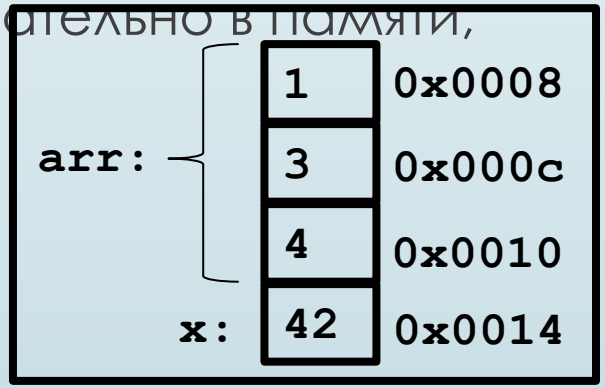
Последовательный контейнеры (Sequential Containers)

- Предоставляют последовательный доступ к элементам
- Позволяют обращаться к элементам по индексу.
- Позволяют контролировать порядок элементов.

- Как организовать хранение элементов?
- Расположим их последовательно в памяти, один за другим.
  - `vector, string .`
  - `int arr[3] = {1,3,4};`
    `int x = 42;`

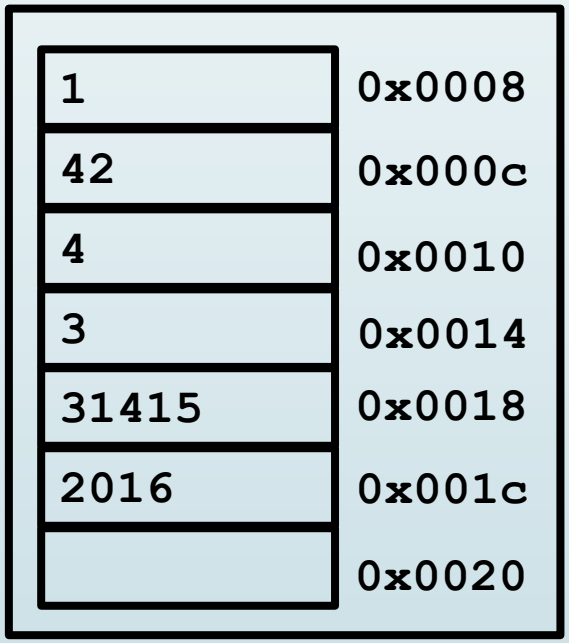


Использование непрерывной области памяти

- Использование непрерывной области памяти позволяем использовать арифметику указателей, но с некоторыми ограничениями…
- Добавление или удаление элемента в середину последовательного контейнера занимает много времени:
  - необходимо переместить все элементы
  - добавление нового элемента может привести к выделению дополнительной области памяти

Хранение элементов не последовательно

- Как организовать последовательный контейнер без использования непрерывной памяти?
- Мы можем запоминать адрес следующего элемента...
  - Указатели!
  - Пусть каждый объект контейнера содержит в себе не только **данные**, но и **указатель на следующий элемент**



Узел(Nodes)

- Каждый элемент списка включающий данные и указатель на следующий элемент назовем узлом (вершиной, node)

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

Данные

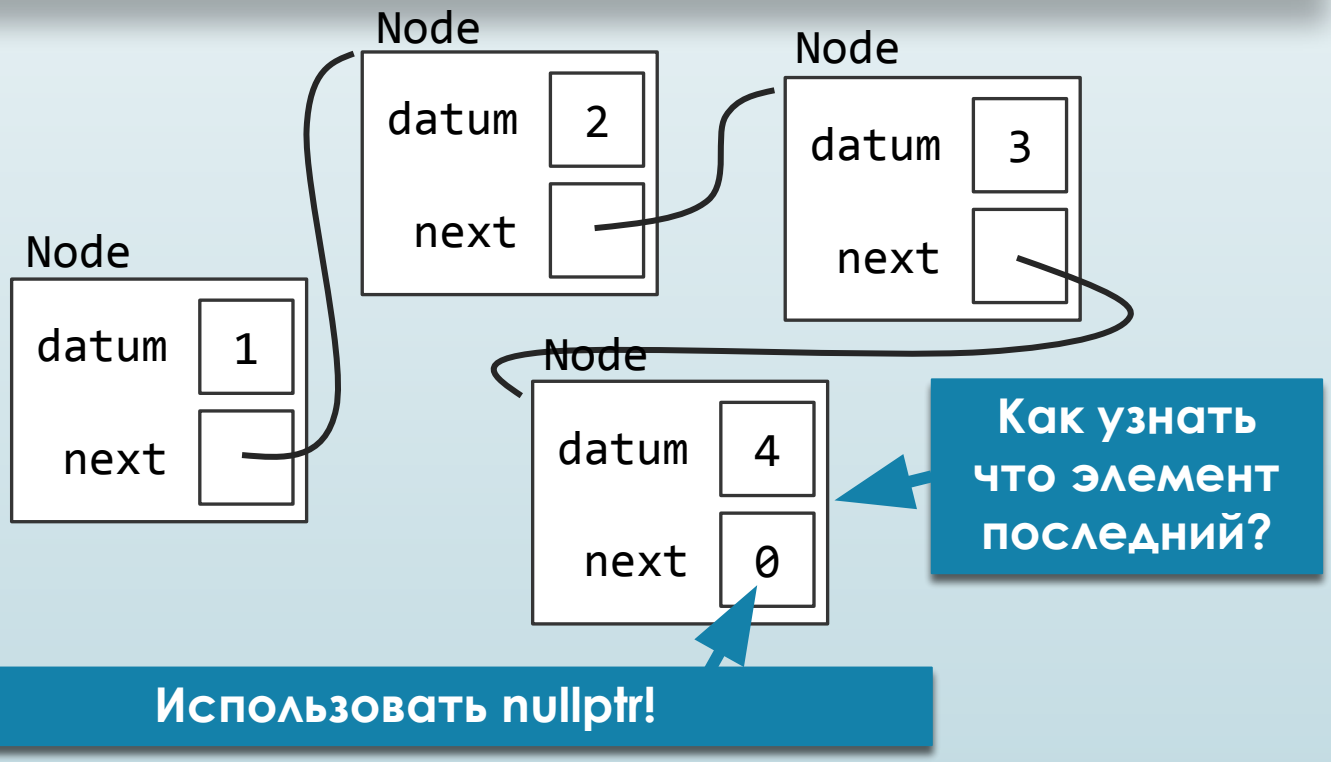
Указатель на следующий элемент

Nodes

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

Данные

Указатель на следующий элемент

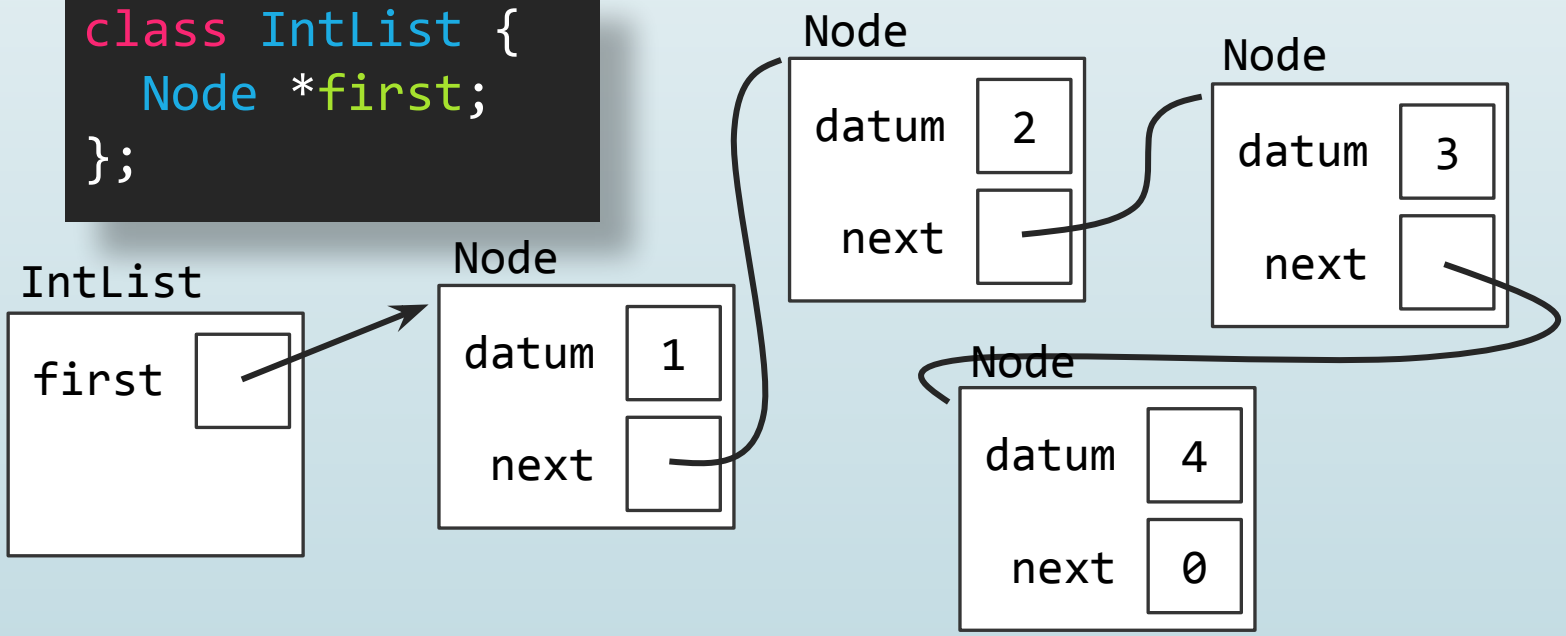


Связный список

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

- Перейдем к использованию класса.

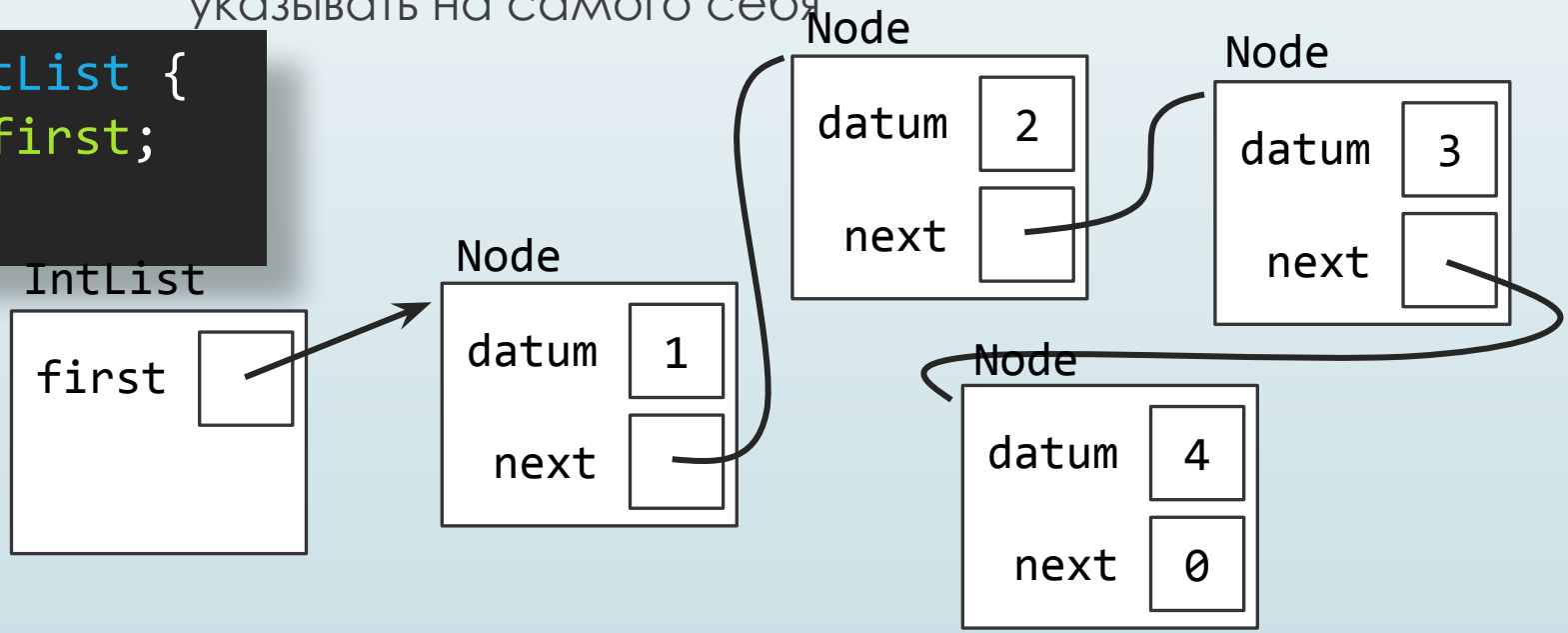
```
class IntList {
  Node *first;
};
```



IntList инварианты представления

- `first` если список пустой содержит nullptr, в противном случае указатель на следующий
- Последний узел **всегда указывает на** `nullptr`
- Указатель на следующий элемент не должен указывать на самого себя

```
class IntList {
  Node *first;
};
```



IntList Объявление

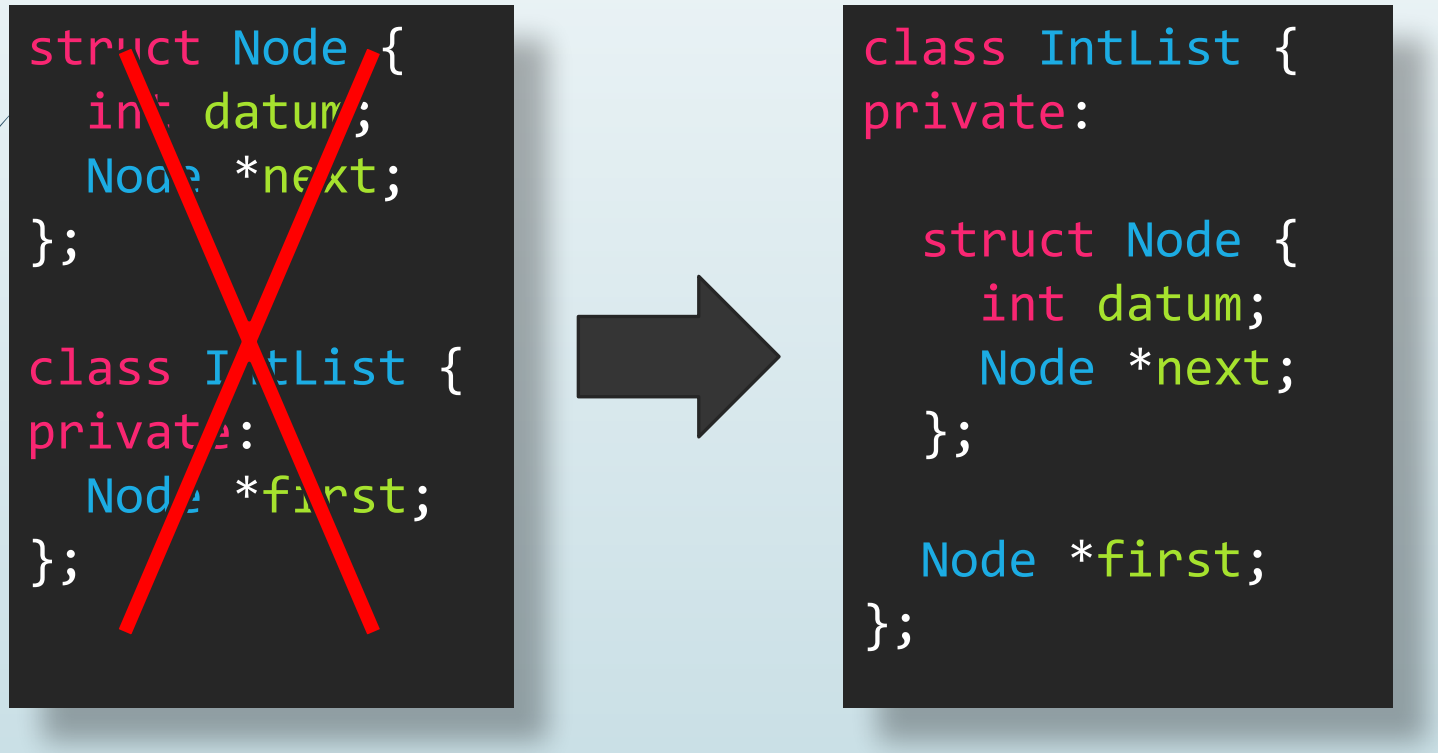
```
class IntList {
public:
  // РЕЗУЛЬТАТ: создает пустой список
  IntList();

  // РЕЗУЛЬТАТ: возвращает true если список пустой
  bool empty() const;

  // ТРЕБОВАНИЕ: список не пустой
  // РЕЗУЛЬТАТ:  Возвращает (по ссылке) первый элемент
  int & front();

  // РЕЗУЛЬТАТ: вставляет данные в начало списка
  void push_front(int datum);

  // ТРЕБОВАНИЕ: список не пустой
  // РЕЗУЛЬТАТ:  удаляет первый элемент
  void pop_front();
  ...
};
```

Использвание IntList

```
int main() {
  IntList list;           // ( )
  list.push_front(1);     // ( 1 )
  list.push_front(2);     // ( 2 1 )
  list.push_front(3);     // ( 3 2 1 )

  cout << list.front(); // 3

  list.front() = 4;       // ( 4 2 1 )

  list.pop_front();       // ( 2 1 )
  list.pop_front();       // ( 1 )
  list.pop_front();       // ( )

  cout << list.empty(); // true (or 1)
}
```

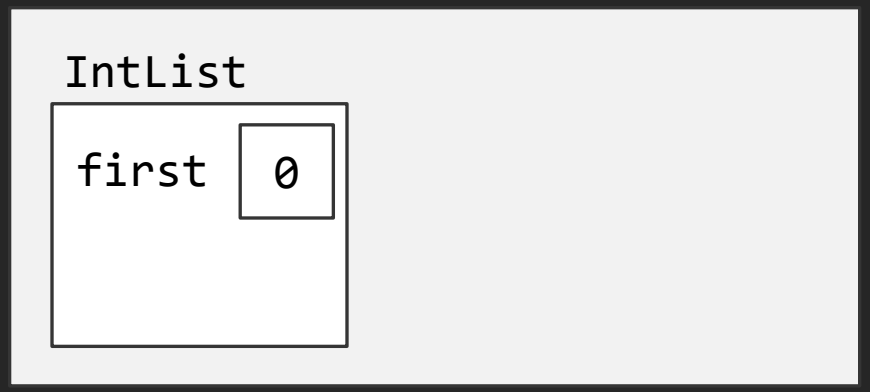
Сокрытие реализации

- Спрячем структуру **Node** в класс **IntList**.



Реализация IntList: Конструктор

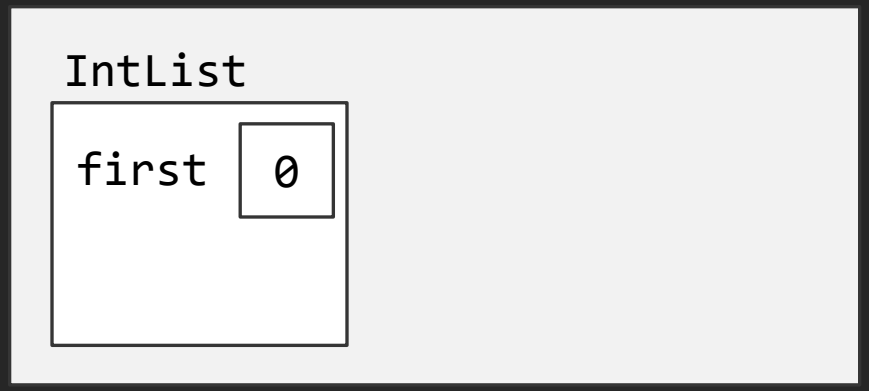
```
class IntList {
private:
  struct Node {
    int datum;
    Node *next;
  };
  Node *first;

public:
  // РЕЗУЛЬТАТ: создает пустой список
  IntList() : first(nullptr) { }

  ...
};
```



Список пустой, первый элемент nullptr.

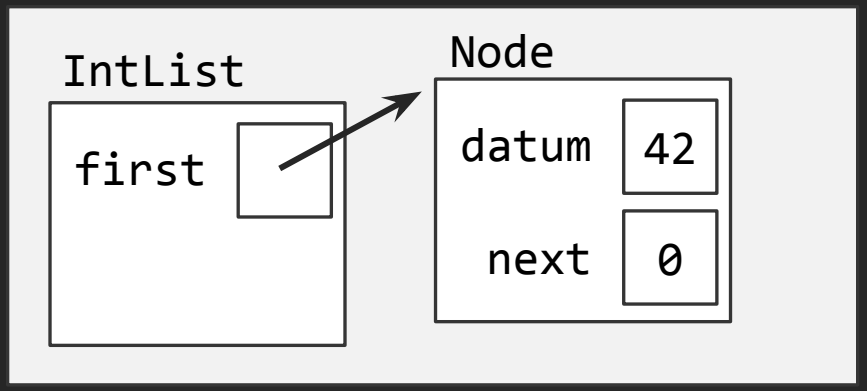
Реализация IntList: empty

```
class IntList {
private:
  struct Node {
    int datum;
    Node *next;
  };
  Node *first;

public:
  // РЕЗУЛЬТАТ: возвращает true если список пустой
  bool empty() const {
    return first == nullptr;
  }

  ...
};
```



Реализация IntList: front

```
class IntList {
private:
  struct Node {
    int datum;
    Node *next;
  };
  Node *first;

public:
   // ТРЕБОВАНИЕ: список не пустой
  // РЕЗУЛЬТАТ:  Возвращает (по ссылке) первый элемент

  int & front() {
    assert(!empty());
    return first->datum;
  }
  ...
};
```



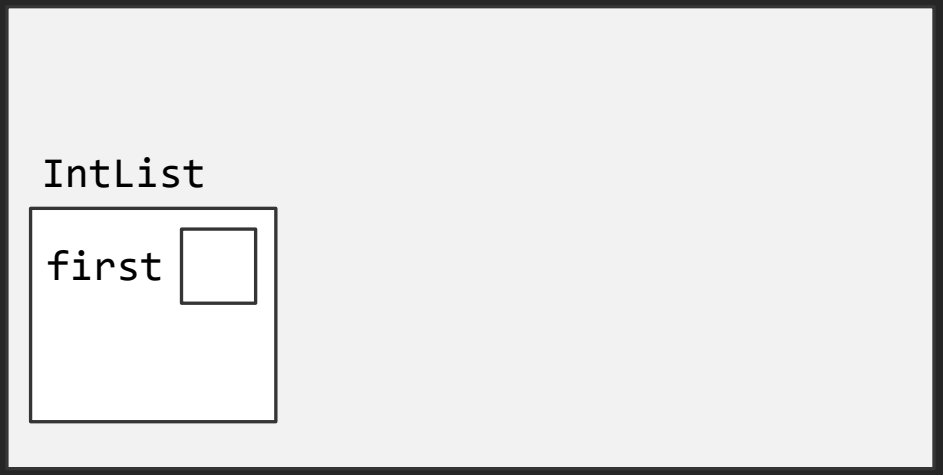
Использование IntList

```
int main() {
  IntList list;           // ( )
  list.push_front(1);     // ( 1 )
  list.push_front(2);     // ( 2 1 )
  list.push_front(3);     // ( 3 2 1 )

  cout << list.front(); // 3

  list.front() = 4;       // ( 4 2 1 )

  list.pop_front();       // ( 2 1 )
  list.pop_front();       // ( 1 )
  list.pop_front();       // ( )

  cout << list.empty(); // true (or 1)
}
```

Возможно так как объект возвращается по ссылке

Реализация IntList: push_front

```
class IntList {
private:
  struct Node {
    int datum;
    Node *next;
  };
  Node *first;

public:
  // РЕЗУЛЬТАТ: вставляет данные в начало списка
    void push_front(int datum) {
    Node *p = new Node;
    p->datum = datum;
    p->next = first;
    first = p;
  }
  ...
};
```



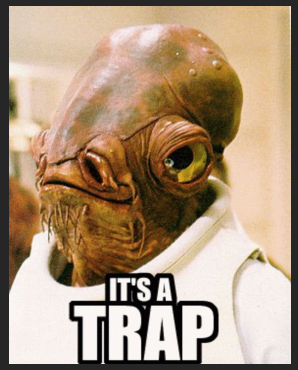
Exercise: pop_front

```
class IntList {
private:
  struct Node {
    int datum;
    Node *next;
  };
  Node *first;

public:
  // ТРЕБОВАНИЕ: список не пустой
  // РЕЗУЛЬТАТ:  удаляет первый элемент
  void pop_front() {


  }
  ...
};
```

Solution: pop_front

```
class IntList {
private:
  struct Node {
    int datum;
    Node *next;
  };
  Node *first;

public:
  // ТРЕБОВАНИЕ: список не пустой
  // РЕЗУЛЬТАТ:  удаляет первый элемент
  void pop_front() {
    assert(!empty());
    delete first;
    first = first->next;
  }
  ...
};
```

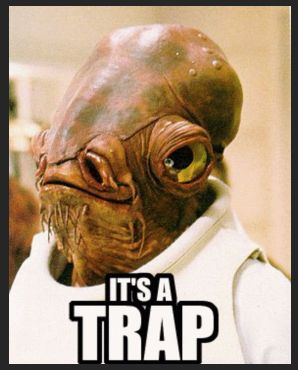
Что не так?



Solution: pop_front

```
class IntList {
private:
  struct Node {
    int datum;
    Node *next;
  };
  Node *first;

public:
  // ТРЕБОВАНИЕ: список не пустой
  // РЕЗУЛЬТАТ:  удаляет первый элемент
  void pop_front() {
    assert(!empty());
    first = first->next;
    delete first;
  }
  ...
};
```

Так лучше?



Solution: pop_front

```
class IntList {
private:
  struct Node {
    int datum;
    Node *next;
  };
  Node *first;

public:
  // ТРЕБОВАНИЕ: список не пустой
  // РЕЗУЛЬТАТ:  удаляет первый элемент
  void pop_front() {
    assert(!empty());
    Node *victim = first;
    first = first->next;
    delete victim;
  }
  ...
};
```

Использование временной переменной для хранения указателя на узел который необходимо удалить

Проход списка

- Использование указателей для прохода по списку.
  - Начиная с первого узла `Node`.
  - Двигаемся к следующему по указателю.
  - На каждом шаге получаем доступ к данным.
  - Останавливаемся когда находим указатель null.

```
class IntList {
public:
  // РЕЗУЛЬТАТ:  вывод списка
  void print(ostream &os) const {
     }
  ...
};
```

Solution: Проход списка

```
class IntList {
private:
  struct Node {
    int datum;
    Node *next;
  };
  Node *first;

public:
  // РЕЗУЛЬТАТ:  Вывод списка
  void print(ostream &os) const {
    for (Node *np = first; np; np = np->next) {
      os << np->datum << " ";
    }
  }
  ...
};
```

Использование IntList

```
int func() {
  IntList list;
  list.push_front(1);
  list.push_front(2);
  list.print(cout);
}
```

Большая тройка

- Как использовать большую тройку в нашей реализации?

Собственная большая тройка

- Когда нужна своя версия?
  - Если нужно полное копирование.
  - Полная копия нужна если объект имеет собственные ресурсы (например динамически выделенная память)
- Советы:
  - Проверьте конструктор. Если в конструкторе выделяется память, то вероятнее всего необходима вся большая тройка.
  - Обратите внимание на поля. Если среди них есть указатели, то вероятнее всего необходима вся большая тройка.

Большая тройка

- Нам нужна большая тройка для `IntList`?

- Да. `IntList` использует динамическую память.

Большая тройка

- Деструктор
  1. Освобождение ресурсов
- Конструктор копий
  1. Копирование полей
  2. Копирование ресурсов
- Оператор присваивания
  1. Проверяет на присвоение самого себя
  2. Освобождает старые ресурсы
  3. Копирует поля
  4. Копирует ресурсы
  5. Возвращает `*this`

Большая тройка

Как избежать дублирования?

- Деструктор
  1. Освобождение ресурсов — `pop_all()`
- Конструктор копий
  1. Копирование полей
  2. Копирование ресурсов — `push_all()`
- Оператор присваивания
  1. Проверка самоприсоединения
  2. Освобождение ресурсов — `pop_all()`
  3. Копирование полей `rhs`
  4. Копирование ресурсов `rhs` — `push_all()`
  5. return `*this`

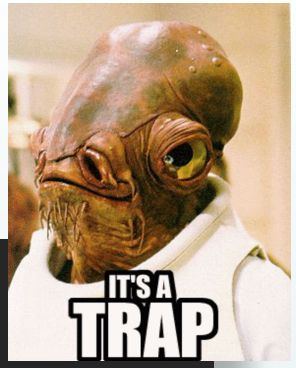
pop_all и push_all

```
class IntList {
...
private:
   ...

   // РЕЗУЛЬТАТ: удаляем все элементы
   void pop_all();

   // РЕЗУЛЬТАТ: копирование из other
   void push_all(const IntList &other);

};
```

Реализация pop_all

```
class IntList {
...
private:
  ...
  // РЕЗУЛЬТАТ: удаляем все элементы
  void pop_all() {
    while (!empty()) {
      pop_front();
    }
  }

  // РЕЗУЛЬТАТ: копирование из other
  void push_all(const IntList &other);

};
```

Реализация push_all




```
class IntList {
...
private:
  ...
  // РЕЗУЛЬТАТ: удаляем все элементы
  void pop_all() {
    while (!empty()) {
      pop_front();
    }
  }

  // РЕЗУЛЬТАТ: копирование из other
  void push_all(const IntList &other) {
    for (Node *np = other.first; np; np = np->next) {
      push_front(np->datum);
    }
  }
};
```

Что не так?

Реализация push_all

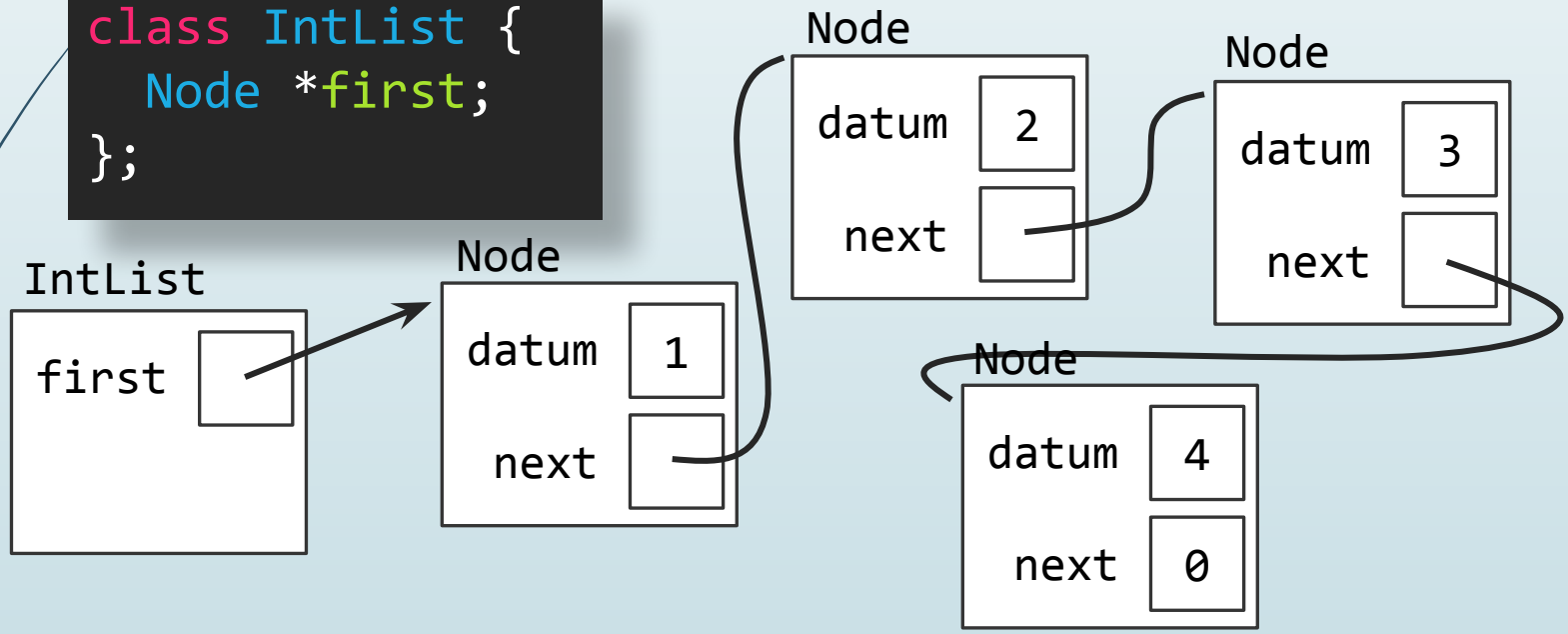
```
class IntList {
...
private:
  ...
  // РЕЗУЛЬТАТ: удаляем все элементы
  void pop_all() {
    while (!empty()) {
      pop_front();
    }
  }

  // РЕЗУЛЬТАТ: копирование из other
  void push_all(const IntList &other) {
    for (Node *np = other.first; np; np = np->next) {
      push_back(np->datum);
    }
  }
};
```

Для избежания копирования в обратном порядке необходимо использовать функцию push_back

Реализация push_back

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

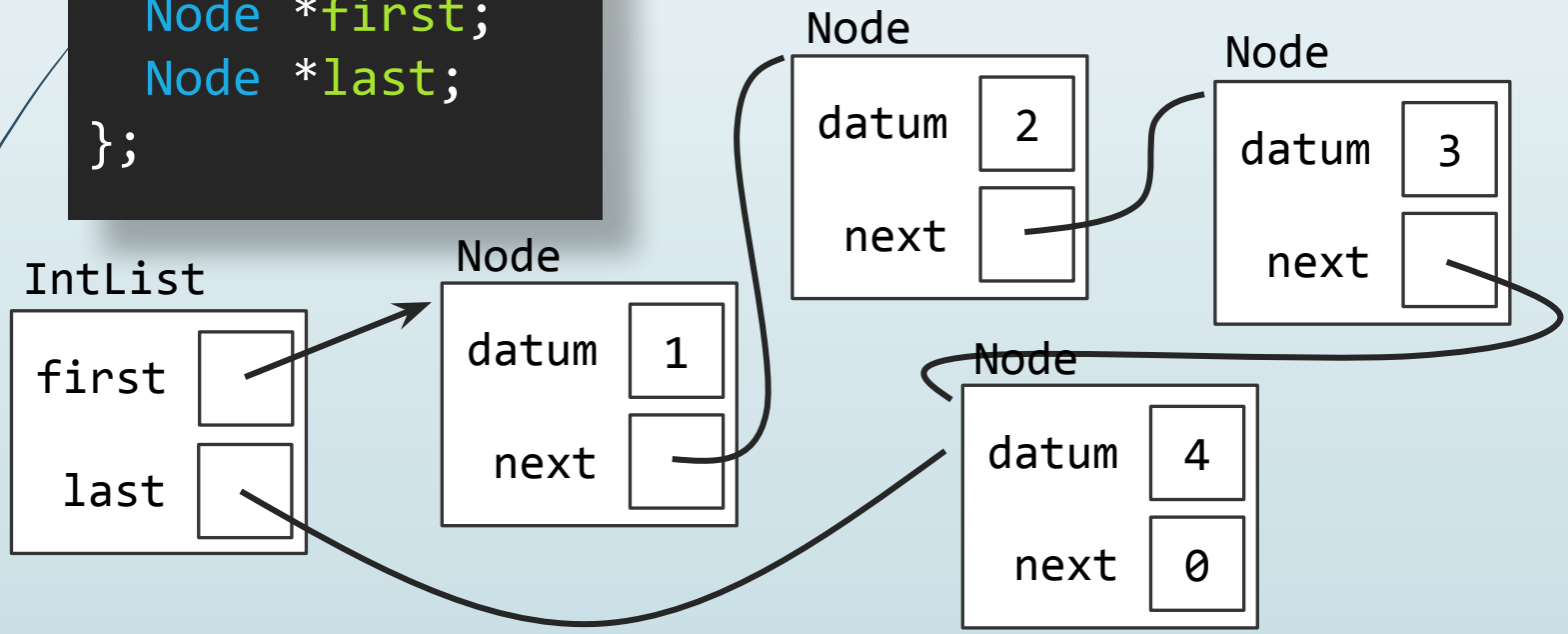
```
class IntList {
  Node *first;
};
```



Реализация push_back

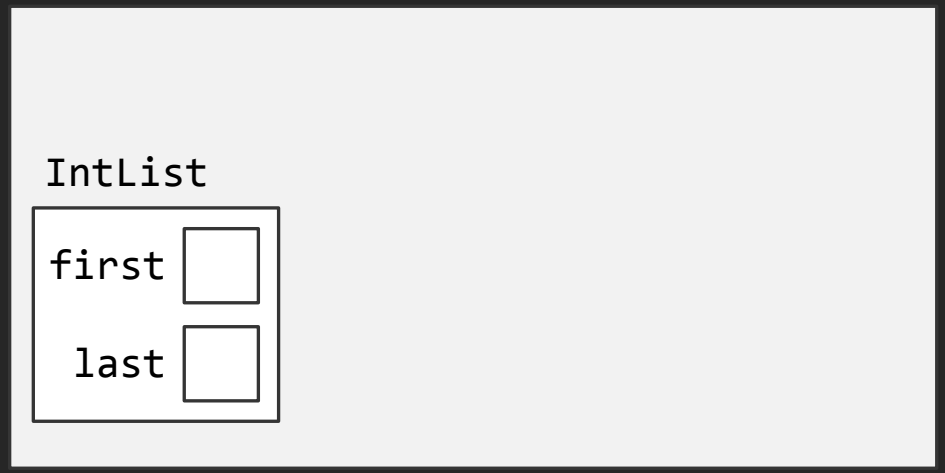
- Что если хотим добавить элемент в конец списка?
  - Нужно прости все элементы начиная с первого!
  - Может изменим представление данных…

```
class IntList {
  Node *first;
  Node *last;
};
```

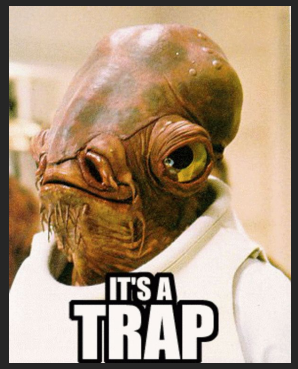


Реализация IntList: push_back

```
class IntList {
private:
  struct Node {
    int datum;
    Node *next;
  };
  Node *first;
  Node *last;
public:
  // добавление элемента в конец списка
  void push_back(int datum) {
    Node *p = new Node;
    p->datum = datum;
    p->next = nullptr;
    last->next = p;
    last = p;
  }
  ...
};
```

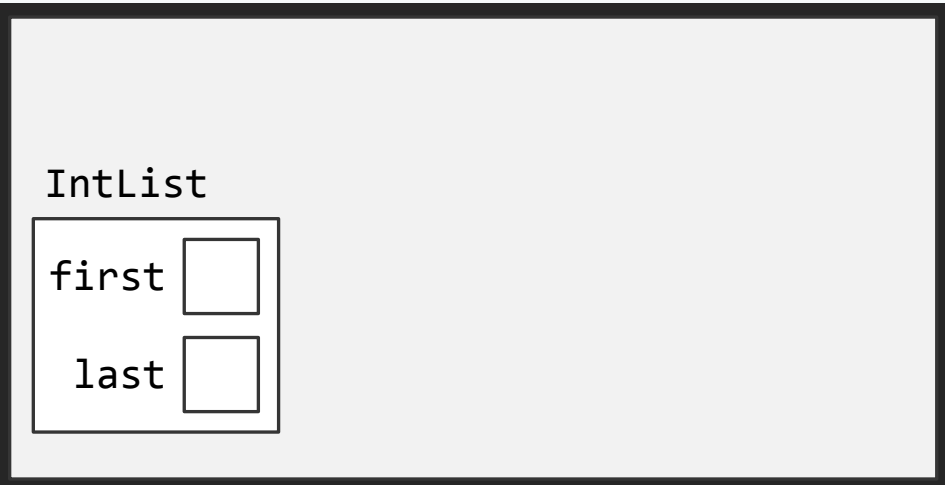


Что не так?



Реализация IntList: push_back

```
class IntList {
private:
  struct Node {
    int datum;
    Node *next;
  };
  Node *first;
  Node *last;
public:
  // Добавление элемента в конец списка
  void push_back(int datum) {
    Node *p = new Node;
    p->datum = datum;
    p->next = nullptr;
    if (empty()) { first = last = p; }
    else {
      last->next = p;
      last = p;
    }
  }
};
```



IntList Большая тройка

```
class IntList {
public:
  ...
  ~IntList() {
    pop_all();
  }

  IntList(const IntList &other)
    : first(nullptr), last(nullptr) {
    push_all(other);
  }

  IntList & operator=(const IntList &rhs) {
    if (this == &rhs) { return *this; }
    pop_all();
    push_all(rhs);
    return *this;
  }
  ...
};
```