C++
Лекция 6
Правило трех

RAII

- Захват ресурса есть инициализация.
  - **Конструктор**
    Объект создается и получает доступ к ресурсы.
  - **Деструктор**
    Объект уничтожается, доступ к ресурсы освобождается.

Деструкторы

- Что делает конструктор?
  - В деструкторе у объекта есть возможность выполнить некоторые действия перед уничтожением
  - Порядок уничтожения:
    - выполняется тело функции деструктора
    - деструкторы для объектов нестатических членов вызываются в порядке, обратном порядку их появления в объявлении класса
    - деструкторы для базовых классов вызываются в порядке, обратном порядку их объявления

Вызов деструктора

- Объект, предоставленный с использованием оператора **new**, можно явно освободить с использованием оператора **delete**.
- Локальный объект выходит за пределы области видимости.
- Время существования временного объекта заканчивается.
- Программа заканчивается, глобальные или статические объекты продолжают существовать.
- Деструктор явно вызываться с использованием полного имени функции деструктора.

UnsortedSet

```
template <typename T>
class UnsortedSet {
public:
  UnsortedSet()
    : elts(new T[DEFAULT_CAPACITY]),
      capacity(DEFAULT_CAPACITY),
      elts_size(0) {}

  ~UnsortedSet() {
    delete[] elts;
  }
  ...

private:
  int *elts;
  int capacity;
  int elts_size;
};
```

Выделение памяти в конструкторе. elts указатель.

Освобождение памяти в деструкторе.

Деструкторы и полиморфизм

```
class Player {
public:
  ...
  ~Player() {}
  ...
};
```

Так как конструктор не виртуальный, при уничтожении всех дочерних объектов будет вызываться этот!

```
class SimplePlayer : public Player {
  vector<Card> hand;
public:
  ...
  ~SimplePlayer() {}
  ...
};
```

```
int main() {
  Player *player = new SimplePlayer(...);
  ... //
  delete player;
}
```

Что-то не так..

Статический тип Player.

Деструкторы и полиморфизм

```
class Player {
public:
  ...
  virtual ~Player() {}
  ...
};
```

```
class SimplePlayer : public Player {
  vector<Card> hand;
public:
  ...
 virtual ~SimplePlayer() {}
  ...
};
```

В классах подтипах деструктор всегда должен быть виртуальный.

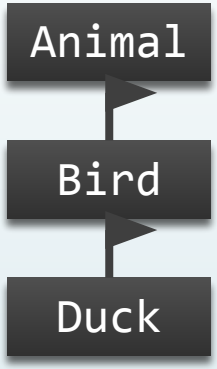
```
int main() {
  Player *player = new SimplePlayer(...);
  ... //
  delete player;
}
```

То что нужно

Динамический тип SimplePlayer.

Наследование: конструкторы и деструкторы

- Когда создается или уничтожается объект дочернего класса, вызывается мн-во конструкторов и деструкторов
  - По одному на каждый уровень иерархии наследования.
- **Конструкторы сверху-вниз**.




```
int main() {
  Duck d("Scrooge");  // Animal ctor, Bird ctor, Duck ctor
  Bird b("Big Bird"); // Animal ctor, Bird ctor
  ...
```

- **Деструкторы снизу-вверх.**

```
  ...
  // b dies: Bird dtor, Animal dtor
  // d dies: Duck dtor, Bird dtor, Animal dtor
};
```

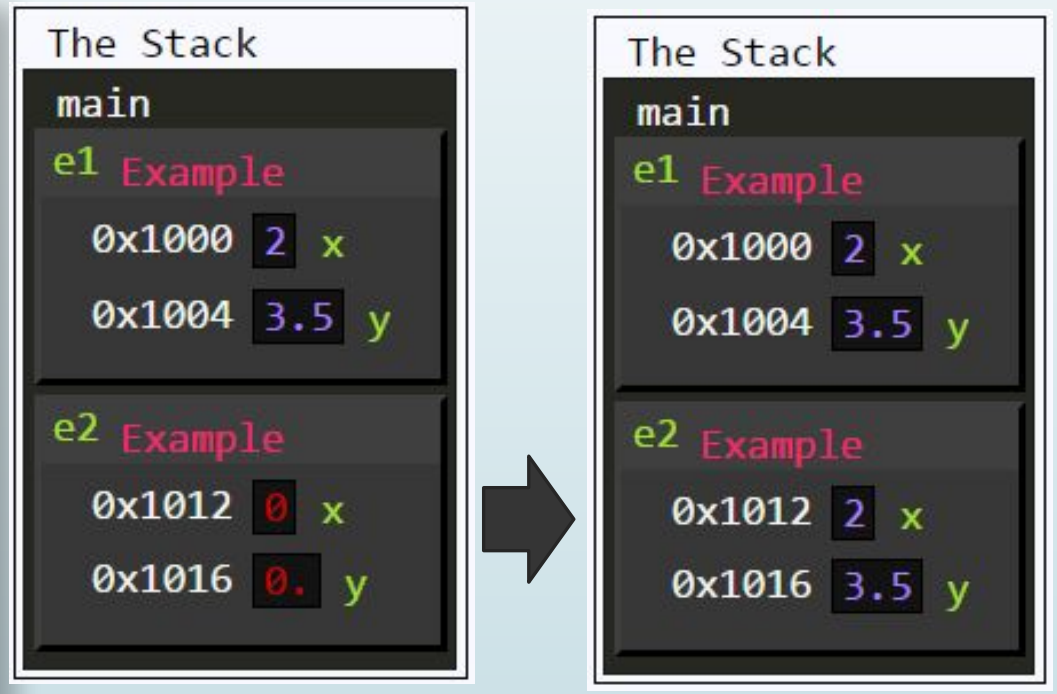
Кое-что еще...

```
int main() {
  UnsortedSet<int> s1;
  UnsortedSet<int> s2;
  s1.insert(2);
  s1.insert(3);

  s2 = s1;
  cout << s1 << endl; // выведет {2, 3}
  cout << s2 << endl; // выведет {2, 3}

  s2.insert(4);

  cout << s2 << endl;
  // выведет {2, 3, 4}, ок

  cout << s1 << endl; // ?
}
```

Копирование объектов

- По умолчанию копирование объектов происходит непосредственно копированием каждого из членов объекта.

```
class Example {
public:
  int x;
  double y;
};

int main() {
  Example e1;
  e1.x = 2; e1.y = 3.5;

  Example e2 = e1;
}
```

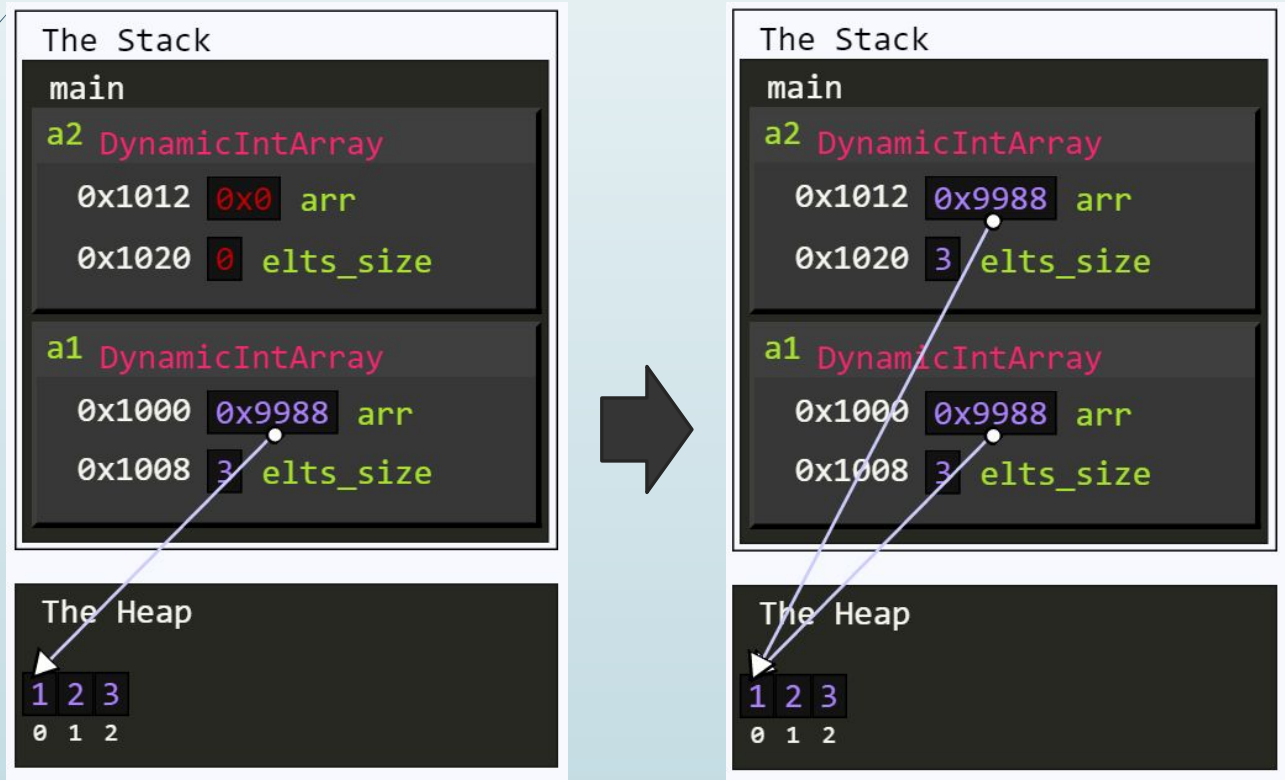


Что происходит при копировании?

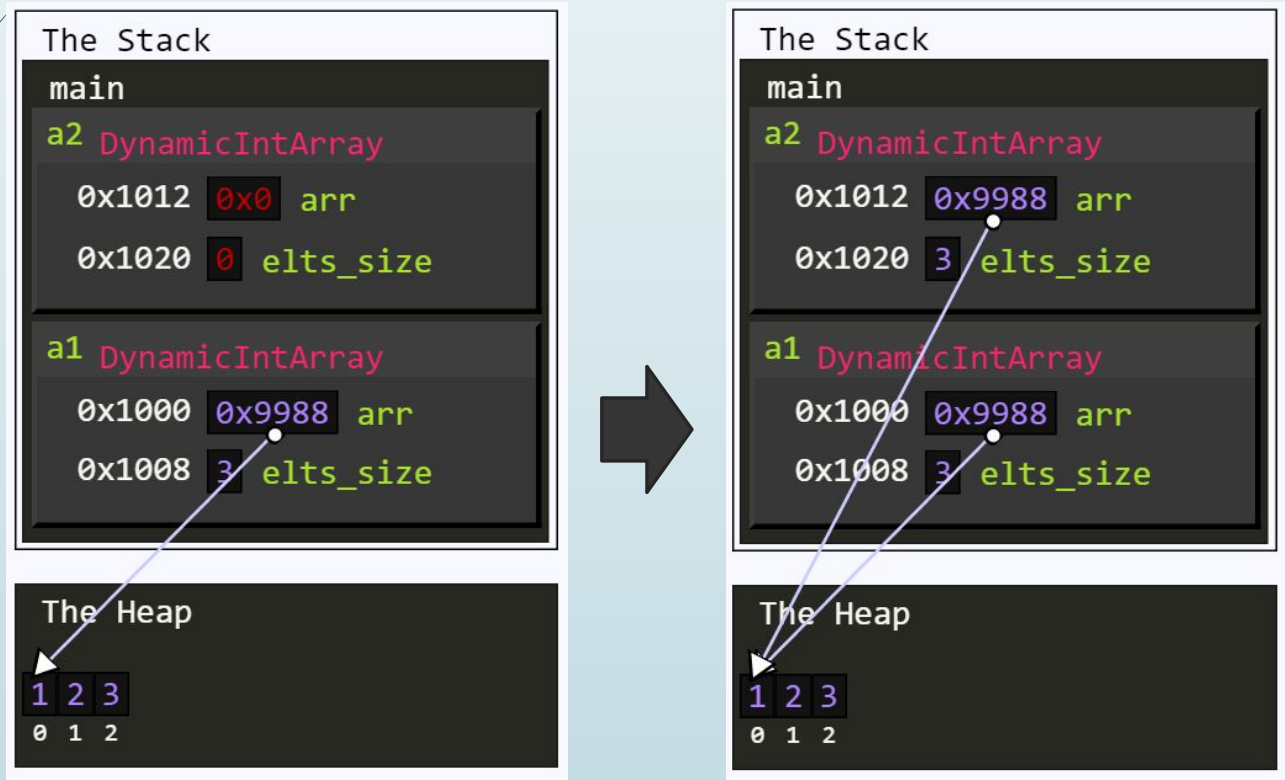
```
int main() {
  UnsortedSet<int> s1;
  UnsortedSet<int> s2;
  s1.insert(2);
  s1.insert(3);

  s2 = s1;
  cout << s1 << endl;
  cout << s2 << endl;

  //
  s2.insert(4);

  cout << s2 << endl;
  cout << s1 << endl;
}
```

Поверхностное копирование

- Поверхностное копирование не работает с объектами, которые используются не на прямую.



- Просто дублирует биты из переменных.
- Вместо данных из динамической памяти, копируется адреса на них.
- Появляется несколько объектов, указывающих на одну область памяти.
- При изменении этой области через один объект, она также изменится и в другом, что в большинстве случаев является нежелательным поведением.



Два вида копирования

- Копирование объектов происходит в двух случаях:
  - **Инициализация (передача параметра)**

```
int func(DynamicIntArray a); // передача по значению
int main() {
  DynamicIntArray a1(3);
  DynamicIntArray a2(a1);  // инициализация a2 копией a1
  DynamicIntArray a3 = a1; // инициализация a3 копией a1
  func(a1);        // инициализация параметра копией a1
}
```

Конструктор копий

- Когда один объект **инициализируется** другим - вызывается **конструктор копий**.

```
class DynamicIntArray {
public:
  DynamicIntArray(const DynamicIntArray &other);
};
```

- Если конструктор копий не создан явно, компилятор создает его сам
  - и просто копирует все поля...
  - Для `DynamicIntArray` это было бы что-то вроде...

```
class DynamicIntArray {
public:
  DynamicIntArray(const DynamicIntArray &other)
    : arr(other.arr),
      capacity(other.capacity) {}
};
```

Конструктор копий

- Почему параметр передается по ссылке?
-

```
class DynamicIntArray {
public:
  DynamicIntArray(const DynamicIntArray &other);
};
```

YES

- Что будет если передать по значению?
-

```
class DynamicIntArray {
public:
  DynamicIntArray(const DynamicIntArray other);
};
```

NO

  - Бесконечная рекурсия!

Два вида копирования

- Копирование объектов происходит в двух случаях:
  - Инициализация (передача параметра)

```
int func(DynamicIntArray a); // pass by value
int main() {
  DynamicIntArray a1(3);
  DynamicIntArray a2(a1);  // инициализация a2 копией a1
  DynamicIntArray a3 = a1; // инициализация a3 копией a1
  func(a1);        // инициализация параметра копией a1
}
```

  - **Присваивание**

```
int main() {
  DynamicIntArray a1(3);
  DynamicIntArray a2(4);
  a2 = a1; // a2 копия a1
}
```

Оператор присваивания

- Чтобы присвоить значение одного объекта другому используется оператор присваивания.

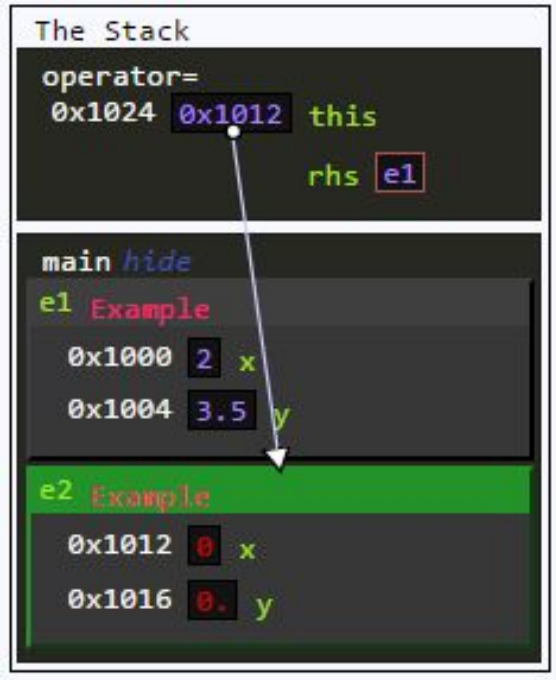
```
class DynamicIntArray {
public:
  DynamicIntArray & operator=(const DynamicIntArray &rhs);
};
```

- Если не реализовать свою версию компилятор создаст свою,
  - которая просто скопирует поля.
  - Для `DynamicIntArray` получится что-то вроде…

```
class DynamicIntArray {
public:
  DynamicIntArray & operator=(const DynamicIntArray &rhs) {
    arr = rhs.arr;
    capacity = rhs.capacity;
    return *this;
  }
};
```

Оператор присваивания

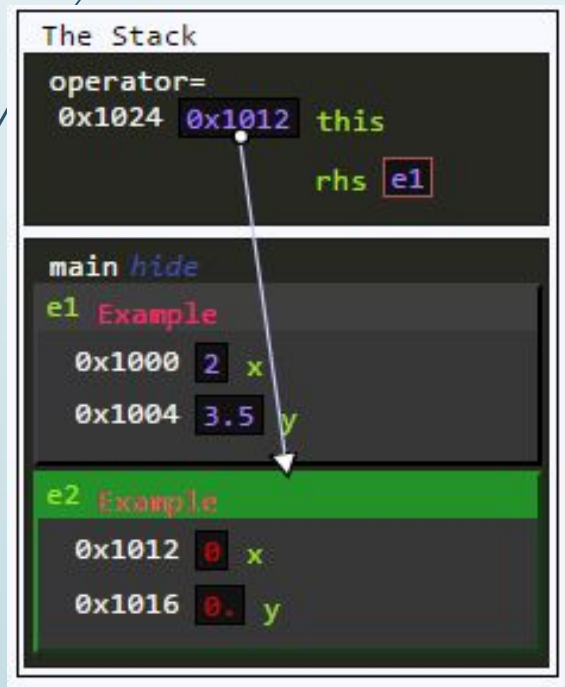
- Перегруженный оператор присваивания должен быть методом класса.
- При вызове функции…
  - **this** слева от знака = .
  - параметр справа от знака =.




```
                                    @e1
Example & operator=(const Example &rhs){
  x = rhs.x;
  y = rhs.y;
  return *this;
}
int main(){
  Example e1;
  e1.x = 2;
  e1.y = 3.5;
  Example e2;
  e2 = e1;
}
```

return *this

- Semantically, an assignment expression is supposed to evaluate back into its left hand side (the object assigned to).
- Example: `cout << (x = 3) << endl;`
- We first assign `3` into `x`, but then the whole thing in parentheses turns back into `x`, which is then printed.




```
                                        @e1
Example & operator=(const Example &rhs){
  x = rhs.x;
  y = rhs.y;
  return *this;
}
int main(){
  Example e1;
  e1.x = 2;
  e1.y = 3.5;
  Example e2;
  e2 = e1;
}
```

*this is just the left hand side object itself!

Копнем глубже

- Иногда необходимо обеспечить собственную реализацию **конструктора копий** и **оператора присваивания**.
  - По умолчанию происходит теневое копирование и часто это не то что нужно.

- `DynamicIntArray` и `UnsortedSet` примеры того когда теневое копирование не работает.
  - В данном случае копирование должно быть полным.
  - Копии должны иметь собственный динамический массив, а не указатель на чужой

Exercise: Пользовательский конструктор копий

- Нам нужна **полная копия** UnsortedSet.

```
template <typename T>
class UnsortedSet {
public:
  UnsortedSet(const UnsortedSet &other)

 {




  }
};
```

Solution: Пользовательский конструктор копий

- Нам нужна **полная копия** UnsortedSet.

```
template <typename T>
class UnsortedSet {
public:
  UnsortedSet(const UnsortedSet &other)
    : elts(new T[other.capacity]),
      capacity(other.capacity),
      elts_size(other.elts_size) {

    for (int i = 0; i < elts_size; ++i) {
      elts[i] = other.elts[i];
    }
  }
};
```

Выделяем память.

Можем обращаться к приватным полям. Мы все еще в области видимости UnsortedSet

Копируем каждый элемент.

Пользовательский оператор присваивания

- Нам нужна **полная копия** UnsortedSet.

```
template <typename T>
class UnsortedSet {
public:
  UnsortedSet & operator=(const UnsortedSet &rhs) {
    if (this == &rhs) { return *this; }

    delete[] elts;
    elts = new T[rhs.capacity];
    capacity = rhs.capacity;
    elts_size = rhs.elts_size;

    for (int i = 0; i < elts_size; ++i) {
      elts[i] = rhs.elts[i];
    }

    return *this;
  }
};
```

Проверка

Удаляем старый

Создаем новый

Копируем элементы

Возвращаем ссылку на объект

Закон большой тройки

- Большая тройка:
  - Деструктор
  - Конструктор копий
  - Оператор присваивания

- Закон большой тройки:
  - Если необходимо реализовать хотя бы что-то из большой тройки…

    …то все остальное также необходимо реализовать.

Неявное определение

- Все объекты имеют большую тройку.
  - Если не реализованы пользовательские версии, то компилятор создает их по умолчанию (неявное определение).

- Неявный конструктор
  - Не заботится об очистки памяти
- Неявный конструктор копий
  - Теневое копирование
- Неявный оператор присваивания
  - Теневое копирование

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

- 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`