Лекция 7

Объектно-ориентированное программирование

Алан Кей – фундаментальные характеристики ООП [Кау 1993]:

Все является объектом.

Объект как хранит информацию, так и способен ее преобразовывать. Любой элемент решаемой задачи (дом, собака, услуга, химическая реакция, город, космический корабль и т. д.) может представлять собой объект.

Программа - совокупность объектов, указывающих друг другу что делать. Для обращения к одному объекту другой объект «посылает ему сообщение». Возможно и «ответное сообщение».

Каждый объект имеет свою собственную «память», состоящую из других объектов. Так программист может скрыть сложность программы за довольно простыми объектами. У каждого объекта есть тип (класс).

Класс (тип) определяет какие сообщения объекты могут посылать друг другу. Каждый объект является представителем класса, который выражает общие свойства объектов (таких, как целые числа или списки). В классе задается поведение (функциональность) объекта. Тем самым все объекты, которые являются экземплярами одного класса, могут выполнять одни и те же действия. Все объекты одного типа могут получать одинаковые сообщения.

Классы организованы в единую древовидную структуру с общим корнем, называемую иерархией наследования. Память и поведение, связанное с экземплярами определенного класса, автоматически доступны любому классу, расположенному ниже в иерархическом дереве.

ООП - совокупность принципов разработки программ, понятий и элементов языка, позволяющих успешно создавать программы большого объема.

Основные принципы ООП

Абстракция данных – выделение наиболее значимых характеристик объектов и объединение их в класс;

Инкапсуляция - скрытие деталей реализации; объединение данных и действий над ними.

Наследование - частичное или полное заимствование функциональности уже существующего класса при описании нового. Позволяет создавать иерархию объектов, в которой объекты-потомки наследуют все свойства своих предков. Свойства при наследовании повторно не описываются. Кроме унаследованных, потомок обладает собственными свойствами. Объект в C++ может иметь сколько угодно потомков и предков.

Полиморфизм - использование объектов с одинаковым интерфейсом без информации о конкретном типе и структуре объектов. Возможность определения единого по имени действия, применимого ко всем объектам иерархии, причем каждый объект реализует это действие собственным способом.

Абстракция. Аппарат абстракции — инструмент для борьбы со сложностью реальных систем. Оотвлекаемся (абстрагируемся) от несущественных характеристик конкретных объектов, определяя только существенные. Например, в абстракции «часы» выделяем характеристику «показывать время», отвлекаясь от характеристик конкретных часов: форма, цвет, материал, цена, изготовитель.

Инкапсуляция. Инкапсуляция и абстракция - взаимодополняющие понятия: абстракция выделяет внешнее поведение объекта, а инкапсуляция

содержит и скрывает реализацию, которая обеспечивает это поведение. Инкапсуляция достигается с помощью информационной закрытости. Обычно скрываются структура объектов и реализация их методов.

Иерархическая организация - формирование из абстракций иерархической структуры. Упрощаются понимание проблем заказчика и их реализация - сложная система становится обозримой человеком.

Иерархическая организация в ОО системах:

- структура из классов («is a»-иерархия)
- структура из объектов («part of»-иерархия)

«is a» - иерархическая структура строится с помощью наследования, определяет отношение между классами, где класс разделяет структуру или поведение, определенные в одном другом (единичное наследование) или в нескольких других (множественное наследование) классах.

«part of» - иерархическая структура базируется на отношении агрегации.

При наследовании нижний элемент иерархии (подкласс) имеет больший уровень сложности (большие возможности), при агрегации - наоборот (агрегат обладает большими возможностями, чем его элементы).

Объект - конкретное представление абстракции.

Объект обладает индивидуальностью, состоянием и поведением.

Структура и поведение подобных объектов определены в их общем классе.

Термины «экземпляр класса» и «объект» взаимо-заменяемы.

Индивидуальность - это характеристика объекта, которая отличает его от всех других объектов.

Состояние объекта характеризуется перечнем всех свойств объекта и текущими значениями каждого из этих свойств

Объекты не существуют изолированно друг от друга. Они подвергаются воздействию или сами воздействуют на другие объекты.

Поведение характеризует, как объект воздействует на другие объекты (или подвергается воздействию) в терминах изменений его состояния и передачи сообщений.

Поведение объекта является функцией его состояния и выполняемых им операций (Купить, Продать, Взвесить, Переместить, Покрасить).

ООП в C++

Класс – тип данных, содержащий поля (переменные) и методы (функции) для их обработки.

Объект – переменная типа класс.

Элементы класса – поля и методы.

ООП в C++:

- статическое размещение объектов в памяти
- неявный вызов конструкторов и деструкторов
- перегрузка операций
- множественное наследование
- вложенность классов

```
class <имя> {
   [private:]
      <описание скрытых элементов>
           public:
      <описание доступных элементов>
};
```

Поля класса:

- могут иметь любой тип, кроме типа этого же класса (но могут быть указателями/ ссылками на этот класс);

- могут быть описаны с модификатором const;

могут быть описаны с модификатором static.

Инициализация полей при описании не допускается.

Классы и объекты

В процедурном подходе - два отдельных “мира”: мир данных и мир кода.

Мир данных - переменные разных типов, мир кода - код, сгруппированный в функции.

Функции могут использовать данные, но не наоборот. Функции могут злоупотреблять данными, т.е. использовать значение несанкционированным образом

Объектный подход предполагает иной способ мышления. Данные и код заключены вместе в один и тот же “мир”, разделенный на классы.

Каждый класс подобен рецепту, который можно использовать для создания полезного объекта. Можно создать столько объектов, сколько нужно для решения проблемы.

Каждый объект обладает набором признаков (свойств) и способен выполнять набор действий (методы).

Объекты являются воплощением идей, выраженных в классах. Объекты взаимодействуют друг с другом, обмениваясь данными или активируя свои методы. Правильно построенный класс (и объекты) способны защитить разумные данные и скрыть их от несанкционированных изменений. Между данными и кодом нет четкой границы: они живут как одно целое в объектах.

Все эти понятия не так абстрактны, все они взяты из реальной жизни и поэтому действительно полезны в компьютерном программировании: они не создают искусственную жизнь - они отражают реальные факты, отношения и обстоятельства.

Пример 1. Транспортные средства

Класс “транспортные средства” очень широк. Нужно определить несколько более специализированных классов (подклассов).

Класс “транспортные средства” - “супер-класс”.

Иерархия - сверху вниз.

Самый общий и широкий класс - находится наверху (супер super), его потомки находятся ниже (subs).

Существует множество классификаций.

Например, подклассы: наземные, водные, воздушные транспортные средства, космические аппараты.

Наземные транспортные средства могут быть:

колесные, гусеничные транспортные средства, суда на воздушной подушке и т.д.

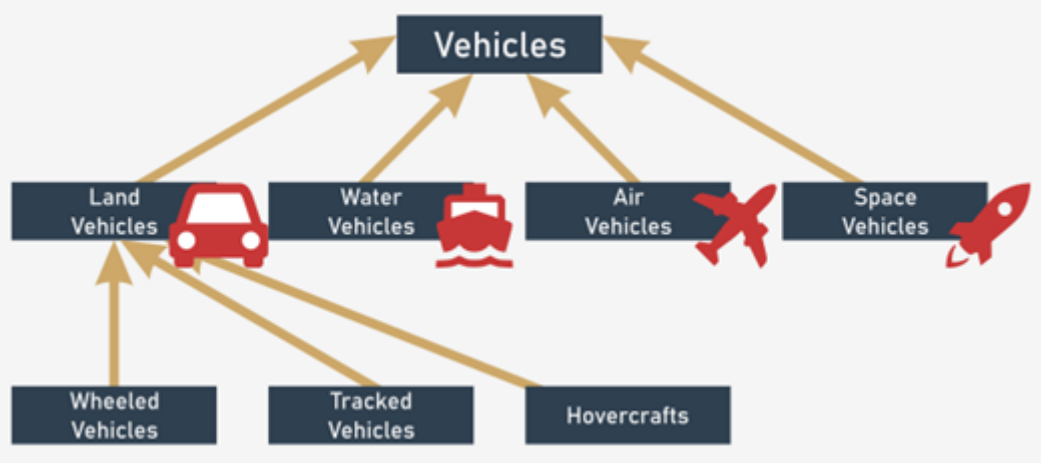

Рис. 6.1. Иерархия транспортных средств

Пример 2. Иерархия животного царства.

Все животные (класс высшего уровня) могут быть разделены:

на подклассы: млекопитающие, рептилии, птицы, рыбы, амфибии.

Подкласс млекопитающие, можно выделить подклассы: дикие и одомашненные млекопитающие. И т.д.

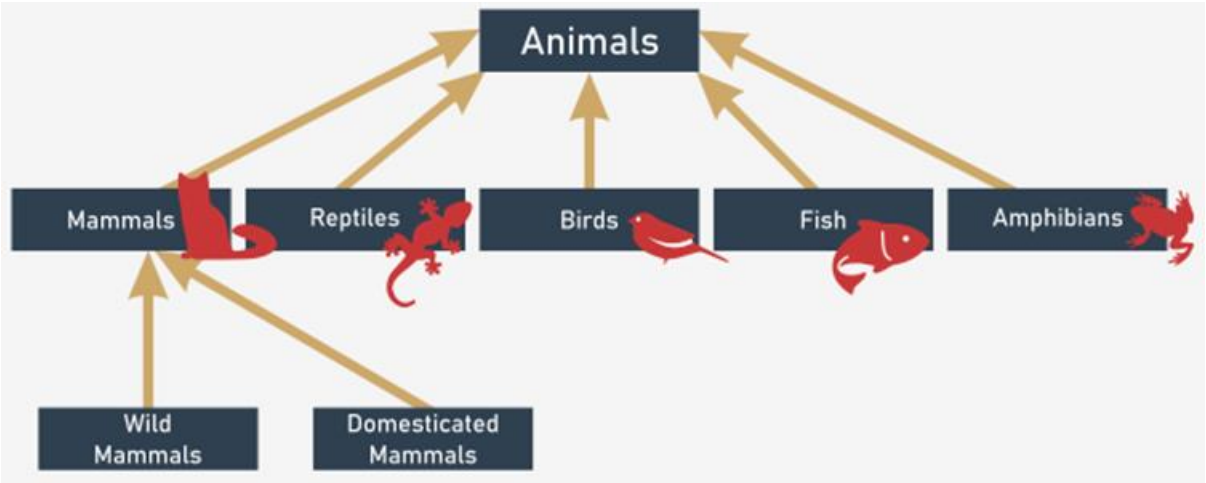

Рис. 6.2. Иерархия царства животных

Класс – это набор объектов.

Объект - это сущность, принадлежащее к классу.

Объект - это воплощение требований, черт и качеств, отнесенных к определенному классу.

Классы образуют иерархию.

Это означает, что объект, принадлежащий к определенному классу, принадлежит ко всем суперклассам одновременно.

Это также может означать, что любой объект, принадлежащий суперклассу, может не принадлежать к некоторым его подклассам.

Например, любой личный автомобиль - это объект, относящийся к классу “колесные транспортные средства”. Это также означает, что один и тот же автомобиль принадлежит ко всем суперклассам своего домашнего класса; следовательно, он также относится к классу “транспортные средства".

Каждый подкласс более специализирован (более специфичен), чем его суперкласс. Каждый суперкласс является более общим (более абстрактным), чем все его подклассы.

Пока предполагаем, что класс может иметь только один суперкласс – это не всегда так

Наследование (inheritance) – одно из фундаментальных понятий ООП.

Любой объект, привязанный к определенному уровню иерархии классов, наследует все характеристики (а также требования и качества), определенные внутри любого из суперклассов.

Домашний класс объекта может определять новые черты (а также требования и качества), которые будут унаследованы любым из его подклассов.

Каждый объект может быть оснащен тремя группами атрибутов:

- у объекта есть имя, которое однозначно идентифицирует его в домашнем пространстве имен namespace (хотя могут быть анонимные объекты);
- объект обладает набором индивидуальных свойств, которые делают его уникальным или выдающимся (некоторые объекты могут вообще не обладать свойствами)
- объект обладает набором способностей для выполнения определенных действий, которые могут изменить сам объект или некоторые другие объекты.

Есть подсказка (это не всегда работает) при описании объекта:

существительное - определяет имя объекта

прилагательное - определяет свойство объекта

глагол - определяет действие (активность) объекта

Пример:

«Мурка – это большая кошка, которая спит весь день»

Имя объекта = Мурка

Класс-предок = Кошка

Свойства = размер (большая)

Действие = спит (весь день)

Объектное программирование - это искусство определения и расширения классов.

Класс - это модель специфической части реальности, отражающая свойства и действия, обнаруженные в реальном мире.

Классы, определенные в начале, обычно слишком общие и неточные, чтобы охватить как можно большее количество реальных случаев.

Можно определить новые, более точные подклассы. Они унаследуют все от своего суперкласса, поэтому работа, затраченная на его создание, не будет потрачена впустую.

Новый класс может добавлять новые свойства и новые действия и, следовательно, может быть более полезным в конкретных приложениях.

Можно использовать его в качестве суперкласса для любого количества вновь созданных подклассов.

Вы можете создать столько классов, сколько вам нужно.

Класс, который вы определяете, не имеет никакого отношения к созданию объектов: существование класса не означает, что любой из совместимых объектов будет создан автоматически.

Сам класс не может создать объект – его нужно создать самостоятельно.

Пример:

```
class OurClass { };
```

Определили класс, у него нет ни свойств, ни методов.

Вновь определенный класс становится эквивалентом типа, можно использовать его в качестве имени типа.

`OurClass our_object;` - создание объекта класса OurClass

Объявляем переменную, которая может хранить объекты этого класса и одновременно создавать объект.

Стек: процедурный и О.О. подходы

Стек (LIFO) - это структура, разработанная для хранения данных особым способом.

Стек - это объект для двух элементарных операций, условно называемых "push" (новый элемент помещается сверху) и "pop" (существующий элемент удаляется сверху).

Процедурный подход. Push (помещает значение в стек):

- название функции - "push"
- функция получает один аргумент типа int (значение, помещаемое в стек)
- функция ничего не возвращает
- функция помещает значение аргумента в первый свободный элемент вектора и увеличивает SP

Функция не проверяет, есть ли место для нового значения

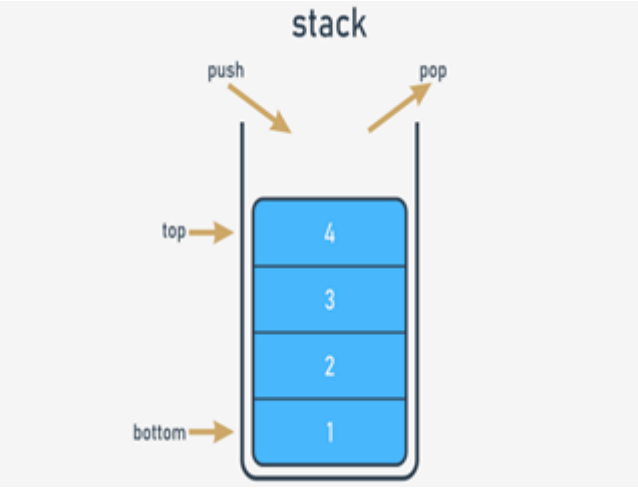

Рис. 6.3. Стек

```
void push(int value)
{
    stack[SP++] = value;
}
```

Функция Pop - извлекает значение из стека:

- название функции - "pop"
- функция не имеет никаких аргументов
- функция считывает значение из верхней части стека и уменьшает SP
- функция возвращает значение, взятое из стека
- Функция не проверяет, есть ли в стеке какой-либо элемент

```
int pop()
{
    return stack[--SP];
}
```

Программа - помещает три числа в стек, извлекает их и выводит их значения на экран.

```
#include <iostream>
using namespace std;
int stack[100];
int SP = 0;
void push(int value)
{
```

```
    stack[SP++] = value;
}
int pop()
{
    return stack[--SP];
}
int main()
{
    push(3);
    push(2);
    push(1);
    cout << pop() << endl;
    cout << pop() << endl;
    cout << pop() << endl;
}
```

Результат:

1

2

3

Недостатки:

- две основные переменные (stack и SP) полностью уязвимы; любой может изменить, фактически уничтожив стек;

- может понадобится больше одного стека; тогда придется создать еще один вектор для хранения стека, указатель стека для нового вектора и, возможно, больше функций push() и pop();
- может понадобятся другие функции, кроме push() и pop(),тогда будет несколько отдельно реализованных стеков;
- используем стек int , может быть нужно использовать стеки для других типов

Объектный подход позволяет решить проблемы:

- возможность скрыть (защитить) выбранные значения от несанкционированного доступа - инкапсуляция;
- когда есть класс, реализующий необходимые модели поведения стека, можно создавать столько стеков, сколько нужно, не нужно копировать части кода;
- возможность добавлять в стек новые функциями происходит из наследования;
- можно создать шаблон - представляющий обобщенный параметризованный класс, его код может адаптироваться к различным требованиям (создавать стеки для работы с другими типами данных

```
class Stack {
    int stackstore[100];
    int SP;
};
```

```
class Stack {
private:
    int stackstore[100];
    int SP;
};
```

Надо инкапсулировать обе переменные и сделать их недоступными для внешнего мира – эти данные закрытые private. Только сам класс может получить к нему доступ и изменять его.

На данном этапе можно не использовать ключевое слово private . Оно принимается по умолчанию.

Функции, которые реализуют операции push и pop.

Язык C++ предполагает, что функция такого рода (являющаяся методом класса) может быть описана двумя способами:

- внутри класса, когда тело класса содержит полную реализацию метода
- вне класса, когда тело содержит только заголовок функции; тело функции размещается вне класса

Рассмотрим оба способа: реализуем функцию pop внутри тела класса, функцию push - за пределами класса.

Реализация. Вызовем эти функции для push и pop значений. Они должны быть доступны каждому пользователю класса - public "общедоступный".

```
class Stack {
private:
    int stackstore[100];
    int SP;
public:
    void push(int value);
    int pop()
    {
        return stackstore[--SP];
    }
};
```

Имена функции, реализующих действия класса и размещенных вне тела класса, должны быть указаны с использованием имени домашнего класса и оператора “::”.

Если его опустить, то функция не будет считаться частью класса. Это будет обычная функция.

Важно: обе функции получают доступ к переменным класса (к stackstore и SP) без каких-либо препятствий.

Компоненты класса полностью видны другим компонентам того же класса.

Новый класс еще не готов к использованию, так как начальное значение SP неизвестно. Стек не будет функционировать правильно, если значение SP не равно 0 в начале операции.

Надо гарантировать, что самой первой операцией, выполняемой стеком, будет обнуление значения SP.

Нужно добавить еще одну функцию в класс, вызываемую неявно каждый раз, когда создается новый стек - “конструктор”, он отвечает за правильное построение каждого нового объекта класса.

Конструктор должен быть назван так же, как и его домашний класс (это требования языка C++).

Конструкторы не являются реальными функциями. У них нет никакого типа возвращаемого значения, даже void.

```
class Stack {
private:
    int stackstore[100];
    int SP;
public:
    void push(int value);
    int pop()
    {
        return stackstore[--SP];
    }
};
void Stack::push(int value)
{
    stackstore[SP++] = value;
}
```

```
class Stack {
private:
    int stackstore[100];
    int SP;
public:
    Stack()
    {
        SP = 0;
    }
    void push(int value);
    int pop()
    {
        return stackstore[--SP];
    }
};
void Stack::push(int value)
{
    stackstore[SP++] = value;
}
```

Класс полный

Можно создать несколько новых стеков и использовать их.

Фрагмент программы. Создали три объекта класса “Стек” Stack и используем их.

```
#include <iostream>
using namespace std;
int main()
{
    Stack little_stack, another_stack, funny_stack;
    little_stack.push(1);
    another_stack.push(little_stack.pop() + 1);
    funny_stack.push(another_stack.pop() + 2);
    cout << funny_stack.pop() << endl;
}
```

Как вызываем функцию из объекта? - это такое же соглашение, которое использовали для строк - функции постоянно привязаны к объектам.

Все функции объявлены общедоступными, можем свободно их использовать.

Вопрос: действительно ли private компоненты недоступны?

Изменим код и добавим строку в основные функции:

`little_stack.SP++;` – ошибка компиляции.

Далее - хотим, чтобы новый класс не только обрабатывал стеки, но и мог найти сумму всех элементов, хранящихся в стеке. Мы не хотим изменять ранее определенный стек. Нужен новый стек с новыми возможностями.

Надо создать подкласс subclass класса Stack.

Определим новый подкласс, который будет использоваться в качестве суперкласса:

```
class AddingStack : Stack {

};
```

Класс еще не определяет никаких новых компонентов, но это не значит, что он пустой.

Он выводит все компоненты, определенные его суперклассом – имя суперкласса записывается после двоеточия после имени нового класса.

Любой объект класса AddingStack может делать все, что делает объект каждого класса Stack .

Хотим, чтобы:

- функция push не только помещала значение в стек, но и добавляла значение в переменную sum
- функция pop не только извлекала значение из стека, но и вычитала значение из переменной sum

Решение:

1- добавим в класс новую переменную. Это будет закрытая переменная, так как не хотим, чтобы кто-то манипулировал sum.

2 - добавляем две функции… но уже есть функции в суперклассе. Собираемся изменить функциональность функций, а не их названия. Можно сказать, что интерфейс класса остается прежним, когда мы одновременно меняем реализацию.

```
class AddingStack : Stack {
private:
    int sum;
public:
    void push(int value);
    int pop();
};
```

Реализация функции push():

- добавить значение в переменную sum
- поместить значение в стек

```
void AddingStack::push(int value)
{
    sum += value;
    Stack::push(value);
}
```

2ая строка: не нужно еще раз определять операцию переноса значения в стек.

Реализация действия - внутри класса Stack . Нужно указать класс, содержащий функцию, чтобы не путать его с другой функцией с тем же именем.

Для этого - префикс Stack:: перед вызовом.

Новая функция pop:

```
int AddingStack::pop()
{
    int val = Stack::pop();
    sum -= val;
    return val;
}
```

Определили переменную sum , но не предоставили функцию для получения ее значения. Мы скрыли переменную.

Как можно раскрыть её, но сделать это так, чтобы сохранялась защита от изменений?

Определим новую функцию - get_sum.

Её задача - вернуть значение суммы.

```
int AddingStack::get_sum()
{
    return sum;
}
```

Нужно задать начальное значение переменной sum.

Оно должно быть обнулено при создании объекта.

Можно сделать это в конструкторе. Надо учесть, что при создании объекта класса AddingStack нужно инициализировать суперкласс.

Новый конструктор:

```
AddingStack::AddingStack(): Stack()
{
    sum = 0;
}
```

Stack() - это запрос на вызов конструктора суперкласса до того, как текущий конструктор начнет свою работу.

Полный код нового класса:

```
class AddingStack : Stack {
private:
    int sum;
public:
    AddingStack();
    void push(int value);
    int pop();
    int get_sum();
};
AddingStack::AddingStack() : Stack()
{
    sum = 0;
}
void AddingStack::push(int value)
{
    sum += value;
    Stack::push(value);
}

int AddingStack::pop()
{
    int val = Stack::pop();
    sum -= val;
    return val;
}
int AddingStack::get_sum()
{
    return sum;
}
```

Для проверки функционирования - функция main():

```
#include <iostream>
using namespace std;
int main()
{
    AddingStack super_stack;
        for(int i = 1; i < 10; i++)
        super_stack.push(i);
    cout << super_stack.get_sum() << endl;
    for(int i = 1; i < 10; i++)
        super_stack.pop();
    cout << super_stack.get_sum() << endl;
}
```

Результат:

45

0

Класс и его компоненты

Класс - это совокупность, состоящая из переменных (поля или свойства) и функций (методы). Переменные и функции являются компонентами класса.

```
class Class {
    int  value;
    void set_val(int value);
    int  get_val();
};
```

Класс имеет три компонента: переменную value типа int и две функции set_val и get_val.

Класс называется Class.

Так как все компоненты объявлены без использования спецификатора доступа (public или private), то все три компонента являются private.

Это означает, что класс A: Следует читать как:

```
class A {
    type var;
};
```

```
class A {
private:
    type var;
};
```

Компоненты set_val и get_val являются public – они доступны всем пользователям класса.

Компонент value является private – он доступен только внутри класса.

```
class Class {
public:
    void set_val(int value);
    int get_val();
private:
    int value;
};
```

Пусть действует объявление: Class the_object;

Любой объект класса оснащен всеми компонентами, определенными в классе.

Это означает, что the_object имеет те же три компонента, что и его базовый класс.

Общедоступные компоненты доступны для использования. Можно:

```
the_object.set_val(0);
```

Частные компоненты скрыты и недоступны. Нельзя:

```
the_object.value = 0;
```

Можно поместить определение функции, объявленной в классе, внутри самого класса или за его пределами. Возможно и то, и другое. Функция,

являющаяся компонентом класса, имеет полный доступ к другим компонентам класса, включая частные. Если какая-либо функция вводит сущность с именем, идентичным любому из компонентов класса, имя компонента класса переопределяется.

К нему можно получить доступ только по квалификации с именем домашнего класса.

Функция set_val использует параметр – value.

Параметр переопределяет компонент класса - value.

Это означает, что назначение: value = value; применяется к параметру value и не имеет ничего общего с одноименным компонентом.

Одна из правильных форм задания: Class::value = value;

Квалификация раскрывает переопределенное имя компонента.

Реализованная функция set_val:

```
class Class {
public:
    void set_val(int value)
    {
            Class::value = value;
    }
    int get_val();
private:
    int value;
};
```

Каждый объект оснащен специальным компонентом:

- его название this
- он не должен быть объявлен явно (это ключевое слово), поэтому его нельзя переопределить
- это указатель на текущий объект – каждый объект имеет свою собственную копию указателя this

Общее правило:

если S является структурой или классом, и у S есть компонент с именем C и

если p - указатель на структуру/класс типа S

к компоненту C можно получить доступ двумя способами:

(*p).C, где p явно разыменован для доступа к компоненту C

p->C, где p неявно разыменован для доступа к компоненту C

Это означает, что переопределенное имя компонента также может быть обнаружено с помощью метода:

```
class Class {
public:
    void set_val(int value)
    {
        this -> value = value;
    }
    int get_val();
private:
     int value;
};
```

Если тело функции класса задано вне тела класса, его имя должно быть дополнено именем класса и оператором ::

Функция, определенная таким образом, имеет такой же полный доступ ко всем компонентам класса, как и любая функция, определенная внутри класса.

Все правила для переопределения имен также действительны.

```
class Class {
public:
  void set_val(int value)
  {
    this -> value = value;
  }
  int get_val();
private:
  int value;
};

int Class::get_val()
{
      return value;
}
```

Имена функций класса могут быть перегружены, как и обычные имена функций.

Значения параметров по умолчанию также могут быть использованы.

Пример - класс с двумя функциями с одинаковым именем: set_val.

Первая имеет один параметр и задает поле value со значением параметра.

Вторая не имеет параметров и устанавливает в поле value значение -2.

Применяются ограничения на построение перегруженных функций класса

```
class Class {
public:
  void set_val(int value)
  {
    this -> value = value;
  }
  void set_val()
  {
    value = -2;
  }
  int get_val()
  {
    return value;
  }
private:
  int value;
};
```

Конструктор - функция с именем, идентичным имени ее класса.

Конструктор предназначен для построения объекта во время его создания, т.е. для инициализации значений полей, выделения памяти, создания других объектов и т.д.

Конструктор может обращаться ко всем компонентам объекта, как и любая другая функция-член класса, но не должен вызываться напрямую.

Конструктор не должен быть объявлен с использованием спецификаций возвращаемого типа, включая спецификации типа void.

Пример. Класс имеет один конструктор (имя Class). Конструктор инициализирует поле value значением -1.

Объявляя объекта класса: Class object; неявно вызывает конструктор.

Можно убедиться, что конструктор выполняет свою работу, используя команду: cout << object.get_val() << endl;

Если за это время никаких других изменений объекта не произошло, то на экран будет выведено значение -1 .

Не разрешается: object.Class() или Class::Class();

```
class Class {
public:
    Class()
    {
        this -> value = -1;
    }
    void set_val(int value)
    {
        this -> value = value;
    }
    int get_val()
    {
        return value;
    }
private:
    int value;
};
```

Конструкторы могут быть перегружены в зависимости от потребностей и требований.

Класс имеет два разных конструктора, которые отличаются количеством параметров. Второй конструктор требует одного параметра (первый не требует) и устанавливает value поле с полученным значением параметра.

Фактический конструктор выбирается во время создания объекта.

```
Class object1, object2(100);
cout << object1.get_val() << endl;
cout << object2.get_val() << endl;
```

Объект object1 будет создан с использованием конструктора без параметров, object2 - с помощью конструктора с одним параметром. Фрагмент кода выведет значения:

-1

100

Нельзя требовать использования несуществующего конструктора, неверно: Class objectx(2,100);

Если у класса есть конструкторы (по крайней мере, один), один из них должен быть выбран во время создания объекта, т.е. не разрешается писать объявление, в котором не указан целевой конструктор.

Пример:

Класс имеет один конструктор с одним параметром.

Это означает, что единственная допустимая форма создания объекта этого класса должна быть похожа на следующую: Class object(2);

Форма создания: Class object; не допускается.

```
class Class {
public:
    Class(int val)
    {
        this -> value = val;
    }
    void set_val(int value)
    {
        this -> value = value;
    }
    int get_val()
    {
        return value;
    }
private:
    int value;
};
```

Существует специальный конструктор для копирования одного объекта в другой.

Конструкторы имеют один параметр, ссылающийся на объект того же класса, и используются для копирования всех важных данных из исходного объекта в создаваемый в данный момент объект.

Они называются конструкторами копирования и неявно вызываются, когда за объявлением объекта следует инициатор.

Слово "копировать" не следует понимать буквально. Данные объекта на самом деле не нужно копировать. Ими можно манипулировать и обрабатывать. Важно то, что данные взяты из другого объекта того же (или аналогичного) класса.

Если конструктор копирования не существует в определенном классе, и инициатор фактически используется во время объявления объекта, его

содержимое будет фактически (в буквальном смысле) скопировано "поле за полем", как если бы объект был клонирован.

Конструктор копирования получает в качестве параметра константную ссылку на объект того же класса.

T (const T&)

Этот конструктор вызывается:

- при описании объекта с инициализацией другим объектом;
- при передаче объекта в функцию по значению;
- при возврате объекта из функции

Если конструктор копирования не указан, он будет создан автоматически.

Ключевое слово const , используемое в объявлении параметра, является обещанием того, что функция не будет пытаться изменять значения, хранящиеся в указанном объекте.

Конструктор копирования также будет использоваться, когда контекст требует копии определенного объекта, например, когда конкретный объект передается функции в качестве фактического параметра, передаваемого по значению.

Пример - два разных класса. У первого есть конструктор копирования, у второго его нет. Два объекта создаются для обоих классов, оба вторых объекта создаются путем копирования первых.

Программа дважды выводит число 200 .

Конструктор копирования Class1 создает новый (скопированный) объект, увеличивая его переменную value на 100 , поэтому object12.value содержит 200.

Конструктор копирования Class2 фактически копирует object21.value и, следовательно, вторые 200.

```
#include <iostream>
using namespace std;
class Class1 {
public:
    Class1(int val)
    {
        this -> value = val;
    }
    Class1(Class1 const &source)
    {
        value = source.value + 100;
    }
    int value;
};
class Class2 {
public:
    Class2(int val)
    {
        this -> value = val;
    }
    int value;
};
int main()
{
    Class1 object11(100), object12 = object11;
    Class2 object21(200), object22 = object21;
    cout << object12.value << endl;
    cout << object22.value << endl;
}
200
200
```

Рассмотренные конструкторы выполнили свою работу, инициировав объекты, но ни одно из их действий не пришлось отменять. Не было ничего, что можно было бы «очистить» после того, как объект закончил свою жизнь.

Это редко встречается в реальном программировании.

Многим объектам выделена память, необходимая для их работы. Эта память должна быть освобождена, когда объект завершит свою деятельность, лучший способ сделать это - выполнить очистку автоматически.

Неспособность очистить память вызовет явление, называемое "утечкой памяти", когда неиспользуемая (но все еще выделенная) память увеличивается в размере, что влияет на производительность системы.

Также можно намеренно спровоцировать утечку памяти.

Пример. Класс Class имеет только один конструктор, который отвечает за выделение памяти размера, указанного значением его параметра.

Объект этого класса создается как локальная переменная внутри функций make_a_leak().

Создание объекта состоит из двух этапов:

создается сам объект, и часть памяти неявно выделяется объекту

конструктор явно выделяет другую часть памяти

Переменная object является примером “автоматической переменной”: переменная автоматически завершает свой срок службы, когда заканчивается выполнение функции, содержащей объявление переменной.

Мы можем захотеть, чтобы возврат из make_a_leak() неявно освободил всю память, выделенную объекту, но память, явно выделенная конструктором, остается выделенной. Что еще хуже, мы также потеряли единственный указатель, в котором содержался адрес этой памяти (он был сохранен полем value , но объект, содержащий это поле, больше не существует).

```
#include <iostream>
using namespace std;
class Class {
public:
    Class(int val)
    {
        value = new int[val];
        cout << "Allocation (" << val << ") done." << endl;
    }
    int *value;
};
void makealeak()
{
    Class object(1000);
}
int main()
{
    makealeak();
}

Allocation (1000) done.
```

Можно избежать этого, определив специальную функцию - деструктор.

Деструкторы имеют ограничения:

- если класс называется X, его деструктор называется ~X
- класс может иметь не более одного деструктора
- деструктор должен быть функцией без параметров
- деструктор не должен вызываться явно

Пример - добавили в класс деструктор. Деструктор освобождает память, выделенную для поля value , защищая от утечки памяти.

```
#include <iostream>
using namespace std;
class Class {
public:
    Class(int val)
    {
        value = new int[val];
        cout << "Allocation (" << val << ") done." << endl;
    }
    ~Class()
    {
        delete [] value;
        cout << "Deletion done." << endl;
    }
    int *value;
};
void makealeak()
{
    Class object(1000);
}
int main()
{
    makealeak();
}
Allocation (1000) done.
Deletion done.
```

Все переменные относятся к одной из двух категорий:

- автоматические переменные, создаваемые и уничтожаемые (иногда неоднократно) автоматически во время выполнения программы.
- статические переменные - существуют непрерывно в течение всего выполнения программы

Все переменные по умолчанию являются автоматическими, если они явно не объявлены как статические.

Пример. Переменная var создается каждый раз, когда вызывается функция fun , и уничтожается каждый раз, когда функция завершает свое выполнение. Можно сказать, что создание и удаление переменной происходит автоматически. Это означает, что функция всегда будет выдавать один и тот же результат:

```
#include <iostream>
using namespace std;
void fun()
{
    int var = 99;
    cout << "var = " << ++var << endl;
}
int main()
{
    for(int i = 0; i < 5; i++)
        fun();
}
```

```
var = 100
var = 100
var = 100
var = 100
var = 100
```

Изменили код, добавили ключевое слово " static " перед " int ".

Код выглядит почти идентично, но его поведение радикально изменено.

Переменная " var " создается и инициируется один раз во время, так называемого, "пролога программы" и уничтожается после завершения программы во время работы так называемого "эпилога программы".

Это означает, что переменная var существует даже тогда, когда функция fun не работает, значение переменной сохраняется между последующими вызовами fun .

```
#include <iostream>
using namespace std;
void fun()
{
    static int var = 99;
    cout << "var = " << ++var << endl;
}
int main()
{
    for(int i = 0; i < 5; i++)
        fun();
}
```

```
var = 100
var = 101
var = 102
var = 103
var = 104
```

Каждый объект, созданный из определенного класса, называется экземпляром класса.

Все компоненты объекта (поля и функции) заключены внутри экземпляра.

Фрагмент кода неверен - ошибка компиляции.

Это вызвано:

- переменная val будет создана не раньше, чем во время создания объекта, и будет столько переменных var , сколько созданных объектов класса Class
- функция print() не может быть вызвана вне объекта, так как она не сможет получить доступ к несуществующей переменной var .

```
#include <iostream>
using namespace std;
class Class {
public:
    int val;
    void print()
    {
        cout << val << endl;
    }
};
int main()
{
    Class::val = 0;
    Class::print();
}
```

```
main.cpp: In function 'int main()':
main.cpp:16:12: error: invalid use of non-static data member 'Class::val'
    Class::val = 0;

main.cpp:7:9: note: declared here
    int val;

main.cpp:17:18: error: cannot call member function 'void Class::print()' without
object
    Class::print();
```

Все предыдущие правила верны, если они относятся к нестатическим компонентам класса (полям и функциям). Язык “C++” позволяет определять другие типы компонентов –“статическими компонентами”.

Статический компонент существует на протяжении всего срока службы программы. Всегда существует только один компонент, независимо от количества экземпляров класса. Можно сказать, что все экземпляры используют одни и те же статические компоненты.

Пример программы вводит класс с двумя полями одного и того же типа: одним статическим и одним нестатическим.

Переменная Static внутри общедоступной части Class является всего лишь объявлением. Это означает, что переменная должна иметь как явно

выраженное отдельное определение, так и возможную инициализацию, и оба они должны быть размещены вне определения класса.

Определение должно быть отделено от тела класса, так как статические переменные фактически не являются частью какого-либо объекта. В программе делаем это с помощью строки кода между телом класса и основной функцией. Удаление этой строки приводит к ошибке.

Каждый вызов функции print() увеличивает поле Static на единицу, и увеличение выполняется до того, как значение поля будет отправлено на экран.

Поскольку это поле существует только в одной копии, его значение является общим для всех объектов Class .

Нестатическое поле NonStatic хранится во всех объектах отдельно, следовательно, у него два разных значения.

```
#include <iostream>
using namespace std;
class Class {
public:
    static int Static;
    int NonStatic;
    void print()
    {
        cout << "Static = " << ++Static <<
                ", NonStatic = " << NonStatic << endl;
    }
};
int Class::Static = 0;
int main()
{
    Class instance1, instance2;
    instance1.NonStatic = 10;
    instance2.NonStatic = 20;
    instance1.print();
    instance2.print();
}
```

Результат:

```
Static = 1, NonStatic = 10
Static = 2, NonStatic = 20
```

Свойства статических переменных класса предопределяют их использование в качестве счетчиков экземпляров определенного класса.

Программа реализует эту идею простым способом. Во-первых, мы оснащаем класс Class статическим полем Counter. Переменная Counter определяется вне класса и присваивается нулевое значение, чтобы показать, что ни один из экземпляров класса пока не существует.

Увеличили Counter внутри конструктора Class и уменьшим его внутри деструктора Class .

Еще раз: доступ к полю Counter осуществляется напрямую, когда оно используется внутри класса, и с помощью оператора "::", когда оно используется вне класса.

Также можно получить доступ к статической переменной через любой из существующих экземпляров класса:

```
cout << b.Counter ;
```

до тех пор, пока доступная переменная является общедоступной.

```
#include <iostream>
using namespace std;
class Class {
public:
     static int Counter;
     Class()
     {
          ++Counter;
     };
     ~Class()
     {
          --Counter;
          if(Counter == 0)
               cout << "Bye, bye!" << endl;
     };
     void HowMany()
     {
          cout << Counter << " instances" << endl;
     }
};
int Class::Counter = 0;

int main()
{
     Class a;
     Class b;
     cout << Class::Counter << " instances so far" << endl;

     Class c;
     Class d;

d.HowMany();
}
```

Результат:

```
2 instances so far
4 instances
Bye, bye!
```

Можно сделать определенную переменную private и статичной одновременно. Это предотвратит прямой доступ к переменной, но это может быть то, что нам нужно, если мы хотим защитить значение от любых несанкционированных изменений.

Программа выдает результат:

2 instances

4 instances

Bye, bye!

Попытки доступа к переменной Counter : Class::Counter = 1; запрещены.

Единственной разрешенной операцией является инициализация переменной: int Class::Counter = 0;

Функции также могут быть объявлены как статические.

Пример - программа, содержащая класс с двумя статическими компонентами: переменной и функцией.

Статическая функция static function, как и статическая переменная, также может быть доступна (точнее, вызываться), когда не было создано ни одного экземпляра класса.

Статическая функция может быть вызвана изнутри класса:

HowMany();

или с помощью любого из существующих экземпляров:

b.HowMany();

Программа выводит на экран строки:

```
0 instances
2 instances
4 instances
Bye, bye!
```

```
#include <iostream>
using namespace std;
class Class {
    static int Counter;
public:
    Class()
    {
        ++Counter;
    };

    ~Class()
    {
        --Counter;
        if(Counter == 0)
            cout << "Bye, bye!" << endl;
    };

    static void HowMany()
    {
        cout << Counter << " instances" << endl;
    }
};
int Class::Counter = 0;
int main()
{
    Class::HowMany();
    Class a;
    Class b;
    b.HowMany();
     Class c;
     Class d;

     d.HowMany();
}
```

Сосуществование статических и нестатических компонентов в одном классе вызывает дополнительные проблемы.

Можно определить четыре конкретных случая, когда оба типа компонентов взаимодействуют друг с другом:

- статический static компонент обращается к static статическому компоненту

- статический static компонент обращается к нестатическому non-static компоненту
- нестатический non-static компонент обращается к статическому static компоненту
- нестатический non-static компонент обращается к нестатическому non-static компоненту

Для упрощения предположим, что собираемся тестировать только компоненты, которые являются функциями. Выводы будут верны и для переменных.

Статические и нестатические компоненты

1. Статическое → статическое взаимодействие

```
#include <iostream>
using namespace std;
class Test {
public:
    static void funS1()
    {
        cout << "static" << endl;
    }
    static void funS2()
    {
        funS1();
    }
};

int main()
{
    Test object;
    Test::funS2();
    object.funS2();
}
```

Статическая функция funS2 пытается вызвать другую статическую функцию funS1.

Подобный случай всегда возможен, так как обе функции доступны в течение всего срока службы программы.

К ним можно получить доступ как из объекта, так и из класса.

Программа может быть успешно скомпилирована.

Результат:

```
static
static
```

2. Статическое → нестатическое взаимодействие

```
#include <iostream>
using namespace std;
class Test {
public:
    void funN1()
    {
        cout << "non-static" << endl;
    }
    static void funS1()
    {
        funN1();
    }
};
int main()
{
      Test object;
      Test::funS1();
      object.funS1();
}
```

Статическая функция funS1 пытается вызвать нестатическую функцию funN1.

Подобный случай невозможен, так как вызываемая функция существует тогда и только тогда, когда любой из объектов, содержащих эту функцию, также существует.

Функция не может быть доступна без указания связанного объекта.

Программа не может быть успешно скомпилирована.

3. Нестатическое → статическое взаимодействие

```
#include <iostream>
using namespace std;
class Test {
public:
    static void funS1()
    {
        cout << "static" << endl;
    }
    void funN1()
    {
        funS1();
    }
};
int main()
{
    Test object;
    object.funN1();
}
```

Нестатическая функция funN1 вызывает статическую функцию с именем funS1.

Подобный случай всегда возможен, так как статическая функция доступна до создания любого объекта.

Программа может быть успешно скомпилирована.

Результат:

```
static
```

4. Нестатическое → нестатическое взаимодействие

Нестатическую функцию возможно вызвать изнутри нестатической функции.

Таблица 6.1. Статический и нестатический вызовы

Компоненты:	Статический	Нестатический
Статический	ДА	НЕТ
Нестатический	ДА	ДА

Объекты могут существовать как динамически создаваемые и уничтожаемые объекты. Объекты могут появляться по требованию – когда они необходимы – и исчезать.

Пример. Определили класс Class (скелет для класса) с одним конструктором и одним деструктором. Они только объявляют, что их вызвали.

Функция main объявляет переменную ptr - указатель на объекты класса Class. Создали объект этого класса - new.

! Можно опустить пустые скобки после имени Class – в любом случае будет активирован конструктор без параметров.

Объект уничтожается - delete (уничтожения объекта начинается с неявного вызова его деструктора).

```
#include <iostream>
using namespace std;
class Class {
public:
    Class()
    {
        cout << "Object constructed!" << endl;
    }
    ~Class()
    {
        cout << "Object destructed!" << endl;
    }
};
int main()
{
    Class *ptr = new Class();
    delete ptr;
}
```

Результат:

```
Object constructed!
Object destructed!
```

Функции-члены, вызываемые для объекта, доступ к которому осуществляется с помощью указателя, также должны быть доступны с помощью оператора стрелки.

Добавили функцию-член в класс - пример.

```
#include <iostream>
using namespace std;
class Class {
public:
    Class()
    {
        cout << "Object constructed!" << endl;
    }

    ~Class()
    {
        cout << "Object destructed!" << endl;
    }
    void inc and print()
    {
        cout << "value = " << ++value << endl;
    }
    int value;
};
int main()
{
    Class *ptr = new Class;

    ptr -> value = 1;
    ptr -> inc and print();
    delete ptr;
}
```

Результат:

```
Object constructed!
value = 2
Object destructed!
```

Если класс имеет более одного конструктора, один из них может быть выбран во время создания объекта путем указания формы списка параметров.

Список должен быть однозначно совместим с одним из доступных конструкторов классов.

Пример: изменили программу, есть два конструктора. Создали два новых объекта внутри основных функций. Они отличаются конструктором, используемым для построения каждого из объектов. По сути, их value имеют разные значения.

Ход выполнения программы:

происходит из вызова конструктора Class() (устанавливает поля value = 0)

происходит из вызова конструктора Class(int value) (устанавливает значение поля в 2)

по мере того, как inc_and_print() увеличивает значение объекта value на 1, на экран отправляется 2 (мы явно установили значение поля в 1 в предыдущей строке)

поскольку значение value в настоящее время равно 2, отображается 3

деструктор объекта представляет содержимое текущего значения объекта value как указано выше

```
#include <iostream>
using namespace std;
class Class {
public:
    Class()
    {
        cout << "Object constructed (#1)" << endl;
    }
    Class(int value)
    {
        this -> value = value;
        cout << "Object constructed (#2)" << endl;
    }
    ~Class()
    {
        cout << "Object destructed! val = " << value << endl;
    }
    void inc and print()
    {
        cout << "value = " << ++value << endl;
    }
    int value;
};
```

```
int main()
{
    Class *ptr1 = new Class, *ptr2 = new Class(2);
    ptr1 -> value = 1;
    ptr1 -> inc and print();
    ptr2 -> inc and print();
    delete ptr2;
    delete ptr1;
}
```

Результат:

```
Object constructed (#1)
Object constructed (#2)
value = 2
value = 3
Object destructed! val = 3
Object destructed! val = 2
```