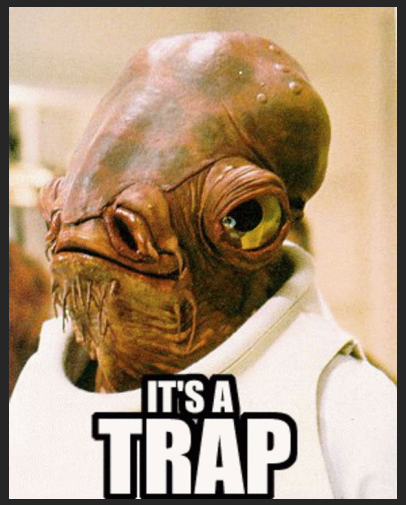
Что не так?

```
int *pointerToThree() {
  int x = 3;
  return &x;
}

void func() {
  int y = 5;
}

int main() {
  int *z = pointerToThree();
  func();
  if (*z != 3) {
    cout << "EXPLODE!" << endl;
  }
}
```



На совести программиста!

Динамичкая память

- Объекты создаются с помощью `new`.
- Объекты уничтожаются с помощью `delete`.
- За выделением и освобождением надо следить!

Оператор new

- Создание динамического объекта.

```
int *ptr = new int(3);
```

Что такое куча (Heap)?

- Куча - хранилище памяти, также расположенное в ОЗУ, которое допускает динамическое выделение памяти и не работает по принципу стека: это просто склад для ваших переменных
- Каждый объект имеет свое время жизни.
- `new` и `delete` операторы для работы с кучей.

- Сколько объектов создастся?

```
void func1(int x) {
  int *ptr = new int(x);
  *ptr = *ptr + 1;
  cout << *ptr << endl;
}

int main() {
  int *p = new int(-1);
  for (int i = 0; i < 3; ++i) {
    func1(i);
  }
}
```

Оператор delete

- delete освобождает выделенную память.

```
int *ptr = new int(3);
delete ptr;
```

```
void func1(int x) {
  int *ptr = new int(x);
  *ptr = *ptr + 1;
  cout << *ptr << endl;
  delete ptr;
}

int main() {
  int *p = new int(-1);
  for (int i = 0; i < 3; ++i) {
    func1(i);
  }
  delete p;
}
```

Динамические массивы

- Размер выделяемой памяти необходимо знать на этапе компиляции.

```
int main() {
  const int NUM_ELEMENTS = 10;
  int arr[NUM_ELEMENTS];
}
```

- Размер может задаваться во время выполнения.

```
int main() {
  cout << "How many elements? ";
  int howMany;
  cin >> howMany;
  int *arrPtr = new int[howMany];
}
```

```
int main() {
  cout << "Количество элементов? ";
  int howMany;
  cin >> howMany;
  int *arrPtr = new int[howMany];
  for (int i = 0; i < howMany; ++i) {
    arrPtr[i] = 42; // каждый элемент 42
  }
}
```

Динамические массивы и delete[]

```
int main() {
  ...
  int *ptr1 = new int(42);
  int *ptr2 = new int[howMany];

  delete ptr1;
  delete[] ptr2;
}
```

- Что будет выведено?

```
class MyClass {
public:
  MyClass (string s_in)
    : s(s_in) {
    cout << "MyClass ctor "
         << s << endl;
  }

  ~MyClass() {
    cout << "MyClass dtor "
         << s << endl;
  }

private:
  string s;
};
```

```
void func() {
  MyClass m3("local in func");
}

int main() {
  MyClass *mPtr;
  func();
  mPtr = new MyClass("dynamic");
  MyClass m4("local in main");
  delete mPtr;
  func();
}
```

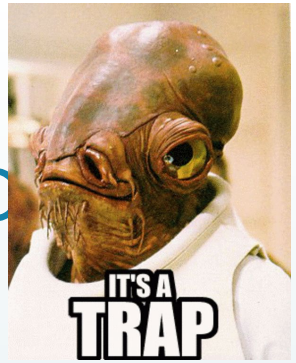
- Что будет выведено?

```
class MyClass {
public:
  MyClass (string s_in)
    : s(s_in) {
    cout << "MyClass ctor "
         << s << endl;
  }

  ~MyClass() {
    cout << "MyClass dtor "
         << s << endl;
  }

private:
  string s;
};
```

```
void func() {
  MyClass m3("local in func");
}

int main() {
  MyClass *mPtr;
  func();
  mPtr = new MyClass("dynamic");
  MyClass m4("local in main");
  delete mPtr;
  func();
}
```

```
MyClass ctor local in func
MyClass dtor local in func
MyClass ctor dynamic
MyClass ctor local in main
MyClass dtor dynamic
MyClass ctor local in func
MyClass dtor local in func
MyClass dtor local in main
```

Проблемы динамическо памяти



- **Утечки памяти (Memory Leaks)**

  Память выделили, но не используем и не освобождаем

- **Зависшая память**

  Когда потеряли адрес объекта в куче, следовательно не может освободить память, память утекла

Пример: Утечка памяти

- Что не так?

```
void example1() {
  int x = 0;
  double *ptr = new double(3.0);

}

int main() {
  example1();
  ...
}
```

Потеряли delete.

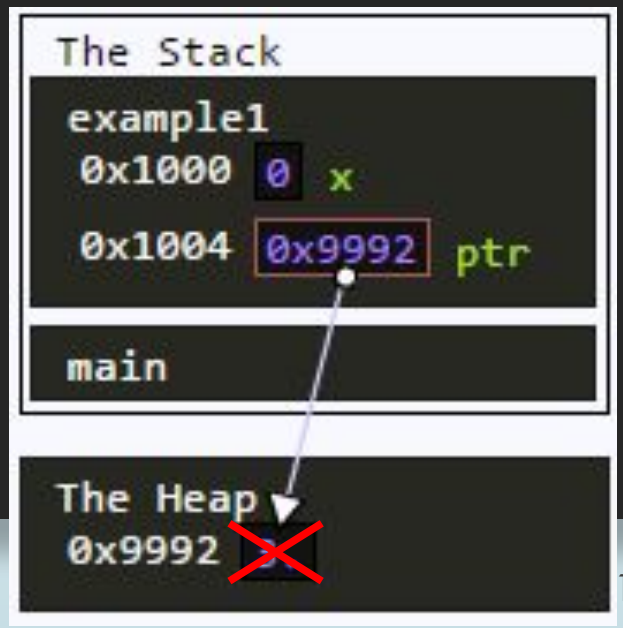
```
void example1() {
  int x = 0;
  double *ptr = new double(3.0);

}

int main() {
  example1();
  ...
}
```

Пропущен `delete`.
Объект все еще существует.

Созданный объект нигде не используется и мы не сможет получить доступ к указателю чтобы освободить память.

delete

```
void example1() {
   int x = 0;
   double *ptr = new double(3.0);
   delete ptr;
}

int main() {
   example1();
   ...
}
```



Утечки памяти

- Где здесь утечка?

```
void example2() {
  // ptr указатель на int
  int *ptr = 0;

  // создадим 5 переменных и выведем
  for (int i = 0; i < 3; ++i) {
    ptr = new int(i);
    cout << *ptr << endl;
  }

  // а теперь просто освободим указатель >:)
  delete ptr;
}
```

```
void example2() {

  int *ptr = 0;

  for (int i = 0; i < 3; ++i) {
    ptr = new int(i);
    cout << *ptr << endl;
  }

  // а теперь просто освободим указатель
  delete ptr;
}
```

Память выделена под 5 объектов

Уничтожаем 1

Exercise: delete

- Что не так?

```
void example3() {
  int x = 0;
  int *ptr1 = new int(1);
  int *ptr2 = new int(2);
  ptr2 = ptr1;

  delete ptr1;
  delete ptr2;
}
```

delete

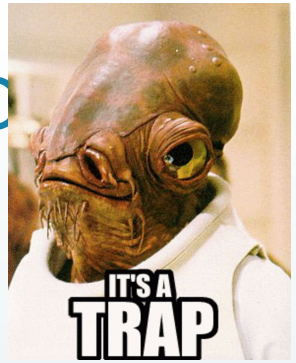
- Что не так?

```
void example3() {
  int x = 0;
  int *ptr1 = new int(1);
  int *ptr2 = new int(2);
  ptr2 = ptr1;

  delete ptr1;
  delete ptr2;
}
```

Переменная в которой хранилось 2 стала зависшей.

Уничтожаем один и тот же объект дважды:(.

Проблемы динамическо памяти



- **Утечки памяти (Memory Leaks)**

  Память выделили, но не используем и не освобождаем

- **Зависшая память**

  Когда потеряли адрес объекта в куче, следовательно не может освободить память, память утекла

- **Двойное освобождение**

  Пытаемся освободить одну и ту же память несколько раз.

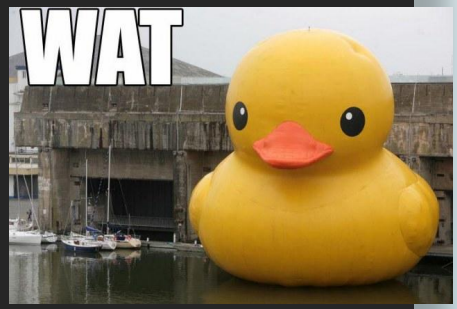
Снова `delete`...

- Если попытаться удалить объект дважды, то программа может перестать работать
- Если удалить null pointer (или `nullptr`), ничего не случиться.
- После того как для объекта вызван оператор `delete` можно продолжать использовать объект, и даже может казаться что все работать нормально
  - `delete` просто сообщает о том что освобожденный участок памяти может использоваться снова.
  - Использование удаленного объекта приводит к неопределенному поведению.

Объекты зомби

- Что случится?

```
void example4() {
  int x = 0;
  int *ptr = new int(42);
  cout << *ptr << endl;
  delete ptr;

  int *ptr1 = new int(3);
  cout << *ptr << endl;
  delete ptr1;
}
```



Объекты зомби

- Добавим защиты

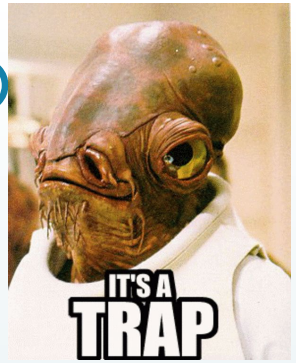
```
void example4() {
  int x = 0;
  int *ptr = new int(42);
  cout << *ptr << endl;
  delete ptr; ptr = nullptr;

  int *ptr1 = new int(3);
  cout << *ptr << endl;
  delete ptr1;
}
```

Теперь программа завершиться с ошибкой. Но такую ошибку намного проще обнаружить!

Зависшие указатели

- После вызова `delete` для указателя, указатель все еще хранит адрес удаленного объекта.
- Легко случайно использовать такой объект!
  - Такие ошибки сложно обнаружить.
    - Иногда программа будет выполняться успешно.
    - Иногда нет.
    - Причина ошибки не очевидна.
- После удаления присваивайте указателям nullptr.

Проблемы динамическо памяти



- **Утечки памяти (Memory Leaks)**

  Память выделили, но не используем и не освобождаем

- **Зависшая память**

  Когда потеряли адрес объекта в куче, следовательно не может освободить память, память утекла

- **Двойное освобождение**

  Пытаемся освободить одну и ту же память несколько раз.

- **Зависшие указатели**

  Использование адреса уже освобожденного объекта.

Exercise: new and delete.

- Где уточки памяти?
  - Подсказка: Их тут много.

```
int *func(int x) {
  int *y = new int(x);
  y = new int[x];
  return y;
}

int main() {
  int *a = func(5);
  int *b = a;
  delete b;
  cout << a[2] << endl;
}
```

Solution: new and delete.

- Где уточки памяти?
  - Подсказка: Их тут много.

```
int *func(int x) {
  int *y = new int(x);
  y = new int[x];
  return y;
}

int main() {
  int *a = func(5);
  int *b = a;
  delete b;
  cout << a[2] << endl;
}
```

Как похоже...

Зависшая память.

Нужно удалить массив.

Память освободили, но используем .

RAII: Resource Acquisition Is Initialization

- Проблема:
  Компилятор плохой менеджер ресурсов.
  - Используем **new** для создания объектов, компилятор не может за нас освободить память.
- Наблюдение:
  Компилятор умеет следить за локальными объектами.
  - Автоматически выделяем память.
  - Удаляет когда время жизни завершается.
- Идея:
  Обернуть работу с памятью в класс и использовать объекты класса для доступа к ресурсам
  - При создании объекта ресурс захватывается.
  - При удалении объекта ресурс освобождается.

Динамический массив

```
class DynamicIntArray {
private:
  int *arr;
  int elts_size;

public:
  DynamicIntArray(int size_in)
    : arr(new int[size_in]), elts_size(size_in) {}

  ~DynamicIntArray() {
    delete[] arr;
  }

  int size() const {
    return elts_size;
  }

  int & operator[](int i) { return arr[i]; }
  const int & operator[](int i) const { return arr[i]; }

};
```

Память на массив выделяется в конст-ре.

Деструктор освобождает массив.

Динамический массив

```
int main() {
  // Create an array with random length
  DynamicIntArray dArr(randInt(1, 11));

  // Fill with integers in [0, size)
  for (int i = 0; i < dArr.size(); ++i) {
    dArr[i] = randInt(0, 100);
  }

  // Use the array for something
  // ...

  cout << "main is ending now..." << endl;

}
```

When dArr is created, its constructor allocates the dynamic array.

We don't have to worry about new or delete here at all!

dArr goes out of scope here and dies. Its destructor cleans up the dynamic array.

Обновим UnsortedSet

```
template <typename T>
class UnsortedSet {
public:
  // Максимальный размер мн-ва.
  static const int ELTS_CAPACITY = 10;

  ...

  virtual void insert(T v) = 0;

  ...
};
```

Избавимся от фиксированного размера

Идея

- Динамически выделять массив…
- Нужен размер побольше? Создаем новый, нужного размера, копируем элементы из старого и удаляем старый!

Использование дин. массив

```
template <typename T>
class UnsortedSet {
...
privte:
  T *elts;

  int capacity;

  int elts_size;

  // Функция увеличения массива в два раза
  void grow();
};
```

Вместо статического массива создадим указатель.

Время жизни массива теперь зависит от объекта UnsortedSet.

Текущий размер массива. Теперь у каждого экземпляра свой размер UnsortedSets.

Количество элементов в множестве.

Используем grow( ) когда необходимо

```
template <typename T>
class UnsortedSet {
public:
  virtual void insert(T v) {
    if (contains(v)) { return; }
    if (elts_size == capacity)) { grow(); }

    elts[elts_size] = v;
    ++elts_size;
  }

private:
  T *elts;
  int capacity;
  int elts_size;


  void grow();
};
```

Когда место в массиве заканчивается выделяем еще

Функция grow()

```
template <typename T>
class UnsortedSet {
...
private:
  T *elts;
  int capacity;
  int elts_size;
  //Функция увеличения массива в два раза
  void grow() {



  }
};
```

По порядку…

1. Создаем новый массив в два раза больше
2. Копируем элементы
3. Обновляем текущий размер
4. Уничтожаем старый
5. Обновляем указатель `elts`

Функция grow()

```
template <typename T>
class UnsortedSet {
...
private:
  T *elts;
  int capacity;
  int elts_size;

  // Функция увеличения массива в два раза
  void grow() {
    T *newArr = new T[2 * capacity];
    for (int i = 0; i < elts_size; ++i) {
      newArr[i] = elts[i];
    }
    capacity *= 2;
    delete[] elts;
    elts = newArr;
  }
};
```

По порядку…

1. Создаем новый массив в два раза больше
2. Копируем элементы
3. Обновляем текущий размер
4. Уничтожаем старый
5. Обновляем указатель `elts` †

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

UnsortedSet Деструктор

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

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

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

Динамически выделяем память.

Освобождаем память.