ООП 2021
Лекция 4

Механизмы разработки библиотек классов. Принципы S.O.L.I.D. Правила трех, пяти, ноля.

oopCpp@yandex.ru

Три основных механизма разработки библиотеки классов

Вывод, полиморфизм, инкапсуляция

```
class Shape{
  public:
    virtual void draw () ;
  protected:
  private:
…
};
class Circle :   public  Shape  {…};
```

Вывод. Это способ построения одного класса из другого так, чтобы новый класс можно было использовать вместо исходного.

Класс Circle является производным от класса Shape, иначе говоря, класс Circle является разновидностью класса Shape или класс Shape является базовым по отношению к классу Circle. Производный класс (в данном случае Circle) получает все члены базового класса (в данном случае Shape) в дополнение к своим собственным.

Полиморфизм (динамический полиморфизм). В языке C++ можно определить функцию в базовом классе и функцию в производном классе с точно таким же именем и типами аргументов, чтобы при вызове пользователем функции базового класса на самом деле вызывалась функция из производного класса.

Когда класс Window вызывает функцию draw () из класса Circle, выполняется именно функция draw () из класса Circle, а не функция draw () из класса Shape.

Инкапсуляция. Посредством закрытых и защищенных членов (private и protected) обеспечивается сокрытие деталей реализации классов, чтоб защитить их от непосредственного доступа, который может затруднить сопровождение программы.

Указатель **this**

Если хотите явно сослаться на объект, из которого вызвана функция-член, то можете использовать зарезервированный указатель **this**.

this используется в операторах присваивания или вывода:

X& operator=(const X& t) { v = t.v; return ***this**; }

X& operator<<(int a) { …. return ***this**; }

```
class Date {
 public:
   int d, m, y;
  int month() const { return this->m; }   // <=>   { return  m; }
};
```

Определения членов класса

Члены класса, являющиеся целочисленными константами, функциями или типами, могут быть определены как в классе, так и вне его.

```
struct S {
 static const int c = 1; //  определение
 static const int c2;    //  объявление
 void f() { }             //  определение
 void f2 ();       // объявление
 struct SS1 { int a; }; //  определение
 struct SS2;    // объявление
};
```

Члены, которые не были определены в классе, должны быть определены "где-то“ еще.

```
const int S::c2 = 7;
void S::f2() { }
struct S::SS2 { int m; };
```

Функции-члены не занимают память, выделенную для объекта.

```
struct S {
 int m;
 void f();
};
```

Здесь sizeof (S) == sizeof (int).

Следует подчеркнуть, что класс с виртуальной функцией имеет один скрытый (дополнительный) член, обеспечивающий виртуальные вызовы – таблицу виртуальных функций.

Производные классы

Класс можно определить производным от других классов. В этом случае он наследует члены классов, от которых происходит (своих базовых классов):

```
struct S {
  int m_s;
  void fs() { };
};
class D : public S {
int m_d; void fd();
}I
```

Здесь класс S имеет два члена: m_s и fs (), а класс D — четыре члена: m_s, fb (), m_d и fd ().

Как и члены класса, базовые классы могут быть открытыми и закрытыми (public или private):

```
class DD : public S1, private S2 { . . . };
```

Если класс имеет несколько непосредственных базовых классов (как, например, класс DD), то говорят, что он использует множественное наследование (multiple inheritance).

Указатель на производный класс D можно неявно преобразовать в указатель на его базовый класс в при условии, что класс в является доступным и однозначным по отношению к классу D.

```
struct B { };
struct B1:B{ };   // B - открытый базовый класс по отношению к классу B1
struct B2: B { }; // B  - открытый базовый класс по отношению к классу B2
struct C { };
struct DD : B1 , B2, private C { }; // множественное наследование

DD* p = new  DD;
 B1* pb1 = p;    // OK
  B* pb = p;  // ошибка: неоднозначность: B1::B или B2::B
C* pc = p;   // ошибка: DD::C — закрытый класс
```

Виртуальные функции

Виртуальная функция (virtual function) — это функция-член, определяющая интерфейс вызова функций, имеющих одинаковые имена и одинаковые типы аргументов в производных классах.

При вызове виртуальной функции она должна быть определена хотя бы в одном из производных классов. В этом случае говорят, что производный класс замещает (override) виртуальную функцию-член базового класса.

```
class Shape {
public:
  virtual void draw();   //   "virtual" означает "может быть  замещена"
  virtual ~Shape() { }   // виртуальный деструктор
};
class Circle : public Shape {
public:
void draw();  // замещает функцию Shape::draw
~Circle();      // замещает функцию Shape::~Shape()
 };
```

Виртуальные функции базового класса (в данном случае класса Shape) определяют интерфейс вызова функций производного класса (в данном случае класса Circle).

Абстрактные классы

Абстрактный класс - это класс, который можно использовать только в качестве базового класса. Объект абстрактного класса создать нельзя.

```
Shape s;       // ошибка: класс Shape является абстрактным

class Circle : public Shape {
 public:
 void draw();  // замещает override Shape::draw
};

 Circle c(p,20);  // ОК: класс Circle не является абстрактным
```

Класс, члены которого только виртуальные функции называют чисто абстрактным классом или Интерфейсом.

Порядок создания и разрушения объектов

Создание производится снизу вверх, т.е. объект базового класса создается до создания членов производного класса.

Члены производного класса и объекты базовых классов создаются в порядке их объявления и уничтожаются в обратном порядке.

Таким образом, конструктор и деструктор всегда работают с точно определенными объектами базовых классов и членов производного класса.

Вообразите себе матрешку. Сначала создается самая маленькая внутренняя матрешка (базовый класс), затем – ее оболочка, матрешка побольше (производный от базового класс), внутри которой оказывается маленькая матрешка. Затем еще больше (производный от производного класса). И т.д. А разрушение матрешек проводится в обратном порядке, начиная с самой большой и до конца – маленькой матрешки.

Пример.

```
struct D : public B1, public B2 {
 M1 m1; M2 m2;
};
```

Предполагая, что классы B1, B2, M1 и M2 определены, можем написать следующий код:

```
void f() {
 D d;          // инициализация по умолчанию
 D d2 = d; // копирующая инициализация
d=D();      // инициализация по умолчанию, за которой следует копирующее
              //  присваивание
}   // в этом месте объекты d и d2 уничтожаются
```

инициализация объекта d по умолчанию выполняется путем вызова четырех конструкторов по умолчанию (в указанном порядке): B1::B1(), B2::B2 (), M1::M1() и M2::M2 (). Если один из этих конструкторов не определен или не может быть вызван, то создание объекта d невозможно. Уничтожение объекта d выполняется путем вызова четырех деструкторов (в указанном порядке): M2:: ~M2 (), M1::~M1(), B2:: ~B2 () и B1:: ~B1 (). Если один из этих деструкторов не определен или не может быть вызван, то уничтожение объекта d невозможно.

Оператор присваивания: operator=

Оператор присваивания бывает двух видов:

- присваивание копированием
- присваивание перемещением

В первом случае сигнатура входящих параметров совпадает с параметрами конструктора копирования:

```
MyClass (const MyClass& obj) {
 // при создании нового объекта
//  что-то копируется из внешнего obj в this->obj
 }
MyClass& operator= (const MyClass& obj) {
 //  что-то копируется из внешнего obj  в уже существующий  this->obj
 }
```

Во втором – совпадает с параметрами конструктора перемещения:

```
MyClass (  MyClass&& obj)
```

Правило 3-х

Правило трёх — правило в C++, гласящее, что если класс или структура определяет один из следующих методов, то они должны **явным** образом определить **все три метода**:

Деструктор

Конструктор копирования

Оператор присваивания копированием

Если один из них должен быть определен программистом, то это означает, что версия, сгенерированная компилятором, не удовлетворяет всем потребностям класса.

Правило пяти

С выходом C++ 11 правило расширилось и теперь называется **правило пяти**. Теперь при реализации конструктора необходимо реализовать:

- Деструктор
- Конструктор копирования
- Оператор присваивания копированием
- Конструктор перемещения
- Оператор присваивания перемещением

Правило Ноля

Мартин Фернандес предложил также правило ноля.

По этому правилу **не стоит определять ни одну из пяти функций** самому;

Надо поручить их создание компилятору (присвоить им значение = **default**;).

Для владения ресурсами вместо простых указателей стоит использовать специальные классы-обёртки, такие как:

std::unique_ptr и

std::shared_ptr.

На самом деле имеются серьезные причины не соглашаться с этим мнением!

Принципы S.O.L.I.D.

Это стилистический стандарт ОО программирования (включая проектирование) , который разработчики должны понимать, чтобы не допускать создания некачественной архитектуры ПО.

Single responsibility (**Принцип единственной ответственности**)

Open-closed (**Принцип открытости/закрытости**)

Liskov substitution (**Принцип подстановки Барбары Лисков**)

Interface segregation (**Принцип разделения интерфейса**)

Dependency inversion (**Принцип инверсии зависимостей**)

Single responsibility (**Принцип единственной ответственности**)

Один класс должен решать только одну задачу. Он может иметь несколько методов, но они должны быть сосредоточены на достижении одной цели.

Если класс имеет несколько назначений, то его нужно разделить на разные классы.

```
struct Car {
   void move(); // Car движется
   void technical_inspection (); // Техосмотр – не в этом классе!
};
struct Car {
public:
    void move(); // Машина движется
};

struct Technical_inspection {
      void Technical_inspection  ( Car* car );    // Техосмотр
};
```

Open-closed (**Принцип открытости/закрытости**)

Любая сущность (например, класс или модуль) должна быть открыта для расширений, но закрыта для изменений.

Каким же образом мы можем написать код, который будет легко расширяем без внесения изменений? Используйте полиморфизм для написания кода в терминах абстракций, после чего при необходимости добавления функциональности это можно будет сделать путем разработки и добавления различных реализаций упомянутых абстракций.

Шаблоны и виртуальные функции образуют барьер для зависимостей между кодом, использующим абстракции, и кодом, их реализующим.

Управление зависимостями обусловлено выбором верных абстракций. Если абстракции несовершенны, добавление новой функциональности потребует изменений интерфейса (а не просто добавления новых реализаций интерфейса), которые обычно влекут за собой значительные изменения существующего кода. Но абстракции потому и называются "абстракциями", что предполагается их большая стабильность по сравнению с "деталями", т.е. возможными реализациями абстракций.

Liskov substitution (Принцип подстановки Барбары Лисков)

«объекты в программе должны быть заменяемыми на экземпляры их подтипов без изменения правильности выполнения программы»

Цель открытого наследования в реализации заменимости.

Цель отрытого наследования не в том, чтобы производный класс мог повторно использовать код базового класса для того, чтобы с его помощью реализовать свою функциональность.

Открытое наследование всегда должно моделировать отношение "является" ("работает как"): все **контракты** базового класса должны быть выполнены, для чего все перекрытия виртуальных функций-членов не должны требовать большего или обещать меньше, чем их базовые версии. Код, использующий указатель или ссылку на Base, должен корректно вести себя в случае, когда указатель или ссылка указывают на объект Derived.

Рассмотрим два класса Square (квадрат) и Rectangle (прямоугольник), каждый из которых имеет виртуальные функции для установки их высоты и ширины. Тогда Square не может быть корректно унаследован от Rectangle, поскольку код, использующий видоизменяемый Rectangle, будет полагать, что функция SetWidth не изменяет его высоту (независимо от того, документирован ли данное условие классом Rectangle явно или нет), в то время как функция Square: :SetWidth не может одновременно выполнить это условие и свой инвариант "квадратности". Но и класс Rectangle не может корректно наследовать классу Square, если его клиенты Square полагают, например, что для вычисления его площади надо возвести в квадрат ширину, либо используют какое-то иное свойство, которое выполняется для квадрата и не выполняется для прямоугольника.

Описание "является" для открытого наследования оказывается неверно понятым при использовании аналогий из реального мира: квадрат "является" прямоугольником в математическом смысле , но с точки зрения поведения Square не является Rectangle. Вот почему вместо “**является**“ предпочитают говорить: “**действует как**“ (или **"используется как"**) для того, чтобы такое описание воспринималось максимально правильно. (Саттер)

Принцип разделения интерфейса (interface segregation principle, ISP)

Программные сущности **не должны зависеть** от методов, которые они не используют.

Принцип разделения интерфейсов говорит о том, что **слишком «толстые»** интерфейсы необходимо **разделять** на более маленькие и специфические, чтобы программные сущности маленьких интерфейсов знали только о методах, которые **необходимы** им в работе. В итоге, при изменении метода интерфейса **не должны меняться программные сущности**, которые этот метод не используют.

Dependency Inversion (DIP) (Принцип инверсии зависимостей)

Данный принцип состоит в следующем. Высокоуровневые модули не должны зависеть от низкоуровневых. И те, и другие должны зависеть от абстракций.

Абстракции не должны зависеть от деталей; вместо этого **детали должны зависеть от абстракций**.

Из DIP следует, что корнями иерархий должны быть абстрактные классы, в то время как конкретные классы в этой роли выступать не должны. **Абстрактные** базовые классы должны беспокоиться об определении **функциональности**, но не о ее реализации.

Принцип инверсии зависимостей имеет три фундаментальных преимущества при проектировании.

Повышение надежности. Менее стабильные части системы (реализации) зависят от более стабильных частей (абстракций). **Надежный** дизайн тот, в котором воздействие изменений ограничено. При плохом проектировании небольшие изменения в одном месте расходятся кругами по всему проекту и оказывают влияние на самые неожиданные части системы. Именно это происходит, когда проект строится **на конкретных** базовых классах.

Повышение гибкости. Дизайн, основанный на **абстрактных** классах, в общем случае более гибок. Если абстракции корректно смоделированы, то при появлении новых требований легко разработать новые реализации. И напротив, дизайн, зависящий от многих конкретных деталей, оказывается более жестким в том смысле, что новые требования приводят к существенным изменениям в ядре системы.

Улучшение модульности. Дизайн, опирающийся на абстракции, обладает хорошей модульностью благодаря **простоте зависимостей**: высокоизменяемые части зависят от стабильных частей, но не наоборот. Дизайн же, в котором интерфейсы перемешаны с деталями реализации, применить в качестве отдельного модуля в другой системе оказывается очень сложно.

KISS («Keep it simple, stupid»)

KISS — принцип проектирования, принятый в ВМС США в 1960.

Принцип KISS утверждает, что большинство систем работают лучше всего, если они **остаются простыми**, а не усложняются.

Поэтому в области проектирования **простота** должна быть одной из ключевых целей, и следует избегать ненужной сложности.

Идиомы c++

Идиома программирования — устойчивый способ выражения некоторой составной конструкции в языке программирования. Идиома является **шаблоном** решения задачи, **записи алгоритма** или **структуры данных** путём **комбинирования** встроенных элементов языка.

Идиому можно считать самым низкоуровневым шаблоном проектирования, применяемым на стыке проектирования и кодирования на языке программирования. Идиома предписывает конкретный способ реализации определённых деталей и отношений между ними средствами конкретного языка.

Инкремент:

```
i = i + 1;
i += 1;
  ++i;
  i++;
```

Идиома copy-and-swap:

Обмен значениями между двумя переменными выглядит следующим образом:

```
x = a;
a = b;
b = x;
```

Идиома бесконечный цикл

```
for (;;) {
    }
```

Или : `while(1) { }`

Идиома RAII

Получение ресурса есть инициализация (Resource Acquisition Is Initialization (RAII))

Получение некоторого ресурса неразрывно совмещается с инициализацией, а освобождение — с уничтожением объекта.

Типичным способом реализации является организация получения доступа к ресурсу в **конструкторе**, а освобождения — в **деструкторе** соответствующего класса.

Деструктор переменной немедленно вызывается при выходе из её области видимости, в том числе в ситуациях, когда возникло исключение, и таким образом, ресурс необходимо освободить, что делает RAII **ключевой концепцией** для написания кода, безопасного при исключениях.

Динамическая идентификация типа данных (run-time type information, run-time type identification, RTTI)

Механизм, который позволяет определить тип данных переменной или объекта **во время выполнения** программы.

Существует множество реализаций такого механизма, но наиболее распространёнными являются:

- таблица указателей на объекты;

- хранение информации об объекте в памяти вместе с ним.

Таким образом, операция определения типа сводится либо к поиску в таблице, либо к просмотру нескольких байт до адреса, на который указывает указатель на объект.

В C++ для динамической идентификации типов применяются операторы **dynamic_cast** и **typeid**.

Пример

```
#include <typeinfo>     // для dynamic_cast
#include <iostream>
using namespace std;
class Base {
     virtual void vertFunc() {  }      // для dynamic cast
};
class Derv1 : public Base { };
class Derv2 : public Base { };
bool isDerv1( Base* pUnknown)
{          // неизвестный подкласс базового
Derv1* pDerv1;
if(pDerv1 = dynamic_cast<Derv1*>(pUnknown))
    return true;
else
    return false;
}
```

```
int main()
{
    Derv1* d1 = new Derv1;
    Derv2* d2 = new Derv2;
if(isDerv1( d1) )
     cout << "d1 - компонент класса Derv1\n";
else
     cout << "d1 - не компонент класса Derv1\n";
if(isDerv1( d2) )
     cout << "d2 - компонент класса Derv1\n";
else
     cout << "d2 - не компонент класса Derv1\n";
   return 0;
}
```

Еще пример

```
#include <iostream>
#include <typeinfo>     //   для typeid ( )
    using namespace std;
class Base {
    virtual void virtFunc() {}    //   для   typeid
};
class Derv1 : public Base { };
class Derv2 : public Base { };
void displayName(Base* pB) {
    cout << "указатель на объект класса "; // вывести имя класса
    cout << typeid(*pB).name() << endl; // на который указывает pB }
int main()  {
    Base* pBase = new Derv1;
    displayName(pBase); // "указатель на объект класса Derv1"
    pBase = new Derv2;
    displayName(pBase); // " указатель на объект класса Derv2“
 return 0;
}
```

Домашнее задание

Проект 28.

Создать абстрактный базовый класс **именем своей фамилии**, записанной латиницей. Например: **Ivanov**.

Создать 2 или 3 производных класса с именами измененного имени базового класса с суффиксами типа «_1» или «child_1», «child_2». Например: Ivanov_child_3.

Наполнить классы несколькими функциями (конструктор, деструктор, прочие операции) так, чтобы объекты классов вели себя по разному.

В функции main создать в динамической памяти несколько объектов типов созданных классов. Можно поместить их в единое хранилище (используя vector). Продемонстрировать различие в поведении этих объектов.

В функциях и членах классов надо использовать string, cout и итераторы. Хотя бы по одной- две операции из каждой темы. Например: string s1="test"; string s2= s1.substr(1,3); cout<<s2;

Контрольная работа 4

Создать базовый полиморфный класс **именем своей фамилии**, записанной латиницей. Например: **Ivanov**.

Определить производный класс названный по своему имени Например: **Ivan**.

Создать в базовом классе член-данных типа **float*, выделить память и** инициализировать ее произвольным числом в конструкторе по умолчанию.

Определить деструкторы в обоих классах.

В функции main создать два объекта типа vector< Base *>.

В один объект положить пару объектов полиморфной иерархии.

А потом провести глубокое копирование из 1 вектора во второй и не забыть освободить ресурсы.