C++基础知识精髓

- 大家好，我是Linux兵工厂，在工作经常发现小伙伴们遇到一些C++的问题都是对基础知识不熟悉或理解混乱所导致的。正所谓万丈高楼平地起，作为一名合格的程序员来说，没有良好的基本功很难达到一定的高度。而工作中大部分编程问题都是基本功不扎实所导致，所以决定花些时间来整理C++相关的基本知识和基本概念供大家参考理解，每一个知识点都结合相关的代码进行验证。本文基本上涵盖了C++最常用的知识点，希望对小伙伴们有所帮助。 关注公众号:Linux兵工厂，我会不定时输出更多干货知识。
- **不因虚度年华而悔恨，不因碌碌无为而羞耻**
- **不积硅步，无以至千里；不积小流，无以成江海**
- **关注我，公众号：Linux兵工厂，后台回复C++获得更多实例代码，获取更多干货知识（Linux、网络、驱动、C/C++、Qt、Python等）**

1. C++是一种面向对象的程序设计语言

- C++支持数据封装，支持数据封装就是支持数据抽象。在C++中，类是支持数据封装的工具，对象则是数据封装的实现。面向过程的程序设计方法与面向对象的程序设计方法在对待数据和函数关系上是不同的。在面向对象的程序设计中，将数据和对该数据进行合法操作的函数封装在一起作为一个类的定义，数据将被隐藏在封装体中，该封装体通过操作接口与外界交换信息。对象被说明具有一个给定类的变量， 类 类似于C语言中的结构，在C语言中可以定义结构，但这种结构中包含数据，而不包含函数。C++中的类是数据和函数的封装体。在C++中，结构可作为一种特殊的类，它虽然可以包含函数，但是它没有私有或受保护的成员。
- C++类中包含 私有 、 公有 和 受保护成员 ， C++类中可定义三种不同访控制权限的成员。一种是私有(Private)成员，只有在类中说明的函数才能访问该类的私有成员，而在该类外的函数不可以访问私

有成员；另一种是公有(Public)成员，类外面也可访问公有成员，成为该类的接口；还有一种是保护 (Protected)成员，这种成员只有该类的派生类可以访问，其余的在这个类外不能访问。

2. 命名空间

- c++所有标志符都在std命名空间作用域中才可见

```
using std::cout;
using std::endl; 或 using namepace std;
using std::cin;
```

- 命名空间中可以嵌套定义命名空间

```
namespace name_3
{
    namespace
    {
       int k = 200;
    };
};
```

- 命名空间取别名

```
namespace name = name_3;
```

3. C++中的struct结构体

- 对比C语言中结构体，C++中结构体不仅可以有变量还可以有函数。

例程中声明一个命名空间Test,Test中声明一个结构体Account，而Account中定义变量和声明函数。

```
namespace Test
{
    struct Account
    {
        char name[30];
        double balance;
        void init(char *myname,double mybalance)
        {
            strcpy(name,myname);
            balance = mybalance;
        }
        void deposit(double amount);
        void withdraw(double amount);
        inline double getBalance()
        {
            return balance;
        }
    };
}
```

4. Const属性修改

- 在c中const的意思是“一个不能被改变的普通变量“，在c中它总是占用存储空间而且它的名字是全局符。
- c++编译器通常并不为const分配空间，它把这个定义保存在符号表中。当const常量被使用时，编译的时候就进行 常量折叠 。
  c++中 编译器不会为一般的const常量分配内存空间, 而是将它们存放符号表中。如果取了这个常量的地址,那么编译器将为此常量分配一个内存空间,生成一个常量副本, 所有通过地址对常量的操作都是针对副本。
  常量折叠，又叫常量替换，c++编译器会在编译时，将const常量的字面值保存在符号表中，在编译时使用这个字面常量进行替换。

> const_cast用法：const_cast<type_id> (expression)
> 该运算符用来修改类型的const或volatile属性。除了const或volatile修饰之外，要求type_id和expression的类型是一样的。
> 常量指针被转化成非常量的指针，并且仍然指向原来的对象；
> 常量引用被转换成非常量的引用，并且仍然指向原来的对象；

```cpp
#include <iostream>

using namespace std;

int main(int argc, const char** argv)
{
    const int a = 10;
    const int *p = &a;

    // (*p) = 11;               // error 不可修改
    std::cout << "a:" << a << "*p:" << *p << std::endl; // a:10 *p:10

    const_cast<int&>(*p) = 11;  // 强制修改
    std::cout << "a:" << a << "*p:" << *p << std::endl; // a:10 *p:11

    return 0;
}
```

5. 引用和指针

- 引用是一个别名，可以把它看作变量本身，但是指针本身也是一个变量
- 引用在定义的时候必须初始化，必须绑定一个对象，如果一个对象本身不存在则取别名也没有意义。所以对指针进行解引用(*)的时候要对指针进行非空检测，但是引用由于定义的时候肯定初始化了，则一定不为空。
- 非const引用不能绑定到const对象，但是const引用可以绑定到非const对象(对象本身可以修改自己，但是不能通过引用修改对象)
- 引用比指针安全，引用只能绑定到一个对象，指针可以指向多个地方，可能会造成内存溢出或悬挂指针等不安全的因素
- 形参传引用效率高，但引用为形参最好是一个const引用，防止实参本身被修改。

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

int main(void)
{
   int a = 10;
  // int &ref;            // error! 必须初始化
  // int &ref = 10;       // error!
  // const int &ref = 10; // ok
   const int &ref_a = a;  // ok
  // ref_a = 11;
   a = 12;
   cout <<"ref_a: "<<a<< endl;

  /* const int b = 20;
   int &ref_b = b;       // error! 非const引用不能绑定到const变量
   const int &res_b = b; // ok


   int c = 11;
   const int &ref_c = c; // ok
   //ref_c++;            // error!
  */

   return 0;
}
```

6. 内联函数

- 内联函数：用inline关键字修饰的函数，不加inline默认叫外联。
  1. 使用了inline关键字，编译器并不一定把此函数当作内联函数处理 内联函数代码一般1-5行，并且函数体中不能出现循环或递归。
  2. 内联函数的声明和定义是在一起的
  3. 在类中声明和定义在一起的成员函数都默认为内联函数
- 内联函数和宏定义
  宏定义：在预处理阶段替换，但是容易产生二义性，不能作为类的成员函数访问私有成员。
  内联函数：在编译阶段函数代码替换函数名，在调用运行的时候就没有函数的压栈和出栈的操作，提高运行效率，是空间换时间。
  如果要用某函数的指针调用它，则该函数不能是内联函数，如果动态链接库中有内联函数，那么该链接库升级的时候需要重新编译。内联函数和宏定义都不支持调试。

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

#define max(a,b) (a>b?a:b)

int max1(int a,int b)
{
    return a>b?a:b;
}


inline int max2(int a,int b)
{
    return a>b?a:b;
}

int main(void)
{
    cout <<"#define: "<<max(11,12)<<endl; //预处理阶段替换
    cout <<"max1: "<<max1(12,11)<<endl;   //函数的压栈出栈
    cout <<"max2: "<<max2(12,11)<<endl;   //运行时内联的展开
    return 0;
}
```

7. 重载

- 什么是函数重载：
  1.函数名相同
  2.参数列表必须不同（参数类型，参数个数，参数的顺序不同）
  3.跟函数返回值没有关系

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

/*
 *默认参数：如果某个参数被默认初始化了，其右边不能出现没有被默认初始化的参数
 * Error:
 * int average(double a=0.5, double b =1.1, double c)
 * {
 *   return (a+b+c)/2;
 * }
 */
int average(double a=0.5, double b =1.1, double c=2.1)
{
    return (a+b+c)/2;
}

int average(int a, int b)
{
    cout <<"average(int, int)" << endl;
    return (a+b)/2;
}

int average(double a, double b)
{
    cout <<"average(double, double)" << endl;
    return (a+b)/2;
}

int average(double a, double b, double c)
{
    cout <<"average(double,double,double)" << endl;
    return (a+b+c)/2;
}

int main(void)
{
    int a = 10, b = 11;
    average(a, b);
    cout <<"--------------" << endl;
    double c = 10.1, d = 10.2, e = 10.3;
    average(c, d);
    cout <<"--------------" << endl;
    average(c, d, e);
    return 0;
}
```

8. const成员函数和const对象

- const成员函数(常成员函数)
  不修改成员变量的函数我们一般定义为const属性的成员函数，常成员函数不能修改成员变量的值。
- const对象 (常对象)
  const属性的对象(如：const Person p)，常对象所有的成员变量都是const属性，不能用常对象调用非const的成员函数（常对象只能调用常成员函数）

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

class Person{
    public:
      Person(){}
      ~Person(){}
      void set(int var)
      {
          this->m_var = var;
      }
      int get() const //常成员函数
      {
        //  m_var2 = 12; //error! 不修改成员变量
          return m_var;
      }
    private:
      int m_var;
      int m_var2;

};

int main(void)
{
    const Person p;   // 常对象
    //p.set(11);      // error! const对象不能修改非const成员函数
    const_cast<Person &>(p).set(11); //ok  只对当前生效
    cout <<"const_cast: "<< p.get() <<endl;

    //p.set(12); //error! 同样是不对的
    return 0;
}
```

9. static成员

- static：静态成员：静态成员变量 和 静态成员函数
  1. 静态成员属于整个类不属于某一个对象，在静态存储区分配内存空间，被所有实例对象共享，如果是公共的(public)则可以在外部加类作用域直接访问。

2. 静态成员变量只能被初始化一次，不要在头文件中定义，为了避免重复定义。不要在构造函数中定义（构造函数可能被调用多次）另外是因为类的声明不分配内存空间.静态成员变量的初始化方式：int Person::m_var = 10;静态成员变量只在静态存储区保留一份拷贝,静态成员变量可以声明为本来的类类型
3. 静态成员函数没有this指针，不能访问普通成员变量。非静态成员函数即可以访问普通成员变量也可以访问静态成员变量

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

class Person{
    public:
      Person(){
         cout <<"constructor" << endl;
         m_counter++;
      }
     ~Person(){
         cout <<"distructor" << endl;
         m_counter--;
      }
    static void out_counter(){
         //cout <<m_var<< endl;  // error 不能访问普通成员变量
         cout <<"people counter: "<<m_counter << endl; //ok
     }

  //  private:
      int m_var;
      static int m_counter;
};

int Person::m_counter = 0;  // 静态成员变量初始化

int  main(void)
{
    Person p1;   //constructor
    Person p2;   //constructor
    Person p3;   //constructor
 // p1.out_counter();
 // p3.out_counter();
   Person *p4 = new Person;  //constructor
   cout <<"people counter: "<<Person::m_counter << endl;  // people counter:4
   delete p4;    //distructor
   cout <<"people counter: "<<Person::m_counter << endl;  // people counter:3
   return 0;
}
```

10. 面向对象

- OOP(Object Oriented Programming)
  特征：封装、继承、多态、抽象
  private和public针对类本身和调用者
  struct的缺省作用域是public
  class的缺省作用域是private

person.h

```
 namespace mystd
 {
 class  Person
 {
  private:
    char name[30];
    int  age;
    bool gender;
  public:
    Person(char *myname, int age, bool gender);
    ~Person();
    void show();          // 显示基本属性信息
    int afterYear(int n); // n年后多少岁
    inline int getAge()
    {
       return age;
    }
 };
 };
```

person.cpp

```
#include "person.h"
#include <iostream>
#include <string.h>
using namespace std;

namespace mystd
{
    Person::Person(char *myname, int myage, bool mygender)
    {
        strcpy(name, myname);
        age = myage;
        gender = mygender;
    }
    Person::~Person(){}
    void Person::show()         // 显示基本属性信息
    {
       cout <<"name: "<<name<<" age: "<<age<<" gender: "<<gender << endl;
    }
    int Person::afterYear(int n)// n年后多少岁
    {
       return age += n;
    }
};
```

main.cpp

```
#include "person.h"
#include <iostream>
using namespace std;
using namespace mystd;

int main(void)
{
    Person p1("Lin", 24, true);
    p1.show();
    cout <<"after ten years: " << p1.afterYear(10)<<endl;
    cout <<p1.age<< endl;
    return 0;
}
```

11. 构造函数和析构函数

- 构造函数
  1.特殊的成员函数，名字跟类名相同，没有返回类型，必须为public，构造函数的作用是初始化对象的属性。
  2.如果没有显示的定义任何构造函数，则编译器会自动合成一个构造函数，如果定义类构造函数，

则编译器不再自己合成构造函数。

3.构造函数可以重载

4.构造函数初始化可以使用初始化参数列表，成员变量的初始化顺序跟初始化列表的顺序无关，是按照成员变量的声明顺序。

5.什么时候一定要用初始化列表

有const成员变量和引用成员变量的时候一定要用初始化列表初始化这两种变量

- 析构函数

  没有返回值，析构函数名称类型前加 ~

  1.如果是栈对象 作用域结束的时候自动调用析构函数

  2.如果是堆对象(new出来的对象),则要程序员delete的时候才调用析构函数

  3.如果是静态对象（在静态存储区/静态数据区中的对象）则在程序装入内存的时候就构造，进程结束的时候析析构。

- 浅拷贝、深拷贝

  当由一个已有的对象来构造一个新的对象时，需要调用拷贝构造函数

  浅拷贝(位拷贝)：对象成员变量没有使用动态分配内存空间的时候，对象和对象之间进行拷贝构造的时候使用浅拷贝就行

  深拷贝：如果对象内存使用了动态分配内存空间，则要使用深拷贝(显示定义进行内存拷贝的拷贝构造函数)

  如果不进行深拷贝，则会造成悬挂指针，多次析构同一块堆内存

- 什么时候会调用拷贝构造函数：

  1.把一个对象作为参数进行值传递的时候(所以说我们一般把对象作为参数的时候传对象的const引用)

  2.把一个对象作为返回值的时候

  如果存在拷贝构造函数的需求，没有显示的定义拷贝构造函数，则编译器会自动生成一个拷贝构造函数。如果显示定义了拷贝构造函数之后，则会调用该显示定义的拷贝构造函数，但是编译器还是会自动生成一个拷贝构造函数。新创建的对象去调用拷贝构造函数。

constructor.cpp

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

class Person
{
   public:
       Person(){cout <<"---constructor---"<< endl;}
      // Person(int a){m_var1 = a;}
       Person(int a):m_var1(a),m_var2(m_var2),m_var3(a)
       {
        // m_var3 = 3; //error
         cout <<"Person(int, int) constructor" << endl;
       }
       ~Person(){cout <<"---distructor---" << endl;}
       void show(){cout <<"m_var1: "<<m_var1 <<" m_var2: "<<m_var2<< endl;}
   private:
     int m_var1;
     int m_var2;
     const int m_var3;
    // int &ref;

};

Person p1(10);

int main()
{
   // Person p(10, 30);
   // p.show();
    cout <<"----------------" << endl;
    Person *p = new Person(10);
    p->show();
    delete p;
    p1.show();
    cout <<"----------------" << endl;
    return 0;
}
```

copy_constructor.cpp

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

class Computer{
     public:
       Computer(){}
       Computer(Computer&){
         cout <<"----computer constructor----" << endl;
       }
       ~Computer(){}
     private:
       string name;
};


class Student{
     public:
       Student(){}
       Student(string name, int age){
           cout <<"----constructor----" << endl;
           Computer com;
           m_comp = com;
           this->name = name;
           this->age = age;
       }
       //拷贝构造函数
        Student(const Student &s){
           cout <<"----copy constructor----" << endl;
           this->name = s.name;
           this->age = s.age;
       }
       ~Student(){}
       //把一个对象作为返回值的时候会调用该对象的一个拷贝构造函数
       Computer getComp(){
            cout <<"getComp()"<< endl;
            return m_comp;

       }
       Student getst(){
            return *this; //会调用该对象的拷贝构造函数
       }
       void disp(){
           cout <<"name: "<<name<<" age: "<<age << endl;
       }
     private:
       string name;
       int    age;
       Computer m_comp;
};
```

```
//把对象作为参数进行值传递的时候拷贝构造函数会被调用
void disp_out(Student s)
{
     s.disp();
}


int main(void)
{
    Student s1("Lin", 20);
    s1.disp();
    Student s2 = s1; // 自动调用拷贝构造函数
    s2.disp();
    Student s3(s2);  // 显式使用拷贝构造函数
    s3.disp();
    cout <<"******************" << endl;
    disp_out(s3);    // 把一个对象作为值传递的时候会调用拷贝构造函数
    cout <<"------------------" << endl;
    s1.getComp();
    s3.getst();
    return 0;
}
```

deep_copy.cpp

```
#include <iostream>
#include <string.h>
using namespace std;

class Mystring{
    public:
      Mystring(char *str, int counter){
         m_str = new char[counter+1];
         memcpy(m_str, str, counter);
         m_counter = counter;
         m_str[counter+1] = '\0';
      }
      //显示定义拷贝构造函数 进行深拷贝(也就是进行内存的拷贝)
       Mystring(const Mystring &str){
        // m_str =  str.m_str; // error!
         cout <<"deep copy constructor" << endl;
         this->m_str = new char[str.m_counter+1];
         memcpy(m_str, str.m_str, str.m_counter);
         this->m_counter = str.m_counter;
         m_str[m_counter+1] = '\0';
      }
     ~Mystring(){
         cout <<"distructor" << endl;
         delete []m_str;
      }
      void disp(){
         cout <<hex<<m_str<< endl;
      }

    private:
     char *m_str;
     int m_counter;
};

int main(void)
{
    Mystring str("briupemsd1109", 13);
    str.disp();
    Mystring str1 = str;  //用一个已有的对象去构造一个新的对象
    str1.disp();
    return 0;
}
```

12. this指针

- 类的任何对象的成员函数都会有一个隐含的this指针，也就是该类对象的一个指针。如果要返回一个对象的引用则可以返回 *this表示对象本身。

address.h

```
#ifndef _ADDRESS_H_
#define _ADDRESS_H_
#include <string>
#include <iostream>
using namespace std;

class Address{
    public:
      Address(){}
      Address(string country, string province, string city, string street)
     {
         this->country = country;
         this->province = province;
         this->city = city;
         this->street = street;
     }
     ~Address(){}
     void setProvince(string province){
         this->province = province;
     }
     string getProvince() const{
         return province;
     }
     void setStreet(string street){
         this->street = street;
     }
     string getStreet() const{
         return street;
     }
     void out(){
         cout <<country<<"-"<<province<<"-"<<city<<"-"<<street<<endl;

     }
    private:
      string country;
      string province;
      string city;
      string street;
};
#endif
```

player.h

```
#ifndef _PLAYER_H_
#define _PLAYER_H_

#include "address.h"
#include <iostream>
using namespace std;
class Player{
   public:
      Player(){addr = NULL;}
      Player(int num, string name, int age, double salary)
      {
          this->num = num;
          this->name = name;
          this->age = age;
          this->salary = salary;
          this->addr = NULL;
      }
      Player(int num, string name, int age, double salary, Address *addr){
          this->num = num;
          this->name = name;
          this->age = age;
          this->salary = salary;
          this->addr = addr;
      }
     ~Player(){}
     void setNum(int num){
          this->num = num;
     }
     int getNum() const{
           return num;
     }
     void setAge(int age){
          this->age = age;
     }
     int getAge() const{
          return age;
     }
     void setSalary(double salary){
          this->salary = salary;
     }
     double getSalary() const{
          return salary;
     }
     void setAddress(Address *addr){
          this->addr = addr;
     }
     Address *getAddress(){
          return addr;
     }
     void out(){
```

```
            cout <<"num:  "<<num<< endl;
            cout <<"name: "<<name<< endl;
            cout <<"age:  " <<age<<endl;
            cout <<"salary: "<<salary <<endl;
            if(addr != NULL){
                addr->out();
            }
            cout <<"******************" << endl;
      }
    private:
       int num;
       string name;
       int age;
       double salary;
       Address *addr;
};

#endif
```

player_test.cpp

```
#include "address.h"
#include "player.h"
#include <iostream>

using namespace std;

int main(void)
{
   Address addr("china", "shanghai", "shanghai","huaihai RD");
   Player p1(17, "Lin", 24, 10000, &addr);
   p1.out();
   cout <<"--------------------" << endl;
   Player p2(24, "Kobe", 34, 20000);
   Address addr2("china", "jiangsu", "kunshan","xuey RD");
   p2.setAddress(&addr2);
   p2.out();
   cout <<"--------------------" << endl;
   Address *addr3 = new Address("USA", "MAIAMI", "MAIAMI", "MM.RD");
   Player *p3 = new Player(6, "James", 28, 20000, addr3);
   p3->out();
   delete addr3;
   delete p3;
   return 0;
}
```

13. 友元

> 类具备封装和信息隐藏的特性。只有类的成员函数才能访问类的私有成员，程序中的其他函数是无法访问私有成员的。非成员函数能够访问类中的公有成员，但是假如将数据成员都定义为公有的，这又破坏了隐藏的特性。另外，应该看到在某些情况下，特别是在对某些成员函数多次调用时，由于参数传递，类型检查和安全性检查等都需要时间开销，而影响程序的运行效率。
> 为了解决上述问题，提出一种使用友元的方案。友元是一种定义在类外部的普通函数，但他需要在类体内进行说明，为了和该类的成员函数加以区别，在说明时前面加以关键字friend。友元不是成员函数，但是他能够访问类中的私有成员。友元的作用在于提高程序的运行效率，但是，他破坏了类的封装性和隐藏性，使得非成员函数能够访问类的私有成员。

- 友元包括：友元函数、友元类
  关键字：friend
  友元函数：
  友元函数的特点是能够访问类中的私有成员的非成员函数。友元函数从语法上看，他和普通函数相同，即在定义上和调用上和普通函数相同。
  友元类：
  友元类的所有成员函数都是另一个类的友元函数，都可以访问另一个类中的隐藏信息（包括私有成员和保护成员）。
  当希望一个类可以存取另一个类的私有成员时，可以将该类声明为另一类的友元类。定义友元类的语句格式如下：
  friend class 类名;
  其中：friend和class是关键字，类名必须是程序中的一个已定义过的类。
  例如，以下语句说明类B是类A的友元类：
  class A
  {
  …
  public:
  friend class B;
  …
  };
  经过以上说明后，类B的所有成员函数都是类A的友元函数，能存取类A的私有成员和保护成员。
- 使用友元类时注意：
  (1) 友元关系不能被继承。
  (2) 友元关系是单向的，不具有交换性。若类B是类A的友元，类A不一定是类B的友元，要看在类中是否有相应的声明。
  (3) 友元关系不具有传递性。若类B是类A的友元，类C是B的友元，类C不一定是类A的友元，同样要看类中是否有相应的申明

注意事项:
1.友元可以访问类的私有成员。

2.只能出现在类定义内部，友元声明可以在类中的任何地方，一般放在类定义的开始或结尾。
3.友元可以是普通的非成员函数，或前面定义的其他类的成员函数，或整个类。
4.类必须将重载函数集中每一个希望设为友元的函数都声明为友元。
5.友元关系不能继承，基类的友元对派生类的成员没有特殊的访问权限。如果基类被授予友元关系，则只有基类具有特殊的访问权限。该基类的派生类不能访问授予友元关系的类。

14. 继承

继承： 类与类之间的关系
父类(基类) 子类(派生类) 继承语法
构建子类对象，先调用父类的构造函数，再调用子类自己的构造函数，析构的时候先调用子类自己的析构函数，再调用父类的析构函数
父类中的public和protected的成员变量和成员函数都会被子类继承下来

覆盖：
如果子类中有和父类函数名相同且参数相同的成员函数，则在子类对象调用该成员函数时会把父类的覆盖掉

隐藏：
如果子类中有和父类函数名相同但参数不同的成员函数，
则会在父类中该名称的成员函数会被隐藏掉

父类的指针绑定子类的对象 OK
子类的指针绑定父类的对象 error!

通过对象指针进行的普通成员函数调用，仅仅与指针的类型有关，而与此刻指针正指向什么对象无关。想要实现当指针指向不同对象时执行不同的操作就必须将基类中相应的成员函数定义为虚函数。

inherit.cpp

```cpp
#include <stdio.h>
#include <iostream>
using namespace std;

// 派生类是基类的具体化，而基类是派生类的抽象。
// 在多继承时，如果省略继承方式，默认为private
// 如果在派生类中声明了一个与基类成员相同名字的函数，派生类的新成员会覆盖基类的同名成员

/* 不管何种继承 基类的私有程序都不能被派生类继承 否则会破坏C++的封装特性
 * 基类的友元函数也不能被继承，友元只是能访问指定类的私有和保护成员的自定义函数，不是被指定类的成员，自然
 * 基类与派生类的静态成员函数与静态成员是共用一段空间的，即静态成员和静态成员函数是可以继承的
 */
// public公有继承时 基类的公用成员public和保护成员protected在派生类中保持原有的访问属性，其私有成员仍为
// protected保护继承 特点是基类的所有公有成员和保护成员都成为派生类的保护成员，并且只能被它的派生类成员函
// private私有继承 私有继承即所有基类成员均变成派生类的私有成员，基类的私有成员仍然不能在派生类中访问

class BASE
{
public:
    void who()
    {
        cout << "this is base !" << endl;
    }

      void Fun()
      {
           cout << "this is base Fun ! " << endl;
      }
};

class CD1:public BASE
{
public:
    void who()
    {
        cout << "this is CD1 !" << endl;
    }
};

class CD2:public BASE
{
public:
    void who()
    {
        cout << "this is CD2 !" << endl;
    }
};


int main(int argc, char* argv[])
```

```
{
    CD1 obj1;
    CD2 obj2;

      obj1.Fun();
      obj2.Fun();

    obj1.who();  //this is CD1 !
    obj2.who();  //this is CD2 !

    return 0;
}
```

15. 虚函数

- 什么是虚函数
  简单地说，就是在成员函数前加关键字virtual，这样这个成员函数就变成了虚函数。
  虚函数允许派生类取代基类所提供的实现。编译器确保当对象为派生类时，派生类的实现总是被调用，即使对象是使用基类指针访问而不是派生类的指针。

> 在函数形参表后面写上= 0以指定纯虚函数，含有纯虚函数的对象（抽象类）不能被实例化，只能作为基类被继承。

virtual.cpp

```cpp
#include <stdio.h>
#include <iostream>
using namespace std;


/*************************
*加virtual与不加virtual的区别
*************************/
class BASE
{
public:
    void who()
    {
        cout << "this is base !" << endl;
    }
/*
    virtual void who()     //virtual
    {
        cout << "this is base !" << endl;
    }
*/
};

class CD1:public BASE
{
public:
    void who()
    {
        cout << "this is CD1 !" << endl;
    }
};

class CD2:public BASE
{
public:
    void who()
    {
        cout << "this is CD2 !" << endl;
    }
};

int main(int argc, char* argv[])
{
    BASE obj;
    BASE *p;
    CD1 obj1;
    CD2 obj2;

    p = &obj;
    p->who();
```

```
    p = &obj1;
    p->who();

    p = &obj2;
    p->who();

    obj1.who();
    obj2.who();

    return 0;
}
```

16. 模板

C++中的一个概念：

泛型编程 ： 所谓泛型编程就是独立于任何特定类型的方式编写代码。模板是泛型编程的基础。

- 模板函数：
  函数模板是生成函数代码的样板，当参数类型确定后，编译时用函数模板生成一个具有确定类型的函数，这个由函数模板而生成的函数称为模板函数。
  模板定义以关键字template开始，后接模板形参表，模板形参表是用尖括号括住的一个或多个模板形参的列表，形参之间以逗号分开。
- inline函数模板：
  函数模板可以用与非模板函数一样的方式声明为inline。说明符放在模板形参表之后、返回类型之前，不能放在关键字template之前。

```
template <typename T> inline T min(const T&, const T&);  //ok
inline template <typename T> T min(const T&, const T&);  //error
```

- 类模板
  由于篇幅有限，关注（Linux兵工厂），后台回复 C++ 获取全部实例代码。

17. 类的大小与成员变量的访问

- 类所占字节大小似于结构体,与成员变量有关

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

class A{
   public:
      A():m_var1(10),m_var2('a'),m_var3('b'){}
     ~A(){}
      void disp(){
        cout <<"m_var1="<<m_var1<<" m_var2="<<m_var2<<" m_var3="<<m_var3<<endl;
      }
   private:
     int  m_var1;
     char m_var2;
     char m_var3;
};

int main(void)
{
    A a;
    a.disp();
    cout <<"m_var1: "<<*((int*)(&a))<< endl;
    cout <<"m_var2: "<<*((char*)(&a)+4)<<endl;
    cout <<"m_var3: "<<*((char*)(&a)+5)<<endl;
    cout <<sizeof(a) << endl;

    /*
    m_var1=10 m_var2=a m_var3=b
    m_var1: 10
    m_var2: a
    m_var3: b
    8
    */

    return 0;
}
```