C++教案

编程语言排行

- TIOBE 编程语言排行榜

  https://www.tiobe.com/tiobe-index/

- 历经 40 年，“永不过时的编程语言”

C++的应用领域

- 游戏（Cocos2d-X）、图像、多媒体、网络、嵌入式
- 数据库（Oracle、MySQL）、浏览器（Chrome）、搜索引擎（Google）
- 操作系统、驱动程序、编译器（GCC、LLVM）、编程语言（Swift）
- HPC（High Performance Computing，高性能计算）
- iOS 开发（Runtime、AsyncDisplayKit）
- Android 开发（NDK、fresco【匿名共享内存，Ashmem，Anonymous Shared Memory】）
- Java 开发（JNI）
- ……
- 总结

C++之所以应用范围如此广泛，得益于它的高效性、稳定性、跨平台性

虽然 C++在很多大型应用中，无法施展拳脚；但在某些领域，如同巨人一般而且是不可或缺的顶梁柱

基本只要是用到 C++的地方，都是高大上的地方

学习 C++的必要性?

■ C++可以说是当今很多流行语言（Java、Python 等）的老祖宗，学习 C++，相当于理解了流行语言的前世今生

■ 多尝试几种不同的编程语言，能提供不同的编程思维视角，站在更高的维度去思考代码

■ C++是一门在面向过程和面向对象方面都比较完善的语言，能让我们更接近真相（本质）

■ C++程序员转什么领域都可以很快上手

■ 如果你想做个普通的程序员，学好所熟悉的语言基本够用，如果你的理想还要更大一点，C++是进阶必备

■ 修炼内功，掌握本质，提升逼格

嵌入式学习过程的三个阶段、 嵌入式 linux 学习方法

第一阶段:基础与理论阶段

主要包括一些理论知识，你至少了解这行业吧，基本的 Linux 系统使用;其次就是嵌入式核心开发语言 C 语言(必须精通);了解 C 语言数据结构及经典算法编程;最后就是要了解嵌入式产品的一个基本的开发流程，这对后续的开发有很大的帮助，不至于是那么的迷茫。

第二阶段:嵌入式系统核心开发

至少这些是你要学会的，当前应用层的开发挺多，特别刚入行前期，神马驱动的、移植的相关的可能你还不熟练，找工作就靠下面这些知识点了。

①嵌入式 Linux 应用编程;　　②嵌入式 Linux 并发程序设计;

③嵌入式 Linux 网络编程;　　④嵌入式数据库开发;

⑤嵌入式 Linux 应用综合项目;　　⑥嵌入式 C++编程

⑦嵌入式 Qt 编程　　⑧ARM 硬件接口开发;

第三阶段:底层

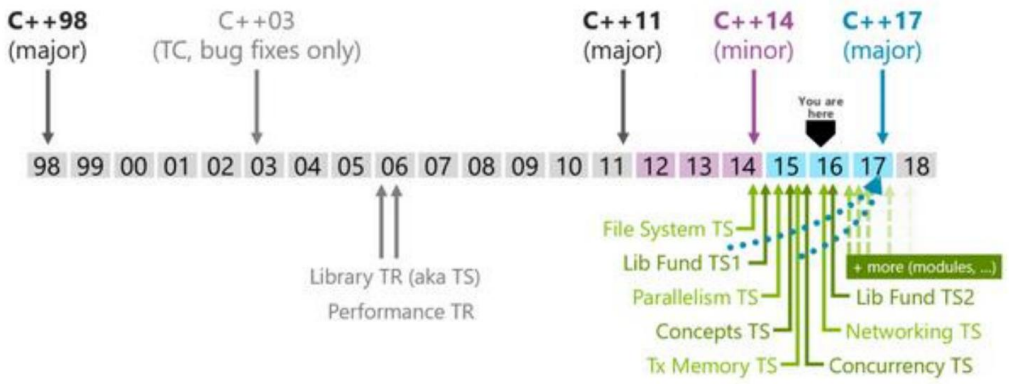
嵌入式底层一般会涉及到，如何把你写的程序移植到开发板上运行，那么就会接触到系统移植、内核驱动开发等等，这是嵌入式工程师最高境界。主要要学以下这些：

①嵌入式 Linux 系统开发;　　③嵌入式 Linux 内核开发;

②嵌入式 Linux 驱动开发基础;　　④嵌入式 Linux 驱动高级开发;

①：多看代码、多写代码　　②：在学习和工作中要脚踏实地

C++自身也在不断发展和完善



学习目录

01-基本的输入输出

C++ 标准库提供了一组丰富的输入/输出功能。

C++ 的 I/O 发生在流中，流是字节序列。如果字节流是从设备（如键盘、磁盘驱动器、网络连接等）流向内存，这叫做**输入操作**。

如果字节流是从内存流向设备（如显示屏、打印机、磁盘驱动器、网络连接等），这叫做**输出操作**。

头文件	函数和描述
<iostream>	该文件定义了 **cin、cout、cerr** 和 **clog** 对象，分别对应于标准输入流、标准输出流、非缓冲标准错误流和缓冲标准错误流。

1.标准输出

(1) cout 对象"连接"到标准输出设备，通常是显示屏。

(2) << 运算符被重载来输出内置类型（整型、浮点型、double 型、字符串和指针）的数据项。

(3) 流插入运算符 << 在一个语句中可以多次使用，如上面实例中所示，**endl** 用于在行末添加一个换行符。

```
#include <iostream>
using namespace std;
int main() {
        cout << "Hello World!!!" << endl;
        return 0;
}
```

2.标准输入

(4) cin 对象附属到标准输入设备，通常是键盘。

(5) cin 是与流提取运算符 >> 结合使用的。

(6) C++ 编译器根据要输入值的数据类型，选择合适的流提取运算符来提取值，并把它存储给定的变量中。

(7) 流提取运算符 >> 在一个语句中可以多次使用，如果要求输入多个数据

```
#include <iostream>
using namespace std;
int main(int argc, char *argv[])
{
    int num1;
    float num2;
    char str[50];
    cin >> num1 >> num2 >> str;//输入整型,浮点型,字符数组
    cout << num1 << endl;   //输出 num1
    cout << num2 << endl;   //输出 num2
    cout << str << endl;   //输出 str
    return 0;
}
```

02-命名空间

问题 1：为什么需要命名空间?

答：①命名空间是 ANSIC++引入的可以由用户命名的作用域，用来处理程序中 常见的同名冲突。
②模块化应用程序，形成具有高内聚低耦合的代码组成。

问题 2：什么是命名空间?

答：实际上就是一个由程序设计者命名的内存区域，程序设计者可以根据需要指定一些有名字的空间域，把一些全局实体分别放在各个命名空间中，从而与其他全局实体分隔开来。

1.定义命名空间

使用关键字 namespace

```
namespace namespace_name {
   // 代码声明
}
```

2.使用命名空间调用

(1) 使用 **using** namespace 指令引入命名空间

① 全部引入 using namespace std;

② 部分引入 using namespace std::cout;

(2) 使用域调用的方式

① 域调用 std::cout << "hello world \n";

3.嵌套命名的定义及引用

1.嵌套命名定义

```
namespace namespace_name {//大的命名空间
    namespace namespace_name_1{//内嵌命名空间 ①
        // 代码声明
    }
    namespace namespace_name_2{//内嵌命名空间 ②
        // 代码声明
    }
}
```

2 使用域调用的方式

```
using namespace namespace_name::namespace_name_1;//①引入
      //namespace_name 的内嵌命名空间①
using namespace namespace_name::namespace_name_2;//②引入
      //namespace_name 的内嵌命名空间②
```

03-函数重载

1. 函数重载的作用

(1) 减少对用户的复杂性。

(2) 减少了函数名的数量，避免了名字空间的污染，有利于程序的可读性。

解释：

函数重载是指在同一作用域内，可以有一组具有相同函数名，不同参数列表的函数，这组函数被称为重载函数。

重载函数通常用来声明几组功能相似的同名函数，但这些同名函数的形式参数必须不同，即用同一个运算符完成不同的运算功能。

重载函数常用来实现功能类似而所处理的数据类型不同的问题。

2. 函数重载的定义

在同一个作用域内，可以声明几个功能类似的同名函数，但是这些同名函数的形式参数（指参数的个数、类型或者顺序）必须不同。您不能仅通过返回类型的不同来重载函数。

重点：①同一个作用域内

②同名函数

③参数的(个数或者类型或者顺序)必须不同

④返回类型的不同

例子：函数 add()求和函数

```
#include <iostream>
using namespace std;
/*************************************
 * ①同一个作用域
 * ②函数名相同
 * ③参数个数/类型/顺序不同
 * ④返回类型不同
 * ************************************/
int add(int num1,int num2)
{
    return num1 + num2;
}

double add(double num1,double num2)
{
    return num1 + num2;
}

int main()
{
    cout << add(10,20) <<endl;
    cout << add(10.0,20.2) <<endl;
    return 0;
}
```

04-默认参数

1.默认参数作用

当您定义一个函数，您可以为参数列表中后边的每一个参数指定默认值。当调用函数时，如果实际参数的值留空，则使用这个默认值。

这是通过在函数定义中使用赋值运算符来为参数赋值的。调用函数时，如果未传递参数的值，则会使用默认值，如果指定了值，则会忽略默认值，使用传递的值。

2.注意事项

(1) 实参个数必须大于或等于无默认值的形参个数
(2) 匹配参数的时候是从左至右去匹配
(3) 参数默认值只能在声明或定义中一处指定.不能同时指定
(4) 默认参数与函数重载的二义性问题

示例：

```
#include <iostream>
using namespace std;
int sum(int a, int b=20)
{
  return a + b;
}

int main ()
{
   // 局部变量声明
   int a = 100;
   int b = 200;
   int result;

   // 调用函数来添加值
   result = sum(a, b);
   cout << "Total value is :" << result << endl;

   // 再次调用函数
   result = sum(a);
   cout << "Total value is :" << result << endl;

   return 0;
}
```

05-extern "C"

1.作用

(1) 为了能够正确实现 C++代码调用其他 C 语言代码。

(2) 由于 C、C++编译规则的不同，在 C、C++混合开发时，可能会经常出现以下操作。

(3) C++在调用 C 语言 API 时，需要使用 extern "C"修饰 C 语言的函数声明

2. 使用方法

(1) extern "C" 修饰的代码

例如：由于 C 语言没有函数重载所以出现命名冲突。

```
//单个声明

extern "C" void func();
extern "C" void func(int age);//报错

void func()
{
    cout << "func()" << endl;
}

void func(int age)
{
    cout << "func(" << age << ")"  << endl;
}
```

```
//同时声明

extern "C" {
    void func();
    void func(int age);//报错
}

void func()
{
    cout << "func()" << endl;
}

void func(int age)
{
    cout << "func(" << age << ")"  << endl;
}
```

例如：使用 C++编译方式就可以使用函数重载

```
//单个声明

extern "C++" void func();
extern "C++" void func(int age);

void func()
{
    cout << "func()" << endl;
}

void func(int age)
{
    cout << "func(" << age << ")"  << endl;
}
```

```
//同时声明
extern "C++" {
    void func();
    void func(int age);
}

void func()
{
    cout << "func()" << endl;
}

void func(int age)
{
    cout << "func(" << age << ")"  << endl;
}
```

06-内联函数

1. 作用
   (1)在 C 语言中，如果一些函数被频繁调用，不断地有函数入栈，即函数栈，会造成栈空间或栈内存的大量消耗。特别的引入了 inline 修饰符，表示为内联函数。
   (2)内联函数是通常与类一起使用。如果一个函数是内联的，那么在编译时，编译器会把该函数的代码副本放置在每个调用该函数的地方。
2. 定义
   (1) 需要在函数名前面放置关键字 inline
   (2) 类结构中所在的类说明内部定义的函数是内联函数。
3. 注意事项
   (1) 在内联函数内不允许使用循环语句和开关语句
   (2) 不要内联超过 10 行的函数
   (3) 内联函数的定义必须出现在内联函数第一次调用之前
   (4) 类结构中所在的类说明内部定义的函数是内联函数
   (5) 如果已定义的函数不符合内联规则，编译器会忽略 inline 限定符。
4. 优点
   (1) 函数体比较小的时候，内联该函数可以令目标代码更加高效。
   (2) 对于存取函数以及其它函数体比较短，性能关键的函数，鼓励使用内联。
5. 缺点
   (1) 滥用内联将导致程序变慢. 内联可能使目标代码量或增或减，这取决于内联函数的大小。
   (2) 内联非常短小的存取函数通常会减少代码大小，但内联一个相当大的函数将戏剧性的增加代码大小。
   (3) 现代处理器由于更好的利用了指令缓存，小巧的代码往往执行更快。
6. 结论
   (1) 一个较为合理的经验准则是，不要内联超过 10 行的函数。
   (2) 谨慎对待析构函数，析构函数往往比其表面看起来要更长，因为有隐含的成员和基类析构函数被调用。

例子：

```
#include <iostream>
using namespace std;
//定义内联
inline int Max(int x, int y)
{
   return (x > y)? x : y;
}
//主程序入口
int main( )
{
   cout << "Max (20,10): " << Max(20,10) << endl;
   cout << "Max (0,200): " << Max(0,200) << endl;
   cout << "Max (100,1010): " << Max(100,1010) << endl;
   return 0;
}
```

07-引用

1. 引用的作用
   (1) 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。
   (2) 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。
2. 引用 vs 指针
   (1) 不存在空引用。引用必须连接到一块合法的内存。
   (2) 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。
   (3) 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。
3. 创建引用，& 读作**引用**
   (1) 支持把引用**作为参数**传给函数，这比传一般的参数更安全。
   (2) 可以从 C++ 函数中**返回引用**，就像返回其他数据类型一样。

```
int&  r = i;
double& s = d;
```

```
#include <iostream>
using namespace std;
int main ()
{
   // 声明简单的变量
   int    i;
   double d;

   // 声明引用变量
   int&    r = i;
   double& s = d;

   i = 5;
   cout << "Value of i : " << i << endl;
   cout << "Value of i reference : " << r  << endl;

   d = 11.7;
   cout << "Value of d : " << d << endl;
   cout << "Value of d reference : " << s  << endl;

   return 0;
}
```

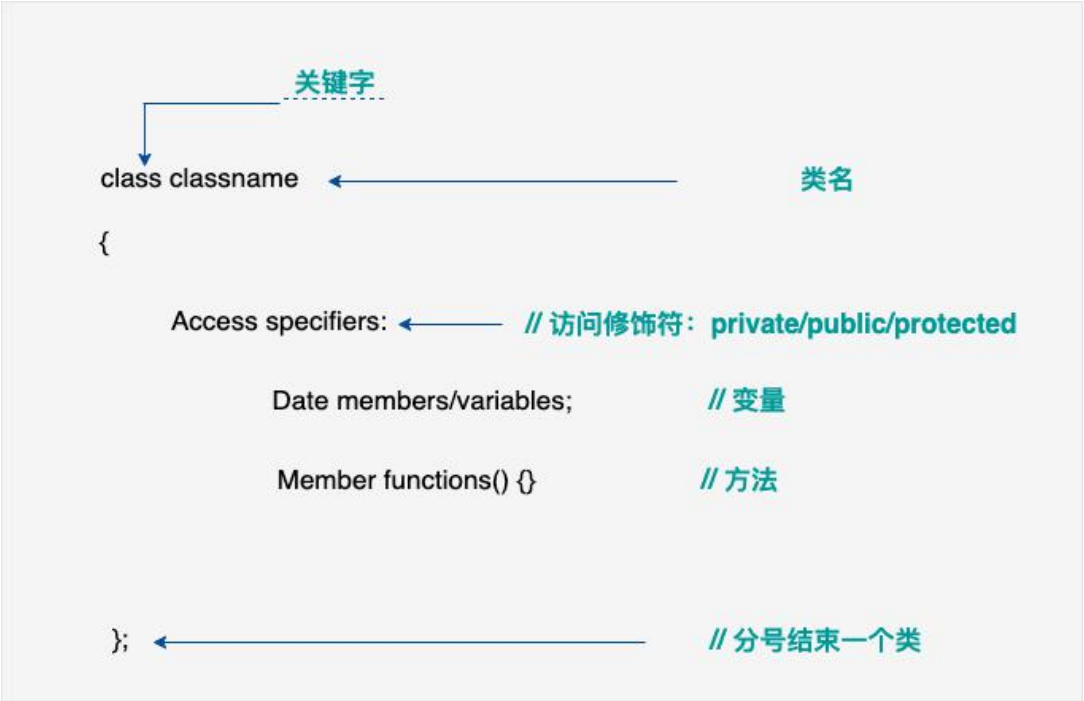
08-类和对象之封装

C++ 在 C 语言的基础上增加了面向对象编程，C++ 支持面向对象程序设计。类是 C++ 的核心特性，通常被称为用户定义的类型。

类用于指定对象的形式，它包含了数据表示法和用于处理数据的方法。类中的数据和方法称为类的成员。函数在一个类中被称为类的成员。

1. 类的定义

类定义是以关键字 **class** 开头，后跟类的名称。类的主体是包含在一对花括号中。类定义后必须跟着一个分号或一个声明列表。



2. 指定类的成员

公有 public　　私有 private　　保护 protected 默认情况下是定义为 private

访问	public 公有	protected 保护	private 私有
同一个类	yes	yes	yes
派生类	yes	yes	no
外部的类	yes	no	no

3. 定义 C++ 对象

```
class Box
{
   public:
      double length;   // 盒子的长度
      double breadth;  // 盒子的宽度
      double height;   // 盒子的高度
};
Box Box1;          // 声明 Box1，类型为 Box
Box Box2;          // 声明 Box2，类型为 Box
```

4. 访问数据成员

(1) 类的对象的公共数据成员可以使用直接成员访问运算符 `.` 来访问

(2) 指针对象的公共数据成员可以使用直接成员访问运算符 `->` 来访问

5. 类构造函数 & 析构函数

(1) 类构造函数

① 构造函数的作用

1) 类的**构造函数**是类的一种特殊的成员函数，它会在每次创建类的新对象时执行。

2) **构造函数**的名称与类的名称是完全相同的，并且不会返回任何类型，也不会返回 void。

3) **构造函数**可用于为某些成员变量设置初始值。

② 带参数的构造函数

```
class Line
{
   public:
      void setLength( double len );
      double getLength( void );
      Line(double len);  // 这是构造函数

   private:
      double length;
};
// 成员函数定义，包括构造函数
Line::Line( double len)
{
    cout << "Object is being created, length = " << len << endl;
    length = len;
}
```

③ 使用初始化列表来初始化字段

```
Line::Line( double len): length(len)
{
    cout << "Object is being created, length = " << len << endl;
}
```

等同于

```
Line::Line( double len)
{
    length = len;
    cout << "Object is being created, length = " << len << endl;
}
```

(2) 析构函数

类的**析构函数**是类的一种特殊的成员函数，它会在每次删除所创建的对象时执行。

析构函数的名称与类的名称是完全相同的，只是在前面加了个波浪号（~）作为前缀，它不会返回任何值，也不能带有任何参数

析构函数有助于在跳出程序（比如关闭文件、释放内存等）前释放资源。

```
~Line(); // 这是析构函数声明
```

```
Line::~Line(void) { cout << "Object is being deleted" << endl; }// 析构函数实现
```

6. 拷贝构造函数

拷贝构造函数是一种特殊的构造函数，它在创建对象时，是使用同一类中之前创建的对象来初始化新创建的对象。

(1) 拷贝构造函数通常用情况

① 通过使用另一个同类型的对象来初始化新创建的对象。

② 复制对象把它作为参数传递给函数。

③ 复制对象，并从函数返回这个对象。

(2) 定义拷贝构造函数

① 如果在类中没有定义拷贝构造函数，编译器会自行定义一个。

② 如果类带有指针变量，并有动态内存分配，则它必须有一个拷贝构造函数。

```
class Line
{
   public:
      int getLength( void );
      Line( int len );             // 简单的构造函数
      Line( const Line &obj);      // 拷贝构造函数
      ~Line();                     // 析构函数

   private:
      int *ptr;
};
```

7. 友元

(1) 友元函数

类的友元函数是定义在类外部，但有权访问类的所有私有（private）成员和保护（protected）成员。

尽管友元函数的原型有在类的定义中出现过，但是友元函数并不是成员函数。

① 声明函数为一个类的友元，需要在类定义中该函数原型前使用关键字 **friend**

```
class Box
{
   double width;
public:
   friend void printWidth( Box box );
   void setWidth( double wid );
};
// 成员函数定义
void Box::setWidth( double wid )
{
    width = wid;
}
// 请注意：printWidth() 不是任何类的成员函数
void printWidth( Box box )
{
   /* 因为 printWidth() 是 Box 的友元，它可以直接访问该类的任何成员 */
   cout << "Width of box : " << box.width <<endl;
}
```

(2) 友元类

① 整个类及其所有成员都是友元

```
#include <iostream>
using namespace std;
class Box
{
    double width;
public:
    friend void printWidth(Box box);
    friend class BigBox;// 声明友元类
    void setWidth(double wid);
};
class BigBox
{
public :
    void Print(int width, Box &box)
    {
        // BigBox 是 Box 的友元类，它可以直接访问 Box 类的任何成员
        box.setWidth(width);
        cout << "Width of box : " << box.width << endl;
    }
};
// 成员函数定义
void Box::setWidth(double wid)
{
    width = wid;
}
// 请注意：printWidth() 不是任何类的成员函数
void printWidth(Box box)
{
    /* 因为 printWidth()是 Box 的友元,它可以直接访问该类的任何成员 */
    cout << "Width of box : " << box.width << endl;
}
// 程序的主函数
int main()
{
    Box box;
    BigBox big;
    box.setWidth(10.0);// 使用成员函数设置宽度
    printWidth(box);   // 使用友元函数输出宽度
    big.Print(20, box);// 使用友元类中的方法设置宽度
    return 0;
}
```

8. this 指针

每个对象都有一个特殊的指针 **this**，它指向对象本身。

每一个对象都能通过 **this** 指针来访问自己的地址。**this** 指针是所有成员函数的隐含参数。因此，在成员函数内部，它可以用来指向调用对象。

友元函数没有 **this** 指针，因为友元不是类的成员。只有成员函数才有 **this** 指针。

```
class Box
{
   public:
      Box()// 构造函数定义
      { //this 指针使用
         this->length = 10.0;
         this->breadth = 11.0;
         this->height = 12.0;
      }
   private:
      double length;     // Length of a box
      double breadth;    // Breadth of a box
      double height;     // Height of a box
};
```

9. 类的静态成员

(1) 静态成员

① 静态成员不占类空间大小

② 类的数据成员和函数成员都可以被声明为静态的。

③ 当我们声明类的成员为静态时，这意味着无论创建多少个类的对象，静态成员都只有一个副本。

④ 静态成员在类的所有对象中是共享的。

⑤ 如果不存在其他的初始化语句，在创建第一个对象时，所有的静态数据都会被初始化为零。

⑥ 我们不能把静态成员的初始化放置在类的定义中，但是可以在类的外部通过使用范围解析运算符 :: 来重新声明静态变量从而对它进行初始化

```
class Box
{
   public:
      Box();
      static int objectCount;   //声明静态成员
   private:
      double length;     // Length of a box
      double breadth;    // Breadth of a box
      double height;     // Height of a box
};
// 初始化类 Box 的静态成员
int Box::objectCount = 0;
```

(2) 静态成员函数

① 如果把函数成员声明为静态的，就可以把函数与类的任何特定对象独立开来。

② 静态成员函数即使在类对象不存在的情况下也能被调用。

③ **静态函数**使用类名加范围解析运算符 :: 就可以访问。

④ 静态成员函数只能访问静态成员数据、其他静态成员函数和类外部的其他函数。

⑤ 静态成员函数有一个类范围，他们不能访问类的 this 指针。

⑥ 可以使用静态成员函数来判断类的某些对象是否已被创建。

⑦ 静态成员函数与普通成员函数的区别：

1) 静态成员函数没有 this 指针，只能访问静态成员（包括静态成员变量和静态成员函数）。

2) 普通成员函数有 this 指针，可以访问类中的任意成员；而静态成员函数没有 this 指针。

```
class Box
{
   public:
      Box();
      static int objectCount;   //声明静态成员
      static int getCount()     //静态成员函数
      {
          return objectCount;
      }

   private:
      double length;     // Length of a box
      double breadth;    // Breadth of a box
      double height;     // Height of a box
};
// 初始化类 Box 的静态成员
int Box::objectCount = 0;
```

09-内存管理

1. 了解动态内存在 C++ 中是如何工作的是成为一名合格的 C++ 程序员必不可少的。
2. 内存分为两个部分：
   ① **栈**：在函数内部声明的所有变量都将占用栈内存。
   ② **堆**：这是程序中未使用的内存，在程序运行时可用于动态分配内存。
      1) **new** 运算符申请空间
      2) **delete** 运算符释放之前 **new** 运算符申请空间
3. new VS malloc
   1. new 是运算符，运行速度快于 malloc 函数
   2. new 不只是分配了内存，它还创建了对象；而 malloc 函数做不到。

例子 1：单个变量分配

```
double* pvalue  = NULL; // 初始化为 null 的指针
pvalue  = new double;   // 为变量请求内存
double* pvalue  = NULL;
if( !(pvalue  = new double ))
   {
      cout << "Error: out of memory." <<endl;
      exit(1);
   }
delete pvalue;//释放 new 申请的空间
```

数组申请空间及释放使用 new [number] 申请,使用 delete []指针名,注意只要申请的空间是连续的 delete 释放的[]内不填内容,C++编译器自动检测。

例子 2：数组动态分配及释放

```
int *array=new int [m];// 动态分配,数组长度为 m
delete [] array;//释放内存
```

4. 对象的动态内存分配
   ① 对象与简单的数据类型没有什么不同

```
#include <iostream>
using namespace std;
class Box
{
   public:
      Box() {
         cout << "调用构造函数！" <<endl; }
      ~Box() {
         cout << "调用析构函数！" <<endl; }
};
int main( )
{
   Box* myBoxArray = new Box[4];
   delete [] myBoxArray; // 删除数组
}
```

10-类的声明和实现分离

1.为什么要将类的声明和实现分离

①首先类声明和类实现分离是为了隐藏实现。

②使用显式声明实现类模板的接口与实现的文件分离

例子：实现接口化编程

```
#pragma once
//Person.h
class Person {
private:
	int m_age;
public:
	void setAge(int age);
	int getAge();
	Person();
	~Person();
};
```

```
#include "Person.h"
//Person.cpp
void Person::setAge(int age) {
	m_age = age;
}
int Person::getAge() {
	return m_age;
}
Person::Person() {
	m_age = 0;
}
Person::~Person() {

}
```

```
#include <iostream>
#include "Person.h"
using namespace std;
//main.cpp
int main() {
	Person person;
	person.setAge(10);

	cout << person.getAge() << endl;

	getchar();
	return 0;
}
```

11-继承

1.为什么要有继承？

①面向对象程序设计中最重要的一个概念是继承。继承允许我们依据另一个类来定义一个类，这使得创建和维护一个应用程序变得更容易。

②达到了重用代码功能和提高执行效率的效果。

③创建一个类时，您不需要重新编写新的数据成员和成员函数，只需指定新建的类继承了一个已有的类的成员即可。

2.继承展示

①已有的类称为**基类**

②新建的类称为**派生类**

例如：哺乳动物是动物，狗是哺乳动物，因此，狗是动物

3.基类和派生类的关系

①基类是已有的类，而派生类是继承来的。所以，派生类里面也有的基类成员

例如：人类有姓名，身高 -> 然后让学生继承人类，那么学生也有姓名和身高。

```
#include <iostream>
using namespace std;
// 基类
class People
{
public:
    void setHeight(int h){
        height = h;
    }
    void setName(const string &TempName){
        name = TempName;
    }
protected:
    string name;
    int height;
};

// 派生类
class Student: public People
{
public:
    void getStudent() {
        cout << "姓名:" << name << ",身高:" << height << "cm" <<endl;
    }
};

int main(void)
{
    Student stu;

    stu.setName("王老师");
    stu.setHeight(165);
    stu.getStudent();

    return 0;
}
```

4.访问控制和继承

访问	public 公有	protected 保护	private 私有
同一个类	yes	yes	yes
派生类	yes	yes	no
外部的类	yes	no	no

- 公有继承：保持父类的访问权限
- 保护继承：将父类公有成员继承为自己的保护成员，保护和私有不变
- 私有继承：将父类所有成员继承为自己的私有成员。

以下情况数据不继承：

①基类的构造函数、析构函数和拷贝构造函数。

②基类的重载运算符。

③基类的友元函数。

5.多继承

多继承即一个子类可以有多个父类，它**继承了多个父类的特性**。

在单继承的基础上用（**逗号**）隔开。

```
class <派生类名>:<继承方式 1><基类名 1>,<继承方式 2><基类名 2>,…
{
<派生类类体>
};
```

6.多继承会生的问题

② 多重继承(环状继承)

③ 菱形继承

解决方法会在后续章节讲解。

12-运算符重载

重载的运算符是带有特殊名称的函数，函数名是由关键字 operator 和其后要重载的运算符符号构成的。

与其他函数一样，重载运算符有一个返回类型和一个参数列表。

1.重载规则

双目算术运算符	+ (加)，-(减)，*(乘)，/(除)，% (取模)
关系运算符	==(等于)，!= (不等于)，< (小于)，> (大于>，<=(小于等于)，>=(大于等于)
逻辑运算符	||(逻辑或)，&&(逻辑与)，!(逻辑非)
单目运算符	+ (正)，-(负)，*(指针)，&(取地址)
自增自减运算符	++(自增)，--(自减)
位运算符	| (按位或)，& (按位与)，~(按位取反)，^(按位异或),，<< (左移)，>>(右移)
赋值运算符	=, +=, -=, *=, /= , % = , &=, |=, ^=, <<=, >>=
空间申请与释放	new, delete, new[ ] , delete[]
其他运算符	()(函数调用)，->(成员访问)，,(逗号)，[](下标)
不可重载符号	

注意

1.不可重载运算符

.：成员访问运算符

.*，->*：成员指针访问运算符

:::：域运算符

sizeof：长度运算符

?:：条件运算符

#：　预处理符号

2.运算符只能重载为成员函数

赋值运算符：=

下标运算符：[ ]

函数运算符：( )

指针访问成员：->

2.重载例子

1	一元运算符重载
2	二元运算符重载
3	关系运算符重载
4	输入/输出运算符重载
5	++ 和 -- 运算符重载
6	赋值运算符重载
7	函数调用运算符 () 重载
8	下标运算符 [] 重载
9	类成员访问运算符 -> 重载

13-构造函数的互相调用

1. 探究为什么要使用构造互调。
   ① 在 c++里，由于构造函数允许有默认参数，使得这种构造函数调用构造函数来重用代码的需求大为减少。
   ② 考虑到长远问题，那么以后代码修改功能修改方便。只需要改最大一个带参构造即可。
2. 实现构造构造函数互调
   ① 方法 1：使用 this 指针（以后 Java 使用方法 1）
   ② 方法 2：使用初始化列表（以后 C++使用方法 2）
3. 为什么不能直接调用带参构造函数？
   答：单纯的在构造函数中调用其它的构造函数，只是会产生一个临时的匿名变量，临时变量属于栈空间，构造函数结束后就会释放。

```
//①this 指针，Java
class Student
{
public:
    Student()
    {
        this("",0);//①this 指针
        cout << "Student()" <<endl;
    }

    Student(const string &name,int age)
    {
        cout << "Student(const string
&name,int age)" <<endl;
        sm_name = new string(name);
        m_age = age;
    }

    ~Student()
    {
        cout << "~Student()" <<endl;
        delete sm_name;
    }

    string* sm_name;
    int m_age;
};
```

```
//②初始化列表，C++
class Student
{
public:
    Student():Student("",0)//②初始化列表
    {
        cout << "Student()" <<endl;
    }

    Student(const string &name,int age)
    {
        cout << "Student(const string
&name,int age)" <<endl;
        sm_name = new string(name);
        m_age = age;
    }

    ~Student()
    {
        cout << "~Student()" <<endl;
        delete sm_name;
    }

    string* sm_name;
    int m_age;
};
```

14-父类的构造函数

1. 子类的构造函数默认会调用父类的无参构造函数。
2. 如果子类的构造函数显式地调用了父类的有参构造函数，就不会再去默认调用父类的无参构造函数。
3. 如果父类缺少无参构造函数，子类的构造函数必须显式调用父类的有参构造函数。

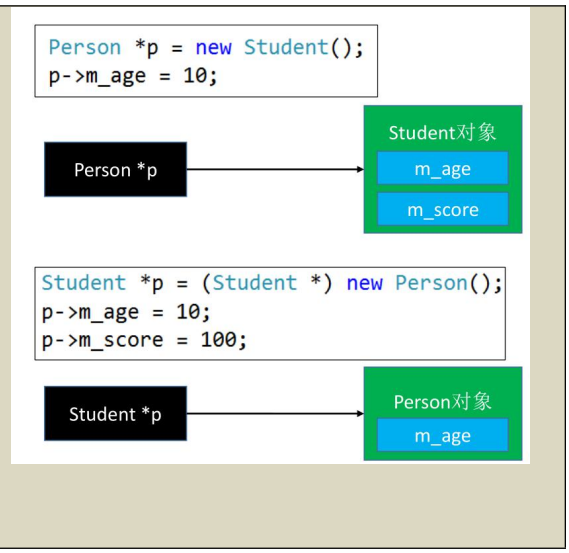
继承体系下的构造函数示例	构造、析构顺序

class Person{
int m_age;
Person():Person(0){
 	
}
 	
Person(int age):m_age(age){
 	
}
};
 	
class Student{
int m_no;
Student():Student(0,0){
 	
}
 	
Student(int age,int no):m_age(age),m_no(no){
 	
}
};
	class Person{
	Person(){
	cout << "Person::Person()" << endl;
	}
	
	~Person(){
	cout << "Person::~Person()" << endl;
	}
	};
	
	class Student{
	int m_no;
	Student(){
	cout << "Student::Student()" << endl;
	}
	
	~Student(){
	cout << "Student::~Student()" << endl;
	}
	
	};

15-多态

1. 父类指针与子类指针
   ① 父类指针可以指向子类对象，是安全的，开发中经常用到（继承方式必须是 public）
   ② 子类指针指向父类对象是不安全的

```
class Person{
    int m_age;
};

class Student:public Person{
    int m_no;
};

int main()
{
    Person *p = new Student;
    return 0;
}
```



2. 多态
   ① 默认情况下，编译器只会根据指针类型调用对应的函数，不存在多态
   ② 多态是面向对象非常重要的一个特性
      1) 同一操作作用于不同的对象，可以有不同的解释，产生不同的执行结果
      2) 在运行时，可以识别出真正的对象类型，调用对应子类中的函数
   ③ 多态的要素
      1) 子类重写父类的成员函数（override）
      2) 父类指针指向子类对象
      3) 利用父类指针调用重写的成员函数

3. 虚函数

① C++中的多态通过虚函数（virtual function）来实现

② 虚函数：被 virtual 修饰的成员函数

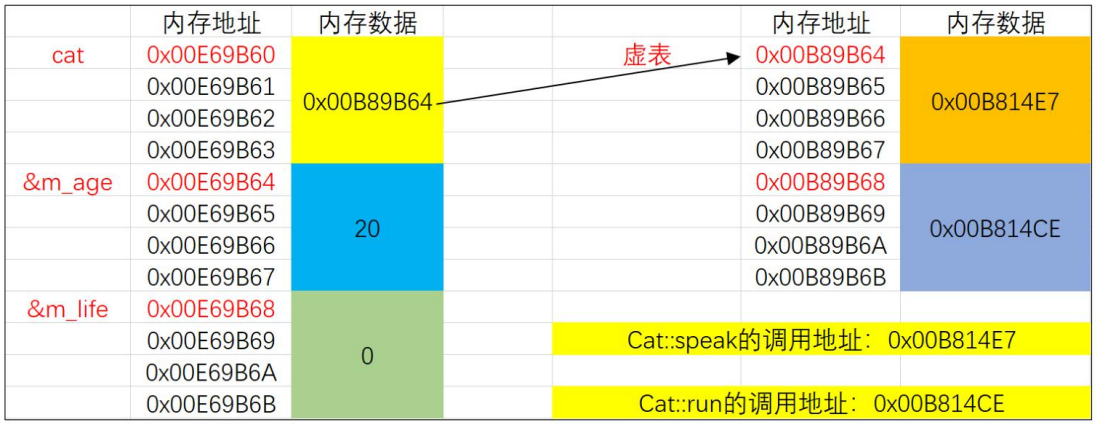
③ 只要在父类中声明为虚函数，子类中重写的函数也自动变成虚函数（也就是说子类中可以省略 virtual 关键字）

```
struct Animal {
    virtual void speak() {
        cout << "Animal::speak()" << endl;
    }
    virtual void run() {
        cout << "Animal::run()" << endl;
    }
};
struct Dog : Animal {
    // 重写（覆写、覆盖、override）
    void speak() {
        cout << "Dog::speak()" << endl;
    }
    void run() {
        cout << "Dog::run()" << endl;
    }
};
struct Cat : Animal {
    void speak() {
        cout << "Cat::speak()" << endl;
    }
    void run() {
        cout << "Cat::run()" << endl;
    }
};
struct Pig : Animal {
    void speak() {
        cout << "Pig::speak()" << endl;
    }
    void run() {
        cout << "Pig::run()" << endl;
    }
};
void liu(Animal *p) {
    p->speak();
    p->run();
}
int main() {

    liu(new Dog());
    liu(new Cat());
    liu(new Pig());


    getchar();
    return 0;
}
```

16-虚表

1. 原理

① 这个虚表里面存储着最终需要调用的虚函数地址，这个虚表也叫虚函数表。



17-虚析构函数

1. 如果存在父类指针指向子类对象的情况，应该将析构函数声明为虚函数（虚析构函数）
2. delete 父类指针时，才会调用子类的析构函数，保证析构的完整性

```
#include <iostream>
using namespace std;
class Animal{
public:
    Animal(){
        cout << "Animal::Animal" <<endl;
    }

    ~Animal(){
        cout << "Animal::~Animal" <<endl;
    }
};

class Pig:public Animal{
public:
    Pig(){
        cout << "Pig::Pig" <<endl;
    }

    ~Pig(){
        cout << "Pig::~Pig" <<endl;
    }
};
int main()
{
    Animal *p = new Pig;
    delete p;

    return 0;
}
```

可以声明为虚函数的函数

首先虚函数的调用需要依赖于虚函数表，而虚函数表的起始地址是保存在对象中的，因此虚函数的调用必须依赖于对象。因此可以得出：

1、可以被声明为虚函数的函数有：类中普通的成员函数、成员函数形式的操作符函数、析构函数；

2、不可以被声明成虚函数的函数有：静态的成员函数、全局函数形式的操作符函数、构造函数、全局函数。

18-纯虚函数

1. 作用

(1) 纯虚函数：没有函数体且初始化为 0 的虚函数，用来定义接口规范。

(2) 使用纯虚析构函数的类是抽象类。

2. 抽象类

(1) 含有纯虚函数的类，不可以实例化（不可以创建对象）

(2) 抽象类不能被用于实例化对象，它只能作为接口使用

(3) 抽象类也可以包含非纯虚函数、成员变量

(4) 如果父类是抽象类，子类没有完全重写纯虚函数，那么这个子类依然是抽象类

例子：定义

```
class Animal{
public:
    Animal(){
        cout << "Animal::Animal" <<endl;
    }

    virtual ~Animal(){
        cout << "Animal::~Animal" <<endl;
    }
    virtual void add() = 0;
};


class Pig:public Animal{
public:
    Pig(){
        cout << "Pig::Pig" <<endl;
    }
    void add()//实现
    {

    }

    ~Pig(){
        cout << "Pig::~Pig" <<endl;
    }

};
```

19-多继承

C++允许一个类可以有多个父类（不建议使用，会增加程序设计复杂度）

```
class Student {
public:
    int m_score;
    void study() {
        cout << "Student::study()" << endl;
    }
};

class Worker {
public:
    int m_salary;
    void work() {
        cout << "Worker::work()" << endl;
    }
};

class Undergraduate : public Student, public Worker {
public:
    int m_grade;
    void play() {
        cout << "Undergraduate::play()" << endl;
    }
};
```

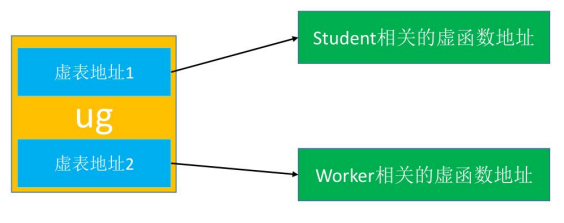
```
Undergraduate ug;
ug.m_score = 100;
ug.m_salary = 2000;
ug.m_grade = 4;
ug.study();
ug.work();
ug.play();
```

		内存地址	内存数据
&ug	&m_score	0x00E69B60	100
		0x00E69B61	
		0x00E69B62	
		0x00E69B63	
	&m_salary	0x00E69B64	2000
		0x00E69B65	
		0x00E69B66	
		0x00E69B67	
	&m_grade	0x00E69B68	4
		0x00E69B69	
		0x00E69B6A	
		0x00E69B6B	

1. 多继承构造函数调用顺序？
   (1) 答：根据继承的顺序从左往右。
2. 多继承如何访问相同名称变量？
   (1) 答：
      a. 多继承继承的变量会根据基础顺序依次排列。
      b. 使用域访问。
3. 多继承-虚函数
   (1) 答：如果子类继承的多个父类都有虚函数，那么子类对象就会产生对应的多张虚表

```
#include <iostream>

using namespace std;

class Student{
public:
    int m_score;
    void study() {
        cout << "student::study()" << endl;
    }
};

class Worker {
public:
    int m_salary;
    void work() {
        cout << "Worker::work()" < endl;
    }
};
```

```
//多继承
class Undergraduate : public student, public Worker
{
public:
    int m_grade;
    void play() {
        cout << "Undergraduate::play()" << endl;
    }
};
```



虚表

	内存地址	内存数据
&Student::m_age	0x00E69B60	20
	0x00E69B61	
	0x00E69B62	
	0x00E69B63	
&Worker::m_age	0x00E69B64	30
	0x00E69B65	
	0x00E69B66	
	0x00E69B67	
&Undergraduate::m_age	0x00E69B68	10
	0x00E69B69	
	0x00E69B6A	
	0x00E69B6B	

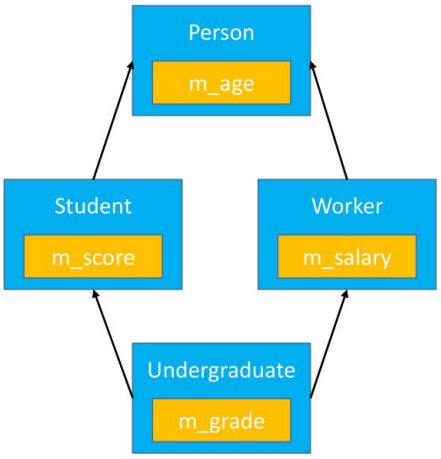
同名变量表

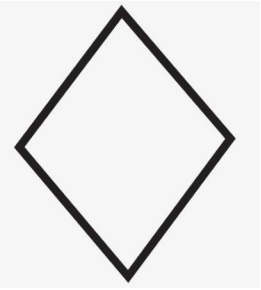
20-菱形继承

1.菱形继承带来的问题

①最底下子类从基类继承的成员变量冗余、重复

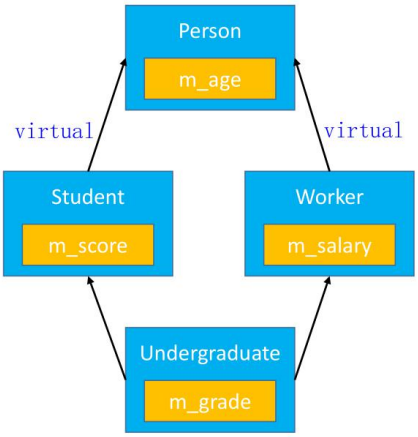
②最底下子类无法访问基类的成员，有二义性





2.解决菱形继承带来的问题

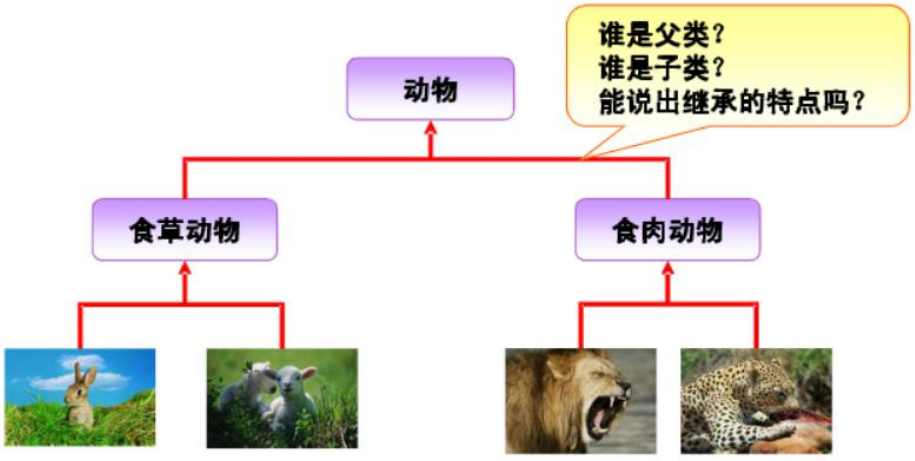
答：使用虚继承可以解决菱形继承带来的问题。

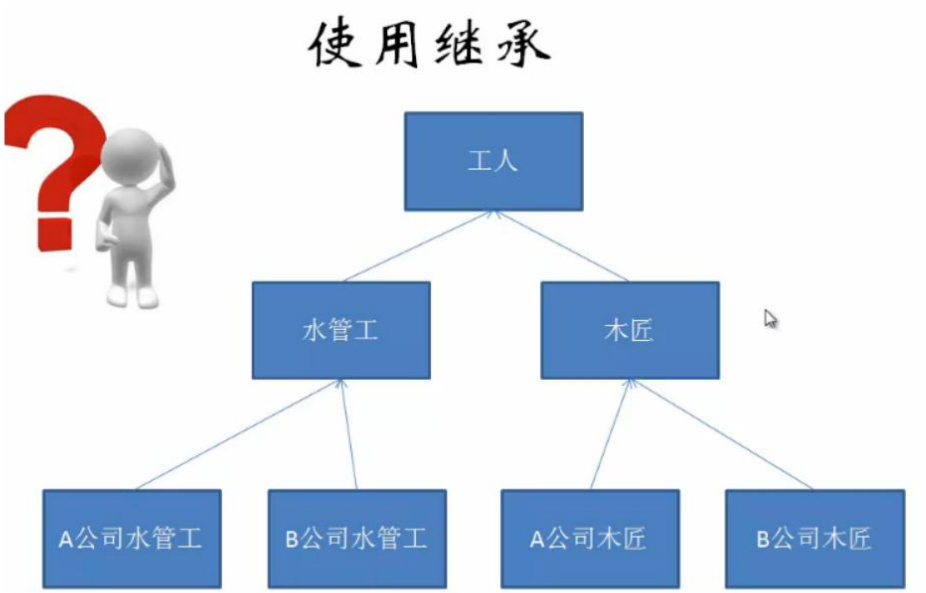



```
class Person {
public:
    int m_age = 1;};

class student: virtual public Person{
public:
    int m_score =2;
};

class Worker : virtual public Person {
public:
    int m_salary =3;
};

class Undergraduate : public student, public Worker {
public:
    int mgrade =4;
};

int main(int argc, char *argv[])
{
    Undergraduate under;
    under.m_age = 10;
    return 0;
}
```

21-多继承的应用


P图
兼容Windows、Linux、Android
刷抖音
订机票
3D渲染
处理文件
可跨终端使用
丰富的云应用
上网课
人机交互方式
全面拟人化
全场景人机交互系统
AliGenie5.0
支持微内核
操作系统
首创多模态自然唤醒
全双工自然对话
7种动态手势操控
智能看护、儿童模式
兼容不同配置的芯片
更安全
更灵活
广阔应用场景


谁是父类?
谁是子类?
能说出继承的特点吗?
动物
食草动物
食肉动物


使用继承
工人
水管工
木匠
A公司水管工
B公司水管工
A公司木匠
B公司木匠

22-static 成员应用

1.单例模式
2.工厂模式
3.桥接模式
4.迭代模式

23-单例模式

单例模式（Singleton Pattern）是 C++ 中最简单的设计模式之一。这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。

这种模式涉及到一个单一的类，该类负责创建自己的对象，同时确保只有单个对象被创建。这个类提供了一种访问其唯一的对象的方式，可以直接访问，不需要实例化该类的对象。

注意：
1、单例类只能有一个实例。
2、单例类必须自己创建自己的唯一实例。
3、单例类必须给所有其他对象提供这一实例。

介绍

意图：保证一个类仅有一个实例，并提供一个访问它的全局访问点。
主要解决：一个全局使用的类频繁地创建与销毁。
何时使用：当您想控制实例数目，节省系统资源的时候。
如何解决：判断系统是否已经有这个单例，如果有则返回，如果没有则创建。
关键代码：构造函数是私有的。
应用实例：

- 1、一个班级只有一个班主任。
- 2、Windows 是多进程多线程的，在操作一个文件的时候，就不可避免地出现多个进程或线程同时操作一个文件的现象，所以所有文件的处理必须通过唯一的实例来进行。
- 3、一些设备管理器常常设计为单例模式，比如一个电脑有两台打印机，在输出的时候就要处理不能两台打印机打印同一个文件。

优点：

- 1、在内存里只有一个实例，减少了内存的开销，尤其是频繁的创建和销毁实例（比如管理学院首页页面缓存）。
- 2、避免对资源的多重占用（比如写文件操作）。

缺点：没有接口，不能继承，与单一职责原则冲突，一个类应该只关心内部逻辑，而不关心外面怎么样来实例化。
使用场景：

- 1、要求生产唯一序列号。
- 2、WEB 中的计数器，不用每次刷新都在数据库里加一次，用单例先缓存起来。
- 3、创建的一个对象需要消耗的资源过多，比如 I/O 与数据库的连接等。

实现

我们将创建一个 *SingleObject* 类。*SingleObject* 类有它的私有构造函数和本身的一个静态实例。*SingleObject* 类提供了一个静态方法，供外界获取它的静态实例。*SingletonPatternDemo* 类使用 *SingleObject* 类来获取 *SingleObject* 对象。

24-const 成员

1.使用说明

1）const 成员：被 const 修饰的成员变量、非静态成员函数
2）必须初始化（类内部初始化），可以在声明的时候直接初始化赋值
3）非 static 的 const 成员变量还可以在初始化列表中初始化
4）const 成员函数（非静态）
5）const 关键字写在参数列表后面，函数的声明和实现都必须带 const
6）内部不能修改非 static 成员变量
7）内部只能调用 const 成员函数、static 成员函数
8）非 const 成员函数可以调用 const 成员函数
9）const 成员函数和非 const 成员函数构成重载
10）非 const 对象（指针）优先调用非 const 成员函数
11）非 const 对象（指针）优先调用非 const 成员函数
12）const 对象（指针）只能调用 const 成员函数、static 成员函数

```
class Car{
    const int mc_wheelsCount = 20; //常量成员
public :
    Car():mc_wheelsCount(10){}
    void run() const{                //常量函数
        cout << " run()" << endl;
    }
};
```

25-拷贝构造

- 拷贝构造函数是构造函数的一种
- 当利用已存在的对象创建一个新对象时（类似于拷贝），就会调用新对象的拷贝构造函数进行初始化
- 拷贝构造函数的格式是固定的，接收一个 const 引用作为参数

```
class Car {
    int m_price;
public:
    Car(int price = 0) :m_price(price) { }
    Car(const Car &car) {
        this->m_price = car.m_price;
    }
};
```

调用情况：

①通过使用另一个同类型的对象来初始化新创建的对象。

②复制对象把它作为参数传递给函数。

③复制对象，并从函数返回这个对象。

26-浅 Vs 深 拷贝

1.浅拷贝：

编译器默认的提供的拷贝是浅拷贝（shallow copy）

1.将一个对象中所有成员变量的值拷贝到另一个对象

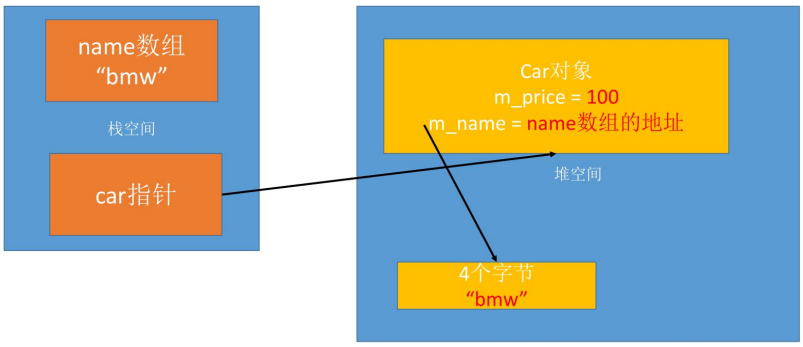
2.如果某个成员变量是个指针，只会拷贝指针中存储的地址值，并不会拷贝指针指向的内存空间

3.可能会导致堆空间多次 free 的问题

2.深拷贝：

如果需要实现深拷贝（deep copy），就需要自定义拷贝构造函数

1. 将指针类型的成员变量所指向的内存空间，拷贝到新的内存空间



深拷贝代码示例

```
class Car {
    int m_price;
    char *m_name;
    void copyName(const char *name) {
        if (name == NULL) return;

        this->m_name = new char[strlen(name) + 1]{};
        strcpy(this->m_name, name);
    }
public:
    Car(int price = 0, const char *name = NULL) :m_price(price) {
        copyName(name);
    }
    Car(const Car &car) {
        this->m_price = car.m_price;

        copyName(car.m_name);
    }
    ~Car() {
        if (this->m_name != NULL) {
            delete[] this->m_name;
        }
    }
};
```

27-对象类型的参数和返值

1.对象型参数和返回值

使用对象类型作为函数的参数或者返回值，可能会产生一些不必要的中间对象

```
class Car {
    int m_price;
public:
    Car() { }
    Car(int price) :m_price(price) { }
    Car(const Car &car) :m_price(car.m_price) { }
};

void test1(Car car) {

}

Car test2() {
    Car car(20); // Car(int price)
    return car;
}
```

```
Car car1(10); // Car(int price)
test1(car1); // Car(const Car &car)

Car car2 = test2(); // Car(const Car &car)

Car car3(30); // Car(int price)
car3 = test2(); // Car(const Car &car)
```

2.匿名对象（临时对象）

匿名对象：没有变量名、没有被指针指向的对象，用完后马上调用析构

```
void test1(Car car) {

}

Car test2() {
    return Car(60);
}
```

```
Car(10); // Car(int price)
Car(20).display(); // Car(int price)

Car car1 = Car(30); // Car(int price)

test1(Car(40)); // Car(int price)

Car car3(50); // Car(int price)
car3 = test2(); // Car(int price)
```

28-编译器生成的构造函数

- C++的编译器在某些特定的情况下，会给类自动生成无参的构造函数，比如
  成员变量在声明的同时进行了初始化
  有定义虚函数
  虚继承了其他类
  包含了对象类型的成员，且这个成员有构造函数（编译器生成或自定义）
  父类有构造函数（编译器生成或自定义）
- 总结一下

对象创建后，需要做一些额外操作时（比如内存操作、函数调用），编译器一般都会为其自动生成无参的构造函数

29-内部类

- 1.如果将类 A 定义在类 B 的内部，那么类 A 就是一个内部类（嵌套类）
- 2.内部类的特点

  1.支持 public、protected、private 权限。

  2.成员函数可以直接访问其外部类对象的所有成员（反过来则不行）。

  3.成员函数可以直接不带类名、对象名访问其外部类的 static 成员。

  4.不会影响外部类的内存布局。

  5.可以在外部类内部声明，在外部类外面进行定义。

```
class B{
public :
    int name;

    class A{
    public:
        int age;
    };
};
```

```
int main()
{
    B::A A;//定义
    A.age = 10;

    cout << A.age << endl;

    return 0;
}
```

声明和实现分离

```
class Point {
    class Math {
        void test();
    };
};

void Point::Math::test() {

}
```

```
class Point {
    class Math;
};

class Point::Math {
    void test() {

    }
};
```

```
class Point {
    class Math;
};

class Point::Math {
    void test();
};

void Point::Math::test() {

}
```

30-局部类

- 在一个函数内部定义的类，称为局部类
- 局部类的特点

  作用域仅限于所在的函数内部

  其所有的成员必须定义在类内部，不允许定义 static 成员变量

  成员函数不能直接访问函数的局部变量（static 变量除外）

```
int m_age1 = 0;

void test() {
    static int s_age2 = 0;
    int age3 = 0;

    class Point {
        int m_x;
        int m_y;
    public:
        static void display() {
            m_age1 = 10;
            s_age2 = 20;
            age3 = 30;
        }
    };

    Point::display();
}
```

31-仿函数

1.仿函数本质：将一个对象当作一个函数一样来使用

对比普通函数，它作为对象可以保存状态

将小括号运算符重载即可。

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

```
Sum sum;
cout << sum(20, 30) << endl;
```

32-模板

1. 泛型，是一种将类型参数化以达到代码复用的技术，C++中使用模板来实现泛型

2. 模板的使用

① template <typename\class T>

② typename 和 class 是等价的

3. 模板的生成

①模板没有被使用时，是不会被实例化出来的

②模板的声明和实现如果分离到.h 和.cpp 中，会导致链接错误

③一般将模板的声明和实现统一放到一个.h 文件中

4. 函数模板与模板函数的区别如下：

(1) 函数模板不是一个函数，而是一组函数的模板，在定义中使用了参数类型。

(2) 模板函数是一种实实在在的函数定义，它的函数体与某个模板函数的函数体相同。

函数模板	类模板

template <class 形参名, class 形参名, ...>
返回类型 函数名(参数列表)
{
 	
函数体
 	
}
	template<class 形参名, class 形参名, …>
	class 类名{ ... };

33-类模板

1. 一些类主要用于存储和组织数据元素；
    1、类模板就是为了数据结构而诞生的；
    2、类中数据组织的方式和数据元素的 具体类型无关；
    3、如：数组类、链表类、Stack 类、Queue 类等；
2. 模板的思想
    1、应用于类，使得类的实现不关注数据元素的具体类型，而只关注类所需实现的功能；
3. 类模板说明使用
    1、以相同的方式处理不同的类型；
    2、在类声明前使用 template 进行标识；
    3、< typename T > 用于说明类中使用的泛指类型 T；

4.类模板的应用
    1、只能显示指定具体类型，无法自动推导；
    2、使用具体类型 < Type > 定义对象；

5.类模板示例

```
template <class Item>
class List {
    int m_size;
    int m_capacity;
    Item *m_data;
public:
    List(int capacity = 0);
    ~List();
    void add(Item value);
    Item get(int index);
    int size();
    void display();
};
```

```
template <class Item>
void List<Item>::add(Item value) {
    if (this->m_size == this->m_capacity) {
        cout << "数组已满" << endl;
        return;
    }

    this->m_data[this->m_size++] = value;
}

template <class Item>
Item List<Item>::get(int index) {
    if (index < 0 || index >= this->m_size) return NULL;

    return this->m_data[index];
}

template <class Item>
int List<Item>::size() {
    return this->m_size;
}
```

6.小结

1、泛型编程的思想可以应用于类；
2、类模板以相同的方式处理不同类型的数据；
3、类模板非常适用于编写数据结构相关的代码；
4、类模板在使用时只能显示指定类型；

34-类型转换

- C 语言风格的类型转换符
  (type)expression
  type(expression)
- C++中有 4 个类型转换符
  static_cast：常用于基本数据类型的转换、非 const 转成 const
  dynamic_cast：一般用于多态类型的转换，有运行时安全检测
  reinterpret_cast：可以将指针和整数互相转换
  const_cast： 一般用于去除 const 属性，将 const 转换成非 const
  使用格式：xx_cast<type>(expression)

1. const_cast

(1) 一般用于去除 const 属性，将 const 转换成非 const

(2) 用于修改类型的 const / volatile 属性

(3) 目标类型必须与源类型相同

提示：如果 a 本身就是 const，就是那块内存被定义为 const 的话，这样的结果是未定义的，在我的编译器和机器上，这个被定义为无法改变

可改变值	无法改变

#include<iostream>
using namespace std;
int main()
{
int a = 12;
const int *ap = &a;
int* tmp =
const_cast<int*>(ap);
*tmp = 11;
cout<< a << endl;
 	
return 0;
}
	#include<iostream>
	using namespace std;
	int main()
	{
	const int a = 12;//本身 const
	const int *ap = &a;
	int* tmp =
	const_cast<int*>(ap);
	*tmp = 11;
	cout<< a << endl;
	
	return 0;
	}

2. dynamic_cast

(1) 一般用于**多态**类型的转换，有运行时**安全检测**

(2) 不安全返回 NULL，安全则返回他的地址

(3) dynamic_cast 是无法用于非多态的对象

(4) 不过关于 dynamic_cast 运算符啊，我们最好还是少用，毕竟谷歌爸爸的 Google Style Guides 里头可是说，当你使用了 dynamic_cast 运算符，代表着你的设计不合理，需要重新设计~

```cpp
#include <iostream>
using namespace std;
class Person{
    // 一定要是  virtual
    virtual int test(){return 0;}
};

class Car{
    virtual int test(){    return 1;}
};

class Student : public Person{

};

int main()
{
    Person *p1 =new Person();
    Person *p2 = new Student();
    //不安全，返回 NULL
    Student *stu1 = dynamic_cast<Student *>(p1);
    Car *car = dynamic_cast<Car *>(p1);
    //安全返回地址值
    Student *stu2 = dynamic_cast<Student *>(p2);
    return 0;
}
```

①一定要 virtual
②安全返回有效地址
③不安全返回 NULL

3. static_cast

(1) 对比 dynamic_cast，缺乏运行时安全检测

(2) 不能交叉转换（不是同一继承体系的，无法转换）

(3) 常用于基本数据类型的转换、非 const 转成 const

(4) 适用范围较广

(5) 其实与 C 语言中的强制类型转换而已。

```
#include <iostream>
using namespace std;
class Person{ };

class Car{ };

class Student : public Person{ };

int main()
{
    Person *p1 =new Person();
    Person *p2 = new Student();

    Student *stu1 = static_cast<Student *>(p1);
    Student *stu2 = static_cast<Student *>(p2);
    //Car *car = static_cast<Car *>(p1);//不是同一继承体系 ×

    return 0;
}
```

4. reinterpret_cast

(1) 属于比较底层的强制转换，没有任何类型检查和格式转换，仅仅是简单的二进制数据拷贝

(2) 可以交叉转换

(3) 可以将指针和整数互相转换

(4) C++ 标准不允许将函数指针转换成对象指针，但有些编译器支持。这种转换提供了很强的灵活性，但转换的安全性只能由程序员的细心来保证了。

```
    Person *p1 = new Person();
    Person *p2 = new Student();
    Student *stu1 =
reinterpret_cast<Student *>(p1);
    Student *stu2 =
reinterpret_cast<Student *>(p2);
    Car *car = reinterpret_cast<Car
*>(p1);

    int *p = reinterpret_cast<int
*>(100);
    int num =
reinterpret_cast<int>(p);
```

35-智能指针

1. 传统指针存在的问题
   ① 需要手动管理内存
   ② 容易发生内存泄露（忘记释放、出现异常等）
   ③ 释放之后产生野指针

2. 智能指针就是为了解决传统指针存在的问题
   ① auto_ptr：属于 C++98 标准，在 C++11 中已经不推荐使用（有缺陷，比如不能用于数组）
   ② shared_ptr：属于 C++11 标准
   ③ unique_ptr：属于 C++11 标准
   ④ weak_ptr 会对一个对象产生弱引用，解决循环引用