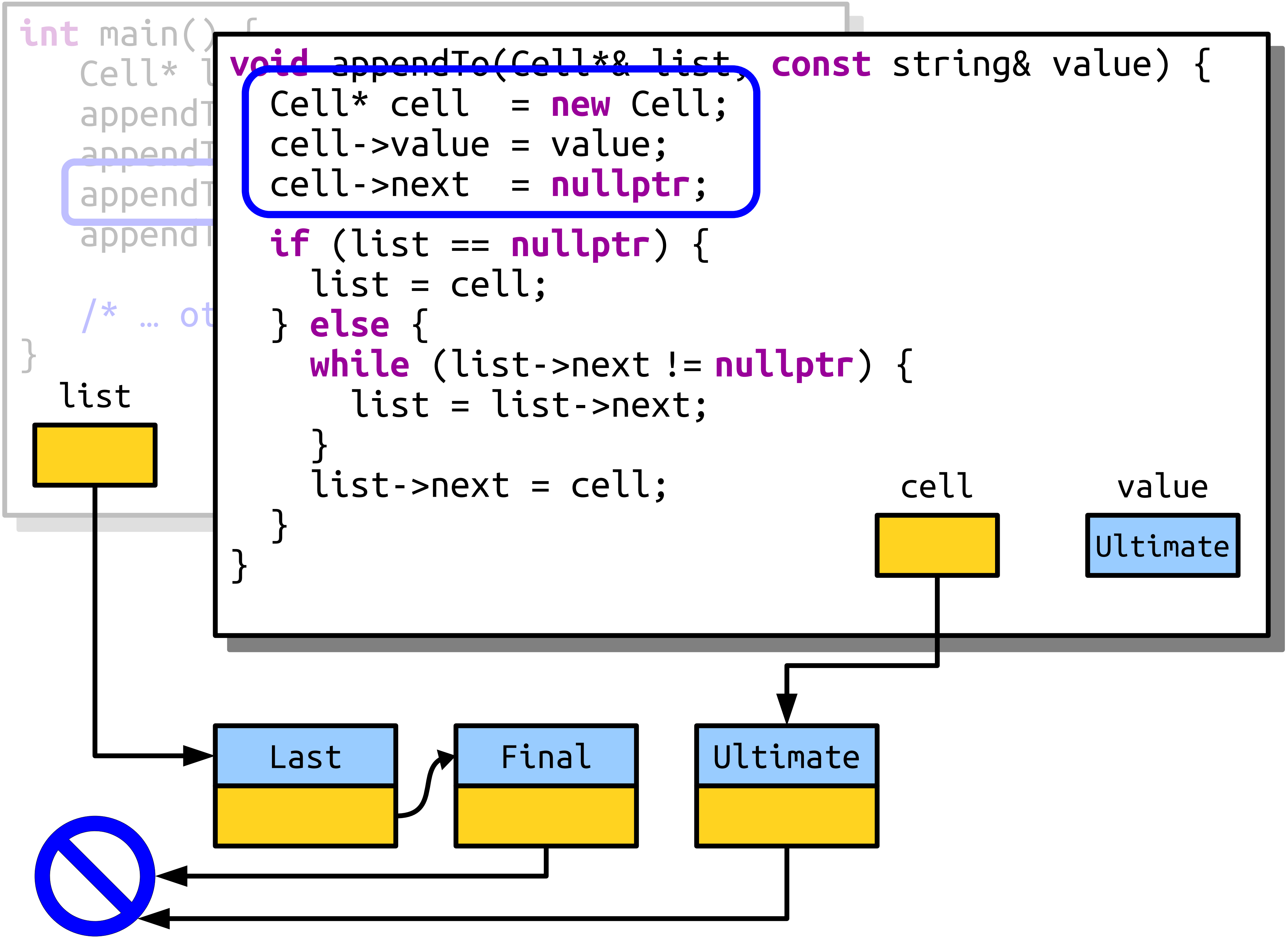


```
int main() {
   Cell* l
   appendT
   appendT
   appendT
   appendT

   /* … ot
}
```

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





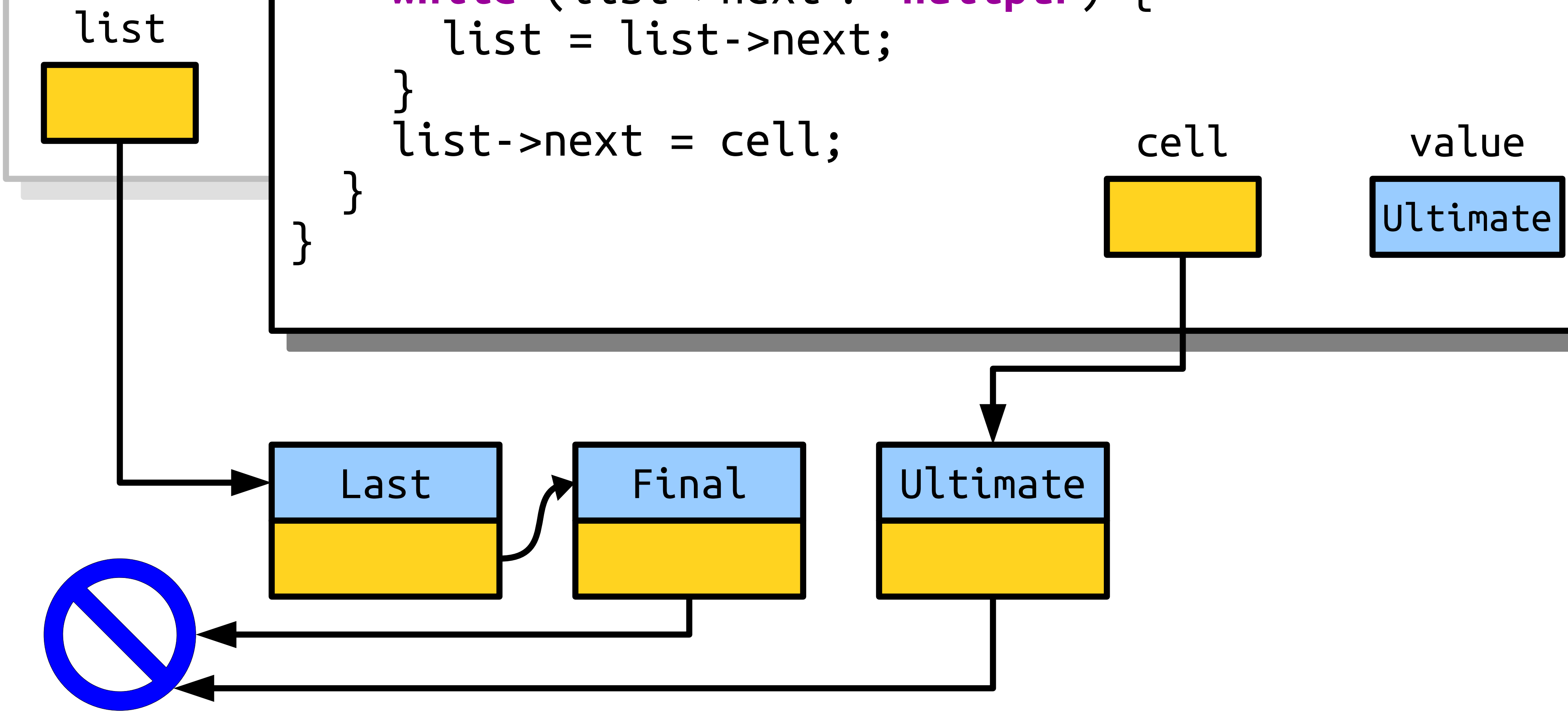


```
int main() {
   Cell* l
   appendT
   appendT
   appendT
   appendT

   /* … ot
}
```

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

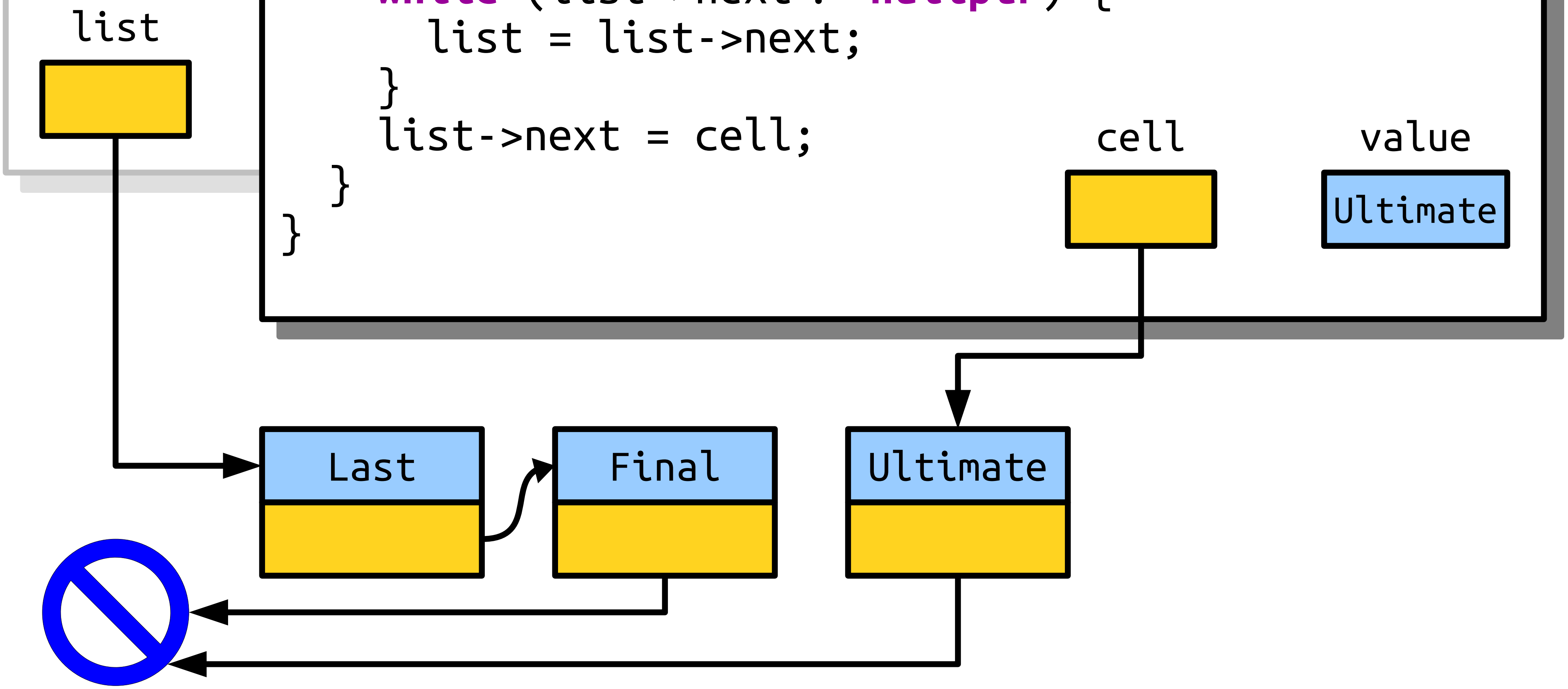
```
int main() {
   Cell* l
   appendT
   appendT
   appendT
   appendT

   /* … ot
}
```

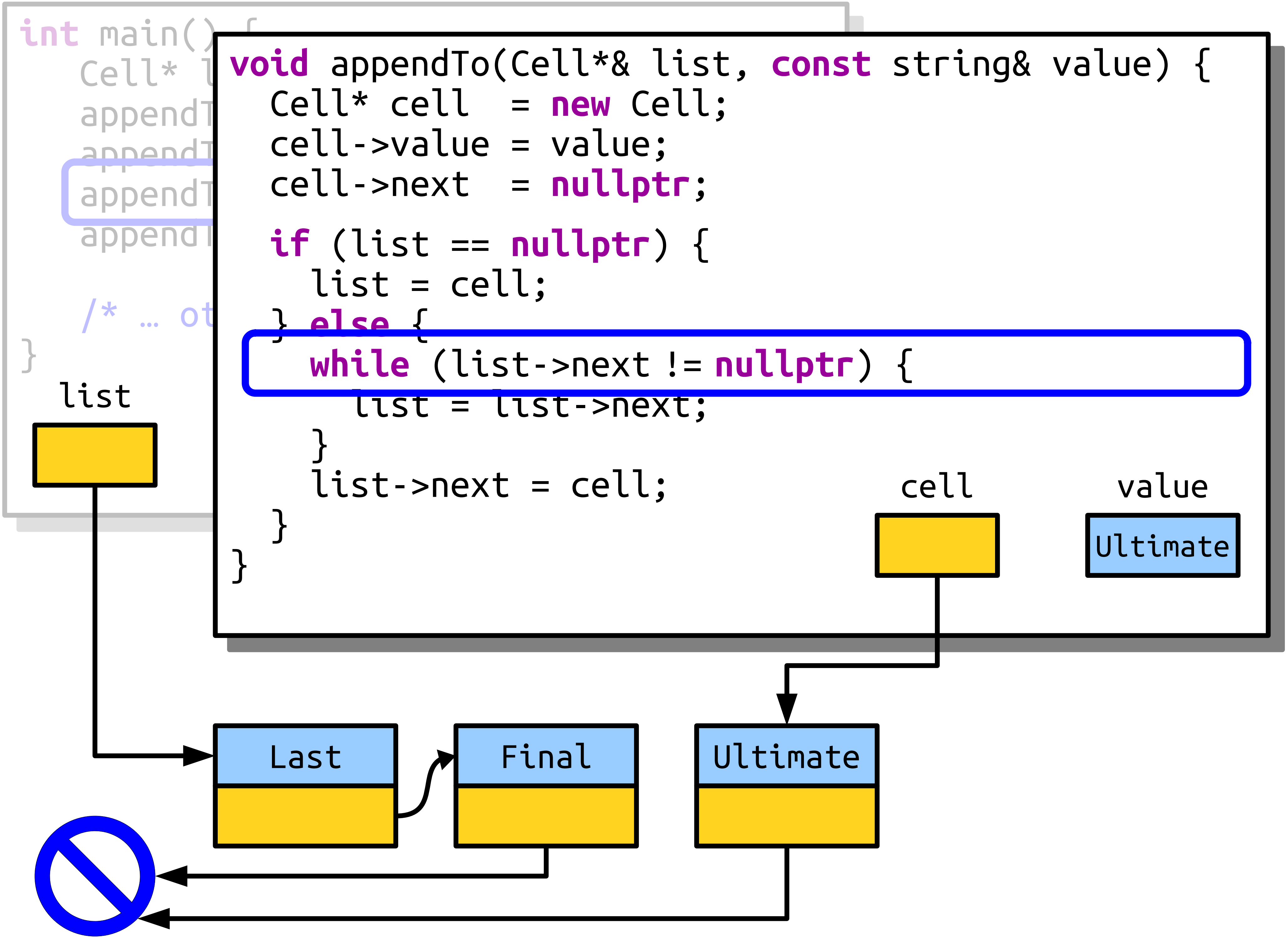
```
void appendTo(Cell*& list, const string& value) {
  Cell* cell  = new Cell;
  cell->value = value;
  cell->next  = nullptr;

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



```
int main() {
   Cell* l
   appendT
   appendT
   appendT
   appendT

   /* … ot
}
```

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

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



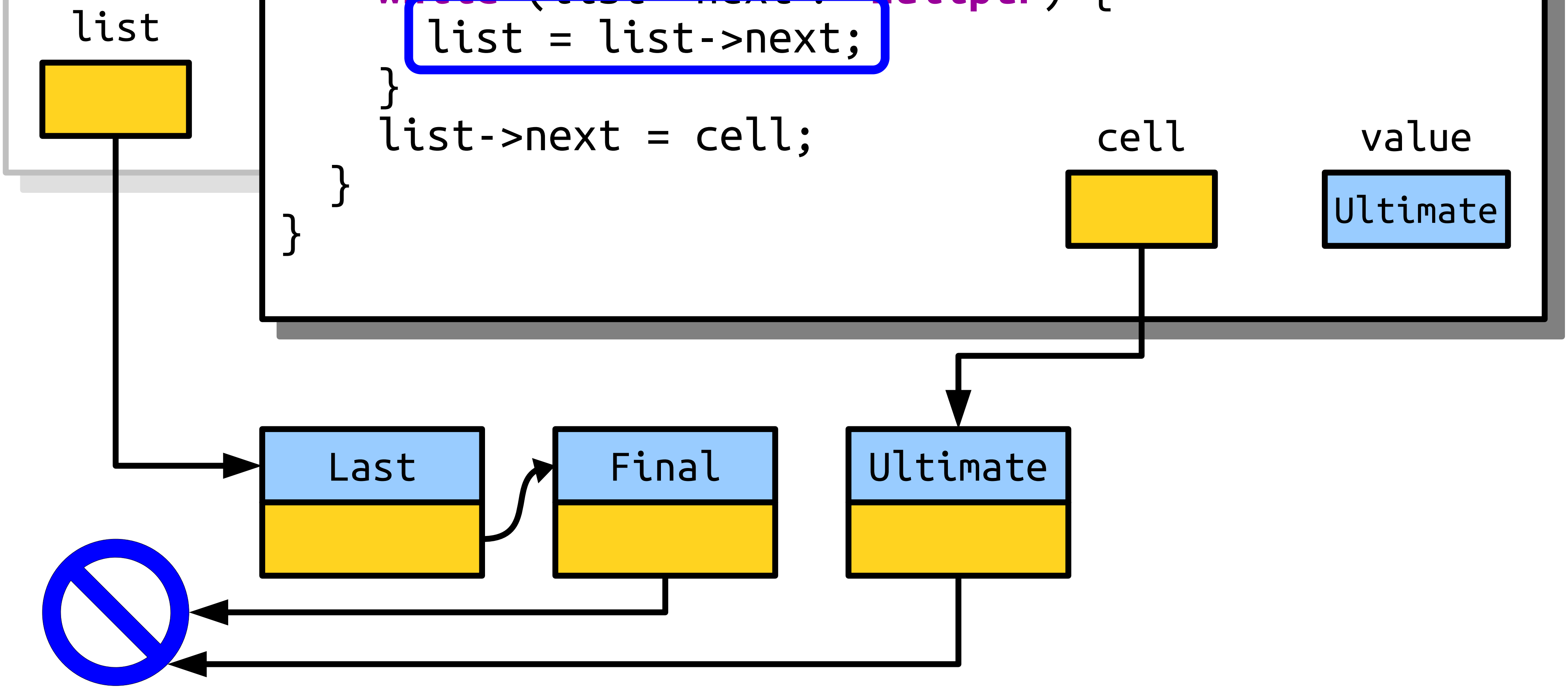


```
int main() {
   Cell* l
   appendT
   appendT
   appendT
   appendT

   /* … ot
}
```

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

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

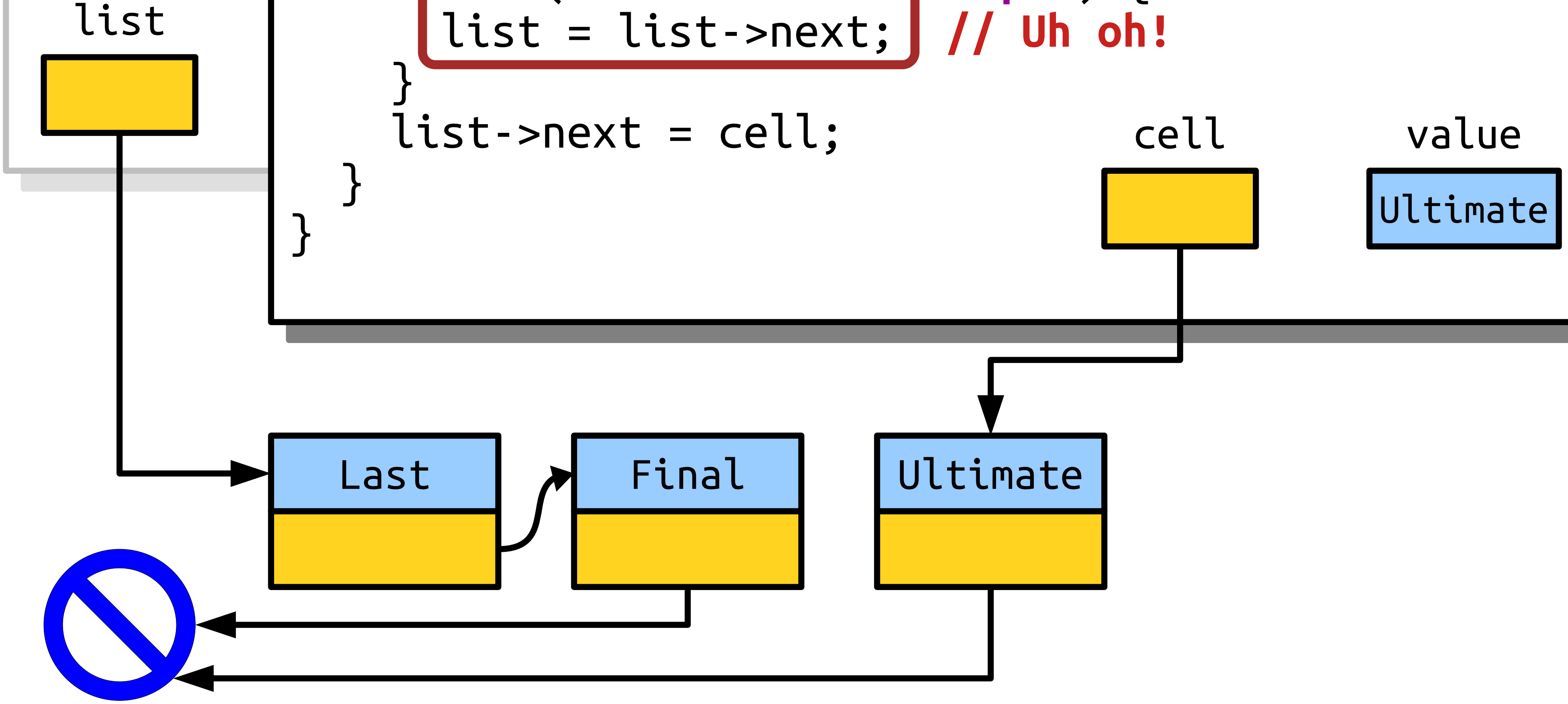
```
int main() {
   Cell* l
   appendT
   appendT
   appendT
   appendT

   /* … ot
}
```

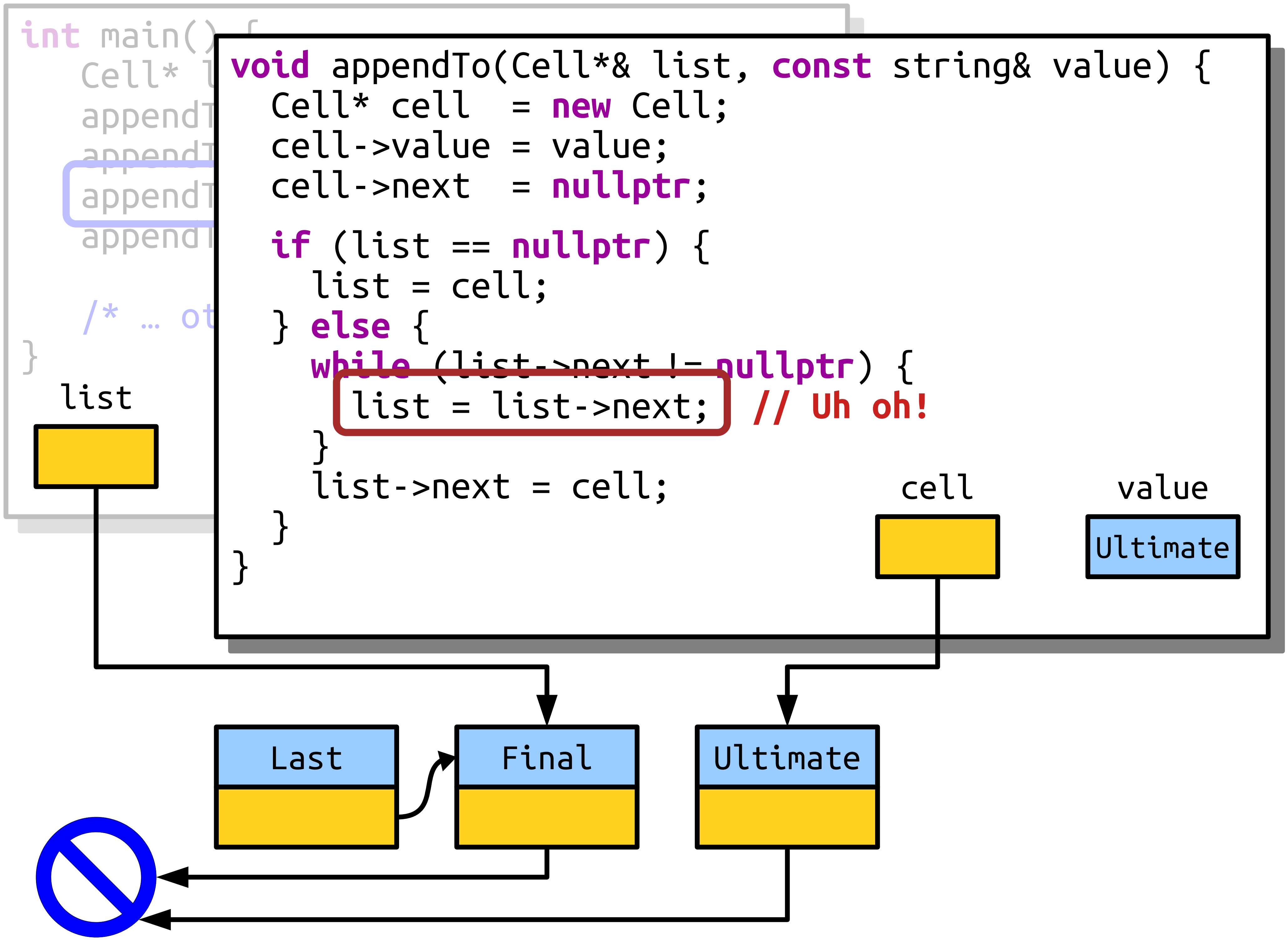
```
void appendTo(Cell*& list, const string& value) {
  Cell* cell  = new Cell;
  cell->value = value;
  cell->next  = nullptr;

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



```
int main() {
   Cell* l
   appendT
   appendT
   appendT
   appendT

   /* … ot
}
```

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

  if (list == nullptr) {
    list = cell;
  } else {
    while (list->next != nullptr) {
      list = list->next;  // Uh oh!
    }
    list->next = cell;
  }
}
```



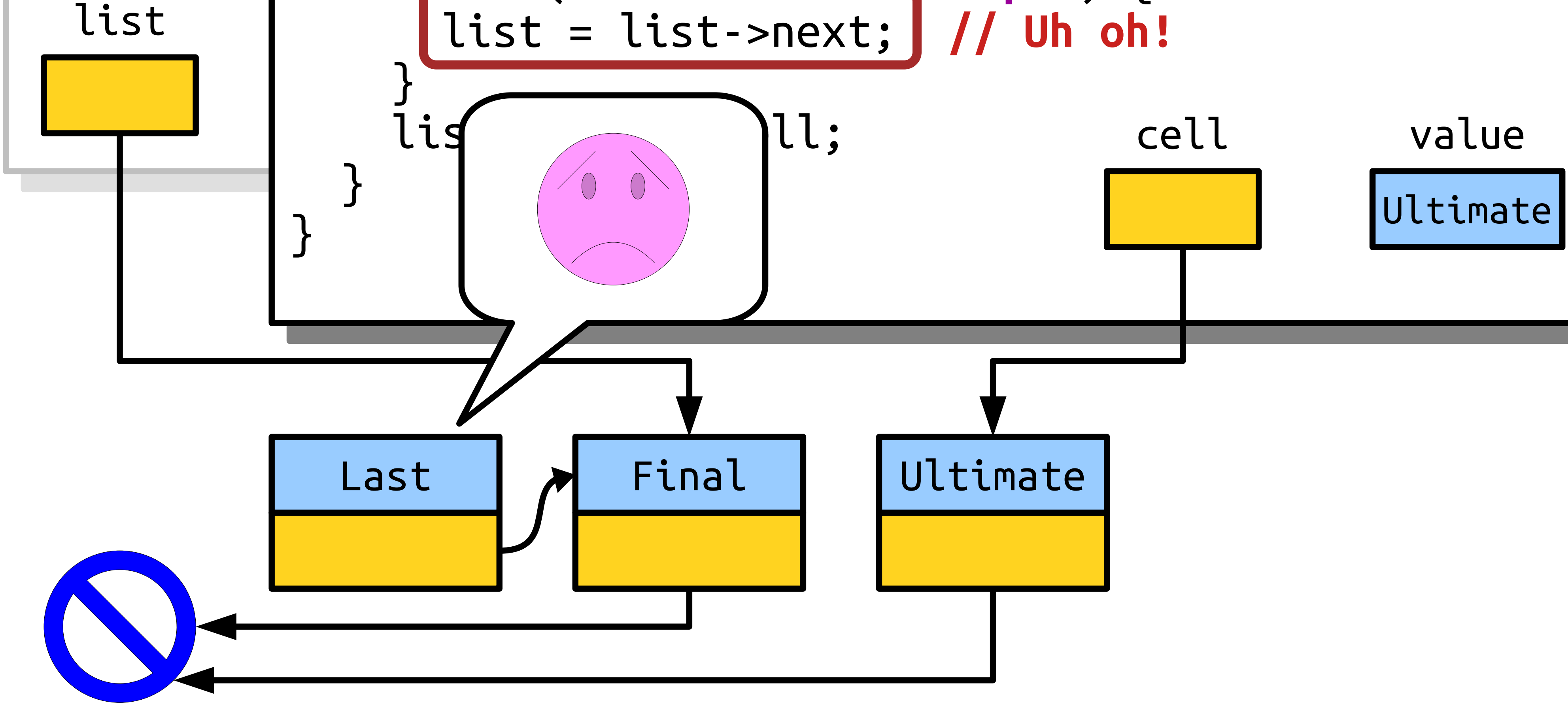


```
int main() {
   Cell* l
   appendT
   appendT
   appendT
   appendT

   /* … ot
}
```

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

  if (list == nullptr) {
    list = cell;
  } else {
    while (list->next != nullptr) {
      list = list->next;  // Uh oh!
    }
    list->next = cell;
  }
}
```

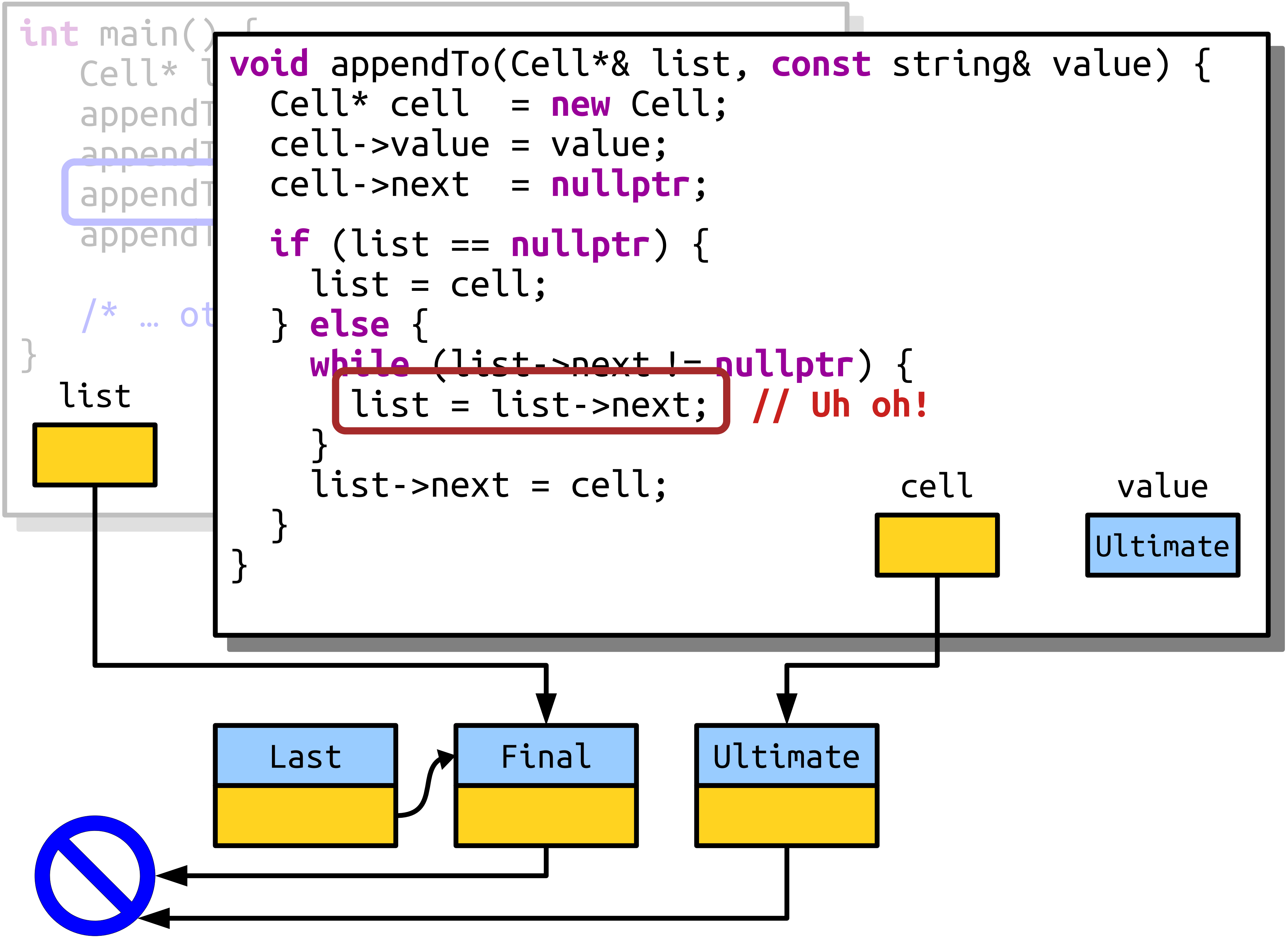
```
int main() {
   Cell* l
   appendT
   appendT
   appendT
   appendT

   /* … ot
}
```

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

  if (list == nullptr) {
    list = cell;
  } else {
    while (list->next != nullptr) {
      list = list->next; // Uh oh!
    }
    lis          ll;
  }
}
```



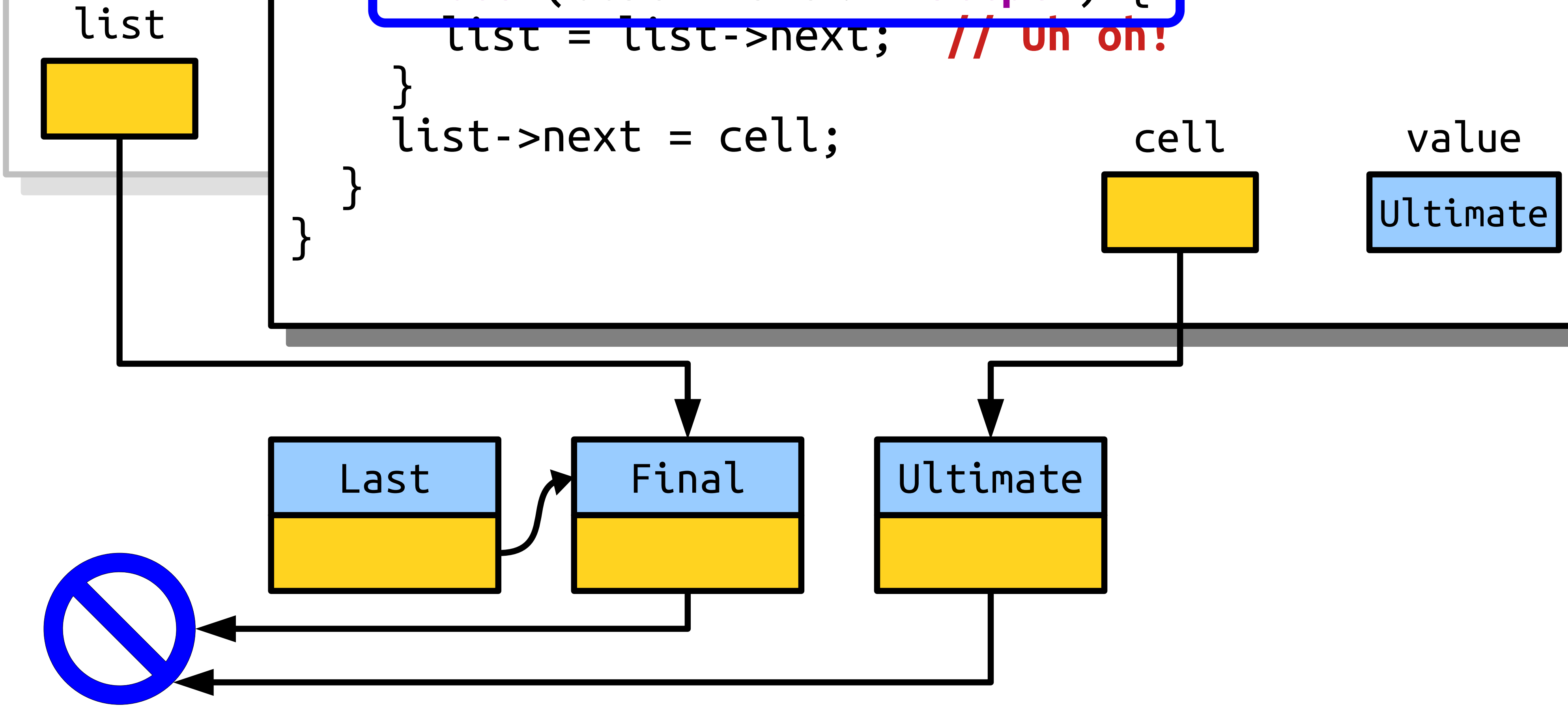
```
int main() {
   Cell* l
   appendT
   appendT
   appendT
   appendT

   /* … ot
}
```

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

  if (list == nullptr) {
    list = cell;
  } else {
    while (list->next != nullptr) {
      list = list->next;  // Uh oh!
    }
    list->next = cell;
  }
}
```



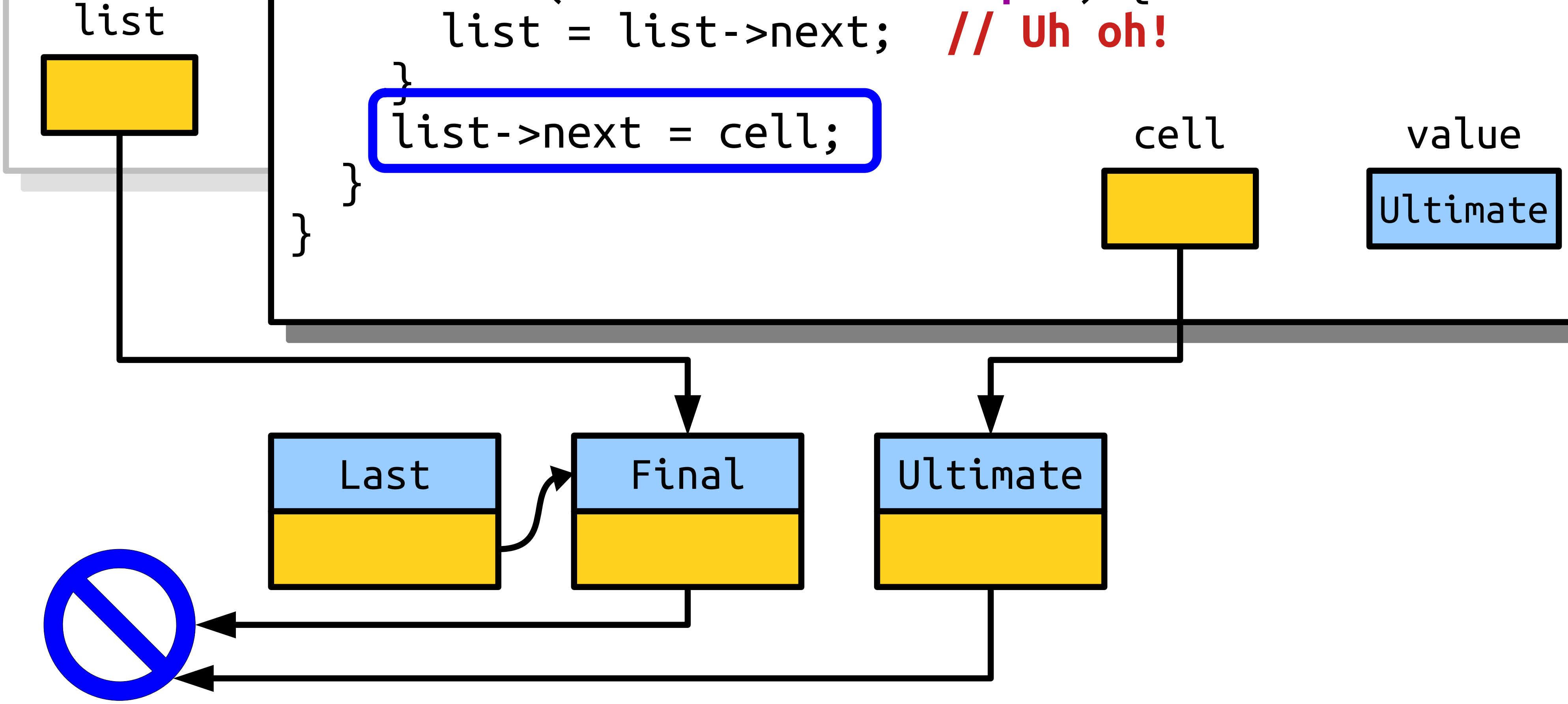
```
int main() {
   Cell* l
   appendT
   appendT
   appendT
   appendT

   /* … ot
}
```

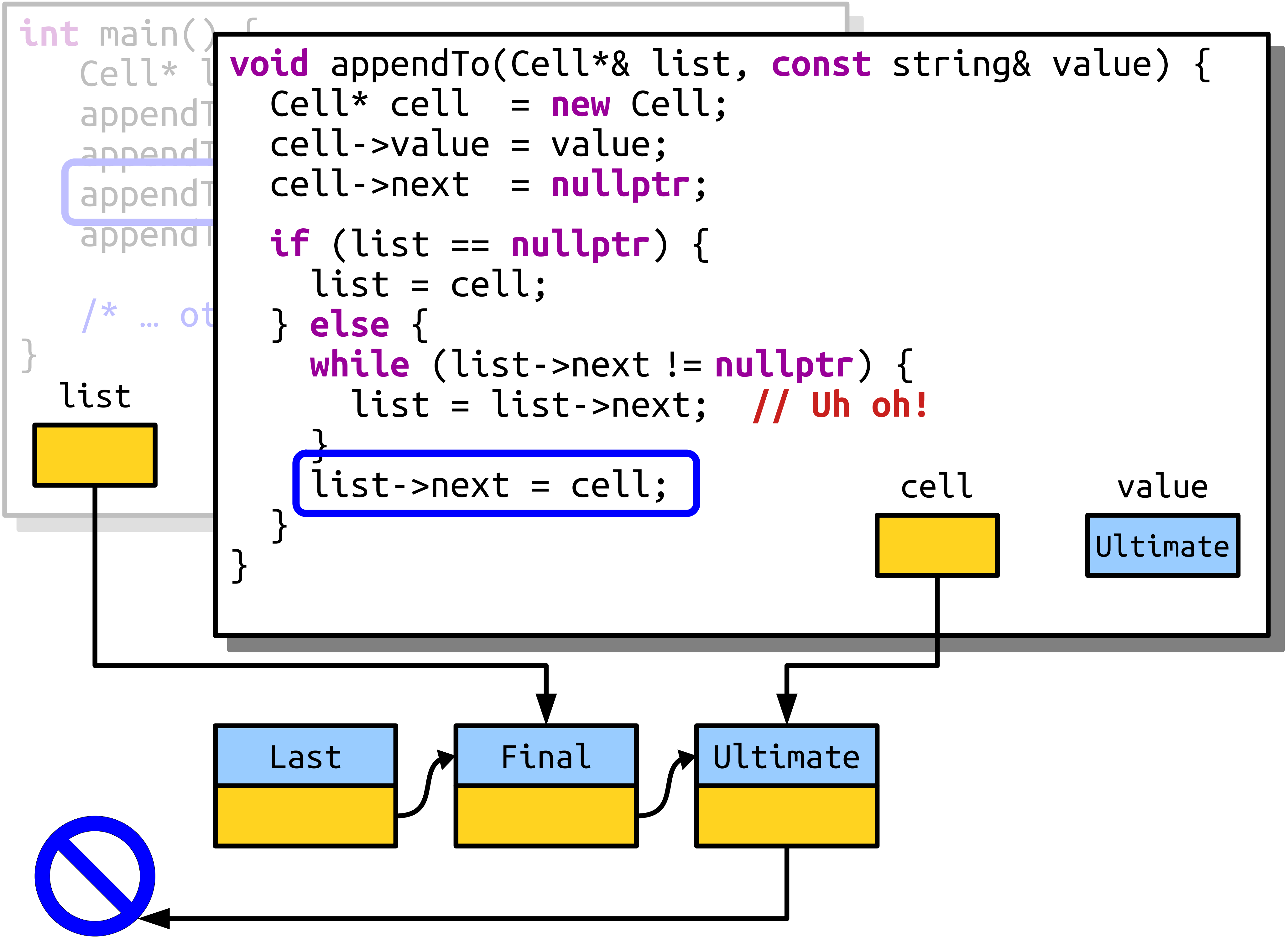
```
void appendTo(Cell*& list, const string& value) {
  Cell* cell  = new Cell;
  cell->value = value;
  cell->next  = nullptr;

  if (list == nullptr) {
    list = cell;
  } else {
    while (list->next != nullptr) {
      list = list->next;   // Uh oh!
    }
    list->next = cell;
  }
}
```



```
int main() {
   Cell* l
   appendT
   appendT
   appendT
   appendT

   /* … ot
}
```

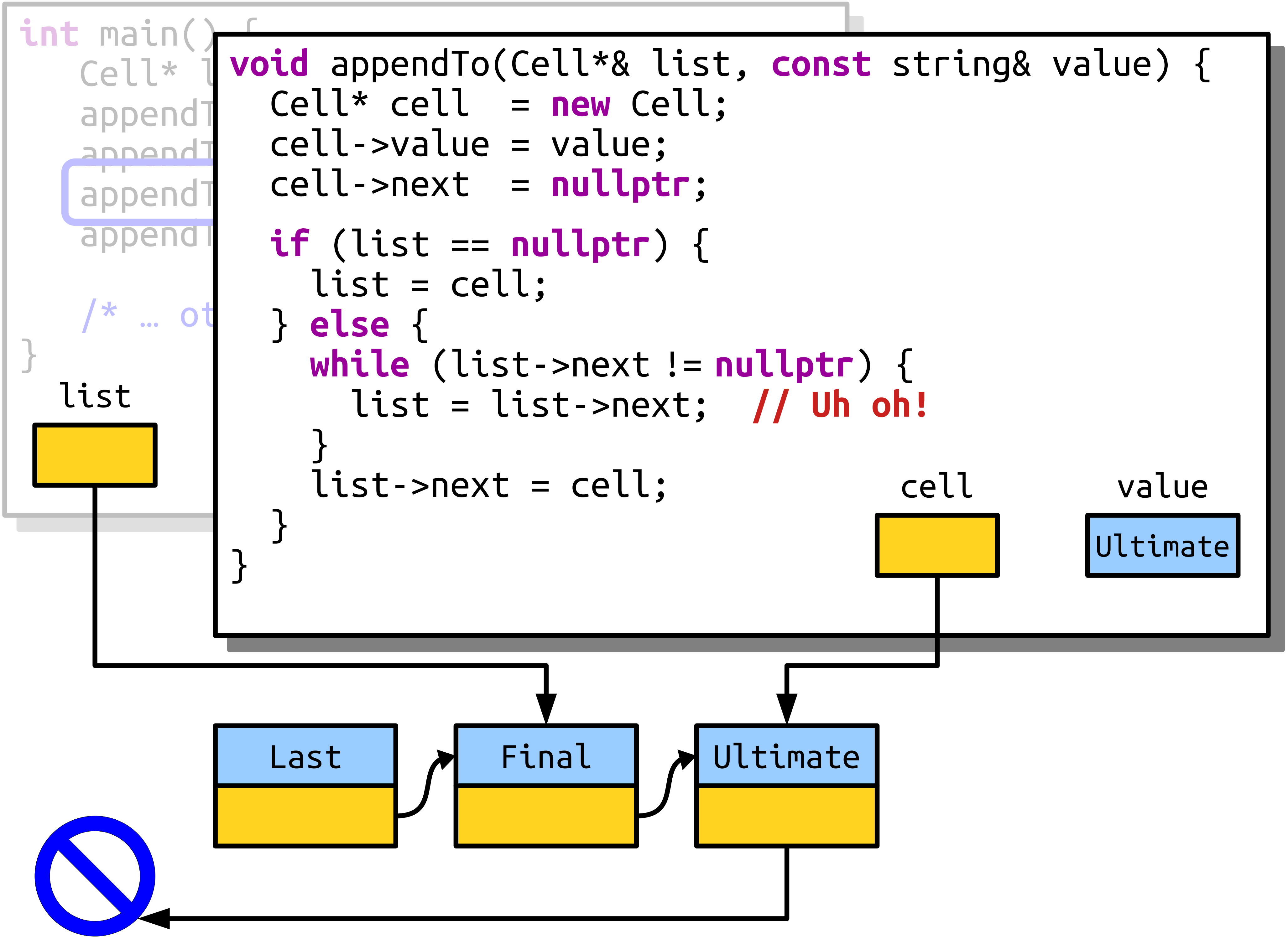
```
void appendTo(Cell*& list, const string& value) {
  Cell* cell  = new Cell;
  cell->value = value;
  cell->next  = nullptr;

  if (list == nullptr) {
    list = cell;
  } else {
    while (list->next != nullptr) {
      list = list->next;   // Uh oh!
    }
    list->next = cell;
  }
}
```





```
int main() {
   Cell* l
   appendT
   appendT
   appendT
   appendT

   /* … ot
}
```

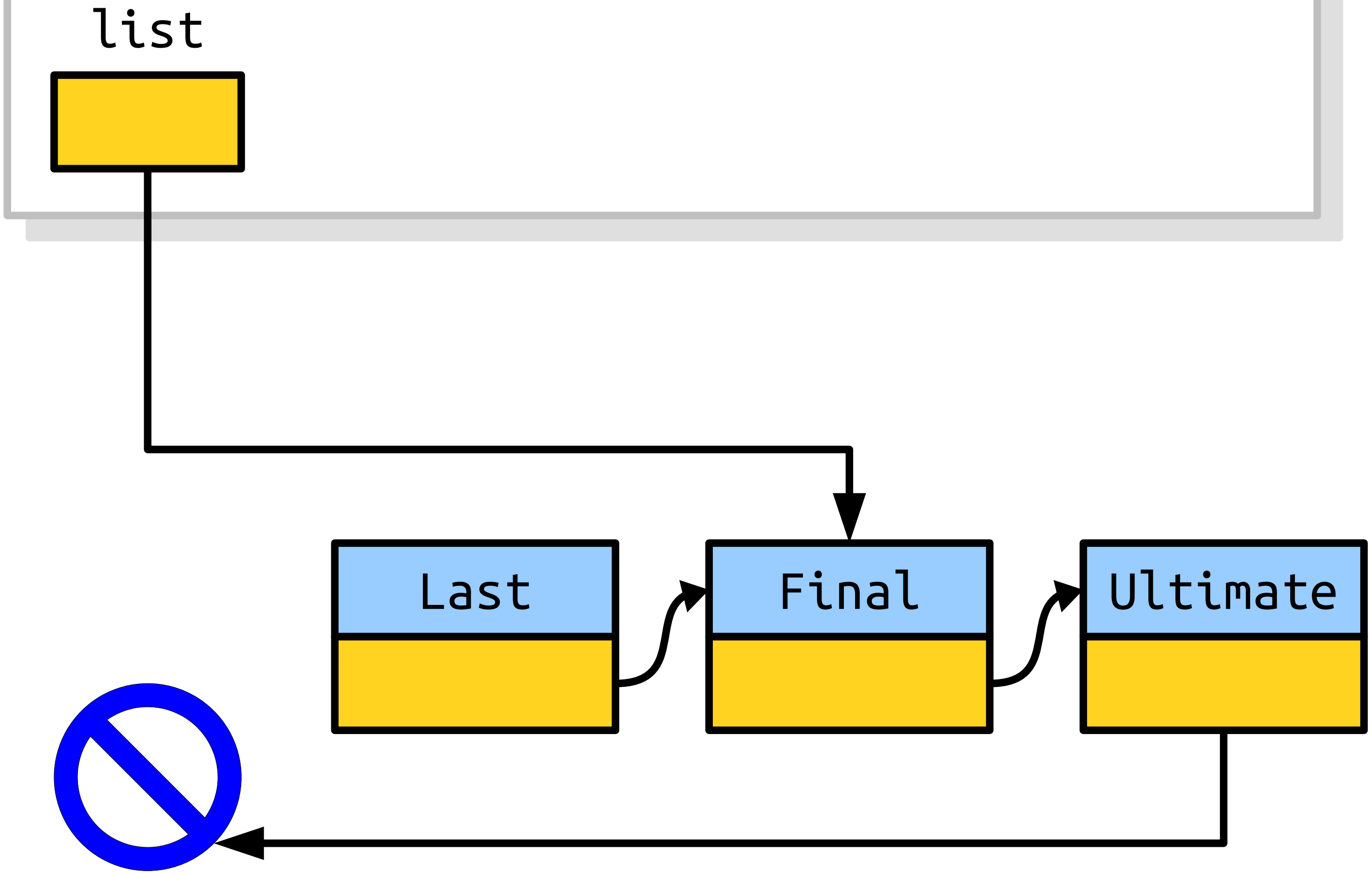
```
void appendTo(Cell*& list, const string& value) {
  Cell* cell  = new Cell;
  cell->value = value;
  cell->next  = nullptr;

  if (list == nullptr) {
    list = cell;
  } else {
    while (list->next != nullptr) {
      list = list->next;  // Uh oh!
    }
    list->next = cell;
  }
}
```



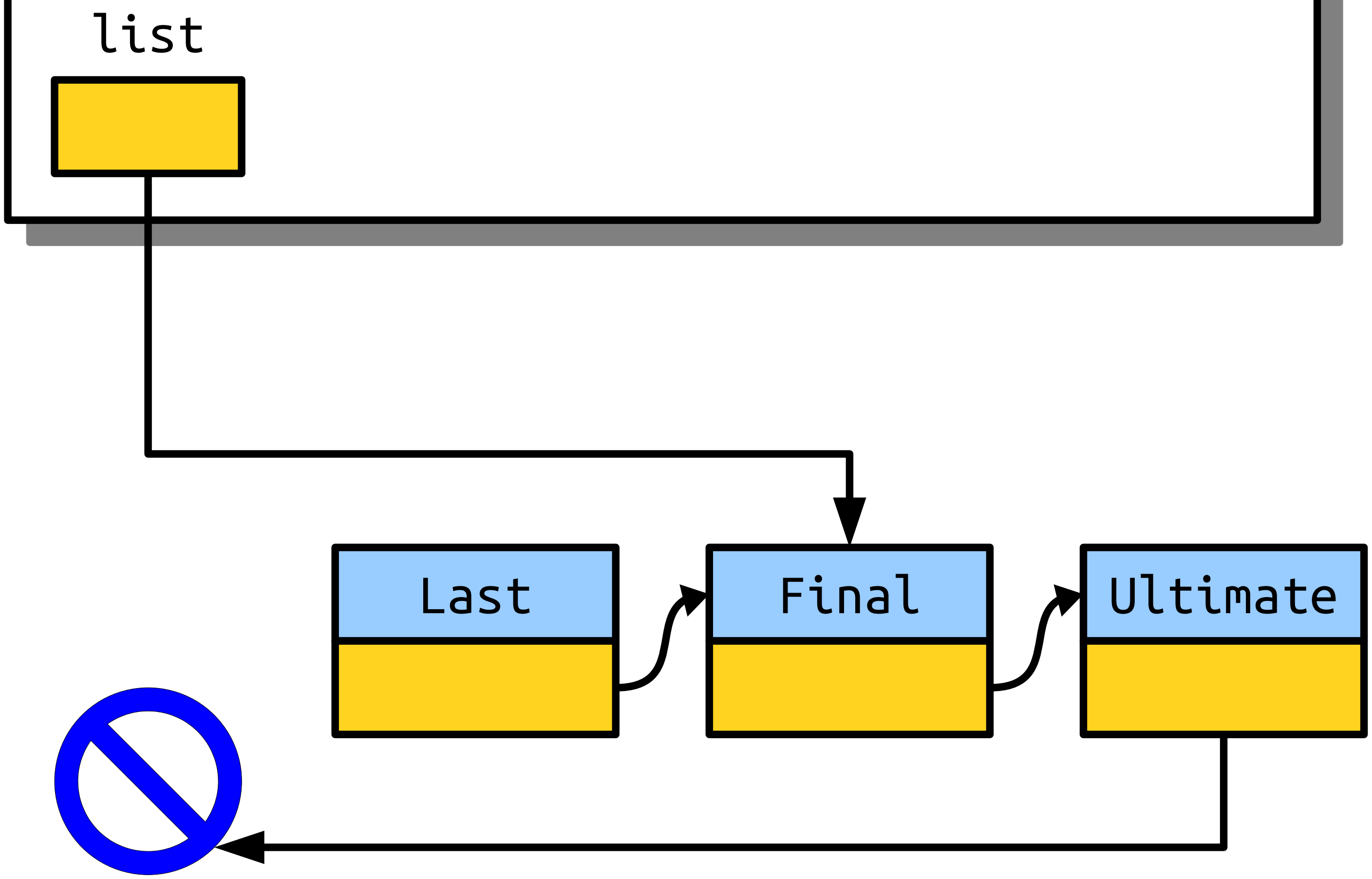
```
int main() {
   Cell* l
   appendT
   appendT
   appendT
   appendT

   /* … ot
}
```

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

  if (list == nullptr) {
    list = cell;
  } else {
    while (list->next != nullptr) {
      list = list->next;   // Uh oh!
    }
    list->next = cell;
  }
}
```

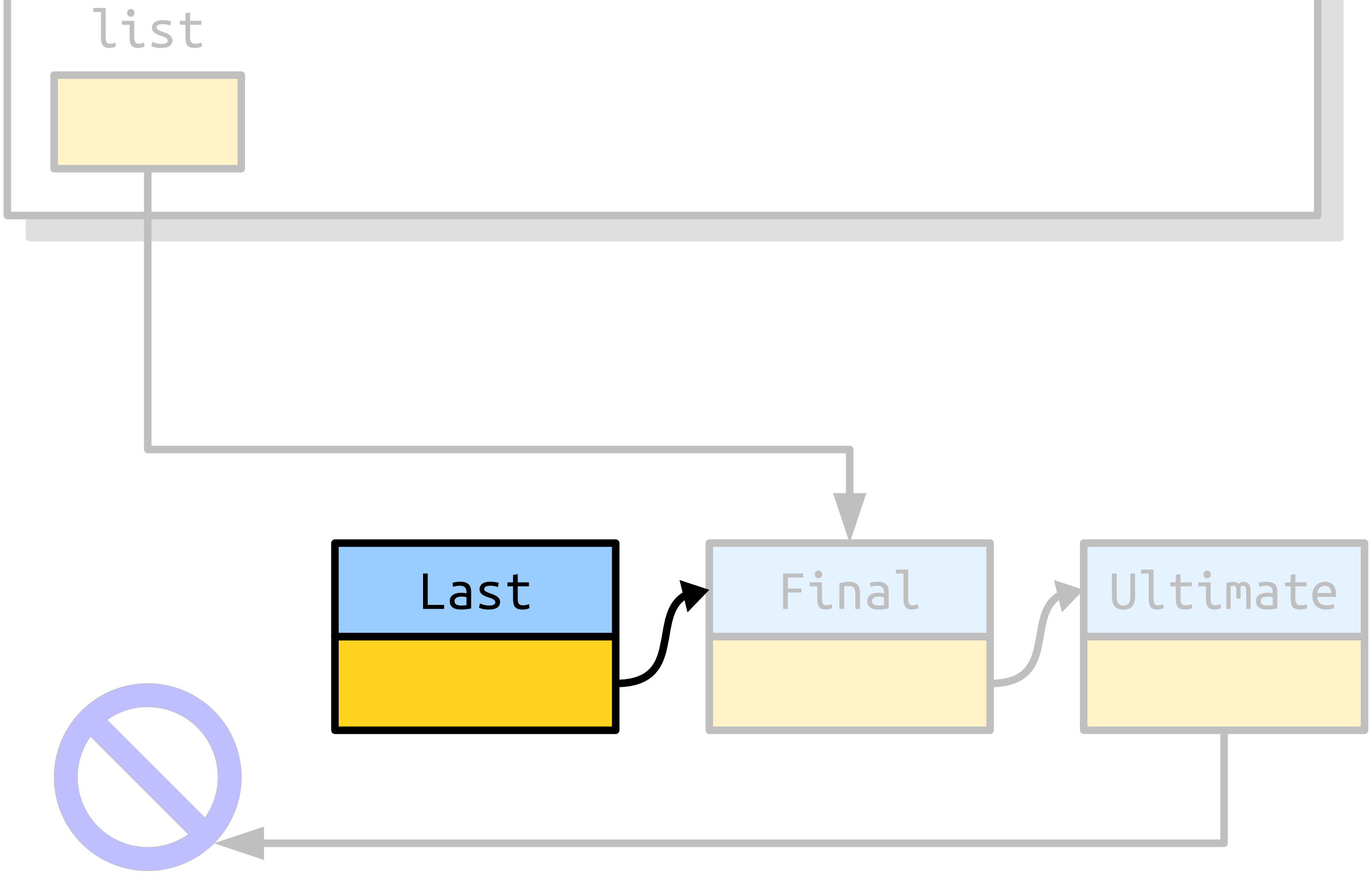
```
int main() {
   Cell* list = nullptr;
   appendTo(list, "Last");
   appendTo(list, "Final");
   appendTo(list, "Ultimate");
   appendTo(list, "Terminal");

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



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

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

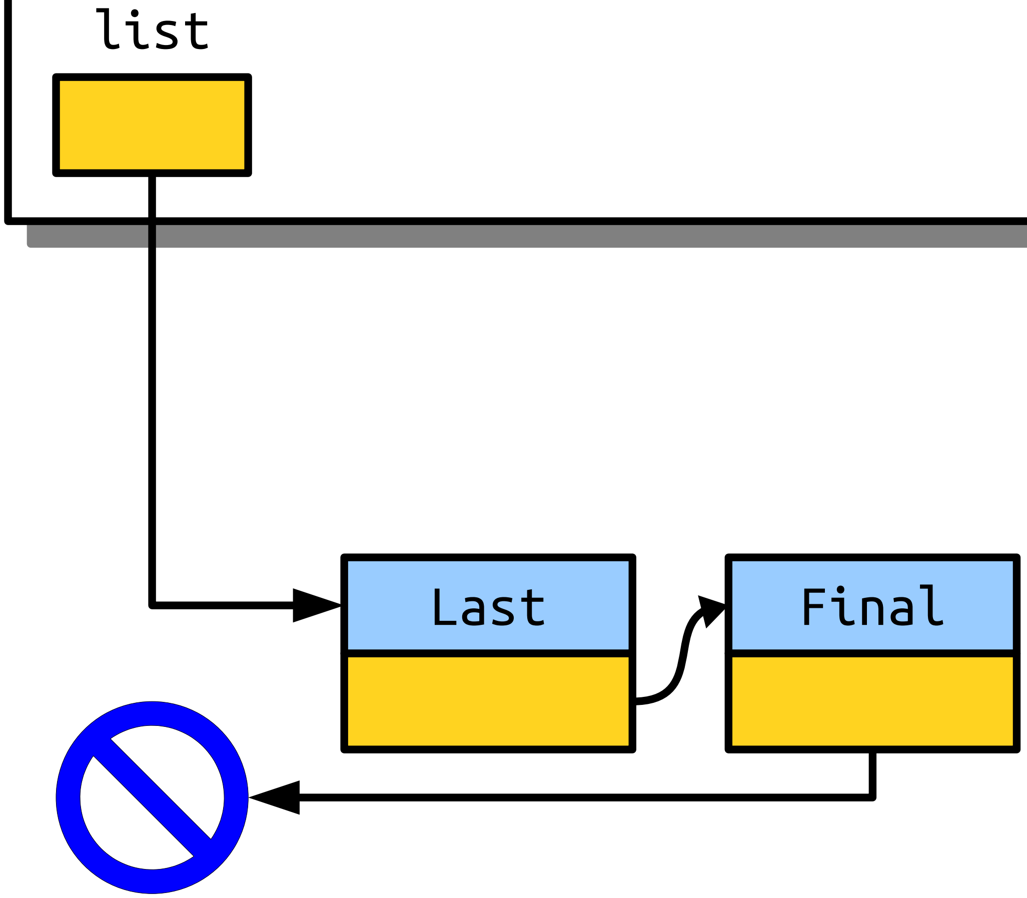


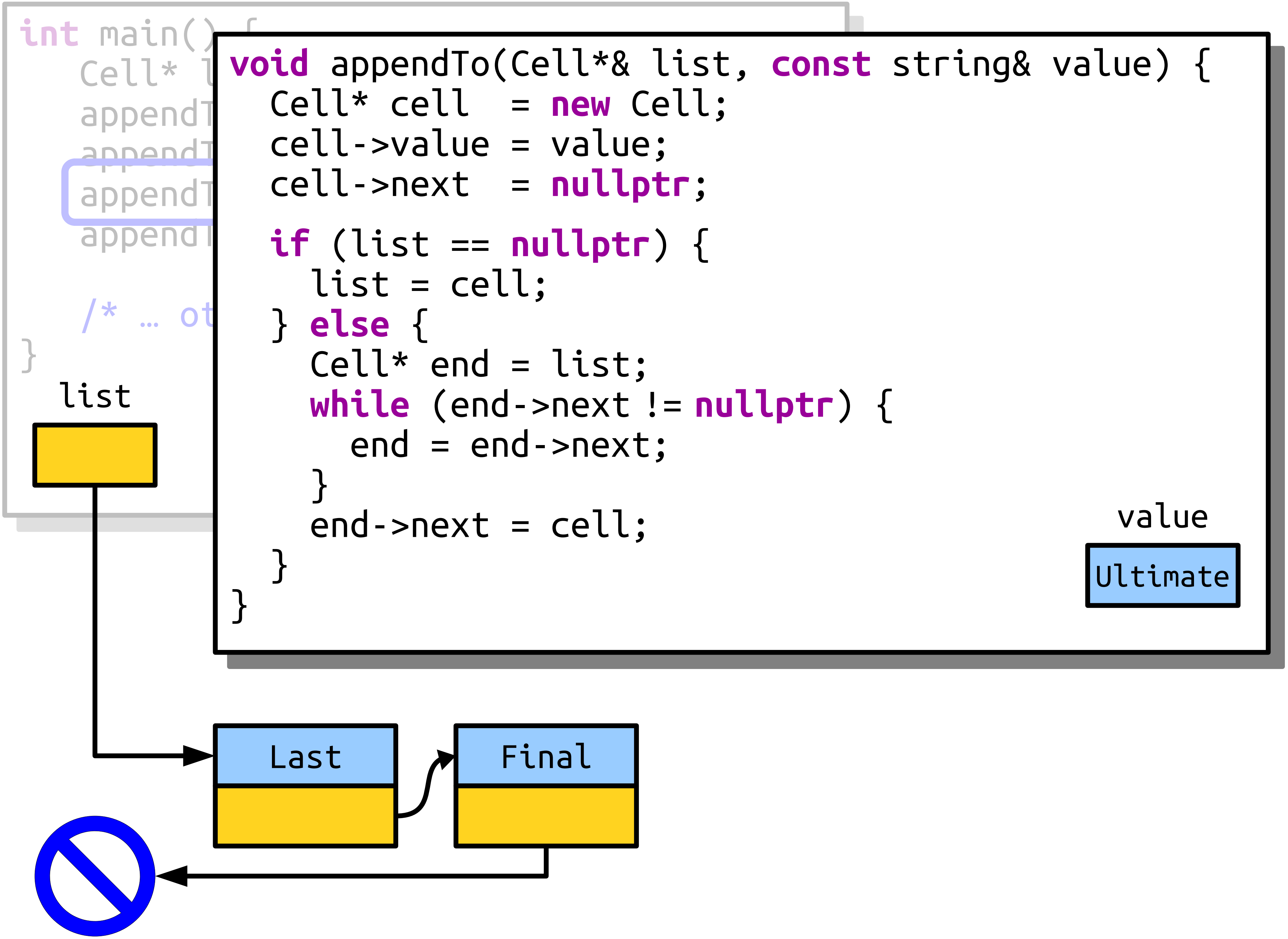
```
int main() {
   Cell* list = nullptr;
   appendTo(list, "Last");
   appendTo(list, "Final");
   appendTo(list, "Ultimate");
   appendTo(list, "Terminal");

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



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

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

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



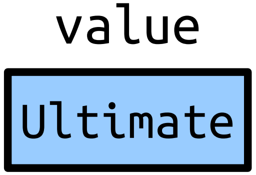


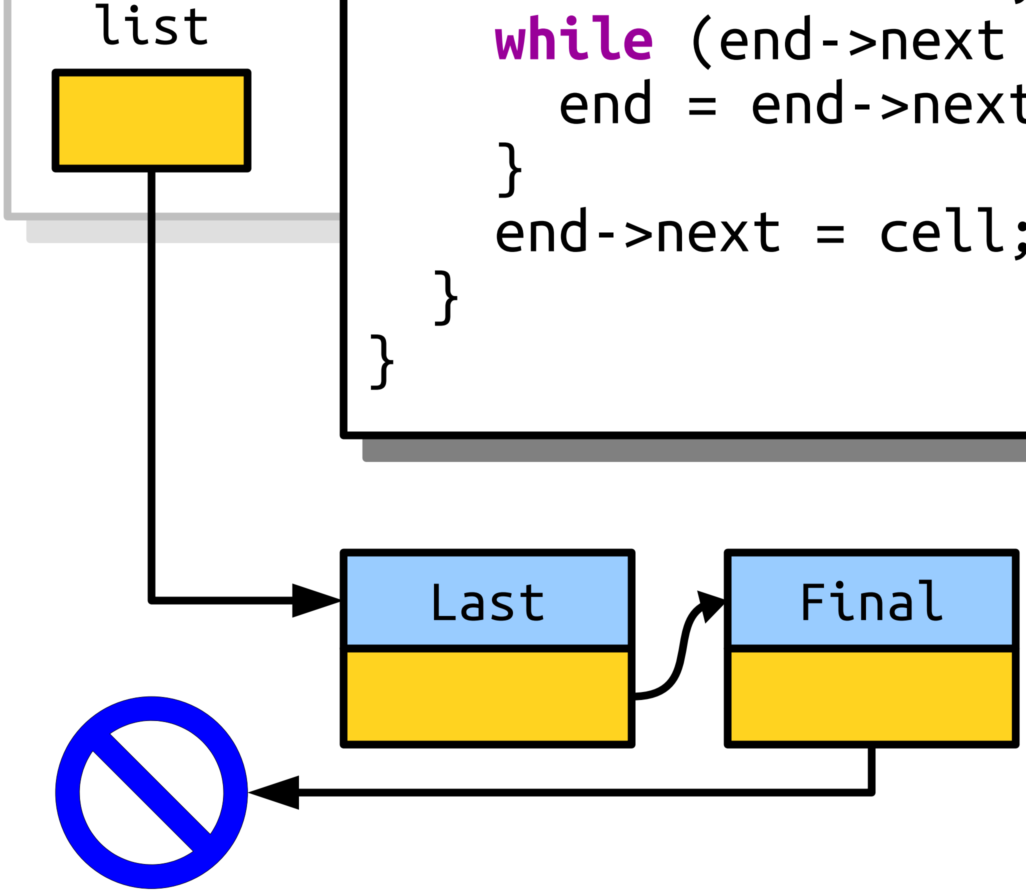
```
int main() {
   Cell* l
   appendT
   appendT
   appendT
   appendT

   /* … ot
}
```

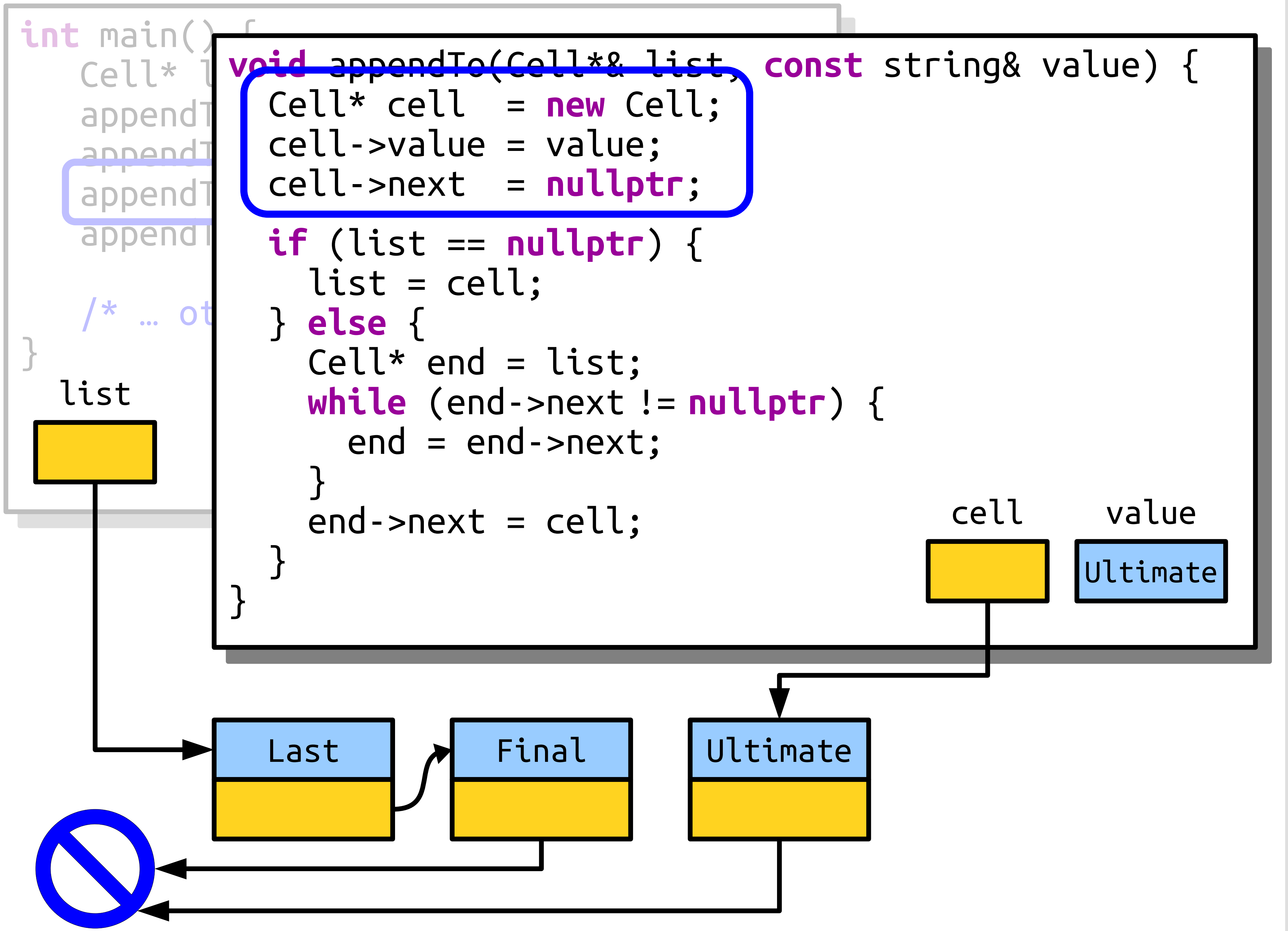
```
void appendTo(Cell*& list, const string& value) {
  Cell* cell  = new Cell;
  cell->value = value;
  cell->next  = nullptr;

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

```
int main() {
   Cell* l
   appendT
   appendT
   appendT
   appendT

   /* … ot
}
```

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





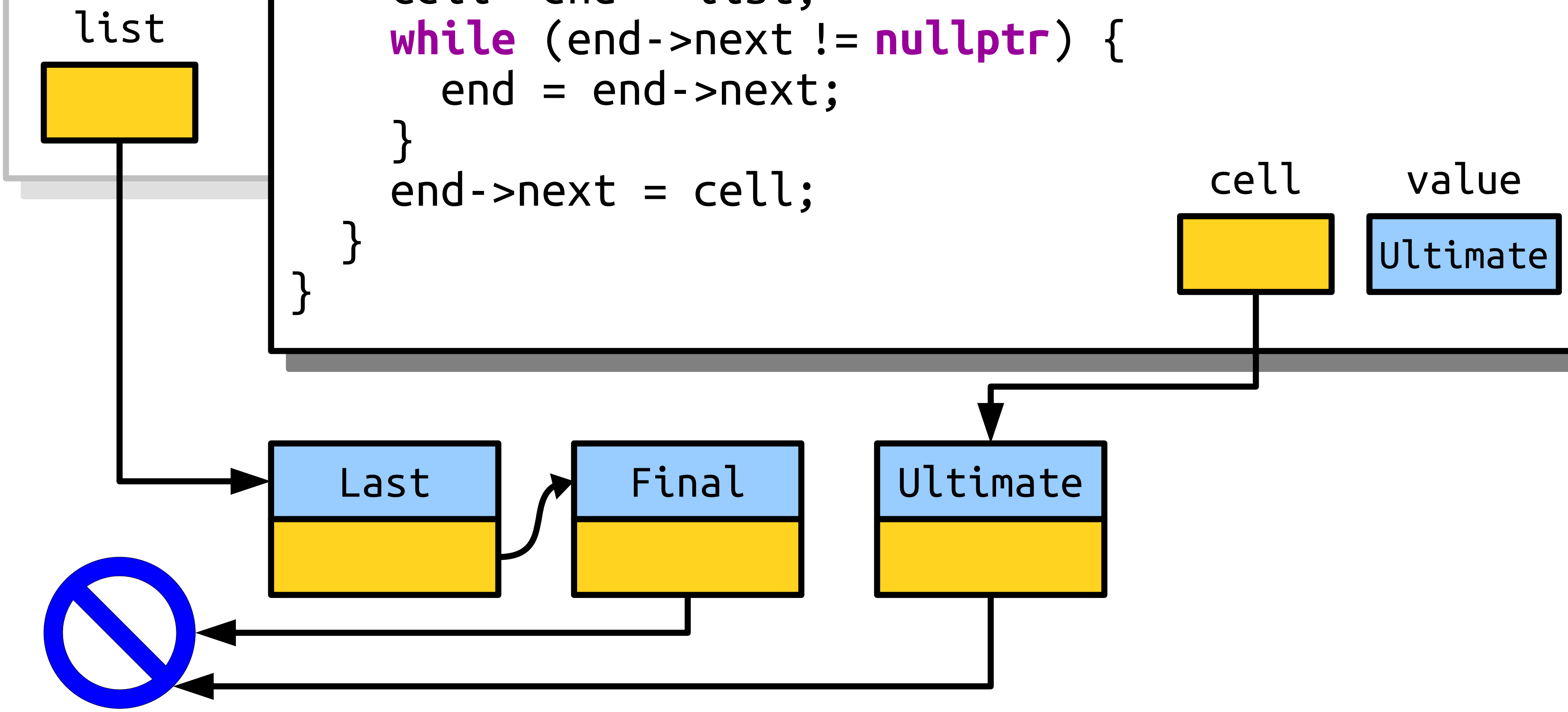


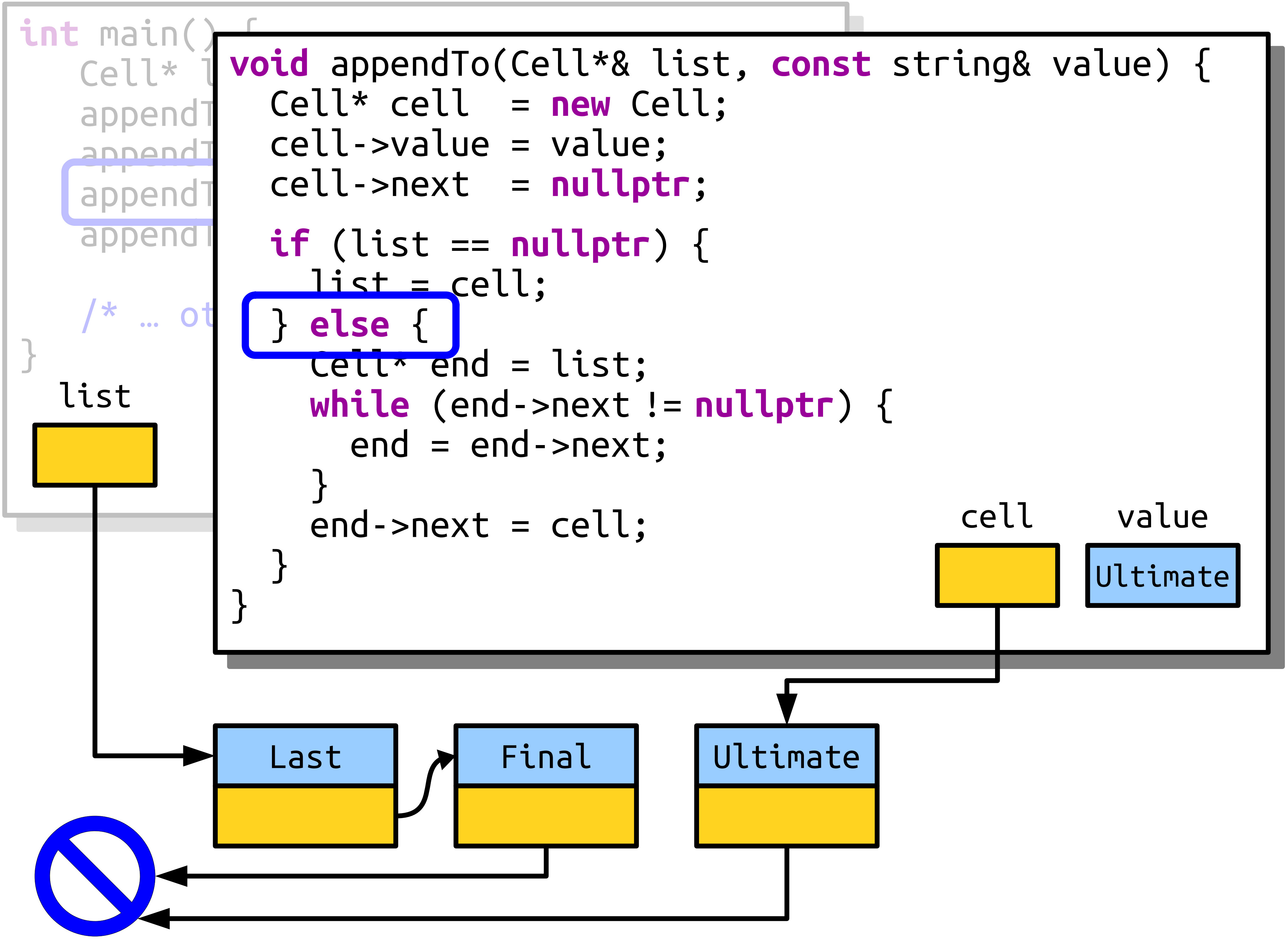
```
int main() {
   Cell* l
   appendT
   appendT
   appendT
   appendT

   /* … ot
}
```

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

```
int main() {
   Cell* l
   appendT
   appendT
   appendT
   appendT

   /* … ot
}
```

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

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

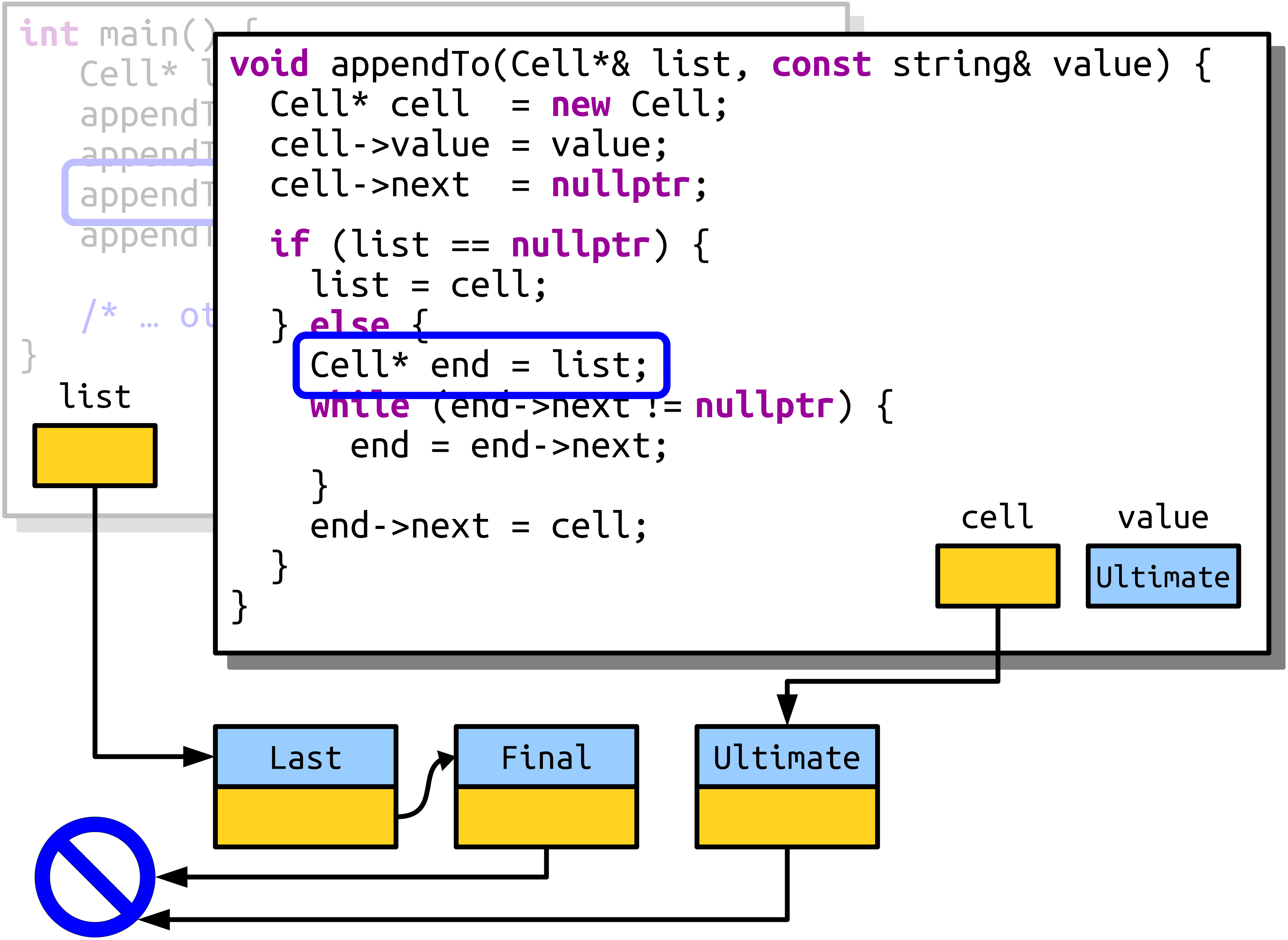


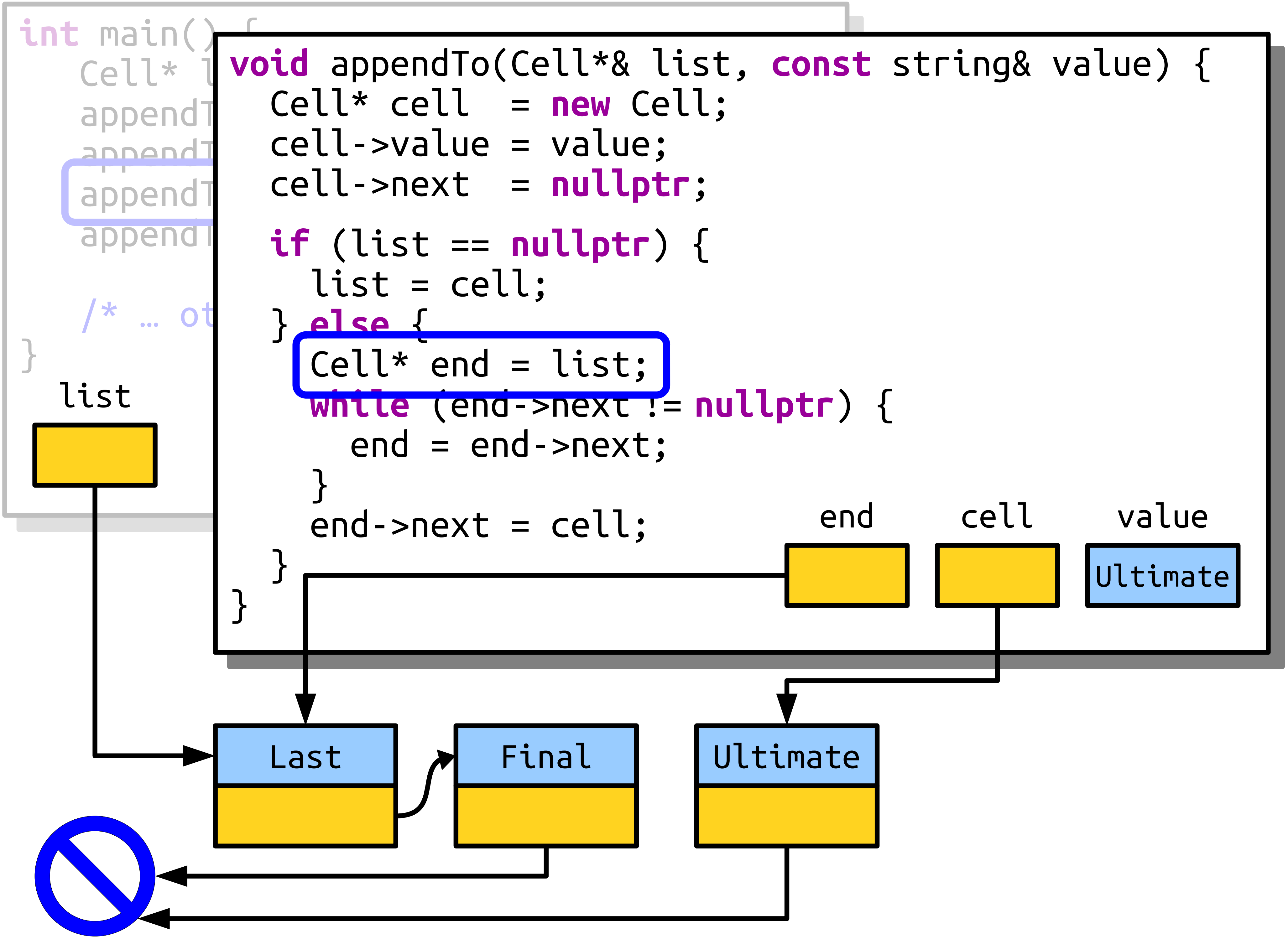


```
void appendTo(Cell*& list, const string& value) {
  Cell* cell  = new Cell;
  cell->value = value;
  cell->next  = nullptr;

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

```
int main() {
   Cell* l
   appendT
   appendT
   appendT
   appendT

   /* … ot
}
```

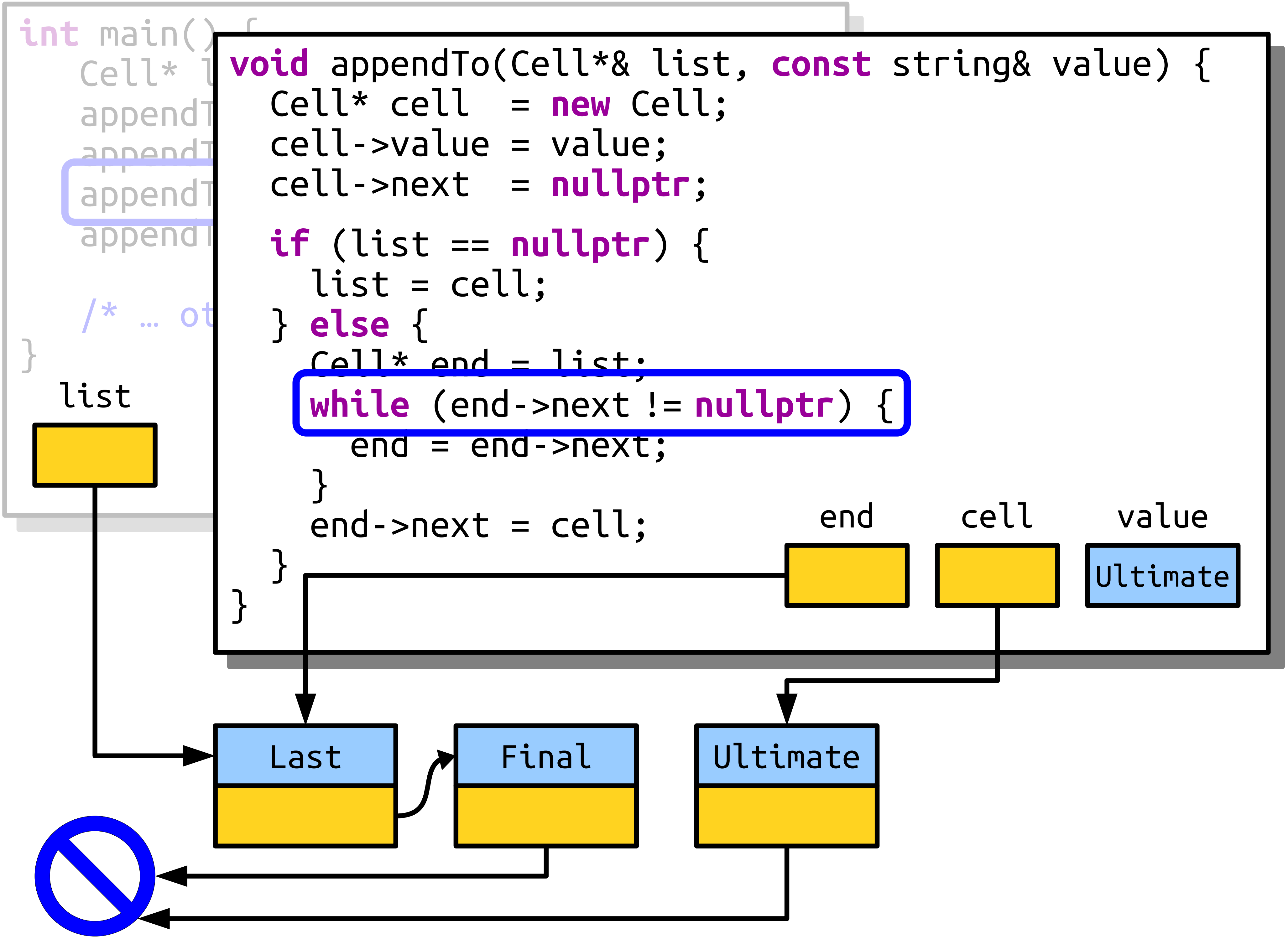
```
void appendTo(Cell*& list, const string& value) {
  Cell* cell  = new Cell;
  cell->value = value;
  cell->next  = nullptr;

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





```
int main() {
   Cell* l
   appendT
   appendT
   appendT
   appendT

   /* … ot
}
```

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

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

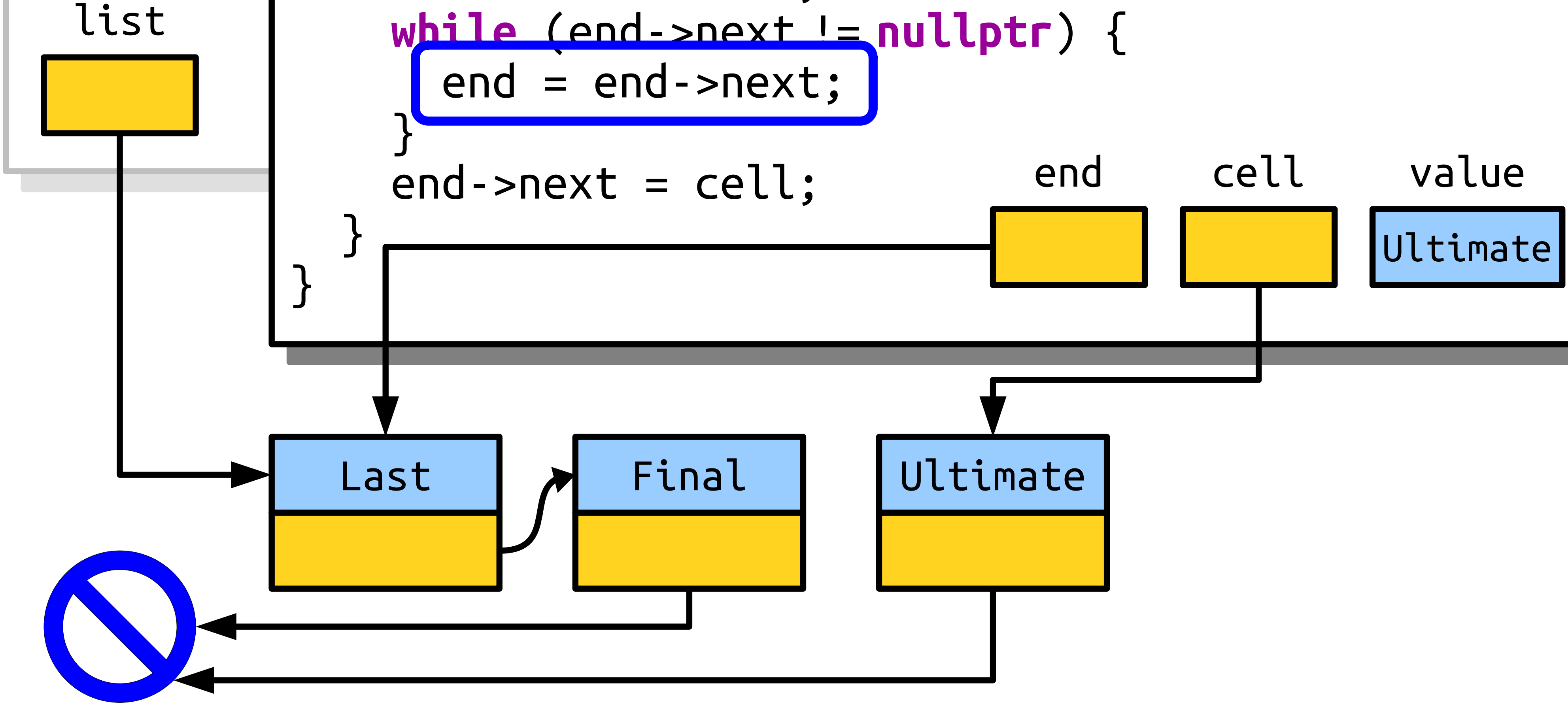


```
int main() {
   Cell* l
   appendT
   appendT
   appendT
   appendT

   /* … ot
}
```

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

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

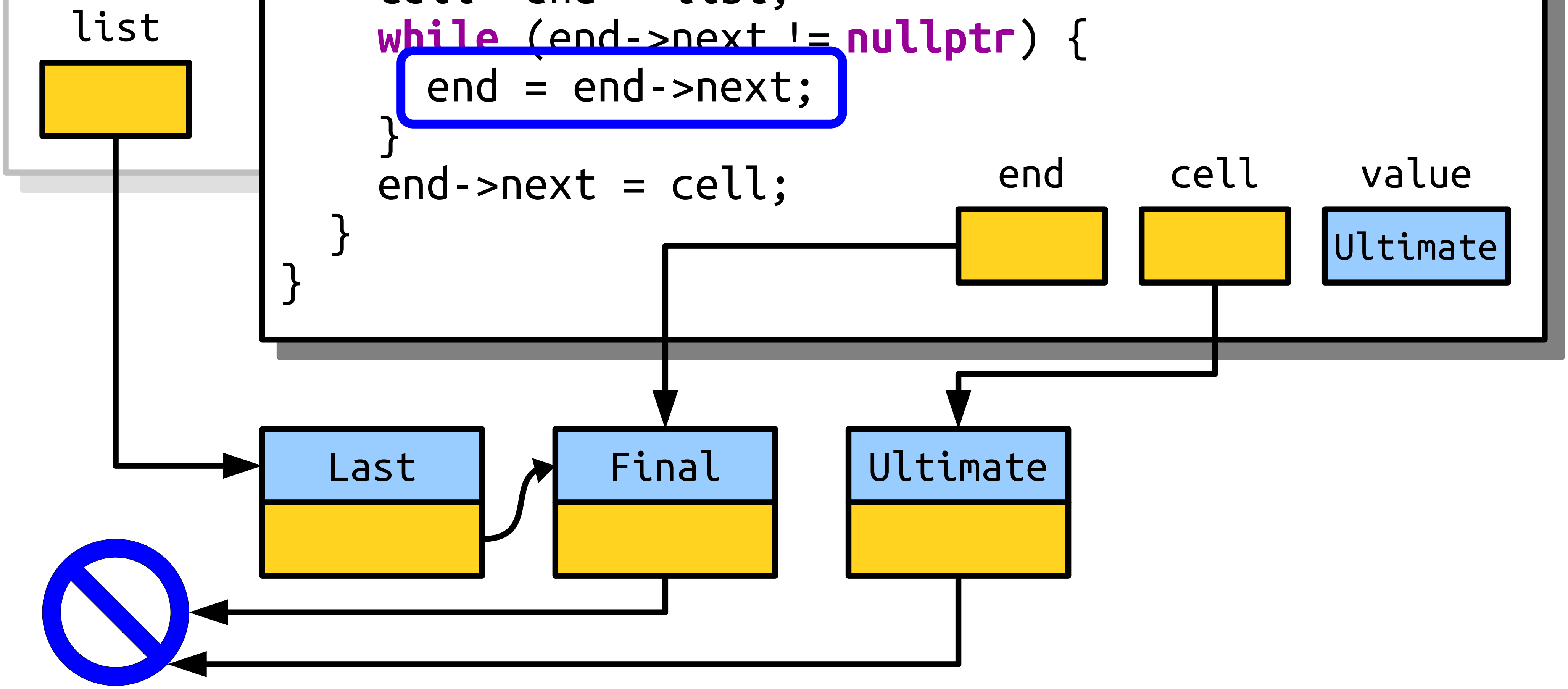
```
int main() {
   Cell* l
   appendT
   appendT
   appendT
   appendT

   /* … ot
}
```

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

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



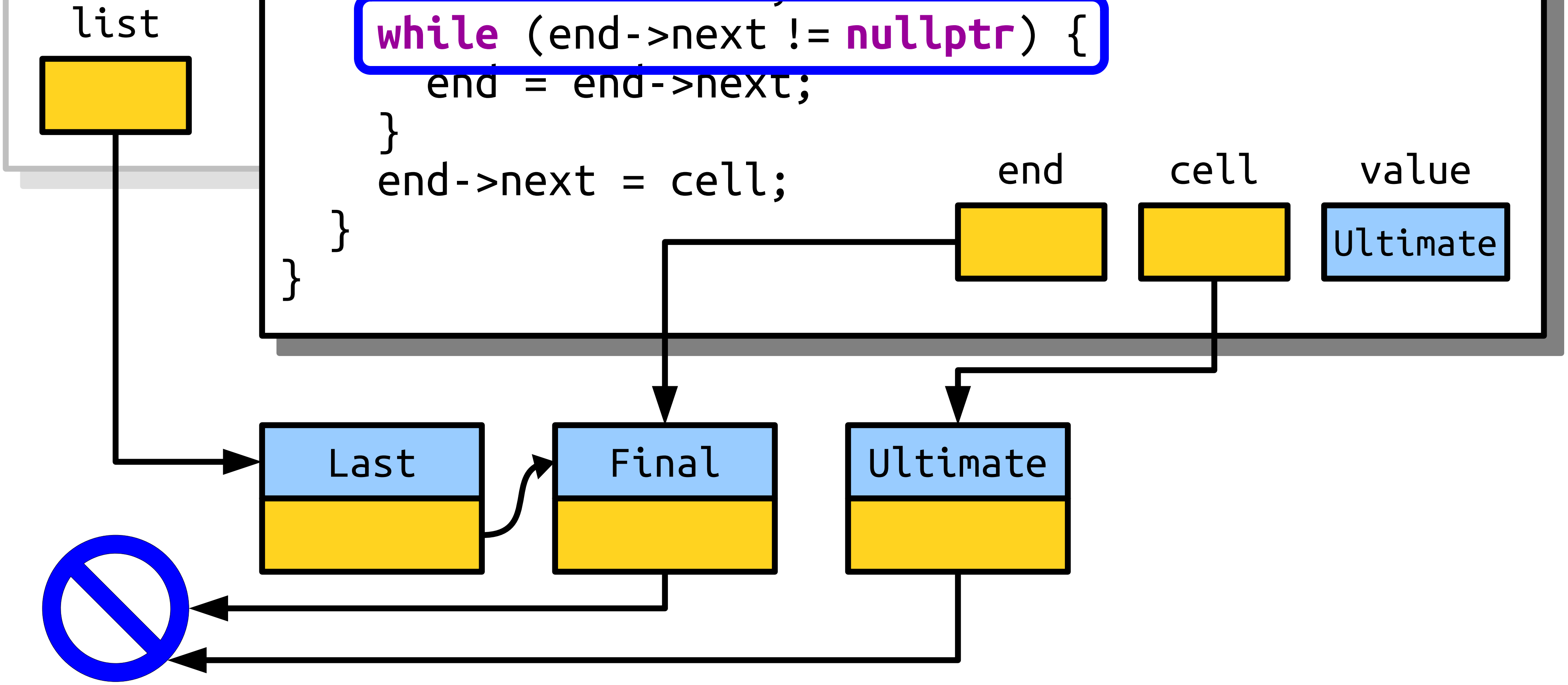
```
int main() {
   Cell* l
   appendT
   appendT
   appendT
   appendT

   /* … ot
}
```

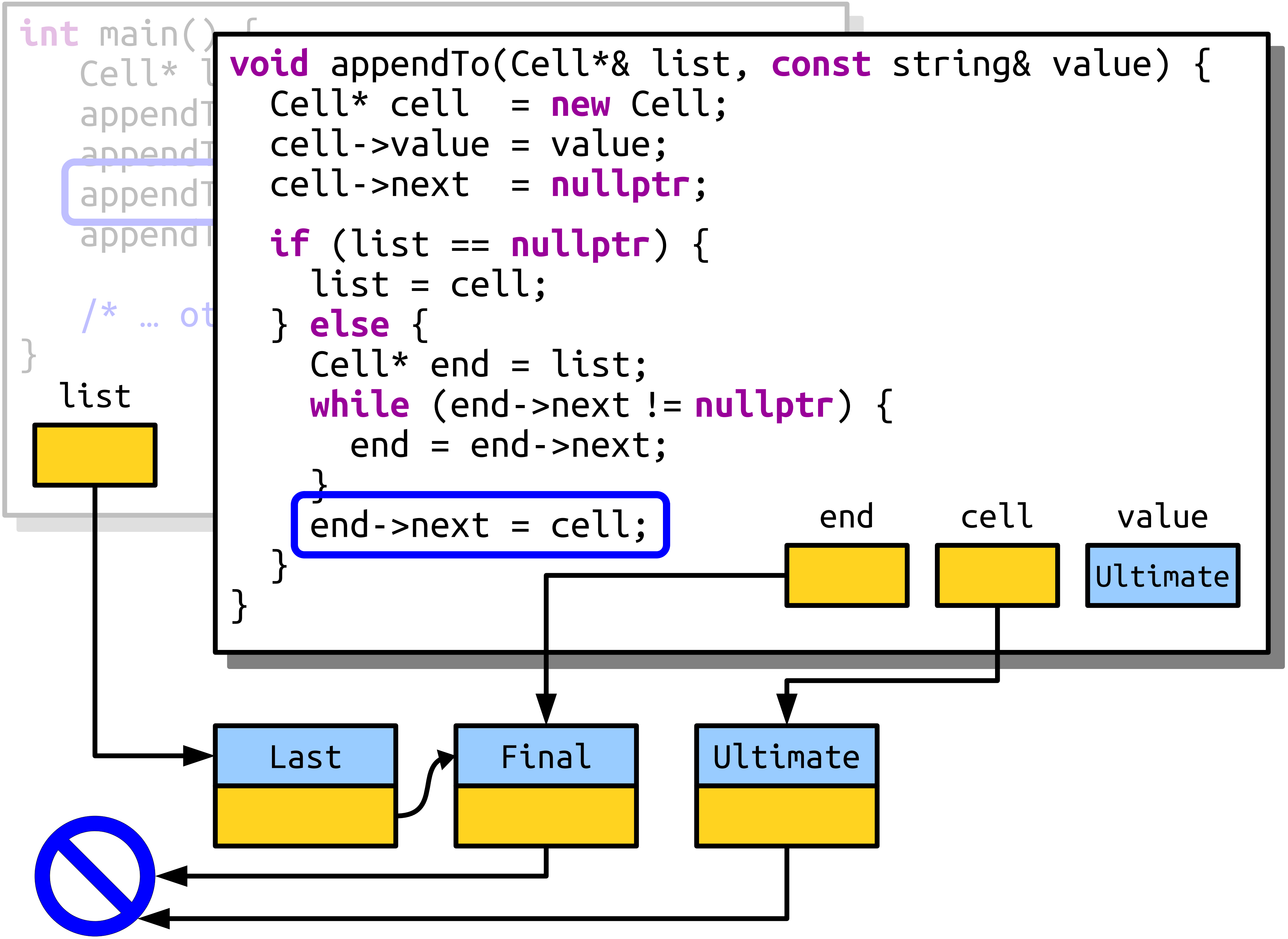
```
void appendTo(Cell*& list, const string& value) {
  Cell* cell  = new Cell;
  cell->value = value;
  cell->next  = nullptr;

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



```
int main() {
   Cell* l
   appendT
   appendT
   appendT
   appendT

   /* … ot
}
```

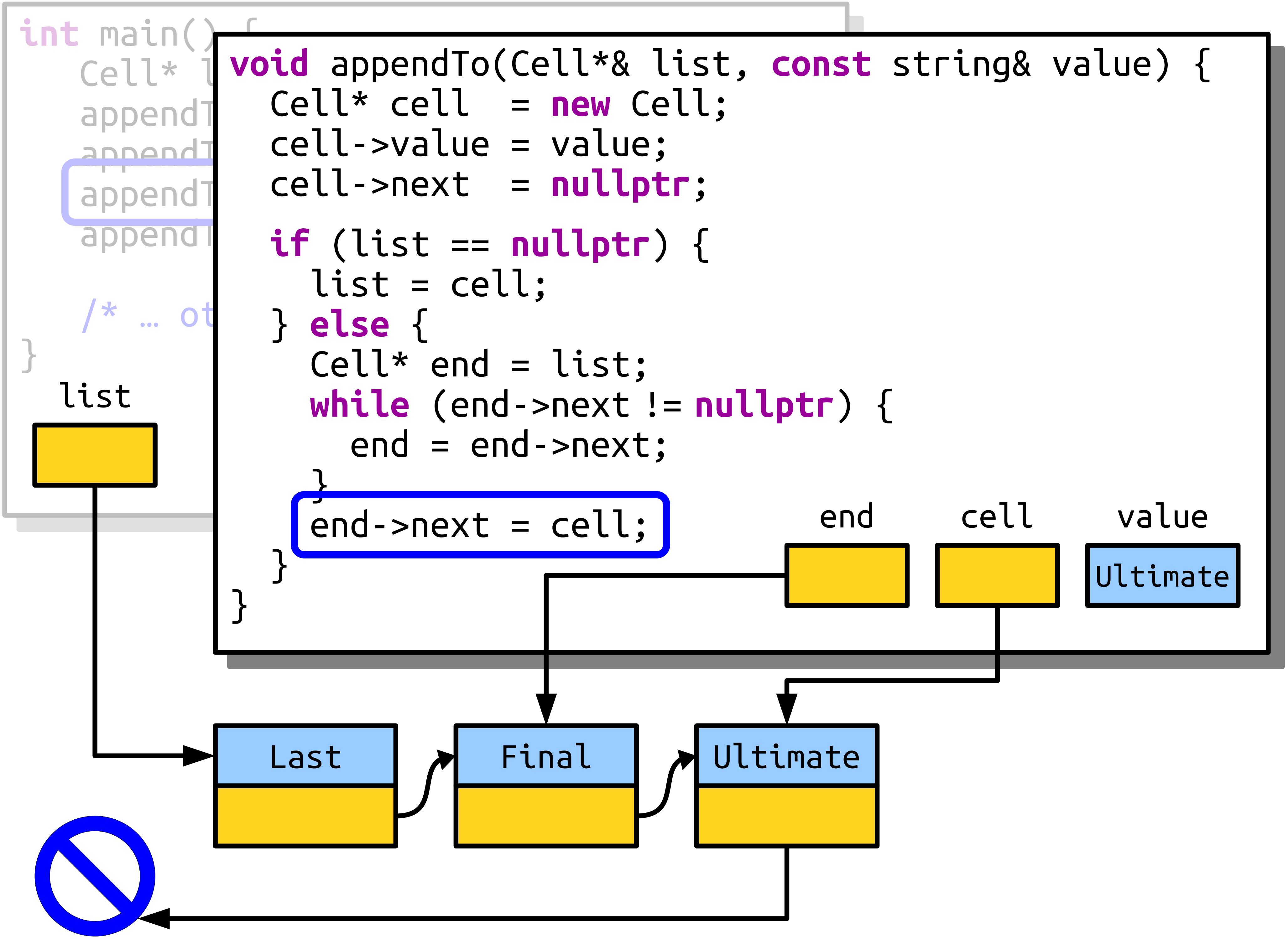
```
void appendTo(Cell*& list, const string& value) {
  Cell* cell  = new Cell;
  cell->value = value;
  cell->next  = nullptr;

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





```
int main() {
   Cell* l
   appendT
   appendT
   appendT
   appendT

   /* … ot
}
```

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

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

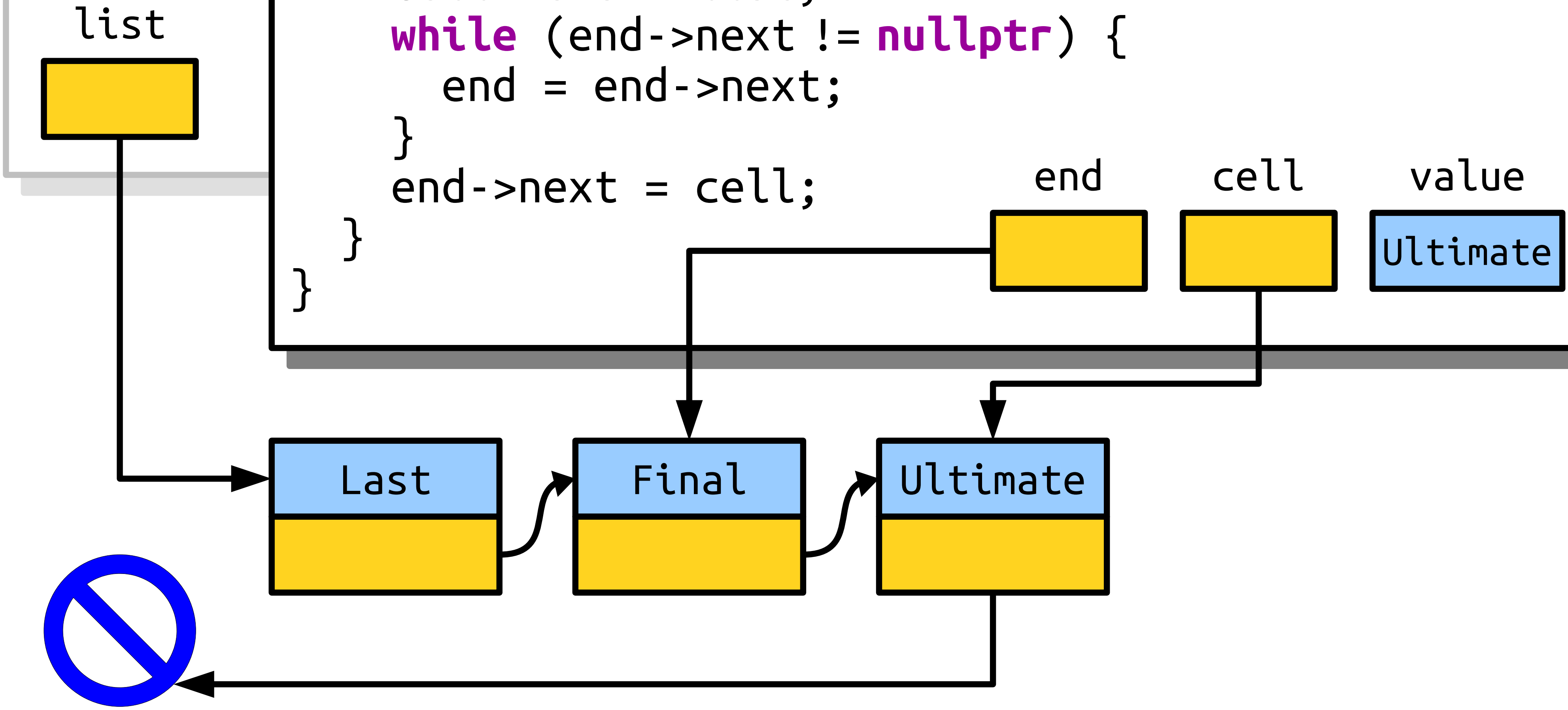


```
int main() {
   Cell* l
   appendT
   appendT
   appendT
   appendT

   /* … ot
}
```

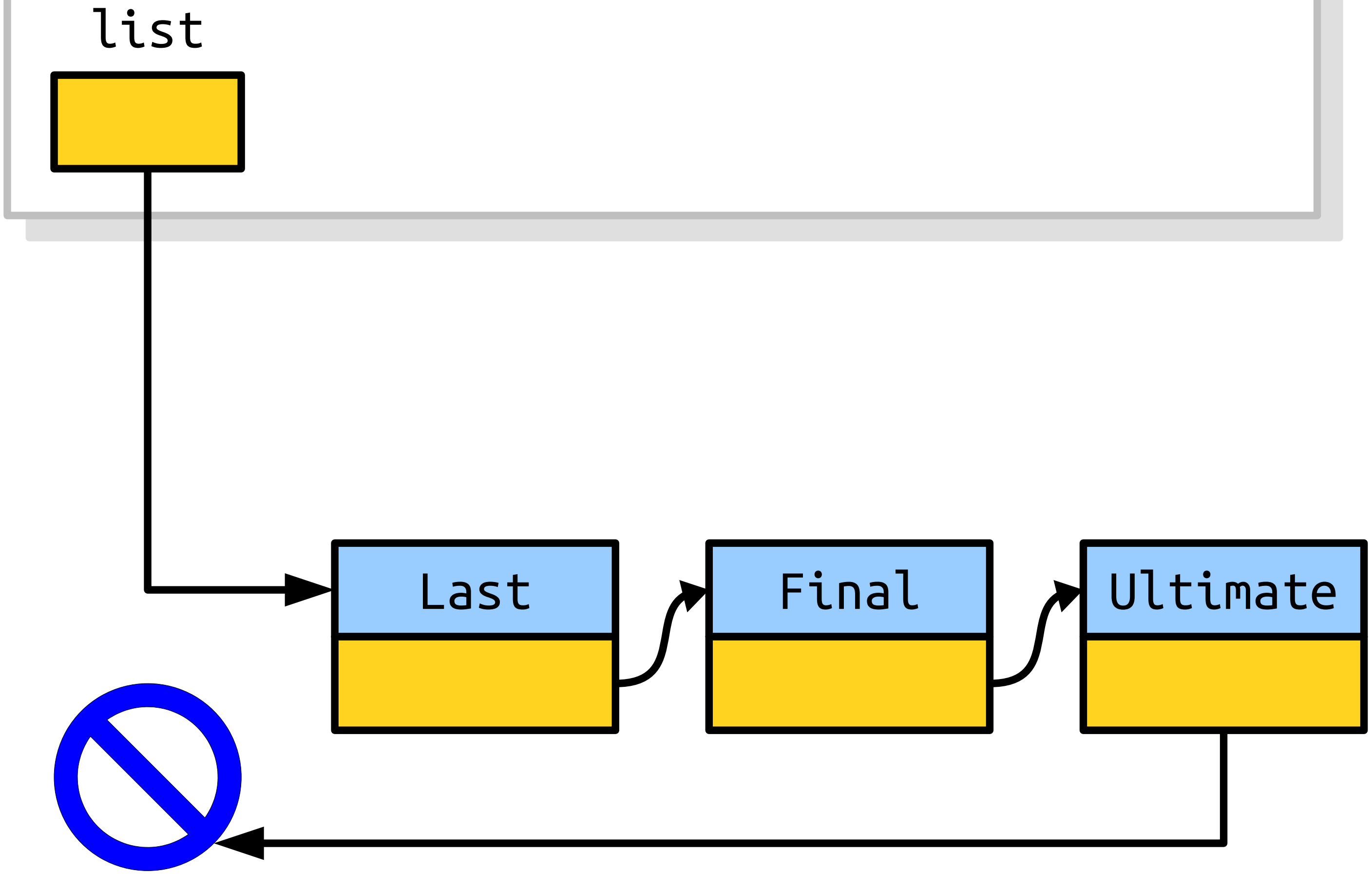
```
void appendTo(Cell*& list, const string& value) {
  Cell* cell  = new Cell;
  cell->value = value;
  cell->next  = nullptr;

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

```
int main() {
   Cell* l
   appendT
   appendT
   appendT
   appendT

   /* … ot
}
```

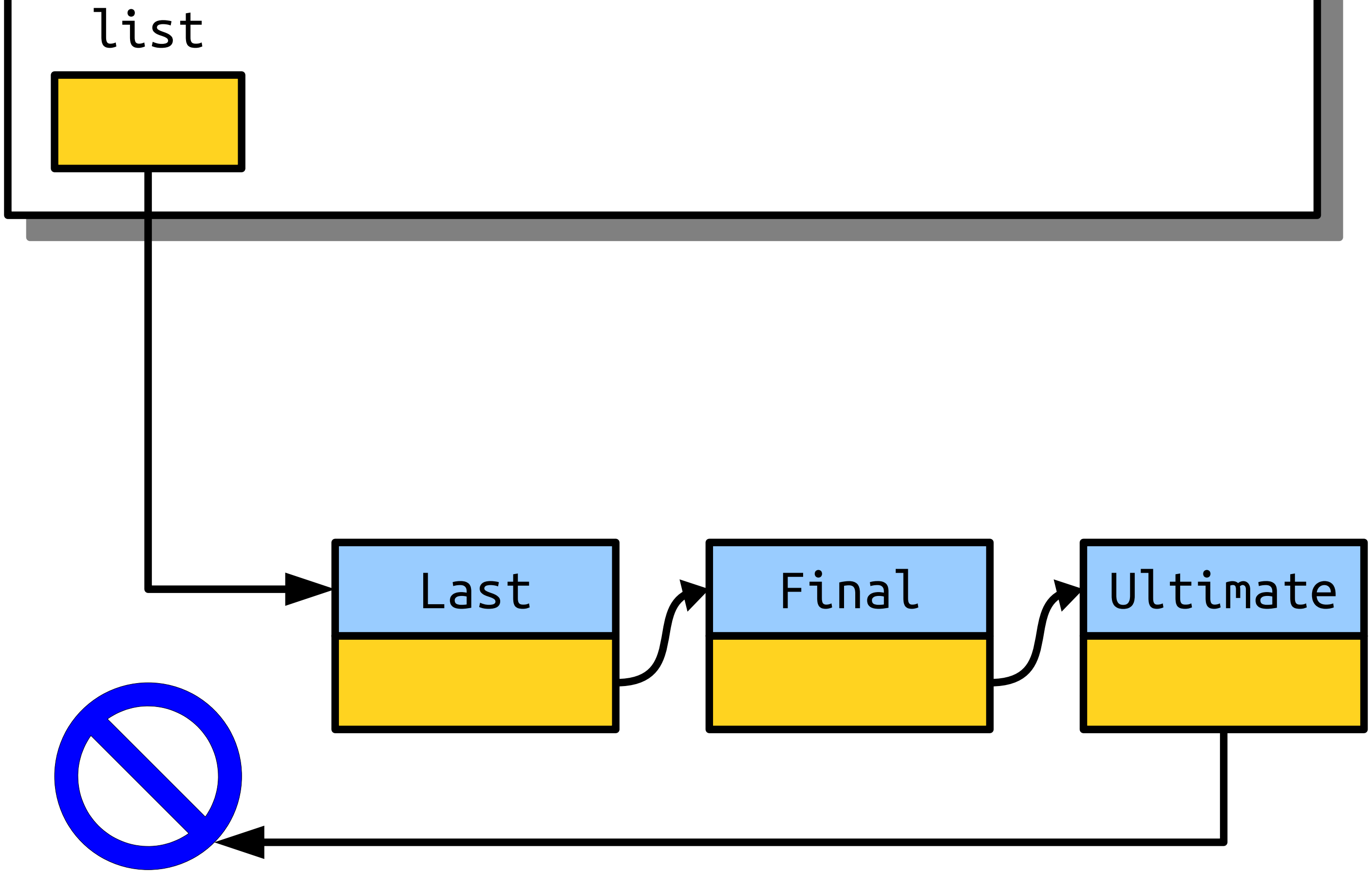
```
void appendTo(Cell*& list, const string& value) {
  Cell* cell  = new Cell;
  cell->value = value;
  cell->next  = nullptr;

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



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

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



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

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



What Went Wrong (Yet Again)?

What is the big-O runtime of this code when appending to a list of length n?

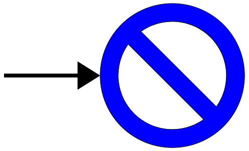
Formulate a hypothesis, but ***don’t post it in chat yet***.

What is the big-O runtime of this code when appending to a list of length *n*?

Now, ***post your best guess in chat***. Not sure? Just answer “??”

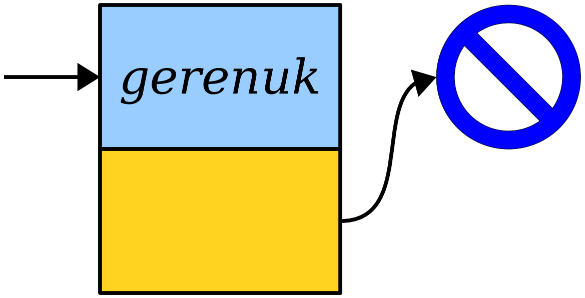
Appending to a List

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



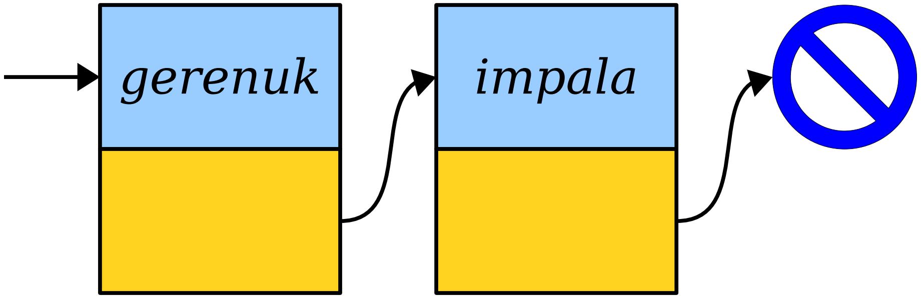
Appending to a List

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



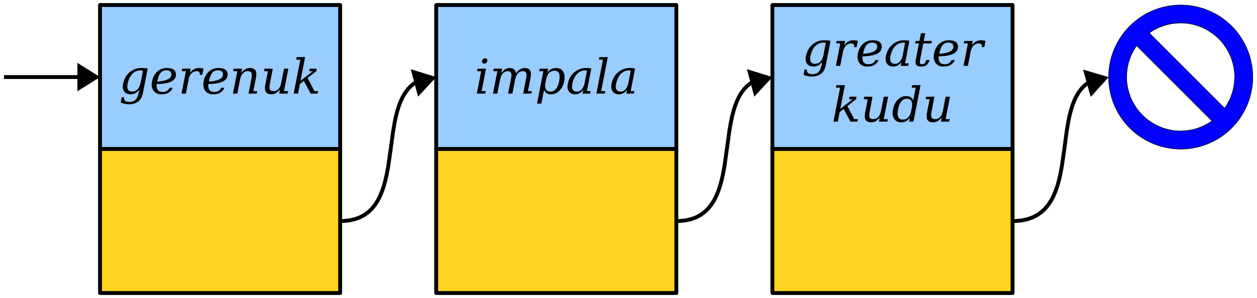
Appending to a List

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



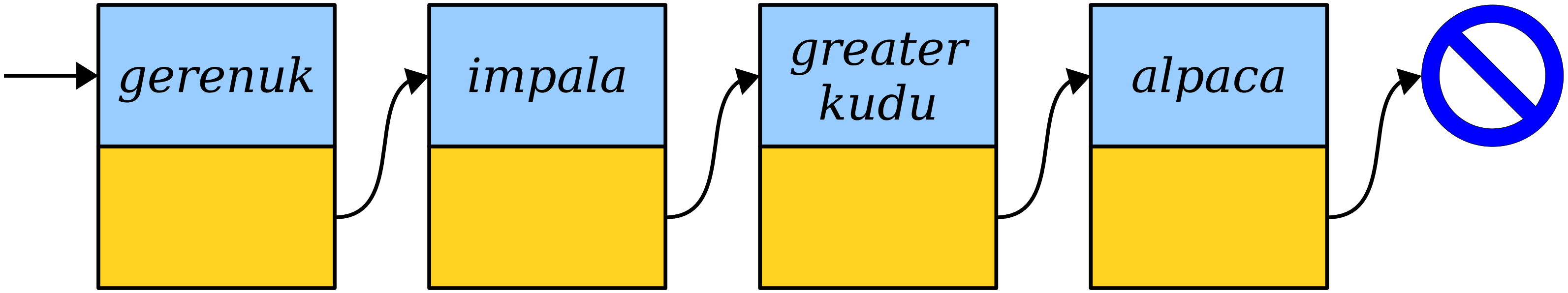
Appending to a List

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



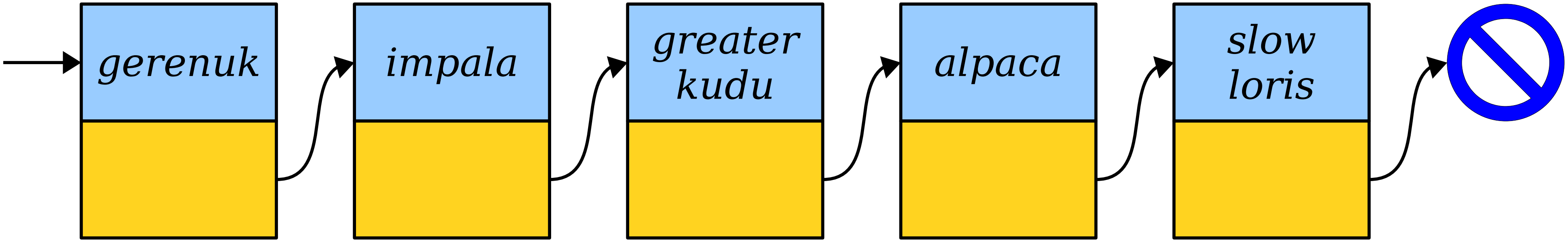
Appending to a List

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



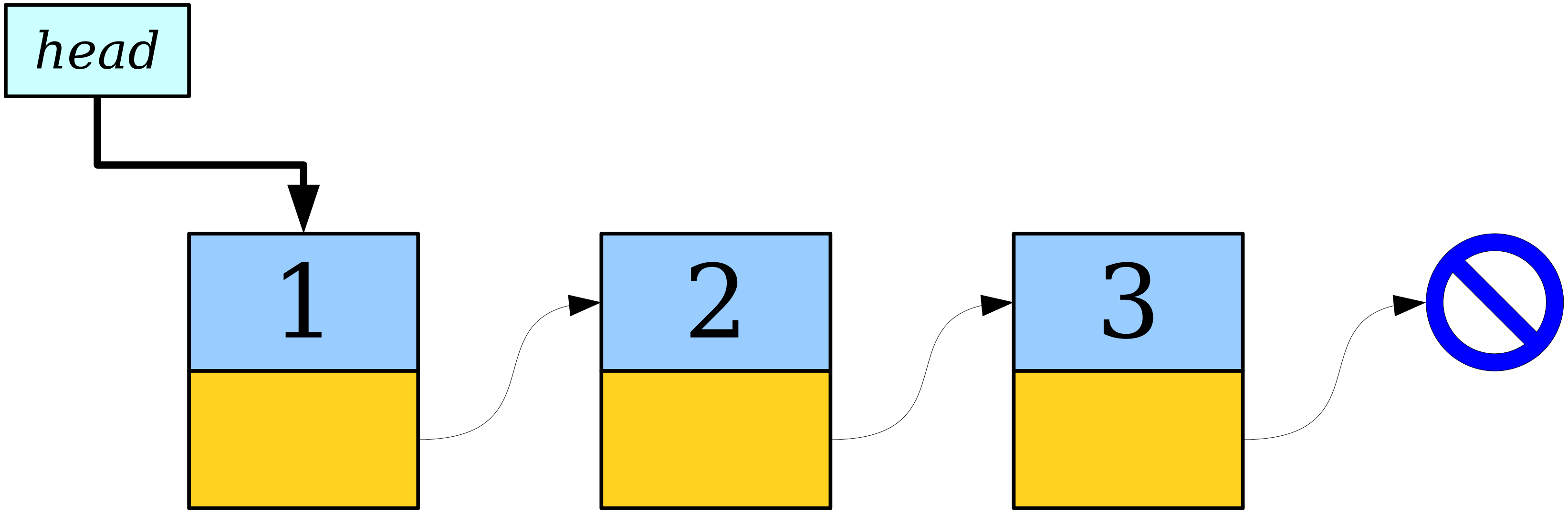
Appending to a List

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



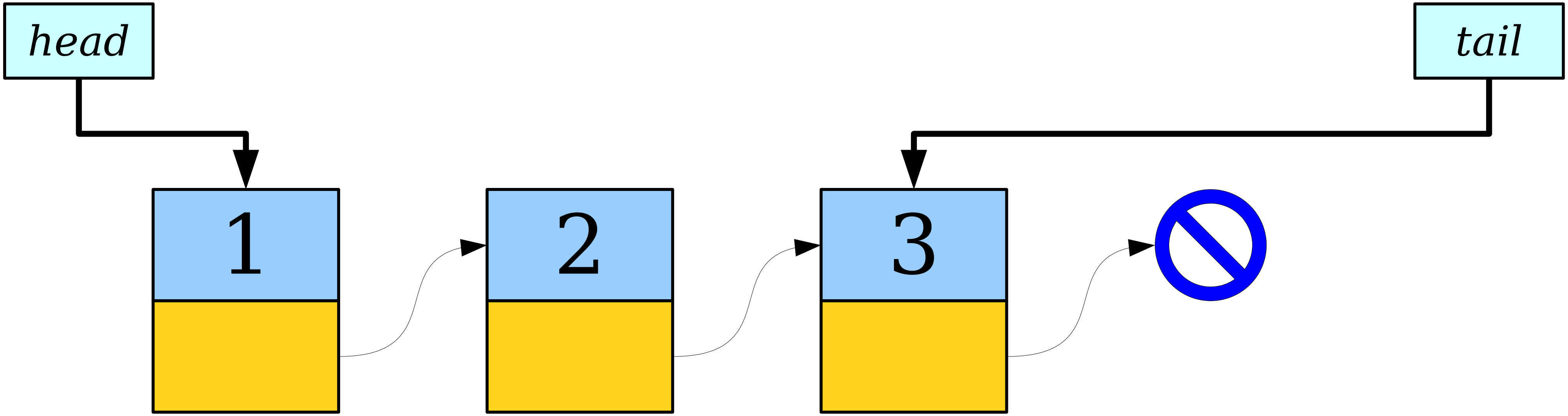
Tail Pointers

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



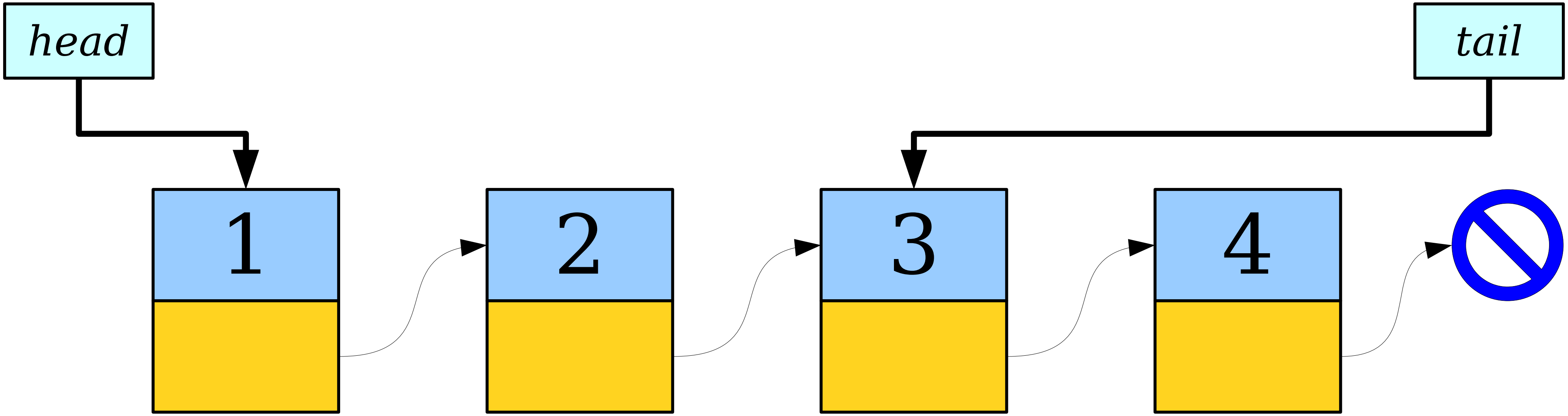
Tail Pointers

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



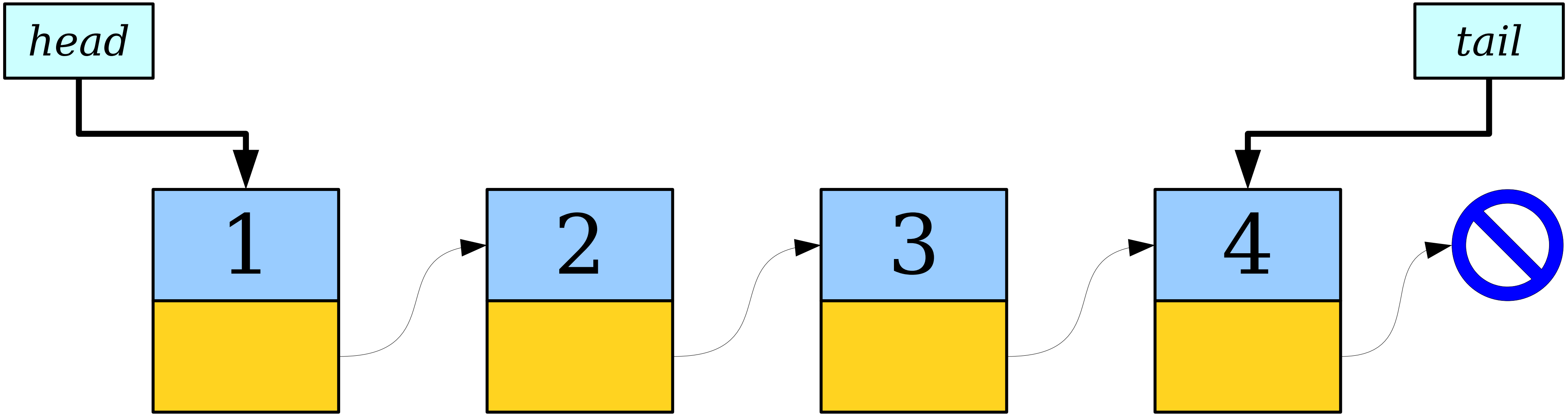
Tail Pointers

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



Tail Pointers

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

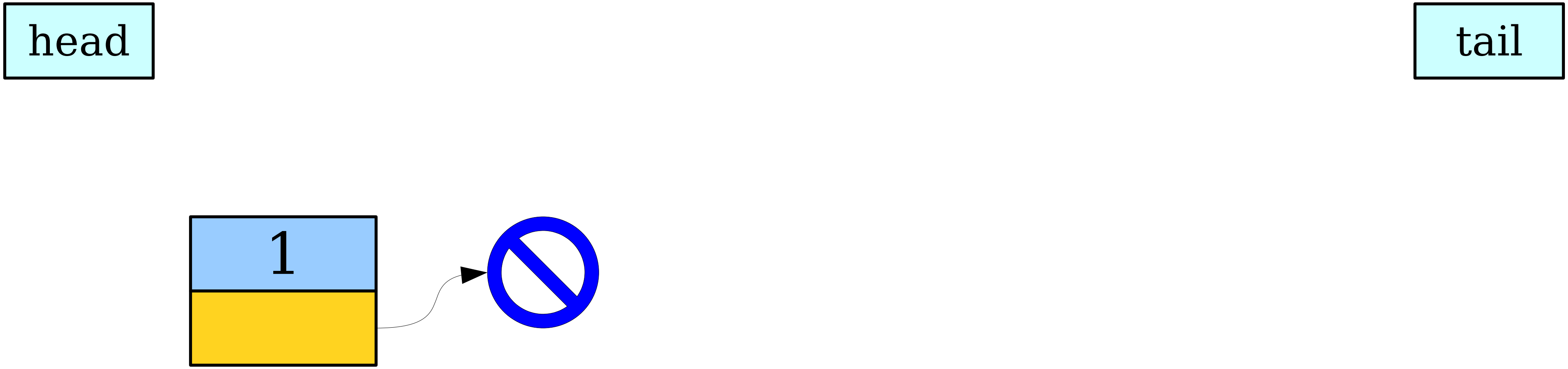


Appending Things Quickly

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

Appending Things Quickly

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



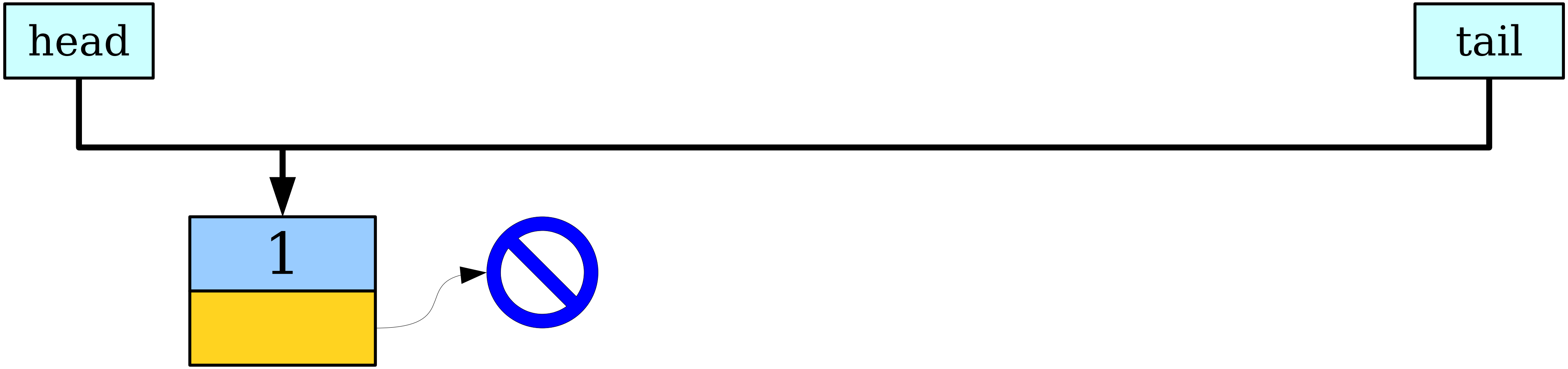
head

tail

1

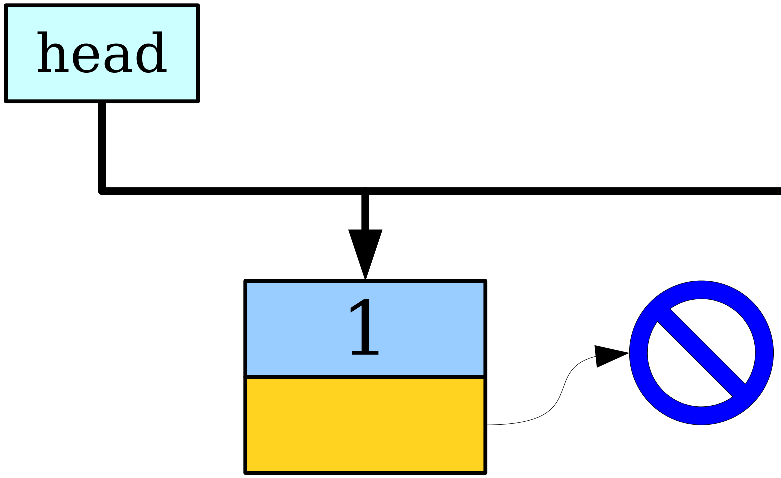
Appending Things Quickly

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

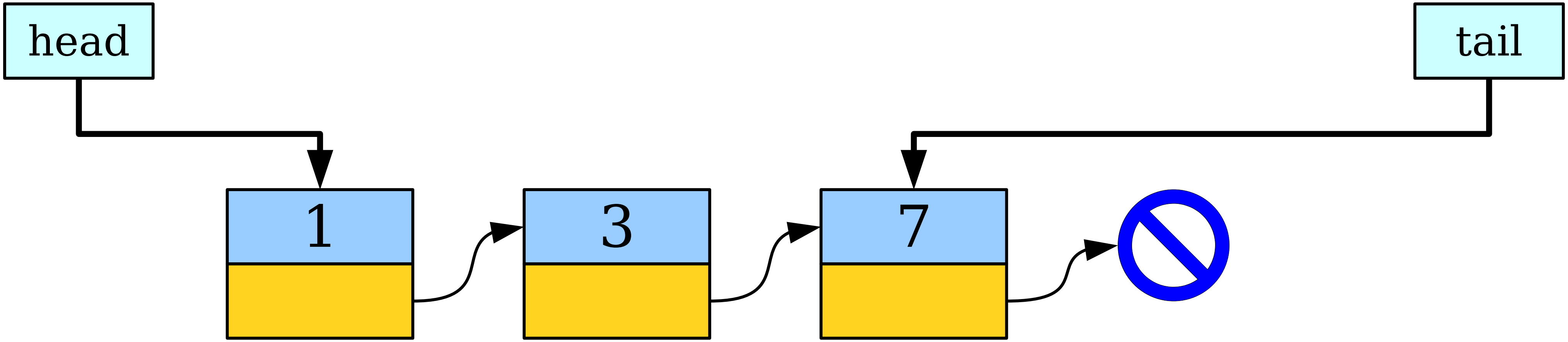


Appending Things Quickly

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

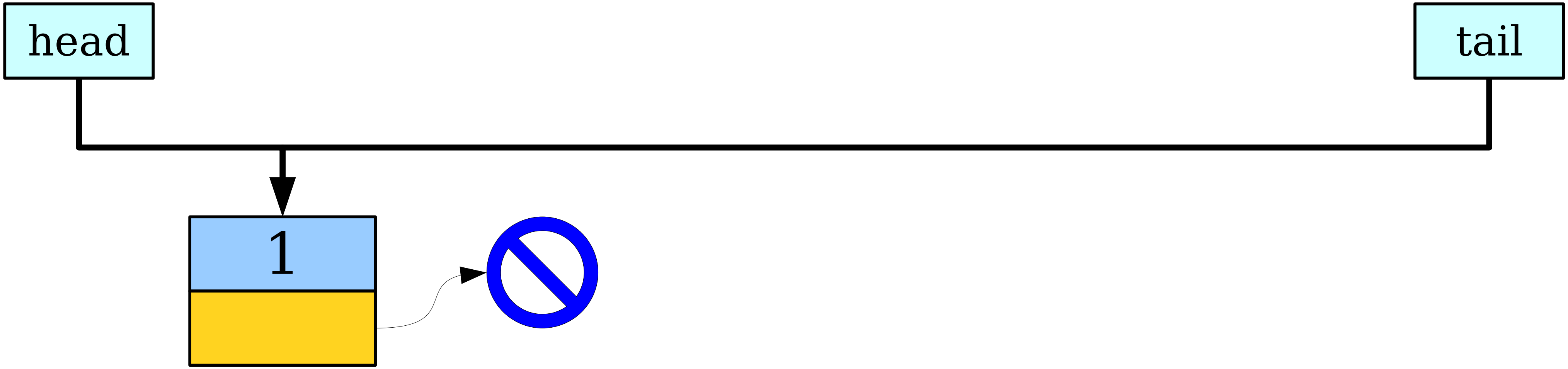


- ***Case 2:*** The list is not empty.

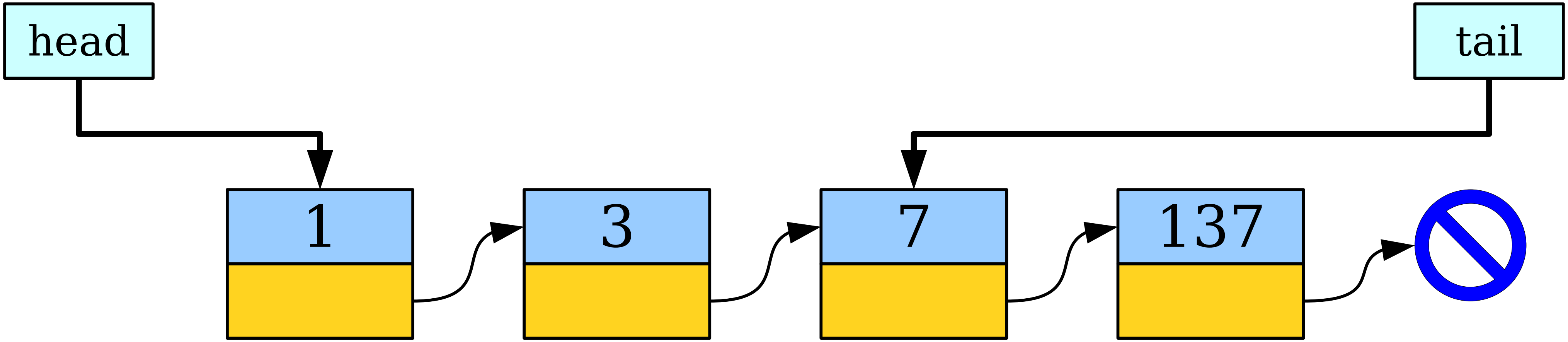


Appending Things Quickly

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

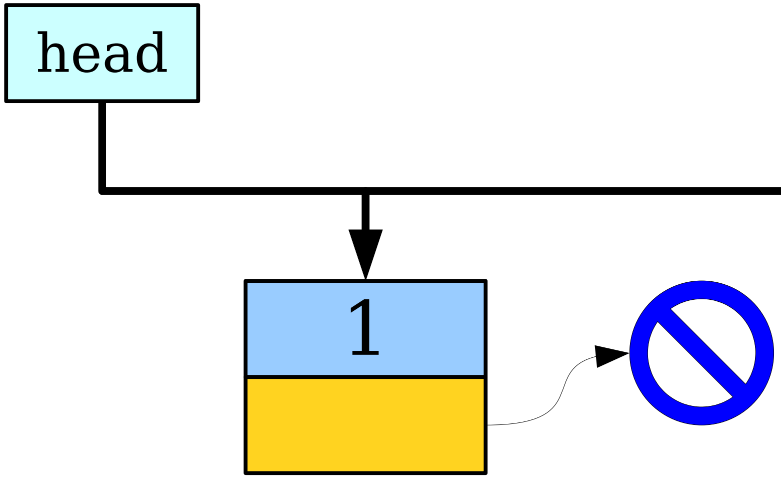


- ***Case 2:*** The list is not empty.

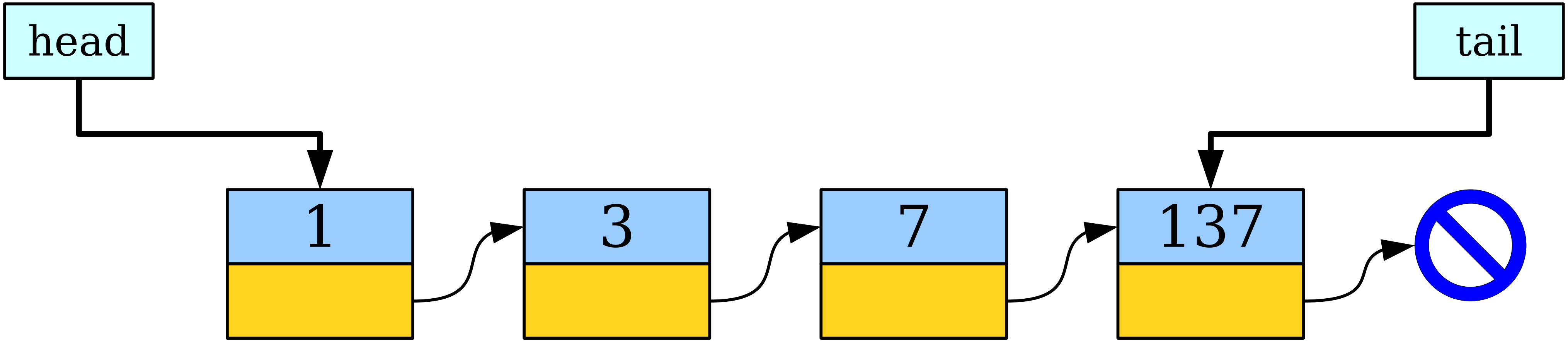


Appending Things Quickly

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



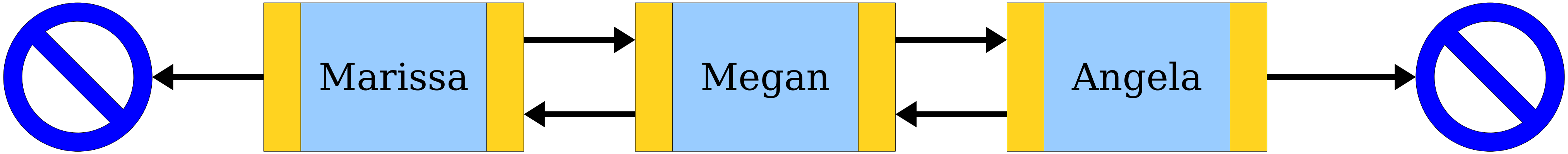
- ***Case 2:*** The list is not empty.



Coda: Doubly-Linked Lists

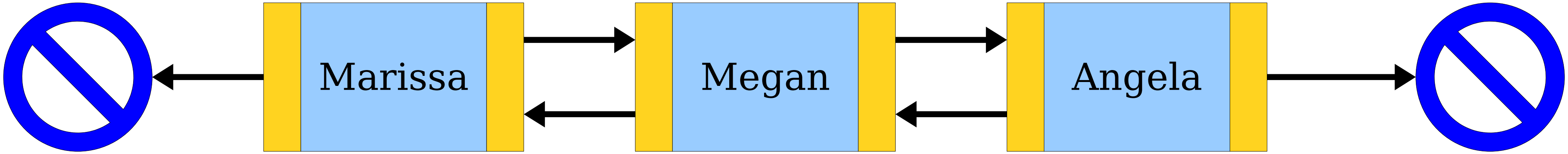
Doubly-Linked Lists

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



Doubly-Linked Lists

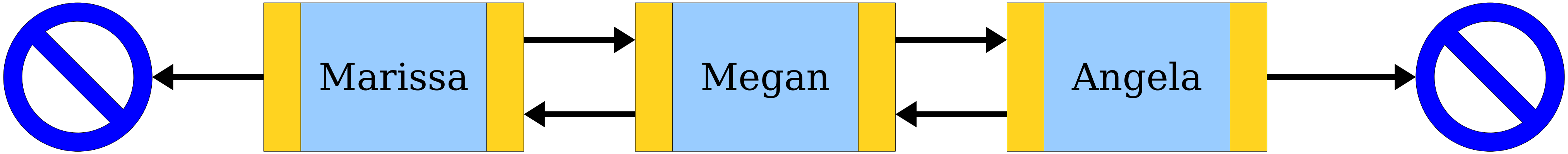
- In many cases, doubly-linked lists are similar to singly-linked lists.
- For example, if you're just moving from the left to the right, then code on doubly-linked lists looks really similar to code on singly-linked lists.



Doubly-Linked Lists

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

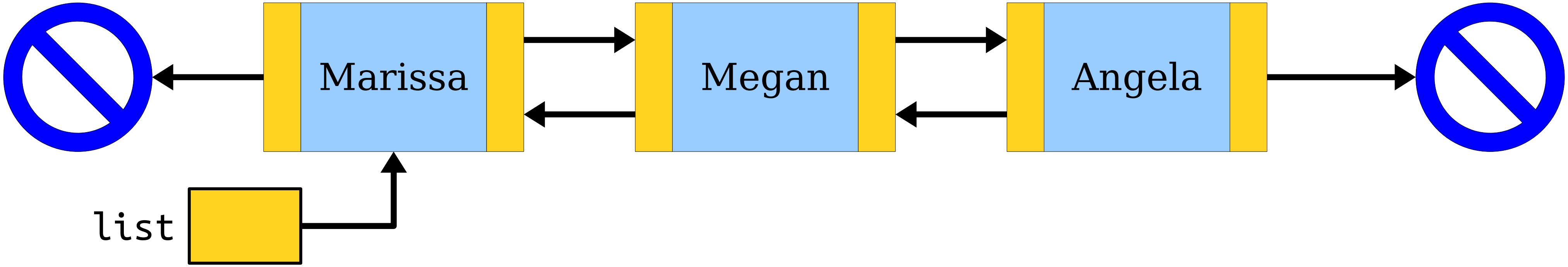
```
Cell* list = /* first cell */;
```



Doubly-Linked Lists

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

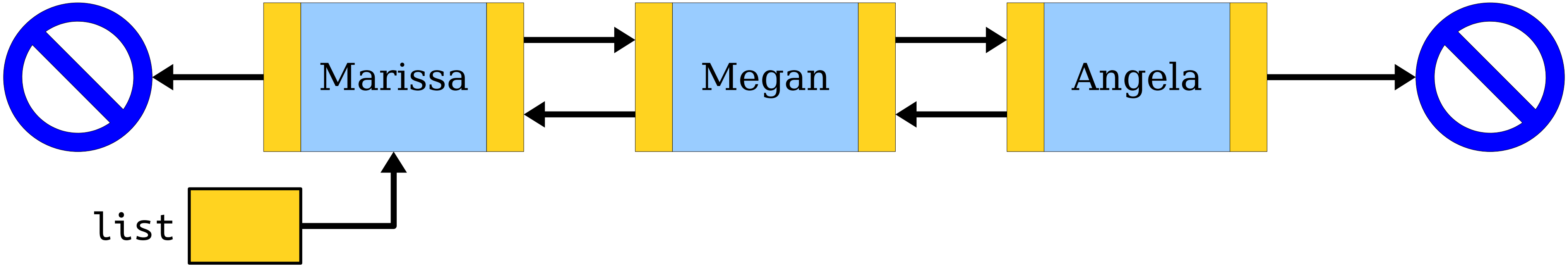
```
Cell* list = /* first cell */;
```



Doubly-Linked Lists

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

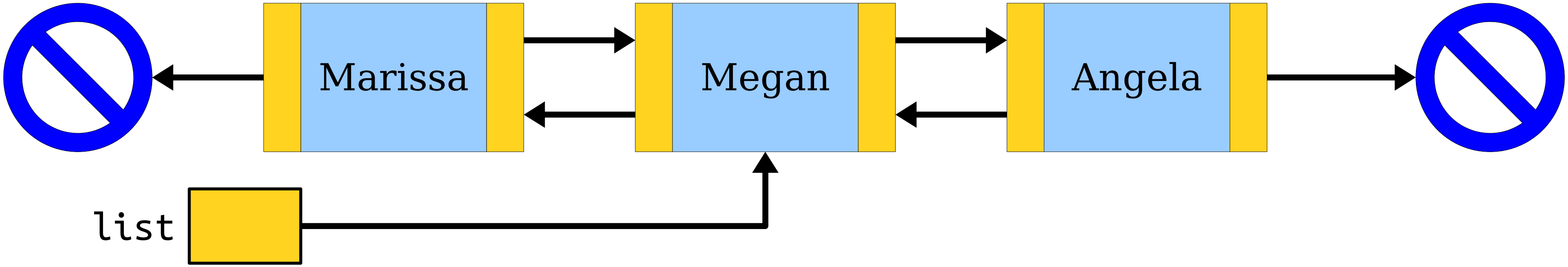
```
Cell* list = /* first cell */;
list = list->next;
```



Doubly-Linked Lists

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

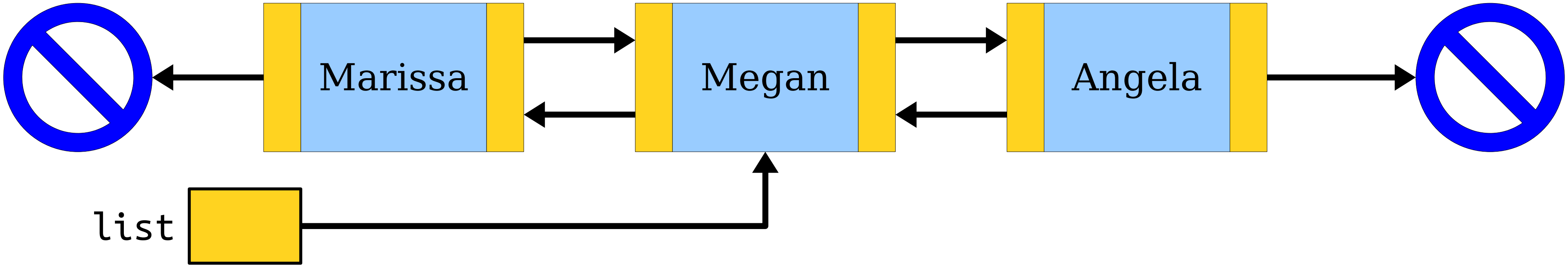
```
Cell* list = /* first cell */;
list = list->next;
```



Doubly-Linked Lists

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

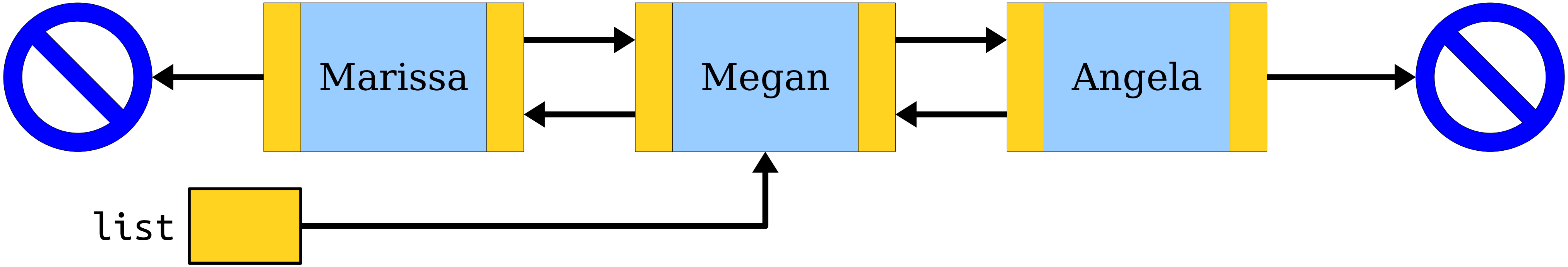
```
Cell* list = /* first cell */;
list = list->next;
```



Doubly-Linked Lists

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

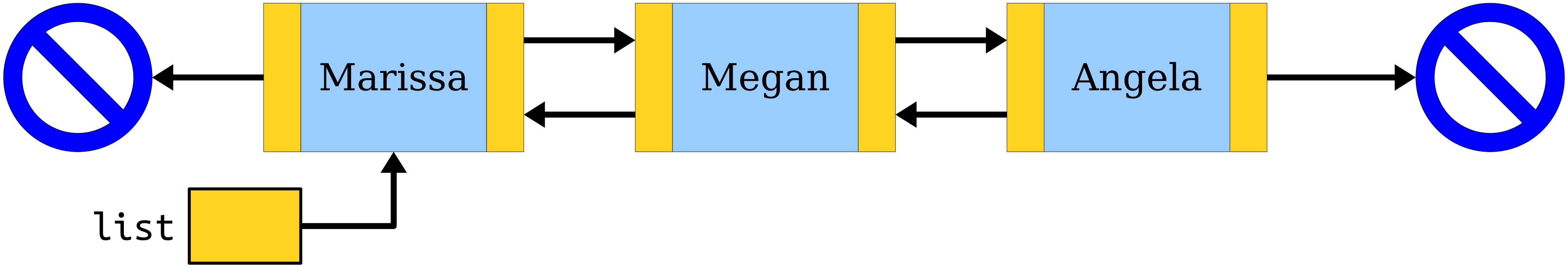
```
Cell* list = /* first cell */;
list = list->next;
list = list->prev;
```



Doubly-Linked Lists

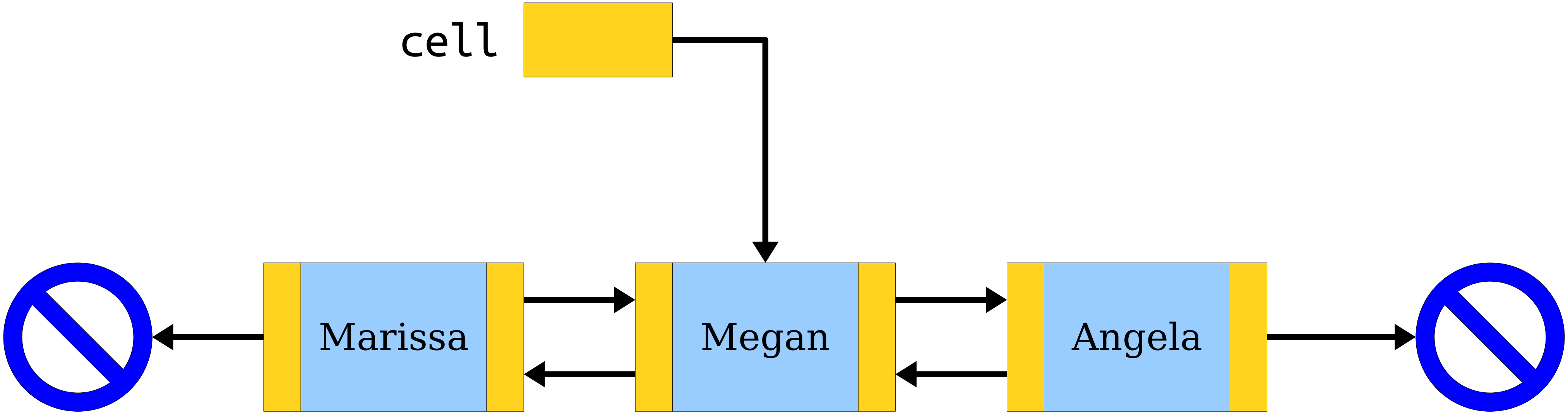
- We can also move backwards in a doubly-linked list.
- Many algorithms are a lot easier to write if you can do this!

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



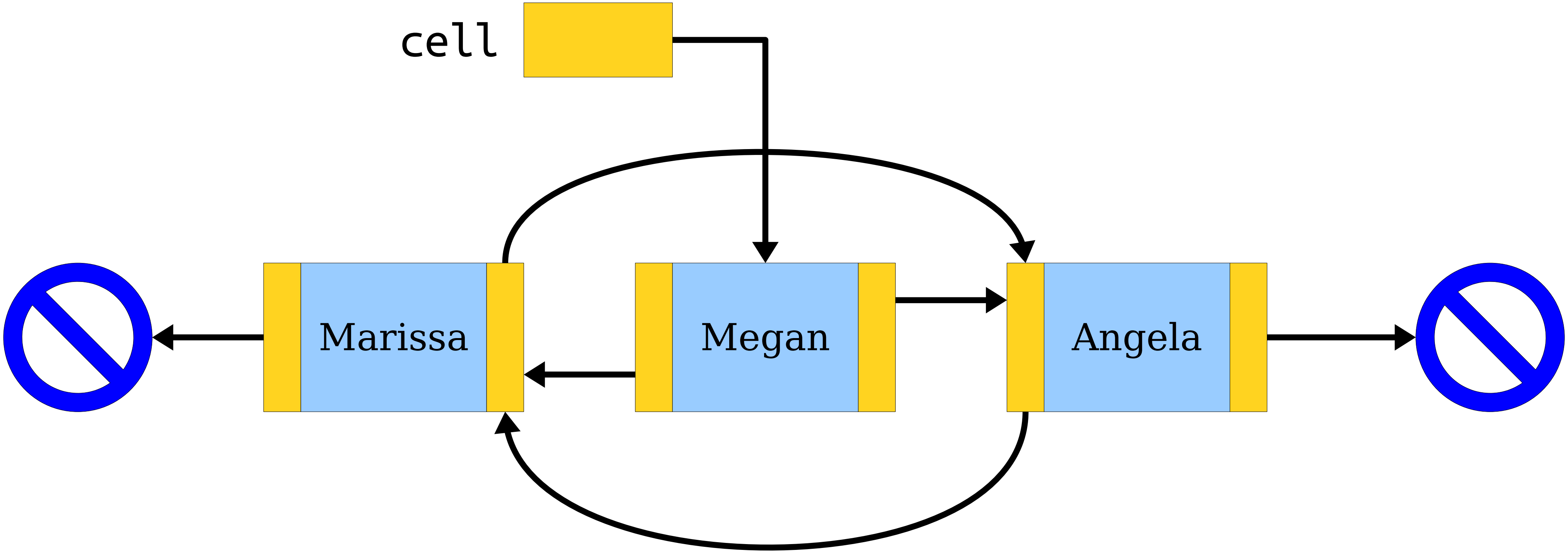
Doubly-Linked Lists

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



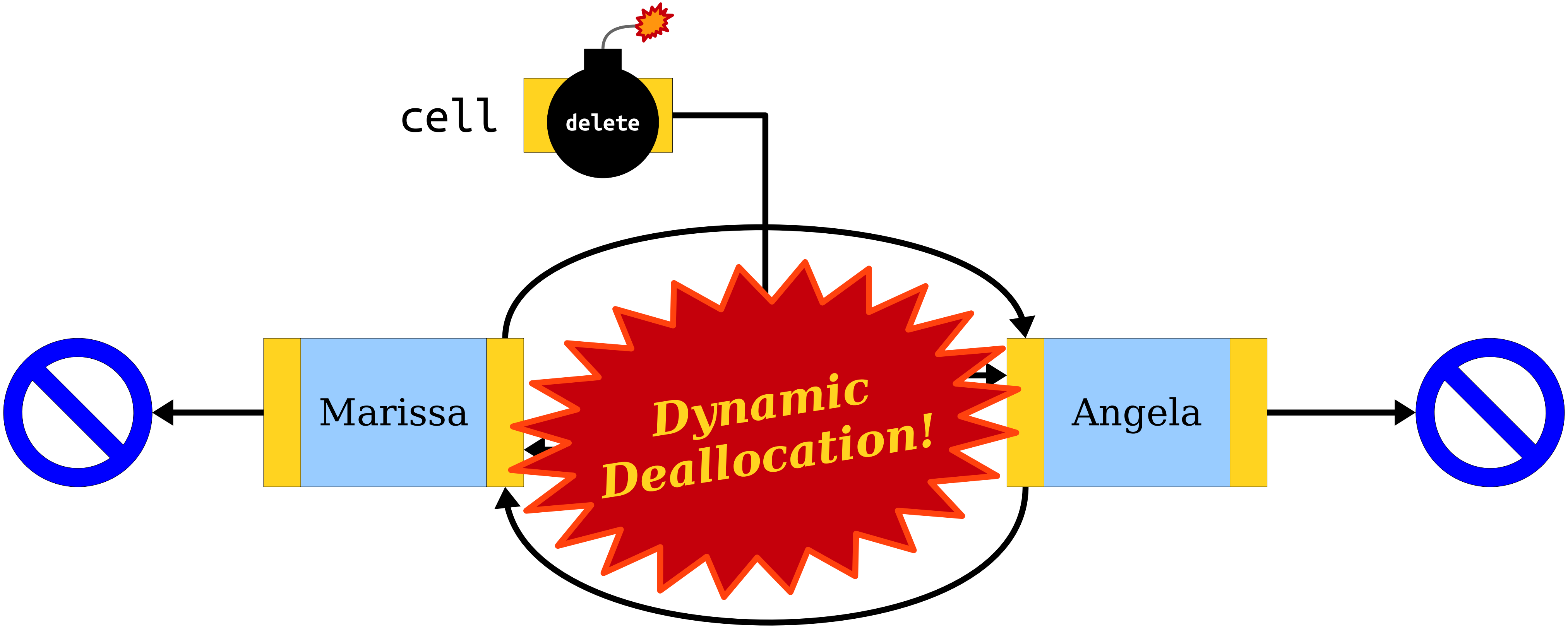
Doubly-Linked Lists

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



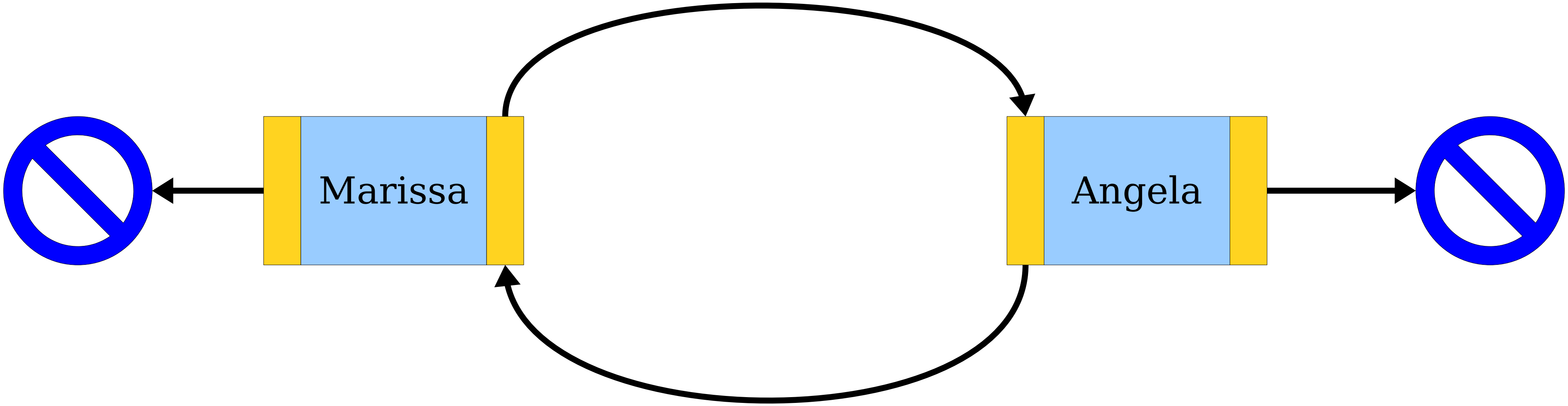
Doubly-Linked Lists

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



Doubly-Linked Lists

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



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

To Recap

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

Your Action Items

- ***Read Chapter 13.***
  - It’s all about different representations for data and the relative tradeoffs. And there’s some great coverage of linked lists in there!
- ***Finish Assignment 7.***
  - Swing by the LaIR, post on EdStem, visit our office hours, or email your SL if you need an help!

Next Time

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