Введение
В 50-60-х гг. XX в. возникла проблема необходимости произведения сложных по тем временам вычислений, а вычисления выполнялись на механических средствах. Возникла проблема: трудно программировать в кодах ЭВМ (60-е). Появились языки Фортран, Фокал (60-е). Основа методологии: алгоритмическая организация программ (60-е). Проблема: увеличение сложности программ; решение - процедуры (60-е). Дальнейшее увеличение сложности -> процедурная декомпозиция (70-е). дальнейшее увеличение сложности -> методология структурного программирования (80-е).

80-90-е гг.: проблема множества задач; в т. ч. невычислительного характера -> объектно-ориентированный подход проектирования, основан на представлении предметной области задачи в виде множества моделей для независимой от языка разработки программной системы. Модель содержит не все признаки и свойства представляемого ею свойства или понятия, а только те, которые существенны для разрабатываемой программной системы. Простота модели по отношению к реальному предмету позволяет сделать её формальной. Благодаря этому можно чётко выделить все зависимости и операции над моделью, создаваемой в программной системе. Это упрощает анализ задачи, разработку и реализацию программного продукта.
Преимущества объектно-ориентированного подхода:
 1) уменьшение сложности ПО;
 2) повышение надёжнрсти ПО;
 3) обеспечение возможности изменения отдельных компонентов ПО без модификации остальных;
 4) обеспечение возможности повторного использования отдельных компонентов.
Организация соревнований:
 1) приглашение спортсменов;
 2) договор со спортивными сооружениями;
 3) продажа билетов;
 4) бронирование мест в гостиннице;
 5) составление расписания;
 ...

 Оргкомитет Гостинницы
Спортивные сооружения

Расписание соревнований
Транспорт
   Спортсмены

Объект - мыслимая либо реальная сущность, обладающая характерным поведением и отличительными характеристиками и являющаяся важной предметной областью, обладает чётко определёнными поведениями и уникальной идентичностью. Состояние - это совокупный результат поведения объекта, одно из стабильных условий, в которых объект может существовать, охарактеризованный количественно.
Поведение
Для каждого объекта существует определённый набор действий, которые с ним можно производить, например: набор действий с файлом в операционной системе: 1) создать; 2) открыть; ... Результат выполнения действий зависит от состояния объекта на момент совершения действия. Программа, написанная с применением объектно-ориентированного подхода, обычно состоит из множества объектов, которые взаимодействуют между собой. Обычно говорят: взаимодействие реализуется путём передачи сообщений между объектами. В ООП синонимами сообщения являются действие и метод.
Внешний интерфейс объекта - это описание того, какие сообщения может принимать объект. Поведение - это действия и реакции объекта, выраженные в терминах передачи сообщений и изменения состояния; видимая извне, воспроизводимая активность объекта.
Уникальность - то, что отличает один объект от другого.

Классы
Все объекты одного и того же класса описываются одинаковым набором аттрибутов. Формально класс - это шаблон поведения объектов определённого типа с заданными параметрами, определяющими состояние. Все экземпляры одного класса (объекты, порождённые от одного класса) имеют один и тот же набор свойств и общее поведение, то есть одинаково реагируют на одинаковые сообщения.
Инкапсуляция - это сокрытие организации класса и отделение его внутреннего представления от внешнего интерфейса. Благодаря сокрытию реализации за внешним интерфейсом можно менять внутреннюю логику отдельного класса, не меняя код остальных компонентов системы; это свойство называется модульностью.
Наследование - это отношение между классами, при котором класс использует структуру или поведение другого класса (одиночное наследование) или других классов (множественное наследование). Наследование входит в иерархию "общее - частное", в котором подкласс наследуется от одного или нескольких более общих суперклассов.
В языке С может использоваться множественное наследование.
Полиморфизм является одним из фундаментальных понятий ООП наряду с наследованием и инкапсуляцией. В переводе с греческого означает "множество форм".
Стили программирования

Стиль программирования - это способ построения программы, основанный на определённых принципах программирования и выборе языка с целью достижения ясности в понимании программы. Существует ряд стилей программирования: процедурно-ориентированный, логический, основанный на правилах и т. д. Каждый стиль программирования ориентирован на решение определённого круга задач. Объектно-ориентированный стиль программирования лучше всего подходит для задач большой сложности. Объектно-ориентированному стилю программирования соответствует 4 основных требования:
 1) абстрагирование;
 2) ограничение доступа;
 3) модульность;
 4) иерархия;
и 3 дополнительных:
 1) типизация;
 2) параллелизм;
 3) устойчивость.
Абстрагирование, абстракция - это такие существенные характеристики некоторого объекта, которые отличают его от всех остальных видов объектов и таким образом чётко определяют особенности данного объекта с точки зрения дальнейшего рассмотрения и анализа. Выбор достаточно полного множества абстракций для заданной предметной области является главной задачей объектно-ориентированного анализа.
Модульность - разделение программы на фрагменты позволяет уменьшить её сложность и улучшить проработку её частей. Если классы и объекты образуют логическую структуру программы, то модули образуют физическую структуру. Обычно модульность реализуется путём разделения программы на отдельно компилируемые фрагменты.
Типизация - это ограничение, накладываемое на класс объектов, которое препятствует взаимозамене различных классов. Типизация поддерживает описание абстракций на уровне языка программирования.
Объектно-ориентированные языки программирования подразделяются на три категории: строго типизированные, нестрого типизированные и совсем не типизированные. В строго типизированном языке все конструкции проходят проверку на соотвествие определённым ранее типам.
Параллелизм: если система должна обрабатывать события, происходящие одновременно, либо время выполнения вычислений не позволяет использовать только один процессор, то вычисления надо распределить между несколькими процессорами.
Устойчивость: любой объект в программе занимает определённое место и существует в течение определённого времени. Устойчивость - это свойство объекта существовать во времени вне зависимости от процесса, породившего данный объект (и в скорости перемещения объекта в адресном пространстве). В порядке возрастания устойчивости можно рассмотреть следующие данные:
 1) промежуточные результаты вычисления выражений: y=(a+b)/(c-d);
 2) локальные переменные в вызове процедур;
 3) глобальные переменные;
 4) данные, сохраняющиеся между вызовами основной программы;
 5) данные, остающиеся без изменений в разных версиях программы;
 6) данные, которые переживают все версии программы.
В настоящее время ни один из распространённых объектно-ориентированных языков программирования не поддерживает в полной мере концепцию устойчивости.

Объектно-ориентированное программирование на языке C++

В языке C++ используются три конструкции, которые в той или иной мере используют понятие "класс": class. struct, union.
Упрощённая структура класса:
 {class}-{имя_класса}-{{}-{спецификатор доступа}-{описание данных}-{спецификатор доступа}-{объявление функций класса}-{}}-{;}
Проблема: некоторая организация (напр., строительная) желает автоматизировать отчёты данных для того, чтобы можно было назначать и определять должность работника, сведения о зарплате, следить за адресами и т. д.
```
 class Worker
 { char name[16];
   char post[25];
   char address[50];
   double salary;
   public:
   void init(char*,char*, char*,double);
   void setPost(char*);
   void setAddr(char*);
   void setSal(double);
   void print();
   char* getName();
   char* getPost();
   char* getAddr();
   double getSal();
 }
 void Worker::init(char* n,char* p,char* a,double s) {

strcpy(name,n);strcpy(post,p);strcpy(address,a);salary
=s;
 }
 void Worker::setPost(char* p){
  strcpy(post,p);
 }
 ...
 char Worker::getName()
 { char* t=new char[15];
   strcpy(t,name);
   return t;
 }
 char Worker::getSal()
 { return salary;
 }
 void main(){
  Worker w1;
  w1.init("Иванов","мастер","Серов, 5, Армейская, 15,
7",4500);
  w1.setSal(4500);
  w1.print();
  Worker w2; w2.init("Петров","штукатур","Одесса,
т.Шевченко, 5, 21", 3700);
  w2.print();
 }
```
Доступ к членам класса в методах класса осуществляется благодаря локальной переменной this, доступной в любой методе (неявный параметр). Переменная this содержит указатель на объект класса. В редких случаях указатель this используется программистом в явном виде.

inline-функции

Как правило, класс содержит очень большое количество мелких методов типа get и set. При вызове таких методов много времени расходуется на передачу управления по адресу и возврат значений в вызываемую программную единицу. Уменьшение времени на выполнение таких методов может быть обеспечено использованием inline-функций.
Функция, описанная в теле класса. автоматически становится inline-функцией. Функции, использующие операторы управления ходом вычислениями, не могут становиться inline-функциями.
Пример:
```
class location
{ int x,y;
class Location
{ int x,y;
  public:
  void init(int x1, int y1){x=x1;y=y1;}
  int getX(){return x;}
  int getY(){return y;}
  void setX(int x1) {x=x1;}
  void setY(int y1) {y=y1;}
  void print(){cout<<"x="<<x<<"y="<<y;}
};
inline void Worker::setSal(double s){
}
```
Конструкторы в языке объектно-ориентированного программирования, как правило, предлагают использовать в классе специальный метод, называемый конструктором. Цель этого метода - создание объекта данного класса. Конструкторы нужны для того, чтобы правильно проинициализировать создаваемый объект, создать этот объект только один-единственный раз; для того, чтобы улучшить контроль кода программы, поскольку объекты можно будет создавать только через конструктор. Имя конструктора всегда совпадает с именем класса, в конструкторе не записывается возвращаемое значение, программа может иметь несколько конструкторов, которые должны отличаться набором и типом параметров; в конструктор, в отличие от других методов класса, не передаётся ссылка на объект.
```
class Worker
{ char name[15];
  char post[20];
  char address[50];
  double salary;
  public:
  Worker(char*,char*,char*,double);
```
Строительная организация принимает большое количество разнорабочих. Это не местные жители, их поселяют в общежитие фирмы и выплачивают всем определённую зарплату.
```
  Worker(char x);
}
Worker::Worker(char* n, char* p, char* a, double s)
{ strcpy(name,n); strcpy(post,p); strcpy(address,a);
salary=s;
Worker::Worker(char* n)
{ strcpy(name,n); strcpy(post,"разнорабочий");
strcpy(address, "Общежитие № 1"); salary=2000;
}
```

```
void main()
{ Worker w1("Иванов", "бригадир", "Одесса...", 5100);
  Worker w2("Чан Кай Ши");
  Worker w3("Исмаилов");
  w1.print();
  w3.setSal(3000);
}
```

Списки инициализации в конструкторе

```
class Location
{ int x,y;
  public:
  Location(int x1, int y1): x(x1),y(y1){}
  ...
};
```

Использование параметров "по умолчанию"

```
class Location
{ int x,y;
  public:
  Location(int x1, int y1=0) {x=x1;y=y1;}
  ...
}

void main()
{ Location L1(5,7), L2(3);
  L1.print();//x=5,y=7
  L2.print();//x=3,y=0
}
```

Ссылочные переменные

Ссылка - альтернативное имя ранее объявленной переменной

```
int a;
int& ra=a;
...
a=5;
cout<<"a="<<a<<"ra="<<ra;//a=5ra=5
```

Ссылка в качестве параметров функций

```
void input(double* x1, double *x2)
{ cout<<"\n Введите 2 числа:";
  cin>>*x1>>*x2;
void main()
{ double a,b;
  input(&a,&b);
}
void input(double &x1, double &x2)
{ cout<<"\nВведите 2 числа:";
  cin>>x1>>x2;}

void main(){
{ double a,b;
 input(a,b);
...
```

При неаккуратном использовании ссылок могут возникнуть побочные эффекты.

```
double sq(double &x)
{ x*=x;
  return x;
}
void main()
{ double a,y; a=3;
  y=sq(a);
  cout<<"\ny="<<y;//9
  cout<<"\na="<<a;//9
  ...
}
{return x*x;}
```

```
//Ссылку можно защитить от изменений. Для этого используется модификатор const.

double sq(const double &x)
{ return x*x; }

void main()
{ double a=3;
  double b=2.5;
  cout<<"\n"<<sq(a);
  cout<<"\n"<<sq(b+a-1.5);
}
```

Если фактический параметр не соответствует формальному подтипу (но совместим либо является константой либо выражением, то компилятор создаёт временную переменную и ссылка производится на неё.

Рекомендации по использованию ссылок в качестве параметров

Рекомендуется всегда (при возможности) использовать модификатор const. Это определяется следующим:
 1) применение const предотвращает появление программных ошибок, которые могут изменить данные;
 2) применение const позволяет функции выполнять обработку фактических параметров как определённых (с модификатором const), так и без модификатора const, в то время как для функций, у которых нет модификатора const, нужно передавать данные, не имеющие статуса const;
 3) использование ссылки с модификатором const позволяет компилятору генерировать и использовать временные переменные по своему усмотрению.

Конструктор копирования

Конструктор, который может быть вызван для создания объекта класса, идентичного существующему объекту, называется конструктором копирования. Конструктор копирования имеет всегда определённый вид заголовка: x::x(const x & x1).
Пример:

```
class x
{ int a;
  double d;
  char *s;
  public:
  x(int,double,char*);
  x(const x &);
  void print();
  int getA();
  double getD();
  char* getS();
  void setA(int);
...
};
x::x(int a1, double d1, char* s1)
{ a=a1; d=d1; s=new char[strlent(s1)+1]; strcpy(s,s1);
x::x(const x & x1)
{ a=x1.a; d=x1.d; s=new char[strlent(x1.s)+1];
strcpy(s1,x1.s);
}
void main()
{ x x1(2,1.5,"строка1"), x2(3,3.7,"строка2");
  x x3(x1);
  x1.print();
  x3.print();
  x x4=x2;
...
}
```

Наследование

Класс A может быть описан как порождённые от класса B. При этом говорят, что он является дочерним, производным, подклассом.

[B]<-[A]

```
                                        +--------------------v
->(class)->[имя класса]->(:)-+->[аттрибут
доступа]--->[имя родительского класса]->({)-[тело
класса]-(})-(;)->

^---------(,)-----------------------------------------+
```

Таблица доступа к членам порождённого класса

Доступ в базовом классе при определении наследования	Аттрибут доступа в порождённом классе
public	public
public	private
private	public
private	private
protected	public
protected	protected
protected	private
private	

```
class x{
 int a;
 public:
 int b;
 protected:
 int c;
 public:
 int d;
.........
};
class y: public x{
 .............
 public:
 ..............
 void print(){
  cout<<"\n a="<<a//ошибка
  <<" b="<<b//ошибка
  <<" c="<<c
  <<" d="<<d;
 }
 ..............
};
void main(){
 +------------+
 |            |
 +============+
```

Переопределение членов базового класса в производном

```
class x{
 protected:
 int a,b;
 public:
 x(int a1, int b1){ a=a1; b=b1; }
 void print(){ cout<<"\n a="<<a<<" b="<<b; }
};
class y: public x{
 int c;
 double b;
 public:
 y(int a2, int b1, int c1, double b2): x(a1,b1){
  c=c1; b=b2;
 }
 void print(){
  cout<<"\n a="<<a<<" x::b="<<x::b<<" c="<<c<<"
b="<<b;
 }
 ..............................
}
void main(){
 x x1(1,2);
 y y1(3,4,5,7.7);
 y1.print();//3 4 5 7.7
 x1.print();//1 2
 y1.x::print();// 3 4
}
```

Изменение вида доступа для члена базового класса в порождённом классе
Допускается изменение вида аттрибута доступа для базового класса в производном классе, но только для восстановления того типа доступа, который существовал в базовом классе.
Пример:

```
class x{
 int a;
 private:
 int b;
 protected:
 int c;
 public:
 int d;
 ........................
};
class y: private x{
 ........................
 public:
 x::a;//ошибка
 x::b;//ошибка
 x::c;//ошибка
 x::d;
 protected:
 x::a//ошибка
 x::b//ошибка
 x::c;
}
```

Необходимо внести изменения в класс Worker для реализации повременной оплаты и доплаты за стаж.

```
class Worker{
 protected:
 char name[15];
 char address[50];
 char post[30];
 double salary;
 .................
};
class WorkerM: public Worker{
 protected:
 int time;
 double tariff;
 int staj;
 int t1,t2;
 double k1,k2;
 public:

WorkerM(char*,char*,char*,double,int,int,double,double
);
 void print();
 void showSal();
 void getSal();
 void setTime(int);
 void setStaj(int);
 ...............
};
WorkerM::WorkerM(char* n, char* a, char* p, double tf,
int st, int t11, int t21, double k11, double
k21):Worker(n,a,p,0){
 tariff=tf; staj=st; t1=t11; t2=t21; k1=k11; k2=k21;
time=0;
}
```

```
void WorkerM::showSal(){
 double s=time*tariff;
 if (staj>=t2) s*=k2;
 else if (staj>=t1) s*=k1;
 salary+=s; time=0;
 cout<<"\n"<<name" начислено:"<<salary;
}
void WorkerM::getSal(){
 showSal();
 cout<<"\n"<<name<<" выплачено: "<<salary;
 salary=0;
}
//вариант
double WorkerM::getSal(){
 ..................
 double s=salary;
 salary=0;
 return s;
}
void WorkerM::setTime(int t){
 time=t;
}
```

Совместимость объектов родительских и порождённых классов

```
class x{
 protected:
 int a,b;
 public:
 x(int,int);
 ..................
};
class y: public x{
 protected:
 int c;
 public:
 y(int,int,int);
 ......................
};
class z:public y{
 protected:
 int d;
 public:
 z(int,int,int,int);
.....................
};
```

```
Объект класса x       Объект класса y       Объект класса z
+-------+             +---------+           +-------+
|   a   |<------------|    a    |<----------|   a   |
+=======+             +---------+           +-------+
|   b   |<------------|    b    |<----------|   b   |
+-------+             +---------+           +-------+
                      |    c    |<----------|   c   |
                      +---------+           +-------+
                                            |   d   |
                                            +-------+
```

Правила совместимости:

1) объекту родительского класса можно присвоить объект порождённого класса;
2) указателю или ссылке на объект родительского класса можно присвоить указатель или адрес объекта порождённого класса;
3) формальному параметру - объекту, указателю или ссылке на объект родительского класса можно присвоить объект-указатель адрес порождённого класса соответственно.

```
class x{
public y
 ..........
.........
 x(........);
z(...............);
 ..............
.........
```

```
class y: public x{
 .................
 y(.............);
 ...............
```

```
class z:
 .......
.......
};
```

```
void main(){
 x x1(........);
 x *px=new x(....);
 y y1(......);
 y *py=new y(.....);
 z z1(.........);
 z *pz=new z(.....);
 x1=y1;
 x1=z1;
 px=py;
 px=pz;
 y1=z1;
 py=pz;
 px=&y1;
 py=&z1;
 ............
}
void f1(x, &y, .....){ ............};
 f1(x1,z1,......);
 f1(z1,x1,.......);
 .................
```

Концепция полиморфизма

Понятие полиморфизма является одной из фундаментальных составляющих ООП. Оно позволяет в родительском классе определить методы, которые будут общими для всех последующих классов. При этом последующий класс может определять специфическую реализацию для некоторых (возможно, всех) полиморфных методов). Важнейший принцип полиморфизма "один интерфейс - несколько методов". Полиморфизм - это техника, позволяющая устанавливать некоторые элементы поведения родительского класса такими же, как и в порождённом классе. Основными механизмами для реализации полиморфизма в ООП являются виртуальные методы и абстрактные элементы.

Виртуальные методы

Метод, при опеределении которого в родительском классе указано ключевое слово virtual и который был переопределён в одном или более наследующих классах, называется виртуальным методом, следовательно, каждый наследующий класс может иметь собственную конкретизацию виртуального метода. Выбор конкретизации виртуального метода осуществляется в соответствии с типом объекта, на который указывает ссылочная переменная во время выполнения программы. То есть не тип ссылочной переменной, а тип объекта, на который указывает ссылка, определяет версию виртуального метода.

Полиморфизм - это техника, позволяющая устанавливать некоторые элементы поведения родительского класса такими же, как и в порождённом классе.

```
+-------------+       +-----------+
| Родительский|<------|Порождённый|
|    класс    | ссылка|  класс    |
+-------------+------>+-----------+
|вир. метод   |       |вир. метод |
+-------------+       +-----------+
```

Если класс x содержит виртуальную функцию vf, а класс y, порождённый от класса x, также содержит функцию vf с той же сигнатурой, то обращение к vf для объекта класса x вызовет vf из y при доступе через указатель или ссылку на объект класса y. В этом случае говорят, что функция порождённого класса подменяет функцию родительского класса.

Правила использования виртуальных методов:

1) если сигнатуры функций в родительском и порождённом классах различны, то механизм виртуальности не включается;
2) виртуальную функцию родительского класса нельзя переопределить функцией в порождённом классе, которая отличается от неё только типом возвращаемого значения;
3) если функция объявлена виртуальной, то все функции с такой же сигнатурой и возвращаемым значением в порождённых классах становятся виртуальными;
4) виртуальные функции не могут быть статическими или дружественными по отношению к другим классам;

5) виртуальные функции наследуются.

```
class x{
 protected:
 int a;
 public:
 x(int a1){a=a1;};
 ..............
 virtual void show(){
  cout<<"\nфункция класса x";
 }
 ..............
};
class y: public x{
 protected:
 int b;
 public:
 y(int a1,int b1):x(a1){b=b1;};
 void show(){
  cout<<"\nфункция класса y";
 }
 ........................
};
class z: public y{
 protected:
 int c;
 public:
 z(int a1,int b1, int c1):y(a1,b1){c=c1;};
 void show(){
  cout<<"\n функция класса z";
 }
 ........................
};
void show(x&x1){
 x1.show();
}
void main(){
 x x1(1);
 y y1(2,3);
 z z1(4,5,6);
 x1.show();//.. класс x
 y1.show();//.. класс y
 z1.show();//.. класс z
 x *px;
 px=&x1;
 px->show();//..класс x
 px=&y1;
 px->show();//..класс y
 px=&z1;
 px->show();//..класс z
 y *py;
 py=&x1;//ошибка
 py=&y1;
 py->show();//..класс y
 py=&z1;
 py->show();//..класс z
 show2(x1);//..класс x
 show2(y1);//..класс y
 show2(z1);//..класс z
};
```

Позднее связывание

Во время компиляции с функцией show2 неизвестно, какой метод будет вызван, определение адреса метода происходит во время исполнения программы. Такой механизм называется поздним связыванием. Позднее связывание замедляет выполнение программы.

Реализация позднего связывания: для каждого класса (не объекта), содержащего хотя бы один виртуальный метод, компилятор создаёт таблицу виртуальных методов (vtbl), в которой для каждого виртуального метода записан его адрес в памяти. Адреса методов располагаются в таблице в порядке их описания в классах, поэтому адрес каждого конкретного виртуального метода имеет в таблице виртуальных методов одно и то же смещение для каждого класса в цепочке иерархии. Каждый объект содержит скрытое дополнительное поле ссылки на таблицу виртуальных методов, называемое обычно vptr. Оно заполняется конструктором при создании объекта. Для этого компилятор добавляет в начало тела конструктора соответствующие конструкции. На этапе выполнения в момент обращения к методу его адрес выбирается из таблицы.

```
class1 <----------------- class2
vtbl                      vtbl
+---+                     +---+
|Vm1+---------+           |Vm1+-------+
+---+         |           +---+       |
|Vm2+-------+ |           |Vm2+-----+ |
+---+       | |           +---+     | |
|Vm3+----+  | |           |Vm3+---+ | |
+---+    |  | |           +---+   | | |
+------+ |  | |           |Vm4+-+ | | |
|кодVm1|<+  | |           +---+ | | | |
+------+    | |       +------+ | | | |
            | |       |кодVm1|<+ | | |
+------+    | |       +------+   | | |
|кодVm2|<---+ |                  | | |
+------+      |       +------+   | | |
              |       |КодVm2|<--+ | |
+------+      |       +------+     | |
|кодVm3|<-----+                    | |
+------+                  +------+ | |
                          |КодVm3|<----+
                          +------+
                                   |
                          +------+ |
                          |КодVm4|<------+
                          +------+
```

```
class *O1;
O1=new Class1();
+-------------
|               |
  ---------------+
```

Схема работы виртуальных методов в приложениях

Родительский класс обозначает интерфейс при помощи виртуальных функций, а порождённые классы обеспечивают набор реализаций для этого интерфейса.

Пример: создать класс "товар" с данными "название", "год выпуска", "количество", "стоимость одной единицы товара". Нужно определить функцию "остаток", которая может выполнять различные действия в зависимости от вида товара.

```
class Goods{
 protected:
 char *name;
 int year;
 int quant;
 double price;
 public:
 Goods(char*,int,int,double);
 void setQ(int);
 void setP(double);
 char* getN();
 .................
 virtual void showRest(){
  cout<<"\nОстаток товара";
 }
 .....................
};
class GSug: public Goods{
 public:
 GSUg(char *n,int g,int q, double p):Goods(n,y,q,p){}
 void showRest(){
  cout<<"\nостаток товара \"сахар\" в количестве"<<50*quant<<" кг";
 }
};
```

Нужно определить функцию-остаток для сахара, вывести название и общий вес.

Нужно определить функцию-остаток для телевизоров, при

этом должно быть сформулировано название и общая стоимость.

```
class GTV: public Goods{
 public:
 GTV (char *n, int y, int q, double p):Goods(n,y,q,p)
{}
 void showRest(){
  cout<<"\n осталось"<<quant<<" телевизоров марки
"<<name<<" общей стоимостью "<<quant*price;
 }
};
void main(){
 Goods *pg;
 GSug s1("сахар песок",2012,99,6);
 CTV t1("Panasonic",2011,8,3580);
 pg=&s1;
 pg->showRest();
 pg->&t1;
 pg->showRest();
 ..................
};
```

Пустые виртуальные функции (NULL-виртуальные функции)

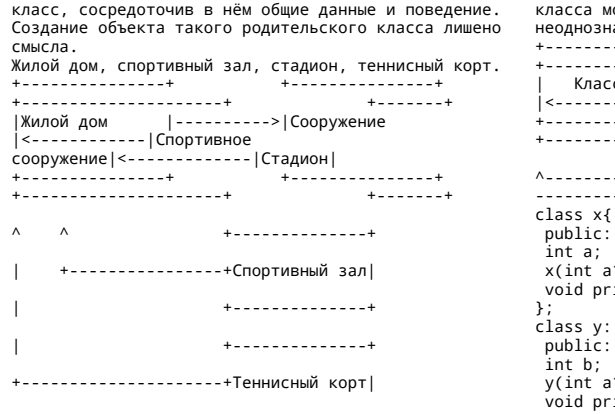
Пустые виртуальные функции используются в цепочке наследования в том случае, когда функция в некотором промежуточном классе иерархии не нужна, но должен сохраниться механизм виртуальных функций для следующих потомков.

```
class A{
 ..............
 public:
 ...................
 virtual void f1(..){..}
 ...................
};
class B: public A{
 ..................
 public:
 void f1(...){}
 ..............
};
class C:public B{
 ...................
 public:
 void f1(...){......}
 ..........
};
void main(){
 A *pA=new A(...);
 B *pB=new B(...);
 C *pC=new C(...);
 pa->f1();
 pB->f1();//нет действия
 pC->f1();
};
```

Абстрактные классы и чистые виртуальные функции

Класс, который может использоваться только в качестве базового для других классов, называется абстрактным классом. Абстрактный класс содержит одну либо несколько чистых виртуальных функций. Чистая виртуальная функция - это функция-член класса, тело которой определено следующим образом: =0 (чистый спецификатор или чистый инициализатор). Для чистой виртуальной функции не приводят действительное определение. Предполагается, что она будет переопределена в порождённых классах. К абстрактным классам применимы следующие правила: абстрактный класс не может использоваться в качестве фактического параметра функции или типа возвращаемого значения. Формальный класс нельзя использовать в явном преобразовании; нельзя определить представителя абстрактного класса, то есть локальную или глобальную переменную, элемент данных. Можно определить ссылку или указатель или ссылку на абстрактный класс. Если класс, производный от абстрактного, не определяет все чистые виртуальные функции абстрактного класса, то он также является абстрактным. Если у ряда классов имеется ряд общих аттрибутов, то для упрощения их описания можно ввести общий для них родительский класс, сосредоточив в нём общие данные и поведение. Создание объекта такого родительского класса лишено смысла.

Жилой дом, спортивный зал, стадион, теннисный корт.




```
class building{
 protected:
 char *addr;
 double area;
 char* owner;
 public:
 building(char*,double,char*);
 char* getAddr();
 double getArea();
 char* getOwner();
 void setOwner(char*);
 virtual void getInfo()=0;
 virtual double getPr()=0;
};
class house:public building{
 protected:
 int app;//квартира
 int lodger;//жильцы
 int appf;//своб. квар.
 double price;
 public:
 house(char*,double,char*,int,double);
 void setLod(int l, int a=0){
  lodger+=l; appf-=a;
 }
 void setFree(int l,int a=0{
  lodger-=l; appf+=a;
 void getInfo(){
  cout<<"\nжилой дом по адресу "<<addr<<" имеет
"<<app<<" квартир, из которых, заселено
"<<app-appf<<", всего жильцов: "<<lodger;
 }
 double getPr(){return price;}
 .............
};
class sportbuilding{
 protected:
 int spectet;
 double price;
 public:
 .................
 double getPr(){return price;}
 ..................
```

Множественное наследование

Порождённый класс может иметь любое число родительских классов, однако конкретный родительский класс не может быть указан более одного раза в списке наследования порождённого класса.

```
class x: public y,public z
```

Вместе с тем, родительский класс может косвенно появиться в порождённом классе более одного раза. Если порождённый класс два или более раз косвенно наследует родительский класс, то при работе с объектами этого класса могут возникнуть проблемы, связанные с неоднозначностью.


```
class x{
 public:
 int a;
 x(int a1){a=a1;}
 void print(){cout<<"\na="<<a;}
};
class y: public x{
 public:
 int b;
 y(int a1,int b1):x(a1){b=b1;}
 void print(){cout<<"\na="<<a<<" b="<<b;}
};
class z: public x{
 public:
 int c;
 z(int a1,int c1):x(a1){c=c1;}
 void print(){cout<<"\na="<<a<<" b="<<b<<" c="<<c;}
};
class v: public y,public z{
 public:
 int d;
 v(int a1,int a2,int b1,int c1,int
d1):y(a1,b1),z(a2,c1){d=d1;}
 void print(){cout<<"\na из y ="<<y::a<<" a из z
="<<z::a<<" b="<<b<<" c="<<c<<" d="<<d;
```

Виртуальные классы

Для решения проблемы неоднозначности при множественном наследовании можно ввести виртуальные классы.

```
class y: virtual public x{...........};
class z: virtual public x{.............};
class v: public y, public z{
 int d;
 v(int a1,int b1,int c1,int d1)........
 void print(){cout<<"\na="<<a<<" b="<<b
 ................
```

Порядок вызова конструктора при наследовании

Конструкторы родительских классов имеют приоритет по сравнению с конструкторами порождённых классов.

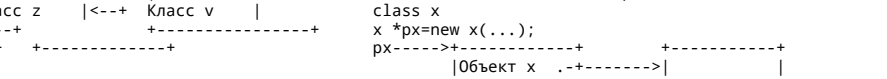
```
class x{
 protected:
 int a;
 public:
 x(int a1){a=a1;}
 void print(){cout<<"\na="<<a;}
};
class y:public x{
 protected:
 int b;
 public:
 y(int a1,int b1):b(b1*5) x(b){}
 void print(){
  cout<<"\na="<<a<<" b="<<b;
 }
};
void main(){
 x x1(2);
 x1.print();//a=2
 y y1(2,6);
 y1.print();//a=1010 b=30
}
```

Если иерархия порождённый классов содержит несколько виртуальных родительских классов, то их конструкторы будут вызываться в порядке их объявления. Затем вызываются конструкторы невиртуальных классов. Если виртуальный класс является порождённым от невиртуального, то сначала вызывается конструктор невиртуального класса.

Деструкторы. Динамические объекты

Деструктор - это специальный метод, который служит для освобождения памяти после работы с объектом.

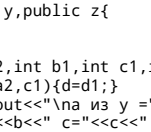
```
class x
x *px=new x(...);
```



Имя деструктора класса x имеет вид ~x.

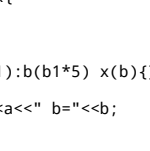
Деструктор не имеет параметров и для него не указывается тип возвращаемого значения. Деструктор вызывается тогда, когда соответствующая переменная удаляется из памяти. Для локальных переменных деструктор вызывается в конце соответствующего блока. Обычно деструктор вызывается посредством операции delete. Операция delete используется для унчтожения объектов, созданных с помощью операции new. Если используется какой-либо другой способ создания объекта, то должен применяться явный вызов соответствующего деструктора.

Пример: составить класс, реализующий стек для работы с символами.




```
class charStack{
 protected:
 int size;//размер стека
 char* s;//указатель на стек
 char* top;//указатель на вершину стека
 public:
 charStack(int size1){
  s=top=new char[size=size1];
 }
 ~charStack(){delete []s;}
 void push(char c){*top++=c;}
 char pop(){return *--top;}
};
void main(){
 charStack *pSt=new charStack(20);
 char mc[]="abcdef"; int i;
 for (i=0;i<4;i++)
  pSt->push(mc[i]);
 for (i=0;i<3;i++)
  cout<<"\n "<<pst->pop();
 delete pSt;
 charStack *pSt2=new charStack(15);
 ..................
}
```

Списки объектов классов




```
class x{
 protected:
 char* s;
 int a,n;
 public:
 x(){cout<<"\n введите целое:";
  cin<<a;
  cout<<"\n макс. дл. строки:";
  cin>>n;
  s=new char[n];
  cout<<"\n строка:";
```

```
  cin>>s;
 }
 ~x(){delete[] s;}
 int getA(){return a;}
 void print(){cout<<"\n............}
};
struct elem{
 x *px;
 elem *next;
};
class List{
 elem *plist;
 public:
 List(){plist=NULL;}
 ~List();
 void add();
 void print();
};
 v :
+-------+
+-------+
+---.---+<----et(2)
 (3)v
+-------+<----x(1)
|       |
+-------+
void list::add(){
 elem *et;
 x *px1;
 px1=new x();//1
 et=new elem;//2
 et->px=px1;//3
 et->next=plist;//4
 plist=et;//5
}
void x::print(){
 elem *et=pList;
 while (et!=NULL){
  et->px->print();
  et=et->next;
 }
}
List::~List(){
 elem *et=plist;
 while (et!=NULL){
  pList=pList->next;
  delete et->px;
  delete et;
 }
};
void main(){
 List *pl;
 pl->add();
 pl->add();
 pl->print();
 ...........
 delete pl;
 pl2->add();
 pl2->print();
 ..........
}
```

Массивы объектов класса

Для создания массива объектов класса необходимо предварительно создать массив указателей на объекты класса, а затем последовательно создавать объекты.

```
class x{
 int a;
 char *s;
 public:
 x(int a1,char *s1){
  a=a1;
  s=new char[strlen(s1)+1];
  strcpy(s,s1);
 }
 ~x(){
  delete[] s;
 void print(){
  cout<<"\na="<<a<<" s="<<s;
 }
};
void main(){
 int b,n,i;
 char s[1000];
 cout<<"\nКоличество элементов массива:";
 cin>>n;
 x **mx=new x*[n];
 for (i=0;i<n;i++){
  cout<<"\nЦелое число:";
  cin>>i;
  cout<<"\nстрока (до 1000 символов):";
  cin>>s;
  x[i]=new x(b,s);
 }
 for (i=0;i<n;i++)
  x[i]->print();
  .................
 for (i=0;i<n;i++)
  delete mx[i];
 delete mx;
 .....................
}
```

Статические члены класса

Данные и методы класса можно определить как статические. Статический член класса рассматривается как глобальная переменная или функция, доступная только в области класса.

Статические данные - члены класса

Данные - члены класса, определённые с модификатором static, являются общими для всех объектов класса, поскольку они имеют только один экземпляр, память выделяется до создания элементов класса. Объявление производится в пределах класса, а описание - за пределами класса.

Пример: составить программу, подсчитывающую количество существующих объектов некоторого класса x.

```
class x{
 int a;
 char *s;
 static int n;
 public:
 x(int a1,char *s1){
  a=a1;
  s=new char[strlen(s1)+1];
  strcpy(s,s1);
  n++;
 }
 ~x(){
  n--;
  delete[] s;
 }
 void print(){
  cout<<"\n объект:"<<a<<", "<<s<<" всего объектов:
"<<n;
 }
};
int x::n=0;
void main(){
 x x1(1, "объект1");
 x1.print();
 x *px=new x(2,"объект2");
 px->print();
 delete px;
 x1.print();
}
```

Статические функции - члены класса

Статическим функциям - членам класса при их вызове указатель this не передаётся, поэтому при необходимости работы с объектами класса им следует передать ссылку или указатель на объект класса.

Основное назначение статических функций - обработка статических данных. В этом случае статическая функция может быть вызвана тогда, когда не создан ещё ни один объект класса. Статические функции не могут быть виртуальными.

Пример: необходимо создать класс Работник с почасовой оплатой труда, при этом учесть подоходный налог.

```
Pn^          /
  |       k2|
  |        /
  |       |
  |   k1 /|
  |     / |
  +----+----------------->Zp
       z1 z2
class Worker{
 protected:
 char name[15];
 int time;
 double zp;
 static:
 double z1,z2;
 static double z1,z2;
 static double k1,k2;
 static double tariff;
 public:
 Worker(char *);
 void setTime(int);
 void getZp();
 static void setZ12(double, double);
 static void setK12(double, double);
 static void setTariff(double);
};
double Worker::z1=100;
double Worker::z2=1000;
double Worker::k1=0.1;
double Worker::k1=0.2;
double Worker::tariff=9.5;
Worker::Worker(char *n){
 strcpy(name,n);
 time=0;
 Zp=0;
}
void Worker::setTime(int t){
 time+=t;
}
void Worker::getZp(){
 double pn=0;
 zp=time*tariff;
 if (zp>z1&&zp<=z2) pn=(zp-z1)*k1;
 if (zp>z2) pn=(z2-z1)*(zp-z2)*k2;
 zp=pn;
 cout<<"\n"<<name<<" к выдаче:"<<zp;
 time=0;
}
void Worker::setZ12(double z11, double z21){
 z1=z11;
 z2=z21;
}
void Worker::setK12(double k11,k21){
 k1=k11;
 k2=k21;
}
void setTariff(double t){
 tariff=t;
}
void main(){
 Worker::setTariff(10,1);
 Worker W1("Петров");
 Worker W2("Сидоров");
 W1.setTime(100);
 W2.setTime(200);
 W1.getZp();
 W2.getZp();
 Worker::setK12(0.11,0.22);
 W1.setTime(175);
 W2.setTime(220);
 W1.getZp();
 .................
}
```

Дружественные функции

Дружественной функцией класса называется функция, которая, не являясь членом класса, имеет доступ к членам класса из области private и protected. При описании дружественной функции используется зарезервированное слово friend. Поскольку дружественная функция не является членом класса, то ей не передаётся указатель this на объект класса, поэтому для обращения к членам класса ей необходимо передать ссылку или указатель на объект класса.

Пример:

```
class x{
 int a;
 public:
 x(int a1){
  a=a1;
 }
 void print(){
  cout<<"\na="<<a;
 }
 friend void f1(x*,int);
};
void f1(x* x1,int b){
 x1->a=b;
}
void main(){
 x x1(2);
 x1.print();//a=2
 f1(&x1,3);
 x1.print();//a=3
```

Функцию - член одного класса можно объявить дружественной для другого класса.

```
class y;
class x{
 ................
 public:
 void fx(y*,....);
 ......................
};
class y{
 ..................
 public:
 friend void x::fx(y*,....);
 ......................
};
```

Можно объявить все функции одного класса дружественными для другого класса.

```
class y;
class x{
 friend y;
 protected:
 int a,b;
 public:
 x(int,int);
 void set(int,int);
 void print();
 ........................
};
class y{
 protected:
 int c,d;
 public:
 y(int,int);
 void f1(x*,int);
 void f2(x*,int);
 void f3(x*);
 .................
};
void main(){
 x x1(1,2);
 y y1(3,4);
 y1.f1(&x1,5);
 y1.f2(&x2,7,8);
 y1.f3();
 ...................
```

```
}
void y::f1(&x x1, int z){
 x1->a=z;
}
void y::f3(&x x1){
 x.a=c;
 x.b=d;
}
```
Пример: определить дружественную функцию для двух классов, описывающих числовые массивы разных типов. Функция находит наибольший элемент в двух объектах.
```
class MInt;
class MDouble{
 double *m;
 int n;
 public:
 MDouble(double[],int);
 ~MDouble();
 void print();
 friend void fmax(MInt&,MDouble&);
};
class MInt{
 int *m;
 int n;
 public:
 MInt(int[], int);
 ~MInt();
 void print();
 friend void fmax(MInt&,MDouble&);
}
MDouble::MDouble(double int[], int n1){
 m=new double [n=n1];
 for (int i=0;i<n;i++)
  m[i]=m1[i];
}
MDouble::~MDouble(){
 delete[] m;
}
void MDouble::print(){
 for(.....................;
}
MInt::MInt(int m1[],int){
 m=new int[n=n1];
 for (int i=0;i<n;i++)
  m[i]=m1[i];
}
MInt::~MInt(){
 delete[] m;
}
MInt::print(){
 .....................
}
void fmax(MInt& mi, MDouble& md){
 double max=mi.m[0];
 int i;
 for (i=1;i<mi.n;i++)
  if (mi.m[i]>max)
   max=mi.m[i];

void fmax(MInt &mi, MDouble &md){
 double max=mi.m[0];
 for (int i=0;i<min;i++)
  if(mi.m[i]>max) max=mi.m[i];
 for (i=0;i<md.n;i++)
  if(md.m[i]>max) max=md.m[i];
 cout<<"\nнаибольшее число в двух объектах:"<<max;
}
void main(){
 double d[]={1.1,1.2,3.4,5.8};
 int i[]={5,-6,7,9,8,-2};
 MInt i1(6,i);
 MDouble d1(4,d);
 fmax(i1,d1);
}
```
Дружественные функции в некотором смысле подобны оператору GOTO. Если оператор GOTO нарушает структурированность программы, то дружественные функции нарушают принцип использования классов. Всегда можно так реструктурировать классы, чтобы избавиться от необходимости использовать дружественные функции.

Переопределение операций

Каждый язык имеет набор операций, который не покрывает даже множество математических операций. Например, в языке C и C++ отсутствует операция возведения в степень. Перерпределение операций - это дополнительный сервис для программиста.

```
X=(A-B)*(C-D)+F^T
a^b^c
```
Проблема №1: какой же будет порядок операций?
```
a*b^c
a^-b
```
Для переопределения используются только существующие операции языка C++, при этом сохраняется их ассоциативность, порядок выполнения, "местность". Не допускается переопределение операций ,, :: и ?.

Префиксная или постфиксная унарная операция # может быть определена двумя способами: нестатической функцией - членом класса без аргументов.
```
ob.operator#();
```
нестатической функцией - членом класса с одним аргументом.
```
ob.operator#(ob);
```
Префиксная или постфиксная бинарная операция может быть переопределена двумя способами: нестатической функцией - членом класса с одним аргументом либо нестатической функцией - не членом класса с двумя аргументами.
```
ob1.operator#(ob2);
operator#(ob1,ob2);
```
Пример: переопределение операций для работы с комплексными числами с использованием дружественных функций.
```
complex{
 double re,im;
 public:
 complex(double r,double i=0);
 complex();
 void print();
 friend complex operator+(complex,complex);//сложение
//Перед тем, как назначить знаки для соответствующих операций, нужно изучить соответствующую предметную область и изучить порядок выполнения операций в этой предметной области.
 friend complex operator-(complex,complex);//вычитание
 friend complex operator*(complex,complex);//умножение
 friend complex operator/(complex.complex);//деление
 inline complex::complex(double r,double i=0){
  re=r; im=i;
 }
 inline complex::complex(){
  re=0; im=0;
 }
 inline void complex::print(){
  cout<<"\nre="<<re<<"\nim="<<im;
 }
 inline complex operator+(complex a,complex b){
  return complex(a.re+b.re,a.im+b.im);
 }
 inline complex operator-(complex a,complex b){
  return complex(a.re-b.re,a.im-b.im);
 }
 inline complex operator*(complex a,complex b){
  return
complex(a.re*b.re-a.im*b.im,a.re*b.im+a.im*b.re);
 }
 inline complex operator/(complex a,complex b){
  return .....................................;
 }
};
void main(){
 complex a(2.5,3.5), b(4.5), c(2), d(), e();
 d=a+b;
 d.print();
 e=b-c;
 e.print();
 d=b*c;
 d.print();
 e=a/b;
 e.print();
}
```
Переопределение операций для комплексных чисел с использованием функций - членов класса.
```
complex{
 double re,im;
 public:
 complex(double r,double i=0){re=r;im=i;}
 complex(){re=0;im=0;}
 void print(){.............};
 complex operator+(complex a){
  return complex(re+a.re,im+a.im);
 }
 complex operator-(complex a){
  return complex(re-a.re,im-a.im);
 }
 complex operator*(complex a){
  return complex(re*a.re-im*a.im,re*a.im+im*a.re);
 }
 complex operator/(complex a){
 .........................
 };
 complex operator+(double d){
  return complex(re+d,im);
 }
 complex operator-(double d){
  return complex(re-d,im);
 }
 complex operator*(double d){
  return complex(re*d,im*d);
 }
 complex operator/(double d){
  return complex(re/d,im/d);
 }
};
void main(){
 complex a(2.1,5.2), b(3,4), c(7.1), d(), e();
 d=(a-c)*8+(b+c)/2.2;
};
```
Переопределение операций присваивания и индексации

По умолчанию операция присваивания предусматривает поэлементное копирование. Если необходимо что-либо другое. то она должна быть переопределена. Операция присваивания имеет стандартный формат:
```
X&operator=(const X&)
```
Операция индексации определяется так: type&operator[](int) как бинарная операция неопределяемой функции статического класса с одним аргументом.

В качестве примера рассмотрим класс "Массив":
```
class Array{
 protected:
 int size;
 double *arr;
 public:
 Array(int,double[]);
 Array(const Array&);
 ~Array();
 void print();
 Array& operator=(const Array&);
 double& operator[](int);
 Array operator+(const Array&);
class Array{
 protected:
 int size:
 double *arr;
 public:
 .............
 virtual ~Array();
```
//Деструкторы рекомендуется делать виртуальными для того, чтобы гарантировать правильное освобождение памяти из-под динамического объекта, поскольку в любой момент времени будет выбран деструктор, соответствующий физическому типу объекта.
```
 .........................
};
Array::Array(int n, double d=0.0){
 arr=new double[size=n];
 for (int i=0;i<size;i++)
  arr[i]=d;
}
Array::Array(int n,double[] m){
 arr=new double[size=n];
 for (int i=0;i<size;i++)
  arr[i]=m[i];
}
Array::Array(const Array &a){
 arr=new double[size=a.size];
 for (int i=0;i<size;i++)
  arr[i]=a.arr[i];
}
Array::~Array(){
 delete[] arr;
}
void Array::print(){
 cout<<"\nмассив:";
 for (...............)
 ........................
}
Array & Array::operator=(const Array &a){
 if (this==&a) return *this;
 delete[] arr;
 arr=new double[size=a.size];
 for (int i=0;i<size;i++) arr[i]=a.arr[i];
 return *this;
}
double & Array::operator[](int i){
 if (i>size-1) return arr[size-1];
 if (i<0) return arr[0];
 return arr[i];
}
Array & Array::operator+(const Array &a){
 Array *b; int i;
 if (size>=a.size){
  b=new Array(*this);
  for (i=0;i<a.size;i++)
   b.arr[i]=a.arr[i];
 } else {
  b=new Array(a);
  for (i=0;i<size;i++)
   b.arr[i]=a.arr[i];
 }
 return *b;
}
void main(){
 double a[]={1.1,2.4,9,5,7.5};
 double b[]={-0.5,-4,2.7};
 double c[]={0};
 Array x(5,a),y(3,b),z(1,c),Q(4,0.5);
 Array p=x;
 p.print();
 z=x+y+q;
 cout<<"\nТретий элемент массива в x="<<x[2];
 x[2]=-10;
 cout<<"\nновое значение - "<<x[2];
 ...........
}

complex{
 protected:
 double re,im;
 public:
 complex &operator+=(complex a){
  re+=a.re;
  im+=a.im;
  return *this;
 }
```

```
 complex &operator-=(complex a){
  re-=a.re;
  im-=a.im;
  return *this;
  complex &operator *=(complex a){
   re=re*a.re-im*a.im;
   im=re*
+-----------
|           |
  -----------+
```
Перегрузка операций new и delete
Операции new и delete переопределяются для создания собственных механизмов управления памятью. Операция new должна возвращать результат типа void* и иметь первый аргумент типа size_t.
Операция delete должна возвращать тип void и иметь первый аргумент типа void*, второй аргумент часто имеет тип size_t.
```
class x{
 ..............
 public:
 ..............
 void* operator new(size_t size){
  ..............
  return newAlloc(size,.....);
 }
.........
 void operator delete(void *p,size_t size,...){
  .................
  return
+-------------
|               |
 ----------------+
```
Если переопределены операции new и delete, то именно их вызывают конструктор и деструктор при создании и удалении объекта. Функции-операции new и delete всегда являются статическими членами класса, независимо от отстутствия или наличия слова static. Эти функции не могут быть виртуальными. Если в некотором классе x переопределены операции new и delete, то стандартные new и delete используются для всех типов, кроме x. Операции new и delete наследуются.
Переопределение операций чтения-ааписи
Операции >> и << уже переопределены. Мы введём своё определение этих операций для своих типов данных.
```
#include <iostream.h>
struct Info{
 char *name;
 double value;
 int units;
};
ostream & operator<<(ostream &s, Info &m){
 s<<m.name<<""<<m.value<<""<<m.units;
 return s;
}
istream &operator>>(istream &s, Info &m){
 s>>m.name>>m.value>>m.units;
 return s;
}
void main(){
 Info m;
 m.name=new char[20];
 cout<<"\nВведите name,value,units";
 cin>>m;
 cout<<"\nНаш объект:";
 cout<<m;
}
```

Преобразование типа объектов класса
Преобразование типа объектов класса называется пользовательским преобразованием. Такие преобразования часто используются в дополнение к стандартным преобразованиям, например, float to double, int to float и т. д. Например, функция, ожидающая параметр типа t1, может быть вызвана с параметром типа t2, если существует преобразование типа t2 в t1.

Пользовательские преобразования применяются только там, где они однозначны. Реализуются они при помощи конструктора и преобразующих функций.
Преобразование посредством конструктора
```
class x{
 int i;
 double d;
 char *s;
 public:
 x(int k){
  i=k;
  s=new char[20];
  strcpy(s,"определено только i");
  d=0;
 }
 x(double d1){
  d=d1;
  i=0;
  s=new char[20];
  strcpy(s,"определено только d");
 }
 x(char *s1){
  s=new char[20];
  strcpy(s,s1);
  i=1; d=1;
 }
 void print(){
  cout<<"\ns="<<s<<" i="<<i<<" d="<<d;
 }
};
void main(){
 x a=3;
 x b="язык C++";
 x c=1.85;
 a.print();
b.print();
 c.print();
 ............
}
```
Преобразующие функции
Функция - член класса x с именем ->(operator)->|имя приведенного типа|-> определяет преобразование из x в приведенный тип. Преобразующие функции могут выполнять два преобразования, которые недоступны конструктору:
1) преобразование из класса в базовый тип;
2) преобразование объекта одного класса в объект другого класса без модификации объявления последнего.
Правила использования преобразующих функций:
1) преобразующая функция не имеет аргументов;
2) преобразующая функция не имеет явной спецификации типа возвращаемого значения, подразумевается тип, который записан после слова operator;
3) преобразующая функция может быть виртуальной;
4) преобразующая функция наследуется.
```
class x{
 protected:
 double d;
 int i;
 public:
 x(double d1,int i1){
  d=d1;
  i=i1;
 }
 void print(){
  cout<<"\nd="<<d<<" i="<<i;
 }
 operator int(){
  return i;
 }
};
void main(){
 x x1(5.1,1),x2(4.2,2),x3(3.7,3),x4(2.3,4);
 int n1=int(x1);//n1=1
 int n2=x2;//n2=2
 int n3=(x1)?x2+x3:x4;
 if (x2-x3) n2=x4;
  else n2=x1;
 ..............
}
class y:public x{
 protected:
 char s[15];
 public:
 y(int, double, char*);
 ......................
};
void main(){
 y y1(7,2.8,"тестирование преобразующих функций");
 int n=y1;//y1.x::operator(int)
}
```
К одному значению может применяться не более одного неявного пользовательского преобразования (конструктором либо преобразующей функцией).
```
class x{
 ........................
 public:
 operator int(){...........};
 ..................
};
class y:public x{
 ..................
 public:
 operator x(){
  ..............
 }
 ....................
};
void main(){
 y a(........);
 int n=a;//ошибка a.operator x().operator int()
 int m=x(a);
}
```
Пример: преобразование из одного класса в другой
```
class x{
 protected:
 int i;
 double d;
 char s[20];
 public:
 x(int i1,double d1,char* s1){
  int i1;
  double d1;
  strcpy(s,s1);
 }
 void print(){
  cout<<.....................;
 }
};
class y{
 int n,m,k;
 public:
 y(int n1, int m1, int k1){
  n=n1; m=m1; k=k1;
 }
 void print(){
  ........................
 }
 operator x(){
  char sy[20];
  itoa(k,sy,19);
  x x1(n,m,sy);
  return x1;
 }
};
void main(){
 y y1(35,5,274);
 y1.print();
 x x1=y1;
 x1.print();
 ................
}
```
Константные объекты и константные функции - члены класса
В языке C++ можно создать объект класса с модификатором const. В этом случае содержимое объекта не должно меняться после его инициализации. Чтобы предотвратить изменение значений данных константного объекта, компилятор генерирует сообщение об ошибке, если объект используется не с константной функцией - членом класса.
Правила оформления константных функций:
1) функция объявляется со словом const, которое следует за списком параметров;
2) функция не может изменять значения данных - членов класса;
3) функция не может вызывать неконстантные функции - члены класса;
4) функция может быть вызвана как для константных, так и для неконстантных объектов класса.
```
class Promises{
 protected:
 char name[6];//наименование
 double s;//площадь
 int km;//количество мест
 int kr;//количество занятых мест
 char time;//вид помещения: a - аудитория, l - лаборатория, p - прочее
 public:
 Promises(char *n, double s1, int km1, char t){
  strcpy(name,n);
  s=s1; k=km1;
  type=t;
  kr=0;
 }
 void print() const{
  cout<<"помещение "<<name<.........;
 }
 void setKr(int m){
  Kr=m;
 }
 char getType() const{
  return type;
 }
 int getKm() const{
  return km;
 }
 char* getName() const{
  char s[6];
  strcpy(s,name);
  return s;
 }
 void setType(char t){
  type=t;
 }
 void getKr(){
  return kr;
 }
}
void main(){
 Promises const p1("307ф",80,75,'a');
 Promises const p2("307ф",80,75,'a');
 p1.print();
 p2.print();
 p1.setKr(40);
 p1.print();
 cout<<"\nколичество рабочих мест"<<p1.getKm();
 cout<<"\nколичество рабочих мест"<<p2.getKm();
}
```
Шаблоны
В языке C++ используются шаблоны функций и шаблоны классов.
Шаблоны функций
```
max(x,y);
int max(int x,int y){
 return (x>y)?x:y;
}
double max(double x, double y){
 return (x>y)?x:y;
```

```
}
#define max(x,y) ((x>y)?x:y)
```
->(template)->(<)->|список параметров шаблона|->(>)->|тип возвращаемого значения|->|имя функции|->(()->|список формальных параметров|->())->({)->|тело функции|->(})->

Каждый элемент списка параметров шаблона представляет собой запись вида

```
 ->(class)->|идентификатор|->
template <class T> T max(T x,T y){
 return (x<y)?x:y;
}
```

При использовании шаблона функции компилятор генерирует функцию в соответствии с типом данных, используемым непосредственно при вызове функции.

```
void main(){
 int n,m,k;
 double a,b,c;
 long ln,lm,lk;
 cout<<"\nВведите ........................";
 cin>>n>>m>>a>>b>>c>>ln>>lm;
 k=max(n,m);
 cout<<"\nk="<<k;
 c=max(a,b);
 cout<<"\nc="<<c;
 lk=max(ln,lm);
 cout<<"\nlk="<<lk;
}
template <class T> void sort(T arr[], int n){
 int i,f;
 T b;
 do{
  f=0;
  for (i=0;i<n-1;i++){
   if (arr[i]>arr[i+1]){
    b=arr[i];
    arr[i]=arr[i+1];
    arr[i+1]=b;
    f=1;
   }
  }
 } while(f);
}
```

Шаблоны классов

Шаблон класса даёт обобщённое определение семейства классов, использующее произвольные типы или константы. Шаблон определяет элементы-данные и элементы - функции класса.

->(template)->(<)->|список форм.