Объектно-ориентированное программирование в C++ (продолжение)

Курс «Технология программирования»

Кафедра управления и информатики НИУ «МЭИ»

Осень 2015 г.

Копирование и перемещение

Копирование

- Новый объект извне неотличим от исходного, *который не меняется*.
- Происходит:
  - при передаче в функцию по значению;
  - во время присваивания;

```
vector<int> x { 1, 2, 3 };
auto copy_of_x = x;
```

  - когда это явно указано.

```
LinkedList x { 1, 2, 3 };
LinkedList y(x);
```

Перемещение

- Исходный объект передает свое состояние новому, *теряя полезные свойства.*
- Происходит:
  - при возврате значения из функции;
  - во время присваивания;

```
auto node =
    make_unique<Node>();
```

  - когда это явно указано.

```
auto holder = move(left);
```

Копирование LinkedList

```
LinkedList(const LinkedList& source)
    : LinkedList()
{
    Node* current = source.first;
    while (current)
    {
        push_back(current->value);
        current = current->next;
    }
}
```

Чтобы можно было вызывать push_back(), нужно инициализировать список пустым.

```
for (auto& item : source)
{
    push_back(item);
}
```

error: no matching function for call to 'LinkedList::begin() const'

Неизменяющие методы

```
for (auto& item : source)
{
    push_back(item);
}
```

LinkedList

```
Iterator begin() const { … }
Iterator end()   const { … }
size_t get_size() const { … }
```

```
for (auto it = source.begin(); it != source.end(); ++it)
{
    auto& item = *it;
    push_back(item);
}
```

Не изменят ли эти методы неизменяемый source?

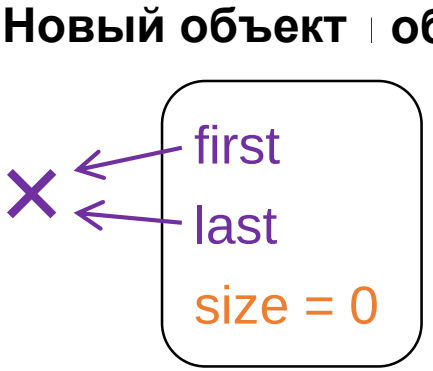
Список неизменяемый: не должно допускаться изменение элементов (*it = 42).

LinkedList :: Iterator

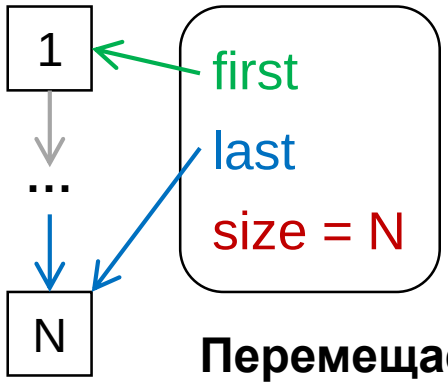
```
double&       operator*()       { … }
const double& operator*() const { … }
```

Перемещение LinkedList

```
void swap(LinkedList& a, LinkedList& b)
{
    swap( a.first, b.first );
    swap( a.last, b.last );
    swap( a.size, b.size );
}
```




```
LinkedList::LinkedList(LinkedList&& source)
    : LinkedList()
{
    swap(*this, source);
}
```



Присваивание LinkedList

Управление возможностью копирования и перемещения

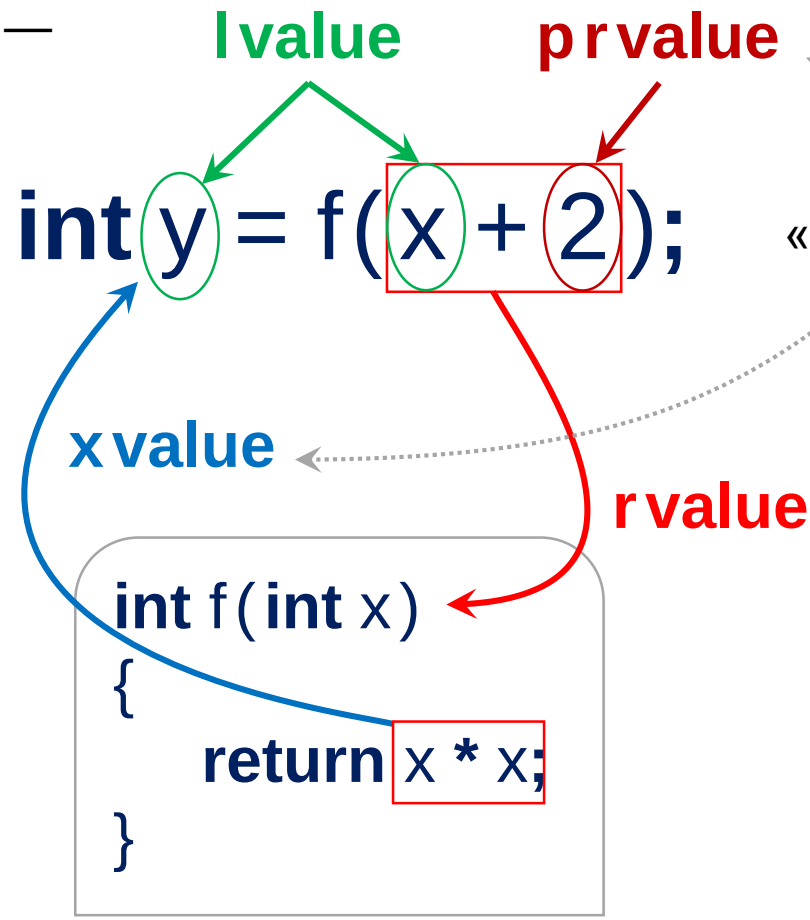
```
template<typename T> // Недопустимо копировать,
struct unique_ptr            // требуется перемещать.
{
   // 1. Запретить копирование:
   unique_ptr(const unique_ptr&) = delete;
   unique_ptr& operator=(const unique_ptr&) = delete;

   // 2. Это запретит и перемещение — а его нужно разрешить.
   //      — Подходит перемещение по умолчанию:
   unique_ptr(unique_ptr&&) = default;
   unique_ptr& operator=(unique_ptr&&) = default;
};
```

Виды выражений и ссылок

«Left-side value» — то, чему можно присвоить значение.

Время жизни «eXpiring value» завершается.



«pure» (просто) значение

«Right-side value» — то, чему значение присвоить нельзя.

результат вычислений

&

&&

Полиморфизм

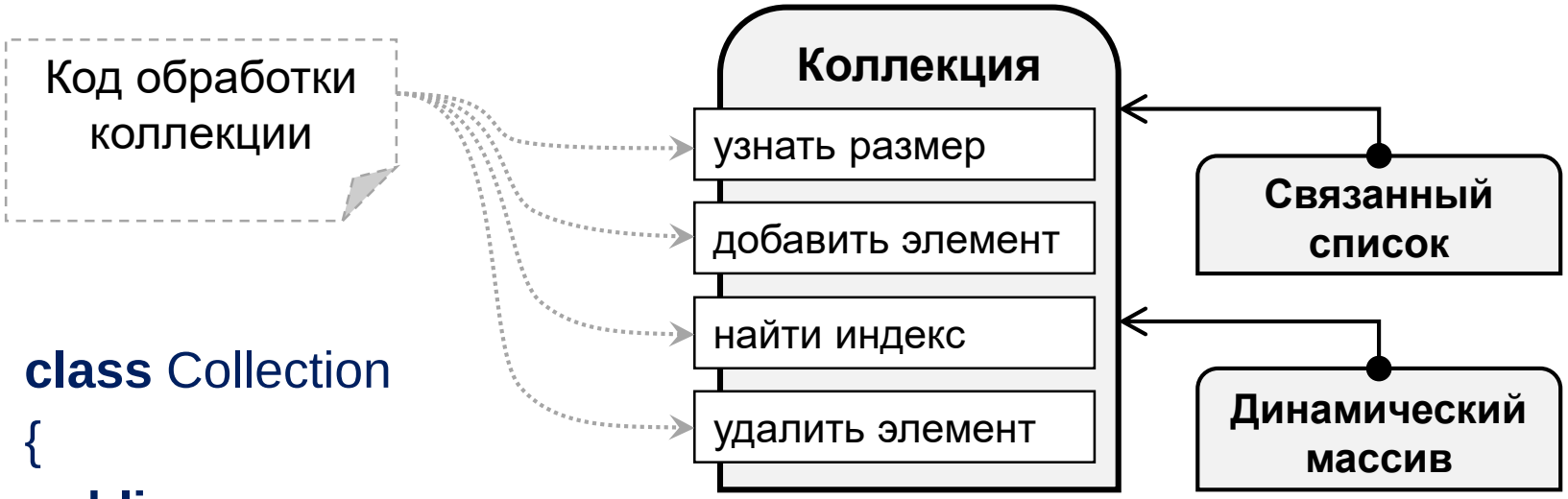
- ```
  void fill(Collection& container)
  {
      for (size_t i = 1; i < 1024; i *= 2)
      {
          container.add(i);
      }
  }
  ```

- ```
  LinkedList  the_list;
  DynamicArray  the_array;
  fill(the_list);
  fill(the_array);
  ```

Может ссылаться на объект любого класса-наследника интерфейса Collection.

Если метод add() виртуальный, будет вызвана его версия из того класса, на который реально ссылается container.

Интерфейс коллекций




```
class Collection
{
public:
   virtual size_t  get_size() const = 0;
   virtual void    push_back(double item) = 0;
   virtual size_t  find(double item) const = 0;
   virtual void    remove(size_t index) = 0;
};
```

В классе Collection нет реализации этих методов…

…реализация есть в классах-наследниках.

Реализация интерфейса

```
class LinkedList : public Collection
{
public:
    size_t  get_size() const override;
    void    push_back(double item) override;
    size_t  find(double item) const override;
    void    remove(size_t index) override;
    // …
};

size_t LinkedList::get_size() const {
    return size;
}
```

Класс LinkedList наследует класс Collection.

Реализации чисто виртуальных методов базового класса (= 0).

Статический полиморфизм

- ```
  LinkedList<double> the_list;
  fill(the_list);
  vector<double> the_vector;
  fill(the_vector);
  ```
- ```
  template<typename C>
  void fill(C&& container)
  {
      for (size_t i = 0; i < 10; ++i)
          container.push_back(i);
  }
  ```

Для каждого вызова fill<C>() компилятор подставит конкретный тип контейнера и вызов C::push_back(), если такой метод есть.

Универсальная ссылка (universal reference) — работает как & или &&, смотря, что требуется.

- Метод шаблонного класса не может быть виртуальным.

− Тип должен быть известен на этапе компиляции.

\+ Не нужно наследовать типы от единого класса-интерфейса.

Виртуальный деструктор

- Указатель на объект базового класса содержит адрес объекта наследника:

```
Collection* data = new LinkedList { 1, 2, 3 };
…
delete data;
```

- Какой деструктор должен быть вызван?
  - «Очевидно», LinkedList.
  - Компилятор не может знать, на что указывает data.
- **virtual** ~Collection () { } // Присутствует и виртуален.
- ~LinkedList () { clear(); } // Виртуален и полезен.
- У каждого класса должен быть виртуальный деструктор.

Наследование реализации

```
struct Capacitor : public Component
{
    Capacitor(double capacity);
    double get_capacity() const;
    double update(double voltage, double dt) override;
};
```

```
struct VariableCapacitor : public Capacitor
{
    VariableCapacitor(double min_C, double max_C);
    void set_capacity(double capacity);
    // double update(double voltage, double dt);
};
```

Защищенные члены класса

Inheritance is the base class of evil!

Sean Parent

- Спецификатор доступа: **protected**.
- Недоступны извне класса (как **private**).
- Доступны в методах классов-наследников.

```
class Component
{
public:
    double get_current() const;
protected:
    void set_current(double current);
};
```

```
class Resistor : public Component
{
public:
    void update(double V, double dt)
    {
        set_current(V / R);
    }
};
```

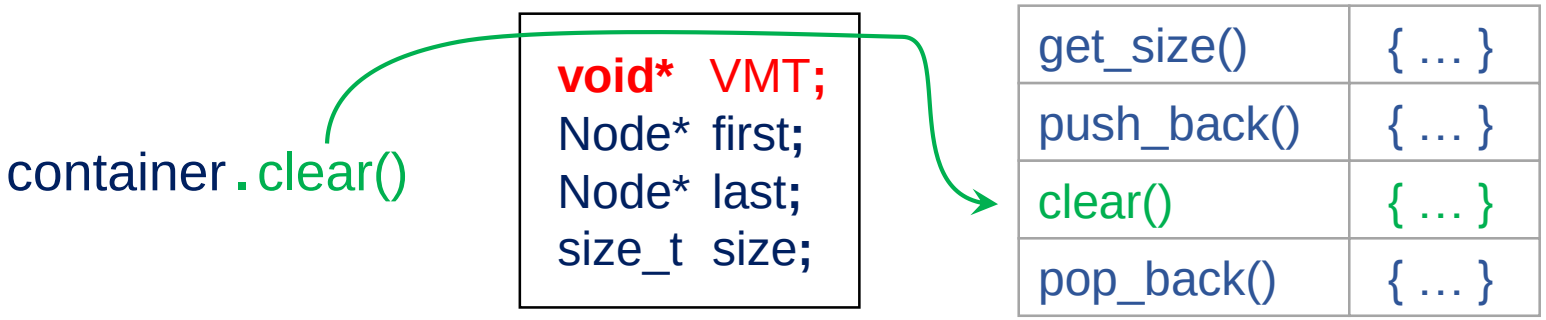
Приведение «вниз» (downcast)

- Container* container = **new** …;
  auto linked_list = **dynamic_cast**<LinkedList*>(container);
  if (linked_list)
  linked_list->find_node_at(…);
  else
  // container указывал не на LinkedList или его наследника
- Версия для ссылок возбуждает std::bad_cast
  при невозможности приведения.
- **N. B.:** грамотно спроектированная программа
  в приведении типов не нуждается…

  …и не существует.

Накладные расходы на позднее связывание

- Позднее связывание ≈ вызов виртуальных методов.
- Вызов виртуального метода дольше обычного.
  - Это важно только при «тонкой» оптимизации.
- Статический полиморфизм «бесплатен».
- Наличие виртуальных методов увеличивает размер объекта.



Литература к лекции

- *Programming: Principles and Practices Using C++:*
  - глава 9, раздел 9.7:
    - проектирование интерфейса класса (**public**-части);
    - неизменяющие методы;
  - глава 14:
    - интерфейсы (абстрактные классы);
    - пример иерархии наследования (GUI).
- *C++ Primer:*
  - часть III «Tools for Class Authors»:
    - глава 13 — копирование и перемещение;
    - глава 15 — наследование, интерфейсы, виртуальные методы.