1.使用sort()函数需要引入头文件#include <algorithm>和using namespace std;  
2.sort()函数可以是三个参数也可以是两个参数;  
3.该函数使用的排序方法类似于快速排序，时间复杂度为 log2N;

4.sort()函数的三个参数：（第三个参数可以不写）  
（1）：第一个数要排序的数组的起始地址。  
（2）：第二个是结束的地址（最后一位要排序的地址）  
（3）：第三个参数是排序的方法，可以是从大到小也可以是从小到大的，还可以不写第三个参数，此时默认的排序方法是从小到大排序。

C++ 的抽象类和接口

C++ 接口是使用**抽象类**来实现的，抽象类与数据抽象互不混淆，数据抽象是一个把实现细节与相关的数据分离开的概念。

如果类中至少有一个函数被声明为纯虚函数，则这个类就是抽象类。纯虚函数是通过在声明中使用 "= 0" 来指定的，

首先：强调一个概念  
定义一个函数为虚函数，不代表函数为不被实现的函数。  
定义他为虚函数是为了允许用基类的指针来调用子类的这个函数。  
定义一个函数为纯虚函数，才代表函数没有被实现。  
定义纯虚函数是为了实现一个接口，起到一个规范的作用，规范继承这个类的程序员必须实现这个函数。

**const修饰数据成员**

const数据成员只在某个对象生存期内是常量，而对于整个类而言却是可变的。因为类可以创建多个对象，不同的对象其const数据成员的值可以不同。所以不能在类声明中初始化const数据成员，因为类的对象未被创建时，编译器不知道const 数据成员的值是什么，例如：

class A  
{  
    const int size = 100; //错误  
    int array[size];       //错误，未知的size  
}

const数据成员的初始化只能在类的构造函数的初始化列表中进行。要想建立在整个类中都恒定的常量，可以用类中的枚举常量来实现，例如：

class A  
{

…  
　　enum {size1=100, size2 = 200 };  
　　int array1[size1];  
　　int array2[size2];

…  
}

枚举常量不会占用对象的存储空间，他们在编译时被全部求值。但是枚举常量的隐含数据类型是整数，其最大值有限，且不能表示浮点数。

**const修饰成员函数**

const修饰类的成员函数，用const修饰的成员函数不能改变对象的成员变量。一般把const写在成员函数的最后：

class A

{

   …

void function()const; //常成员函数, 它不改变对象的成员变量. 也不能调用类中任何非const成员函数。

}

**const修饰普通变量和指针**

const修饰变量，一般有两种写法：

const TYPE value;

TYPE const value;

这两种写法在本质上是一样的。它的含义是：const修饰的类型为TYPE的变量value是不可变的。对于一个非指针的类型TYPE，无论怎么写，都是一个含义，即value值不可变。 例如：

const int nValue；    //nValue是const

int const nValue；    //nValue是const

但是对于指针类型的TYPE，不同的写法会有不同情况：

（）指针本身是常量不可变

(char\*) const pContent;

（）指针所指向的内容是常量不可变

const (char) \*pContent;

(char) const \*pContent;

（）两者都不可变

const char\* const pContent;

识别const到底是修饰指针还是指针所指的对象，还有一个较为简便的方法，也就是沿着\*号划一条线：

如果const位于\*的左侧，则const就是用来修饰指针所指向的变量，即指针指向为常量；

如果const位于\*的右侧，const就是修饰指针本身，即指针本身是常量。

2.1 使用virtual关键字   
    考虑下面的类层次：  
  
class A  
{  
public:  
    virtual void foo();  
};  
  
class B: public A  
{  
public:  
    void foo();    // 没有virtual关键字!  
};  
  
class C: public B  // 从B继承，不是从A继承！  
{  
public:  
    void foo();    // 也没有virtual关键字！  
};  
  
    这种情况下，B::foo()是虚函数，C::foo()也同样是虚函数。因此，可以说，基类声明的虚函数，在派生类中也是虚函数，即使不再使用virtual关键字。

继承中的二义性

在C++中，派生类继承基类，对基类成员的访问应该是确定的、唯一的，但是常常会有以下情况导致访问不一致，产生二义性。

    1.在继承时，基类之间、或基类与派生类之间发生成员同名时，将出现对成员访问的不确定性——同名二义性。

    2.当派生类从多个基类派生，而这些基类又从同一个基类派生，则在访问此共同基类中的成员时，将产生另一种不确定性——路径二义性。

    1.

“倒三角”问题——同名二义性

#include "iostream"

using namespace std;

class Parent\_f

{

public:

void show()

{

cout<<"This is Parent\_f\n";

}

};

class Parent\_m

{

public:

void show()

{

cout<<"This is Parent\_m\n";

}

};

class Son:public Parent\_f,public Parent\_m

{

public:

void display()

{

cout<<"This is son\n";

}

};

int main()

{

Son son;

son.show();

son.display();

cout << "Hello world!" << endl;

return 0;

}

上面的代码中，2个父类派生一个子类，但两个父类中都有同名的成员函数。派生出的子类产生二义性问题，编译时会报：

error: request for member 'show' is ambiguous

    解决方法：

    （1）利用作用域限定符（：：），用来限定子类调用的是哪个父类的show（）函数

son.Parent\_f::show();

    （2）在类中定义同名成员，覆盖掉父类中的相关成员

class Son:public Parent\_f,public Parent\_m

{

public:

void display()

{

cout<<"This is son\n";

}

void show()

{

cout<<"show:This is son.\n";

}

};

2.

“菱形”问题——路径二义性

    有最基类A，有A的派生类B、C，又有D同时继承B、C，那么若A中有成员a，那么在派生类B，C中就存在a，又D继承了B，C，那么D中便同时存在B继承A的a和C继承A的a，那么当D的实例调用a的时候就不知道该调用B的a还是C的a，就导致了二义性。

#include "iostream"

using namespace std;

class Grandpa

{

public:

int year\_old;

};

class Parent\_f:public Grandpa {};

class Parent\_m:public Grandpa {};

class Son:public Parent\_f,public Parent\_m {};

int main()

{

Son son;

son.year\_old = 10;

cout << "Hello world!" << endl;

return 0;

}

    Grandpa为Parent\_f和Parent\_m的基类，而Son又继承Parent\_f和Parent\_m，当Son访问Grandpa的year\_old时，会出现二义性错误.

    解决方法：

    （1）使用作用域限定符，指明访问的是哪一个基类的成员。

            注意：不能是Grandpa作用域下限定，因为Son直接基类的基类才是Grandpa，纵使指明了访问Grandpa的成员的话，对于Son来说，还是模糊的，还是具有二义性。

    （2）在类中定义同名成员，覆盖掉父类中的相关成员。

    （3）虚继承、使用虚基类

        教科书上面对C++虚基类的描述玄而又玄，名曰“共享继承”，名曰“各派生类的对象共享基类的的一个拷贝”，其实说白了就是解决多重多级继承造成的二义性问题。父类对祖父类的继承方式改为虚继承，那么子类访问自己从祖父类那里继承过来的成员就不会有二义性问题了，也就是将子类对象里的祖父类对象统一为一个，继承的只有一个祖父类对象，代码如下。

#include "iostream"

using namespace std;

class Grandpa

{

public:

int year\_old;

void show()

{

cout << "year\_old:" << year\_old <<endl;

}

};

class Parent\_f:virtual public Grandpa {};

class Parent\_m:virtual public Grandpa {};

class Son:public Parent\_f,public Parent\_m {};

int main()

{

Grandpa grp;

Parent\_f pa\_f;

Parent\_m pa\_m;

Son son;

grp.year\_old = 100;

pa\_f.year\_old = 55;

pa\_m.year\_old = 50;

son.year\_old = 10;

grp.show();

pa\_f.show();

pa\_m.show();

son.show();

cout << "Hello world!" << endl;

return 0;

}

# C++ 引用

引用变量是一个别名，也就是说，它是某个已存在变量的另一个名字。一旦把引用初始化为某个变量，就可以使用该引用名称或变量名称来指向变量。

**C++ 引用 vs 指针**

引用很容易与指针混淆，它们之间有三个主要的不同：

* 不存在空引用。引用必须连接到一块合法的内存。
* 一旦引用被初始化为一个对象，就不能被指向到另一个对象。指针可以在任何时候指向到另一个对象。
* 引用必须在创建时被初始化。指针可以在任何时间被初始化。
* 无论是对变量的改变还是对引用的改变都将会引起原始变量的改变，一荣俱荣，一损俱损

函数如果传参穿的是引用，那么对参数的改变将会引起原变量的改变；如果传的是值，则对参数的改变不会引起原变量的改变。

                                      c++类的构造函数详解                          
  
一、 构造函数是干什么的  
  
class Counter  
{  
  
public:  
         // 类Counter的构造函数  
         // 特点：以类名作为函数名，无返回类型  
         Counter()  
         {  
                m\_value = 0;  
         }  
           
private:  
        
         // 数据成员  
         int m\_value;  
}  
  
  
       该类对象被创建时，编译系统对象分配内存空间，并自动调用该构造函数->由构造函数完成成员的初始化工作  
  
eg:    Counter c1;  
        编译系统为对象c1的每个数据成员(m\_value)分配内存空间，并调用构造函数Counter( )自动地初始化对象c1的m\_value值设置为0  
  
故：  
  
        构造函数的作用：初始化对象的数据成员。  
  
  
二、 构造函数的种类  
  
class Complex   
{           
  
private :  
        double    m\_real;  
        double    m\_imag;  
  
public:  
  
        //    无参数构造函数  
        // 如果创建一个类你没有写任何构造函数,则系统会自动生成默认的无参构造函数，函数为空，什么都不做  
        // 只要你写了一个下面的某一种构造函数，系统就不会再自动生成这样一个默认的构造函数，如果希望有一个这样的无参构造函数，则需要自己显示地写出来  
        Complex(void)  
        {  
             m\_real = 0.0;  
             m\_imag = 0.0;  
        }   
          
        //    一般构造函数（也称重载构造函数）  
        // 一般构造函数可以有各种参数形式,一个类可以有多个一般构造函数，前提是参数的个数或者类型不同（基于c++的重载函数原理）  
        // 例如：你还可以写一个 Complex( int num)的构造函数出来  
        // 创建对象时根据传入的参数不同调用不同的构造函数  
        Complex(double real, double imag)  
        {  
             m\_real = real;  
             m\_imag = imag;           
         }  
          
        //    复制构造函数（也称为拷贝构造函数）  
        //    复制构造函数参数为类对象本身的引用，用于根据一个已存在的对象复制出一个新的该类的对象，一般在函数中会将已存在对象的数据成员的值复制一份到新创建的对象中  
        //    若没有显示的写复制构造函数，则系统会默认创建一个复制构造函数，但当类中有指针成员时，由系统默认创建该复制构造函数会存在风险，具体原因请查询 有关 “浅拷贝” 、“深拷贝”的文章论述  
        Complex(const Complex & c)  
        {  
                // 将对象c中的数据成员值复制过来  
                m\_real = c.m\_real;  
                m\_img    = c.m\_img;  
        }              
      
        // 类型转换构造函数，根据一个指定的类型的对象创建一个本类的对象  
        // 例如：下面将根据一个double类型的对象创建了一个Complex对象  
        Complex::Complex(double r)  
        {  
                m\_real = r;  
                m\_imag = 0.0;  
        }  
  
        // 等号运算符重载  
        // 注意，这个类似复制构造函数，将=右边的本类对象的值复制给等号左边的对象，它不属于构造函数，等号左右两边的对象必须已经被创建  
        // 若没有显示的写=运算符重载，则系统也会创建一个默认的=运算符重载，只做一些基本的拷贝工作  
        Complex &operator=( const Complex &rhs )  
        {  
                // 首先检测等号右边的是否就是左边的对象本，若是本对象本身,则直接返回  
                if ( this == &rhs )   
                {  
                        return \*this;  
                }  
                  
                // 复制等号右边的成员到左边的对象中  
                this->m\_real = rhs.m\_real;  
                this->m\_imag = rhs.m\_imag;  
                  
               // 把等号左边的对象再次传出  
               // 目的是为了支持连等 eg:    a=b=c 系统首先运行 b=c  
               // 然后运行 a= ( b=c的返回值,这里应该是复制c值后的b对象)      
                return \*this;  
        }  
  
};  
  
下面使用上面定义的类对象来说明各个构造函数的用法：  
  
void main()  
{  
        // 调用了无参构造函数，数据成员初值被赋为0.0  
        Complex c1，c2;  
  
        // 调用一般构造函数，数据成员初值被赋为指定值  
        Complex c3(1.0,2.5);  
        // 也可以使用下面的形式  
        Complex c3 = Complex(1.0,2.5);  
          
        //    把c3的数据成员的值赋值给c1  
        //    由于c1已经事先被创建，故此处不会调用任何构造函数  
        //    只会调用 = 号运算符重载函数  
        c1 = c3;  
          
        //    调用类型转换构造函数  
        //    系统首先调用类型转换构造函数，将5.2创建为一个本类的临时对象，然后调用等号运算符重载，将该临时对象赋值给c1  
        c2 = 5.2;  
          
       // 调用拷贝构造函数( 有下面两种调用方式)   
        Complex c5(c2);  
        Complex c4 = c2;  // 注意和 = 运算符重载区分,这里等号左边的对象不是事先已经创建，故需要调用拷贝构造函数，参数为c2  
  
          
          
}  
  
三、思考与测验  
  
1. 仔细观察复制构造函数  
  
        Complex(const Complex & c)  
        {  
                // 将对象c中的数据成员值复制过来  
                m\_real = c.m\_real;  
                m\_img = c.m\_img;  
        }      
          
为什么函数中可以直接访问对象c的私有成员？  
  
2. 挑战题，了解引用与传值的区别  
  
  Complex test1(const Complex& c)  
  {  
          return c;  
  }  
    
  Complex test2(const Complex c)  
  {  
         return c;  
   }  
     
   Complex test3()  
   {  
          static Complex c(1.0,5.0);  
          return c;  
   }  
    
  Complex& test4()  
  {  
         static Complex c(1.0,5.0);  
         return c;  
  }  
    
  void main()  
  {  
        Complex a,b;  
      
        // 下面函数执行过程中各会调用几次构造函数，调用的是什么构造函数？  
      
       test1(a);  
       test2(a);  
       
       b = test3();  
       b = test4();  
       
       test2(1.2);  
       // 下面这条语句会出错吗？  
       test1(1.2);     //test1( Complex(1.2 )) 呢？  
  }  
  
四、附录(浅拷贝与深拷贝)  
  
       上面提到，如果没有自定义复制构造函数，则系统会创建默认的复制构造函数，但系统创建的默认复制构造函数只会执行“浅拷贝”，即将被拷贝对象的数据成员的值一一赋值给新创建的对象，若该类的数据成员中有指针成员，则会使得新的对象的指针所指向的地址与被拷贝对象的指针所指向的地址相同，delete该指针时则会导致两次重复delete而出错。下面是示例：  
  
【浅拷贝与深拷贝】  
  
#include <iostream.h>  
#include <string.h>  
class Person   
{  
public :  
          
        // 构造函数  
        Person(char \* pN)  
        {  
              cout << "一般构造函数被调用 !\n";  
              m\_pName = new char[strlen(pN) + 1];  
              //在堆中开辟一个内存块存放pN所指的字符串  
              if(m\_pName != NULL)   
              {  
                 //如果m\_pName不是空指针，则把形参指针pN所指的字符串复制给它  
                   strcpy(m\_pName ,pN);  
              }  
        }          
          
        // 系统创建的默认复制构造函数，只做位模式拷贝  
        Person(Person & p)      
        {   
                  //使两个字符串指针指向同一地址位置           
                 m\_pName = p.m\_pName;           
        }  
     
        ~Person( )  
        {  
                delete m\_pName;  
        }  
          
private :  
  
        char \* m\_pName;  
};  
  
void main( )  
{   
        Person man("lujun");  
        Person woman(man);   
          
        // 结果导致   man 和    woman 的指针都指向了同一个地址  
          
        // 函数结束析构时  
        // 同一个地址被delete两次  
}  
  
  
// 下面自己设计复制构造函数，实现“深拷贝”，即不让指针指向同一地址，而是重新申请一块内存给新的对象的指针数据成员  
Person(Person & chs);  
{  
         // 用运算符new为新对象的指针数据成员分配空间  
         m\_pName=new char[strlen(p.m\_pName)+ 1];  
  
         if(m\_pName)           
         {  
                 // 复制内容  
                strcpy(m\_pName ,chs.m\_pName);  
         }  
        
        // 则新创建的对象的m\_pName与原对象chs的m\_pName不再指向同一地址了  
}

重载

一、基本定义

1、C++预定义的运算符只能用于基本数据类型的运算：整型、实型、字符型、逻辑型，但是不能作用于对象之间。

2、运算符重载的目的是扩展C++中提供的运算符的适用范围，使之能作用于对象。

3、运算符重载的实质是函数重载，可以重载为普通成员函数，也可以重载为成员函数。

4、根据实参的类型决定调用哪个运算符函数。

二、运算符重载的基本形式

不管什么样的，返回值是什么（除掉返回值是bool类型），最好是都加上&符号，如果返回的是一个临时变量的话，也就是在重载里面自己定义了一个变量的话，这个时候最好不加&符号。

返回值类型& operator 运算符（形参表）

{

}

或者

返回值类型 operator 运算符（形参表）

{

}

1

2

3

4

5

6

7

8

9

10

11

1、只有==、>、<这三个的返回值一定是是bool类型，其余的[]等返回值类型要根据题意自己定

2、如果作用的是同一个类型的话重载为成员函数就好，如果有不同类型的话，则必须用friend，定义为友元函数。

3、一般（）里面包括：类名& 名称；&号千万别忘记；

三、运算符目数

(1)、单目：只需要一个操作的意思，例如：++a --a a &a

(2)、双目：需要两个操作数的意思 例如： a++ a-- a+b a-b ab a/b a%b

(3)、三目：是需要三个操作数的意思 比如 a=c>b?c:b;

四、详解

重载为成员函数时，参数个数为运算符目数减一；

重载为普通函数时，参数个数为运算符目数；

五、注意事项

1、c++不允许定义新的运算符

2、重载后运算符的含义应该符合日常习惯

3、运算符重载不改变运算符优先级

4、”.” “.\*” “::” “?:” sizeof

5、重载运算符（）[] -> =的时候运算符重载函数必须声明为类的成员函数

1普通函数：属于全局函数，不受具体类和对象的限制，可以直接调用。

2普通成员函数：C++ 普通成员函数本质上是一个包含指向具体对象this指针的普通函数，即c++类的普通成员函数都隐式包含一个指向当前对象的this指针。

静态成员函数

首先介绍类的静态成员变量： 类体中的数据成员的声明前加上static关键字，该数据成员就成为了该类的静态数据成员。

静态成员变量的性质：

1）static型变量只被初始化一次，下次执行初始化语句会直接跳过。

2）static 说明一个类的成员为静态成员，经过static修饰的成员变量属于这个类，不再仅仅属于具体的对象。

再介绍静态成员函数： 类体中的成员函数的声明前加上static关键字，该成员函数就成为了该类的静态成员函数。

静态成员函数的性质：

1）不可以调用类的非静态成员。

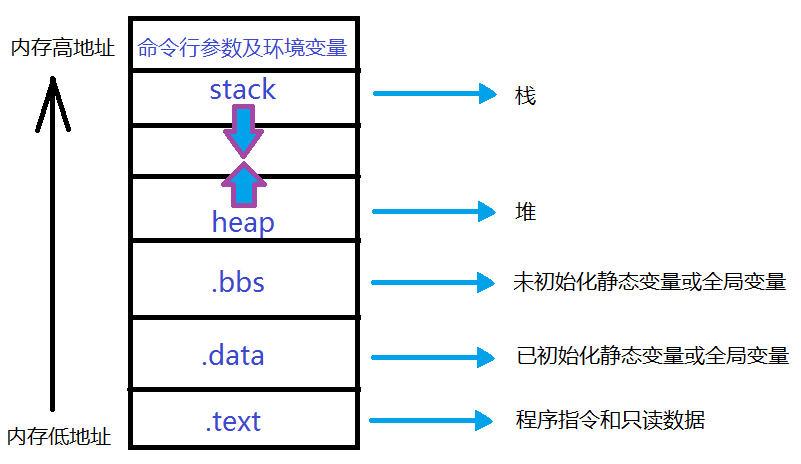
2）静态成员函数不含this指针。 静态成员函数属于这个类，不再仅仅属于具体的对象。

因此类的静态成员函数和类的普通成员函数的区别是：

静态成员函数不包含指向具体对象的this指针；

普通成员函数包含一个指向具体对象的this指针

[程序在内存中的分布](https://www.cnblogs.com/Lynn-Zhang/p/5449199.html)

****

**代码段**(.text)，也称文本段(Text Segment)，存放着程序的机器码和只读数据，可执行指令就是从这里取得的。如果可能，系统会安排好相同程序的多个运行实体共享这些实例代码。这个段在内存中一般被标记为只读，任何对该区的写操作都会导致段错误（Segmentation Fault）。

**数据段**，包括已初始化的数据段(.data)和未初始化的数据段（.bss），前者用来存放保存全局的和静态的已初始化变量，后者用来保存全局的和静态的未初始化变量。数据段在编译时分配。

**堆栈段**分为堆和栈：

* **堆**（Heap）：用来存储程序运行时分配的变量。

       堆的大小并不固定，可动态扩张或缩减。其分配由malloc()、new()等这类实时内存分配函数来实现。当进程调用malloc等函数分配内存时，新分配的内存就被动态添加到堆上（堆被扩张）；当利用free          等函数释放内存时，被释放的内存从堆中被剔除（堆被缩减） 堆的内存释放由应用程序去控制，通常一个new()就要对应一个delete()，如果程序员没有释放掉，那么在程序结束后操作系统会自动回收。

* **栈**（Stack）是一种用来存储函数调用时的临时信息的结构，如函数调用所传递的参数、函数的返回地址、函数的局部变量等。 在程序运行时由编译器在需要的时候分配，在不需要的时候自动清除。

        栈的特性: 最后一个放入栈中的物体总是被最先拿出来，这个特性通常称为先进后出(FILO)队列。

        栈的基本操作：**PUSH操作**：向栈中添加数据，称为压栈，数据将放置在栈顶； **POP操作**：POP操作相反，在栈顶部移去一个元素，并将栈的大小减一，称为弹栈。

**堆和栈的区别：**

1.**分配和管理方式不同** ：

* 堆是动态分配的，其空间的分配和释放都由程序员控制。
* 栈由编译器自动管理。栈有两种分配方式：静态分配和动态分配。

                    静态分配由编译器完成，比如局部变量的分配。

                    动态分配由alloca()函数进行分配，但是栈的动态分配和堆是不同的，它的动态分配是由编译器进行释放，无须手工控制。

2.**产生碎片不同**

* 对堆来说，频繁的new/delete或者malloc/free势必会造成内存空间的不连续，造成大量的碎片，使程序效率降低。
* 对栈而言，则不存在碎片问题，因为栈是先进后出的队列，永远不可能有一个内存块从栈中间弹出。

3.**生长方向不同**

* 堆是向着内存地址增加的方向增长的，从内存的低地址向高地址方向增长。
* 栈的生长方向与之相反，是向着内存地址减小的方向增长，由内存的高地址向低地址方向增长。

6、C++的内存分区· 栈区（stack）：主要存放函数参数以及局部变量，由系统自动分配释放。· 堆区（heap）：由用户通过 malloc/new 手动申请，手动释放。· 全局/静态区：存放全局变量、静态变量；程序结束后由系统释放。· 字符串常量区：字符串常量就放在这里，程序结束后由系统释放。· 代码区：存放程序的二进制代码。 堆和栈的区别· 栈由编译器自动分配释放，存放函数参数、局部变量等。而堆由程序员手动分配和释放；· 栈是向低地址扩展的数据结构，是一块连续的内存的区域。而堆是向高地址扩展的数据结构，是不连续的内存区域；· 栈的默认大小为1M左右，而堆的大小可以达到几G，仅受限于计算机系统中有效的虚拟内存。

友元函数

友元函数的声明在类中，但是友元函数的定义可以在类中也可以在类外

定义list的类型

list\_t  List;  //定义一个空的链表

list\_t  List(count); //建一个含count个默认值是0的元素的链表

list\_t  List{count}; //建一个含count个默认值是0的元素的链表

list\_t  List(count, info); //建一个含count个默认值是info的元素的链表

list\_t  List(List2);   //建一个的copy链表

list\_t  List{list2};   //建一个的copy链表

list\_t  List(List2.begin(),List2.end());   //含区间的元素[First,Last] 使用迭代器来构造list对象

1. A a[5] = { 1, 3, 2, 4, 2 };
2. A b[7] = { 10, 30, 20, 30, 30, 40, 40 };
3. list<A> lst1(a, a + 5), lst2(b, b + 7);

可以将一个数组通过此种方式初始化为一个链表

pop\_back() 删除最后一个元素

pop\_front() 删除第一个元素

push\_back() 在list的末尾添加一个元素

push\_front() 在list的头部添加一个元素