12) поддержка классов

Для чего можно использовать:

- позволяют создавать пользовательские типы данных, похожие на встроенные

- для статического (с помощью шаблонов) и динамического полиморфизма (с помощью наследования и виртуальных методов)

- повторное использование кода

- объектно-ориентированного проектирования

- объектно-ориентированного программирования

У классов есть поля (атрибуты, свойства) и методы (функции-члены).

Класс - абстрактный тип данных, определяемый пользователем

Классы и структуры почти идентичны. Единственное отличие, что если не указать модификаторы доступа (public, private), то в классе все поля будут закрытыми (private), а в структуре - открытыми (public).

13) В классах предусмотрены специальные функции инициализации (конструтор) и завершения (деструктор).

И язык нам гарантирует, что перед использованием будет автоматически вызван конструктор, а перед удаление переменной класса будет вызван деструктор.

Если мы сами не определим конструктор и деструктор, то компилятор их сделает за нас.

Конструкторы можно перегружать как и другие функции.

Методы класса описанные в самом классе являются подставляемыми (inline) функциями.

Нестатические методы класса неявно принимают указатель на объект, для которого они вызваны. К полям этого объекта можно обращаться просто по имени.

Этот указатель называется this:

ClassType \* const this;

Операторы можно перегружать, определяя их как методы класса. В этом случае первым (или единственным) аргументом будет объект этого класса, доступный внутри определения оператора через this.

/////////////////////////////////////////////////////////////////////////////////////////

**3 взгляда на ООП!**

1.Исторический взгляд. Хочется писать программы быстрее, больше по функциональности, без ошибок, группами разработчиков. В жертву как обычно идут ресурсы. Чем быстрее на языке писать, тем программа требует больше памяти и медленнее выполняется. Началось всё с **двоичных кодов** (проблема в том, что код переноса значения из ячейки в ячейку – в одной архитектуре 113, в другой 115, потом **ассемблер** (уже мнемоники MOV eax, ebx), потом появились **языки типа Си** (конструкции с УДОБНЫМИ ветвлением и циклами, в асм они неудобные, появились удобные printf, в асм надо было писать в спец. область памяти и вызывать прерывание, чтобы отобразить что-то), дальше хочется работать с абстракциями, бухгалтер хочет описывать человека набором величин, можно завести массивы конечно, но удобнее сгруппировать данные касающиеся одного человека, это **структуры**, потом захотелось ещё и функции для работы с такими сложными типами группировать здесь же, появились **классы**.

2.Лингвистический взгляд. Если взглянуть на программы процедурных языков (например Си), то это текст, состоящий из сказуемых: нарисуй\_дом(), открой\_файл(), покрась\_дом(), и т.д. ООП программа же представляет собой текст, состоящий и из подлежащих и сказуемых. Дом.нарисуйся, Файл.откройся, Дом.покрасься, Генератор.дайчисло.

3.Практический взгляд. Хочется всё абстрагировать, собирать программу из готовых более-менее кубиков, которые могут взаимодействовать, по возможности, не вдаваясь в подробности кубика, чтобы его было трудно поломать. Берём кубики, один что-то просит у другого, из кубиков собираем программу. Что хотелось бы от кубиков? а) Кубик должен быть крепким, его нельзя поломать б) Сокрытие внутренностей кубика (например, кубик – матрица, что может быть не так с матрицей? выход за пределы массива, если вообще массивом реализовано, неправильная инициализация, удаление … много чего). Если кубик это все правильно делает, то это здорово в) Заменяемость кубиков. Пусть есть программа, обсчитывающая трехмерные сцены, геометрия, информация об объектах хранится в виде матриц. То есть программа умеет работать с кубиком – матрицей 3Д. Должен быть способ set(установить значение), get(прочитать значение), getSize(узнать размеры), должен быть add(способ прибавить к одной матрице другую). Оказалось, что в моей системе иногда матрицы дают разряженные (много нулей), хранить все нули накладно, хочется компактнее хранить (например в виде троек i,j,val 1,5,8 – то есть на 1,5 месте стоит 8). Хотелось бы чтобы системе было всё равно, разряженная матрица стала или нет, или ещё какая-то другая. Хочется в любом случае использовать get, set, getSize, add и не переписывать программу из-за кубика г) Облегчить создание новых кубиков. Имеется ввиду: был кубик матрица, стал кубик разряженная матрица, наверное какие-то вещи у них отличаются, set, get наверняка будут по-разному реализованы (список и массив), но сложение например, наверняка будет одинаково работать, подсчёт детерминанта, обратной матрицы. Хочется использовать старые кубики и доделывая их получать новые кубики.

На лекциях будем заниматься всеми этими тремя пунктами, синтаксис, примеры, как создаются такие кубики. На этом философская часть про ООП закончена.

Параметры функций по умолчанию: в конце списка.

Инкапсуляция (encapsulation) - это механизм, который объединяет данные и код, манипулирующий зтими данными, а также защищает и то, и другое от внешнего вмешательства или неправильного использования. В объектно-ориентированном программировании код и данные могут быть объединены вместе; в этом случае говорят, что создаётся так называемый "чёрный ящик". Когда коды и данные объединяются таким способом, создаётся объект (object). Другими словами, объект - это то, что поддерживает инкапсуляцию.

Внутри объекта коды и данные могут быть закрытыми (private). Закрытые коды или данные доступны только для других частей этого объекта. Таким образом, закрытые коды и данные недоступны для тех частей программы, которые существуют вне объекта. Если коды и данные являются открытыми, то, несмотря на то, что они заданы внутри объекта, они доступны и для других частей программы. Характерной является ситуация, когда открытая часть объекта используется для того, чтобы обеспечить контролируемый интерфейс закрытых элементов объекта.

На самом деле объект является переменной определённого пользователем типа. Может показаться странным, что объект, который объединяет коды и данные, можно рассматривать как переменную. Однако применительно к объектно-ориентированному программированию это именно так. Каждый элемент данных такого типа является составной переменной.

Наследование (inheritance) - это процесс, посредством которого один объект может приобретать свойства другого. Точнее, объект может наследовать основные свойства другого объекта и добавлять к ним черты, характерные только для него. Наследование является важным, поскольку оно позволяет поддерживать концепцию иерархии классов (hierarchical classification). Применение иерархии классов делает управляемыми большие потоки информации. Например, подумайте об описании жилого дома. Дом - это часть общего класса, называемого строением. С другой стороны, строение - это часть более общего класса - конструкции, который является частью ещё более общего класса объектов, который можно назвать созданием рук человека. В каждом случае порождённый класс наследует все, связанные с родителем, качества и добавляет к ним свои собственные определяющие характеристики. Без использования иерархии классов, для каждого объекта пришлось бы задать все характеристики, которые бы исчерпывающи его определяли. Однако при использовании наследования можно описать объект путём определения того общего класса (или классов), к которому он относится, с теми специальными чертами, которые делают объект уникальным. Наследование играет очень важную роль в OOП.

Полиморфизм (polymorphism) (от греческого polymorphos) - это свойство, которое позволяет одно и то же имя использовать для решения двух или более схожих, но технически разных задач. Целью полиморфизма, применительно к объектно-ориентированному программированию, является использование одного имени для задания общих для классов действий. Выполнение каждого конкретного действия будет определяться типом. Например, для языка Си, в котором полиморфизм поддерживается недостаточно, нахождение абсолютной величины числа требует трёх различных функций: abs(), labs() и fabs(). Эти функции подсчитывают и возвращают абсолютную величину целых, длинных целых и чисел с плавающей точкой соответственно. В С++ каждая из этих функций может быть названа abs(). Тип данных, который используется при вызове функции, определяет, какая конкретная версия функции действительно выполняется. В С++ можно использовать одно имя функции для множества различных действий. Это называется перегрузкой функций (function overloading).

В более общем смысле, концепцией полиморфизма является идея "один интерфейс, множество реализаций". Это означает, что можно создать общий интерфейс для группы близких по смыслу действий. Преимуществом полиморфизма является то, что он помогает снижать сложность программ, разрешая использование того же интерфейса для задания единого класса действий. Выбор же конкретного действия, в зависимости от ситуации, возлагается на компилятор. Вам, как программисту, не нужно делать этот выбор самому. Нужно только помнить и использовать общий интерфейс. Пример из предыдущего абзаца показывает, как, имея три имени для функции определения абсолютной величины числа вместо одного, обычная задача становится более сложной, чем это действительно необходимо.

  Полиморфизм может применяться также и к операторам. Фактически во всех языках программирования ограниченно применяется полиморфизм, например, в арифметических операторах. Так, в Си, символ + используется для складывания целых, длинных целых, символьных переменных и чисел с плавающей точкой. В этом случае компилятор автоматически определяет, какой тип арифметики требуется. В С++ вы можете применить эту концепцию и к другим, заданным вами, типам данных. Такой тип полиморфизма называется перегрузкой операторов (operator overloading).

Ключевым в понимании полиморфизма является то, что он позволяет вам манипулировать объектами различной степени сложности путём создания общего для них стандартного интерфейса для реализации похожих действий.

Класс без динамических полей Person:

class Person

{

private:

string name;

string phoneNumber;

unsigned int id;

public:

void output()

{

cout << id << "," << name << "," << phoneNumber << endl;

}

Person(string name\_="", string phoneNumber\_="", unsigned int id\_=0) :

name(name\_), phoneNumber(phoneNumber\_), id(id\_)

{

cout << "constructor Person" << endl;

id = id\_; }

};

int main()

{

Person p1("Ivan", "89502457834", 1);

p1.output();

}

Конструктор, списки инициализации: Липпман страница 342, Прата страница 640