

Инкапсуляция и способы ее реализации.

Поведение класса

- Понятие инкапсуляции, интерфейса и реализации, принцип маскировки информации
- Состав интерфейса и реализации класса
- Уровни доступа
- Свойства и методы доступа к ним
- Перегрузка операций
- Специальные методы
- Нарушения инкапсуляции

Классификация методов

- Привязка к объекту/классу
 - динамические методы
 - статические методы
- Сигнатура и назначение метода полностью или частично определяются компилятором
 - специальные методы
 - операции
 - методы, определяемые программистами
- Возможность работы с константными объектами
 - константные методы
 - неконстантные методы

Классификация методов

- Назначение

- Порождающие методы
- Инициализирующие методы
- Завершающие методы
- Методы доступа к свойствам
- Методы преобразования
- Методы клонирования
- Контрактные методы

Статические методы класса

- Методы, которые применяются к отдельным объектам, называются **динамическими**
- Методы, которые относятся к классу в целом, называются **статическими**
- **Динамические** методы выполняются над **конкретным** объектом, а **статические** методы выполняются **безотносительно** какому-либо объекта
- **Статические методы** чаще всего используются для **порождения специфических экземпляров** класса

Правила использования статических элементов класса

- Статические методы могут обращаться непосредственно только к статическим свойствам и вызывать только другие статические методы класса
- Обращение к нестатическим свойствам и методам экземпляра класса возможно, если передать экземпляр класса как параметр метода
- Статические элементы доступны как через имя класса, так и через имя объекта

<имя класса>::<имя метода>

<имя объекта>.<имя метода>

Пример использования статического метода

- У класса `QDate` имеется **статический метод**, который возвращает текущую дату:

```
QDate currentDate ()
```

- Пример использования статического метода для порождения нового объекта:

```
// Создаем и запоминаем сегодняшнюю дату  
QDate cDate = QDate::currentDate();
```

Специальные методы класса

- Помимо обычных методов в классе имеется ряд **специальных**, сигнатура и назначение которых predeterminedены
- Любой класс обязательно имеет специальные методы. Если они не определяются программистом, то компилятор **генерирует** их **автоматически**

Специальные методы класса

- Конструктор с параметрами
- Конструктор по умолчанию (без параметров)
- Деструктор
- Конструктор копии
- Операция присваивания
- Операции выделения и освобождения памяти
- Операции приведения

Понятие конструктора

- **Конструктор** - это метод, который **автоматически** вызывается при **создании** объекта, т.е. после выделения памяти под поля объекта
- **Конструктор** чаще всего используется для **задания** **первичных** значений данным объекта и/или **выделения** динамической **памяти**, т.е. выполняет роль **инициализирующего** метода
- Таким образом, **создание** и **инициализация** являются **нераздельными** понятиями — одно без другого невозможно

Понятие конструктора

- В классе возможно объявление **нескольких** конструкторов. В этом случае конструкторы определяют **различные способы инициализации** объектов
- **Логика** конструктора полностью определяется **разработчиком** класса

Пример использования конструкторов

- У класса `QDate` имеются следующие конструкторы:

```
QDate ()
```

```
QDate ( int y, int m, int d )
```

- Следовательно, возможны следующие варианты создания переменных типа `QDate`:

```
QDate nullDate;           // «неопределенная» дата
```

```
QDate birthday(1990, 3, 23); // конкретная дата  
                               // рождения
```

Правила работы с конструкторами

- **Имя конструктора** должно совпадать с **именем класса**
- Конструктор **не может возвращать значение**, поэтому возвращаемый тип данных не указывается
- Если не определен ни один конструктор, то компилятор **генерирует конструктор по умолчанию**, не имеющий параметров и ничего не выполняющий

Правила работы с конструкторами

- Если объявлен конструктор с параметрами, то компилятор **не генерирует конструктор по умолчанию**, т.е. он отсутствует в классе
- **Нельзя** явно **вызывать** конструктор
- Конструктор **не наследуется** в том смысле, что он должен быть задан в производном классе **заново**
- Конструктор **не может быть виртуальным**

Список инициализации элементов конструктора

- Конструктор может содержать список инициализации элементов
- **Список инициализации** отделяется двоеточием от заголовка конструктора и содержит поля, разделенные запятыми
- Список инициализации **выполняется до** исполнения тела конструктора

Список инициализации элементов конструктора

- Синтаксис списка инициализации

<заголовок конструктора>

:<имя поля>(<значение>)

{

<тело конструктора>

}

Причины использования списка инициализации элементов

- Инициализация констант, принадлежащих классу
- Инициализация ссылок, принадлежащих классу
- Инициализация вложенных объектов (создаваемых не динамически), конструкторы которых требуют одного или нескольких параметров
- ~~Передача параметров конструкторам базовых классов~~

Пример конструктора

```
// Точка в двумерном пространстве
class TPoint2D
{
public:
    TPoint2D(int x, int y);

private:
    // Координаты в двумерном пространстве
    float f_x, f_y;
};

// Внимание!! Так как имеется конструктор с
// параметрами, то нельзя создать точку в
// неопределенной позиции – отсутствует
// конструктор по умолчанию
```

Пример конструктора

// Первый вариант реализации конструктора

```
TPoint2D::TPoint2D(int x, int y)
```

```
{
```

```
    f_x = x;
```

```
    f_y = y;
```

```
}
```

// Второй вариант реализации конструктора

```
TPoint2D::TPoint2D(int x, int y)
```

```
    : f_x(x), f_y(y)
```

```
{
```

```
}
```

Задание

Придумайте пример использования строки инициализации, когда без нее нельзя обойтись

Понятие деструктора

- Деструктор – это метод, который вызывается автоматически перед уничтожением объекта
- Таким образом, деструктор является завершающим методом
- Деструктор чаще всего используется для уничтожения динамически выделенной памяти
- В языках, имеющих сборщики мусора, понятие деструктора отсутствует

Правила работы с деструкторами

- **Имя** деструктора должно совпадать с именем класса и иметь префикс ~
- Деструктор **не может** иметь **параметров**
- Деструктор **не может возвращать** значение
- Компилятор **генерирует деструктор по умолчанию**, если он не определен в классе. Этот деструктор ничего не выполняет

Правила работы с деструкторами

- В отличие от конструктора деструктор, как и обычный метод, **может** быть **вызван** явно
- Деструктор **не наследуется**
- Деструктор может быть **виртуальным**

Пример деструктора

```
// Питон, состоящий из сегментов
class TPhyton
{
public:
    .....
    // Деструктор
    ~TPhyton();
    .....
private:
    // Сегменты, из которых состоит питон
    TPoint *fSegments[100];
    // Текущая длина питона
    unsigned int fLength;
};
```

Пример деструктора

```
// Деструктор
TPhyton::~~TPhyton()
{
    // Удаляем память, занимаемую сегментами
    for(int i = 0; i < fLength; i++)
    { delete fSegments[i]; }

    fLength= 0;
}
```

Понятие конструктора копии

- **Конструктор копии** является конструктором специального вида: он воспринимает в качестве **аргумента** константную **ссылку на объект** класса:

`const` <имя класса> &

или **простую** ссылку на объект:

<имя класса> &

- **Конструктор копии вызывается** в тех ситуациях, когда **создается новый объект**, который инициализируется значениями другого объекта

Ситуации использования конструктора копии

- Создание объектной переменной и ее инициализация другим объектом
- Передача объекта в качестве параметра функции
- Передача объекта в качестве возвращаемого значения функции

Примеры вызова конструктора копии

```
// Дублирование строки
QString duplicate(QString &str, int count)
{
    QString result;

    for(int i = 0; i < count; i++)
    { result += str; }

    return result; // вызов конструктора копии
}

void main(void)
{
    QString str("example");
    QString result = str; // вызов конструктора копии

    result = duplicate(str, 3);
}
```

Правила работы с конструктором копии

- Если конструктор копии не объявлен, то он создается компилятором **по умолчанию**
- Конструктор копии **по умолчанию** "создает" (в смысле инициализирует) **точную копию объекта** и, скорее всего, будет не пригодным для объектов, содержащих указатели и ссылки
- Конструктор копии **необходимо** объявлять, если объект использует **динамические** структуры данных

Пример конструктора копии

```
// Питон, состоящий из сегментов
class TPhyton
{
public:
    .....
    // Конструктор копии
    TPhyton(TPhyton &other);
    .....
private:
    // Сегменты, из которых состоит питон
    TPoint *fSegments[100];
    // Текущая длина питона
    unsigned int fLength;
};
```

Пример конструктора копии

```
// Конструктор копии
TPhyton::TPhyton(TPhyton &other)
{
    // Для нового питона выделяем память и
    // копируем туда значения сегментов
    fLength = other.fLength;

    fSegments = new TPoint[fLength];

    copySegments(fSegments, other.fSegments,
                 fLength);
}
```

Пример вызова конструктора копии

```
// Питон, созданный по умолчанию  
TPhyton defaultPhyton;  
  
// Создание питонов-копий  
// (вызывается конструктор копирования)  
TPhyton Phyton2(defaultPhyton);  
TPhyton Phyton3 = defaultPhyton;
```

Понятие операции присваивания

- Операция присваивания является методом с именем `operator =`, который воспринимает единственный аргумент типа

`const` <имя класса> & или <тип класса> &

- Операция присваивания вызывается компилятором, когда объекту присваивается значение другого объекта

- Операция присваивания вызывается, если изменяется содержимое существующего объекта, а конструктор копии, если создается новый объект, который инициализируется значениями другого объекта

Правила работы с операцией присваивания

- Компилятор **генерирует** присваивание по умолчанию, если операция не определена явно
- Присваивание по умолчанию производит **побитовое копирование**, поэтому оно будет не пригодным для объектов, содержащих указатели и ссылки

Правила работы с операцией присваивания

- Если операция присваивания возвращает ссылку на объект, то обеспечивается семантика, допускающая **последовательные присвоения**
- В целях безопасности операция присваивания должна **проверить** возможность **присвоения** объекта **самому себе**
- Операция присваивания **не наследуется**

Пример операции присваивания

```
// Питон, состоящий из сегментов
class TPhyton
{
public:
    .....
    // Операция присваивания
    TPhyton & operator=(TPhyton & other);
    .....
private:
    // Сегменты, из которых состоит питон
    TPoint *fSegments[100];
    // Текущая длина питона
    unsigned int fLength;
};
```

Пример операции присваивания

```
// Операция присваивания
TPhyton & TPhyton::operator=(TPhyton & other)
{
    if(this != &other) // проверка на
                        // самоприсваивание
    {
        // Удаляем память, занимаемую сегментами
        delete[] fSegments;

        . . . . .
    }
}
```

Пример операции присваивания

....

```
// Для питона заново выделяем память
// и копируем туда значения сегментов
fLength = other.fLength;
fSegments = new TPoint[fLength];
copySegments(fSegments, other.fSegments,
             fLength);
}
```

```
return *this; // возвращаем ссылку на себя
}
```

Пример использования операции присваивания

```
// Вызывается конструктор по умолчанию  
TPhyton Phyton1;
```

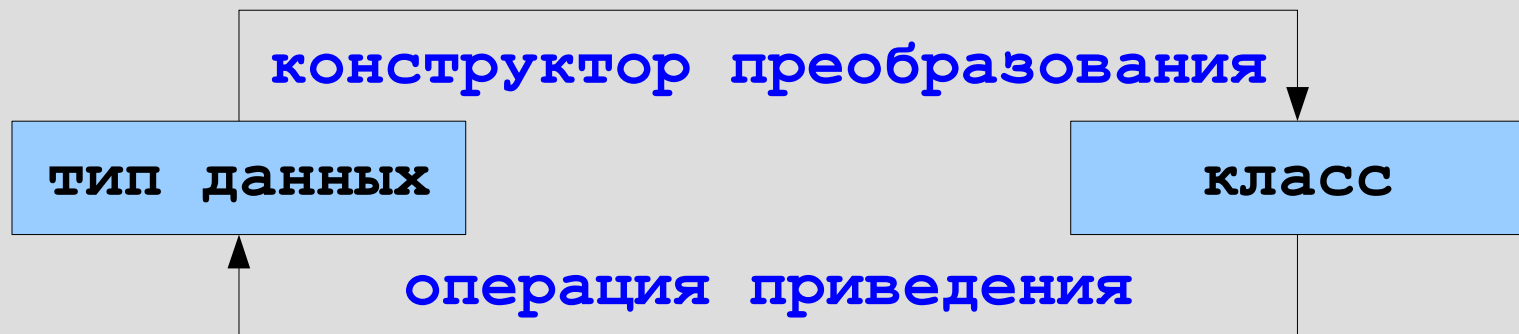
```
// Вызывается конструктор копирования  
TPhyton Phyton2 = Phyton1;
```

```
// Вызывается конструктор по умолчанию  
TPhyton Phyton3;
```

```
// Копирование питонов. Вызывается операция  
// присваивания. Порядок действия таков:  
// Phyton2.operator=(Phyton1);  
// Phyton3.operator=(Phyton2);  
Phyton3 = Phyton2 = Phyton1;
```

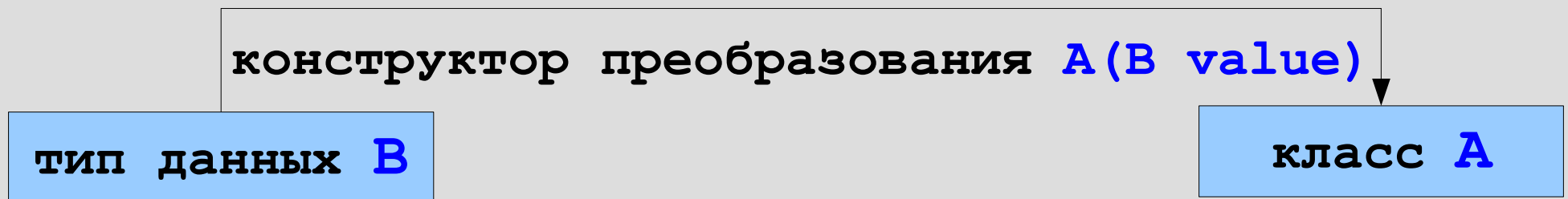
Операции преобразования

- Для преобразования переменной некоторого типа данных к заданному классу используется **конструктор преобразования**
- Для преобразования экземпляра заданного класса к другому типу данных используется **операция приведения**



Понятие конструктора преобразования

- Если конструктор класса **A** принимает **единственный** аргумент типа **B**, то говорят, что **B** может быть приведен к **A** с помощью преобразования конструктором



- Конструктор преобразования **вызывается** при явных и неявных **преобразованиях** типов данных к классу

Пример использования конструктора преобразования

- В классе QString имеются **конструктор преобразования**

```
QString(QChar chr);
```

- и **операция равенства**

```
bool operator== ( const QString & other ) const
```

- **Неявное использование** конструктора преобразования:

```
QString str("A");    // строка-объект
```

```
QChar symbol('C');   // символ-объект
```

```
// Сначала символ-объект неявно преобразуется
```

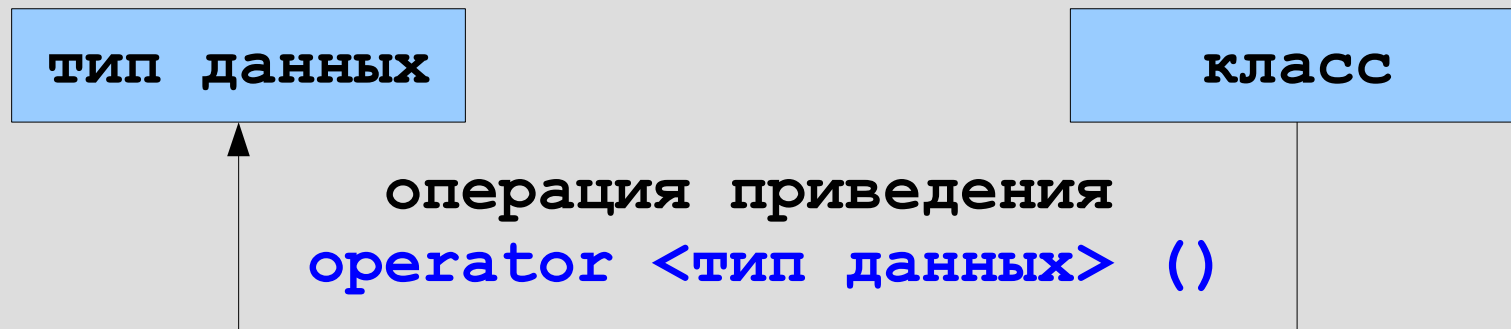
```
// в строку-объект, а затем выполняется операция
```

```
// сравнения двух строк
```

```
str == symbol;
```

Понятие операции приведения

- Метод, который осуществляет явное **преобразование класса к другому типу**, называется операцией приведения



Правила использования операции приведения

- Объявление:

`operator` <тип данных> ();

- Операция приведения **не имеет аргументов**
- Операция приведения не имеет явной спецификации **типа возвращаемого значения**
- Операция приведения **наследуется**
- Операция приведения может быть **виртуальной**

Пример использования операции приведения

- Пусть в классе `QString` имеется операция приведения к целому числу

```
operator int ( ) const
```

- Использование операции приведения:

```
QString str("35");  
int number;
```

```
number = str;
```

```
number = 1 + (int)str;
```

Перегрузка операций

- Класс может определять **свое поведение** для стандартных **операций**
- Введение в класс таких методов позволяет **строить выражения**, аналогичные арифметическим и булевым выражениям с обычно применяемыми **знаками операций** и **сохранением приоритетов операций**

Синтаксис метода-операции

- Объявление операции

<тип> **operator**<знак операции>

(<список параметров>)

- **Бинарная операция** должна быть представлена с помощью метода **с одним параметром**, при этом вызвавший ее объект считается левым операндом, а параметр метода – правым операндом

Синтаксис метода-операции

- Унарная операция представляется методом без параметров
- Если метод-операция возвращает объект или ссылку на объект, то допускается последовательное выполнение операций
- Тип возвращаемого значения может быть произвольным
- Метод-операцию можно вызывать как обычный метод, а можно как операцию

Пример метода-операции

- Объявление операции «равно» для класса QString:

```
bool operator == (const char * other) const
```

- Вызов метода-операции как обычного метода

```
QString str("Строка 1");  
bool isEqual = str.operator==( "Строка 2");
```

- Вызов метода-операции как операции

```
QString str("Строка 1");  
bool isEqual = str == "Строка 2";
```

Пример метода-операции

- Объявление операции «сдвига» для класса QStringList:

```
QStringList & operator<< (const QString & str)
```

- Последовательное выполнение операций

```
QStringList text;  
text << QString("Строка 1")  
    << QString("Строка 2");
```

Функция-операция как свободная функция

- Бинарная **функция-операция** определяется **вне класса**, при этом она должна иметь два параметра, один из которых имеет <тип класса> или <тип класса &>
- Формально «внешняя» функция-операция **не является частью класса**
- «Внешняя» функция-операции объявляется когда **левый операнд не является экземпляром** класса
- Пример: объявление операции «плюс»:

```
const QString operator+ ( const char * s1,  
                           const QString & s2 )
```

Примеры использования «внешней» функции-операции

```
const QString str("Строка 1");  
QString result("");
```

```
// левый операнд – не экземпляр класса  
result = "Строка 3" + str;
```

Методы, реализующие контрактные обязательства класса

- Открытые методы, которые **модифицируют** и **анализируют** значение не конкретного свойства, а **всего объекта в целом**. Пример: метод вставки символа в строку
- Открытые методы, реализующие действия, **не связанные с модификацией и анализом** состояния объекта. Пример: метод отрисовки самого себя у графического примитива

«Внутренние» типы данных класса

- Внутри класса могут быть **объявлены типы данных**, определяемые пользователем (структуры, объединения, перечисления и классы)

Причины использования «внутренних» ТИПОВ ДАННЫХ

- **Управление именами** — класс является отдельным пространством имен. Обращение к (общедоступному) внутреннему типу данных возможен только через имя внешнего класса

`<имя внешнего класса>::<имя типа данных>`

- **Управление доступом** - внутренний и внешний классы более тесно взаимодействуют друг с другом, чем с другими классами. Внутренний (закрытый) класс может иметь общедоступные поля, но они будут доступны только внешнему классу

Пример использования «внутренних» КЛАССОВ

```
// Связанный список целых чисел
class LinkedList
{
public:

    // Итератор
    class Iterator // Вложенный класс
    {
    public:
        void insert(int x) ;
        int erase() ;
        ...
    };

    ...
}
```

Пример использования «внутренних» классов

```
...  
private:  
  
    // Элемент списка  
    class Link // Вложенный класс  
    {  
    public:  
        Link *next;  
        int data;  
        ...  
    };  
};
```