

Санкт-Петербургский государственный университет телекоммуникаций
им. проф. М.А. Бонч-Бруевича

Объектно- ориентированное программирование

СПб ГУТ)))

Структурный подход

В основе структурного подхода лежит декомпозиция (разбиение на части) сложных систем с целью последующей реализации в виде отдельных небольших подпрограмм. С появлением других принципов декомпозиции (объектного, логического и т.д.) данный способ получил название процедурной декомпозиции. Поддержка принципов структурного программирования была заложена в основу так называемых процедурных языков программирования. Среди наиболее известных языков этой группы стоит назвать PL/1, ALGOL-68, Pascal, C.

Дальнейший рост сложности и размеров разрабатываемого программного обеспечения потребовал развития структурирования данных. Как следствие этого в языках появляется возможность определения пользовательских типов данных. Одновременно усилилось стремление разграничить доступ к глобальным данным программы, чтобы уменьшить количество ошибок, возникающих при работе с глобальными данными. В результате появилась и стала развиваться технология модульного программирования.

Модульное программирование предполагает выделение групп подпрограмм в отдельно компилируемые модули (библиотеки подпрограмм), например, модуль графических ресурсов. Связи между модулями при использовании данной технологии осуществляются через специальный интерфейс, в то время как доступ к реализации модуля (телам подпрограмм и некоторым “внутренним” переменным) запрещен. Эту технологию поддерживают современные версии языков Pascal и C (C++), языки Ада и Modula.

```

1  #include <cstdlib>
2  #include<iostream>
3  #include<string.h>
4  using namespace std;
5  struct Car{
6      char mark[4];
7      char color[4];
8      int age;
9  };
10 int main(int argc, char** argv) {
11     struct Car C;
12     strcpy(C.mark, "BMW");
13     strcpy(C.color, "RED");
14     C.age=10;
15     cout<<"Mark:"<<C.mark<<endl;
16     cout<<"Color:"<<C.color<<endl;
17     cout<<"Age:"<<C.age<<endl;
18     strcpy(C.mark, "VOLVO");
19     strcpy(C.color, "GREEN");
20     C.age=20;
21     cout<<"Mark:"<<C.mark<<endl;
22     cout<<"Color:"<<C.color<<endl;
23     cout<<"Age:"<<C.age<<endl;

```

main > strcpy(C.color, "GREEN") >

Терминал Уведомления Вывод x

asm_param (Собрать, Выполнение) x asm_param (Выполнение) x oop (Собрать, Выполнение) x oop (Выполнение) x

```

Mark: BMW
Color: RED
Age: 10
Mark: VOLVGR
Color: GREE
Age: 20

```

Объектно-ориентированное программирование (ООП) определяется как технология создания сложного программного обеспечения, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного типа (класса), а классы образуют иерархию с наследованием свойств. Взаимодействие программных объектов в такой системе осуществляется путем передачи сообщений.

Основным достоинством объектно-ориентированного программирования является “более естественная” декомпозиция программного обеспечения, которая существенно облегчает его разработку. Это позволяет вести практически независимую разработку отдельных частей (объектов) программы. Кроме этого, объектный подход предлагает новые способы организации программ, основанные на механизмах наследования, полиморфизма, композиции, наполнения. Эти механизмы позволяют конструировать сложные объекты из сравнительно простых. В результате существенно увеличивается показатель повторного использования кодов и появляется возможность создания библиотек классов для различных применений.

Бурное развитие технологий программирования, основанных на объектном подходе, позволило решить многие проблемы. Так, были созданы среды, поддерживающие визуальное программирование, например, Delphi, C++ Builder, Visual C++ и т.д. При использовании визуальной среды у программиста появляется возможность проектировать некоторую часть, например, интерфейсы будущего продукта, с применением визуальных средств добавления и настройки специальных библиотечных компонентов. Результатом визуального проектирования является заготовка будущей программы, в которую уже внесены соответствующие коды.

Объект в ООП — это совокупность переменных (свойств) и связанных с ними методов (операций). Упомянутые методы определяют, как объект взаимодействует с окружающим миром. Под методами объекта понимают функции, объявление которых включено в описание объекта и которые выполняют действия над его свойствами. Возможность управлять свойствами объекта посредством вызова методов в итоге и определяет поведение объекта. Эту совокупность методов часто называют интерфейсом объекта. Инкапсуляция — это механизм, который объединяет данные (свойства) и методы, манипулирующие этими данными, и защищает и то и другое от внешнего вмешательства или неправильного использования.

Применяя инкапсуляцию, мы защищаем данные, принадлежащие объекту, от возможных ошибок, которые могут возникнуть при прямом доступе к этим данным. Кроме того, применение этого принципа очень часто помогает локализовать возможные ошибки в коде программы. А это намного упрощает процесс поиска и исправления этих ошибок. Можно сказать, что инкапсуляция подразумевает под собой сокрытие данных, что позволяет защитить эти данные. Однако применение инкапсуляции ведет к снижению эффективности доступа к свойствам объекта. Это обусловлено необходимостью вызова методов для изменения свойств объекта. Однако при современном уровне развития вычислительной техники эти потери в эффективности не играют существенной роли.

Наследование — это процесс, посредством которого один объект может наследовать свойства другого объекта и добавлять к ним черты, характерные только для него. В итоге создается иерархия объектных типов, где данные и методы “предков” автоматически являются и данными и методами “потомков”. Основное достоинство наследования заключается в том, что не надо каждый раз заново (“с нуля”) описывать новый класс, а можно указать “родителя” (базовый класс) и описать отличительные особенности нового класса. В результате новый класс будет обладать всеми свойствами родительского класса плюс своими собственными отличительными особенностями.

Полиморфизм — это свойство, которое позволяет одно и то же имя использовать для решения нескольких технически разных задач. Полиморфизм подразумевает такое определение методов в иерархии типов, при котором метод с одним именем может применяться к различным родственным объектам. В общем смысле концепцией полиморфизма является идея “один интерфейс — множество методов”. Преимуществом полиморфизма является то, что он помогает снижать сложность программ, разрешая использование одного интерфейса для единого класса действий. Выбор конкретного действия, в зависимости от ситуации, возлагается на компилятор.

Классы.

Для реализации ООП в С++ используются классы. Для примера опишем класс, характеризующий студента.

```
class Student {  
private:  
    char* Fam;  
    char* Name;  
    int age;  
public:  
    void SetFam(char*NFam);  
    void SetName(char*NName);  
    void SetAge(int Nage);  
    char * GetFam(char*NFam);  
    char * GetName(char*NName);  
    int GetAge( );  
};
```

В данном классе описано 3 поля: Fam, Name, Age. Созданы методы для установки и получения информации из этих полей.
{ Set, Get — методы/ правила класса }
Поля определены в закрытой части класса, методы- в открытой.

Разделы класса

Public (компонент может быть использован любой функцией) { открытый раздел}

Private (компонент может быть использован только методами и друзьями класса)

Protected (то же самое, что Private + компоненты доступны из производных классов).

Хорошим стилем считается описывать свойства (данные) в закрытых разделах (private или protected), а методы в открытых(public).

Обычно описание класса помещают в заголовочный файл{*.h}, а реализацию методов класса помещают в файл{*.cpp}

```
void Student:: SetFam(char*NFam)
{
strcpy(Fam, NFam);
}
```

Это проще, но лучше делать с контролем памяти.

```
void Student:: SetFam(char*NFam)
{
if (strlen(NFam)>strlen(Fam))
{
delete Fam;
Fam=new char[strlen(NFam)+1];
}
strcpy(Fam,NFam);
}
```

Аналогичным образом реализуется метод
SetName

Метод SetAge реализуется проще:

```
void Student:: SetAge(int NAge)
{
    Age=NAge;
}
```

Реализация методов получения информации

```
int Student:: GetAge( )  
{  
    return Age;  
}  
char* Student:: GetFam( char*NFam)  
{  
    strcpy(NFam; Fam);  
    return Nfam;  
}
```

Если бы мы реализовали метод GetFam следующим образом

```
char * Student:: GetFam( )  
{  
    return Fam;  
}
```

то мы бы вернули адрес закрытого свойства класса. Таким образом программа пользователя, получив доступ к закрытой части класса, могла бы модифицировать это поле, минуя методы класса, что противоречит основам ООП.

В нашем случае (предыдущий слайд) в метод передается строка NFam, куда и будет скопировано свойство Fam. Метод GetFam вернет адрес именно этой строки.

Конструкторы и деструкторы.

Конструктор - это метод, который вызывается в момент создания объекта. Имя конструктора всегда совпадает с именем класса. Класс может иметь несколько конструкторов, при этом они отличаются числом или типом параметров.

Конструктор не имеет типа. Класс может не иметь конструктора, при этом считается, что компилятор сам создает конструктор. Конструктор без параметров называется конструктором по умолчанию. В основном конструкторы используются для инициализации полей класса. Особенно часто это нужно, если класс имеет динамические свойства.

Виды конструкторов

- **Конструктор по умолчанию**
- **Конструкторы с параметрами**
- **Конструктор копирования**

Рассмотрим класс Employee, описанный в строках 4-26.

```
4. class Employee /******* Класс "служащий"
5. {
6. public:
7. Employee( char *NName, char *NFam, char *NDep, int
   NAge);
8. virtual void print();
9. void SetName( char *NName);
10. int GetAge();
11. char *GetDep( char *NDep);
12. char *GetFam( char *NFam);
13. char *GetName( char *NName);
14. void SetAge( int NAge);
15. void SetDep( char *NDep);
16. void SetFam( char *NFam);
17. Employee();
18. virtual ~Employee();
19. friend istream& operator >>(istream&,Employee&);
20. friend ostream& operator <<(ostream&,Employee&);
```

```
21. protected:
22. int Age;
23. char *Dep;
24. char *Name;
25. char *Fam;
26. };
```

Этот класс описан в заголовочном файле Employee.h.
Он имеет 3 динамических свойства:

Dep

Name

Fam

строки(23-25).

Класс имеет конструктор по умолчанию(стр. 17) и конструктор с параметрами(стр. 7).

Реализация конструктора приведена в строках 75-84, которые находятся в файле Employee.cpp.

```
75. Employee::Employee()  
76. {  
77.   Name=new char[strlen("no name")+1];  
78.   strcpy(Name,"no name");  
79.   Fam=new char[strlen("no fam") +1];  
80.   strcpy(Fam,"no fam");  
81.   Dep=new char[strlen("no dep") +1];  
82.   strcpy(Dep,"no dep");  
83.   Age=0;  
84. }
```

Действия конструктора:

- выделяет память;
- копирует в нее строки по fam, по name...

В строках 135-144 приведена реализация конструктора с параметрами.

```
135. Employee::Employee(char *NName, char
    *NFam, char *NDep, int NAge)
136. {
137.   Name=new char[strlen(NName)+1];
138.   strcpy(Name,NName);
139.   Fam=new char[strlen(NFam)+1];
140.   strcpy(Fam,NFam);
141.   Dep=new char[strlen(NDep)+1];
142.   strcpy(Dep,NDep);
143.   Age=NAge;
144. }
```

Вызывается конструктор в момент создания объекта класса. В строке 288 создается объект x типа Employee путем вызова конструктора умолчания.

288. Employee x;

289. Employee y("AA","BB","CC",100);

В строке 289 создается объект y путем вызова конструктора с параметрами.

В строке 290 создается и инициализируется массив z типа Employee.

290. Employee

z[]={Employee("AA1","BB1","CC1",1001),Employee("AA2","BB2","CC2",1002),Employee("AA3","BB3","CC3",1003)};

Деструктор - это метод, который вызывается при уничтожении объекта. Имя деструктора всегда совпадает с именем класса; начинается со значка~
Класс Employee имеет деструктор (строка 18).

18. virtual ~Employee();

Реализация деструктора приведена в строках 85-90.

```
85.     Employee::~~Employee()  
86.     {  
87.         delete [ ] Name;  
88.         delete [ ] Fam;  
89.         delete [ ] Dep;  
90.     }
```

Как видно из этих строк, он освобождает выделенную ранее динамичную память. Деструктор, как и конструктор, нельзя вызвать напрямую. Деструктор не имеет типа и не принимает параметров.

Класс может иметь только один деструктор (конструкторов много) или не иметь его вообще. В основном деструкторы применяются, если у класса имеется динамические поля. В этом случае действия деструктора заключается в освобождении динамической памяти.

Конструктор копирования

Конструктор копирования нужен для того, чтобы создавать «реальные» копии объектов класса, а не побитовую копию объекта. Иногда это принципиально важно. Такую «реальную» копию объекта надо создавать в нескольких случаях:

- когда объект передается в какую-либо функцию в виде параметра;
- когда функция должна вернуть объект класса в результате своей работы;
- когда мы один объект класса инициализируем другим объектом класса.

Например, мы передаем объект в функцию в виде параметра. Функция будет работать не с самим переданным объектом, а с его побитовой копией. Допустим в [конструкторе класса](#), при создании объекта, выделяется определенный объем памяти, а деструктор класса эту память освобождает. Указатель побитовой копии объекта будет хранить тот же адрес памяти, что и оригинальный объект. Когда при завершении работы функции и уничтожении побитовой копии объекта, сработает деструктор, он обязательно освободит память, которая была занята объектом-оригиналом. В придачу, еще и при завершении работы программы, деструктор сработает повторно и попытается еще раз освободить этот объем памяти, что неизбежно приведет к ошибкам программы. Та же участь постигнет и память, выделенную для указателя объекта, если будет удаляться побитовая копия возвращаемого функцией объекта, и побитовая копия при инициализации объекта класса другим объектом.

- Чтобы избежать этих проблем и ошибок существует конструктор копирования. Его работа заключается в том, чтобы создать реальную копию объекта со своей личной выделенной динамической памятью. Синтаксис конструктора копирования следующий:

имяКласса (const имяКласса & object)

```
{  
    //код конструктора копирования  
}
```

- Рассмотрим простой пример. В нем создадим класс, который будет содержать обычный конструктор, конструктор копирования и деструктор. В этом примере будут описаны все три случая для которых надо применять конструктор копирования. Для того, чтобы пример не был слишком большим, мы не будем заставлять конструкторы выделять память, а деструктор освобождать память. Пропишем в них только вывод определенного сообщения на экран. Таким образом, можно будет посмотреть, сколько раз сработают конструкторы и деструктор. Как вы понимаете, если бы деструктор освобождал память, то количество его вызовов не должно превышать количество вызовов конструкторов.

```

1. #include <iostream>
2. using namespace std;
3.
4. class SomeClass
5. {
6.     int *ptr; // указатель на какой-либо участок памяти
7. public:
8.
9.     SomeClass() // конструктор
10. {
11.     cout << "\nОбычный конструктор\n";
12. }
13. SomeClass(const SomeClass &obj)
14. {
15.     cout << "\nКонструктор копирования\n";
16. }
17. ~SomeClass()
18. {
19.     cout << "\nДеструктор\n";
20. }
21. };
22. void funcShow(SomeClass object)
23. {
24.     cout << "\nФункция принимает объект, как параметр\n";
25. }
26. SomeClass funcReturnObject()
27. {
28.     SomeClass object;
29.     cout << "\nФункция возвращает объект\n";
30.     return object;
31. }

```

В строках 4 — 21 определен класс. Обычный конструктор, строки 9 — 12, будет автоматически вызваться при создании новых объектов класса. Конструктор копирования, строки 13 — 16, будет автоматически вызваться при создании копии объекта. Деструктор, строки 17 — 20, вызывается всякий раз когда удаляется либо объект, либо его копия. Ниже определены две функции funcShow() и funcReturnObject(). Первая принимает объект в виде параметра, вторая при вызове впервых создает новый объект (тут сработает обычный конструктор), а во-вторых возвращает объект (тут сработает конструктор копирования т.к. при возврате объекта из функции создается его временная копия).

```

32. int main()
33. {
34. setlocale(LC_ALL, "rus") ;

35. cout << "1 ~~~~~\n";
36. SomeClass obj1; // создаем объект класса

37. cout << "2 ~~~~~\n";
38. funcShow(obj1); // передаем объект в функцию

39. cout << "3 - 4 ~~~~~\n";
40. funcReturnObject(); // эта функция возвращает объект

41. cout << "5 ~~~~~\n";
42. SomeClass obj2 = obj1; // инициализация объекта при создании
43. }

```

```

1 ~~~~~
Обычный конструктор
2 ~~~~~
Конструктор копирования
Функция принимает объект, как параметр
Деструктор
3 - 4 ~~~~~
Обычный конструктор
Функция возвращает объект
Конструктор копирования
Деструктор
Деструктор
5 ~~~~~
Конструктор копирования
Деструктор
Деструктор
Для продолжения нажмите любую клавишу . .

```

Объекты в динамической памяти.

Пример.

```
Student *St;
```

```
St=new Student;
```

St-указатель на объект типа Student. При выполнении оператора new создается объект типа Student путем вызова конструктора умолчания, его адрес присвоится указателю St. Если имеется указатель на объект, то для вызова его методов вместо точки используется->

Например:

```
St-> SetAge(100);
```

Для удаления объектов из динамической памяти используется оператор delete.

```
Student *St;  
St=new Student;  
St-> SetAge(100);  
delete St;
```

Наследование

Процесс, посредством которого один объект может наследовать свойства другого объекта, добавлять к ним черты, характерные только для него. (Автомобиль-автобус).

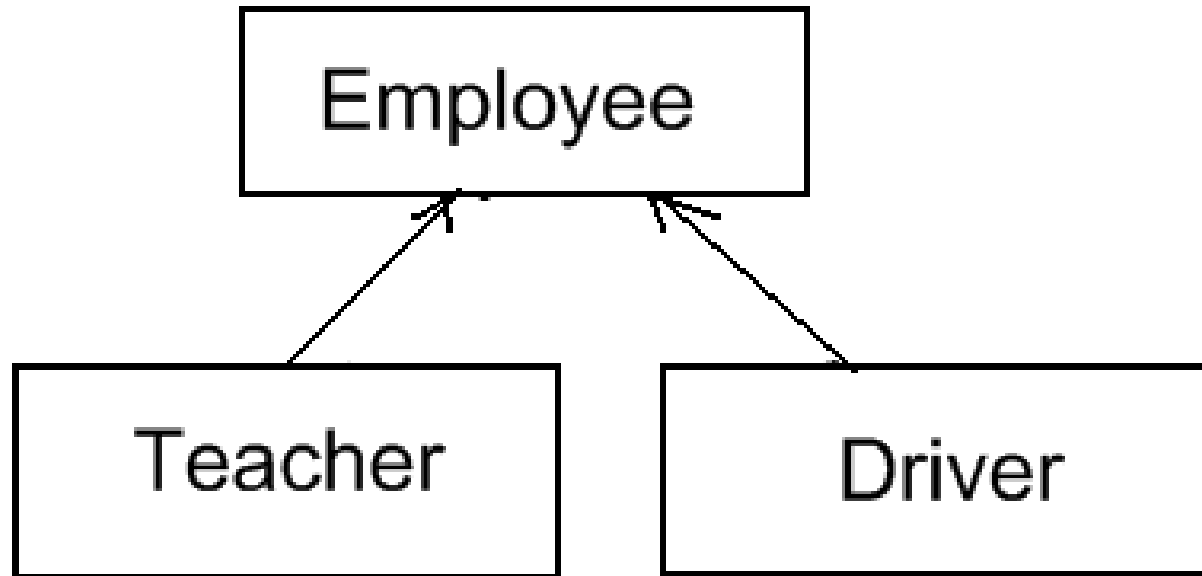
Таким образом не надо каждый раз описывать все с нуля. Это и есть главное преимущество наследования.

В ООП класс, от которого наследуют, принято называть базовым/ родительским.

Класс, который получается называется порожденным/производным.

Пример наследования.

В строках 4-26 описан класс Employee, характеризующий сотрудника какой-либо компании. Породим от этого класса 2 класса Teacher и Driver, характеризующих водителя и учителя



Класс Driver описан в файле Driver.h (стр. 32-46), класс Driver является производным от класса Employee (стр32).

```
class Driver : public Employee
{
public:
    Driver(char *NName, char *NFam, char *NDep, int NAge, char *NCarNum, int NLen);
    void print();
    char * GetCarNum( char *NCarNum);
    int GetLen();
    void SetCarNum( char *NCarNum);
    void SetLen( int NLen);
    Driver();
    virtual ~Driver();

protected:
    int Len;
    char * CarNum;
};
```

Класс Driver унаследовал от Employee все его поля и методы. Помимо этого, класс Driver имеет 2 собственных поля- характеризующие водителя (номер машины, водительский стаж) (стр. 44,45).

Класс Driver имеет методы Set и Get для управления этими полями (стр. 38-40). Как и класс Employee, класс Driver имеет 2 конструктора и деструктор + метод print для распечатки полей.

Методы класса Driver во многом похожи на методы класса Employee. Особое внимание следует уделить конструктору с параметрами. При создании объекта производного класса сначала выполняются конструкторы базовых классов в порядке их перечисления, затем конструктор производного класса.

