继承与多态-深入掌握oop语言最强大的机制

1.继承的基本意义
2.派生类的构造过程
3.重载,隐藏,覆盖
4.虚函数， 静态绑定和动态绑定--面试重点
5.虚析构函数--重点在于什么呢时候用
6.再讨论虚函数和动态绑定
7.理解多态到底是什么
8.理解抽象类----纯虚函数
9.笔试问题讲解

继承与多态-多重继承

1.虚基类和虚继承
    本节内容
2.菱形继承---怎么解决?
    本节内容
    面试问题： 怎么理解多重继承的?---重点
3.c++提供的四种类型转换
    本节内容

STL组件

1.整体学习内容
2.vector容器
3.deque和list
    deque--双端队列容器
    list--链表容器
4.vector,deque,list对比
    主要内容
    面经问题
5.详解容器适配器--stack, queue, priority_queue
    容器适配器
    stack-栈
    queue-队列
    priority_queue-优先级队列
    总结
6.无序关联容器
    关联容器
    unordered_set
    unordered_map
    应用
7.有序关联容器 - set,map
    set
    map
    pair注意
8.迭代器iterator
9.函数对象--仿函数

10.泛型算法和绑定器

继承与多态-深入掌握oop语言最强大的机制

1.继承的基本意义

1. 继承的本质和原理?
   继承的本质: 1.做代码复用; 2.
   类和类的关系:
   组合: a part of ... 一部分的关系
   继承: a kind of ... 一种的关系

```cpp
#include <iostream>
using namespace std;

class A
{
public:
    int ma;
protected:
    int mb;
private:
    int mc;
};
// A 12字节

//class B : public A //继承    A是基类/父类  B是派生类/子类
//{
//public:
//  void func()
//  {
//      cout << "ma:" << ma << endl;
//  }
//  int md;
//  int me;
```

```
//  int mf;
//};


//class B : protected A //继承    A是基类/父类  B是派生类/子类
//{
//public:
//  void func()
//  {
//
//  }
//  int md;
//  int me;
//  int mf;
//};

class B : private A //继承    A是基类/父类  B是派生类/子类
{
public:
    void func()
    {

    }
    int md;
    int me;
    int mf;
};

//B有两部分, A+B本身, 24字节

class C : public B //继承    A是基类/父类  B是派生类/子类
{
public:
    void func()
    {

    }
```

```
61      int md;
62      int me;
63      int mf;
64  };
65
66  int main()
67  {
68
69  }
```

2. 继承的细节

继承方式	基类的访问限定	派生类的访问限定	(main)外部的访问限定
public	public	public	ok
	protected	protected	no(main只能访问公有的)
	private	不可见,无法访问,不是私有	no
protected	public	降级, protected	no
	protected	protected	no
	private	不可见,无法访问,不是私有	no
private	public	private	no
	protected	private	no
	private	不可见,无法访问,不是私有	no

> private只有自己和友元能访问!!

3. 对于更多的继承, class C ,要看直接继承的B里面的各个访问限定和继承方式
4. 总结:
   外部只能访问对象public的成员, protected和private的成员无法直接访问
   继承结构中, 派生类从基类可以继承过来private的成员, 但是派生类无法访问
   protected和private的区别?----基类中定义的成员, 想被派生类访问, 但

是不想被外部访问，那么基类可以定义为protected；　若 派生类和外部都不访问，基类中设置为private

5. 默认继承方式------要看派生类是class(private)还是struct(public)!!!

2.派生类的构造过程

1. 派生类怎么初始化从基类继承的成员变量呢?
   派生类可以从继承得到 继承的所有的成员(变量+方法)，除了构造和析构
   解决办法：调用基类的 构造函数
   派生类的构造和析构，负责初始化和清理派生类部分
   派生类从基类继承来的成员--的初始化和清理由谁负责?----由基类的构造和析构

2. 派生类对象构造和析构过程?
   派生类调用基类构造(初始化基类继承来的成员)--->派生类自己构造--->派生类自己的析构--->调用基类的析构

```
#include <iostream>
using namespace std;

class Base
{
public:
    Base(int data) :ma(data) { cout << "Base()" << endl; }
    ~Base() { cout << "~Base()" << endl; }
protected:
    int ma;
};

class Derive : public Base
{
public:
    /*Derive(int data) :ma(data), mb(data)*/
    //改为
    Derive(int data) :Base(data), mb(data)
    {
        cout << "Derive()" << endl;
```

```
    }
    ~Derive()
    {
        cout << "~Derive()" << endl;
    }
private:
    int mb;
};

int main()
{
    Derive d(20);

    return 0;
}

/*输出:
Base()
Derive()
~Derive()
~Base()*/

```

3.重载,隐藏,覆盖

1. 重载关系
   一组函数要重载，必须在一个作用域内;且函数名相同，参数列表不同
   基类和派生类是两个不同的作用域!!
2. 隐藏的关系
   在继承结构中，派生类的同名成员，把基类的同名成员给 隐藏了，也就是默认调用派生类的同名成员
   想要基类，就要 加基类作用域

```
#include <iostream>
using namespace std;

class Base
{
```

```
public:
    Base(int data=10) :ma(data) {  }
    ~Base() {  }
    void show() { cout << "Base:show" << endl; }
    void show(int) { cout << "Base:show(int)" << endl;
}
protected:
    int ma;
};

class Derive : public Base
{
public:
    Derive(int data=20) :Base(data), mb(data)
    {
    }
    ~Derive()
    {
    }
    void show() { cout << "Derive:show" << endl; }
private:
    int mb;
};

int main()
{
    Derive d;
    d.show(); // 隐藏了 基类的同名成员,  没有才去基类
Derive:show
    d.Base::show();  //Base:show
    //d.show(10);//Derive 没有函数重载
    return 0;
}

//输出
Derive:show
Base:show

```

1. 把继承结构，也理解为 从上(基类)到下(派生类)的结构

2．基类对象变为⟶派生类对象，派生类对象变为--->基类对象，基类对象(引用)指向⟶派生类对象，派生类对象(引用)指向--->基类对象

这是不强转时，直接=的情况，实际强转还是可以的

```
int main()
{
    Base b(10);
    Derive d(20);
    //派生类对象--->基类对象     类型从下到上的转换 yes
    //怎么理解？派生类看做学生，基类看做人，想要人，把学生给你，是可以的
    b = d;  //相当于把派生类里继承的基类给了 基类

    //基类对象⟶派生类对象    类型从上到下的转换 no
    //d = b; //这是不行的，多出来了内存 派生类自己的

    //基类对象(引用)⟶指向  派生类对象  类型从下到上的转换 yes
    Base* pb = &d; // 是可以的，解引用这能访问 Base那么大的区域，后面的派生类自己的那部分，管不着，本来也管不着
    pb→show(); //yes  想要访问派生类show，可以强转
    pb→show(20); //yes

    //派生类对象(引用)--->指向  基类对象 类型从上到下的转换 no
    //Derive* pd = &b; //会访问b没有的内存区域，属于非法访问



    return 0;
}
```

3．总结：**继承结构中，只支持从下到上的类型转换!!!!-----前提是不强转**

4.虚函数， 静态绑定和动态绑定--面试重点

1. 什么时候是动态绑定?---并不是有虚函数就是动态绑定， 这是误区， 这个老师这节课开始没说明白!!到了第六节课才说

```
1.函数是虚函数：基类中的函数必须声明为 virtual。

2.通过 指针或引用 调用：通过基类指针或基类引用调用虚函数。

这是 必要条件!!!二者必须满足

即
    Derive d(20);
    Base *pb=&d;
```

2. 无虚函数时：

```
#include <iostream>
using namespace std;

class Base
{
public:
    Base(int data=10) :ma(data) {  }
    ~Base() {  }
    void show() { cout << "Base:show" << endl; }
    void show(int) { cout << "Base:show(int)" << endl; }
protected:
    int ma;
};

class Derive : public Base
{
public:
    Derive(int data=20) :Base(data), mb(data)
    {
    }
    ~Derive()
```

```
    {
    }
    void show() { cout << "Derive:show" << endl; }
private:
    int mb;
};

int main()
{
    Derive d(50);
    Base* pb = &d;
    pb->show(); // 静态(编译时期)的绑定(函数的调用) Base::show (07FF7C6ED1406h)
    pb->show(10); // 静态(编译时期)的绑定(函数的调用) Base::show (07FF7C6ED10DCh)

    cout << sizeof(Base) << endl;  // 4
    cout << sizeof(Derive) << endl; // 8

    cout << typeid(pb).name() << endl; //class Base* __ptr64

    cout << typeid(*pb).name() << endl; // class Base

    return 0;
}
```

3. 虚函数的总结-1:
   如果类里面定义了虚函数，编译阶段 就会给这个 类类型 产生一个 惟一的 vftable 虚函数表，这里面主要存储的内容就是 RTTI指针和虚函数的地址
   RTTI--run-time type infomation
   程序运行时，每一张虚函数表 都会加载到内存的 .rodata区，只能读，不能写

4. 虚函数的总结-2:
   一个类里有虚函数，这个类定义的对象，在其运行时，内存的开始部分， 多存储一个 vfptr虚函数指针， 指向相应类型的 虚函数表vftable. 一个类型定义的n个对象，vfptr都指向同一个虚函数表

5. 虚函数的总结-3:
   一个类里 虚函数的个数，不影响对象的大小(对象里只是指针)， 影响的是虚函数表的大小

6. **虚函数的总结-4:----覆盖!!!**
   如果派生类中的方法和基类继承来的某个方法， 函数名,参数列表,返回值都一样， 切基类这个方法是虚函数virtual,
   则 派生类的 该方法 会 自动处理成虚函数---这是cpp标准
   因此， 派生类的 虚函数表里，该函数会覆盖基类的

7. 有虚函数:
   Base将不再是只有ma了， 还有虚函数指针vfptr， 指向虚函数表
   因此不再是4字节， 而是8字节

8. 静态绑定和动态绑定
   静态(编译时期)的绑定(函数的调用)
   动态(运行时期)的绑定(函数的调用)

9. 从汇编看 静动态绑定

```
//静态绑定的汇编
mov          rcx,qword ptr [pb]    //this指针存储
call         Base::show (07FF7C6ED1406h)


//动态绑定:  x64的反汇编
 mov          rax,dword ptr [pb]    //把pb里面的地址给eax, 即虚函数表地址
 mov          rax,dword ptr [rax]   //取存储虚函数表地址里的地址,即虚函数表的地址

mov          rcx,qword ptr [pb]    //this指针存储
call         qword ptr [rax+8] //调用虚函数表里偏移8字节的地址里面的地址, 即 show的地址, 8字节是RTTI的信息

```

10. 总体代码:
    该代码在show 是不是虚函数时， 输出不一样的-----x64

```
//是虚函数-------注意考虑内存对齐
Derive:show
Base:show(int)
16
```

```
24
class Base * __ptr64
class Derive

//不是虚函数
Base:show
Base:show(int)
4
8
class Base * __ptr64
class Base
```

```
#include <iostream>
using namespace std;

class Base
{
public:
    Base(int data=10) :ma(data) {  }
    ~Base() {  }
    virtual void show() { cout << "Base:show" << endl; }
    virtual void show(int) { cout << "Base:show(int)" << endl; }
protected:
    int ma;
};

class Derive : public Base
{
public:
    Derive(int data=20) :Base(data), mb(data)
    {
    }
    ~Derive()
    {
    }
    void show() { cout << "Derive:show" << endl; }
private:
```

```
26      int mb;
27  };
28
29  int main()
30  {
31      //有虚函数时
32      Derive d(50);
33      Base* pb = &d;
34      pb->show(); // 静态(编译时期)的绑定(函数的调用)
35      //静态绑定的汇编   call  Base::show
    (07FF7C6ED1406h)
36      // pb 是Base指针, 如果, Base::show 是普通函数, 则进行
    静态绑定
37      // pb 是Base指针, 如果, Base::show 是虚函数, 就进行动
    态绑定
38
39      /*
40      动态绑定:  x64的反汇编
41       mov          rax,dword ptr [pb]   把pb里面的地址给
    eax, 即虚函数表地址
42       mov          rax,dword ptr [rax]  取存储虚函数表地
    址里的地址,即虚函数表的地址
43
44      mov          rcx,qword ptr [pb]   this指针存储
45       call         qword ptr [rax+8] 调用虚函数表里偏移8字
    节的地址里面的地址, 即 show的地址, 这个+8, 不是RTTI的, 一般
    RTTI是反偏移的,  是因为覆盖是原来的不要了, 新的放在最下面
46      */
47
48
49      pb->show(10); // 静态(编译时期)的绑定(函数的调用)
    Base::show (07FF7C6ED10DCh)
50
51      cout << sizeof(Base) << endl;
52      cout << sizeof(Derive) << endl;
53
54      cout << typeid(pb).name() << endl; //class Base*
    __ptr64
55
```

```
56      cout << typeid(*pb).name() << endl; // class
    Base
57
58      return 0;
59  }
60
```

11. 命令行展示 虚函数结构
    vs命令行工具--->进入文件目录--->输入
    `cl 文件.cpp /d1reportSingleClassLayoutDerive`

12. 使用命令行也解释了， call qword ptr [rax+8]

```
1  Derive::$vftable@:
2          | &Derive_meta
3          |  0
4   0      | &Base::show    // 重载在这里，看不出来，实际这个
    是 int
5   1      | &Derive::show  // 覆盖是 之前最上面的不要了，新
    的放到最下面
```

5.虚析构函数--重点在于什么呢时候用

1. 哪些函数不能实现成 虚函数?--接上一节
   虚函数能产生函数地址
   虚函数表位置在 vfptr里，vfptr在内存里----- 对象必须存在，**依赖对象**
   构造函数--不能是--虚函数，构造函数中调用虚函数，不会发生动态绑定，**构**
   造函数中调用的任何函数，都是静态绑定---->这是因为在构造函数执行期间，
   对象的动态类型尚未完全确定，虚函数表（vtable）也没有完全初始化。
   派生类构造过程⟶先调用基类构造⟶才调用⟶派生类构造
   static静态成员方法⟶不能是--->虚函数， 因为不依赖对象
   析构函数⟶可以!!⟶虚函数， 因为析构时,对象是存在的

2. **特别注意:基类析构和派生类虚构!!**
   基类虚构是虚函数-----> 派生类析构 自动成为虚函数---> 尽管名字不一
   样!!

3. 虚析构使用实例：
为什么 不是虚析构时，会出问题？因为使用了
因为此时是静态绑定，Base类的指针即使指向Derive类，使用的还是Base的析构

派生类使用虚析构，就有了虚函数表，派生类的虚函数表还会覆盖 基类 的 虚析构，使用自己的虚析构，使得可以 正确析构派生类，--->动态绑定
pb是Base类，但是指向Derive，动态绑定，使得本来是使用Base的析构，但是发现是虚析构，于是去找虚函数表，此时虚函数表的析构已经被 Derive覆盖了，因此使用了 Derive的析构， 后续继承基类的部分也会正确析构

```cpp
#include <iostream>
using namespace std;

class Base
{
public:
    Base(int data=10) :ma(data) { cout << "Base" << endl; }
    virtual ~Base() { cout << "~Base" << endl; }
    void show() { cout << "call Base:show" << endl; }

protected:
    int ma;
};

class Derive : public Base
{
public:
    Derive(int data=20) :Base(data), mb(data)
    {
        cout << "Derive" << endl;
    }
    ~Derive()
    {
        cout << "~Derive" << endl;
    }
    void show() {  }
private:
    int mb;
```

```
};

int main()
{
    Base* pb = new Derive(10);
    pb->show(); // 静态绑定,pb是Base*类型,  而*pb类型取决于show是不是虚函数, 若是, 则是Derive,因为指向了派生类的show, 且是动态绑定
    delete pb;  // 不使用虚析构, 会导致派生类析构没有调用->若有指针.容易内存泄漏



   /* Derive d(50);
    Base* pb = &d;
    pb->show(); */
    return 0;
}


```

4. **什么时候用?---特别重点**
   基类的指针(引用)指向 堆上new出来的 派生类, delete 基类指针时

6.再讨论虚函数和动态绑定

1. 虚函数的调用就是 动态绑定?
   不是!!!
   构造函数就是例外, 构造函数中调用虚函数, 不会发生动态绑定, **构造函数中调用的任何函数,都是静态绑定**
2. 什么时候发生动态绑定?

```
1.函数是虚函数：基类中的函数必须声明为 virtual。

2.通过 指针或引用 调用：通过基类指针或基类引用调用虚函数。

这是 必要条件!!!二者必须满足
```

3．实例:---- **一定要搞清楚什么时候动态绑定!!!**

```cpp
#include <iostream>
using namespace std;

class Base
{
public:
    Base(int data=10) :ma(data) { cout << "Base" << endl; }
    virtual ~Base() { cout << "~Base" << endl; }
    virtual void show() { cout << "call Base:show" << endl; }

protected:
    int ma;
};

class Derive : public Base
{
public:
    Derive(int data=20) :Base(data), mb(data)
    {
        cout << "Derive" << endl;
    }
    ~Derive()
    {
        cout << "~Derive" << endl;
    }
    void show() {  }
private:
    int mb;
```

```
};

int main()
{
    Base b;
    Derive d;
    b.show();  // 二者都是静态绑定,
    d.show();

    Base* pb1 = &b; //二者是动态绑定
    pb1->show();

    Base* pb2 = &d;
    pb2->show();

    Base& rb1 = b;//二者是动态绑定
    pb1->show();

    Base& rb2 = d;
    pb2->show();

    Derive* pd2 = (Derive*)&b;//动态绑定---满足两个条件-----这里必须强转, 回看第三节的总结
    pd2->show();//---这里是调用的基类的show, 因为实际的b是Base, 其虚函数表是基类的, 这种强转并不会 改变原本的虚函数表指向


    return 0;
}

```

7.理解多态到底是什么

1. 如何理解多态?
   静态的 多态: --- 编译阶段就确定好 --- **函数重载和模板(函数模板,类模板),**

动态的 多态： --- 继承结构中，基类指针(引用) 指向派生类对象，通过该指针(引用)调用同名覆盖方法(虚函数),基类指针指向哪个派生类，就调用哪个派生类对象的同名覆盖方法，称为多态

多态底层是通过 动态绑定来实现的的

2. 实例：虚函数配合基类，完成 开-闭 设计， 使用指针时切记虚析构

```cpp
#include <iostream>
using namespace std;

// 动物的基类
class Animal
{
public:
    Animal(string name) : _name(name) {}
    virtual void bark() {}
protected:
    string _name;
};

class Cat : public Animal
{
public:
    Cat(string name) : Animal(name) {}
    void bark() { cout << _name << " bark: miao miao!" << endl; }
};

class Dog : public Animal
{
public:
    Dog(string name) : Animal(name) {}
    void bark() { cout << _name << " bark: wang wang!" << endl; }
};

class Pig : public Animal
{
public:
    Pig(string name) : Animal(name) {}
```

```
32      void bark() { cout << _name << " bark: heng
    heng!" << endl; }
33  };
34
35
36  //下面的API无法达到 软件设计的 开-闭 原则: 对修改关闭, 对扩展
    开放
37  //void bark(Dog &dog)
38  //{
39  //    dog.bark();
40  //}
41  //
42  //void bark(Pig& pig)
43  //{
44  //    pig.bark();
45  //}
46  //
47  //void bark(Cat& cat)
48  //{
49  //    cat.bark();
50  //}
51
52
53  //使用基类指针
54  void bark(Animal *p)
55  {
56      p->bark(); // 虚函数 覆盖
57  }
58
59  int main()
60  {
61      Dog dog("dog");
62      Pig pig("pig");
63      Cat cat("cat");
64
65      bark(&dog);
66      bark(&cat);
67      bark(&pig);
68
69      return 0;
```

```
70 }
71
```

3. **继承的好处?**
   1.做代码复用
   2.在基类中给所有的派生类提供统一的虚函数接口，让派生类重写，然后就可以多态了

8.理解抽象类----纯虚函数

1. 类 是 抽象一个实体的类型
2. 什么是纯虚函数?
   纯虚函数（Pure Virtual Function）是 C++ 中用于定义抽象类的一种机制。纯虚函数在基类中声明但不提供实现，要求派生类必须重写（实现）该函数。包含纯虚函数的类称为**抽象类**，抽象类不能被实例化。

   在函数声明的末尾加上 `= 0`，表示这是一个纯虚函数
   `virtual void foo() = 0;`
3. 什么是 抽象类?
   拥有纯虚函数的类，叫做抽象类
   抽象类不能再实例化对象，但是可以定义指针和引用变量
   一般是基类作为抽象类，派生类去实例化对象
4. 实例： --- 注意 外部接口是 基类

```cpp
#include <iostream>
using namespace std;

// 动物的基类
class Animal
{
public:
    Animal(string name) : _name(name) {}
    virtual void bark() {}
protected:
    string _name;
};

class Cat : public Animal
```

```
{
public:
    Cat(string name) : Animal(name) {}
    void bark() { cout << _name << " bark: miao miao!" << endl; }
};

class Dog : public Animal
{
public:
    Dog(string name) : Animal(name) {}
    void bark() { cout << _name << " bark: wang wang!" << endl; }
};

class Pig : public Animal
{
public:
    Pig(string name) : Animal(name) {}
    void bark() { cout << _name << " bark: heng heng!" << endl; }
};


//下面的API无法达到 软件设计的 开-闭 原则: 对修改关闭, 对扩展开放
//void bark(Dog &dog)
//{
//    dog.bark();
//}
//
//void bark(Pig& pig)
//{
//    pig.bark();
//}
//
//void bark(Cat& cat)
//{
//    cat.bark();
//}
```

```
51
52
53  //使用基类指针
54  void bark(Animal *p)
55  {
56      p->bark(); // 虚函数 覆盖
57  }
58
59  int main()
60  {
61      Dog dog("dog");
62      Pig pig("pig");
63      Cat cat("cat");
64
65      bark(&dog);
66      bark(&cat);
67      bark(&pig);
68
69      return 0;
70  }
71
72
73
```

5. 实例-2:

```
1   #include <iostream>
2   #include <string>
3   using namespace std;
4
5   // 抽象基类 Car
6   class Car {
7   public:
8       Car(string name) : _name(name) {}
9       virtual double getMilesPerGallon() = 0; // 纯虚函
    数
10      string getName() { return _name; }
11      double getLeftMiles(double fuel) {
12          return fuel * getMilesPerGallon();
13      }
14  protected:
```

```
    string _name;
};

// 派生类 Audi
class Audi : public Car {
public:
    Audi(string name) : Car(name) {}
    double getMilesPerGallon() override {
        return 18.0;
    }
};

// 派生类 BMW
class BMW : public Car {
public:
    BMW(string name) : Car(name) {}
    double getMilesPerGallon() override { //override 是 C++11 引入的一个关键字，用于显式地标记派生类中的函数是对基类虚函数的重写。它的主要作用是提高代码的可读性和安全性，帮助开发者避免一些常见的错误。
        return 19.0;
    }
};

// 给外部提供一个统一的获取汽车剩余路程数的API
void showCarLeftMiles(Car &car, double fuel) {
    cout << car.getName() << " left miles: " << car.getLeftMiles(fuel) << " 公里" << endl;
}

int main() {
    Audi a("奥迪");
    BMW b("宝马");

    showCarLeftMiles(a, 10.0); // 假设有10加仑的油
    showCarLeftMiles(b, 10.0);

    return 0;
}
```

9.笔试问题讲解

1. 实例-1

重点：函数调用，参数压栈是在编译时期 就确定的

因此，派生类虚函数参数默认值 是用不到的

默认参数值是静态绑定的，而虚函数的调用是动态绑定的。即使 基类无,派生类有,也没用

```cpp
#include <iostream>
using namespace std;

class Base {
public:
    virtual void show(int i = 10) {
        cout << "call Base::show i:" << i << endl;
    }

    virtual ~Base() {} // 虚析构函数
};

class Derive : public Base {
public:
    void show(int i = 20) override {
        cout << "call Derive::show---" << i << endl;
    }
};

int main() {
    Base* p = new Derive(); // 使用基类指针指向派生类对象
    p->show(); // 动态绑定，调用Derive::show，但是输出却是 10，不是20
    delete p; // 释放内存
    return 0;
}

//为什么会是10 呢
从函数调用角度讲，先压参数列表，才压函数符号
```

```
而在编译阶段，看不到动态绑定，只能看到是Base*类，因此压入的是10
push 0Ah   编译时
mov eax, dword ptr[p]  运行时
mov ecx, deord ptr[eax]
call eax

push的是死的，不会因为后面调虚函数而改变


```

2. 实例-2

重点:成员的权限，是在编译阶段 确定好的!!!!!!

编译阶段只能看见 p是Base的，而基类里是 public的

千万不要去 运行时看 成员权限!!

```cpp
#include <iostream>
using namespace std;

class Base {
public:
    virtual void show() {
        cout << "call Base::show i:" << endl;
    }

    virtual ~Base() {} // 虚析构函数
};

class Derive : public Base {
private:
    void show() override {
        cout << "call Derive::show---"  << endl;
    }
};

int main() {
    Base* p = new Derive();
    p->show(); // 运行时确定
    delete p;
    return 0;
}
```

```
26
27
28
29
```

3. 实例-3

重点：**vfptr什么时候拿到虚函数表地址？ 是重点!!!**

每个类（无论是基类还是派生类）都有自己的虚函数表（vtable）。

每个对象（无论是基类对象还是派生类对象）都有自己的虚函数表指针（vfptr），指向其所属类的虚函数表。

```cpp
#include <iostream>
using namespace std;

class Base {
public:
    Base() {
        /*
    push ebp
    mov ebp, esp
    sub esp, 4Ch
    rep stos esp←→ebp     0xCCCCCCCC (windows VS GCC/G++)
    此时，就会进行  vfptr→vftable地址
    进入函数后，第一件事就是虚函数表指针的存储
    */
        cout << "call Base()" << endl;
        clear();
    }
    void clear()
    {
        memset(this, 0, sizeof(*this));
    }
    virtual void show() {
        cout << "call Base::show()" << endl;
    }

    virtual ~Base() {
        cout << "call ~Base()" << endl;
    }
```

```cpp
};

class Derive : public Base {
public:
    Derive() {
        cout << "call Derive()" << endl;
    }

    void show() override {
        cout << "call Derive::show()" << endl;
    }

    ~Derive() {
        cout << "call ~Derive()" << endl;
    }
};

int main() {
    //Base* pbl = new Base();
    //pbl->show(); // 动态绑定
    //delete pbl;  //这一段肯定会出错，vfptr是0了，肯定访问不到了

    Base* pb2 = new Derive();
    pb2->show(); // 动态绑定，
    delete pb2;  // 这一段是可以的，先基类构造，再派生类构造
    //涉及到了  vfptr什么时候得到的vftable的 地址，在构造函数里
    //将会把派生类虚函数地址写入vfptr，



    return 0;
}
```

继承与多态-多重继承

1.虚基类和虚继承

本节内容

1. 多重继承?
   代码复用，一个派生类 有多个基类
   抽象类---有纯虚函数的类
2. 虚基类
   virtual的两种修饰
   1. 修饰成员方法----叫做虚函数
   2. 修饰继承方式--->虚继承. 被虚继承的类，称为虚基类

```
1
2  class A { private: int ma; };
3  class B : virtual public A
4  {
5  private:
6      int mb;
7  };
```

3. 看一下虚基类的结构
   基类被虚继承后的 内存布局 如下： 多了 **vbptr 和 vbtable**，注意区别 vfptr与vftable
   virtual base 与 virtual func

```
1  class A size(4):
2          +---
3   0      | ma
4          +---
5
6
7
8
9  class B size(12):  ----  x86上
10         +---
```

```
 0       | {vbptr}       -----  重点 :  指向vbtable
 4       | mb
         +---
         +--- (virtual base A)   ----  这里开始是A
 8       | ma
         +---

B::$vbtable@:
 0       | 0
 1       | 8 (Bd(B+0)A)       ----- 重点, 相较于vbptr的偏移量


```

4. 根据这个结构，思考下面这个例子为什么出错?

```cpp
#include <iostream>
using namespace std;


class A
{
private:
    int ma;
public:
    virtual void func()
    {
        cout << "call A:func" << endl;
      }

    //添加一个 new重载, 看看 new和delete的一不一样
    void operator delete(void* ptr)
    {
        cout << "delete p: " << ptr << endl;
        free(ptr);

    }
};
class B : virtual public A
{
private:
```

```
    int mb;
public:
    void func()
    {
        cout << "call B:func" << endl;
    }
    //添加一个 new重载, 看看 new和delete的一不一样
    void* operator new(size_t size)
    {
        void* p = malloc(size);
        cout << "new p: " << p << endl;
        return p;
    }
};

int main() {

    // 基类指针指向派生类对象, 永远指向 派生类内存起始地址
    // 正式由于这个原因, 使得虚基类中, 派生类继承的虚基类, 在内存结构最下面, 起始是vbptr,  使得堆上释放会出错
    A *a= new B();
    a->func();  // 可以正确调用
    delete a;   // 但是释放有问题

    return 0;
}


/*
new p: 00BB6FA0
call B:func
delete p: 00BB6FA8
*/

//发现new和delete的不是一块地方, --- 编译器不同, 可能会没有这个问题
//vs不行, 但是linux的g++确是正确的

//实测发现 g++ (Ubuntu 9.4.0-1ubuntu1~20.04.2) 9.4.0 的g++ 貌似也不行,没搞明白
```

```
63
```

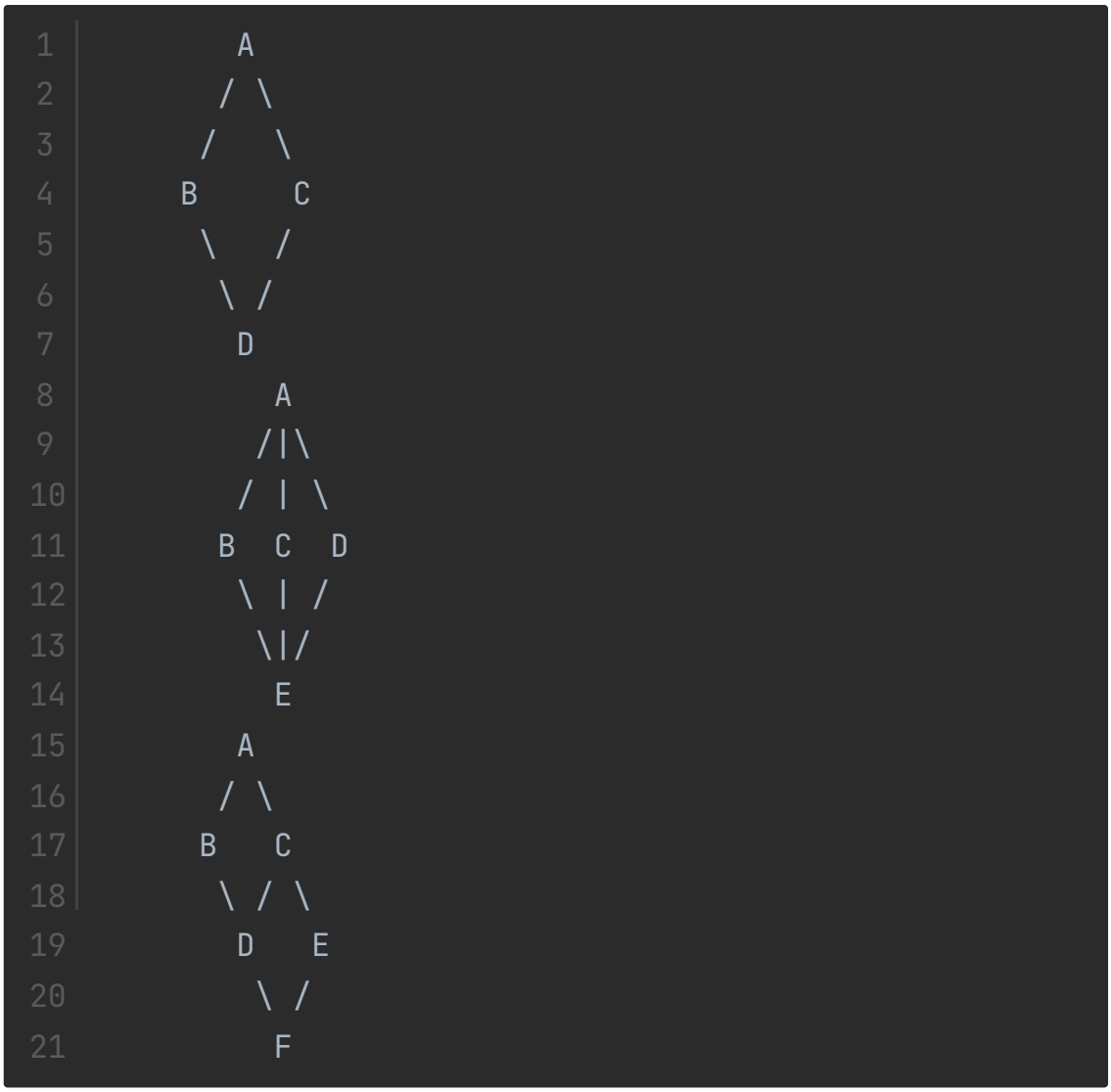
vs里不使用堆，就没有这个问题

```
B b;
A *a= &b;
a→func();
```

2.菱形继承---怎么解决?

本节内容

1. 菱形继承示意图



2. 菱形继承面临的问题
   第一个图中,在菱形继承中，**D** 会包含两份 **A** 的成员（通过 **B** 和 **C**），这可能导致二义性和资源浪费。
3. cpp开源代码,很少有 多重继承
4. 实例：重复构造，浪费资源

```cpp
#include <iostream>
using namespace std;


class A
{
private:
    int ma;
public:
    A(int data) :ma(data) { cout << "A()" << endl; }

    ~A() { cout << "~A()" << endl; }

};


class B : public A
{
private:
    int mb;
public:
    B(int data) :A(data), mb(data) { cout << "B()" << endl; }

    ~B() { cout << "~B()" << endl; }

};

class C : public A
{
private:
    int mc;
public:
```

```
    C(int data) :A(data), mc(data) { cout << "C()" << endl; }

    ~C() { cout << "~C()" << endl; }

};

class D : public B, public C
{
private:
    int md;
public:
    D(int data) :B(data),C(data), md(data) { cout << "D()" << endl; }

    ~D() { cout << "~D()" << endl; }

};

int main() {

    D d(10);

    return 0;
}



A()
B()
A()
C()
D()
~D()
~C()
~A()
~B()
~A()
```

使用虚继承解决， 继承的A全部换为虚继承

5. 特别注意:**由于B,C都是虚继承， D继承B,C时， A的vbtable是在D里面的，因此D里面必须对A构造，默认构造是不需要这样的，但是A里面写了 自定义带参构造，就需要写上了**

```cpp
#include <iostream>
using namespace std;


class A
{
private:
    int ma;
public:
    A(int data) :ma(data) { cout << "A()" << endl; }

    ~A() { cout << "~A()" << endl; }

};


class B : virtual public A
{
private:
    int mb;
public:
    B(int data) :A(data), mb(data) { cout << "B()" << endl; }

    ~B() { cout << "~B()" << endl; }

};

class C : virtual public A
{
private:
    int mc;
public:
    C(int data) :A(data), mc(data) { cout << "C()" << endl; }

    ~C() { cout << "~C()" << endl; }
```

```
36
37 };
38
39 class D : public B, public C  //B和C里面的vbptr存储的偏移量 是不同的
40 {
41 private:
42     int md;
43 public:
44     D(int data) :A(data), B(data),C(data), md(data) { cout << "D()" << endl; }
45
46     ~D() { cout << "~D()" << endl; }
47
48 };
49
50 int main() {
51
52     D d(10);
53
54     return 0;
55 }
56
57
58
59
60
61 A()
62 B()
63 C()
64 D()
65 ~D()
66 ~C()
67 ~B()
68 ~A()
```

面试问题： 怎么理解多重继承的?---重点

好处:可以更多的 代码复用
坏处： 菱形继承会出现 派生类 有多份 间接基类的数据 设计的问题

3.c++提供的四种类型转换

本节内容

1. c中类型强转:
   `int a = (int)b;`
2. cpp里的语言级别强转

```
const_cast:
    去掉常量属性的类型转换.

static_cast:
    能提供编译器认为安全的类型转换.---做有关联的类型转换

reinterpret_cast:
    类似于c风格的强转.

dynamic_cast:
    主要用于继承结构中, 可以支持RTTIL类型识别的上下转换
```

3. 代码:

```
#include <iostream>
using namespace std;


class A
{
public:
    virtual void func() { cout << "~A()" << endl; }

};

```

```
12
13  class B : public A
14  {
15
16  public:
17
18      void func() { cout << "~B()" << endl; }
19
20  };
21
22  class C : public A
23  {
24
25  public:
26
27      void func() { cout << "~C()" << endl; }
28
29      //添加新功能,
30      void funcss()
31      {
32          cout << "sssss" << endl;
33      }
34
35  };
36
37  void classShow(A* p) //指针若是C类型, 就调用funcss, 而不
    是func
38  {
39      //typeid(*p).name()=="C",  不用这个, 用
    dynamic_cast
40      //dynamic_cast会检查 p指针 是否指向的 是一个 C类型的对
    象
41      // p→vfptr→vftable RTTI信息, 如果是, dynamic转型
    成功,
42      // 返回C对象的地址给 c2, 否则返回 nullptr
43      //编译时期类型转换, 不能识别RTTI信息
44
45      C* c2 = dynamic_cast<C*>(p);    //static_cast在这
    里永远都能强转成功
46      if (c2 != nullptr)
```

```
47      {
48          c2->funcss();
49      }
50      else p->func();
51  }
52
53  int main() {
54
55      //const int a = 10;
56
 //*************************************************
*************************************************
*********
57      //const_cast
58      //int* p = (int*)&a;
59      //int* p2 = const_cast<int*>(&a);  //二者在汇编上
是一样的，但是const_cast不能跨类型转换，只能基本类型一样的转
换
60      /*
61       int* p = (int*)&a;
62  00D51FC7  lea         eax,[a]
63  00D51FCA  mov         dword ptr [p],eax
64
65      int* p2 = const_cast<int*>(&a);
66  00D51FCD  lea         eax,[a]
67  00D51FD0  mov         dword ptr [p2],eax
68      */
69
70      //double* p3 = (double*)&a;
71      //double* p4 = const_cast<double*>(&a);  // 这就
是不行的
72
 //*************************************************
*************************************************
***********
73      //static_cast----编译时期类型转换
74      //int a = 10;
75      //char b = static_cast<int>(a);
76      //int* p = nullptr;
```

```
    //short* c = static_cast<short*>(p); // 只能做有联系的类型转换
    /*
1. 什么是有关系的类型?

继承关系：例如基类指针和派生类指针。

基本类型的隐式转换：例如 int 到 double。

用户定义的类型转换：例如类中定义了转换运算符。

在这些情况下，static_cast 可以安全地使用。

2. 什么是无关的类型?

完全不同的指针类型：例如 int* 和 double*，char* 和 float* 等。

不相关的类类型：例如两个没有继承关系的类。
    */


 //*************************************************************************************************
    //reinterpret_cast  类似与c风格，啥都不管


 //*************************************************************************************************
    //dynamic_cast
    A a;
    B b;
    C c;
    classShow(&a);
    classShow(&b);
    classShow(&c);
    //结合classShow处看

```

```
109
110
111
112     return 0;
113 }
114
115
```

4. **什么是 RTTI?**

   RTTI 是 C++ 的一种特性，允许程序在运行时获取对象的类型信息。它主要由以下两部分组成：

   - `typeid` **运算符**：用于获取对象的类型信息。
   - `dynamic_cast` **运算符**：用于在继承层次中进行安全的类型转换。

   RTTI 需要编译器支持，并且在运行时需要额外的内存来存储类型信息。

STL组件

1.整体学习内容

```
一、标准容器  c++11里提供了array   forword_list
1. 顺序容器
vector
deque
list
2. 容器适配器
stack
queue
priority queue
3. 关联容器
无序关联容器  链式哈希表  增删查O(1)
unordered_set
unordered_multiset
unordered_map
unordered_multimap
有序关联容器  红黑树  增删查O(log2n)  2是底数（树的层数，树的高度)
set
multiset
map
multimap

二、近容器
数组, string, bitset(位容器)

三、迭代器
iterator和const iterator
reverse_iterator 和 const_reverse_iterator

四、函数对象(类似c的函数指针)
greater, less

五、泛型算法
sort,find,find_if,binary_search,for_each
```

2.vector容器

1. vector--- 向量容器
   底层数据结构--->动态开辟的数组，每次以原来空间大小的2倍进行扩容
2. 常用方法介绍：

```
vector<int> vec;
1. 增加
vec.push_back(20); //末尾添加元素-O(1)-可能导致容器扩容------回顾空间配置器allocator
vec.insert(it, 20); //指定位置增加元素--O(n)--因为后续的元素都要移动

2. 删除
vec.pop_back(); // 末尾删除-O(1)
vec.erase(it); //删除指定位置元素 O(n)

3.查询:
operator[]  下标随机访问: vec[5] -- O(1)
iterator迭代器遍历
find, for_each  --- 泛型查询
foreach ⟹ 通过迭代器实现的

注意: 对容器进行连续插入或者删除(insert,erase), 一定要更新迭代器, 否则会导致容器迭代器失效----回顾容器迭代器失效

常用方法:
1.size()
2.empty() //判断是否为空 , true(1)空, 0为非空
3.reserve(20) //给vector预留空间, 只开辟空间, 并不添加元素, 容器依然是空的, 元素是0, size()是0, empty()是1
4.resize(20)  //重置大小,扩容
5.swap: //交换两个元素

```

3. 代码：
   注意区分 reserve和resize 的区别
   注意 insert和erase的逻辑问题

```cpp
#include <iostream>
using namespace std;
#include <vector>

int main()
{
    vector<int> vec;

    //vec.reserve(20); // 预留空间
    //cout << vec.empty() << endl; // 1
    //cout << vec.size() << endl; // 0

    vec.resize(20); // 会放入元素0
    cout << vec.empty() << endl; // 0
    cout << vec.size() << endl; // 0

    for (int i = 0; i < 20; ++i)
    {
        vec.push_back(rand() % 100+1);
    }
    cout << vec.empty() << endl; //0   0
    cout << vec.size() << endl;//20    40


    #if 0
    int size = vec.size();

    for (int i = 0; i < size; ++i)
    {
        cout << vec[i] << " ";
    }
    cout << endl;

    //删除所有偶数
    auto it2 = vec.begin();
    //for (; it2 != vec.end(); ++it2)
    //{
    //  if (*it2 % 2 == 0)
    //  {
    //      vec.erase(it2); //删除全部, 需要更新迭代器
```

```
41      //        //break; //只删除一个，it2不用管了
42      //    }
43      //}
44      for (; it2 != vec.end(); )
45      {
46          if (*it2 % 2 == 0)
47          {
48              it2 = vec.erase(it2); //删除全部，需要更新迭代器
49              //break; //只删除一个，it2不用管了
50          }
51          else  // 注意逻辑问题
52          {
53              ++it2; //由于更新了，要判断一下当前位置
54          }
55      }
56
57
58
59      auto it = vec.begin();
60
61      for (; it != vec.end(); ++it)
62      {
63          cout << *it << " ";
64      }
65      cout << endl;
66
67      //给vector容器前所有奇数前面都添加一个小于奇数1的偶数
68      for (auto it1 = vec.begin(); it1 != vec.end(); ++it1)
69      {
70          if (*it1 % 2 != 0)
71          {
72              it1 = vec.insert(it1, *it1 - 1);
73              ++it1; // 注意逻辑问题
74          }
75      }
76      it = vec.begin();
77
78      for (; it != vec.end(); ++it)
```

```
    {
        cout << *it << " ";
    }
    cout << endl;

    #endif

    return 0;
}

```

3.deque和list

二者 比vector 多出来的 常用方法: push_front, pop_front

deque--双端队列容器

1. deque--双端队列容器--分块存储 --- 默认第二维开辟 4096/sizeof(int) = 1024个位置
   底层数据结构---> 动态开辟的二维数组，第一维数组个数从2开始，以2倍方式扩容，每次扩容后，原来第二维的数组，从新的第一维数组的下标oldsize/2开始存放，方便首尾元素添加

```
第一维：中央映射表（指针数组），存储指向各个缓冲区的指针。

第二维：每个缓冲区（固定大小的数组），存储实际的数据元素。
(std::deque（双端队列）的底层实现可以理解为一个动态的二维数组，但这种描述需要进一步澄清。实际上，deque的底层数据结构是一个分段连续的空间，由多个固定大小的数组（称为缓冲区或块）组成，并通过一个中央映射表（通常是一个指针数组）来管理这些缓冲区。)---deepseek

中央映射表（Map）
+---+---+---+---+---+---+---+
|   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+
  |       |       |       |
  v       v       v       v
+---+  +---+  +---+  +---+
```

```
| B | | B | | B | | B |  ← 缓冲区 (Buffer)
+---+ +---+ +---+ +---+
  |     |     |     |
  v     v     v     v
+---+ +---+ +---+ +---+
| 1 | | 2 | | 3 | | 4 |  ← 缓冲区中的元素
| 2 | | 3 | | 4 | | 5 |
|...| |...| |...| |...|
+---+ +---+ +---+ +---+
```

一般first和last是在最中间，因为是双端队列，两边都能加元素
扩容后，复制过去的旧数据也将会在中间位置(oldsize/2--用于计算放入的位置)
2→4 2/2=1 0,1,2,3 放在1,2处
4→8 4/2=2 0,1,2,3,4,5,6,7 放到 2,3,4,5处

2．常用方法：

```cpp
#include<deque>
deque<int> deq;
1. 增加
deq.push_back(20); // last 末尾添加 - O(1)
deq.push_front(20);  // first 从首部添加 - O(1)
//vec.insert(vec.begin(), 20) - O(n)
deq.insert(it, 20); // O(n)

2.删除
deq.pop_back(); //O(1)
deq.pop_front(); //O(1)
deq.erase(it); //O(n)

3.查询搜索
iterator(连续的insert和erase一定要考虑迭代器失效)
无 operator[]
```

list--链表容器

1. list--链表容器
   底层数据结构--双向的循环链表 （pre,data,next）
2. 常用方法：

```
#include<list>
deque<int> mylist;
1. 增加   --- 与deque 一模一样, 除了insert时间复杂度
mylist.push_back(20); // last 末尾添加 - O(1)
mylist.push_front(20);  // first 从首部添加 - O(1)
//vec.insert(vec.begin(), 20) - O(n)
mylist.insert(it, 20); // O(1)  --- 不涉及其他元素  //但是链表进行insert前 需要进行 query查询, 链表查询效率低

2.删除
mylist.pop_back(); //O(1)
mylist.pop_front(); //O(1)
mylist.erase(it); //O(1)

3.查询搜索
iterator(连续的insert和erase一定要考虑迭代器失效)
无 operator[]
```

4.vector,deque,list对比

主要内容

1. 不要只学习表面，多看看底层
2. 特点回顾

```
1.vector特点:
    动态数组,内存是连续的, 2倍的方式进行扩容, vector<int> vec; reserve和resize区别

2.deque特点:
    动态开辟的二维数组, 第一维数组个数从2开始, 以2倍方式扩容, 每次扩容后, 原来第二维的数组, 从新的第一维数组的下标oldsize/2开始存放, 方便首尾元素添加


```

面经问题

1. deque的底层内存是不是连续的? -- 分块存储
   并不是, 每一个 第二维的是 连续的, 但是 第二维之间 不是连续的
2. vector 与 deque 区别?

```
1. 底层数据结构不同
2. 前中后 插入删除元素的时间复杂度: 中间O(n)和结尾O(1)相同, 但是 前不同, deque-O(1) vector-O(n)
3. 对于内存的使用效率, vector需要的内存是连续的, deque 可以分块 进行数据存储, 不需要内存空间 必须连续
4.在中间进行insert或erase, vector和deque他们的效率谁能好一点?
//虽然都是 都在一个量级O(n)   vetor更好, deque差
//由于deque的第二维空间不是连续的, 所以在deque中间进行元素的insert或者earse.造成元素移动要慢,

```

3. vector 与 list 区别?

```
1.  底层数据结构不同  list是双向循环链表
//一般的数组和链表,  数组:增加删除O(n), 查询O(n), 随机访问(1)
//链表, 增删本身是O(1), 但是还要查询,O(n), 没有随机访问


```

5.详解容器适配器--stack, queue, priority_queue

容器适配器

1. 注意区别 容器适配器和容器空间配置器

```
stack, queue, priority_queue之所以叫做适配器，是因为它们没有自己底层的数据结构，是依赖另外的容器来实现的功能，它们的方法，也是通过它们依赖的容器相应的方法实现的。
```

2. 怎么理解适配器？ --**有一种设计模式就是适配器模式**
   底层没有自己的数据结构，他是对另外容器的封装，他的方法 全部由 底层依赖 的容器 进行实现

```
stack  源码，底层用的就是deque
_EXPORT_STD template <class _Ty, class _Container = deque<_Ty>>

```

3. **容器适配器:stack, queue, priority_queue** ---- 重点，使用频率高

stack-栈

1. 常用方法: ===>依赖deque

```
1. push---入栈
2. pop--出栈
3. top--查看栈顶元素
4. empty--判空
5. size--返回元素个数

```

2. 代码:

```
#include <iostream>
using namespace std;
```

```cpp
#include <vector>
#include  <stack>




int main()
{
    stack<int> s1;

    for (int i = 0; i < 20; ++i)
    {
        s1.push(rand() % 100 + 1);
    }

    cout << s1.size() << endl;

    while (!s1.empty())
    {
        cout << s1.top() << " ";
        s1.pop();
    }

    return 0;
}

```

queue-队列

1. 常用方法: fifo 先入先出 ===>依赖deque

```
1. push---入队列
2. pop--出队列，队头出
3. front--查看队头
4. back--查看队尾
5. empty--判空
6. size--返回元素个数

```

priority_queue-优先级队列

1. 常用方法：　　===>依赖vector　　默认大根堆

```
1. push---入队列
2. pop--出队列
3. top--查看队顶元素
4. empty--判空
5. size--返回元素个数

```

总结

1. 代码：

```
#include <iostream>
using namespace std;
#include  <stack>
#include <queue>



int main()
{
    stack<int> s1;
```

```cpp

    for (int i = 0; i < 20; ++i)
    {
        s1.push(rand() % 100 + 1);
    }

    cout << s1.size() << endl;

    while (!s1.empty())
    {
        cout << s1.top() << " ";
        s1.pop();
    }
    cout << endl;
    cout << "----------------------------------------
--" << endl;

    queue<int> qe;
    for (int i = 0; i < 20; ++i)
    {
        qe.push(rand() % 100 + 1);
    }

    cout << qe.size() << endl;

    while (!qe.empty())
    {
        cout << qe.front() << " ";
        qe.pop();
    }

    cout << endl;
    cout << "----------------------------------------
--" << endl;

    priority_queue<int> pqe;
    for (int i = 0; i < 20; ++i)
    {
        pqe.push(rand() % 100 + 1);
    }
```

```
50
51      cout << pqe.size() << endl;
52
53      while (!pqe.empty())
54      {
55          cout << pqe.top() << " ";
56          pqe.pop();
57      }
58
59      return 0;
60  }
61
```

2. 为什么有的依赖 deque(queue, strack), 有的依赖 vector(priority_queue)

```
1  1. 为什么选择deque?
2      首先vector初始内存使用效率低, 没有deque好, vector是 0-1-2-4-8慢慢扩容, deque则是先开辟好大的,    虽然vector有reserve函数, 但是有修改成本
3      其次, queue需要支持 尾部插入, 头部删除, 因此在这两个操作上,需要时间复杂度要求, 而deque正好是O(1), vector却是O(n)和O(1)
4      vector需要大片的连续内存, 而deque只需要分段内存, 当存储大数据时, deque内存利用率更高
5
6  2. 为什么选择vector?
7      priority_queue 底层默认是大根堆结构, 使用 类似奇数和偶数下标的形式(了解二叉树应该明白这个), 来查找访问元素.    就像二叉树 是用数组存储的, 下标很重要----------deque则不行, 第二维的数组,不同的块, 内存都不是连续的
8
9
10
```

6.无序关联容器

关联容器

关联容器分为：无序和有序 关联容器
集合set，映射表map

1. 以**链式哈希表作为底层数据结构**的**无序关联容器**有：： --- **unordered_...**
   增删查--O(1)

```
unordered_set    单重集合    单重就是不允许数重复
unordered_multiset   多重集合    //
#include<unordered_set>
unordered_map
unordered_multimap          // #include<unordered_map>


```

2. 以**红黑树作为底层数据结构**的有序关联容器有：
   增删查--O(log2 n) 2是底数,树的高度

```
set
multiset     // #include<set>
map
multimap     // #include<map>
```

3. 关联容器 与 vector,deque, list的 函数使用注意点
   与 vector,deque, list不同,这些的insert是两个参数，因为是线性表，需要指定位置
   但是 由于关联容器 底层是 哈希表 或 红黑树，插入的位置不是随机的,一个是按哈希公式，一个是根据红黑树性质

```
unordered_set<int> set1;

set1.insert(20)

```

unordered_set

1. **切记：单重集合不存储 重复元素!!!!!!!!!!!!!!**
2. find()------有则返回迭代器，不存在则返回末尾迭代器
3. c++11 的 foreach 正规名叫 **基于范围的 `for` 循环** (range-based for loop)
4. 关联容器常用方法：

```
增: insert
删: erase(key), erase(it)  --- key和迭代器都行
遍历: iterator, find(key)
```

5. 代码：

```cpp
#include <iostream>
#include <unordered_set>
#include <string>
using namespace std;

int main()
{
    // 不允许key重复 改成unordered_multiset自行测试
    unordered_set<int> set1;    // ---- 注意只有一个参数
    for (int i = 0; i < 100; ++i)
    {
        set1.insert(rand() % 100 + 1);
    }
    cout << set1.size() << endl;   // 65, 不是100, 单重集合不存储 重复元素

/*============================================================================*/

    unordered_multiset<int> mulset1;    // ---- 注意只有一个参数
    for (int i = 0; i < 100; ++i)
    {
        mulset1.insert(rand() % 100 + 1);
    }
```

```
    cout << mulset1.size() << endl;   // 100


/*=============================================================*/

    auto it1 = set1.begin();
    for (;it1 != set1.end();++it1)
    {
        cout << *it1 << " ";
    }
    cout << endl;

    //c++11有 foreach形式用于遍历
    for (auto v : set1)
    {
        cout << v << " ";
    }
    cout << endl;

/*=============================================================*/

    set1.erase(20);  //按key值删除元素


/*=============================================================*/

    // 寻找是否存在20并删除
    it1 = set1.find(20);   // 有则返回迭代器，不存在则返回末尾迭代器
    if (it1 != set1.end())
    {
        set1.erase(it1);
    }

    // count返回set中有几个50=》最多是1个
    cout << set1.count(50) << endl;
    return 0;
```

```
55 }
```

unordered_map

1. map是存储键值对[key, val], set只存储 key
2. first→key, second→val   ===> 依赖于 pair 类
3. operator[] 要注意, 看代码
4. 代码:

```
1  #include <iostream>
2  #include <unordered_map>
3  #include <string>
4  using namespace std;
5
6  int main()
7  {
8      // 定义一个无序映射表
9      unordered_map<int, string> map;
10     // 无序映射表三种添加[key,value]的方式
11     map.insert({ 1000, "aaa" });  // 注意打包键值对
12     map.insert(make_pair(1001, "bbb"));
13     map[1002] = "ccc"; // operator[]   添加
14
15     //删除
16     map.erase(1002);
17
18     //查询
19     cout << map.size() << endl;  //2
20     cout << map[1000] << endl;
21     cout << map[1003] << endl;   // []重载,不仅能查询,而且key不存在时, 会添加这个键值对, string() , 实际打印出来就什么也没有, []还能修改
22     cout << map.size() << endl;  //3
23
24
25     // 遍历map表1
26     auto it = map.begin();    // 迭代器是 打包的 pair对象
27     for (; it != map.end(); ++it)
```

```
    {
        cout << "key:" << it->first << " value:" << it->second << endl;
    }
    // 遍历map表2
    for (pair<const int, string>& pair : map)   // 这里是const, map里key不可修改, 但是, 实际可以用auto更方便
    {
        cout << "key:" << pair.first << " value:" << pair.second << endl;
    }

    //// 查找key为1000的键值对, 并且删除
    //auto it1 = map.find(1000);
    //if (it1 != map.end())
    //{
    //  map.erase(it1);
    //}

    return 0;
}
```

应用

1. **无序map的一个应用：海量数据查重**

```
#include <iostream>
#include <unordered_map>
#include <string>
using namespace std;

int main()
{
    const int ARR_LEN = 100;
    int arr[ARR_LEN] = { 0 };
    for (int i = 0; i < ARR_LEN; ++i)
    {
        arr[i] = rand() % 20 + 1;
    }

```

```
    unordered_map<int, int> map1;
    for (int k : arr)
    {
        //auto it = map1.find(k);
        //if (it == map1.end())
        //{
        //  map1.insert({ k,1 });
        //}
        //else
        //{
        //  it->second++;  //出现过，就增加次数
        //}
        map1[k]++; // 初始是[k,int()]即[k,0]
    }

    for (auto& pair : map1)
    {
        if (pair.second > 1)
        {
            cout << "key: " << pair.first << "
count: " << pair.second << endl;
        }
    }


    return 0;
}
```

2. set应用，去重代码：

```
#include <iostream>
#include <unordered_map>
#include <string>
#include <unordered_set>
using namespace std;

int main()
{
    const int ARR_LEN = 100;
    int arr[ARR_LEN] = { 0 };
    for (int i = 0; i < ARR_LEN; ++i)
```

```
    {
        arr[i] = rand() % 20 + 1;
    }

    //去重打印
    unordered_set<int> set1;

    for (int k : arr)
    {
        set1.insert(k);
    }

    for (int v : set1)
    {
        cout << v << " ";
    }
    cout << endl;


    return 0;
}
```

3. 哈希桶？ -- 不是很明白这个到底是啥

```
#include <iostream>
#include <string>
#include <unordered_map>
int main()
{
    std::unordered_map<std::string, std::string>
mymap =
    {
        { "house", "maison" },
        { "apple", "pomme" },
        { "tree", "arbre" },
        { "book", "livre" },
        { "door", "porte" },
        { "grapefruit", "pamplemousse" }
    };
    unsigned n = mymap.bucket_count(); //获取哈希桶的个
数
```

```
    std::cout << "mymap has " << n << " buckets.\n";
    for (unsigned i = 0; i < n; ++i) // 逐个遍历哈希桶中的链表
    {
        std::cout << "bucket #" << i << " contains: ";
        for (auto it = mymap.begin(i); it ≠ mymap.end(i); ++it)
            std::cout << "[" << it→first << ":" << it→second << "] ";
        std::cout << "\n";
    }

    return 0;
}
```

7.有序关联容器 - set,map

1. 单重set，不重复元素，但有序
2. 常用方法 与 unordered 一模一样

set

1. 自定义类型呢？需要手动 提供自定义类型的 operator<.

```
//不加入自己的 重载时，会报以下错误
//二进制“<”:“const _Ty”不定义该运算符或到预定义运算符可接收的类型的转换


#include <iostream>
#include <string>
#include <set>
using namespace std;

class Student
{
```

```
public:
    Student(int id, string name): _id(id), _name(name){}
    bool operator< (const Student& stu)const  //不修改成员变量，只访问，尽量写常方法
    {
        return _id < stu._id;
    }
private:
    int _id;
    string _name;
    friend ostream& operator<< (ostream& out, const Student& stu);
};

ostream& operator<< (ostream& out, const Student& stu)
{
    out << "id: " << stu._id << " name: " << stu._name << endl;
    return out;
}

int main()
{
    set<Student> set1;
    set1.insert(Student(1001, "hzh2"));
    set1.insert(Student(1000, "hzh1"));

    for (auto v : set1)
    {
        cout << v ;
    }
    cout << endl;


    return 0;
}
```

map

1. 代码：

```cpp
#include <iostream>
#include <string>
#include <set>
#include <map>
using namespace std;

class Student
{
public:
    Student(int id=0, string name="hzh") : _id(id), _name(name){}

private:
    int _id;
    string _name;
    friend ostream& operator<< (ostream& out, const Student& stu);
    friend ostream& operator<< (ostream& out, const pair<int, Student>& p);
};

ostream& operator<< (ostream& out, const Student& stu)
{
    out << "id: " << stu._id << " name: " << stu._name << endl;
    return out;
}

ostream& operator<< (ostream& out, const pair<int, Student>& p)
{
    out << "id: " << p.second._id << " name: " << p.second._name << endl;
    return out;
}

```

```cpp
int main()
{
    map<int, Student> map1;    // map按key就可以自动排
    map1.insert({ 1001, Student(1001, "hzh1") });
    map1.insert({ 1000, Student(1000, "hzh0") });
    cout << map1[1000] << endl;    // 这样的operator[] 需要有默认的构造函数，不使用[]，是可以不需要 默认构造函数的

    for (auto& v : map1)
    {
        cout << v ;    // 对应第二个<<重载
    }
    cout << endl;

    //还有迭代器方式
    for (auto it = map1.begin(); it != map1.end(); ++it)
    {
        cout << "key: " << it->first << "value: " << it->second << endl; //后面的用到了<<重载
    }


    return 0;
}
```

pair注意

1. 元素访问，可以是 it->first， 也可以是 (*it).first， 一般使用第一个，更方便

8.迭代器iterator

iterator, const_iterator, reverse_iterator, const_reverse_iterator

1. iterator 是 普通的 正向迭代器 ---- 不仅能读,还能修改

```cpp
#include <iostream>
#include <string>
#include <vector>
using namespace std;


int main()
{
    vector<int> vec;
    for (int i = 0; i < 20; ++i)
    {
        vec.push_back(rand() % 100 + 1);
    }

    //vector<int>::iterator it = vec.begin();  // 这个要会
    auto it = vec.begin();

    for (; it != vec.end(); ++it)
    {
        cout << *it << " ";
        if (*it % 2 == 0)
        {
            *it = 0;
        }
    }
    cout << endl;

    it = vec.begin();
    for (; it != vec.end(); ++it)
    {
        cout << *it << " ";
    }

    return 0;
}
```

2. const_iterator常量的正向迭代器 ----- 只能读，不能写

```cpp
#include <iostream>
#include <string>
```

```cpp
#include <vector>
using namespace std;


int main()
{
    vector<int> vec;
    for (int i = 0; i < 20; ++i)
    {
        vec.push_back(rand() % 100 + 1);
    }

    vector<int>::const_iterator it = vec.begin();
 // 这个要会

    for (; it != vec.end(); ++it)
    {
        cout << *it << " ";
        //if (*it % 2 == 0)
        //{
        //  *it = 0;
        //}
    }
    cout << endl;

    it = vec.begin();
    for (; it != vec.end(); ++it)
    {
        cout << *it << " ";
    }

    return 0;
}
```

3. 为什么 iterator转化为const_iterator是对的?

```
vector<int>::const_iterator it = vec.begin();

实际上, 源码里, const_iterator是iterator的基类

class const_iterator
```

```
{
public:
    const T& operator*() { return *_ptr; }
};

class iterator : public const_iterator
{
public:
    T& operator*() { return *_ptr; }
};
```

4. reverse_iterator 反向迭代器， **搭配rbegin()**
   同样有 const_reverse_iterator

```
rbegin()返回的最后一个元素的 反向迭代器
rend()返回的首元素前驱位置的 反向迭代器

#include <iostream>
#include <string>
#include <vector>
using namespace std;


int main()
{
    vector<int> vec;
    for (int i = 0; i < 20; ++i)
    {
        vec.push_back(rand() % 100 + 1);
    }

    vector<int>::reverse_iterator it = vec.rbegin();
  // 这个要会

    for (; it != vec.rend(); ++it)   // 这里还是++
    {
        cout << *it << " ";
        //if (*it % 2 == 0)
        //{
        //  *it = 0;
```

```
        //}
    }
    cout << endl;



    return 0;
}
```

9.函数对象--仿函数

1. 函数对象⟶ 就是 c里面的 函数指针
   函数对象（Function Object），也称为**仿函数**（Functor），是 C++ 中的一个重要概念。它是一个类或结构体，通过重载 `operator()` 运算符，使得该类的对象可以像函数一样被调用。
2. 对比下面两个：

```
int sum(int a, int b)
{
    return a + b;
}

int ret = sum(10, 20);
```

```
class Sum
{
public:
    int operator() (int a, int b)
    {
        return a + b;
    }
};

Sum sum;
int ret = sum(10, 20);
```

3. 关于内联和函数指针代码：

```cpp
#include <iostream>
using namespace std;

template<typename T>
bool mygreater(T a, T b)
{
    return a > b;
}

template<typename T>
bool myless(T a, T b)
{
    return a < b;
}

// compare是C++的库函数模板
template<typename T, typename Compare>
bool compare(T a, T b, Compare comp)
{
    // 通过函数指针调用函数，是没有办法内联的，效率很低，因为有函数调用开销
    return comp(a, b);
}

int main()
{
    cout << compare(10, 20, mygreater<int>) << endl;
    cout << compare(10, 20, myless<int>) << endl;
    return 0;
}
```

```
这段代码里，如果把 myless和mygreater 换成内联函数，编译阶段是comp识别不了用哪个函数的
因为 这是使用函数指针间接调用的，运行时才会去找

下面这个是可以识别的，编译阶段会展开
inline bool func()
{
    ...;
}
bool compare(T a, T b, Compare comp)
{
    func();
    return comp(a, b);
}
```

4. 使用函数对象（仿函数）解决函数指针调用开销问题

```
#include <iostream>
using namespace std;


//c++函数对象的版本
template<typename T>
class mygreater
{
public:
    bool operator() (T a,T b)  // ()重载的两个参数叫做二元函数对象，一个参数就叫做一元函数对象
    {
        return a > b;
    }
};

template<typename T>
class myless
{
public:
    bool operator() (T a, T b)
    {
```

```
        return a < b;
    }
};

template<typename T, typename Compare>
bool compare(T a, T b, Compare comp)
{
    // 通过函数指针调用函数，是没有办法内联的，效率很低，因为有函数调用开销
    return comp(a, b);
}

int main()
{
    cout << compare(10, 20, mygreater<int>()) << endl;
    cout << compare(10, 20, myless<int>()) << endl;
    return 0;
}
```

5. **函数对象好处**

```
1. 通过函数对象调用operator()，可以省略函数的调用开销，比通过函数指针调用函数(不能使用内联)效率高
2. 因为函数对象是用类生成的，所以可以添加 相关的 成员变量，用于记录函数对象使用时的更多信息


```

6. priority_queue默认是大根堆，即从大到小排列，改为小根堆

```
#include <iostream>
#include <queue>
#include <vector>
using namespace std;

int main() {
    // 最大堆---默认的
    priority_queue<int> maxHeap;
    for (int i = 0; i < 10; ++i) {
        maxHeap.push(rand() % 100);
```

```
    }
    cout << "Max Heap: ";
    while (!maxHeap.empty()) {
        cout << maxHeap.top() << " ";
        maxHeap.pop();
    }
    cout << endl;

    // 最小堆  -- 看一下源代码参数，改一下less
    //template <class _Ty, class _Container =
vector<_Ty>, class _Pr = less<typename
_Container::value_type>>
    using MinHeap = priority_queue<int, vector<int>,
greater<int>>;
    MinHeap minHeap;
    for (int i = 0; i < 10; ++i) {
        minHeap.push(rand() % 100);
    }
    cout << "Min Heap: ";
    while (!minHeap.empty()) {
        cout << minHeap.top() << " ";
        minHeap.pop();
    }
    cout << endl;

    return 0;
}
```

同理，set也行
stl里 这种一般都是 less 和 greater

7. using

```
using 是 C++ 中一个非常强大的关键字，主要用途包括：

类型别名：定义类型别名，类似于 typedef。

模板别名：为模板定义别名。

命名空间成员引入：引入命名空间中的特定成员。

```

```
命名空间整体引入：引入整个命名空间。

基类成员引入：在派生类中引入基类的成员。

构造函数继承：继承基类的构造函数。

模板中使用：在模板中定义类型别名。

函数中使用：在函数内部定义类型别名。
```

10.泛型算法和绑定器

1. 泛型算法头文件

```
#include <algorithm>
```

2. 泛型算法特点： --- c++ primer书里有很多 泛型算法

```
1. 接收的都是 迭代器
    sort, find, find_if:有条件的find, binary_search:二分查找, for_each

2. 还能接受函数对象

3.模板实现的+迭代器+函数对象
```

3. 绑定器：

```
bind1st: 把二元函数对象的operator()的第一个形参绑定起来
bind2nd:把二元函数对象的operator()的第二个形参绑定起来

#include <functional>  包含函数对象和绑定器
```

绑定器+二元函数对象⟹一元函数对象

4. 代码：

```
#include <iostream>
#include <vector>
```

```cpp
#include <algorithm>
#include <functional>
using namespace std;

int main() {
    int arr[] = { 1, 2, 5, 4, 3 };
    size_t size = sizeof(arr) / sizeof(arr[0]); // 计算数组大小

    // 使用范围构造函数将数组元素放入 vector
    vector<int> vec(arr, arr + size);

    // 输出 vector 中的元素
    for (int val : vec) {
        cout << val << " ";
    }
    cout << endl;

    sort(vec.begin(), vec.end());
    // 输出 vector 中的元素
    for (int val : vec) {
        cout << val << " ";
    }
    cout << endl;

    if (binary_search(vec.begin(), vec.end(), 5))
    {
        cout << "5 is yes" << endl;
    }

    //从大到小
    sort(vec.begin(), vec.end(), greater<int>()); // 这个可比自己写快多了
    for (int val : vec) {
        cout << val << " ";
    }
    cout << endl;

    //有序的容器，使用二分查找是更快的   log2 n    二分查找默认是在升序的容器里找
```

```
    if (binary_search(vec.begin(), vec.end(), 5, greater<int>()))
    {
        cout << "5 is yes" << endl;
    }

    /*
    _EXPORT_STD template <class _FwdIt, class _Ty, class _Pr>
_NODISCARD _CONSTEXPR20 bool binary_search(_FwdIt _First, _FwdIt _Last, const _Ty& _Val, _Pr _Pred) {
    // test if _Val equivalent to some element
    _STD _Adl_verify_range(_First, _Last);
    auto _UFirst      = _STD _Get_unwrapped(_First);
    const auto _ULast = _STD _Get_unwrapped(_Last);
    _UFirst           = _STD lower_bound(_UFirst, _ULast, _Val, _STD _Pass_fn(_Pred));
    return _UFirst != _ULast && !_Pred(_Val, *_UFirst);
}

_EXPORT_STD template <class _FwdIt, class _Ty>
_NODISCARD _CONSTEXPR20 bool binary_search(_FwdIt _First, _FwdIt _Last, const _Ty& _Val) {
    // test if _Val equivalent to some element
    return _STD binary_search(_First, _Last, _Val, less<>{});
}
    */

    // 使用find()  ,, 二分效率高
    auto it = find(vec.begin(), vec.end(), 4);
    if (it != vec.end())
    {
        cout << "4 is yes--find" << endl;
    }


    // find_if 有条件的find,  一元函数对象   greater和less是二元函数对象
```

```cpp
    // 将4插入到vec里，找第一个小于 4的(降序)
    // 使用绑定器   找第一个小于4的
    //greater表示大于，则绑定第一个为4  即 4>b
    //less表示小于，则绑定第二个为4  即 a<4


    /*auto it2 = find_if(vec.begin(), vec.end(), bind1st(greater<int>(), 4));
    vec.insert(it2, 4);*/

    auto it2 = find_if(vec.begin(), vec.end(), bind2nd(less<int>(), 4));
    vec.insert(it2, 4);
    for (int val : vec) {
        cout << val << " ";
    }
    cout << endl;

    //c++11提供了 比绑定器和函数对象更简便的   lamda表达式---底层就是函数对象
    // []表示捕获外部变量,val就是捕获的 bool是返回值类型
    auto it3 = find_if(vec.begin(), vec.end(), [](int val)->bool {return val < 6; });
    vec.insert(it3, 6);
    for (int val : vec) {
        cout << val << " ";
    }
    cout << endl;

    //for_each 可以遍历容器所有元素，可以自行添加合适的元素对象，可以过滤元素
    //  打印偶数
    for_each(vec.begin(), vec.end(), [](int val)->void
        {
            if (val % 2 == 0)
            {
                cout << val << " ";
            }
        });
```

```
106     cout << endl;
107
108
109     return 0;
110 }
```