1. 静态成员变量和静态成员函数
   1. 静态成员变量
      1. 编译阶段分配内存
      2. 所有对象共享数据
      3. 通过对象访问、通过类名访问
      4. 有权限控制
      5. 类内声明 类外初始化
   2. 静态成员函数
      1. 可以访问静态成员变量，不可以方法普通成员变量
      2. 普通成员函数 都可以访问
      3. 静态成员函数也有权限
      4. 可以通过对象访问，也可以通过类名进行访问

|  |
| --- |
| class Person  {  public:  Person()  {  //m\_Age = 10;  }  static int m\_Age; //加入static就是 静态成员变量 ，会共享数据  //静态成员变量，在类内声明，类外进行初始化  //静态成员变量 也是有权限的  int m\_A;  //静态成员函数  //不可以访问 普通成员变量  //可以访问 静态成员变量  static void func()  {  //m\_A = 10;  m\_Age = 100;  cout << "func调用" << endl;  };  //普通成员函数 可以访问普通成员变量，也可以访问静态成员变量  void myFunc()  {  m\_A = 100;  m\_Age = 100;  }  private:  static int m\_other; //私有权限 在类外不能访问  static void func2()  {  cout << "func2调用" << endl;  }  };  int Person::m\_Age = 0; //类外初始化实现  int Person::m\_other = 10; |

1. 单例模式案例 – 主席
   1. 目的 为了让类中只有一个实例，实例不需要自己释放
   2. 将 默认构造 和 拷贝构造 私有化
   3. 内部维护一个 对象指针
   4. 私有化唯一指针
   5. 对外提供 getInstance方法来访问这个指针
   6. 保证类中只能实例化唯一一个对象

|  |
| --- |
| //创建主席类  //需求 单例模式 为了创建类中的对象，并且保证只有一个对象实例  class ChairMan  {  //1构造函数 进行私有化  private:  ChairMan()  {  //cout << "创建国家主席" << endl;  }  //拷贝构造 私有化  ChairMan(const ChairMan&c)  {}  public:  //提供 get方法 访问 主席  static ChairMan\* getInstance ()  {  return singleMan;  }  private:  static ChairMan \* singleMan;  };  ChairMan \* ChairMan::singleMan = new ChairMan; |

|  |
| --- |
| void test01()  {  /\*ChairMan c1;  ChairMan \* c2 = new ChairMan;  ChairMan \* c3 = new ChairMan;\*/  /\*ChairMan \* cm = ChairMan::singleMan;  ChairMan \* cm2 = ChairMan::singleMan;  \*/  //ChairMan::singleMan = NULL;  ChairMan \* cm1 = ChairMan::getInstance();  ChairMan \* cm2 = ChairMan::getInstance();  if (cm1 == cm2)  {  cout << "cm1 与 cm2相同" << endl;  }  else  {  cout << "cm1 与 cm2不相同" << endl;  }  /\*ChairMan \* cm3 = new ChairMan(\*cm2);  if (cm3 == cm2)  {  cout << "cm3 与 cm2相同" << endl;  }  else  {  cout << "cm3 与 cm2不相同" << endl;  }\*/  } |

1. 单例模式案例 – 打印机案例
   1. 类似主席案例
   2. 提供打印功能
   3. 提供统计打印次数功能

|  |
| --- |
| class Printer  {  private:  Printer(){ m\_Count = 0; };  Printer(const Printer& p);  public:  static Printer\* getInstance()  {  return singlePrinter;  }  void printText(string text)  {  cout << text << endl;  m\_Count++;  cout << "打印机使用了次数为： " << m\_Count << endl;  }  private:  static Printer\* singlePrinter;  int m\_Count;  };  Printer\* Printer::singlePrinter = new Printer;  void test01()  {  //拿到打印机  Printer \* printer = Printer::getInstance();  printer->printText("离职报告");  printer->printText("入职报告");  printer->printText("加薪申请");  printer->printText("升级申请");  printer->printText("退休申请");  } |

1. C++对象模型初探
   1. 成员变量和成员函数是分开存储的
   2. 空类的大小 1
   3. 只有非静态成员属性才属于对象身上

|  |
| --- |
| class Person  {  public:  int m\_A; //非静态成员变量，属于对象身上  void func( ) {}; //非静态成员函数 不属于对象身上  static int m\_B; //静态成员变量 ，不属于对象身上  static void func2(){}; //静态成员函数 ，不属于对象身上  double m\_C; // 12 错误 16正确  };  //结论 ： 非静态成员变量 ，才属于对象身上  void test01()  {  // 6 、 0 、 4 、 1  cout << "sizo of (Person) = " << sizeof(Person) << endl;  //空类的大小为 1 每个实例的对象 都有独一无二的地址，char维护这个地址  // Person p[10] p[0] p[1]  }  void test02()  {  //this指针指向被调用的成员函数所属的对象  Person p1;  p1.func(); //编译器会偷偷 加入一个 this指针 Person \* this  Person p2;  p2.func();  } |

1. this指针使用
   1. 指针永远指向当前对象
   2. 解决命名冲突
   3. \*this 指向对象本体
   4. 非静态的成员函数才有this指针

|  |
| --- |
| // this可以解决命名冲突  class Person  {  public:  Person( int age)  {  this->age = age;  }  //对比年龄  void compareAge( Person & p)  {  if (this->age == p.age )  {  cout << "年龄相等" << endl;;  }  else  {  cout << "年龄不相等" << endl;;  }  }  //年龄相加  Person& PlusAge(Person & p)  {  this->age += p.age;  return \*this; //\*this指向对象本体  }  int age;  };  void test01()  {  Person p1(10);    cout << "p1的年龄" << p1.age << endl;  Person p2(10);  p1.compareAge(p2);  p1.PlusAge(p2).PlusAge(p2).PlusAge(p2); //链式编程  cout << "p1的年龄" << p1.age << endl;  } |

1. 空指针访问成员函数
   1. 如果成员函数没有用到this，那么空指针可以直接访问
   2. 如果成员函数用的this指针，就要注意，可以加if判断，如果this为NULL就return

|  |
| --- |
| class Person  {  public:  void show( )  {  cout << "Person show" << endl;  }  void showAge()  {  if (this == NULL)  {  return;  }  cout << this->m\_Age << endl; // NULL -> m\_Age  }  int m\_Age; //  };  void test01()  {  Person \* p = NULL;  p->show();  p->showAge();  } |

1. 常函数 常对象
   1. 常函数 void func() const {} 常函数
   2. 常函数 修饰是this指针 const Type \* const this
   3. 常函数 不能修改this指针执行的值
   4. 常对象 在对象前 加入 const修饰 const Person p1
   5. 常对象 不可以调用普通的成员函数
   6. 常对象 可以调用常函数
   7. 用mutable修饰的关键字是在常函数可以修改的

|  |
| --- |
| class Person  {  public :  Person()  {  //构造中修改属性  //this 永远执行本体    this->m\_A = 0;  this->m\_B = 0;  }  void showInfo() const //常函数 不允许修改指针指向的值  {  //this->m\_A = 1000;  this->m\_B = 1000;  // const Person \* const this  cout << "m\_A = " << this->m\_A << endl;  cout << "m\_B = " << this->m\_B << endl;  }  void show2() const  {  //m\_A = 100;  }  int m\_A;  mutable int m\_B; //就算是常函数 我还是执意要修改  };  void test01()  {  Person p1;  p1.showInfo();  //常对象 不允许修改属性  const Person p2;  //cout << p2.m\_A << endl;  p2.show2();  //常对象 不可以调用普通成员函数  //常对象 可以调用常函数  } |

1. 友元
   1. 全局函数做友元函数
      1. 全局函数写到 类中做声明 并且最前面写关键字 friend

|  |
| --- |
| class Building  {  //让全局的好基友函数 变为我的好朋友 友元函数  friend void goodGay(Building \* building);  public:  Building()  {  this->m\_SittingRoom = "客厅";  this->m\_BedRoom = "卧室";  }  //客厅 卧室  public:  string m\_SittingRoom; //客厅  private:  string m\_BedRoom; //卧室  };  //全局函数 好基友  void goodGay( Building \* building )  {  cout << "好基友正在访问 " << building->m\_SittingRoom << endl;  cout << "好基友正在访问 " << building->m\_BedRoom << endl;  }  //友元函数 目的访问类中的私有成员属性  void test01()  {  Building \* building = new Building;  goodGay(building);  } |

* 1. 让整个类 做友元类
     1. friend class 类名
     2. 友元类 是单向，不可传递的

|  |
| --- |
| class Building;  class goodGay  {  public:  goodGay();  void visit();  private:  Building \* building;  };  class Building  {  //让好基友类 作为 Building的好朋友  friend class goodGay;  public:  Building();  public:  string m\_SittingRoom; //客厅  private:  string m\_BedRoom; //卧室  };  goodGay::goodGay()  {  building = new Building;  }  void goodGay::visit()  {  cout << "好基友正在访问： " << this->building->m\_SittingRoom << endl;  cout << "好基友正在访问： " << this->building->m\_BedRoom << endl;  }  Building::Building()  {  this->m\_SittingRoom = "客厅";  this->m\_BedRoom = "卧室";  } |

* 1. 让成员函数做友元函数
     1. friend void goodGay::visit();

|  |
| --- |
| //只让 visit可以作Building的好朋友 visit2 不可以访问私有属性  class Building;  class goodGay  {  public:  goodGay();  void visit();  void visit2();  private:  Building \* building;  };  class Building  {  //让成员函数 visit作为友元函数  friend void goodGay::visit();    public:  Building();  public:  string m\_SittingRoom; //客厅  private:  string m\_BedRoom; //卧室  };  goodGay::goodGay()  {  building = new Building;  }  void goodGay::visit()  {  cout << "好基友正在访问： " << this->building->m\_SittingRoom << endl;  cout << "好基友正在访问： " << this->building->m\_BedRoom << endl;  }  void goodGay::visit2()  {  cout << "好基友正在访问： " << this->building->m\_SittingRoom << endl;  //cout << "好基友正在访问： " << this->building->m\_BedRoom << endl;  }  Building::Building()  {  this->m\_SittingRoom = "客厅";  this->m\_BedRoom = "卧室";  } |