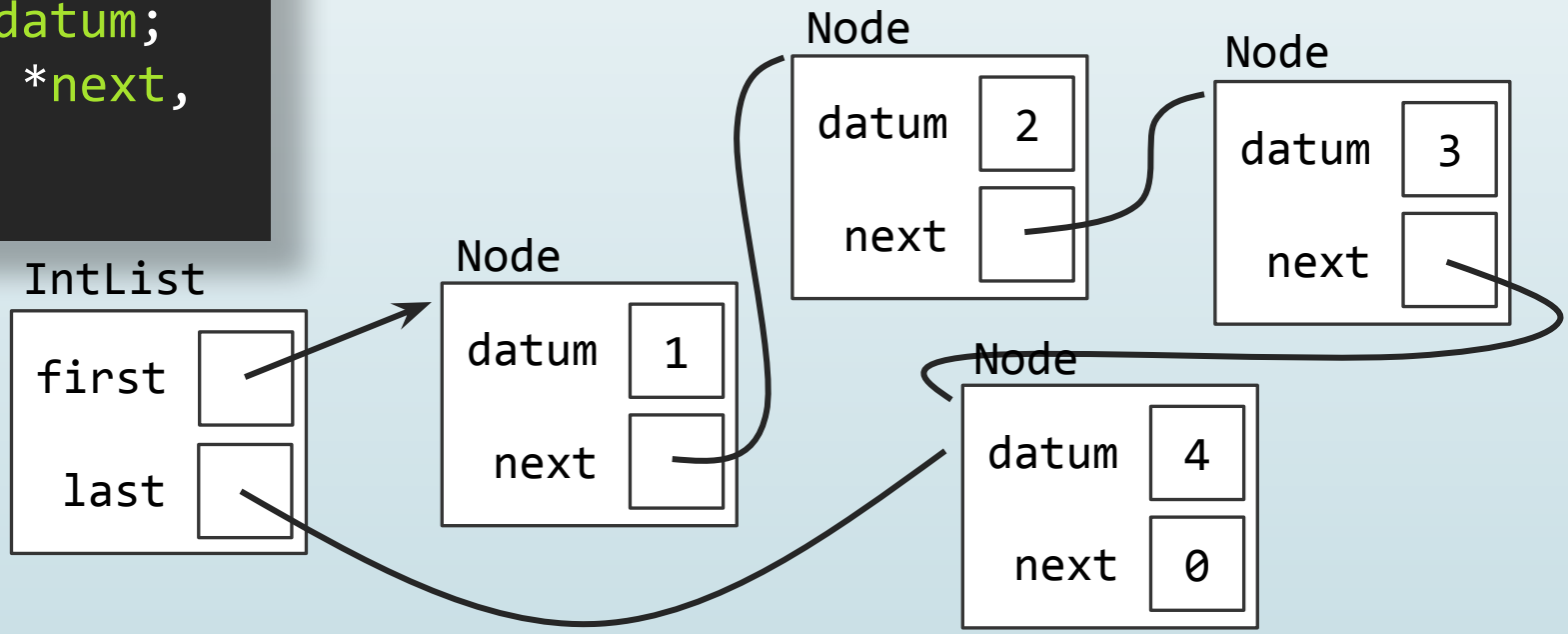
Реализация pop_back

- Что если хотим добавить элемент в конец списка?
  - Нужно пройти все элементы начиная с первого!

Может изменим представление данных…

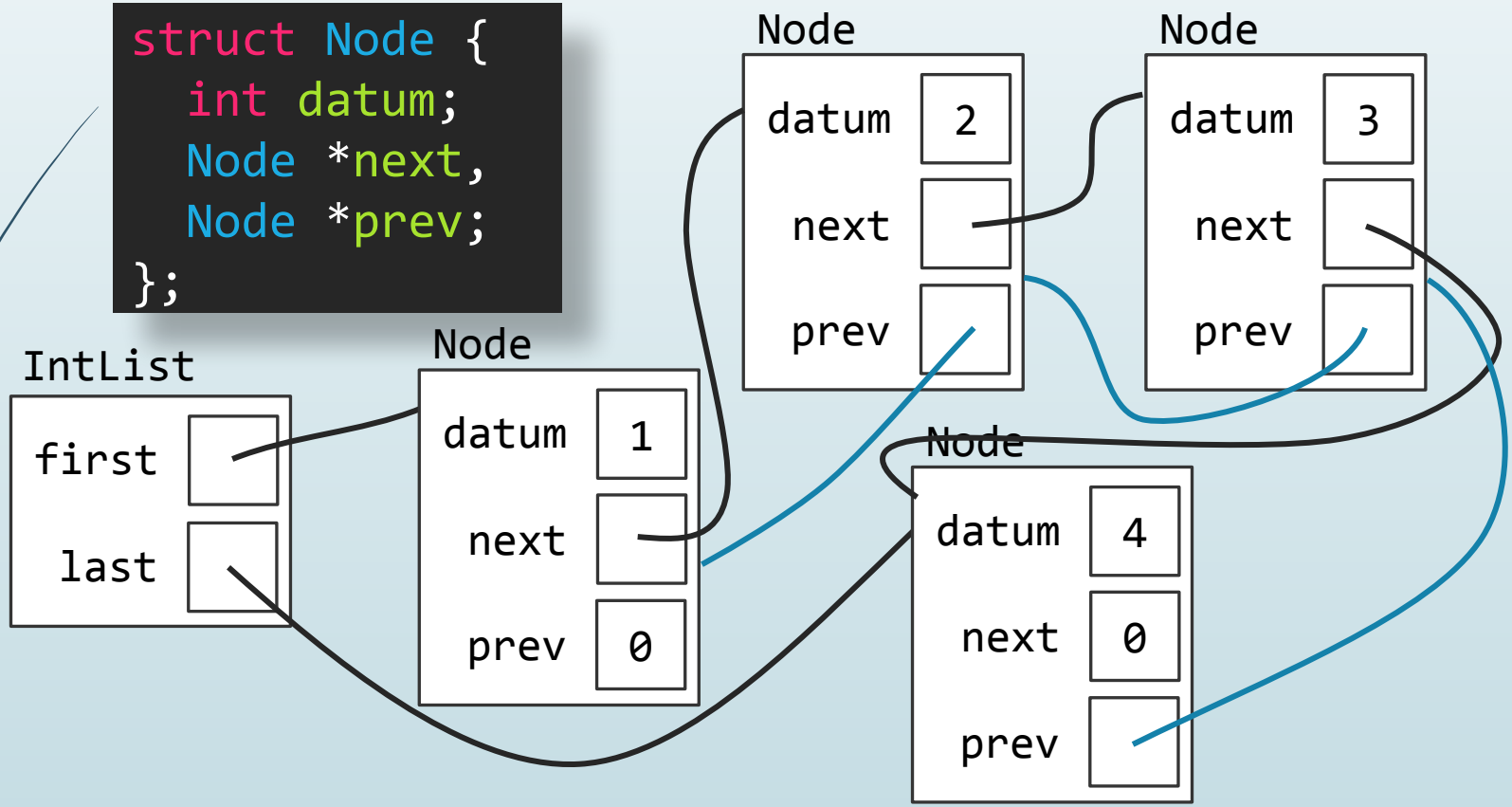
```
struct Node {
  int datum;
  Node *next,
};
```



Реализация pop_back

- Что если хотим добавить элемент в конец списка?
  - Нужно пройти все элементы начиная с первого!
  - Может изменим представление данных…

```
struct Node {
  int datum;
  Node *next,
  Node *prev;
};
```



Linked List Шаблон

List.h

```
template <typename T>
class List {
public:
  void push_front(T v);
  T & front();
private:
  struct Node {
    T datum;
    Node *next;
  };
  Node *first;
};
```

```
#include "List.h"
int main() {
  List<int> list1;
  List<Duck> list2;
}
```

Компилятор сам подставляет нужный тип данных.

```
class List<int> {
public:
  void push_front(int v);
  int & front();
private:
  struct Node {
    int datum;
    Node *next;
  };
  Node *first;
};
```

```
class List<Duck> {
public:
  void push_front(Duck v);
  Duck & front();
private:
  struct Node {
    Duck datum;
    Node *next;
  };
  Node *first;
};
```

Итераторы списка List

- Что если мы хотим пройти по элементам списка…

```
int main() {
  Llama l1("Paul");
  Llama l2("Carl");

  List<Llama> llamas;
  llamas.push_back(l1);
  llamas.push_back(l2);

  for (...) {
    // хотим накормить всех лам в списке
  }
}
```

Итераторы

- Есть один способ…

```
int main() {
  List<int> list;
  int arr[3] = { 1, 2, 3 };
  fillFromArray(list, arr, 3);

  for (List<int>::Node *np = list.first; np; np = np->next) {
    cout << np->datum << endl; // вывод элементов
  }
}
```

- Проблема:
  - Придется выносить в public поля класса

Обход по указателю

```
int const SIZE = 5;
int arr[SIZE] = { 1, 2, 3, 4, 5 };
```

- Обход по указателю
  - Когда хотим обратиться к элементу разыменовываем указатель

***end - на самом деле указатель на следующий после последнего**

Продолжаем пока не дойдем до конца.

```
int *end = arr + SIZE;
for (int *ptr = arr; ptr != end; ++ptr) {

    cout << *ptr << endl;
}
```

Указатель на первый элемент

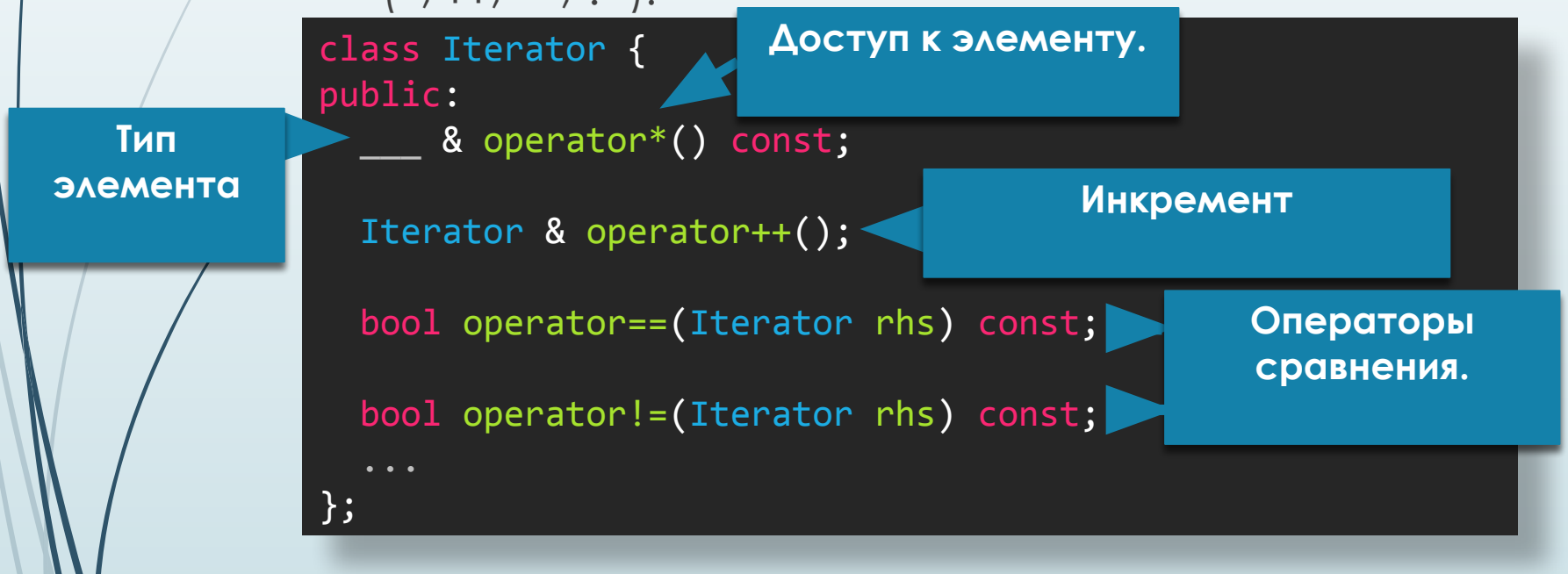
Разыменовываем указатель

Интерфейс итератора

- Итераторы обеспечивают интерфейс для обхода последовательностей элементов.
  - Итератор на элемент в контейнере можно итерировать для доступа к следующему элементу
- Итераторы должны поддерживать следующие операции:
  - Разыменование - доступ к элементу.
    `*it`
  - Инкремент – переход к следующему элементу.
    `++it`
  - Сравнение – проверка итераторов на эквивалентость.
    `it1 == it2`
    `it1 != it2`

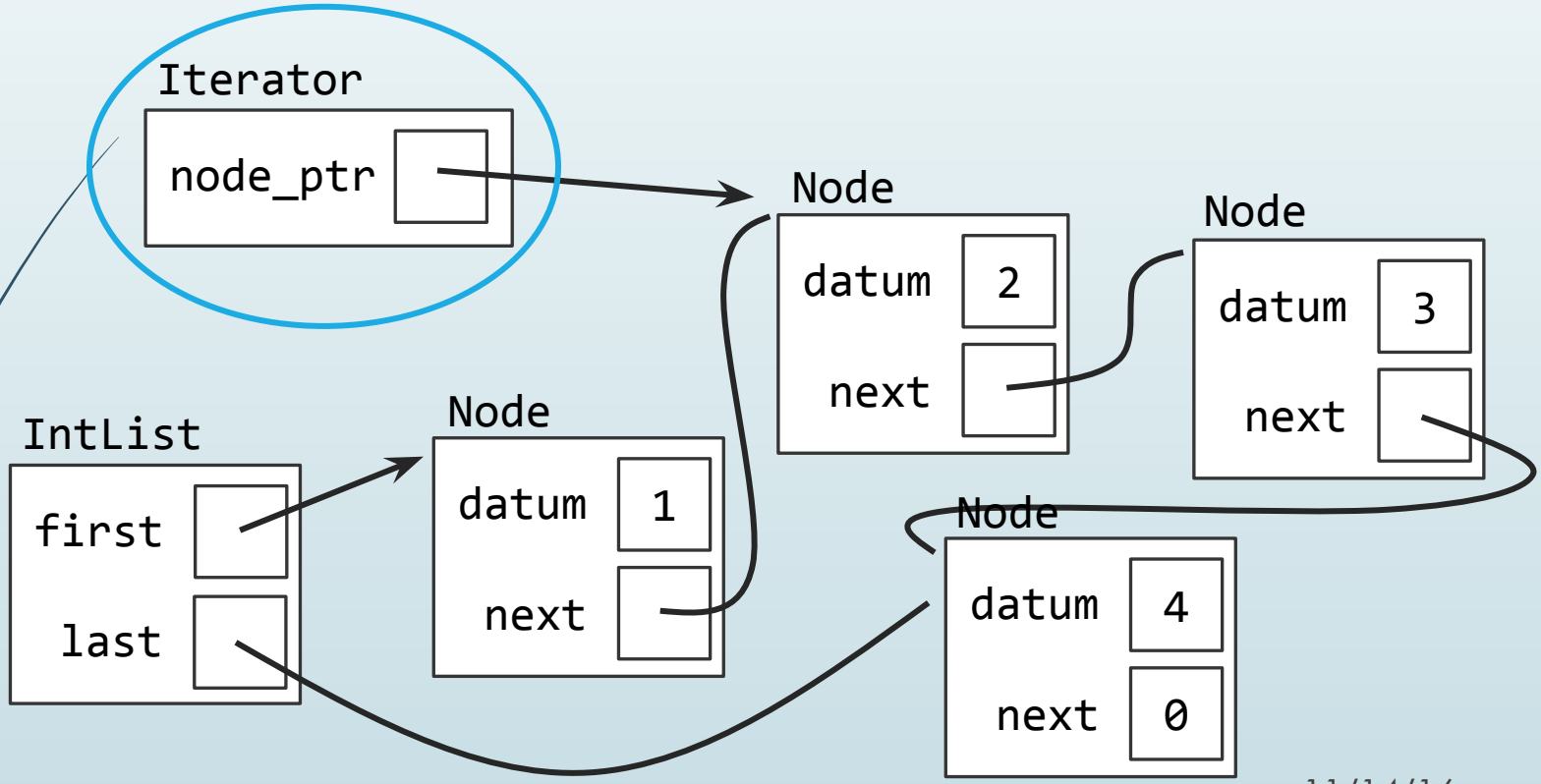
Что такое итератор?

- Итератор это объект, который "работает как указатель".
- Может быть реализован с помощью класса, который перегружает соответствующие операторы
- (*, ++, ==, !=).



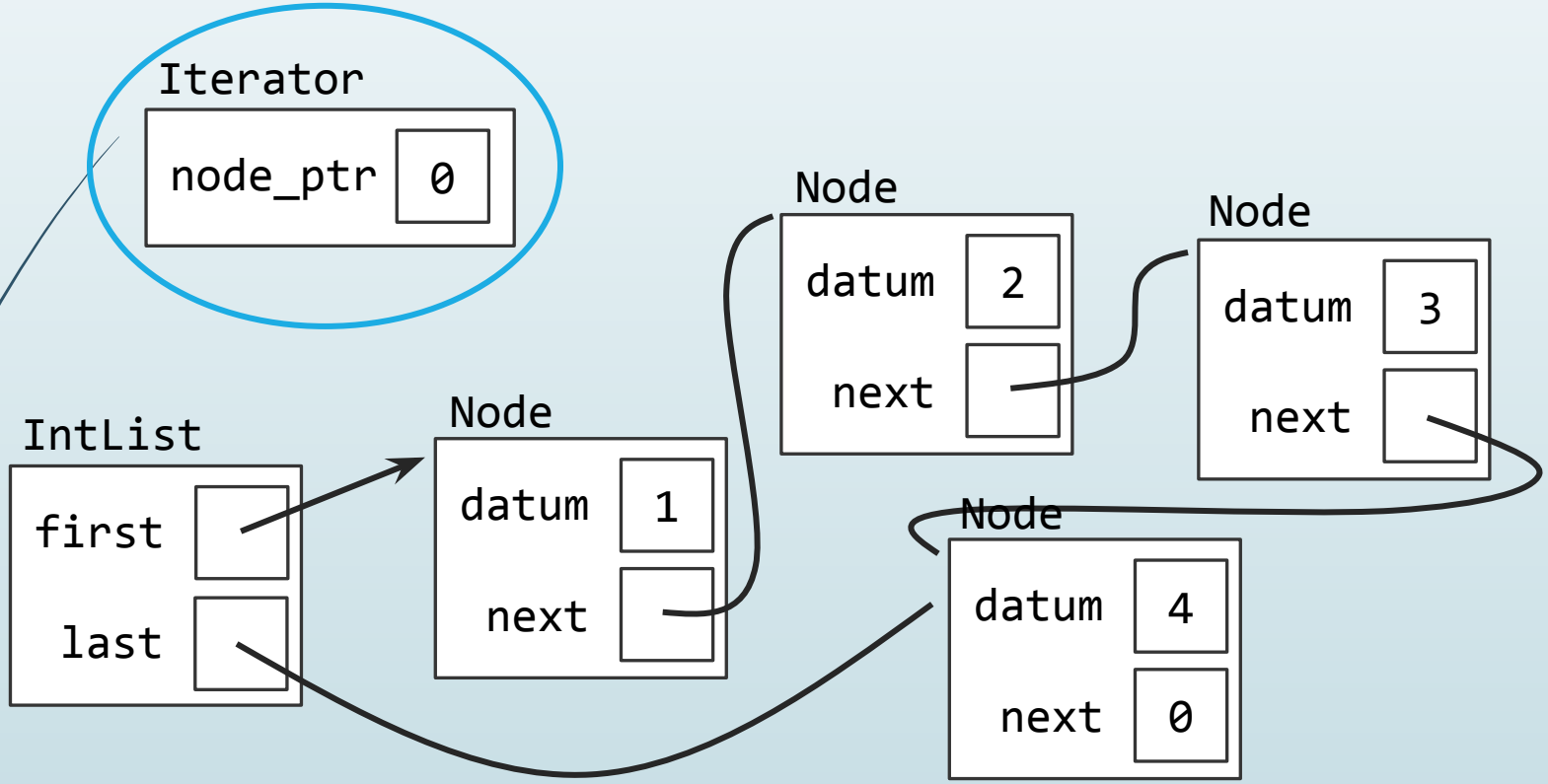
List Iterator: Представление данных

- Доступ через поле datum.
- Переход к следующему через next указатель.



List Iterator: Представление данных

- Как обозначить конец итератор?
  - "Следующий после последнего"
  - Использовать указатель на ноль.



Реализация `List` Iterator

- Представление данных
  - Храним указатель на текущий элемент
- Класс **`Iterator`** определен внутри класса **`List`**.
  - Позволяет дать классу **`Iterator`** доступ к закрытым полям.
  - **`Iterator`** будет использовать тот же тип данных (T) что и класс List.

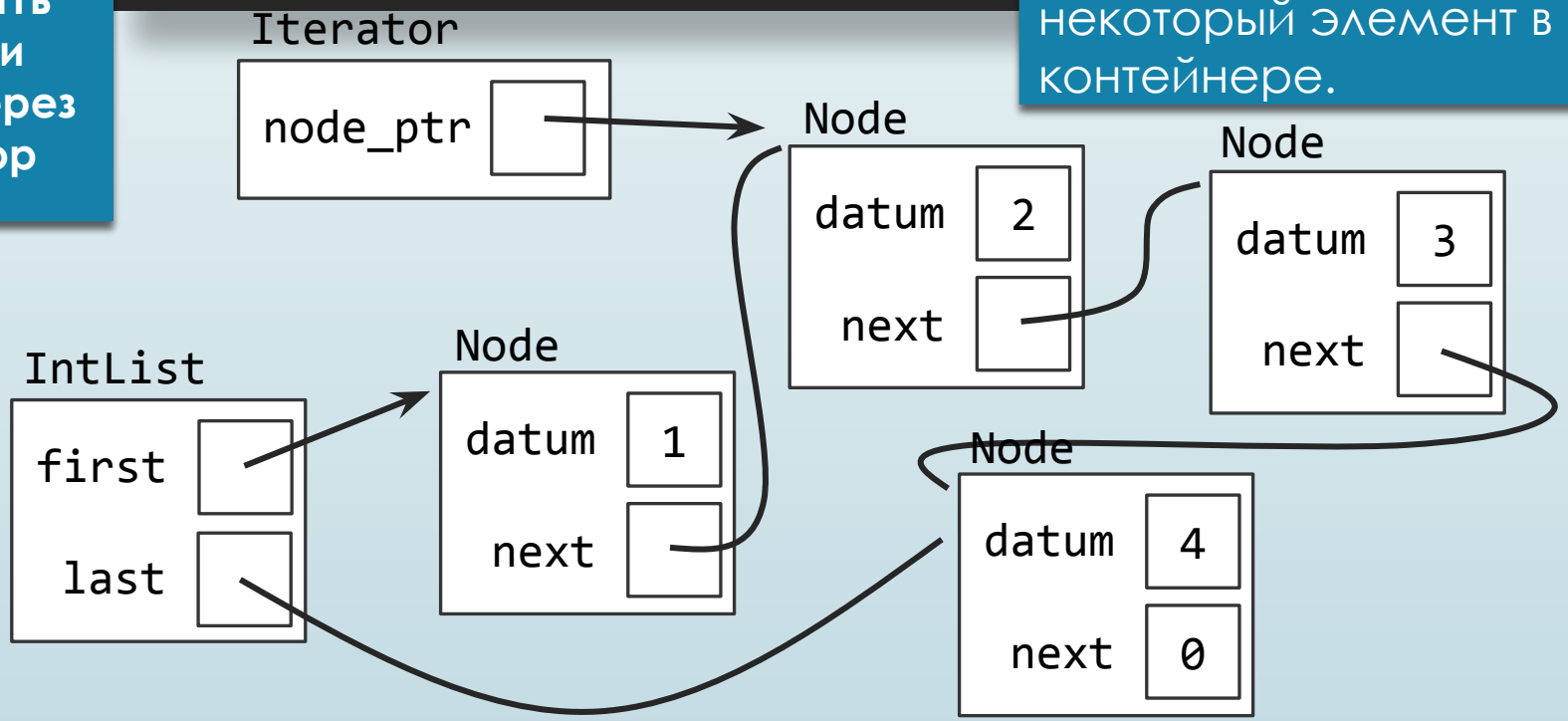
```
template <typename T>
class List {
public:
  ...
  class Iterator {
  public:
    T & operator*() const;
    ...
  private:
    Node *node_ptr;
  };
  ...
private:
  struct Node {
    T datum;
    Node *next;
  };
  Node *first;
  Node *last;
  ...
};
```

List Iterator: The * operator

```
// ТРЕБОВАНИЕ: разыменованный указатель
// РЕЗУЛЬТАТ:  Возвращает элемент на который указывает итератор.
template <typename T>
T & List<T>::Iterator::operator*() const {
  assert(node_ptr);
  return node_ptr->datum;
}
```

Возвращаем по ссылке чтобы разрешить запись и чтение через итератор

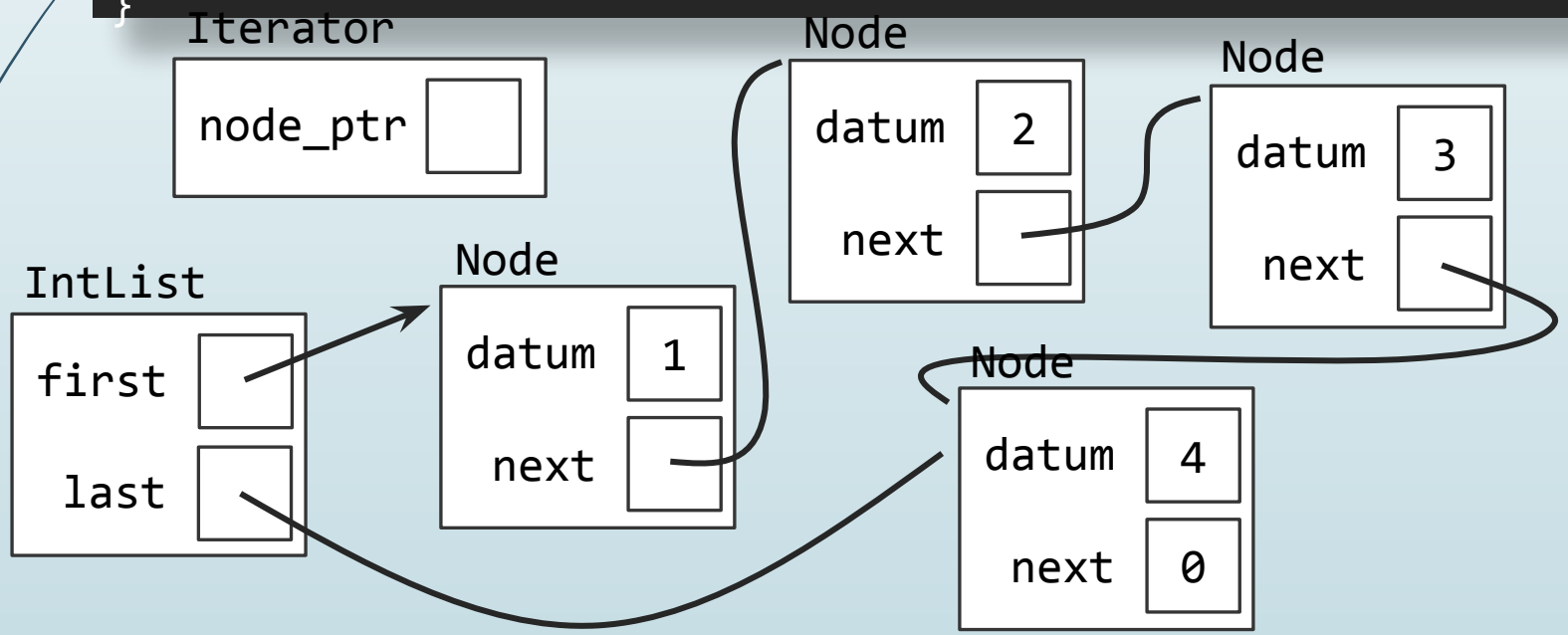
Итератор является разыменованным, если он указывает на некоторый элемент в контейнере.



List Iterator: The ++ operator (prefix[1])

```
// ТРЕБОВАНИЕ: разыменованный указатель
// РЕЗУЛЬТАТ:  Возвращает элемент на который указывает итератор.
template <typename T>
typename List<T>::Iterator & List<T>::Iterator::operator++()
{
  assert(node_ptr);
  node_ptr = node_ptr->next;
  return *this;
}
```

Ключевое слово



1 The postfix increment operator can also be overridden.

Ключевое слово typename

Ключевое слово **typename** требуется при использовании типа, вложенного в другой тип, который зависит от параметра шаблона.

```
template <typename T>
void func() {

  IntList::Iterator it1;

  List<int>::Iterator it2;

  typename List<T>::Iterator it3;

}
```

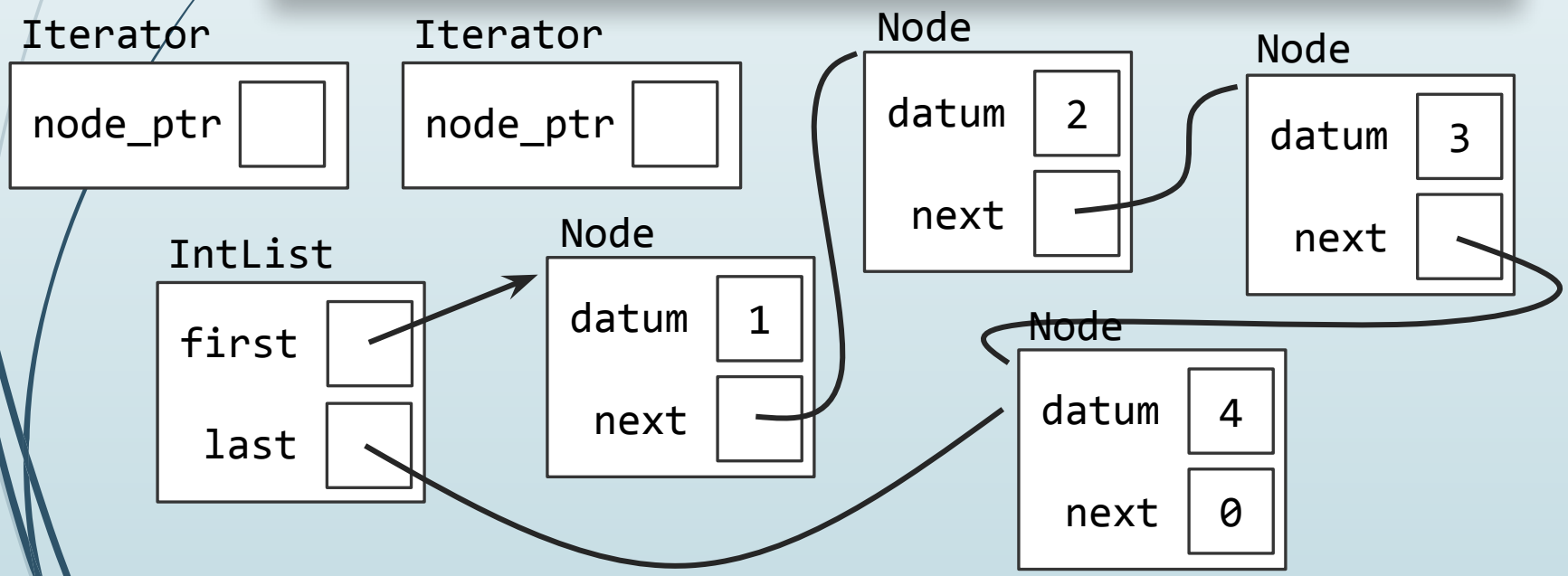
typename не обязателен если IntList не зависит от параметра T.

typename не обязателен если List<int> не зависит от параметра

typename обязательно, List<T> зависит от параметра T.

List Iterator: The == operator[1]

```
template <typename T>
bool List<T>::Iterator::operator==(Iterator rhs) const {
  return node_ptr == rhs.node_ptr;
}
```



1 Оператор != operator определяется аналогично.

Создание итератора

- Создадим 2 конструктора для `Iterator`.

```
class Iterator {
public:
  Iterator()
    : node_ptr(nullptr) { }
  ...

private:
  // Создает итератор который указывает на определенный
// узел.
  Iterator(Node *np)
    : node_ptr(np) { }

  Node *node_ptr;
};
```

Получение итератора для контейнера

- Как получить итератор для контейнера…

```
List<int> list;
int arr[3] = { 1, 2, 3 };
fillFromArray(list, arr, 3);


List<int>::Iterator end = _____;
for (List<int>::Iterator it = _____; it != end; ++it) {
  cout << *it << endl;
}
```

Итераторы для начала и конца

- Реализуем функции **begin ()** и **end ()** для класса **List**

begin() и end()

```
template <typename T>
class List {
public:
  ...
  class Iterator {
  public:
    Iterator() : node_ptr(nullptr) { }
    ...
  private:
    Iterator(Node *np) : node_ptr(np) { }
    Node *node_ptr;
  };

  Iterator begin() { return Iterator(first); }

  Iterator end() { return Iterator(); }
  ...
private:
  Node *first;
  ...
};
```

Что не так?

begin() использует конструктор для создания итератора для первого элемента

end() использует конструктор по умолчанию.

Friend Declarations

```
template <typename T>
class List {
public:
  ...
  class Iterator {
    friend class List;
  public:
    Iterator() : node_ptr(nullptr) { }
    ...
  private:
    Iterator(Node *np) : node_ptr(np) { }
    Node *node_ptr;
  };

  Iterator begin() { return Iterator(first); }

  Iterator end() { return Iterator(); }
  ...
private:
  Node *first;
  ...
};
```

Сделаем класс дружественным чтобы получить доступ к закрытым полям Iterator.

Теперь можем обращаться к приватным функциям f

Обход по итератору

- Теперь можем реализовать обход списка по итератору

```
List<int> list;
int arr[3] = { 1, 2, 3 };
fillFromArray(list, arr, 3);


List<int>::Iterator end = list.end();
for (List<int>::Iterator it = list.begin(); it != end; ++it) {
  cout << *it << endl;
}
```


```
int main() {
  List<int> list;
  int arr[3] = { 1, 2, 3 };
  fillFromArray(list, arr, 3);

  List<int>::Iterator it1 = list.begin();
  ++it1;
  List<int>::Iterator it2 = it1;
  ++it2;
  it1 = list.end();

}
```

Интерфейс итератора

- Итераторы обеспечивают интерфейс для обхода последовательностей элементов.
- Позволяют повторно использовать код для работы с различными контейнерами для которых реализован интерфейс итератора.
- Контейнеры STL работают аналогично:

```
vector<int> vec;
// заполним вектор элементами

vector<int>::iterator end = vec.end();
for (vector<int>::iterator it = vec.begin(); it != end; ++it) {
  cout << *it << endl;
}
```

Основные функции

- Основная особенность итераторов заключается в том, что мы можем писать функции для работы с итераторами, а не с конкретным контейнером.
- Это позволяет использовать ту же функцию с различными контейнерами!

- STL контейнеры, работают с функциями аналогично, например `std::sort`

```
int main() {
  vector<int> vec;
  sort(vec.begin(), vec.end());
}
```

Пример: max_element

```
template <typename Iter_type>
Iter_type max_element(Iter_type begin, Iter_type end) {

  Iter_type maxIt = begin;

  for (Iter_type it = begin; it != end; ++it) {
    if (*it > *maxIt) {
      maxIt = it;
    }
  }
  return maxIt;
}

int main() {
  vector<int> vec; // заполнен числами
  cout << *max_element(vec.begin(), vec.end()) << endl;
}
```

Изначально считаем первый элемент максимальным.

Обход по итератору, проверка каждого элемента.

Как только находим больший элемент обновляем значение

Разыменовываем итератор для доступа к элементу

Использование max_element

- Пока мы работаем с контейнером, поддерживающим итераторы, нам не нужно писать цикл поиска!

```
int main() {
  vector<int> vec; // fill with numbers
  cout << *max_element(vec.begin(), vec.end()) << endl;

  List<int> list; // fill with numbers
  cout << *max_element(list.begin(), list.end()) << endl;

  List<Card> cards; // fill with Cards
  cout << *max_element(cards.begin(), cards.end()) << endl;

  int const SIZE = 10;
  double arr[SIZE]; // fill with numbers
  cout << *max_element(arr, arr + SIZE) << endl;
}
```

Указатели работают как итераторы

Exercise: no_duplicates

- Напишите шаблон функции, который принимает итераторы начала и конца и определяет, содержит ли данный диапазон повторяющиеся элементы.
- Пример:

```
bool no_duplicates(int arr[], int size) {
  for (int i = 0; i < size; ++i) {
    for (int k = i + 1; k < size; ++k) {
      if (a[i] == a[k]) {
        return false; // Если есть повторения возвращаем
//false
      }
    }
  }
  return true; //
}
```

Solution: no_duplicates

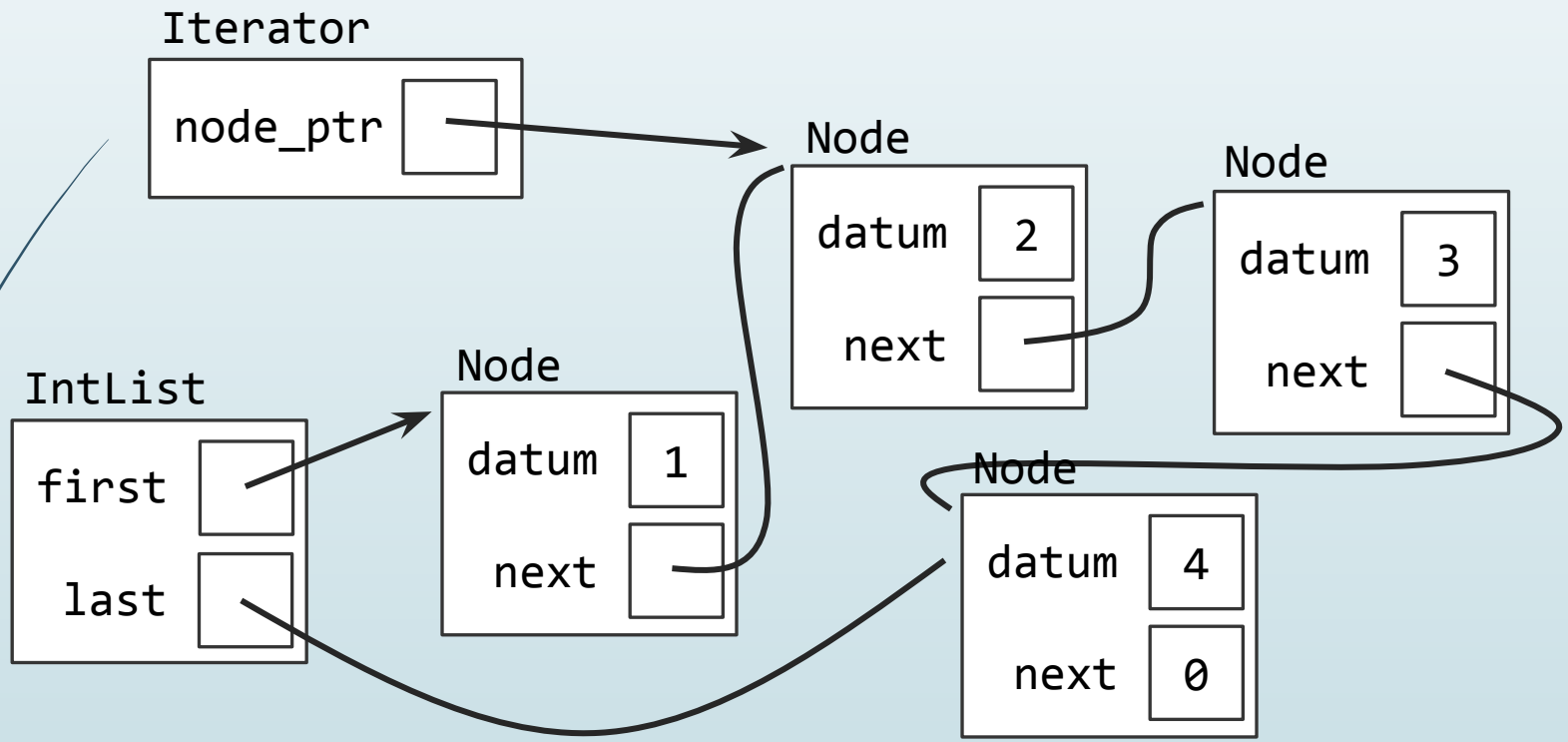
- Напишите шаблон функции, который принимает итераторы начала и конца и определяет, содержит ли данный диапазон повторяющиеся элементы.

```
template <typename Iter_type>
bool no_duplicates(Iter_type begin, Iter_type end) {
  for (Iter_type it1 = begin; it1 != end; ++it1) {
    Iter_type it2 = it1;
    ++it2;
    for (; it2 != end; ++it2) {
      if (*it1 == *it2) {
        return false; // Если есть повторения возвращаем
//false

      }
    }
  }
  return true;
}
```

Iterator большая тройка?

- Нам нужна своя реализация большой тройки для итераторов?



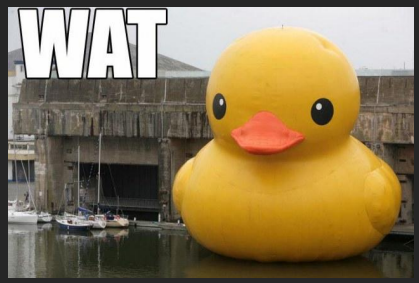
Невалидные итераторы

```
int main() {
  List<int> list;
  list.push_back(1);
  list.push_back(2);

  List<int>::Iterator it = list.begin();
  cout << *it << endl;

  list.pop_front();

  cout << *it << endl; // EXPLODE
}
```



Невалидные итераторы

- Невалидные итераторы как же опасны как указатели.
- Казалось бы, безобидные операции над контейнером могут привести к невалидности итератора.
  -
  - Например, итераторы, указывающие на вектор, недействительны, если контейнер изменился (например стал больше).