Типы связей между объектами. Композиция

Для реализации композиции объект и часть должны иметь следующие отношения:
— часть является частью объекта (класса);
— часть может принадлежать только одному объекту (классу) в моменте;
— часть существует, управляемая объектом (классом);
— часть не знает о существовании объекта (класса).

Композиции являются одними из самых простых типов отношений для реализации на языке C++.

Это обычные структуры или классы с обычными членами.

Поскольку члены существуют непосредственно как части структур/классов, то их продолжительность жизни напрямую зависит от продолжительности жизни объектов этих структур/классов.

Типы связей между объектами. Агрегация

Для реализации агрегации целое и его части должны соответствовать следующим отношениям:

— часть является частью целого (класса);
— часть может принадлежать более чем одному целому (классу) в моменте;
— часть существует, не управляемая целым (классом);
— часть не знает о существовании целого (класса).

В композиции:
— используются обычные переменные-члены;
— используются указатели, если класс реализовывает собственное управление памятью (происходит динамическое выделение/освобождение памяти);
— класс ответственный за создание/уничтожение своих частей.

В агрегации:
— используются указатели/ссылки, которые указывают/ссылаются на части вне класса;
—класс не несет ответственности за создание/уничтожение своих частей.

Типы связей между объектами. Ассоциация

В ассоциации два несвязанных объекта должны соответствовать следующим отношениям:

— первый объект не связан со вторым объектом (классом);

— первый объект может принадлежать одновременно сразу нескольким объектам (классам);

— первый объект существует, не управляемый вторым объектом (классом);

— первый объект может знать или не знать о существовании второго объекта (класса).

Типы связей между объектами

Свойства	Композиция	Агрегация	Ассоциация
Отношения	Части-целое	Части-целое	Объекты не связаны между собой
Члены могут принадлежать одновременно сразу нескольким классам	Нет	Да	Да
Существование членов управляется классами	Да	Нет	Нет
Вид отношений	Однонаправленные	Однонаправленные	Однонаправленные или двунаправленные
Тип отношений	«Часть чего-то»	«Имеет»	«Использует»

Наследование

Основные понятия

Наследование – это такое отношение между классами, когда один класс повторяет структуру и поведение другого класса (одиночное наследование) или других (множественное наследование) классов.

Основные понятия

Класс, поведение и структура которого наследуется называется базовым (родительским) классом, а класс, который наследует – производным или дочерним.

В производном классе структура и поведение базового класса дополняются и переопределяются. Производный класс является уточнением базового класса.

«Лакмусовая бумажка» наследования – обратная проверка; так, если B не есть A, то B не стоит производить от A.

Основные понятия

В C++ наследование используется в двух целях:

1. Наследование интерфейсов

2. Наследование структуры и реализации класса

Одиночное наследование

Наследование должно использоваться только для выражения общности объектов. Когда выполняется правило «является». Нежелательно использовать наследование для компоновки объектов. Наследование есть классификация.

Одиночное наследование

```
class Shape {
// …
};

class Circle final : public Shape {
// …
};
```

В данном случае указывается, что класс Circle наследует классу Shape. Наследование означает, что производный класс повторяет структуру базового класса и наследует все его методы.

Спецификаторы доступа при наследовании

Ключ доступа	Спецификатор в базовом классе	Доступ в производном классе
private	private protected public	нет private private
protected	private protected public	нет protected protected
public	private protected public	нет protected public

Отличия структур и объединений от классов

Структуры	Классы	Объединения
struct	class	union
Доступ по умолчанию: public	Доступ по умолчанию: private	Доступ по умолчанию: public
		Используется только один элемент за раз
		Инициализируется только первое поле

Одиночное наследование

Базовый класс Shape (фигура) содержит координаты фигуры и метод перемещения фигуры в заданную позицию. Также он содержит перегруженный конструктор, создающий фигуру в начале координат или в точке, указанной пользователем

Одиночное наследование

```
class Shape {
   vec2d pos;
public:
  Shape() noexcept;
  explicit Shape(const vec2& aPos) noexcept;
  void MoveTo(const vec2d& newPos) noexcept;
  virtual ~Shape() = default;
};

Shape::Shape() : pos(0.0,0.0) noexcept{}
Shape::Shape(const vec2& aPos) : pos(aPos) noexcept{}
void Shape::MoveTo(const vec2d& newPos) noexcept{
  pos = newPos;
}
```

Одиночное наследование

Класс Circle (окружность) наследует классу Shape (фигура). Здесь выполняется правило «окружность является фигурой». Класс привносит новый член данных radius – атрибут, специфичный для окружности и два метода – Draw (нарисовать фигуру) и Erase (стереть фигуру).

Методы Draw и Erase не имеет смысла определять в классе Shape, поскольку не известно, как нарисовать или стереть абстрактную фигуру.

Одиночное наследование

```
class Circle final : public Shape {
  double radius;
 public:
  explicit Circle(double aRadius) noexcept;
  explicit Circle(double aRadius,const vec2d& aPos) noexcept;
  void Draw() const noexcept;
  void Erase() const noexcept;
};

Circle::Circle(double aRadius) noexcept : radius(aRadius) {}

Circle::Circle(double aRadius, const vec2d& aPos) noexcept :
  Shape(aPos),
  radius(aRadius)
{}
```

Одиночное наследование

При реализации конструктора, создающего окружность с заданным радиусом, инициализируется только член данных radius. Положение фигуры (окружности) в пространстве инициализируется по умолчанию конструктором класса Shape. То есть до создания объекта Circle вначале автоматически создаётся унаследованная от Shape часть и вызывается её конструктор.

В случае с конструктором, создающим окружность с заданным радиусом в заданных координатах, конструктор Shape вызывается явно с передачей ему требуемого положения в пространстве.

Виртуальные методы

Наследование интерфейсов

```
class Shape {
public:
  void Draw() const;
  void Erase() const;
  void MoveTo(const vec2d& newPos);
  virtual ~Shape() = default;
};

Shape *sh;
sh = new Circle(10.0);
sh->Draw();
```

В третьей строке вызывается метод Draw. Однако, в текущей реализации будет вызван метод Draw класса Shape (который мы определили как пустой), поскольку указатель имеет тип Shape* и компилятор считает, что он указывает на объект типа Shape, а не на созданный объект класса Circle. В реальности указатель, действительно, указывает на ту часть объекта Circle, которая унаследована от Shape.

Виртуальные функции

Необходимо указать компилятору, чтобы при указании на объект посредством указателя на объект базового класса, он учитывал при вызове метода тип реального объекта, который скрывается за указателем. Для этого служит механизм **виртуальных функций**.

Виртуальная функция (виртуальный метод) – это функция-член класса, которая вызывается с учётом типа объекта, для которого она была вызвана вне зависимости от типа указателя, через который она была вызвана.

Виртуальные функции

```
class Shape {
 public:
  virtual void Draw() const;
  virtual void Erase() const;
  virtual void MoveTo(const vec2d &newPos);
  virtual ~Shape() = default;
};

class Circle final : public Shape {
  double radius;
 public:
  void Draw() const noexcept override;
  void Erase() const noexcept override;
};
```

Виртуальные функции

Теперь будет вызван метод Draw класса Circle.

```
Shape *sh;
sh = new Circle(10.0);
sh->Draw();
```

Виртуальные функции

Виртуальные функции реализуют динамический полиморфизм в C++. В данном случае полиморфизм заключается в том, что в зависимости от типа объекта будет вызван свой метод (т.е. продемонстрировано различное поведение для различных объектов). А динамическим он является потому, что решение о том, какой метод должен быть вызван принимается в ходе выполнения программы, а не во время компиляции, как это происходит в случае перегруженных функций.

Виртуальные функции

Наследование интерфейсов позволяет оградить клиентский код от знания подробностей реализации иерархии наследования и составляющих её реальных объектов.

Например, при реализации метода ClearScreen, стирающего все объекты с экрана, клиентскому коду достаточно знать о существовании интерфейсного класса Shape и о том, что объекты, которые ему передаются, наследуют этом классу:

Виртуальные функции

```
extern int nShapes;
extern Shape *shapes[MaxShapes];

 void ClearScreen() {
  for (int i=0; i < nShapes; ++i)
    shapes[i]->Erase();
 }
```

Виртуальные функции

При этом реальные объекты могут быть произвольных типов:

```
int nShapes;
Shape *shapes[MaxShapes];

void CreateObjects() {
  nShapes=0;
  shapes[nShapes++] = new Circle(0.0);
  shapes[nShapes++] = new Rectangle(10.0,20.0);
}
```

Абстрактные классы

Абстрактные классы

Если для какой-то функции (метода) реализация не определена (не имеет смысла в данном классе), она может быть объявлена как «чисто виртуальная функция». Т.е. функция, которая не имеет реализации.

Класс, имеющий хотя бы один чисто виртуальный метод, называется абстрактным. Невозможно создавать объекты абстрактного класса. Для того, чтобы создать объект класса, все чисто виртуальные функции базовых классов должны быть переопределены.

Абстрактные классы

Чисто виртуальные функции позволяют явно указать на то, что данные методы для данного базового класса нереализуемы и что объекты такого класса создавать нельзя, поскольку сами по себе они не имеют смысла.

Чисто виртуальная функция обозначается символами = 0, следующими до точки с запятой, завершающими объявление функции.

Абстрактные классы

```
class Shape {
 public:
  virtual void Draw()  const = 0;
  virtual void Erase() const = 0;
  virtual void MoveTo(const vec2d& newPos);
  virtual ~Shape() = default;
};
```

Множественное наследование

Множественное наследование

Множественным наследованием называется ситуация, когда класс наследует более чем одному базовому классу. Множественное наследование может использоваться для наследования структуры и методов нескольких классов. Однако такое его применение не рекомендуется.

Основное применение множественного наследования – наследование нескольким интерфейсным классам с тем, чтобы обеспечить возможность использования объекта в различных контекстах.

Множественное наследование

Рассмотрим несколько базовых классов, определяющих взаимодействие с объектом, отображаемым на экране.

Класс KeyboardControlled представляет интерфейс объекта, который способен обрабатывать нажатия клавиш. Он предоставляет метод KeyPressed, который вызывается, когда необходимо сообщить объекту, что была нажата клавиша на клавиатуре.

Множественное наследование

```
class KeyboardControlled {
public:
 virtual void KeyPressed(int key_code) = 0;
};
```

Множественное наследование

Класс MouseControlled представляет интерфейс объекта, который способен обрабатывать перемещения мыши и изменения состояния её клавиш. Он предоставляет соответствующий набор методов.

Множественное наследование

```
class MouseControlled {
public:
 virtual void MouseMoved(int x, int y) = 0;
 virtual void ButtonPressed() = 0;
 virtual void ButtonReleased()= 0;
};
```

Множественное наследование

Класс, реализующий кнопку в диалоговом окне может наследовать обоим классам с тем, чтобы получить соответствующие интерфейсы и иметь возможность рассматриваться и как объект, обрабатывающий события мыши и как объект, обрабатывающий события клавиатуры.

Множественное наследование

```
class Button : public KeyboardControlled,
public MouseControlled {
 public:
  virtual void KeyPressed(int key_code);
  virtual void MouseMoved(int x, int y);
  virtual void ButtonPressed();
  virtual void ButtonReleased();
};
```