МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ УКРАИНЫ
ГВУЗ «Донецкий национальный технический университет»

Конспект лекций по курсу «Объектно-ориентированное программирование»

(для студентов направления подготовки 6.050101 «Компьютерные науки»)

Рассмотрено на заседании кафедры
Прикладной математики и информатики
Протокол № 1 от «29» августа 2013г.

Утверждено на заседании
Учебно-издательского совета ДонНТУ
Протокол № ___ от «___»_____2013г.

Донецк 2013

УДК 681.3.07

Конспект лекций по курсу «Объектно-ориентированное программирование» (для студентов направления подготовки 6.050101 «Компьютерные науки»). Сост.: И. А. Коломойцева. – Донецк, 2013. – 76 с.

автор: Коломойцева Ирина Александровна, кафедра Прикладной математики и информатики, ДонНТУ

Объектно-ориентированное программирование. Основные понятия.

Объектно-ориентированное программирование – это подход к построению программ, где главной отправной точкой служит объект с его свойствами и поведением. Основные принципы ООП были разработаны в языках Simula-67 (1967 – год создания) и Smalltalk, но в то время не получили широкого применения из-за трудностей освоения и низкой эффективности реализации.

ООП оперирует с таким понятием как класс.

Класс – это тип данных, определяемый пользователем. Класс представляет собой совокупность данных, характеризующих объект этого класса, и операций, которые могут быть с этими данными выполнены. Данные называют переменными-членами, полями или свойствами класса, а операции над данными – функциями-членами или методами класса. Существенным свойством класса является то, что детали его реализации скрыты от пользователей класса интерфейсом. Интерфейсом класса являются заголовки его методов.

Объектом называется экземпляр класса, т. е. переменная, в качестве типа которой указано имя класса. Объекты взаимодействуют между собой, посылая и получая сообщения.

Сообщение – это запрос на выполнение действия, содержащий набор необходимых параметров. Механизм сообщений реализуется с помощью вызова соответствующих функций. Таким образом, с помощью ООП можно легко реализовать событийно-управляемую модель, когда данные активны и управляют вызовом того или иного фрагмента программного кода. Примером событийно-управляемой модели может служить любая программа, управляемая с помощью меню. *После запуска такая программа пассивно ожидает действия пользователя и должна уметь правильно отреагировать на любое из них. Следует заметить, что событийно-управляемая модель не является частью ООП и может быть реализована без использования объектов. Например, программирование на языке C под Windows с использованием функций API. Противоположностью событийной модели является директивная, когда код управляет данными: программа после старта предлагает пользователю выполнить некоторые действия в соответствии с жестко заданным алгоритмом.*

Объектно-ориентированное программирование обладает следующими **свойствами**:

- инкапсуляция;
- наследование;

автор: Коломойцева Ирина Александровна, кафедра Прикладной математики и информатики, ДонНТУ

- полиморфизм.

Инкапсуляция (encapsulation) – это соединение в одной структуре данных (классе) данных и операций над данными в сочетании со скрытием ненужной для использования этих данных информации. Инкапсуляция повышает степень уровня абстракции программы, *то есть данные класса и реализация его функций лежат ниже уровня абстракции и для написания программы информация о них не требуется*. Инкапсуляция также позволяет изменить реализацию класса без модификации основной части программы, если интерфейс остался прежним. Инкапсуляция позволяет использовать класс в другом окружении и быть уверенным, что он не испортит не принадлежащие ему области памяти. А также создавать библиотеки классов для применения во многих программах.

Наследование означает такое соотношение между классами, когда один класс использует структурную или функциональную часть (свойства) одного или нескольких других классов (соответственно, мы имеем простое или множественное наследование). Свойства повторно не описываются, что сокращает объем программы. Класс, использующий данные или методы другого класса, называется производным или подклассом, или субклассом. Класс, который предоставляет свои данные и методы другому классу, называется базовым или надклассом, или суперклассом.

Иерархия классов представляется в виде древовидной структуры, в которой более общие классы располагаются ближе к корню, а более специализированные – на ветвях и листьях. В C++ каждый класс может иметь сколько угодно потомков и предков.

Полиморфизм – это свойство ООП, при котором одно и тоже имя может вызывать различные действия на этапе выполнения. Самый простой пример полиморфизма – перегрузка функций, когда из нескольких вариантов выбирается наиболее подходящая функция по соответствию ее прототипа передаваемым параметрам. Второй пример – использование шаблонов функций, когда один и тот же код видоизменяется в соответствии с типом, переданным в качестве параметра. Но чаще всего понятие полиморфизма связывается с механизмом виртуальных методов.

Описание класса

Описание класса приблизительно выглядит так:

```
class <имя> {
    [private:]
        <описание скрытых элементов>
    protected:
        <описание защищенных элементов>
    public:
        <описание доступных элементов>
}; // описание заканчивается точкой с запятой обязательно!!!
```

Управление доступом к элементам класса

автор: Коломойцева Ирина Александровна, кафедра Прикладной математики и информатики, ДонНТУ

Главная задача класса – скрыть как можно больше информации. Существует три вида пользователей класса:

– сам класс;
– обычные пользователи;
– производные классы.

Каждый пользователь обладает разными уровнями доступа. Этих уровней три, и они описываются тремя словами:

- private;
- public;
- protected.

Любое объявление, появляющееся до ключевого слова управления доступом, считается приватным по умолчанию.

Private – наиболее ограниченный доступ. Только сам класс (или классы, объявленные как дружественные (friend)) имеет доступ к приватным членам.

Public – общедоступный уровень доступа. Свойства и методы с таким уровнем доступа могут использоваться любым пользователем.

Protected – защищенный уровень доступа. Переменные и методы такого уровня доступа могут использоваться самим классом и классами, порожденными от него.

Свойства полей класса

Поля класса:

– могут иметь любой тип, кроме типа этого же класса, но могут быть указателями или ссылками на этот класс;

– могут быть описаны с модификатором const, при этом они инициализируются только один раз (с помощью конструктора) и не могут изменяться;

– могут быть описаны с модификатором static, но не с auto, extern и register.

Инициализация полей при описании не допускается, так как память под класс не выделяется, пока не будет создан экземпляр класса

Виды классов

Классы бывают:

– глобальные (объявленные вне любого блока);

– локальные объявленные внутри блока, например, функции или другого класса).

Свойства локальных классов

– внутри локального класса можно использовать типы, статические (static) и внешние (extern) переменные, внешние функции и элементы перечислений из области, в которой он описан; запрещается использовать автоматические переменные из этой области;

– локальный класс не может иметь статических элементов;

– методы этого класса могут быть описаны только внутри класса;

автор: Коломойцева Ирина Александровна, кафедра Прикладной математики и информатики, ДонНТУ

– если один класс вложен в другой класс, они не имеют каких-либо особых прав доступа к элементам друг друга и могут обращаться к ним только по общим правилам.

Пример определения класса

```
class point
{
        double x,y;
public:
        point(double xVal=0.0, double yVal=0.0) {x=xVal, y=yVal;}
        void PutPoint (double xVal, double yVal);
        double GetX() {retun x;}
        double GetY(){retun y;}
};
```

Приведен классический способ организации методов доступа к членам класса: все переменные являются закрытыми (private), а все методы - открытыми (public).

Все методы имеют непосредственный доступ к скрытым полям класса, то есть тела функций класса входят в область видимости private элементов класса.

В приведенном примере содержится три определения методов и одно объявление (PutPoint). Если тело метода определено внутри класса, он является встроенным (inline). *Как правило, встроенными делают короткие методы.* Если внутри класса записано только объявление (заголовок) метода, сам метод должен быть определен в другом месте программы. Выполняется это при помощи операции доступа к области видимости (::):

```
void point::PutPoint (double xVal, double yVal)
{x=xVal, y=yVal;}
```

Такой код реализации называется внешним.

Чтобы дать указание компилятору рассматривать функцию, реализованную вне класса, как встроенную, необходимо использовать ключевое слово inline. Например:

```
inline void point::PutPoint (double xVal, double yVal)
{x=xVal, y=yVal;}
```

Описание объектов

Создание объекта может происходить по одному из трех сценариев:

1) создание объектов с инициализацией по умолчанию;
2) создание объектов со специальной инициализацией;
3) создание объектов путем копирования других объектов.

```
point p; //объект класса point с параметрами по умолчанию
point p1(10,10); //объект с явной инициализацией
point p2[200]; //массив объектов с параметрами по умолчанию
point *p3 = new point(10); //динамический объект (второй параметр
                           // задается по умолчанию)
point &p4=p1;
```

автор: Коломойцева Ирина Александровна, кафедра Прикладной математики и информатики, ДонНТУ

При создании каждого объекта выделяется память, достаточная для хранения всех его полей, и автоматически вызывается конструктор, выполняющий их инициализацию. При выходе объекта из области видимости он уничтожается, при этом автоматически вызывается деструктор.

Доступ к элементам объекта аналогичен доступу к полям структуры. Для этого используется операция . (точка) при обращении к элементу через имя объекта и операция -> при обращении через указатель. Например:

```
p.PutPoint(40,60);
cout<<p2[0].GetX()<<endl;
cout<<p3->GetY()<<endl;
```

Указатель this

Каждый объект содержит свой экземпляр полей класса. Методы класса находятся в памяти в единственном экземпляре и используются всеми объектами совместно. Поэтому возникает необходимость обеспечить работу методов с полями именно того объекта, для которого они были вызваны. Это обеспечивается передачей в функцию скрытого параметра this, в котором хранится константный указатель на вызвавший функцию объект. Указатель this неявно используется внутри метода для ссылок на элементы объекта. В явном виде этот указатель применяется в основном для возвращения из метода указателя ***(return this;)*** или ссылки ***(return *this;)*** на вызвавший объект.

Пример. Определим метод, который из двух точек возвращает ссылку на ту, у которой больше координата x. Первая точка вызывает метод, вторая передается в качестве параметра.

```
point &the_greater(point &p)
{
    if (x>p.GetX()) return *this;
    return p;
}
…
point p(20,20);
point p1(10,10);
point &Greater=p.the_greater(p1);
// новый объект инициализируется полями объекта p.
```

Указатель this можно также применять для идентификации поля класса в том случае, когда его имя совпадает с именем формального параметра метода. Другой способ идентификации поля использует операции доступа к области видимости.

автор: Коломойцева Ирина Александровна, кафедра Прикладной математики и информатики, ДонНТУ

```
void MovePoint(double x, double y)
{
        this->x+=x;
        point::y+=y;
}
```

Перегрузка функций

Перегрузка – это придание функции более чем одного значения, то есть имя у функции остается прежним, а параметры и действия могут быть разными.

Конструкторы

Конструктор – это метод, который вызывается при создании объекта и производит инициализацию переменных членов. Это метод имеет такое же имя, как и класс.

Свойства конструкторов:

– конструктор не возвращает значение, даже типа void; нельзя получить указатель на конструктор;

– класс может иметь несколько конструкторов с разными параметрами для разных видов инициализации (при этом используется механизм перегрузки);

– конструктор, вызываемый без параметров, называется конструктором по умолчанию;

– параметры конструктора могут иметь любой тип, кроме этого же класса; можно задать параметры по умолчанию, но их может содержать только один из конструкторов;

– если программист не указал ни одного конструктора, то компилятор создаст его автоматически;

– конструкторы не наследуются;

– конструкторы нельзя описывать с модификаторами const, virtual и static;

– конструкторы глобальных объектов вызываются до вызова функции main; локальные объекты создаются, как только становится активной область их действия; конструктор запускается и при создании временного объекта (например, при передаче объекта из функции);

– конструктор вызывается, если в программе встретилась какая-либо из синтаксических конструкций:

имя_класса имя_объекта [(список параметров)];

//список параметров не должен быть пустым

имя_класса (список параметров);

//создается объект без имени (список параметров может быть пустым)

имя_класса имя_объекта = выражение;

//создается объект без имени и копируется

Например,

```
point p1(10,10), p2(20), p3;
point p4 = point(100);
point p5 = 200;
```

В первом операторе создаются три объекта. Значения не указанных параметров устанавливаются по умолчанию.

Во втором операторе создается безымянный объект со значением параметра x=100 (значение второго параметра устанавливается по умолчанию). Выделяется память под объект p4, в которую копируется безымянный объект.

Во третьем операторе создается безымянный объект со значением параметра x=200 (значение второго параметра устанавливается по умолчанию). Выделяется память под объект p5, в которую копируется безымянный объект. Такая форма создания объекта возможна в том случае, если для инициализации объекта допускается задать один параметр.

Пример. Использование перегруженных конструкторов.

```
enum color {red, green, blue};
class point
{
        double x,y;
        color cp;
        char *name;
    public:
        point(double xVal=0.0, double yVal=0.0);
        point(color cl);
        point(char *np);
…
};
point::point(double xVal, double yVal)
{x=xVal; y=yVal;}
point::point(color cl)
{
    switch (cl)
    {
        case red:   x=100;y=100;cp=red;name=0;break;
        case green: x=200;y=200;cp=green;name=0;break;
        case blue:  x=300;y=300;cp=blue;name=0;break;
    }
}
point::point(char *np)
{
    name=new char[strlen(np)+1];
    strcpy(name,np);
    x=100; y=100; cp=red;
}
```

автор: Коломойцева Ирина Александровна, кафедра Прикладной математики и информатики, ДонНТУ

```
…
point p1(10,10),p2(10),p3;
point p4(green);
point p5("X1");
```

Есть еще один способ инициализации полей в конструкторе – с помощью списка инициализаторов, расположенных после двоеточия между заголовком и телом конструктора.

Деструкторы

Деструкторы - это функции, которые используются для выполнения определенных операций при удалении объекта. Обычно деструкторы выполняют операции, обратные тем, которые выполняли конструкторы. Например, если конструктор выделяет динамическую память для членов класса, то деструктор ее освобождает.

Деструктор вызывается автоматически, 4

– для локальных объектов – при выходе из блока, в котором они объявлены;

– для глобальных – как часть процедуры выхода из main;

– для объектов, заданных через указатели, деструктор вызывается неявно при использовании операции delete (автоматический вызов деструктора объекта при выходе из области действия указателя на него не производится).

Между конструктором и деструктором существует ряд различий.

1. Деструкторы могут быть виртуальными, а конструкторы – нет.
2. Деструкторам нельзя передавать аргументы.
3. В каждом классе может быть объявлен только один деструктор.

Имя деструктора состоит из имени класса, перед которым стоит ~ (тильда).

```
class point
{…
public:
        point () {x=0; y=0;}
        ~point() {};
        …
};
```

Статические элементы класса

Статические поля и методы объявляются с помощью модификатора static. Их можно рассматривать как глобальные переменные или функции, доступные только в пределах области класса.

Статические поля

Статические поля применяются для хранения данных, общих для всех объектов класса, например, количества объектов или ссылки на разделяемый всеми объектами ресурс. Эти поля существуют для всех объектов класса в единственном экземпляре, то есть не дублируются.

автор: Коломойцева Ирина Александровна, кафедра Прикладной математики и информатики, ДонНТУ

Свойства статических полей:

1) память под статическое поле выделяется один раз при его инициализации независимо от числа созданных объектов (и даже при их отсутствии) и инициализируется с помощью операции доступа к области действия, а не операции выбора (определение должно быть записано вне функций):

```
#include <iostream.h>
class Example
{public:
        static int value; //объявление в классе
};
int Example::value; //определение статического поля в глобальной области, по умолчанию инициализируется нулем.
// int Example::value=10; //пример инициализации произвольным значением
```

2) статические поля доступны как через имя класса, так и через имя объекта

```
Example object1, *object2;
…
cout<<Example::value<<object1.value<< object2->value;
```

3) на статические поля распространяется действие спецификаторов доступа, поэтому статические поля, описанные как private, нельзя изменить с помощью операции доступа к области действия; это можно сделать только с помощью статических методов;

4) память, занимаемая статическим полем, не учитывается при определении размера с помощью операции sizeof.

Статические методы

Статическая функция класса может обращаться только к статическим полям класса и вызывать только другие статические методы классы, так как им не передается скрытый указатель this. В то время как нестатические функции класса должны вызываться только через объекты класса, то статические функции такого ограничения не имеют. Статические функции вызывается или через имя класса, или через имя объекта.

```
#include <iostream.h>
class Simple
{public:
        static int sum(int v1, int v2) {return v2+v1;}
};
void main()
{cout<<Simple::sum(10,20)<<"\n"; //первый способ вызова функции
 Simple s1;
 cout<<s1.sum(10,20); //второй способ вызова функции
}
```

автор: Коломойцева Ирина Александровна, кафедра Прикладной математики и информатики, ДонНТУ

Функции и объекты, объявляемые с декларацией const

Константный объект – это объект, значения полей которого изменять запрещено. К нему должны применяться только константные методы.

Константный метод:

- объявляется с ключевым словом const после списка параметров;
- не может изменять значения полей;
- может вызывать только константные методы;
- может вызываться для любых (не только константных объектов).

Рекомендуется описывать как константные только те методы, которые предназначены для получения значений полей.

```
class point
{
    double x,y;
public:
    …
    double GetX() const {return x;}
    double GetY() const {return y;}
    …
};
…
point p10(10,10); //неконстантный объект
cout<<p10.GetX()<<" "<<p10.GetY()<<endl;
const point p10(10,10); //константный объект
cout<<p10.GetX()<<" "<<p10.GetY()<<endl;
```

Имитационное моделирование с применением ООП

Рассмотрим процесс моделирования работы микроволновой печи.

Описание системы:

1) имеется единственная кнопка управления, которая доступна для пользователя печи. Если дверь печи закрыта, и пользователь нажимает кнопку, то печь будет готовить пищу в течение 1 минуты;

2) если пользователь нажимает на кнопку во время работы печи, получаем дополнительную минуту работы;

3) если дверь открыта, нажатие кнопки не имеет эффекта ;

4) внутри печи есть электролампа, во время работы она должна быть включена; когда дверь печи открыта, электролампа должна быть включена;

5) можно приостановить процесс приготовления пищи открытием двери, в этом случае время сбрасывается;

6) начальная конфигурация: дверь закрыта, лампа погашена;

7) если время истекло, выключается питание и электролампа.

Функцию управления берет на себя пользователь, то есть управление осуществляется с клавиатуры.

Идентификаторы состояния:

1) включена/выключена электролампа;
2) включено/выключено электропитание;
3) открыта/закрыта дверь;
4) сколько времени осталось до конца приготовления пищи.

Электролампа	Дверь	Питание
0	0	Невозможный набор
0	1	0
1	0	0
1	1	1

Необходимые функции (сообщения):
1) нажать кнопку;
2) изменить положение двери;
3) пища готова.

//stove.h

```
#include <stdio.h>
#include <conio.h>
enum lamp {on, off};
enum door {open, close};
enum power {yes, no};
class stove
{       lamp curr_lamp;
        door curr_door;
        power curr_power;
        int time;
        void take_door(void);
        void take_button(void);
        void ready(void);
public:
        stove();
        void Run();};
stove::stove()
{       curr_lamp=off;
        curr_door=close;
        curr_power=no;
        time=-1;
}
void stove::take_door(void)
{       if (curr_door==open)
        {       curr_door=close;
                curr_lamp=off;
                printf("Дверь закрыта \n");
```

автор: Коломойцева Ирина Александровна, кафедра Прикладной математики и информатики, ДонНТУ

```
        }
        else
        {       curr_door=open;
                curr_lamp=on;
                time=-1;
                printf("Дверь открыта \n");
                if (curr_power==yes)
                        printf("Процесс прерван \n");
                curr_power=no;
        }
        return;
}
void stove::take_button(void)
{       if (curr_door==open)
                printf("No effect\n");
        else
        {       if (curr_power==yes)
                {       time+=60;
                        printf("Вам добавлена минута \n");
                }
                else
                {       curr_power=yes;
                        curr_lamp=on;
                        time=60;
                        printf("Начало приготовления пищи \n");
                }
        }
        return;
}
void stove::ready(void)
{       curr_power=no;
        curr_lamp=off;
        time=-1;
        printf("\a");
        printf("Пища готова \n");
        return;
}
void stove::Run(void)
{       int i,flag=0;
        while (flag!=1)
        {       while (!kbhit() && time>0)
                        time--;
                if (time==0) ready();
                i=getch();
```

```
                switch (i)
                {       case 98: take_button(); break;
                        case 100: take_door(); break;
                        case 27: flag=1; break;
                }
                fflush(stdin);
        }
        return;}
```

//stove.cpp

```
#include "stove.h"
void main()
{       stove MyStove;
        MyStove.Run();
}
```

Дружественные классы

Дружественность – возможность использования метода двумя и более объектами различных классов, связанных отношениями общности.

Дружественные классы необходимы в том случае, если не связанным отношением родства классам необходим доступ к закрытым или защищенным секциям одного из них.

```
#include <iostream.h>
```

//не правильный вариант

```
class A
{double x;
public:
        A() {x=3.14;}
};
class System: public A
{public:
        void f() {cout<<x;} //доступ к переменной x закрыт
};
```

//правильный вариант

```
class A
{       friend class System;
        double x;
public:
        A() {x=3.14;}
};
class System
{public:
        A obj;
        void f() {cout<<obj.x<<endl;}
};
```

Свойства друзей:

автор: Коломойцева Ирина Александровна, кафедра Прикладной математики и информатики, ДонНТУ

1) отношение дружественности не наследуются, то есть, если А дружественен В, а С порожден от А, то это не означает, что С становится автоматически дружественным В;

2) свойство дружественности не транзитивно, то есть, если класс А дружественен классу В, а класс В – классу С, то А не становится автоматически дружественным классу С;

3) свойство дружественности не коммутативно, то есть, если А дружественен В, то это не означает, что В дружественен А.

Но при этом А можно объявить дружественным В.

Взаимодружественные классы

```
class A; //неполное объявление класса
class B
{       friend class A;
public:
        void f(A* c1) {};};
class A;
{       friend class B;};
```

Раздел, в котором помещено объявление friend, не имеет значения. Обычно для наглядности объявление friend для класса помещается в первой секции. Когда дружественной объявляется функция, объявление обычно помещается там, где была бы записана эквивалентная функция-член. Единственное ограничение, налагаемое на объявление friend, заключается в том, что это объявление должно находиться внутри объявления класса.

Дружественные функции

Дружественные функции применяются для доступа к скрытым полям класса и представляют собой альтернативу методам.

Пример использования дружественных функций: переопределенные операции вывода объектов.

Правила описания и особенности дружественных функций:

1) дружественная функция объявляется внутри класса, к элементам которого ей нужен доступ, с ключевым словом friend; в качестве параметра ей должен передаваться объект или ссылка на объект класса;

2) дружественная функция может быть обычной функцией или методом другого ранее определенного класса; на нее не распространяется действие спецификаторов доступа, место размещения ее объявления в классе безразлично;

3) одна функция может быть дружественной сразу нескольким классам.

```
class One;
class   Two
{       char *s2;
public:
        Two(){s2="1, 2, 3";}
        void Show(One &c1);};
class One
```

```
{       friend void Two::Show(One &c1);
        char *s1;
public:
        One() {s1="Testing...";}};
void Two::Show(One &c1)
{       cout<<c1.s1<<s2<<endl;}

void main()
{       One c1;
        Two c2;
        c2.Show(c1);
        return;}
```

Использование дружественных функций не рекомендуется, поскольку они нарушают принцип инкапсуляции.

Перегрузка операций

Перегрузка операций позволяет определить действия для объектов в выражениях.

<имя_класса> operator op(<имя_класса > a, < имя_класса > b) – общий вид перегруженной операции op.

В C++ существует 40 операций, которые можно перегрузить.

Нельзя перегрузить следующие операции: «.», «.*»,«?:», «::», «sizeof», «#», «##».

При создании описания класса существует две возможности определения операций:

1) определение операции как функции класса;
2) определение операции как дружественной функции.

Если бинарная перегруженная операция реализована как функция класса, то ее первым операндом является объект, которому принадлежит сообщение, следовательно, такой функции передается только второй член операции, следовательно, у нее 1 параметр.

Если унарная перегруженная операция реализована как функция класса, то у нее нет параметров.

Если перегруженная операция реализована как дружественная функция, то ей передается 1 или 2 параметра.

C++ «не понимает» семантики перегруженного оператора. C++ не может выводить сложные операторы из простых. Нельзя изменять синтаксис перегруженных операций. Нельзя изобретать новые операторы, а можно использовать только 40.

Пример (работа с комплексными числами):

Вариант 1.

//f1.h

```
class Complex
{       float real, imag;
public:
```

```
        Complex(float aReal, float aImag): real(aReal), imag(aImag) {};
        Complex() {real=imag=0.0;}
        Complex(float aReal): real(aReal), imag(0.0) {};
        float GetReal() {return real;}
        float GetImag() {return imag;}
        friend Complex operator+(Complex &c1, Complex &c2);
        friend Complex operator-(Complex &c1, Complex &c2);
};

Complex operator+(Complex &c1, Complex &c2)
{       return Complex(c1.real+c2.real, c1.imag+c2.imag);}

Complex operator-(Complex &c1, Complex &c2)
{       return Complex(c1.real-c2.real, c1.imag-c2.imag);}
//f1.cpp
#include <iostream.h>
#include "f1.h"
void main()
{
        Complex a(1.0,2.0), b(3.0,4.0);
        a=a+b;
        cout<<a.GetReal()<<endl<<a.GetImag()<<endl;
}
```

Вариант 2.

```
//f1.h
class Complex
{       float real, imag;
public:
        Complex(float aReal, float aImag): real(aReal), imag(aImag) {};
        Complex() {real=imag=0.0;}
        Complex(float aReal): real(aReal), imag(0.0) {};
        float GetReal() {return real;}
        float GetImag() {return imag;}
        void operator+(Complex &c1);
        void operator-(Complex &c1);
};

void Complex::operator +(Complex &c1)
{       real+=c1.real;
        imag+=c1.imag;
}
void Complex::operator -(Complex &c1)
{       real-=c1.real;
        imag-=c1.imag;
```

```
}
//f1.cpp
#include <iostream.h>
#include "f1.h"
void main()
{
        Complex a(1.0,2.0), b(3.0,4.0);
        a+b;
        cout<<a.GetReal()<<endl<<a.GetImag()<<endl;
}
```

Нельзя смешивать два типа объявлений.

Контейнерные классы

Контейнерные классы – это классы, которые содержат в своем описании один или несколько объектов или указатели на объекты. В этом случае имеет место отношение «содержит».

```
//f1.h
#include <iostream.h>
class Tail
{int length;
public:
        Tail(int n) {length=n;}
        int GetTail() {return length;}
};
class Dog
{Tail tail;
public:
        Dog(int n):tail(n) {};
        void DisplayPar() {cout<<tail.GetTail()<<endl; return;}
};
//f1.cpp
#include "f1.h"
void main()
{ Dog d(20);
  d.DisplayPar();
}
```

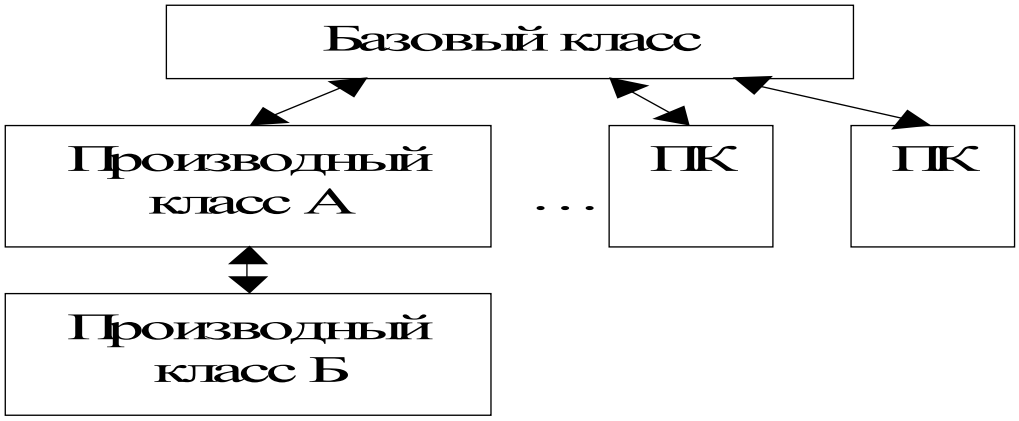
Сначала инициализируются все поля–объекты, которые содержатся в описании класса, причем в том порядке, в котором они объявлены. Деструкторы вызываются в порядке, обратном инициализации.

Иерархия классов

Простое наследование

Простое наследование – это такое наследование, при котором порождаемые классы наследуют методы и свойства одного базового класса.

автор: Коломойцева Ирина Александровна, кафедра Прикладной математики и информатики, ДонНТУ



Производный класс А является базовым для класса Б.

Производные классы могут наследовать любые данные и функции базового класса, кроме конструктора и деструктора.

Не существует ограничений на количество производных классов.

```
#include <iostream.h>
class TBase
{private:
        int count;
public:
        TBase() {count=0;}
        int GetCount() {return count;}
        void SetCount(int n) {count =n;}
};
class TDerived:public TBase
{public:
        TDerived():TBase() {};
        void incr(int n) {SetCount(GetCount()+n);} //в этой функции нельзя написать строку count+=n; так как такая строка вызовет ошибку компиляции, потому что у производного класса нет прав доступа к переменным и функциям базового класса с уровнем доступа private
};

void main()
{TDerived d;
 d.SetCount(15);
 d.incr(10);
 cout<<d.GetCount(); //на экране будет значение 25;
}
```

Спецификаторы доступа базовых классов

Когда класс Second порождается от First со спецификатором прав доступа private, например:

```
class First {…};
```

class Second:First {…};

или

class First {…};

class Second: private First {…};

то все наследуемые (т. е. защищенные и общедоступные) имена базового класса становятся приватными в производном классе.

Когда класс Second порождается от First со спецификатором прав доступа protected, например:

class First {…};

class Second: protected First {…};

то все наследуемые (т. е. защищенные и общедоступные) имена базового класса становятся защищенными в производном классе.

Когда класс Second порождается от First со спецификатором прав доступа public, например:

class First {…};

class Second: public First {…};

то все общедоступные имена базового класса будут общедоступными, а все защищенные будут защищенными в производном классе.

Порядок вызова конструкторов

При создании экземпляра класса вызывается его конструктор. Если класс является производным, то должен быть вызван конструктор базового класса. Порядок вызова в C++ фиксирован. Если базовый класс, в свою очередь, является производным, то процесс рекурсивно повторяется до тех пор, пока не будет достигнут корневой класс.

Например,

class First {…};

class Second: public First {…};

class Third: public Second {…};

При создании экземпляра класса Third конструкторы вызываются в следующем порядке:

First::First()

Second::Second()

Third::Third()

Порядок вызова конструкторов

Деструкторы для производных классов вызываются в порядке обратном вызову конструкторов. Таким образом, порядок вызовов деструкторов, сгенерированных для разрушения экземпляра класса Third, будет следующим:

Third::~Third()

Second::~Second()

First::~First()

автор: Коломойцева Ирина Александровна, кафедра Прикладной математики и информатики, ДонНТУ

Разрешение области видимости

Порождая один класс от другого, можно прийти к такой ситуации, когда в нескольких классах используются переменные и функции с одинаковыми именами.

Например:

```
class A
{public:
        int fun() {return 1;}
};
class B : public A
{public:
        int fun() {return 2;}
};
void main()
{       A a;
        B b;
        int i = a.fun(); //i=1
        int j=b.fun(); //j=2
}
```

В этом случае компилятор действует по следующему алгоритму: если имя в базовом классе переобъявляется в производном, то имя в производном классе подавляет соответствующее имя в базовом.

В C++ можно заставить компилятор «видеть» за пределами текущей области видимости. Для этого используются оператор разрешения видимости. Общая форма этого оператора такова:

<имя класса>::<идентификатор из класса>,

где <имя класса> - это имя базового или производного класса, а <идентификатор из класса> - это имя любой переменной или функции, объявленной в классе.

Модифицируем наш класс B следующим образом:

```
class B : public A
{public:
        int fun() {return 2;}
        int fun1() {return A::fun();}
};
```

Теперь вызов функции B.fin1() приведет к вызову функции fun() класса A.

Динамическое связывание в ООП (полиморфизм)

Полиморфизм – это свойство ООП, при котором одно и тоже сообщение может вызывать различные действия на этапе выполнения.

Полиморфизм – это характеристика функций-членов, а не объектов. Несмотря на то, что полиморфизм реализуется через архитектуру класса, полиморфными могут быть только функции-члены класса, а не весь класс. В C++ полиморфная функция привязывается к одной из возможных функций только тогда, когда ей передается конкретный объект, т. е. в C++ вызов

функции в исходном коде только *обозначается*, без точного указания на то, какая именно функция вызывается. Этот процесс называется **позднее связывание**. Процесс, при котором компилятор (как в традиционных языках), базируясь на исходном коде, вызывает фиксированные идентификаторы функций, а компоновщик заменяет эти идентификаторы физическими адресами, называется **раним связыванием.** То есть идентификаторы функций ассоциируются с физическими адресами до этапа выполнения, еще на стадии компиляции и компоновки. При раннем связывании программы выполняются быстрее, но существенно ограничены возможности разработчика. При позднем связывании остро встает вопрос об эффективности исполняемой программы.

Способность объектно-ориентированных языков автоматически определять тип объекта на этапе выполнения программы называется RTTI (run-time type identification – идентификация во время выполнения).

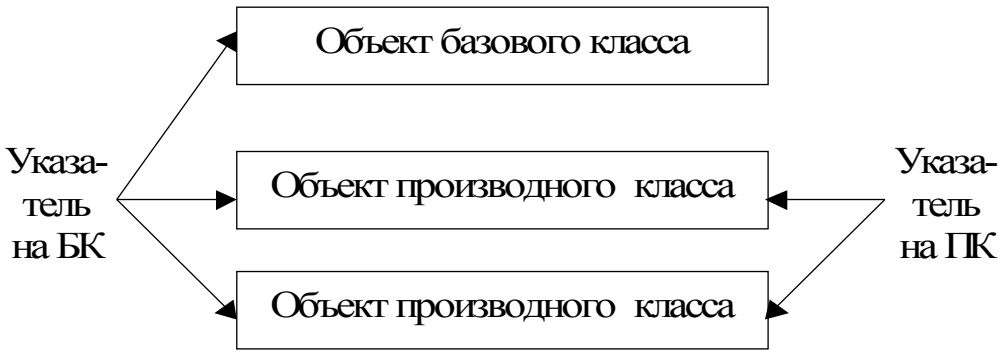
Виртуальные функции

В C++ позднее связывание для функции определяется при ее объявлении с помощью ключевого слова virtual. Позднее связывание имеет смысл только для объектов, являющихся частью иерархии классов. Объявление функции виртуальной для класса (не используемого в качестве базового) синтаксически корректно, но приведет только к потере времени в момент выполнения.

Виртуальные функции – функции, вызов которых зависит от типа объектов. С помощью виртуальных функций объект определяет свои действия.

Правило: указатель на базовый класс может ссылаться на объект этого класса или любого другого, производного от базового.

```
A* aobject;        A
                   ↑
B* bobject;        B
                   ↑
C* cobject;        C
```



cobject=&aobject // так нельзя делать
aobject=&cobject.

Пример:

автор: Коломойцева Ирина Александровна, кафедра Прикладной математики и информатики, ДонНТУ

```
//f1.h
#include <iostream.h>
class Animal
{public:
/*virtual*/  char* speak() {return "";}
};
class Dog: public Animal
{public:
        char * speak() {return "Gav!!!";}
};
//f1.cpp
#include "f1.h"
void sound(Animal& i)
{cout<<i.speak()<<endl;}
void main()
{       Dog Sharic;
        sound(Sharic); //"" (на экран будет выведена пустая строка)
}
```

Решение проблемы – позднее связывание. Функцию speak() класса Animal достаточно объявить виртуальной, после чего компилятор запустит механизмы позднего связывания.

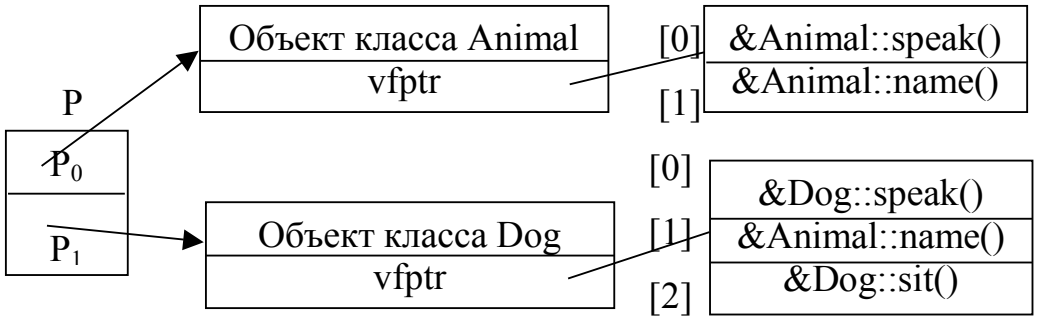
```
//f1.h
#include <iostream.h>
#include <string.h>
class Animal
{
protected:
    char *pname;
public:
    Animal(char *AnName)
    {
       pname=new char[strlen(AnName)+1];
       strcpy(pname, AnName);
    }
    virtual  char* speak() {return "";}
    virtual char *name() {return pname;}
};

class Dog: public Animal
{
public:
    Dog(char *name):Animal(name) {}
    char *speak()
    {
```

```
        char *phrase;
        phrase=strdup(pname); //дублирует строку, при этом вызывая функцию malloc()
        return strcat(phrase," Say Gav! ");
    }
    virtual char *sit()
    {
        char *phrase; phrase=strdup(pname); return strcat(phrase," sits");
    }
};
//f1.cpp
#include "f1.h"
void main()
{
    Animal* p[2]={new Animal("a"), new Dog("Sharic")};
    cout<<p[0]->speak()<<endl; //выведет на экран пустую строку
    cout<<p[1]->speak()<<endl; //выведет на экран строку “Sharic Say Gav!”
    //cout<<p[1]->sit()<<endl; //ошибка компиляции: 'sit' : is not a member of 'Animal'
    cout<<((Dog*)p[1])->sit()<<endl;
}
```

Механизм работы позднего связывания

Компилятор для каждого класса (не объекта), содержащего хотя бы один виртуальный метод, создает таблицу виртуальных функций VFTABLE. В эту таблицу помещаются адреса виртуальных функций класса в порядке их описания. Адрес любого виртуального метода имеет в VFTABLE одно и то же смещение для каждого класса в пределах иерархии. В каждом классе также помещается указатель VFPTR на таблицу виртуальный функций.

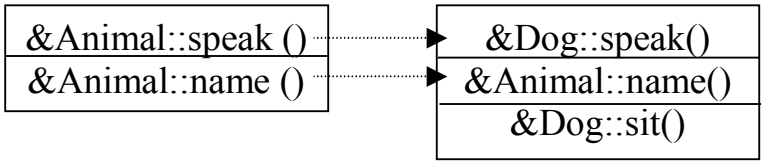


При явном связывании мы получаем абсолютные адреса функций. Теперь, вызывая функцию name(), мы говорим, что надо вызвать функцию по адресу VFPTR+1.

Наследование и таблица виртуальных функций

В случае со строкой p[1]->sit(), компилятор выдает ошибку.

автор: Коломойцева Ирина Александровна, кафедра Прикладной математики и информатики, ДонНТУ



Компилятор запрещает вызовы тех виртуальных функций, которые присутствуют только в производном классе классе, а так как у базового класса Animal нет функции sit(), поэтому компилятор выдает ошибку при попытке ее использования.

Но чтобы избежать ошибки, необходимо обращаться к функции sit() следующим образом:

((Dog*) p[1])->sit();

Будем считать, что классы Animal и Dog объявлены как в предыдущем примере.

```
//f1.cpp
#include "f1.h"
void main()
{       Dog druzhok("Druzhok");
        Animal* p1=&druzhok;
        Animal& p2=druzhok;
        Animal p3("Druzhok");
        cout<<p1->speak()<<endl;    //”Druzhok Say Gav!” (Позднее связывание)
        cout<<p2.speak()<<endl;     //”Druzhok Say Gav!” (Позднее связывание)
        cout<<p3.speak()<<endl; //” “ (Явное связывание)
}
```

Абстрактные базовые классы

Класс становится абстрактным, когда в него добавляется чистая виртуальная функция:

virtual void f() = 0;

Нельзя создавать объекты абстрактного базового класса. Когда абстрактный класс наследуется, все чистые виртуальные функции должны быть переопределены, иначе производный класс становится абстрактным. Такие функции позволяют вставлять заглушки.

```
//f1.cpp
#include <iostream.h>
#include <string.h>
class Animal
{public:
        virtual         char* speak() =0;
        virtual char *eat() =0;
};
```

автор: Коломойцева Ирина Александровна, кафедра Прикладной математики и информатики, ДонНТУ

```
char* Animal::speak()
{return "Animal::speak()";}
char* Animal::eat()
{return "Animal::eat()";}
class Dog: public Animal
{public:
        char *speak() {return Animal::speak();}
        char *eat() {return Animal::eat();}
};
//f1.cpp
#include "f1.h"
void main()
{       Dog Sharik;
        cout<<Sharik.speak()<<endl;
        cout<<Sharik.eat()<<endl;

}
```

Определение чистых виртуальных функций в базовом классе не дает права создавать объект абстрактного базового класса. Такой способ удобен, когда существует общий фрагмент кода, который нет необходимости дублировать в каждой функции. Вызов виртуальной функции требует на две ассемблерные инструкции больше, чем вызов обычной функции.

Виртуальные деструкторы

В отличие от конструкторов деструкторы могут быть виртуальными.

```
//f1.h
include <iostream.h>
class Base1
{public:
        ~Base1(){cout<<"~Base1()"<<endl;}
};
class Derived1: public Base1
{public:
        ~Derived1(){cout<<"~Direved1()"<<endl;}
};
class Base2
{public:
        virtual ~Base2(){cout<<"~Base2()"<<endl;}
};
class Derived2: public Base2
{public:
        ~Derived2(){cout<<"~Direved2()";}
};
//f1.cpp
#include "f1.h"
```

```
void main()
{
        Base1* bp1=new Derived1;
        delete bp1; //~Base1()
        Base2* bp2=new Derived2;
        delete bp2; //~Derived2()~Base2()
}
```

Деструкторы вызываются в порядке обратном вызову конструкторов.

Чистые виртуальные деструкторы

В отличие от чистых виртуальных функций для чистых виртуальных деструкторов необходимо обеспечить тело функции. При реализации деструкторов производных классов их тело задавать не нужно.

```
//f1.h
#include <iostream.h>
class AbstractBase
{public:
        virtual ~AbstractBase()=0;
};
AbstractBase::~AbstractBase()
{cout<<"Destructor Abstract Class"<<endl;}
class Derived: public AbstractBase
{
};
//f1.cpp
#include "f1.h"
void main()
{       Derived d;
}
```

Виртуальные функции в деструкторах

Механизм позднего связывания не действителен для виртуальных и невиртуальных деструкторов. Вызывается локальная версия виртуальной функции.

```
//f1.h
#include <iostream.h>
class Base
{public:
        virtual ~Base() {cout<<"~Base()"<<endl; f();}
        virtual void f() {cout<<"Base::f"<<endl;}
};
class Derived: public Base
{public:
        ~Derived() {cout<<"~Derived()"<<endl;}
        virtual void f() {cout<<"Derived::f"<<endl;}
};
```

автор: Коломойцева Ирина Александровна, кафедра Прикладной математики и информатики, ДонНТУ

```
//f1.cpp
#include "f1.h"
void main()
{       Base *bp = new Derived;
        delete bp; //"~Derived()","~Base()","Base::f"
}
```

Перегруженные функции при динамическом связывании

Общее правило перегрузки функций: перегруженная (переопределенная) функция базового класса в производных классах прячет все другие версии этой функции в базовом классе.

```
//f1.h
#include <iostream.h>
class Base
{public:
        virtual int f() {cout<<"Base::f()"<<endl;return 1;}
        virtual void f(char *s) {cout<<s<<endl;}
        virtual void g() {};
};
class Derived1: public Base
{public:
        void g() {cout<<"Derived1::g()"<<endl;}
};
class Derived2: public Base
{public:
        int f() {cout<<"Derived2::f()"<<endl;return 2;}
};
/*class Derived3: public Base
{public:
        void f() {cout<<"Derived3::f()"<<endl;} //Ошибка, так как компилятор не позволяет изменять тип возвращаемого значения для перегруженной виртуальной функции
};*/
//f1.cpp
#include "f1.h"
void main()
{       char *s;
        s="Hello!";
        Derived1 d1;
        int x=d1.f(); //Base::f()
        d1.f(s); //"Helo!"
        Derived2 d2;
        x=d2.f(); //Direved2:f()
        //d2.f(s); //ошибка! функция f не имеет ни одного параметра
        Base* b2=&d2;
```

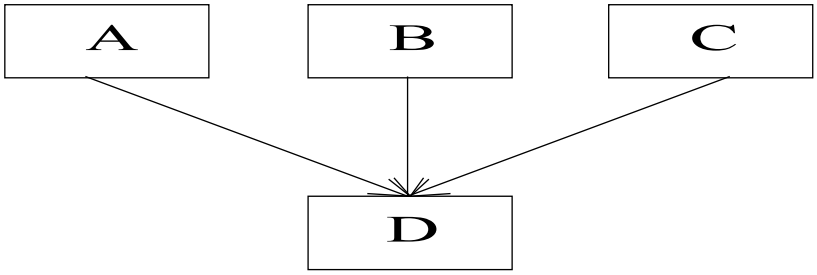
автор: Коломойцева Ирина Александровна, кафедра Прикладной математики и информатики, ДонНТУ

```
        b2->f();//Direved2:f()
        b2->f(s);//Hello!
}
```

Множественное наследование

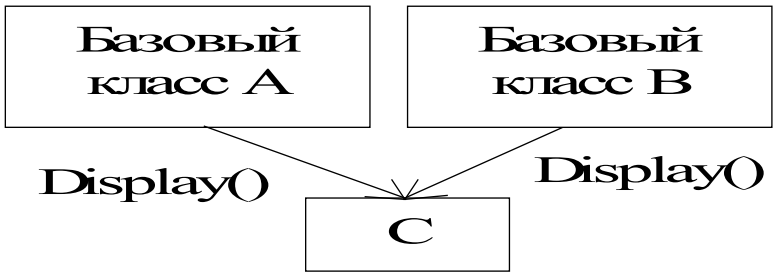
Множественное наследование описывает родство между классами, при котором один класс может иметь несколько базовых.

```
class A {…};
class B {…};
class C {…};
class D: public A, public B, public C {…};
```



Вызов конструкторов происходит в том порядке, в котором были объявлены базовые классы:

```
A::A()
B::B()
C::C()
D::D()
```




```
//f1.h
class A
{public:
        void Display(void) {…}
};
class B
{public:
        void Display(void) {…}
};

class C: public A, public B
```

автор: Коломойцева Ирина Александровна, кафедра Прикладной математики и информатики, ДонНТУ

```
{public:
        void f(void);
};
```

Если имеем реализацию:

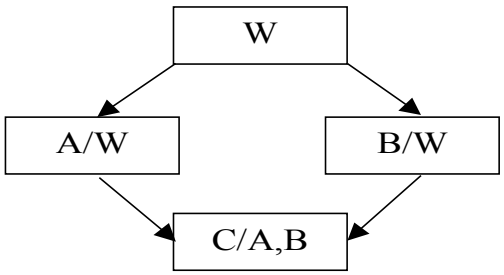
```
void C::f(void)
{Display();}
```

то компилятор не будет знать, какую функцию Display() вызывать. Нужно указывать явно.

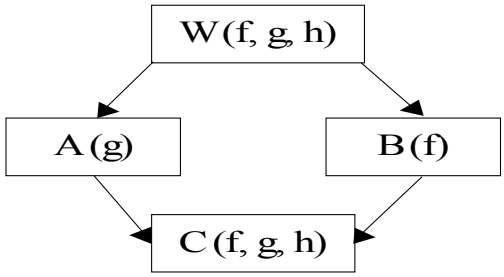
```
void C::f(void)
{       A::Display();
        B::Display();
}
```

Конструктор класса, у которого несколько родителей, перед вызовом своего конструктора вызывает все конструкторы базовых классов в порядке их объявления. Их можно вызвать явно:

```
class C: public A, public B
{public:
        C():A(), B() {};
        void f(void);
};
```



Если через объект класса C попытаться обратиться к методам класса W, то компилятор выдаст ошибку. Выходом из этой проблемы будет использование виртуального базового класса




```
//f1.h
#include <iostream.h>
class W
{public:
        virtual void f() {cout<<"W::f()"<<endl;}
        virtual void g() {cout<<"W::g()"<<endl;}
```

автор: Коломойцева Ирина Александровна, кафедра Прикладной математики и информатики, ДонНТУ

```
        virtual void h() {cout<<"W::h()"<<endl;}
};
class A:public virtual W
{public:
        void g() {cout<<"A::g()"<<endl;}
};
class B:public virtual W
{public:
        void f() {cout<<"B::F()"<<endl;}
};
class C: public A, public B
{public:
        void f() {cout<<"C::f()"<<endl;}
};
//f1.cpp
#include "f1.h"
void main()
{       C* pc = new C;
        pc->f(); //C::f()
        pc->g(); //A::g()
        pc->h(); //W::h()
```

((A*)pc)->f(); //C::f(). pc – указатель на C, насильственно преобразуем его к указателю на A; в описании класса A функции f нет, она описана в базовом классе W, но как виртуальная; далее происходит анализ класса, на который указывает pc, то есть класса C, в нем обнаруживается реализация функции f(), вызов которой и происходит. Если слово virtual перед f() в W убрать, тогда будет вызвана f из W.

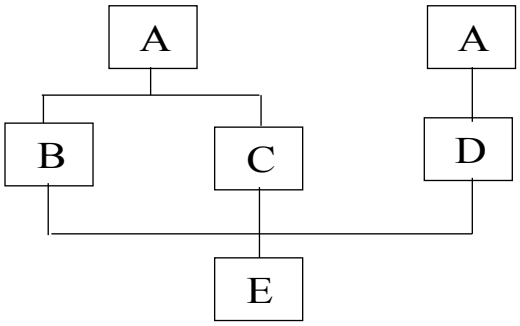
```
        ((W*)pc)->f(); //C::f()
        B* pb = new B;
        pb->f(); //B::f()
        pb->g(); //W::g();
        pb->h(); //W::h()
        ((W*) pb)->f(); //B:f()
        A* pa = new A;
        pa->f(); //W::f()
        pa->g(); //A::g()
        pa->h(); //W::h()
        ((W*) pa)->g(); //A::g()
}
```

Для описания иерархий множественного наследования используется прямой ациклический граф. Виртуальные базовые классы инициализируются перед любыми невиртуальными базовыми классами в том порядке, в котором они появляются в прямом ациклическом графе наследования при просмотре

его снизу вверх и слева направо. Для приведенного выше примера порядок вызова конструкторов следующий: W(), A(), B(), C().

```
class A {};
class B: public virtual A {};
class C: public virtual A {};
class D: public A {};
class E: public B, public C, public D {};
```

Дерево наследования приведенной выше иерархии изображено на рисунке ниже.



Порядок вызова конструкторов для класса E таков: A(), B(), C(), D(), E().

автор: Коломойцева Ирина Александровна, кафедра Прикладной математики и информатики, ДонНТУ

ПОТОКИ

В языке C++ нет средств для ввода-вывода. Эти средства можно просто создать на самом языке.

В языке C++ производится ввод-вывод потоков (streams) байтов. Поток представляет собой последовательность байтов. В операциях ввода байты пересылаются от устройства (например, от клавиатуры, дисковода или соединения сети) в оперативную память. При выводе байты пересылаются из оперативной памяти на устройства.

Чтение данных из потока называется извлечением, вывод в поток – помещением, или включением.

Основная задача потоковых средств ввода-вывода - это процесс преобразования объектов определенного типа в последовательность символов в битовом их представлении и наоборот. Указанная схема ввода-вывода является основной. Многие схемы двоичного ввода-вывода можно свести к ней.

Обмен с потоком для увеличения скорости передачи данных производится через специальную область оперативной памяти – через буфер. Фактическая передача данных выполняется при выводе после заполнения буфера, а при вводе – если буфер исчерпан.

По направлению обмена потоки можно разделить на входные (данные вводятся в память), выходные (данные выводятся из памяти) и двунаправленные (допускающие как извлечение, так и включение).

По виду устройств, с которыми работает поток, можно разделить потоки на стандартные, файловые и строковые.

Стандартные потоки предназначены для передачи данных от клавиатуры и на экран дисплея, файловые потоки – для обмена информацией с файлами на внешних носителях данных, строковые потоки – для работы с массивами символов в оперативной памяти.

Язык C++ предоставляет возможности для ввода-вывода как на «низком», так и на «высоком» уровне. При вводе-выводе на низком уровне (то есть неформатированный ввод-вывод) каждый байт является самостоятельным элементом данных. Передача на низком уровне позволяет осуществлять пересылку больших по объему потоков ввода-вывода с высокой скоростью. Недостаток программирования низкоуровневого потокового ввода-вывода состоит в его трудоемкости.

При высокоуровневом (или форматированном) потоковом вводе-выводе байты группируются в значащие элементы данных, например, целые числа, символы, строки, вещественные числа, а также данные типов, определенных пользователем. Такой способ потокового ввода-вывода неэффективен только для файлов очень большого объема.

ЗАГОЛОВОЧНЫЕ ФАЙЛЫ БИБЛИОТЕКИ IOSTREAM

автор: Коломойцева Ирина Александровна, кафедра Прикладной математики и информатики, ДонНТУ

Заголовочный файл <ios.h> содержит описание базового класса потоков ввода-вывода.

Заголовочный файл <iosfwd.h> содержит предварительные объявления средств ввода-вывода.

Библиотека <iostream.h> потокового ввода-вывода реализует строгий типовой и эффективный способ символьного ввода и вывода целых, вещественных чисел и символьных строк. Также библиотека является базой для расширения, рассчитанного на работу с пользовательскими типами данных.

Заголовочный файл <iostream.h> определяет интерфейс потоковой библиотеки. В ранних версиях потоковой библиотеки использовался файл <stream.h>. <iostream.h> определяет полный набор средств. <stream.h> определяет подмножество, которое совместимо с ранними потоковыми библиотеками.

Заголовочный файл <iostream.h> включает объекты cin, cout, cerr и clog. cin соответствует стандартному потоку ввода, cout – стандартному потоку вывода, cerr – небуферизованному стандартному потоку вывода сообщений об ошибках, clog – буферизованному стандартному потоку вывода сообщений об ошибках. Для этих объектов предусмотрены возможности для форматированного и для неформатированного ввода-вывода.

Заголовочный файл <istream.h> содержит описание шаблона потока ввода.

Заголовочный файл <ostream.h> содержит описание шаблона потока вывода.

Заголовочный файл <iomanip.h> содержит информацию, необходимую для обработки форматированного ввода-вывода при помощи параметризованных манипуляторов потока.

Заголовочный файл <fstream.h> содержит информацию для проведения операций с файлами.

Заголовочный файл <sstream.h> содержит описание потоков ввода-вывода в строки.

Заголовочный файл <streamига.h> содержит описание функций буферизации ввода-вывода.

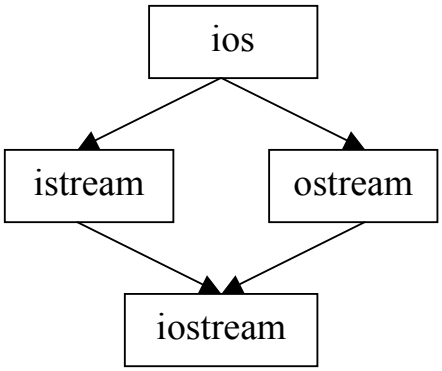
Также есть заголовочные файлы <amstream.h>, <austream.h>, <ddstream.h> и <mmstream.h>, который содержат информацию, необходимую для управления специализированными устройствами, предназначенными, например, для ввода-вывода аудио- и видеоданных.

автор: Коломойцева Ирина Александровна, кафедра Прикладной математики и информатики, ДонНТУ

Классы и объекты потоков ввода-вывода

Библиотека iostream содержит много классов для обработки операций ввода-вывода. Например, класс istream поддерживает операции по вводу потоков, класс ostream поддерживает операции по выводу объектов, класс iostream поддерживает оба типа операций: и ввод, и вывод потоков.

Класс istream и класс ostream являются прямыми потомками класса ios. Класс iostream является производным классом множественного наследования классов istream и ostream. Иерархию наследования можно представить так:

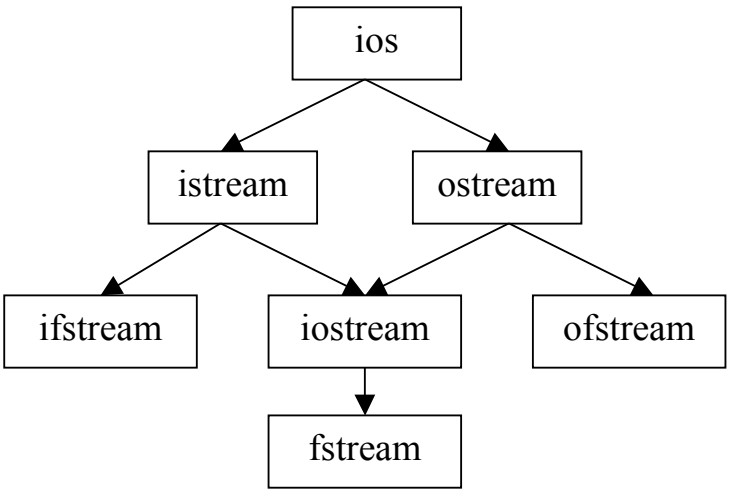


Операции «поместить в поток» и «взять из потока» – это перегруженные операции сдвига влево (<<) и вправо (>>). Эти операции применяются к объектам стандартных потоков cin, cout, cerr и clog.

При обработке файлов C++ используются следующие классы:

– класс ifstream, который выполняет операции ввода из файлов;
– класс ofstream, который выполняет операции вывода в файлы;
– класс fstream, который выполняет операции ввода-вывода файлов.

Класс ifstream наследует класс istream, класс ofstream наследует класс ostream, а класс fstream – класс iostream. Иерархия классов потоков ввода-вывода с ключевыми классами обработки файлов выглядит так:



Пример использования объекта cerr:

автор: Коломойцева Ирина Александровна, кафедра Прикладной математики и информатики, ДонНТУ

```
cerr << "x = " << x << endl;
```

Здесь cerr обозначает стандартный поток ошибок. Так, если x типа int со значением 123, то приведенный оператор выдаст

```
x = 123
```

Вывод встроенных типов

Для управления выводом встроенных типов определяется класс ostream с операцией << (вывести):

```
class ostream : public virtual ios
{
        // ...
    public:
        ostream& operator<<(const char*);    //строки
        ostream& operator<<(char);
        ostream& operator<<(short i)
        { return *this << int(i); }
        ostream& operator<<(int);
        ostream& operator<<(long);
        ostream& operator<<(double);
        ostream& operator<<(const void*);    // указатели
        // ...
};
```

Функция operator<< возвращает ссылку на класс ostream, из которого она вызывалась, чтобы к ней можно было применить еще раз operator<<. Так, если x типа int, то

```
cerr << "x = " << x;
```

понимается как

```
(cerr.operator<<("x = ")).operator<<(x);
```

Это означает, что если несколько объектов выводятся с помощью одного оператора вывода, то они будут выдаваться в естественном порядке: слева - направо.

Функция ostream::operator<<(const void*) напечатает значение указателя в такой записи, которая более подходит для используемой системы адресации. Программа

```
main()
{
    int i = 0;
    int* p = new int(1);
    cout << "local " << &i<< ", free store " << p << '\n';
}
```

выдаст, например,

```
local 0x0066FDF4, free store 0x00790DA0
```

автор: Коломойцева Ирина Александровна, кафедра Прикладной математики и информатики, ДонНТУ

Ввод встроенных типов

Класс istream определяется следующим образом:

```
class istream : public virtual ios
{
        //...
    public:
        istream& operator>>(char*);     // строка
        istream& operator>>(char&);     // символ
        istream& operator>>(short&);
        istream& operator>>(int&);
        istream& operator>>(long&);
        istream& operator>>(float&);
        istream& operator>>(double&);
         //...
};
```

Функции ввода operator>> определяются так:

```
istream& istream::operator>>(T& tvar)
 {
     // пропускаем обобщенные пробелы
     // каким-то образом читаем T в`tvar'
     return *this;
 }
```

Для контроля соответствия количества введенных символов и объема зарезервированной памяти можно использовать функцию get.

```
    class istream : public virtual ios {
        //...
        istream& get(char& c);                  // символ
        int get();
        istream& get(char* p, int n, char ='n');   // строка
        inline istream& getline(        char *,int,char ='\n');
    };
```

Эти функции предназначены для таких операций ввода, когда не делается никаких предположений о вводимых символах.

Функция istream::get(char&) вводит один символ в свой параметр. Поэтому программу посимвольного копирования можно написать так:

```
int main()
{
        char c;
        while (cin.get(c)) cout.put(c);
}
```

Функция с тремя параметрами istream::get() вводит в символьный вектор не менее n символов, начиная с адреса p. При всяком обращении к get() все символы, помещенные в буфер (если они были), завершаются 0, поэтому если второй параметр равен n, то введено не более n-1 символов.

автор: Коломойцева Ирина Александровна, кафедра Прикладной математики и информатики, ДонНТУ

Третий параметр определяет символ, завершающий ввод. Типичное использование функции get() с тремя параметрами сводится к чтению строки в буфер заданного размера для ее дальнейшего разбора, например так:

```
void f()
{
   char buf[100];
   cin >> buf;              // подозрительно
   cin.get(buf,100,'\n');   // надежно
   //...
}
```

Операция cin>>buf может работать некорректно, поскольку строка из более чем 99 символов переполнит буфер.

Функция get без аргументов вводит одиночный символ из указанного потока и возвращает это символ в качестве значения вызова функции. Этот вариант функции get возвращает EOF, когда в потоке встречается признак конца файла.

Пример. Использование функций get, put и eof.

```
int main()
{
    char c;
    cout<<"Перед вводом cin.eof равен: "<<cin.eof()<<endl;
    while ((c=cin.get())!=EOF) cout.put(c);
    cout<<"После ввода cin.eof равен: "<<cin.eof()<<endl;
    return 0;
}
```

Функция getline действует подобно функции get, вводящей строку, но, в отличие от функции get, функция getline удаляет символ-ограничитель из потока и не сохраняет его в символьном массиве.

Пример. Символьный ввод функцией getline.

```
int main()
{
    const int SIZE=80;
    char buff[SIZE];
    cout<<"Vvedite predlozhenie\n";
    cin.getline(buff,SIZE);
    cout<<"Vvedennoye predlozhenie: \n"<<buff<<endl;
    return 0;
}
```

Функция ignore пропускает заданное число символов (по умолчанию один символ) или завершает свою работу при обнаружении заданного ограничителя. По умолчанию символом-ограничителем является EOF, который заставляет функцию ignore пропускать символы до конца файла.

Функция putback возвращает обратно в этот поток предыдущий символ, полученный из входного потока с помощью get. Функция полезна

для приложений, которые просматривают входной поток с целью поиска записи, начинающейся с заданного символа. Когда этот символ введен, приложение возвращает его в поток, так что он может включен в те данные, которые будут вводиться.

Функция peek возвращает очередной символ из входного потока, но не удаляет этот символ из него

Неформатированный ввод-вывод.

Неформатированный ввод-вывод выполняется с помощью функций read и write. Каждая из них вводит или выводит некоторое число байтов в символьный массив в памяти или из него. Эти байты не подвергаются какому-либо форматированию.

Например, вызов

```
char buff[SIZE]="ABCDEFGHIJKLMNOPRSTUVWXYZ";
cout.write(buff,10);
```

или

```
cout.write("ABCDEFGHIJKLMNOPRSTUVWXYZ",10);
```

выводит первые 10 байтов символьного массива.

Функция read вводит в символьный массив указанное число символов. Если считывается меньшее количество символов, то устанавливается флаг failbit.

Функция gcount сообщает о количестве символов, прочитанных последней операцией ввода.

Пример работы функций неформатированного ввода-вывода.

```
int main()
{
    const int SIZE=80;
    char buff[SIZE];
    cout<<"Vvedite predlozhenie:";
    cin.read(buff,20);
    cout<<"Vvedennoye predlozhenie:";
    cout.write(buff,cin.gcount());
    return 0;
}
```

Форматирование данных

В потоковых классах форматирование выполняется тремя способами – с помощью флагов, манипуляторов и форматирующих методов.

Манипуляторы потоков

автор: Коломойцева Ирина Александровна, кафедра Прикладной математики и информатики, ДонНТУ

В языке С++ есть возможность использовать манипуляторы потоков, которые решают задачи форматирования. Манипуляторы потоков позволяют выполнить следующие операции: задание ширины полей, задание точности, установку и сброс флагов формата, задание заполняющего символа полей, сброс потоков, вставку в выходной поток символа новой строки и сброс потока, вставку нулевого символа в выходной поток и пропуск символов разделителей во входном потоке.

Манипуляторы потоков, задающие основание чисел

Для указания основания вывода чисел используются следующие манипуляторы без параметров:

dec – устанавливает вывод десятичных чисел;

oct – устанавливает вывод чисел в восьмеричной системе счисления;

hex – устанавливает вывод чисел в шестнадцатеричной системе счисления.

Основание выводимых чисел можно также изменить с помощью манипулятора setbase. Этот манипулятор принимает целый параметр со значениями 10, 16 или 8. Так как манипулятор setbase принимает параметр, он называется параметризованным манипулятором. Использование параметризованных манипуляторов требует подключения заголовчного файла iomanip.h.

Основание потока является установленным до тех пор, пока оно не будет изменено явным образом.

Пример. Использование потоков манипуляторов, задающих основание чисел.

Пример. Использование манипуляторов hex, dec, oct и setbase для задания основания выводимых чисел.

```
#include <iostream.h>
#include <iomanip.h>
int main()
{
    int n;
    cout<<"Vvedite chislo: ";
    cin>>n; //60
    cout<<"16 format: "<<hex<<n<<endl; //3c
    cout<<"10 format: "<<dec<<n<<endl; //60
    cout<<"8 format: "<<oct<<n<<endl; //74
    cout<<"10 format: "<<setbase(10)<<n<<endl; //60
    return 0;
}
```

Манипуляторы потоков, задающие формат вывода вещественного числа

Число печатаемых разрядов справа от десятичной точки для вещественного числа можно задать с помощью манипулятора потока

setprecision или метода precision. Вызов любой из этих установок точности действует для последующих операций вывода до тез пор, пока не будет произведена следующая установка точности.

В классе ios объявлено два перегруженных метода precision.

int ios::precision() – возвращает значение точности представления при выводе вещественных чисел;

int ios::precision(int) – устанавливает значение точности представления при выводе вещественных чисел, возвращает старое значение точности.

Пример. Использование метода precision и манипулятора setprecision для задания точности вывода вещественных чисел.

```
#include <iostream.h>
#include <iomanip.h>
int main()
{
    cout.precision(10);
    cout<<sqrt(3.0)<<endl; //1.732050808
    cout<<cout.precision()<<endl; //10
    cout<<setprecision(4)<<sqrt(3.0)<<endl; //1.7321
    return 0;
}
```

Манипуляторы потоков, устанавливающие ширину поля

Метод width класса ios и манипулятор setw устанавливают ширину поля. Под шириной поля понимается число символьных позиций, в которые значение будет выведено, или число символов, которые будут выведены. Если обрабатываемые значения имеют меньше символов, чем заданная ширина поля, то для заполнения лишних позиций используются заполняющие символы. Если число символов в обрабатываемом значении больше, чем заданная ширина поля, то лишние символы не отсекаются.

Установка ширины поля применяется только к текущей операции «поместить в поток» или «взять из потока». После выполнения этих операций ширина поля устанавливается неявным образом на 0. Такая установка означает, что поле для представления выходных данных будет необходимой ширины.

Формат манипулятора setw: setw(int).

Метод width имеет две реализации:

int ios::width() – возвращает значение ширины поля ввода/вывода;

int ios::width(int) – возвращает значение ширины поля ввода/вывода, возвращает старое значение ширины.

Пример. Использование метода width и манипулятора setw для задания ширины поля.

```
#include <iostream.h>
#include <iomanip.h>
int main()
```

```
{
    cin.width(15);
    cin>>str; //012345678901234567890
    cout.width(10);
    cout<<str; //01234567890123
    cout<<endl;
    cin.width(5);
    cin>>str; //012345678901234567890 или 01
    cout<<setw(10)<<str; //xxxxxx0123 или xxxxxxxx01 (x – пробел)
    return 0;
}
```

Замечание. Для того чтобы обеспечить размещение во входной строке нулевого символа, считывается на один символ меньше, чем установленная ширина поля.

Манипуляторы, определяющие вид вводимых–выводимых значений

ws – извлечение пробельных символов;
endl – вставка символа новой строки и очистка потока;
ends – вставка оконечного пустого символа в строку;
flush – сброс на диск и очистка ostream;
setfill(int c) – установка символа-заполнителя в с.

```
#include <iostream.h>
#include <iomanip.h>
int main()
{
    cin.width(5);
    cin>>str; //012345678901234567890
    cout<<setfill('#')<<setw(10)<<str; //######0123
    return 0;
}
```

Работать с символами заполнения можно также с помощью двух перегруженных методов класса ios:

char ios::fill() – возвращает текущий символ заполнения;

char ios::fill(char) – устанавливает значение текущего символа заполнения; возвращает значение старого.

Создание собственных манипуляторов потока

Пользователи могут создавать собственные манипуляторы потокв.

Пример. Создание и использование манипулятора tab.

```
#include <iostream.h>
#include <iomanip.h>
ostream & tab(ostream &output)
{return output<<'\t';}
int main()
```

автор: Коломойцева Ирина Александровна, кафедра Прикладной математики и информатики, ДонНТУ

```
{
    cout<<tab<<tab<<str<<endl;
    return 0;
}
```

Флаги состояний формата

Различные флаги формата (format flags) задают виды форматирования, которые выполняются во время операций ввода–вывода. В таблице приведены различные флаги формата.

Флаг	Описание
ios::skipws	
ios::left	
ios::right	
ios::internal	
ios::dec	
ios::oct	
ios::hex	
ios::showbase;	

ios::showpoint	Определяет, что вещественные числа должны выводиться с десятичной точкой. Этот флаг обычно используется для обеспечения определенного числа цифр справа от десятичной точки
ios::uppercase	Предписывает использовать прописные буквы в шестнадцатеричных числах и экспоненциальной форме вещественных чисел.
ios::showpos	Определяет, что числа должны выводиться со знаком
ios::scientific	Определяет вывод вещественных чисел в экспоненциальной форме
ios::fixed	Определяет вывод вещественных чисел в форме с фиксированной точкой
ios::showpoint	Определяет, что вещественные числа должны выводиться с десятичной точкой. Этот флаг обычно используется для обеспечения определенного числа цифр справа от десятичной точки

Для работы с флагами можно использовать методы класса ios:

long ios::flags() – возвращает текущие флаги.

long ios::flags(long) – присваивает новые значения флагам и возвращает старые.

long ios::setf(long, long) – перед установкой некоторых флагов позволяет сбросить флаги, которые не могут быть установлены одновременно с ними. В качестве первого параметра передается

устанавливаемый флаг, а качестве второго параметра передается одна из следующих статических констант класса ios:

adjustfield (left | right | internal)

basefield (dec | oct | hex)

floatfield (scientific | fixed)

long ios::setf(long) – устанавливает флаги формата, заданные в качестве аргумента и возвращает предыдущие установки этих флагов.

long ios::unsetf(long) – сбрасывает указанные флаги формата и возвращает предыдущие их значения.

Для работы с флагами можно также воспользоваться манипуляторами с параметрами:

setiosflags(long) – устанавливает флаги формата, указанные в аргументе.

resetiosflags(long) – сбрасывает флаги формата, указанные в аргументе.

Пример. Использование флагов состояния формата

```
#include <iostream.h>
#include <iomanip.h>
int main()
{
    cout.setf(ios::showpoint); //устанавливаем флаг вывода вещественных чисел в виде числа с десятичной точкой
    cout<<9.0000<<endl; //9.00000
    cout<<9.9000<<endl; //9.90000
    cout<<9.9900<<endl; //9.90000
    cout.unsetf(ios::showpoint); //снимаем флаг вывода вещественных чисел в виде числа с десятичной точкой
    cout<<9.0000<<endl; //9
    cout<<9.9000<<endl; //9.9
    cout<<9.9900<<endl; //9.99
    cout.setf(ios::hex, ios::basefield); //устанавливаем флаг вывода целого числа в шестнадцатеричном виде
    int x=12345;
    cout<<x; //3039
    cout.unsetf(ios::hex); //снимаем флаг вывода целого числа в шестнадцатеричном виде
    cout<<endl<<setw(30)<<setiosflags(ios::left)<<x; //12345 (устанавливаем флаг вывода символов по левому краю поля)
    cout<<resetiosflags(ios::left)<<endl; // снимаем флаг вывода символов по левому краю поля
    cout.flags(ios:: showpos | ios:: internal); //устанавливаем флаг вывода знака числа и флаг вывода знака числа, прижатого к левому краю, и самого числа, прижатого к правому краю
    cout<<setw(10)<<setfill('#')<<x<<endl;//+####12345
```

автор: Коломойцева Ирина Александровна, кафедра Прикладной математики и информатики, ДонНТУ

```
        cout.flags(ios::left); // устанавливаем флаг вывода символов по левому краю поля
        cout<<setw(10)<<setfill('%')<<x<<endl; //+12345%%%%
        cout.setf(ios::showbase); //устанавливаем флаг специфического отображения шестнадцатеричных и восьмеричных чисел
        cout<<oct<<x<<endl;  //030071
        cout<<hex<<x<<endl; //0x3039
        cout<<dec<<x<<endl; //+12345
        double y=9000, z=0.0009;
        cout.unsetf(ios::showpos); // снимаем флаг вывода знака числа
        cout.setf(ios::scientific | ios::uppercase); //устанавливаем флаг вывода вещественных чисел в экспоненциальной форме и флаг вывода прописных в представлении чисел
        cout<<y<<" "<<z<<endl; //9.00000E+003 9.000000E-004
        cout.unsetf(ios::uppercase); //снимаем флаг вывода прописных в представлении чисел
        cout<<y<<" "<<z<<endl; //9.00000e+003 9.000000e-004
        cout.setf(ios::fixed, ios::floatfield); //устанавливаем флаг вывода вещественных чисел в виде числа с десятичной точкой
        cout<<y<<" "<<z<<endl; //9000.000000 0.000900
        cout.setf(ios::hex | ios::uppercase); ///устанавливаем флаг вывода целого числа в шестнадцатеричном виде и флаг вывода прописных в представлении чисел
        cout<<31<<endl; //0X1F
        return 0;
    }
```

СОСТОЯНИЕ ОШИБОК ПОТОКА

Состояние ошибок потока можно проверить с помощью битов класса ios. Совокупность этих битов определена в поле state класса ios.

```
enum io_state {  goodbit = 0x00,
                 eofbit  = 0x01,
                 failbit = 0x02,
                 badbit  = 0x04 };
```

Состоянием потока можно управлять с помощью методов и операций.

Бит **eofbit** автоматически устанавливается, когда встречается признак конца файла. Для проверки состояния этого бита можно использовать метод **int ios::eof()**. Вызов

cin.eof();

возвращает true, если встретился признак конца файла, и false – в противном случае.

Бит **failbit** устанавливается для потока, если в потоке происходит ошибка форматирования, но символы не утеряны. Проверить состояние этого

бита можно с помощью метода **int ios::fail()**, который возвращает ненулевое значение, если бит **failbit** установлен.

Бит **badbit** устанавливается для потока при возникновении ошибки, которая приводит к потере данных. Проверить состоянии этого бита можно с помощью методов **int ios::fail()** и **int ios::fail()** класса **ios**. Если бит **badbit** установлен, эти функции возвращают ненулевое значение.

Бит **goodbit** устанавливается, если нет ошибок в потоке. Состояние этого бита проверяет функция **int ios::good()**. В случае отсутствия ошибок эта функция возвращает истину.

Метод **int ios::rdstate()** возвращает текущее состояние потока. Вызов **cout.rdstate()**, например, вернул бы состояние потока, которое затем могло бы использоваться в операторе **switch**, который бы проверял состояние битов ошибок потока.

Пример использования метода rdstate():

```
int k=cin.rdstate();
if(k & ios::eofbit) cout<<"Достигнут конец файла"<<endl;
```

Метод **int ios::clear(int = 0)** можно использовать для установления состояния ошибки. Если вызвать этот метод без параметров (вызов по умолчанию), то поток устанавливается в нормальное состояние, при котором можно продолжать выполнять операции ввода-вывода данного потока. Вызов без параметров:

cin.clear();

Установка бита **failbit**:

cin.clear(failbit);

Подобный вызов может использоваться, если возникают проблемы при обработке **cin** с типом, определенным пользователем.

Метод **ios::operator!()** возвращает ненулевое значение, если установлен хотя бы один бит ошибки.

Метод **ios::operator void*()** возвращает ненулевой указатель, если установлен хотя бы один бит ошибки.

Пример. Проверка состояний ошибок.

```
#include <iostream.h>
int main()
{
    int x;
    cout<<"До ошибочной операции ввода:"<<endl;
    cout<<"cin.rdstate(): "<<cin.rdstate()<<endl; //0
    cout<<"cin.eof(): "<<cin.eof()<<endl; //0
    cout<<"cin.fail(): "<<cin.fail()<<endl; //0
    cout<<"cin.bad(): "<<cin.bad()<<endl; //0
    cout<<"cin.good(): "<<cin.good()<<endl; //1
    cin>>x; //вводится символ
    cout<<"После ошибочной операции ввода:"<<endl;
    cout<<"cin.rdstate(): "<<cin.rdstate()<<endl; //2
```

автор: Коломойцева Ирина Александровна, кафедра Прикладной математики и информатики, ДонНТУ

```
    cout<<"cin.eof(): "<<cin.eof()<<endl; //0
    cout<<"cin.fail(): "<<cin.fail()<<endl; //2
    cout<<"cin.bad(): "<<cin.bad()<<endl; //0
    cout<<"cin.good(): "<<cin.good()<<endl; //0
    cout<<!cin<<endl; //2
    cin.clear();
    cout<<"После cin.clear():" <<endl;
    cout<<"cin::fail(): "<<cin.fail()<<endl; //0
    cout<<"cin.good(): "<<cin.good()<<endl; //1
    cout<<!cin<<endl; //0
    return 0;
}
```

Связывание выходного и входного потоков

Интерактивные приложения обычно включают класс **istream** для ввода и класс **ostream** для вывода. Очевидно, что в таких приложениях приглsd,uiашение на ввод должно появляться до осуществления операции ввода. В языке C++ есть метод **tie**, который выполняет синхронизацию (связывание) выполнения операций над потоками **istream** и **ostream**. Этот метод гарантирует, что вывод появится раньше последующего ввода. В C++ происходит автоматическое связывание объектов **cin** и **cout**:

cin.tie(&cout);

При программировании следует связывать другие пары потоков классов **istream** и **ostream**.

Например, для того чтобы развязать входной поток InputStream от выходного:

InputStream.tie(0);

Файлы и потоки

В C++ каждый файл рассматривается как последовательность байтов. Каждый файл завершается маркером конца файла (end-of-file marker). Когда файл открывается, то создается объект и с этим объектом связывается поток.

Для обработки файлов в C++ должны быть включены в программу заголовочный файлы <**iostream.h**> и <**fstream.h**>. Файл <**fstream.h**> включаеи определения классов потоков **ifstream** (для ввода из файла), **ofstream** (для вывода в файл) и **fstream** (для ввода-вывода файлов). Файлы открываются путем создания объектов этих классов потоков. Так эти классы потоков являются производными от классов **istream**, **ostream** и **iostream**, то они могут использовать методы, операции и манипуляторы, определенные в их базовых классах.

Создание файла последовательного доступа

автор: Коломойцева Ирина Александровна, кафедра Прикладной математики и информатики, ДонНТУ

Так как C++ не предписывает файлу никакой структуры, то программист должен задавать структуру файлов в соответствии с требованиями прикладных программ.

Пример. Занесение данных о студенте, его группе и среднем балле в текстовый файл.

```
#include <iostream.h>
#include <fstream.h>
int main()
{
```

/*Так как файл открывается для ввода, то создается объект класса ofstream. Конструктору этого класса передается два параметра – имя файла и режим открытия файла.*/

```
ofstream outGroupFile("student.dat",ios::out);
```

/*Для проверки того, успешно ли открылся файл, используется перегруженный метод opearator! класса ios. Если файл не открыт, он возвращает ненулевое значение.*/

```
if(!outGroupFile)
{
   cerr<<"File not open"<<endl;
   return 0;
}
char name[30],group[10];
double AverMark;
cout<<"?:  ";
while(cin>>name>>group>>AverMark)
```

{ /*Условие в заголовке оператора while автоматически вызывает перегруженный метод operator void* класса ios. Эта функция превращает поток в указатель, который можно проверить на равенство 0. Ввод признака конца файла (Ctrl+Z) установит бит failbit для cin, после проверки которого метод operator void* вернет 0, и условие цикла while станет ложным. Таким образом, метод operator void* можно использовать для проверки конца файла в объекте ввода вместо явного вызова метода eof.*/

```
   outGroupFile<<name<<' ' <<group<<' '<<AverMark<<'\n';
```

/* информация записывается в файл «student.dat» с помощью объекта outGroupFile, связанного с этим файлом, и операции «поместить в поток» <<.*/

```
   cout<<"?:  ";
}
outGroupFile.close();
```

/*метод close() осуществляет явный вызов деструктора объекта outGroupFile, закрывающего файл.

```
return 0;
}
```

автор: Коломойцева Ирина Александровна, кафедра Прикладной математики и информатики, ДонНТУ

Таблица. Режимы открытия файлов

Режим	**Описание**
ios::app	Записать данные в конец файла без модификации имеющихся данных в файле.
ios::ate	Переместиться в конец исходного открытого файла. Данные могут быть записаны в любое место файла.
ios::in	Открыть файл для ввода.
ios::out	Открыть файл для вывода. Если в файле есть данные, то они уничтожаются. Если файла не существует, он создается.
ios::trunc	Уничтожать содержимое файла, если он существует.
ios::binary	Открыть файл для двоичного ввода или вывода.

Таблица. Соответствие между режимами открытия файла класса ios и режимами открытия файла, описанными в <stdio.h>

Комбинация флагов ios					Эквивалент stdio
binary	in	out	trunc	app	
		+			“w”
		+		+	“a”
		+	+		“w”
	+				“r”
	+	+			“r+”
	+	+	+		“w+”
+		+			“wb”
+		+		+	“ab”
+		+	+		“wb”
+	+				“rb”
+	+	+			“r+b”
+	+	+	+		“w+b”

Замечание. Кроме как с помощью конструктора, открыть файл в программе можно с помощью метода **open**, имеющего такие же параметры как и конструктор. Например,

```
ofstream outGroupFile;
outGroupFile.open("student.dat",ios::out);
```

Пример. Чтение данных из файла последовательного доступа и вывод их на экран.

```
#include <iostream.h>
#include <fstream.h>
#include <iomanip.h>
void OutputLine(const char* name, const char *group, double mark)
{
    cout<<setiosflags(ios::left)<<setw(20)<<name
        <<setw(10)<<group<<setw(8)<<setprecision(2)
        <<resetiosflags(ios::left)<<mark<<'\n';
}
```

автор: Коломойцева Ирина Александровна, кафедра Прикладной математики и информатики, ДонНТУ

```
int main()
{
    ifstream inGroupFile;
```

/* Как и класс ofstream класс ifstream имеет конструктор с двумя аргументами – именем файла и режимом его открытия. Если режим открытия файла не указан, то значением по умолчанию является ios::in. Таким образом, можно создать объект класса ifstream и передать ему один параметр – имя файла. Например,

ifstream inGroupFile("student.dat");

Для открытия файла на чтение можно также воспользоваться созданием объекта класса ifstream с помощью конструктора без параметров с последующим вызовом для него метода open*/

```
    inGroupFile.open("student.dat",ios::in);
    if(!inGroupFile)
```

{/*К объекту класса ifstream, как и к объекту класса ofstream, можно применить перегруженный метод operator! для того, чтобы проверить, успешно ли открылся файл*/

```
       cerr<<"File not open"<<endl;
       return 0;
    }
    char name[30],group[10];
    double AverMark;
    cout<<setiosflags(ios::left)<<setw(20)<<"Name of student"
        <<setw(10)<<"Group"<<"Av. mark\n"<<
    setiosflags(ios::fixed|ios::showpoint);
    while(inGroupFile>>name>>group>>AverMark)
```

/*В условии цикла while используется перегруженный метод operator void*() для чтения данных из файла. Этот метод возвращает 0, если будет достигнут конец файла.*/

```
       OutputLine(name,group,AverMark);
    inGroupFile.close();
    return 0;
}
```

МЕТОДЫ ПОЗИЦИОНИРОВАНИЯ УКАЗАТЕЛЯ В ФАЙЛЕ

Классы **istream** и **ostream** содержат методы для позиционирования указателя файла. Под указателем понимается порядковый номер следующего байта в файле, который должен быть считан или записан. Любой объект класса **istream** имеет указатель «**get**», который показывает номер в файле очередного вводимого байта. Любой объект класса **ostream** имеет указатель «**put**», который показывает номер в файле очередного выводимого байта.

Функция

istream& seekg(streamoff offset, ios::seek_dir origin);

при извлечении данных из файла перемещает указатель на **offset** байт от позиции, заданной параметром **origin**.

автор: Коломойцева Ирина Александровна, кафедра Прикладной математики и информатики, ДонНТУ

Функция

ostream& seekp(streamoff offset, ios::seek_dir origin);

во время чтения данных из файла перемещает указатель на **offset** байт от позиции, заданной параметром **origin**.

Параметр **offset** должен принимать целое значение.

Параметр **origin** представляет собой перечисление, которое имеет следующие значения:

ios::beg – смещение от начала;

ios::cur – смещение от текущей позиции;

ios::end – смещение от конца.

Функции

ostream& seekp(streampos position);

istream& seekg(streampos position);

перемещают указатель от начала файла на **position** байт, соответственно, при чтении и извлечении данных из файла.

Функция

streampos tellg();

возвращает значение указателя взять из потока «**get**».

Функция

streampos tellp()

возвращает значение указателя поместить в поток «**put**».

Например,

istream FileObject;

FileObject.seekg(n); //позиционирование FileObject на n-й байт от начала файла

FileObject.seekg(n,ios::cur); //позиционирование FileObject на n байтов вперед

FileObject.seekg(k,ios::end); //позиционирование FileObject на k-й байт от конца файла

FileObject.seekg(0,ios::end); //позиционирование FileObject на конец файла

location=FileObject.tellg();//присваивание переменной location значения файлового указателя «get»

Файлы произвольного доступа

Файлы последовательного доступа не подходят для решения задач, требующих немедленного доступа к локализованной записи из большого объема данных (системы резервирования билетов, банковские системы, система терминалов для производства платежей в месте совершения покупок, банковские автоматы). Для решения таких задач используются файлы произвольного доступа. Для создания файлов произвольного используется несколько методов. Наиболее простым из них является метод, при котором к записям в файле предъявляется требование одинаковой длины. В этом случае

местоположение любой записи определяется функцией, зависящей от размера записи и ключа записи.

Данные могут быть вставлены в файл прямого доступа без разрушения других данных файла. Данные, которые в нем хранятся, могут быть изменены или удалены без перезаписи всего файла.

автор: Коломойцева Ирина Александровна, кафедра Прикладной математики и информатики, ДонНТУ

ШАБЛОНЫ ФУНКЦИЙ И КЛАССОВ

Шаблоны – одно из последних нововведений стандарта языка C++. Шаблоны дают возмжоность определять при помощи одного фрагмента кода целый набор взаимосвязанных функций (*перегруженных*), называемых *шаблонными функциями*, или набор связанных классов, назыввемых *шаблонными классами*.

Например, можно написать один шаблон функции сортировки массива, на основе которого C++ будет автоматически генерировать отдельные шаблонные функции, сортирующие массивы типов int, float, массив строк и т.д. Или достаточно описать один шаблон класса стеков, а затем C++ будет автоматически создавать отдельные шаблонные классы типа стек целых, стек вещественных чисел, стек строк и т.д.

Шаблоны функций

Перегруженные функции обычно используются для выполнения похожих операций над различными типами данных. Если для каждого типа данных должны выполняться идентичные операции, то оптимальным решением является использование шаблонов функций. При этом программист должен написать всего одно описание шаблона функции. Основываясь на типах аргументов, использованных при вызове этой функции, компилятор будет автоматически генерировать объектные коды функций, обрабатывающих каждый тип данных. *В языке C эта задача выполнялась при помощи макросов, определяемых директивой препроцессора #define. Однако при выполнении макросов компилятор не выполняет проверку соответствия типов, из-за чего нередко возникают ошибки. Шаблоны функций, являясь таким же компактным решением, как и макросы, позволяют компилятору полностью контролировать соответствие типов.*

Все описания шаблонов функций начинаются с ключевого слова **template**, за котором следует список формальных параметров шаблона, заключаемый в угловые скобки (< и >); каждому формальному параметру должно предшествовать ключевое слово **class** или **typename**. Например,

template <class T>

или

template <typename ElementType>

или

template <class BorderType, class FillType>

Формальные параметры в описании шаблона используются (наряду с параметрами встроенных типов или типов, определяемых пользователем) для определения типов параметров функции, типа возвращаемого функцией значения и типов переменных, объявляемых внутри функции. Далее, за этим заголовком, следует обычное описание функции. Ключевое слово **class** или

автор: Коломойцева Ирина Александровна, кафедра Прикладной математики и информатики, ДонНТУ

typename, используемое в шаблоне функции при задании типов параметров, означает «любой встроенный тип или тип, опередеяемый пользователем».

Замечание. Ошбикой является отсутствие ключевого слова **class** (или **typename**) перед каждым формальным параметром типа шаблона функции.

```
template <class T>
T& inc_value(T &val)
{
    ++val;
    return val;
}
int main()
{
    int x = 0;
    x=(int)inc_value<int>(x);
    cout<<x<<endl;
    char c = 0;
    c=(char)inc_value<char>(c);
    cout<<c<<endl;
    return 0;
}
template <class T1>
void PrintArray(const T1* array, const int count)
{
      for (int i=0; i<count; i++)
            cout<<array[i]<<" ";
      cout<<endl;
}
```

В шаблоне функции PrintArray() объявляется формальный параметр T1 для массива, который будет выводиться функцией PrintArray().

T1* называется *параметром типа*. Когда компилятор обнаруживает в тексте программы вызов функции PrintArray(), он заменяет T1 во всей области определения шаблона тип первого параметра функции PrintArray() и C++ создает шаблоную функцию вывода массива указанного типа данных. После этого вновь созданная функция компилируется.

```
int main()
{
    const int aCount=5, bCount=7, cCount=6;
    int a[aCount]={1,2,3,4,5};
    double b[bCount] = {1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7};
    char c[cCount]="HELLO"; //6-я позиция для null
    cout<<"Array a:"<<endl;
    PrintArray(a,aCount); //шаблон для integer
    cout<<"Array b:"<<endl;
    PrintArray(b,bCount);  //шаблон для double
```

автор: Коломойцева Ирина Александровна, кафедра Прикладной математики и информатики, ДонНТУ

```
    cout<<"Array c:"<<endl;
    PrintArray(c,cCount); //шаблон для character
    return 0;
}
```

Шаблоны функций расширяют возможности многократного использования программного кода, но программа может создавать слишком много копий шаблонных функций и шаблонных классов. Для этих копий могут потребоваться значительные ресурсы памяти.

В примере шаблонный механизм позволяет программисту избежать необходимости написания трех отдельных функций с прототипами:

```
void PrintArray(const int*, const int);
void PrintArray(const double*, const int);
void PrintArray(const char*, const int);
```

которые используют один и тот же код, кроме кода для типа T1.

автор: Коломойцева Ирина Александровна, кафедра Прикладной математики и информатики, ДонНТУ

Перегрузка шаблонных функций

Шаблонные функции и перегрузка функций тесно связаны друг с другом. Все родственные функции, полученные из шаблона, имеют одно и то же имя; поэтому компилятор использует механизм перегрузки для того, чтобы обеспечить вызов соответствующей функции.

Сам шаблон функции может быть перегружен несколькими способами. Во–первых, можно определить другие шаблоны, имеющие то же самое имя функции, но различные наборы параметров. Например,

```
template <class T1>
void PrintArray(const T1* array, const int count)
{
      for (int i=0; i<count; i++)
            cout<<array[i]<<" ";
      cout<<endl;
}
template <class T1>
void PrintArray(const T1* array, const int lowSubscript, const int highSubscript)
{
        for (int i=lowSubscript; i<=highSubscript; i++)
        cout<<array[i]<<" ";
        cout<<endl;
}
int main()
{
        const int aCount=5, bCount=7, cCount=6;
        int a[aCount]={1,2,3,4,5};
        double b[bCount] = {1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7};
        char c[cCount]="HELLO"; //6-я позиция для null
        …
        cout<<"Array a from 1 to 3:"<<endl;
        PrintArray(a,0,2); //шаблон для integer
        cout<<"Array b from 4 to 7:"<<endl;
        PrintArray(b,3,6);  //шаблон для double
        cout<<"Array c from 3 to 5:"<<endl;
        PrintArray(c,2,4); //шаблон для character
        return 0;
}
```

Шаблон функции может быть также перегружен, если ввести другую нешаблоную функцию с тем же самым именем, но другим набором

параметров. Например, шаблон функции PrintArray можно перегрузить версией нешаблонной функции, которая выводит массив символов в столбик.

```
template <class T1>
void PrintArray(const T1* array, const int count)
{…}
template <class T1>
void PrintArray(const T1* array, const int lowSubscript, const int highSubscript)
{…}
void PrintArray(char* array, const int count)
{
    for (int i=0; i<count; i++)
        cout<<array[i]<<endl;
}

int main()
{
      const int cCount=6;
      char c[cCount]="HELLO"; //6-я позиция для null
      cout<<"Array c:"<<endl;
      PrintArray(c,cCount);
      …
      return 0;
}
```

Компилятор действует по следующему алгоритму. Сначала он пытается найти и использовать функцию, которая точно соответствует по своему имени и типам параметров вызываемой функции. Если такая функция не находится, то компилятор ищет шаблон функции, с помощью которого он может сгенерировать шаблонную функцию с точным соответствием типов параметров и имени функции. Если такой шаблон обнаруживается, то компилятор генерирует и использует соответствующую шаблонную функцию.

Замечание. Компилятор ищет шаблон, полностью соотвествующий вызываемой функции по типу всех параметров; автоматическое преобразование типов не производится (возможно, char к int, double к int и т.д.).

Шаблоны классов

С помощью шаблона классов можно достаточно просто определить и реализовать без потерь в эффективности выполнения программы и, не отказываясь от статического контроля типов, такие контейнерные классы, как списки и ассоциативные массивы. Кроме того, шаблоны класса

позволяют определить сразу для целого семейства типов обобщенные (генерические) функции, например, такие, как sort (сортировка).

Рассмотрим простой шаблон класса stack (стек). Понятие «стек» не зависит от типа данных, которые помещаются в стек. Тип данных должен быть задан только тогда, когда приходит время создать стек «фактически». Поэтому достаточно создать некое общее описание понятия стека и на основе этого родового класса создавать классы, являющиеся специфическими версиями для конкретного типа данных.

Шаблоны классов называются еще *параметризованными типами*, так как они имеют один или большее количество параметров типа, определяющих настройку подового шаблона класса на специфический тип данных при создании объекта класса.

```
template<class T>
class stack
{
      T* v;
      T* p;
      int sz;
  public:
      stack(int s) { v = p = new T[sz=s]; }
      ~stack() { delete[] v; }
      void push(T a) { *p++ = a; }
      T pop() { return *--p; }
      int size() const { return p-v; }
};
```

Префикс template<class T> указывает, что описывается шаблон класса с параметром T, обозначающим тип классов Stack которые будут создаваться на основе этого шаблона. Идентификатор T определяет тип данных– элементов, хранящихся в стеке.

Имя шаблонного класса, за которым следует тип, заключенный в угловые скобки <>, является именем класса (определяемым шаблоном класса), и его можно использовать как все имена класса. Например, ниже определяется объект sc класса stack<char>:

```
stack<char> sc(100);   // стек символов
```

Если не считать особую форму записи имени, класс stack<char> полностью эквивалентен классу определенному так:

```
class stack_char {
      char* v;
      char* p;
      int sz;
public:
      stack_char(int s) { v = p = new char[sz=s]; }
      ~stack_char() { delete[] v; }
      void push(char a) { *p++ = a; }
```

```
        char pop() { return *--p; }
        int size() const { return p-v; }
};
```

Имея определение шаблонного класса stack, можно следующим образом определять и использовать различные стеки:

```
typedef struct _Point
{int x,y;}Point;
class shape {};
void f(stack<Complex>& sc)  // параметр типа 'ссылка на complex'
{
    stack<shape*> ssp(200);  // стек указателей на фигуры
    stack<Point> sp(400);    // стек структур Point
    sc.push(Complex(1,2));
    Complex z=sc.pop();
    z*2.5;
    cout<<z;
    stack<int>*p = 0;        // указатель на стек целых
        p = new stack<int>(800);  // стек целых размещается в свободной памяти
    for ( int i = 0; i<400; i++)
    {
        p->push(i);
        Point p1;
        p1.x=i;
        p1.y=i+400;
        sp.push(p1);
    }
    cout<<sp.size()<<endl;
    //…
}
```

В случае внешней реализацию методов шаблона классов stack сам шаблон определяется следующим образом:

```
template<class T>
class stack
{
        T* v;
        T* p;
        int sz;
public:
        stack(int);
        ~stack();
        void push(T);
        T pop();
        int size() const;
```

автор: Коломойцева Ирина Александровна, кафедра Прикладной математики и информатики, ДонНТУ

```
};
```

В этом случае определение методов шаблона классов stack выполняется, как и для функций-членов обычных, нешаблонных классов. Подобные функции так же параметризируются типом, служащим параметром для их шаблонного класса, поэтому определяются они с помощью шаблона класса для функции. Если это происходит вне шаблонного класса, это надо делать явно:

```
template<class T> void stack<T>::push(T a)
{*p++ = a;}
template<class T> T stack<T>::pop()
{return *--p;}
template<class T> int stack<T>::size() const
{return p-v;}
template<class T> stack<T>::stack(int s)
{v = p = new T[sz=s];}
template<class T> stack<T>::~stack()
{delete[] v;}
```

Для приведенного примера (функция f) компилятор должен создать определения конструкторов для классов stack<shape*>, stack<Point>, stack<Complex> и stack<int>, деструкторов для stack<Complex>, stack<shape*> и stack<Point>, версии функций push() для stack<complex>, stack<int> и stack<Point> и версию функции pop() для stack<complex>. Такие создаваемые функции будут совершенно обычными методами, например:

```
void stack<complex>::push(complex a) { *p++ = a; }
```

Здесь отличие от обычного метода только в форме имени класса. Точно так же, как в программе может быть только одно определение метода класса, возможно только одно для метода шаблонного класса. Е

Важно составлять определение шаблона классов таким образом, чтобы его зависимость от глобальных данных была минимальной. Дело в том, шаблон классов будет использоваться для порождения функций и классов на основе заранее неизвестного типа и в неизвестных контекстах. Практически любая, даже слабая зависимость от контекста может проявиться как проблема при отладке программы пользователем, который, вероятнее всего, не был создателем шаблона классов.

Шаблоны класса для списка

```
template<class type>
class CListTemplate
{
    struct SNode
    {
    type *data;
    SNode *next;
```

автор: Коломойцева Ирина Александровна, кафедра Прикладной математики и информатики, ДонНТУ

```
    };
    SNode *head,*p;
  public:
    CListTemplate() {head=p=0;}
    void AddElem(type *d);
    bool RemoveElem(type *d);
    void DisplayList();
    void clear();
    ~CListTemplate(){clear();}
};
template<class type>
void CListTemplate<type>::AddElem(type *d)
{
 if(head)
 {
    p->next = (SNode*)malloc(sizeof(SNode));
    p = p->next;
 }
 else
 {
    head = (SNode*)malloc(sizeof(SNode));
    p=head;
 }
 p->data = d;
 p->next = 0;
}
template<class type>
bool CListTemplate<type>::RemoveElem(type *d)
{
    SNode *curr = head;
    SNode *p1;
    if (*(head->data)==*d)
    {
       p1=head;
       head=head->next;
       delete p1;
       return true;
    }
    while(curr->next && *(curr->next->data)!=*d)
       curr=curr->next;
    if (curr->next)
    {
       ``p1=curr->next;
       curr->next=curr->next->next;
```

```
            delete p1;
            return true;
        }
        else
            return false;
    }
    template<class type>
    void CListTemplate<type>::DisplayList()
    {
            SNode *curr=head;
            while (curr)
            {
                cout<<*(curr->data)<<" ";
                curr=curr->next;
            }
            cout<<endl;
    }
    template<class type>
    void CListTemplate<type>::clear()
    {
        SNode *curr=head;
        while(head)
        {
            curr=head;
            head=head->next;
            delete curr;
        }
    }
    int main()
    {
        CListTemplate<Complex> cl;
        CListTemplate<int> c_int;
        for (int i=0;i<10; i++)
        {
            cl.AddElem(new Complex(10.0f*i,10.0f+i));
            c_int.AddElem(new int(10*i));
        }
        cl.DisplayList();
        Complex c_ob(40.0f,14.0f);
        cl.RemoveElem(&c_ob);
        cl.DisplayList();
        c_int.DisplayList();
        i=90;
        c_int.RemoveElem(&i);
```

автор: Коломойцева Ирина Александровна, кафедра Прикладной математики и информатики, ДонНТУ

```
    c_int.DisplayList();
    return 0;
}
```

Шаблоны классов и нетиповые параметры

Шаблон класса stack, рассмотренный ранее, использовал только параметр типа в заголовке шаблона. Но в шаблонее имеется возможность использовать и нетиповые параметры. Например, в заголовок шаблона можно добавить нетиповой параметр int elements, который будет содержать колчиество элементов стека:

template <class T, int elements>

Тогда оператор

Stack <double, 100> s1;

задает шаблонный класс Stack с именем s1, состоящий из 100 элементов типа double.

Теперь в описании шаблона класса вместо объявлений в разделе закрытых членов-данных

T* v;

T* p;

необходимо поместить следующее:

T p[elements];

int count;

Замечание. Если можно определить размер класса-контейнера (такого как массив или стек) во время компиляции (нарпимер, при помощит нетипового параметра), это повысит скорость выполнения программы, устранив потери времени на динамическое выделение памяти при выполнении программы.

Вложенные шаблонные классы

Объявление вложенного шаблонного класса выглядит следующим образом:

```
template<class ta>
class A
{
 public:
    template<class tai>
    class AI
    {
       public:
          AI(ta ta1,tai tai1):ta_(ta1),tai_(tai1){}
       private:
          ta ta_;
```

```
        tai tai_;
    };
};
int main()
{
    A<int> a;
    A<int>::AI<int> ai(10,20);
    return 0;
}
```

В функции main() показана форма вызова конструктора вложенного шаблонного класса.

Пример вызов функций из вложенных шаблонных классов внутри родительского класса:

```
template<class ta>
class A
{
 public:
    template<class tai>
    class AI
    {
       public:
          void func(){}
    };
    void myfunc()
    {
       AI<double> a;
       a.func();
    }
};
```

Добавим в консруктор вложенного класса параметры:

```
template<class ta>
class A
{
 public:
    template<class tai>
    class AI
    {
       public:
          AI(const ta &v1,const tai &v2){}
    };
    void myfunc()
    {
       AI<double> a;
       a.func();
```

```
    }
};
};
int main()
{
    A<int> a;
    //a.myfunc();
    return 0;
}
```

Пример успешно откомпилируется. Но если раскомментировать строку с функцией a.myfunc(), то компилятор выдаст ошибку. В данном случае имеет место следующая особенность шаблонов: компилятор не проверяет ни параметры, ни вызываемые функции до тех пор, пока они не используются.

Данные особенности поведения шаблонов позволяют написать такой фрагмент программы:

```
template<class ta>
class A
{
 public:
    template<class tai>
    class AI
    {
       public:
          AI(const ta &v1,const tai &v2)
          {
             v1.showValue();
             v1.checkValue(10);
          }
    };
    public:
       void myfunc()
       {
          AI<double> a(100,20.3);
       }
};
int main()
{
    A<int> a;
    //a.myfunc();
    return 0;
}
```

В теле конструктора вложенного шаблонного класса AI(const ta &v1,const tai &v2) вызываются две функции showValue() и checkValue(10) от типа-шаблона, причем в данном случае компилятор не проверяет, есть ли у

данного типа такие функции-члены. Раскомментировав в функции main() строку a.myfunc(), мы получим ошибку компиляции. А теперь добавим к этому примеру определение структуры SBase:

```
struct SBase
{
    int val;
    SBase(int v=0):val(v){}
    void showValue()const{ printf("value: %d\n",val); }
    void checkValue(int)const{}
};
template<class ta=SBase>
class A
{
 public:
    template<class tai>
    class AI
    {
       public:
          AI(const ta &v1,const tai &v2)
          {
             v1.showValue();
             v1.checkValue(10);
          }
    };
    public:
       void myfunc()
       {
          AI<double> a(100,20.3);
       }
};
int main()
{
    A<SBase> a;
    a.myfunc();
    A<> a1;
    a1.myfunc();
    return 0;
}
```

В строке AI<double> a(100,20.3) в качестве первого параметра передано число, т.е. предполагается, что шаблонный тип имеет конструктор с типом int – в нашем случае именно для этого определен конструктор структуры SBase – SBase(int v=0):val(v){}.

Теперь рассмотрим такой пример:

```
template<class T>
```

```
class A
{
    template<class T1>
    class Iterator
    {
       private:
          void calc(){}
    };
 public:
    void func()
    {
       Iterator<int> it;
       it.calc();
    }
};
int main()
{
    A<double> a;
    a.func();
    return 0;
}
```

В этом примере была предпринята попытка обратится к private-функции вложенного класса - компилятор, естественно, нам этого не позволил. Чтобы исправить ситуацию нужно объявить родительский класс дружественным для вложенного. Делается это так:

```
template<class T>
class A
{
    template<class T1>
    class Iterator
    {
       friend class A<T>;
       private:
          void calc(){}
    };
 public:
    void func()
    {
       Iterator<int> it;
       it.calc();
    }
};
 int main()
 {
```

```
    A<double> a;
    a.func();
    return 0;
}
```

Теперь все работает.

Наследование шаблонных классов

Механизм наследования шаблонных классов практически ничем не отличается от механизма наследования обычных классов, отличия только в необходимости объявлять шаблонные типы-параметры. Смотрим пример:

```
template<class T>
class A
{
 public:
    A(const T &t_):t(t_){}
 private:
    T t;
};
template<class T1>
class B : public A<T1>
{
 public:
    B(const T1 &t_):A<T1>(t_){}
};
int main()
{
    B<double> b(10.0);
    return 0;
}
```

Рассмотрим применение на практике полезных особенностей шаблонных классов.

Например, есть несколько сходных классов, необходимо перекрыть одну (или несколько) виртуальную функцию. Подразумевается, что данные базовые классы логически связаны, т.е. перекрываемая виртуальная функция выполняет одинаковое для всех классов действие, например возвращает координату объекта или еще что-либо. Эту проблему можно решить двумя способами.

1. От каждого класса произвести потомка с перекрытой соответствующей функцией. В этом случае программа будет работать корректно, но в ней будет присутствовать дублирование кода в нескольких классах.

2. Использование шаблона. Пример:

```
class A
```

```
{
 public:
    virtual void print(){ printf("A::print()\n"); }
};
class B
{
 public:
    virtual void print(){ printf("B::print()\n"); }
};
class C
{
 public:
    virtual void print(){ printf("C::print()\n"); }
};
template<class T>
class D : public T
{
 public:
    virtual void print(){ printf("template D::print()\n"); }
};
int main()
{
    D<A> a1;
    a1.print(); // template D::print()
    D<B> b1;
    b1.print(); // template D::print()
    D<C> c1;
    c1.print();// template D::print()
    return 0;
}
```

Рассморитм второй пример участия шаблонов в иерархии классов:

```
class A
{
 public:
    void calc(){}
};
template<class T>
class B : public T
{
 public:
    void print(){}
};
class C : public B<A>
{
```

```
 public:
    void anyfunc(){}
};
int main()
{
    C c;
    c.print();
    c.anyfunc();
    c.calc();
    return 0;
}
```

В качестве вывода можно заметить следующее. Шаблоны и наследование связаны друг с другом следующим образом:

- шаблон класса может быть производным от шаблонного класса;
- шаблон класса может являться производным от нешаблонного класса;
- шаблонный класс может быть производным от шаблона класса;
- нешаблонный класса может быть производным от шаблона класса.

Шаблоны и друзья

Для шаблонов классов могут определены отношения дружественности. Дружественность может быть установлена между шаблоном класса и глобальной функцией, функцией членом другого класса (возможно, шаблонно класса) или целым классом (возможно, шаблонным классом).

Оформление отношений дружественности достаточно сложно.

Ели внутри шаблона класса X, который объявлен как

template <class T> class X

находится объявление дружественной функции

friend void f1();

то функция f1 является дружественной для каждого шаблонного класса, полученного из данного шаблона.

Если внутри шаблона класса X, который объявлен как

template <class T> class X

находится объявление дружественной функции в форме

friend void f2(X <T> &);

то для конкретного типа T, например, float, дружественной для класса X<float> будет только функция f2(X <float> &).

Внутри шаблона классов можно объявить функцию-член другого класса дружественной для любого шаблонного класса, полученного из данного шаблона. Для этого нужно использовать имя функции-члена другого класса, имя этого класса и бинарную операцию разрешения области дейстаия. Например, если внутри шаблона класса X, который был объявлен как

template <class T> class X

объявляется дружественная функция в форме

friend void A::f4();

то функция-член f4 класса A будет дружественной для каждого шаблонного класса, полученного из данного шаблона.

Внутри шаблона класса X, который был объявлен объявлен следующим образом:

template <class T> class X

объявление дружественной функции в виде

friend void C<T>::f5(X <T>&);

для конкретного типа T, например int, сделает функцию-член

C<int>::f5(X <int>&)

другом только шаблонного класса X<int>.

Внутри шаблона класса X, объявленного как

template <class T> class X

можно обяъвить дружественный класс Y

friend class Y;

В результате чего каждая из функций-членов класса Y будет дружественной для каждого шаблонного класса, произведенного из шаблона класса X.

Если внутри шаблона класса X, который был объявлен как

template <class T> class X

объявлен второй класс Z в виде

friend class Z<T>;

то при создании шаблонного класса с конкретным типом T, например, типом double, все члены класса Z<double> будут друзьями шаблонного класса X <double>.

Шаблоны и статические члены

Напомним: в нешаблонных классах одна копия статического члена данных используется всеми представителями класса, и статический член данных должен быть инициализирован в области действия файла.

Каждый шаблонный класс, полученный из шаблона классов, имеет собственную копию каждого статического члена данных шаблона; все экземпляры этого шаблонного класса используют свой статический член данных. Статические члены данных шаблонных классов должны быть инициализированы в области действия файла. Каждый шаблонный класс получает собственную копию статической функции-члена шаблонного класса.

```
template<class T>
class A {
    public:
       static int a;
```

```
};
template<class T>
int A<T>::a=10;
int main() {
    A <double> b1;
    b1.a=20;
    A <double> b2;
    b2.a=50;
    cout<<b1.a<<endl; //50
    A <char> b3;
    cout<<b3.a<<endl;
    A <char> b4;
    b4.a=b3.a*2;
    cout<<b3.a<<endl;
    return 0;
}
```

автор: Коломойцева Ирина Александровна, кафедра Прикладной математики и информатики, ДонНТУ

Литература

1. Буч Г. Объектно-ориентированное проектирование с примерами применения. – М.: Конкорд, 1992. – 519 с.
2. Фейсон Т. Объектно-ориентированное программирование на Borland C++ 4.5 – К.: Диалектика, 1996. – 544 с.
3. Грегори К. Использование Visual C++ 6.0. Специальное издание. – М.; СПб.; К.: Издательский дом «Вильямс», 1999. – 864 с.
4. C/C++. Программирование на языке высокого уровня/ Т. А. Павловская. – СПб.: Питер, 2002. 464 с.
5. Лафоре Р. Объектно-ориентированное программирование в C++. пер. с англ. А. Кузнецов и др. - 4-е изд. - СПб. : Питер, 2003. - 928с. : ил
6. Павловская Т.А., Щупак Ю.А. C++. Объектно-ориентированное программирование: Практикум. – СПб.: Питер, 2006. – 265 с.
7. Страуструп Б. Язык программирования C++ / Под ред. Ф. Андреева и А.Ушакова; Пер. с англ. С. Анисимова и М.Кононова. - Спец. изд. - М.: Бином-Пресс, 2005. - 1104 с.: ил.

автор: Коломойцева Ирина Александровна, кафедра Прикладной математики и информатики, ДонНТУ