Лист (двусвязный список)

Найти С++ 10!!!!!

Литература: обьектно-ориентированное программирование на языке С++ (луцик, комличенко)

Как программировать на С++ (дейтел)

Язык программирования С++ (страуструп)

Концепция объектно-ориентированного программирования

К базовым принципам объектно-ориентированного программирования относятся:

1. Инкапсуляция (пакетирование)
2. Наследование
3. Полиморфизм
4. Передача сообщений

Инкапсуляция означает, что в качестве единого целого, называемого объектом, рассматривается некоторая структура данных и некоторая группа методов (функций) для манипуляции этими данными. В С++ данные хранятся в структурах, которые напоминают обычные структуры языка С, а поведение объектов реализуется с помощью методов (функций), связанных с этими данными специальным описанием.

Struct STUD

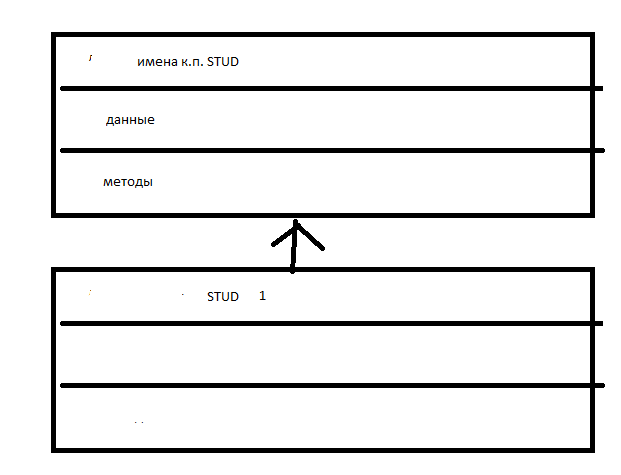
{ char fio[40];

Int god;

Void vvod;

Void show();

};

Наследование. Механизм наследования позволяет закрепить и передать черты, присущие объектам от одного поколения к другому. Наследование представляется диаграммой классов, которая изображается как 

UML (универсальный язык моделирования)

Полиморфизм. Позволяет использовать одни и те же функции для решения разных задач. Поддержка полиморфизма в ООП осуществляется через виртуальные функции и переопределение операторов.

Int a=5, b=6, c;

C=a+b;

Chsr f[30];

Char ff[30];

Char fff[60];

Gets(f): //иванов

Gets(ff): //иван

Fff=f+ff /// ОШИБКА!!!!

Объекты и классы

Базовыми блоками объектно-ориентированной программы являются объекты и классы. Объект можно представить как что то ощущаемое или воображаемое и имеющее определенное поведение. Объект – это объект окружающей нас реальности, т.е. существует во времени и пространстве. Объект имеет состояние, поведение и может быть однозначно идентифицирован, т.е. имеет уникальное имя. Класс – это множество объектов, имеющих общую структуру и общее поведение

Int x;

Int x, y, z; // обьекты, имеющие одинаковую структуру, т.е. объекты одного класса.

Состояние объекта объединяет все его поля данных (статический компонент) и текущее значение каждого из этих данных (динамический компонент). Поведение объекта определяет, как объект изменяет свое состояние и взаимодействие с другими объектами. Идентификация объекта – это свойство, которое позволяет отличать один объект от других того же либо других классов.

Язык программирования называется объектно-ориентированным, если:

1. Поддерживается механизм наследования
2. Объекты имеют связанные с ними типы (классы)
3. Поддерживаются абстрактные типы данных (объекты с определенным интерфейсом и скрытым внутренним состоянием)

Программа будет объектно-ориентированной только при соблюдении всех трех требований

Ключевые слова языка С++

Class, New, Delete, Private, Public, Protected, this, inline, try, catch, throw, template, friend, virtual, operator.

Операции

:: - расширение контекста, либо принадлежности

New, delete (выделение/освобождение памяти)

С++ поддерживает все типы данных, определенные в языке Си, кроме того вводит новые данные – классы и ссылки. Ссылки расширяют и упрощают используемую в языке Си передачу аргументу функции.

Простейший ввод/вывод.

В С++ ввод и вывод данных производится потоками байт. Поток – это логическое устройство, которое выдает и принимает информацию от пользователя и связано с физическими устройствами ввода/вывода. При операциях ввода байты направляются от устройства в основную память, в операциях вывода же – наоборот, из памяти на устройство. Имеются 4 потока (связанных с ними объекта), обеспечивающих ввод и вывод информации и определенных в заголовочном файле

#include <iostream>

Using namespace std;

Cin – поток стандартного ввода с клавиатуры

Cout – поток стандартного вывода (экран)

Cerr - поток стандартной ошибки

Clog – буферизированный поток стандартных ошибок

В заголовочном файде iostream перегружены 2 оператора побитового сдвига << (поместить в выходной поток) и >>(поместить во входной поток). Общий формат записи CIN следующий

Cin>>имя\_переменной;

Cout<<имя\_переменной<<///

Cout не выполняет автоматический перевод курсора на новую строку при выдаче информации. Для этого используется \n либо endl

Передача параметру функции по ссылке.

В общем виде, ссылка объявляется следующим образом

Тип\_данных

Ссылка вводит другое имя или синоним для объекта. В результате можно передать ссылку в другую функцию в виде другого параметра. Доступ к переменной может осуществляться через ее имя в вызывающей программе, или через имя-синоним в вызываемой программе. После завершения вызываемой программы, имя – синоним уничтожается, однако измененное значение переменной в вызывающей программе сохраняется.

Операторы для динамического выделения и освобождения памяти

Используются встроенные операторы new delete. В общем виде, оператор new можно записать

New имя типа[(инициализирующее значение)]

В простейшем виде, его можно записать

New имя\_типа;

New (имя\_типа);

New имя\_типа(инициализирующее значение);

New имя\_типа[инициализирующее значение];

Оператора new возвращает указатель на объект типа имя\_типа, для которого выполняется выделение памяти.

Int \*mas;

Mas=new int;

Mas=new(int);

Mas=new int(5); /// придание инту значения 5!!!

Mas=new int[5] /// выделение 5 ячеек под 5 интов!!!

Int n;

Int m;

Int \*mas;

Mas=new int[n\*m] ///странная щтука. Вроде матрица, но записана в строку…..

Как получить доступ к элементу?

Mas+(i\*m)+j;

For(int i=0; i<n; i++)

For(int j=0; j<m; j++)

Cin>>mas[(i\*m)+j]

Int n, m;

Int \*\*mas;

Mas=new int\*[n];

For(int i=0; i<n; i++)

Mas[i] = new int [m]

For (int i=0; i<n; i++)

For (int j=0; j<m; j++)

Cin>>mas[i][j];

Удаление

For(int i=0; i<n; i++)

Delete []mas[i];

Delete [] mas;

Дан одномерный массив на n элементов, память отводится динамически, реализовать функции

1. Выделение памяти под массив
2. Ввод данных в массив
3. Вывод массива на экран
4. Записать функцию пересечения массивов и обьединения массивов

Т= 1 2 3 4 5

И =1 2 4 5 6 7 8

Пересечение т и =1 2 4 5

Обьединение массивов

1 2 3 4 5 6 7 8

Передача аргументов в функцию по умолчанию

Передача аргументов в функцию по умолчанию означает что при описании функции, аргументам присваиваются некоторые значения, и при вызове данной функции число фактических параметров может быть меньше числа формальных параметров. В этом случае принимается умалчиваемое значение соотствующего параметра.

Int sm (int i1, int i2, int i3=0, int i4=0)

{ return i1+i2+i3+i4

}

Int main ()

{ cout << sm (1,2);

Cout << sm (1,2,8)

Cout << sm (1,2,3,4)

}

Классы и объекты.

Класс – это абстракция либо тип, определяемый пользователем.

Struct point

{

Int x,y;

}

Void setpixel (int, int);

Void getpixel (int\*, int\*);

Сравнить

Struct point /// компоненты данные

{

Int x,y;

Void setpixel (int, int); /// компоненты функции

Void getpixel (int\*, int\*);

}

Чтобы обьявить объект данного типа, нужно

Имя\_классса имя\_обьекта

Int a=50, b=100;

Point my\_point;//объявили объект

Point \*pointer //указатель на объект

My\_point.set\_pixel(a,b);

Pointer -> get\_pixel(&a,&b);

Синтаксис описания функции, принадлежащей структуре, имеет следующий вид

Void point::set\_pixel(int a, int b)

Гдe point – класс функции

Set\_pixel – имя функции

Int – тип аргумента

A,b – имя аргумента

:: - оператор пренадлежности

В языке С++ класс можно объявить с помощью следующих ключевых слов

Struct (структура), class (класс) и union (объединение). Компоненты класса имеют ограничение на доступ. Эти ограничения определяются следующими ключевыми словами: private, protected, public. Для ключевого слова class по умолчанию ставится атрибут доступа private. Для struct – public, union – public. Ограничения на доступ можно изменить, записав перед этим компонентом ключевое слово доступа и поставить двоеточие. В общем виде синтаксис атрибутов доступа следующий:

Class имя\_класса

{ компоненты данных ///если хотим изменить атрибуты для некоторых класссов, то:

Public:

Void set\_pixel();

Protected:

Компоненты данных;

};

Смысл атрибутов доступа следующий. Член классов с атрибутом private может использоваться методами собственного класса, а также функциями-друзьями этого же класса. Protected – почти то же, что и private, но дополнительный член класса может использоваться методами и функциями-друзьями этого класса.

Public – может использоваться любыми функциями программы. Защита на доступ снимается.

Class string

{

Char str[25];

Public:

Void setstring (char\*);

Void display\_string ();

Char\* return\_string();

};

Void string ::set\_string (char\*s)

{

Strcpy (str, s);

}

Void string::display\_string();

{

cout<<str<<endl;

}

Char\* string :: return\_string()

{return str;

}

Void main()

{

string str1;

str1.set\_string(“Minsk”);

str1.display\_string();

cout<<str1.return\_string();

}

Использование функций для установки начальных значений данных объекта, часто приводит к ошибкам. В С++ для этой цели используется функция, которая инициализирует данные объекта в процессе объявления объекта. Эта функция называется конструктором.

Конструктор имеет следующие отличительные особенности:

1. Всегда имеет то же самое имя, что и класс, в котором он определен.
2. Никогда не должен возвращать значение, даже типу void
3. Всегда выполняется при создании нового объекта
4. Может определяться пользователем, или создаваться по умолчанию.
5. Не может быть вызван явно из пределов программы, т.е. он не может быть вызван как обычный метод. Он вызывается компилятором ЯВНО при создании объекта, и НЕЯВНО при выполнении оператора new для выделении памяти объекта

Класс может иметь несколько конструкторов, что позволяет использовать несколько способов для инициализации объекта

Деструктор. Противоположные действия, по отношению к действиям конструктора, выполняют деструкторы (разрушители), которые уничтожают объект. Основные правила использования деструкторов:

1. Имя деструктора совпадает с именем класса, в котором он объявлен, с префиксом ~
2. Деструктор не возвращает значения. Даже типу void
3. Деструктор не имеет аргументов.
4. В классе может быть только один деструктор
5. Деструктор может быть виртуальным, т.е. виртуальной функцией.
6. Если деструктор отсутствует в описании класса, то он автоматически генерируется компилятором
7. Деструктор может быть вызван явно или неявно. Неявный вызов деструктора связан с прекращением существования объекта. Явное уничтожение объекта производится оператором delete, который вызывает деструктор

#include <iomanip>

#include <iostream>

Using namespace std;

Class massiv

{ int n;

Int \*mas;

Public:

Massiv (int a=0);

~massiv ();

Void vvod();

Void show();

Massiv sum (massiv&ob, massiv&ob1);

};

Massiv::massiv (int a)

{

Mas=new int[a];

///////

Massiv massiv::sum (Massiv &ob, Massiv &ob1)

{

Massiv out (ob.n);

For (int i=0; i<ob.n; i++)

{

Out.mas [i] = ob.mas[i]+ob1.mas[i];

}

Return out;

}

Void massiv::vvod ()

{

For (int i=0; i<n; i++)

Cin>>mas[i];

}

Void massiv::show()

{

For (int i=0; i<n; i++)

Cout<<’\t’<<mas[i];

}

Int main()

{

Int n;

Cout<<”vvedite n”;

Cin>>n;

Massiv m1=n, m2=n, m3=n;

Cout>>”vvedite m1: “;

M1.vvod();

M2.vvod();

M3=m1.sum(m1,m2);

M3.show();

}

Данная программа не будет работать корректно, т.к. необходимо вводить компилятор присваивания, либо конструктор копирования

Если в функцию аргументы передаются по значению, то в стеке данной функции создается копия объекта. В копии объекта содержится указатель на массив. Когда функция sum заканчивает работу, то по умолчанию вызывается деструктор объектов ob и ob1, который уничтожает копию объекта, а соответственно уничтожется адрес массива. А далее, все что бы мы ни делали, приводит к ошибке, т.к. адрес массива уничтожен.

Для того, чтобы этого не было, необходим конструктор копирования, который создаст локальную копию аргумента ob.

Конструктор копирования (копирующий конструктор)

Общий вид конструктора копирования следующий:

Имя\_класса (const имя\_класса&)

Вызов конструктора копирования осуществляется:

1. Если в функцию передаются объекты по значению.
2. При возвращении локального объекта из функции в вызывающую программу, т.е. оператор return вызывает конструктор копирования.

Если в программе необходимо получить копию объекта

Massiv m1(n)

M1.vvod();

Massiv m2(m1);

Massiv (const massiv& ob)

Massiv::massiv(const massiv& ob)

{

N=ob.n;

Massiv = new int[n];

For (int i=o; i<n; i++)

Mas[i]=ob.mas[i]

}

Указатель this. Каждый объект имеет свой скрытый от пользователя указатель. Т.е. когда создается объект, под него отводится память. В памяти есть специальное поле, содержащее скрытый указатель, который адресует начало выделенной под объект памяти. Получить значение указателя в компонентах класса указателя можно с помощью слова this. Его не надо дополнительно объявлять. Он передается как скрытый аргумент во все нестатические (т.е. не имеющие спецификатора static) компоненты функции своего объекта. This – локальная переменная, недоступная за пределами объекта. Разрешается обращаться this; либо \*this;

Конструктор explicit. В С++ для конструктора с 1 аргументом могут автоматически выполняться неявные преобразования. В результате этого тип, получаемый конструктором, преобразуется в объект класса, для которого определен данный конструктор.

Пример

Class array

{

Int size;

Int \*ms;

Public:

(explicit) Array (int n=1); ///если ввести эксплисит, будет выдавать ошибку.

~array ();

Void print (const array&);

};

Array::array (int n)

{

Size = n;

Ms = new int [size];

For (int i=0; i<size; i++)

Ms[i] = 0;

}

Array::~array()

{

Delete [] ms;

}

Void array::print(const array& obj)

{ for (int i=0; i<size; i++)

Cout<<setw(4)<<obj.ms[i];

Void main ()

{

Array obj(10); //// int r; cout<<”vvedite r”<<endl; cin>>r; array obj(r);

Obj.print (obj);

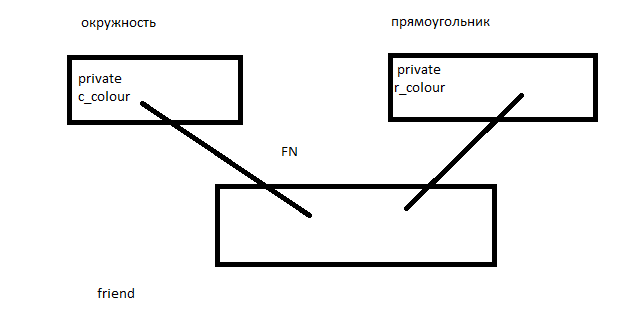
Obj.print(r);

}

При вызове функции, компилятором не находится функция print с аргументом int, поэтому выполняется проверка на наличие в классе array конструктора, способного выполнить преобразование в объект класса array. Так как в классе array такой конструктор есть, то этот конструктор и производит преобразование данного типа в тип array. В некоторых случаях такие преообразования нежелательны, поэтому в C++ есть ключевое слово explicit, которое вводит запрещение неявных преобразований.

ДРУЖЕСТВЕННЫЕ ФУНКЦИИ

В некоторых случаях может возникнуть необходимость доступа из одной функции к локальным компонентам разных классов.



Функция объявленная со спецификатором friend является дружественной. Дружественная функция не является членом класса, хоть она и объявлена в классе, следовательно, дружественной функции недоступен this. Дружественные функции имеют доступ ко всем данным класса. В целом, функция со спецификатором friend является глобальной функцией.

Class my\_class2; //предварительное объявление класса

Class my\_class1;

{

Int a;

Public:

My\_class1(int n)

{ a = n;}

//объявляется дружественная функция

Friend void fn (my\_class1& ob, my\_class2& ob1);

}

//полное объявление класса

Class my\_class2

{

Int a;

Public:

My\_class2 (int n)

{ a=n; }

Friend void fn(my\_class1& ob, my\_class2& ob1)

Void fn(my\_class& ob, myclass2& ob1)

{

If (ob.a==ob1.a)

Cout<<”sovpadayut”<<endl;

Else

Cout<<”ne sovpadayut”<<endl;

}

Int main ()

{my\_class1 mc(100);

My\_class2 mc(200);

Fn (mc1, mc2);

}

Компонент функции одного класса может быть объявлен friend для другого класса

Class x

{ ////

Void fun();

};

Class y

{

Friend void x::fun();

};

В приведенном примере, функция fun имеет доступ к локальным компонентам класса y. Данная запись говорит о том, что функция fun принадлежит классу Х, а спецификатор friend разрешает доступ к локальным компонентам класса y/

Class nalogi;

Class work

{

Char s[20]; //фамилия работника

Int zp;// зарплата работника

Public:

Work () {}

~work () {}

Float raschet (nalogi& ob);

Void inpt ()

{ cout<< “vvedite FIO I oplatu”<<endl;

Cin >> s >> zp;

}

};

Class nalogi

{

Float pd; //подоходный налог

Float st; // соц страхование

Friend float work::raschet (nalogi&);

Public:

Nalogi (float f1, float f2)

{pd = f1; st = f2;}

`nalogi () {}

Float work::raschet (nalogi n1)

{cout << s << setw (6) << zp << endl;

Cout << n1.pd << setw (8) <<n1.st << endl;

Return zp\*n1.pd(100+zp\*n1slkdfj;askdjfhws;dv

}

Встроенные функции, спецификатор inline

В ряде случаев, в качесте компонента класса используются достаточно простые функции. Для таких функций целесообразно поместить код функции в место выполнения перехода к функции. В этом случае при выполнении функции (обращения к ней или ее вызова) выполняется сразу ее код. Функции с такими свойствами являются встроенными. Иначе говоря, если описание функции включается в клласс, такая функция называется встроенной.

Class stroke

{char str[100];

Public:

Int size ()

Если функция объявлена в классе, а алгоритм – за ее пределами, то

Class stroke

{ char str[100];

Public:

Int size();

};

(inline) Int stroke::size()/// нужно ввести inline

{

For (int i=0;\*(str+1); i++);

Return I;

};

Спецификация инлайн может быть проигнорирована компилятором, поскольку введение сторонних функций может быть нежелательным.

Перегрузка операторов

Программы на С++ используют некоторые ранее определенные типы данных, например int, double, float.

Int a;

Float b;

В языке определено множество операций над указанными типами + - \* / % и т.д.

Каждый оператор можно применить к операндам определенного типа. Например

Int a, b, c;

A=6;

B=2;

C=a+b; - возможно.

Float d;

D = 5,5;

C=a+d;// выдаст значение 11

C=a%b;

C=a%d//выдаст ошибку

Операторы языка можно определить к тем объектам, для которых они были определены. Например, в С++ нет операций для работы с матрицами, однако, эти операции модно определить через классы. Например – пусть нам задано некоторое множество

А={a1, a2, a3};

B={a3, a4, a5};

Объединение и пересечение множеств

& /

A&b={a3};

A/B = {a1, a2, a3, a4, a5};

В языке С++ не поддерживаются эти операции для работы с множествами, однако можно объявить класс SET, и далее определить операции над этим классом, выразив их с помощью знаков операторов, которые уже есть в языке С++. В результате эти операции «и» и «или» можно будет использовать как и ранее, а также оснастить их дополнительными функциями для работы с множествами. Как определить, какую функцию должен выполнять этот оператор? Нужно посмотреть на типы операндов. Если операнды – объекты целого типа, то нужно выполнить операцию побитового «и» или побитового «или». Если операнды – объекты типа SET, то необходимо выполнить объединение или пересечение множеств.

Функция «оператор» operator. Может быть использована для расширения области приложения всех известных операторов, кроме . .\* :: ?: sizeof

Для перегрузки оператора, разрабатываются функции, являющиеся компонентами класса, либо friend функциями того класса, для которого они используются

Общая форма записи функции operator, являющейся компонентом класса, имеет вид:

Типвозвращаемогозначения имя\_класса :: operator #(список\_аргументов)

{

Действие, выполняемые применительно к классу

}

Вместо знака # ставится значок перегружаемого оператора. Например +, -, ?: и т.д.

Если перегружается бинарный оператор, то в

Объявление и вызов функции operator осуществляется так же, как и вызов любой другой функции, но только разрешается использовать ее сокращенную форму вызова. Например

Вместо operator # (a, b)

Пишем a # b;

Запишем перегрузку оператора & для операции пересечения множеств.

Class set

{

Char \*str;

Public:

Set (char \*s = “”)

{str = new char [strlen(s) +1];

Strcpy (str, s);

}

Set (const set& ob)

{

Str = new char [strlen (ob.str)+1];

Strcpy (str, ob.str);

}

~set() {delete [] str;}

//пусть она нам возвращает строку

Char\* operator& (set);

};

Char\* set :: operator & (set S)

{

Int t=o, l=0;

While (str[t++] != ‘\0’); //// можно еще записать while (this->str[t++] != ‘\0’);

Char \*s1=new char[t];

For (int j=0; str[j] != ‘\0’; j++)

For(int k=0; s.str[k] != ‘\0’; k++)

If(str[j] == s.str [k])

{

S1[l]=str[j];

L++;

Break;

}

S1[l]=’\0’;

Return s1;

}

Void main(void)

{

Shar ss[80];

Char \*ss1;

Cout<<”vvedite 1 stroku”<<endl;

Cin>>ss;// d gfvznm xfhf ddjlbncz lj gthdjuj ghj,tk

Set S1(ss);//передаем ему значение строки SS

Cout<<”vvedite vtoruiy stroke”<<endl;

Cin>>ss;

Set s2(ss);

Ss1=s1&s2; //либо ss1=operator&(s1, s2);

Cout<<”rezultat “<<ss1<<endl;

}

Перегрузка операции присваивания.

Class String

{

Char \*str;

Public:

String (char \*s=””)

{

Str = new char [strlen(s)+1;

Strcpy (str, S);

}

~string() {delete [] str;}

};

Int main()

{

Char ss[50];

Cout<”vvod stroke”;

Cin>>ss;

String obj(ss);

String obj1;

Obj1 = obj;

Return 0;

После этой строки будут ошибки. Одна из них – в объекте obj1.str будет заменена на obj.str, т.е внутри obj1 и obj будут ОДИН И ТОТ ЖЕ str., и после деструктор будет дважды удалять одну и ту же память, что есть ошибка. Чтоб избежать данной ошибки, необходимо перегрузить оператор присваивания. Перегруженный оператор присваивания должен выполнять след. Действия:

1. Освободить динамическую память объекта obj1 (если она была выделена)
2. Выделить динамическую память под данные объекта obj1 в соответствии с размерами компонентов obj
3. Осуществить копирование данных объекта oobj в данные объекта obj1

Перегруженный оператор присваивания для данного класса будет иметь вид:

String & operator = (const string& ob)

{

If(this != &ob)

{

Delete []str;

Str = new char[strlen(ob.str)+1];

Strcpy(str, ob.str);

}

Else return \*this;

}

}

После данных исправлений, ошибки не будет.

/////////////////////////////////////////////////

Пример

Class massiv

{float \*f;

Int n;

Public:

Massiv (int i=0)

{

N=I;

F = new float [n];

}

~massiv

{delete [] f;}

Void vvod ();

Void show ();

Float operator [] (int i);

};

Void massiv :: vvod ()

{

For (int i=0: i<n; i++)

Cin >> f[i];

}

Void Massiv :: show ()

{

For (int i=0; i<n; i++)

Cout << setw [4] << f[i];

}

Cout << endl;

Float &massiv :: operator [] (int i) //смотри сноску

{

If ( i<0 || (или) i>n)

Return 0;

Else return f[i];

}

}

Int main ()

{

Int r;

Int index;

Double rez;

Cout << “vvod razmera massiva” << endl;

Cin >> r;

Massiv obj(r);

Cout << “vvedite massiv” << endl;

Obj.vvod();

Cout << “massiv “ << endl;

Obj.show()

Cout << “wwedite index” << endl;

Cin >> index;

Rez = obj[index]; /// вызов перегрузки []

}

Данная программа при небольшой можификации при небольшой модификации может использовать оператор [] как слева, так и справа от знака присваивания. Для этого необходимо, чтоб оператор [] возвращала не элемент, а ссылку на него.

Перегрузка оператора ()

В общем виде, этот оператор имеет вид:

Тип\_возвращаемого\_результата имя\_класса :: operator () (список аргументов)

{тело функции}

Пример

Class matr

{ int a,b;

Int \*\*m;

Public:

Matr (int i=0, int j=0)

{

A=I; b=j;

M=new int \*[a];

For (int i=0; i<a; i++)

M[i] = new int [b];

}

~matr ()

{ for (int i=0; i<a; i++)

Delete []m[i];

Delete []m;

}

Void vvod ();

Void show ();

Int operator () (int I, int j);

};

Vvod matr::vvod()

{

For (int i=0; i<b; i++)

For (int j=0; j<b; j++)

Cin >> m [i][j];

}

Void matr::show()

{

For (int i=0; i<a; i++)

Cout << endl:

For (int j=0; j<b; j++)

Cout << setw (4) << m[i][j]

}

Cout << endl;

Int matr::operator () (int i, int j)

{

If (i<0 || i>=a || j<0 || j>= b)

Return m[0][0]; /// можно написать return 0

Else return m[i][j];

}

Int main ()

{

Int m, n;

Int I, j;

Cout << “vvod razmerov \*m i \*n” <<endl;

Cin >> m >> n;

Matr mt(n,m);

Cout << “vvedite massiv” << endl;

Mt.vvod ();

Cout << “massiv: “ << endl;

Mt.show();

Cout << “vvod indexov I I J” << endl;

Cin >> I >> j;

Rez = mt (I,j);

}

Return 0;

Перегрузка унарного оператора

При перегрузке унарной операции, функция operator не имеет параметров.

Когда перегружается оператор ++ в виде ++i, то вызывается функция operator в виде operator ++(). Если же перегружается ++ постфиксный, i++, то вызывается operator ++(int);

Пример

Class dek\_koord

{

Int x,y;

Public:

Dek\_koord(int I, int j)

{x=I; y=j}

Void operator ++();

Void operator ++(int);

Dek\_koord operator = (const dek\_koord);

Void see;

};

Void dek\_koord::operator ++()

{x++;}

Void dek\_koord::operator ++(int)

{y++;}

Dek\_koord dek\_koord::operator = (const dek\_koord ob)

{

X=ob.x; y=ob.y;

Return \*this;

}

Int main ()

{

Dek\_koord A(1,2), B, C; ///// dek koord (int i=0, j=0)

A.see ();

A++;

A.see ();

++A;

A.see();

B=C=A;

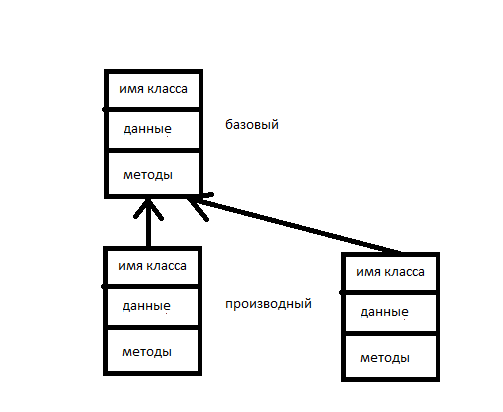
B.see();

C.see();

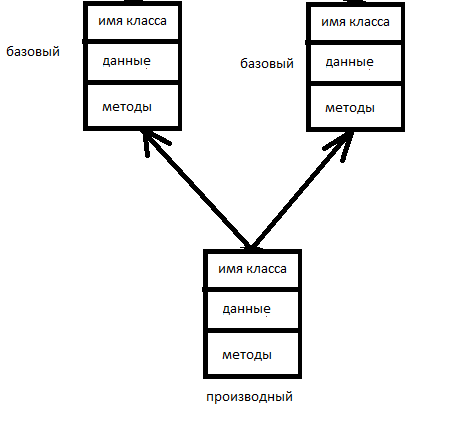
} return 1;

Наследование

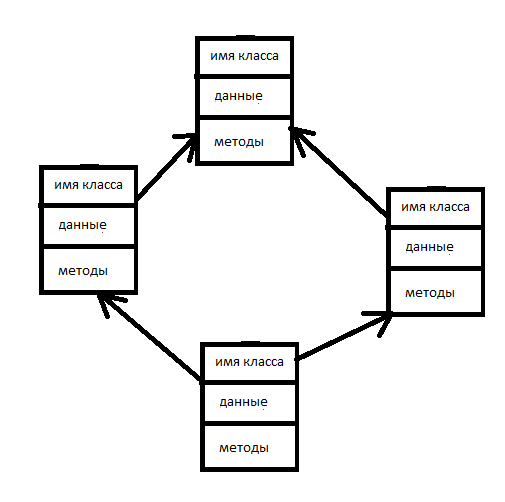
Наследование – механизм получения нового класса из существующего. Существующий класс может быть дополнен или изменен для создания произвольного класса. При создании нового класса, вместо написания полостью новых данных и функций, можно указать, что новый класс должен наследовать данные и функции ранее определенного базового класса. Этот новый класс называется ПРОИЗВОДНЫМ. Каждый производный класс сам является кандидатом на роль базового класса для будущих производных классов. при простом наследовании, производный класс порождается одним базовым классом.



При множественном наследовании, производный класс порождается несколькими базовыми.



Виртуальное наследование



Производный класс обычно добавляет свои данные и функции, так что производный класс, в общем случае, больше своего базового класса.

Шаблон объявления производного класса определяется следующим образом:

Ключ\_класса имя\_производного\_класса:необязательный\_модификатор\_доступа имя\_базового\_класса

{

тело производного класса

};

Class Point

{

Int x, y;

Public:

/

/

/

};

Ключ\_класса – class, struct

необязательный\_модификатор\_доступа – private, public; (если class, а модификатор не указан, то по умолчанию private. если struct, и не указан модификатор – по умолчанию public)

class LINE: public Point

{

\

\

\

};

Модификатор прав доступа используется для изменения прав доступа к наследуемым элементам базового класса в соответствии со следующим правилами:

|  |  |  |
| --- | --- | --- |
| Ограничение на доступ в базовом классе | Модификатор прав доступа (с каким наследуется) | Ограничения на доступ в производном классе |
| private | private | Нет доступа |
| protected | private | private |
| public | private | private |
| private | public | Нет доступа |
| protected | public | protected |
| public | public | public |

Пример

Class A

{

Int a1;

Public:

Int a2;

Void f1(void):

};

Class B: public A

{

Int b1;

Public:

Void f2(void);

};

Void A::f1()

{

A1 = 10;

A2 = 20;

}

Void B::f2()

{

A1=1; ///ошибка наследования

B1 = 10;

A2 = 100;

}

Void main ()

{

A a\_ob1; a\_ob1.a2+=1;

B b\_ob1; b\_ob1.b1+=2; //ошибка наследования

}

Int b1;

Public:

Void f2();

Int set()

{return b1;}

Если базовый класс имеет конструктор с параметрами, то любой производный класс должен иметь конструктор с параметрами.

Множественное наследование

Общий синтаксис описания конструктора производного класса представляется в следующем виде:

Имя\_конструктора\_производного\_класса (список аргументов производного класса): имя\_базового\_класса1(список аргументов базового класса 1), ……….., имя)базового\_кллассаN(список аргументов базового класса N)

{тело конструктора производного класса}

Class BASE1

{

Int a;

Protected:

Int b;

Public:

BASE1(int x, int y);

~BASE1();

Void show1()

{

Cout << “a=” << a << endl;

Cout << “b=” << b << endl;

}

};

Class base2

{

Protected:

Int c;

Public:

Base2 (int x);

~base2();

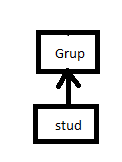
Void show2()

{

Cout << “c=” << c << endl;

}

};



Структура наследования

Class grup

{

Protected:

Char \*fak;

Long gr;

Public:

Grup (char \*FAK = “”, long GR = 0)

{

fak = new char[strlen(FAK)+1];

strcpy(fak, FAK);

gr = GR;

}

~grup () {delete [] fak;}

Virtual void see();

};

Class stud: public grup

{

Char \*fam;

Int OC[4];

Public:

Stud (char \*FAK = “”, long GR = 0, char \*FAM = “”, int OC[]) // OC надо зануллить, мб // ={0})//

Grup (FAK, GR)

{

Fam = new char[strlen (fam) +1];

Strcpy (fam, FAM);

For (int i=0; i<4; i++)

oc[i] = OC[i];

}

~stud()

{

delete [] fam;

}

Virtual void see();

};

Void grup::see()

{

Cout << setw(10) <<fak << setw(8) << gr << endl;

}

Void stud::see()

{

Grup :: see();

Cout << setw(10) << fam << “ “;

For (int i=0; i<4; cout << oc[i++]<< “ “);

Cout << endl;

}

Int main ()

{

Grup gr1(“FAK 1”; 123456); int OC[] = {8, 8, 9, 9};

Stud st(“FAK 1”, 123456, “ivanov”, OC);

Gr1.see ();

St.see ();

Obj = &st;

Obj -> see();

Stud st1 [4];

For (int i=1; i<4; i++)

{

Cout << “vvod nazvaniya fakulteta”;

Cin.getline (fk, 40);

Cout << “vvod nomera gruppi”;

Cin >> gr;

Cout << “vvod familii”;

Flush all();

Cin.getline (fm, 40);

For (int i=0; i<4; i++)

Cin >> OC[i];

Stud obj (fk, gr, (m, OC);

St1 [i] = obj;

}

}

Если функция была объявлена как виртуальная в некотором классе (базовом), то она остается виртуальной через все уровни иерархии, которые она прошла

Основные свойства и правила использования виртуальных функций:

1. Виртуальный механизм поддерживает полиморфизм на этапе выполнения программы. Это значит, что требуемая версия функции выбирается на этапе выполнения программы, а не компиляции.
2. Класс, содержащий хотя бы одну виртуальную функцию, называется полиморфным.
3. Виртуальные функции можно объявить только в классах, которые обьявляются с помощью слова class, а также с помощью слова struct
4. Виртуальными функциями могут быть только нестатические функции, т.е. без спецификатора static

Для вызова виртуальной функции, требуется больше времени, чем для невиртуальной, при этом требуется дополнительная память для хранения виртуальной таблицы.

При использовании полного имени при вызове некоторой виртуальной функции, виртуальный механизм НЕ ПРИМЕНЯЕТСЯ. (игнорируется)

Абстрактные классы

Базовый класс обычно содержит ряд виртуальных функций, которые часто фиктивны и имеют пустое тело, т.е. не имеют реализации. Эти функции существуют как некоторые абстракции, и конкретное значение им придается в классах. Такие функции называются чисто виртуальными. Общая форма записи чисто виртуальной функции имеет следующий вид:

Virtual прототип\_функиции = 0;

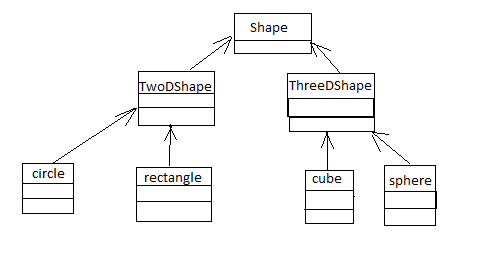
Класс, который имеет хотя бы одну чисто виртуальную функцию, называется абстрактным. Если класс является производным класса с чисто виртуальной функцией, и эта функция в нем не описана, то такая функция остается чисто виртуальной и в этом производном классе, следовательно, такой производный класс называется абстрактным.

Хотя иерархия классов не требует обязательного включения в нее каких – либо абстрактных классов, однако программы, использующие объектно-ориентированное программирование имеют иерархию, порожденную абстрактным базовым классом.

Объект абстрактного класса создать невозможно.

Класс можно сделать абстрактным, даже если все его функции определены (описаны). Это можно сделать, например, чтобы быть уверенным, что объект этого класса создан не будет. Для этих целей, объявляется чистым виртуальный деструктор.

Абстрактные классы могут составлять несколько уровней иерархии. В качестве примера, приведем иерархию форм



Class Shape

{

Public

Virtual void print() = 0;

Virtual void area () = 0;

Virtual void volume() = 0;

Shape ()

{cout << “const Shape” << endl;}

~shape()

{cout << “destr shape” << endl:}

};

Class TwoDShape:public shape

{

Pprotected

Int r;

Public:

Twodshape (int a)

{

Cout << “const twodshape” << endl;

R=a;

}

~twodshape()

{cout << “destr twodshape” << endl;

Virtual void print() = 0;

Virtual void area() = 0;

Virtual void volume()

{cout << “nel’za vichislit ob’em ploskoi figure” << endl;

}

};

Class threedshape : public shape

{

Protected:

Int h;

Public:

Threedshape (int a)

{

cout <<const threedshape << endl;

h=a;

}

~threedshape()

{cout << “destr threedshape” << endl;

}

Virtual void print() = 0;

Virtual void volume() = 0;

Virtual void area ()

{cout << “nel’za vichislit ploschad’ 3d figure” << endl;

}

};

Class circle : public twodshape

{

Public:

Circle(int a):twodshape(a)

{cout << “const circle” << endl;}

Void print()

{cout << “circle” << endl;}

Void area()

{cout << “area circle =” << 3.14\*2\*r << endl;}

};

Class rectangle : public twodshape

{

Int n1;

Public:

Rectangle (int a, int b):twodshape (a)

{h1 = b;

Cout << “const rectangle” << endl;

}

~rectangle

{cout << “destr rectangle” << endl;}

Void area()

{cout << “area rectangle is: “ << r\*h1 << endl;}

};

///аналогично прописываются в тридшэйп куб и сфера

Int main ()

{

Shape ob; // ошибка!!!!

Shape \*ptr[4];

Int a,b;

Cout << “vvod razmerov” << endl;

Cin >> a >> b;

Ptr[o] = new circle (a);

Ptr[1] = new rectangle (a,b);

Ptr[2] = new cube (a);

Ptr{3} = new sphere(a);

For (int i=0; i<4; i++)

{

Ptr[i] -> print();

Ptr[i] -> area();

Ptr[i] -> volume();

}

For (int i=0; i<4; i++)

Delete ptr[i];

}

Виртуальные деструкторы

Виртуальные деструкторы необходимы при использовании указателей на базовый класс при выделении памяти под динамически создаваемые объекты производных классов. Это обусловлено тем, что если объект удаляется явно, т.е. используется операция delete, то вызывается деструктор только класса, совпадающего с типом указателя на уничтожаемый объект. Если объявить деструктор базового класса виртуальным, то все деструкторы производных классов становятся также виртуальными. При этом если будет происходить явное уничтожение объекта производного класса, то вначале вызывается деструктор производного класса, а затем – по иерархии до деструктора базового класса.

Основные правила использования виртуальных деструкторов

Деструктор может быть виртуальным, а конструктор нет.

Виртуальный деструктор позволяет вызвать «правильнй» деструктор через указатель на базовый класс, даже если базовый указатель содержит значение, указывающее на производный класс. Если базовый класс содержит виртуальный деструктор, то деструктор производного класса тоже будет виртуальным.

Организация ввода-вывода

Система ввода/вывода в С++ основывается на понятии потока. Поток в С++ - это абстрактное понятие, относящееся к переносу информации от источника информации к приемнику. В С++ реализованы 2 иерархии классов, обеспечивающих операцию ввода/вывода, базовыми операциями которого являются ios и streambuf.

В С++ используются способы перегрузки оператора вводи и вывода. Для обеспечения работы с потоками ввода/вывода, необходимо включение файла iostream

Общая форма перегрузки оператора ввода/вывода имеет вид

Istream& operator >> (istream & поток, имя\_кл & объект) //ввод

Ostream& operator << (ostream& поток, const имя\_кл& объект); //вывод

Class cls

{

Char c;

Int I;

Public:

Cls(char k=’ ‘, int a=0)

{c=k; i=a;}

~cls(){}

Friend istream& operator>>(istream& in, cls& ob);

Friend ostream& operator<<(ostream& out, cls& ob);

}

Istream& operator >> (istream& in, cls& ob)

{

In >> ob.c >> ob.i;

Return in;

}

Ostream& operator << (ostream& out, const cls& ob)

{ out << setw(4) << ob.c << setw(4) ob.i << endl;

}

Int main()

{

Cls s;

Cout << “vvedite simvol I chislo” << endl;

Cin >> ss; //a 10

Cout << ss //

} wwww

Put()

Ostream::put();

Char c=’a’;

Cout.put(c);

Cout.put(c). put(‘b’).put(’\u’);

Get90

Istream::get()

EOF

Int main()

{

Char c;

Cout << vvod teksta” << endl;

While99C=CIN.GET())!= EOF)

Cout put(c);

}

1. Функция вводится без переменных, вводит из соответствующего потока одиночный символ и возвращает его значение. Если из потока прочитан симфол конца файла, то функция возвращает EOF
2. Функция get используется с одним символьным аргументом. Функция возвращает false при считывании признака конца файла. Иначе – ссылку на объект istream, для которого вызывалась функция get.
3. Функция get принимает 3 параметра: 1 – указатель на строку,куда читается, 2 – максимальное число символов, которые нужно считать, 3 – ограничитель ввода (по умолчанию ‘\n’). Ввод прекращается, когда считано число симолов на 1 меньше максимального, или считан символ ограничитель ввода. Во вводимую строку добавляется ноль-символ.

Символ ограничитель из потока не удаляется. При наличии функции get, он приведет к началу новой строки, будучи неудаленным.

Char s[30];

Cin.get(s, 30, ‘\n’);

Функция getline действует аналогично функции get с тремя параметрами, с тем отличием, что функция get удаляет символ-ограничитель из потока.

Int a;

Istream::getline()

Cin>>a; /// ‘\n’

Cin.getline (s, 30, ‘\n’);

Cin,getline(s, 30);

Шаблоны функций.

Они позволяют по одному фрагменту кода применять шаблонные функции. Например, можно написать шаблон обработки (сортировки) массива, на основе которого будут формироваться отдельные шаблонные функции, которые будут сортировать массивы типов int, float, double и т.д.

Все описания шаблонов функций начинаются с ключевого слова template, за которым следует список формальных параметров шаблона, заключенных в угловые скобки. < >

Каждому формальному параметру должно предшествовать ключевое слово class или typename. Например:

template <class T>

template <typename T>

template <class T, class T1>

формальные параметры в описании шаблона используются для определения типов параметров функции, типа возвращаемого значения функции и типов переменных, объявляемых внутри функции. За данным заголовком следует обычное описание функции. Каждый формальный параметр из списка описания шаблона должен появиться в списке параметров функции хотя бы один раз.

Template <class T>

T add (Ta, T b)

{T sum;

Sum = a+b;

Return sum;

}

Int main ()

{

Int k, d, sm;

Cout << “vvod K I D” << endl;

Cin >> k >> d;

Sm = add (k, d);

Float z, y, f;

Cout << “vvod Z I Y” << endl;

Cin >> z >> y;

F = add (z,y)

}

Когда компилятор обнаруживает в тексте программы вызов функции add, он заменяет t во всей области определения шаблона на тип аргумента функции и генерирует функцию указанного типа.

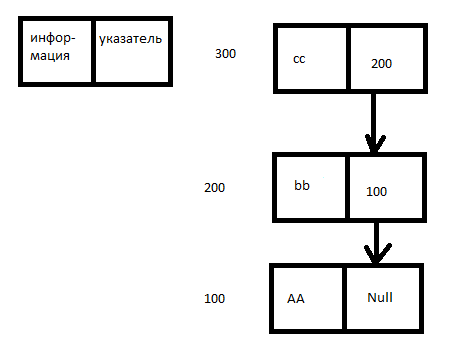
Шаблоны классов

Шаблонный (параметризованный) класс – это шаблон, на основе которого можно строить другие классы. Параметрический полиморфизм позволяет использовать один и тот же код для различных типов данных. Шаблоны – определения класса и шаблоны – определения функции дают возможность многократно использовать корректно код по отношению к различным типам. Спецификация шаблона класса имеет следующий вид:

Template <список параметров>

Class объявление класса

Шаблонный класс-стек.



Chel.h

Class chel

{

int god;

char fio[40];

public:

chel (int a=0, char \*s = “ “)

{

god = a;

strcpy(fio,s);

}

~chel () {}

//перегрузка ввода-вывода

Friend istream& operator >> (istream& in, chel& ob);

Friend ostream& operator << (ostream& out, chel& ob);

Chel& operator = (const chel& ob);

};

//новый файл

Chel.cpp

#include “chel.h”

#include <iomanip>

Using namespace std;

&istream operator >> (istream& in, chel& ob)

{

Cout << “vvedite god rozhdeniya’ << endl;

In >> ob.god;

Cout << “vvedite FIO” << endl;

Flushall();

In.getline (ob.fio, 40);

Return in;

}

//перегрузка вывода

Ostream& operator << (ostream& out, const chel& ob)

{

Out << setw960 << ob.god << setw(20) << ob.fio << endl;

Return out;

}

Chel& chel::operator = (const chel& ob)

{

If(thir != &ob)

{

God = ob.god;

Strcpy (fio, ob.fio);

}

Return \*this;

}

//СТЕЕЕЕЕЕЕЕЕЕЕЕК

Stack.h

Template <class T>

Struct item

{

T i;

Item <T> \*next; // так писать, поскольку он шаблонный

};

Template <class T>

Class stack

{

Item <T> \*head; // указатель на вершину стека

Public:

Stack(); //конструктор

~stack (); //деструктор

Void push (T);

Void pop();

T pop1();

Void show();

Bool empty(); //показывает, пуст ли стек

};

Stack.cpp

#include “stack.h”

Stack <T>::stack()

{ Head = NULL;}

Template <class T>

Stack <T>::~stack()

{delete head;}

Template <class T>

Void stack<T>::push(T value)

{if (head == NULL)

{head = new item <T>;

Head -> I = value:

Head -> next = NULL;

}

Else

{

Item <T> \*newitem = new item <T>;

Newitem -> I = value;

Newitem -> next = head;

Head = newitem;

}

}

Template <class T>

Void stack<T>::pop()

{

If (head == NULL)

Return;

Item<T> \*p = head;

Head = head->next;

Delete p;

}

Template <class T>

T stack <T>::pop1()

{item <T> \*p = head;

Head = head->next;

T item1 = p->I;

Delete p:

Return item1;

}

Template <class T>

Void stack <T>::show()

{item <T> \*p = this->head;

While (p != NULL)

{cout << p->I << endl;}

}

Template <class T>

Bool stack<T>::empty()

{

If (head == NULL)

Return false;

Else

Return true;

}

Класс интерфейс

Interface.h

Class Interface

{

T ob;

Public:

Interface(){}

~interface(){}

Void menu();

Void fun();

};

Interface.cpp

#include “chel.h”

#include “stack.cpp”

#include “Interface.h”

Template <class T>

Void Interface <T>::fun()

{

Stack <T> st;

T value;

Int ch;

Do

{

System (“cls”);

Cout << “1. Add element” << endl;

Cout << “2. Show element” << endl;

Cout << “3. Delete element” << endl;

Cout << “4. Delete all” << endl;

Cout << “5. Quit” << endl;

Cout << “viberite” << endl;

Cin >> ch;

Switch (ch)

{case 1:

Cout << “vvod ob’ecta”

Cin >> value;

St.push (value);

Break;

Case 2^

St.show();

Cout << “press any key”;

Getch();

Break;

Case 3:

St.pop();

Break;

Case 4:

While (st.empty())

St.pop();

Case 5:

Break;

}//close switch

} while (ch != 5);

Template <class T>

Void Interface <T> :: menu()

{

Char ch;

Do {system (“cls”);

Cout << “1. Rabota cint” << endl;

Cout << “2. Rabota s << endl;

Cout << “3. End” << endl;

Ch = getch();

Switch (ch)

{case ‘1’:

Interface <int> obj;

Obj.fun();

Break;

Case ‘2’:

Interface <chel> obj;

Obj.fun();

Break;

Case ‘3’:

Break;

} //switch

} while (ch != ‘3’)

}

Main

#include “Interface.cpp”

Int main ()

{

Interface <int> obj1;

Obj1.menu()

Return 0;

}

Потоки istream u ostream имеют связанные с ними состояния. Состояние потока фиксируется в элементах перемещения, объявленного в классе ios, называемое

enum iostate{

Goodbit;

Failbit;

Eofbit;

Badbit.

Состояние потока принимает (ненулевое) значение goodbit или eofbit, если последняя операция ввода прошла успешно. Если при вводе достигнут признак конца файла – eofbit. Отличия между состояниями failbit и badbit:

В состоянии failbit предполагается, что в потоке происходит ошибка форматирования, но поток не испорчен, и символы в потоке не потеряны. В состоянии badbit данные могут быть испорчены.

Все это значит, что состояние потока характеризуется БИТАМИ.

В классе IOS есть несколько функций, для «проверки» и «установки» потока. Эти функции таковы:

Inline int ios :: bad () const; // проверка на бэдбит

Inline int ios :: eof () const; //

Inline int ios :: fail () const;//

Inline int ios :: good () const; // проверяет, изменился ли хоть один из вышеупомянутых.

Кроме отмеченных функция rdstate

Inline int ios :: rdstate () const // проверка всех битов

Inline void ios :: clear (int \_i = 0) //восстановить состояние потока в первоначальное состояние

Void main()

{

Int x;

Cout << “no error” << endl;

Cout << “cin.rdstate():” << cin.rdstate() << endl;

Cout << “cin.eof():” << cin.eof() << endl;

Cout << “cin.good():” << cin.good() << endl;

Cout << “cin.fail():” << cin.fail() << endl;

Cout << “cin.bad():” << cin.bad() << endl;

Cout << “vvod chisla: “ << endl;

Cin >> x; //нарочно введем не число, а символ, т.е. создадим ошибку

Cout << “error!!!” << endl;

Cout << “cin.rdstate():” << cin.rdstate() << endl;

Cout << “cin.eof():” << cin.eof() << endl;

Cout << “cin.good():” << cin.good() << endl;

Cout << “cin.fail():” << cin.fail() << endl;

Cout << “cin.bad():” << cin.bad() << endl;

Cin.clear(0); // восстановим состояние потока

Cout << “cin.rdstate():” << cin.rdstate() << endl; //должно быть все то же, что и в 1

Cout << “cin.eof():” << cin.eof() << endl;

Cout << “cin.good():” << cin.good() << endl;

Cout << “cin.fail():” << cin.fail() << endl;

Cout << “cin.bad():” << cin.bad() << endl;

} //end of main

Int main()

{

Int I;

Int flag;

Do

{

Flag = 0;

Cout << “vvedite chislo” << endl;

Cin >> I; // намеренно введем ошибку.

Flag = cin.rdstate(); //здесь можно воспользоваться if(!cin.good())

If (flag & ios::failbit)

{

Cout “vvedeno ne chislo” << endl;

Cin.clear();

Flushall();

}; //end of if

}; // end of do

While (flag);

} //end of main

Еще, в классе ios выполнена перегрузка операции НЕ. Т.Е.

Inline int ios::operator ! () const;

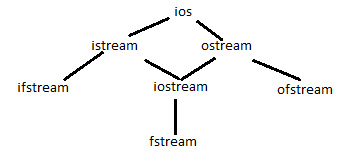
Операция НЕ возвращает ненулевое значение, в случае установки одного из бита badbit или failbit. Иначе возвращает нулевое значение.

В С++ для организации работы с файлами, ииспользуются классы потоков

istream (ввод или чтение)

ofstream (вывод или запись в файл)

fstream (чтение / запись)



В С++ файл открывается путем стыковки его с соответствующим потоком. Для этого, используются конструкторы ifstream и ofstream. Общий вид их следующий:

Ofstream (const char\* Name, int nMode = ios::out, int nPot = filebuf::openprot);

Ifstream (const char\* Name, int nMode = ios::in, int nPot = filebuf::openprot);

Const char\* Name (первый аргумент) – имя файла,единственное обязательное к указанию.

int nMode = ios::in – задает режим для открытия файла и представляет собой битовое «или» следуюющих режимов:

ios::out – поток создается для вывода (записи). Если файл уже существует, то файл уничтожается, на его место записывается новый. В режиме ofstream это можно не указывать. Но для ifstream - нужно указывать для чего открывается файл.

Ios::in – поток создается для ввода или чтения.

Ios::app - запись добавляется в конец файла. Объекты дописываются в конец файла, даже если текущая позиция в этом файле была изменена.

Ios::binary – ввод-вывод будет выполняться в двоичном виде, т.е. это бинарный режим. (по умолчанию указан текстовый)

Ios::trunk – если файл уже существует, то его содержимое уничтожается. Этот режим действует по умолчанию, если файл записан ios::out.

int nPot = filebuf::openprot – используется для установки атрибутов доступа к открываемому файлу. Возможные значения nprot:

filebuf::sh\_compat – режим совместного использования

filebuf::sh\_none – не используется совместно

filebuf::sh\_read – режим совместного использования для чтения

filebuf::sh\_write – режим совместного использования для записи

пример программы:

ofstream ouT(“text” \*/ имя файла/\*); //открываем файл для записи (создаем его)

//можно ofstream ouT(“text”, ios::out);

Ifstream In(“text”);

//можно ifstream In(“text”, ios::in);

Fstream Fl(‘text”, ios::out|ios::in|ios::binary);

Можно открыть файл, используя конструктор потоков без аргументов, а позже, вызвать функцию OPEN, которая имеет те же самые аргументы, что и у конструкторов классов istream и ofstream

Void open (const char\*name, int mode, int prot);

Ofstream file;

File.open (“text”, ios::out);

Пример программы:

#include <iostream>

#include <fstream>

#include <string.h>

#define N 50

Using namespace std

Class String

{

Char st[N];

Int size;

Public:

String(char \*S)

{ size = strlen(s);

Strcpy (st, S);}

~String () {}

Friend ostream& operator << (ostream&, const String&);

Friend istream& operator >> (istream&, String&);

};//endofclass

//перегрузка

Ostream& operator << (ostream& out, const String& obj)

{

Out << obj.st << endl;

Return out;

}

//перегрузка ввода

Operator >> (istream& in, String& obj)

{

In >> ob.st;

Return in;

}

Int main ()

{

String S, SS;

Cout << “vvedite object S” << endl;

Cin >> S;

Ofstream out \*/логическое имя/\* (“file” \*/физическое имя/\*);

If (!out)

{

Cout << “file ne otkrit” << endl;

Return;

}

Do {

Cout << “vvedite object S: “;

Cin >> S;

Out << S; //запись в файл

Cout >> “continue? Y/N “;

C = cin.get();

} while (c ==’Y’);

Out.close();

Ifstream in(“file”);

//проверка на открытие файла

While(1)

{

In >> SS;

If (in.eof()) break;

Cout << SS;

}

In.close();

Return 0;

} //end of main

Дз. В иерархии наследования выбрать один класс и записать его в файл в текстовой форме.

Бинарные файлы

Чаще всего, в бинарные файлы записываются объекты целиком. Для этого используется функция write принадлежащая классу ostream, и функция read принадлежащая классу istream. Функция write записывает фиксированное число байтов, начиная от заданного места в памяти в заданный поток. Функция read читает фиксированное число байтов из заданного потока в память, начиная с указанного адреса. Прототипы этих функций таковы:

Istream& istream::read(reinterpret\_cast<char\*> (&s), streamsize m);

Ostream& ostream::write(reinterpret\_cast<char\*> (&s), streamsize m);

Так как функции read и write первым аргументом требуют строку char\*, необходимо использовать оператор reinterpret\_cast<char\*> для преобразования адреса объекта (&s) в строку <char\*>.

Пример

Struct A

{

Char p[40];

Char r[20];

Int nomer;

};

Class stud

{

A a;

Public:

Void getData(); //записывать в файл

Void showData(); //прочесть из файла

};

Void stud::getData()

{

Char c=’n’;

Fstream file;

File.open (“FILE.DAT”, ios::out | ios::binary);

Cout << “vvod informacii” << endl;

Do

{

Cout << “vvod FIO” << endl;

Flushall();

Cin.getline(a.p, 40);

Cout << “vvod PIN” << endl;

Cin.getline(a.r, 20);

Cout << “vvod nomer” << endl;

Cin >> a.nomer;

File.write(reinterpret.cast<char\*> (&a), sizeof(A));

Cout << “continue? y/n”;

Cin >> c;

} //end of do

While (c==’y’ && c==’Y’);

File.close();

} //end of getdata

Void stud::showData()

{

Fstream file;

File.open (“FILE.DAT”, ios::in | ios::binary);

Cout << setw(15) << “client” << setw(15) << “pin” << setw(15) << “nomer” << endl;

While (1)

{

File.read(reinterpret\_cast<char\*> (&a), sizeof (A));

If (file.eof())

Break;

Cout << setw(15) << a.p << setw(15) << a.r << setw(15) << a.nomer << endl;

} //end of while

File.close();

} // end of showdata class

Int main()

{

Stud obj;

Obj.getData();

Obj.showData();

Return 0;

}

И для файлов произвольного доступа необходимо требование, чтобы ввсе записи в файле были одинаковой фиксированной длины. Это дает возможносто программе определять точное местоположение любой записи относительно начала или конца файла.

Классы istream и ostream содержат члены для позиционировани я указателя позиции файла (это порядковый номер следующего байта в файле, который должен быть считан или записан). Эти функции seekg (позиционирования для чтения/извлечения из потока) и seekp (позиционирование для помещения/записи в поток)

Прототипы одинаковы.

Istream& seekg(streampos pos);

Istream& seekg(streamoff off, ios::seek\_dir dir);

Аналогично ostream seekp

Любой объект класса istream имеет так называемый указатель get, который показывает номер в файле очередного вводимого байта. Любой же объект класса ostream имеет указатель put, который имеет номер очередного выводимого байта.

Fstream infile;

Infile.seekg(0); //данная запись позиционирует указатель позиции в начало файла

// аргумент streampos pos является целым числом (long)

, ios::seek\_dir dir – показывает направление позиционирования

Ios::beg – позиционирование относительно начала файла (по умолчанию)

Ios::cur- позиционирование относительно текущей позиции указателя в потоке

Ios::end – позиционирование относительно конца потока

Пример

Infile.seekg(n);

Позиционирование на n-ный байт относительно начала потока

Infile.seekg(n, ios::cur);

Позиционирование на n байтов вперед от курсора

Infile.seekg(0, ios::end);

Установка на конец файла

То же самое для функции seekp

Tellg();

Tellp();

Это возвращает текущие позиции указателя в численном формате.

Long loc=infile.tellg(); - запись позиции курсора

Пример

Char towns [] [15] = {“Grodno”,”Brest”,”Gomel”,”Vitebsk”,”Mogilev”}

Char a[15] = “Minsk”;

#define size sizeof (towns)/15;

Void main()

{

Fstream ff;

Char line [15];

ff.open(“my.txt”, ios::out);

for (int i=0; i<size; i++)

ff.write(towns[i],15);

ff.close();

ff.open(“my.txt”,ios::in);

cout << “file” << endl;

while (1)

{

ff.read(line, 15);

if(ff.eof()) break;

cout << line << endl;

}// end of while

ff.close();

ff.open(“my.txt”, ios::in | ios::out);

ff.seekg(0, ios::beg);

ff.read(line, 15);

ff.seekp(0, ios::beg);

ff.write (a, 15);

ff.seekp (-15, ios::end);

ff.write (line, 15);

ff.seekg(0, ios::beg);

} //end of main

Оператор dynamic\_cast

Этот оператор предназначен для приведения указателя или ссылки на объект базового класса к указателям или ссылкам на объекты производных классов

Приведение из базового класса в производный – называется понижающим приведением, приведение же из производного в базовый – называется повышающим приведением. Приведение между производными классами одного базового – называется перекрестным приведением.

Оператор dynamic\_cast имеет 2 операнда.

Dynamic\_cast <T\*> (P)

<…> - из которого

(…) – к которому

Class chelovek

{

Char fio[50];

Int god;

Public:

Chelovek(char \*s = “”, int n = 0)

{

Strcpy (fio, s);

God = n;

}

Friend ostream& operator << (ostream& out, chelovek& ob);

Friend istream& operator >> (istream& in, chelovek& ob);

};

Class turist : public chelovek

{

Int nomer\_pas;

Char granica[50];

Public:

Turist(char \*s = “”, int n=0, int nom = 0, vhar \*ss=””);

Chelovek (s, n)

{

Nomer\_pas = nom;

Strcpy (granica, s1);

}

Friend ostream& operator << (ostream& out, turist& ob);

Friend istream& operator >> (istream& in, turist& ob);

}

istream& operator >> (istream& in, chelovek& ob)

{

Cout << “vvod FIO” << endl;

In.getline (ob.fio, 50);

Cout << “vvod goda” << endl;

In >> ob.god;

Return in;

}

ostream& operator << (ostream& out, chelovek& ob)

{

Out << setw(10) << ob.fio << setw(10) << ob.god << endl;

Return out;

}

Istream& operator >> (istream& in, turist& ob)

{

In >> dynamic\_cast <chelovek&> (ob);

Cout << “vvod nomera pasporta” << endl;

Cin >> nomer\_pas;

Cout << “vvod granica” << endl;

Flushall();

In.getline (ob.granica, 40);

Return in;

}

Ostream& operator << (ostream& out, turist& ob)

{

Out << dynamic\_cast <chelovek&> (ob);

Out << setw(10) << ob.nomer\_pas << setw(10) << ob.granica << endl;

Return out;

}

Int main ()

{

Chelovek ob;

Turist ob1;

Cin >> ob;

Cout << ob;

Cin >> ob1;

Cout << ob1;

}

Преобразование типов, определяемых в программе.

Class my\_class

{

Double x, y;

Public:

My\_class (double X)

{

x = X;

y = x/3;

}

Double summa();

Operator double()

{return x;}

};

Double my\_class::summa()

{return x+y;}

Int main()

{

Double d;

My\_class my\_obj1(15);

My\_class my\_obj2(15);

My\_obj3 = 15;

}

При перегрузке оператора () мы задали данные прямого типа. Т.е. этот оператор преобразует значения к типу double и возвращает значения этого компонента.

Человек

Лист (двунаправленный список

В файл

На большую оценку – извлечь из файла и вывести на экран