

类与对象

北京邮电大学

网络与交换技术国家重点实验室

宽带网研究中心

主要内容

➤ 面向对象的程序设计

➤ C++的类与对象

- 类的定义

- 成员

- 构造函数

- 析构函数

面向对象的程序设计

面向过程与面向对象

➤ 面向对象程序设计 (OOP)

- ❑ Object Oriented Programming
- ❑ 目前一种主流程序设计方法

➤ C++是混合编程语言

- ❑ C++保持与 C 相兼容
- ❑ 既支持面向对象程序设计方法，也支持面向过程程序设计方法，是一种混合编程语言。

面向过程程序设计方法

➤ 机制

- 首先定义所要实现的功能
- 然后为这些功能设计所必要的步骤或过程
 - ⊗ 解决问题的重点是如何实现过程的细节
 - ⊗ 数据与操作这些数据的过程分离
 - ⊗ 围绕功能（过程或操作）实现来设计程序
- 采用自顶向下（瀑布式的流程），功能分解法
 - ⊗ 采用逐步求精的方法来组织程序的结构
- 程序组成形式 —— 主模块 + 子模块
 - ⊗ 它们之间以数据作为连接纽带
 - ⊗ 程序 = 算法 + 数据结构
- 数据处于次要的地位，而过程是关心的重点

面向过程程序设计

➤ 缺点：

- ❑ 数据与操作这些数据的过程分离，一旦问题改变（数据结构发生变化），就需要重新修改问题的解决方法（操作数据的过程）
- ❑ 软件维护成本高
- ❑ 不利于代码重用（re-use）
 - ⊗ 以函数（功能、过程）的方式实现代码重用，效率低
 - ⊗ 理想的方式：问题的解决方案能够重用
- ❑ 不适于中大型、巨型软件程序设计

面向对象程序设计

➤ 面向对象概念

- ❑ 是一种解决问题的方法或观点
- ❑ 认为自然界是由一组彼此相关联并相互作用（通信）的实体所组成，称为对象（Object）

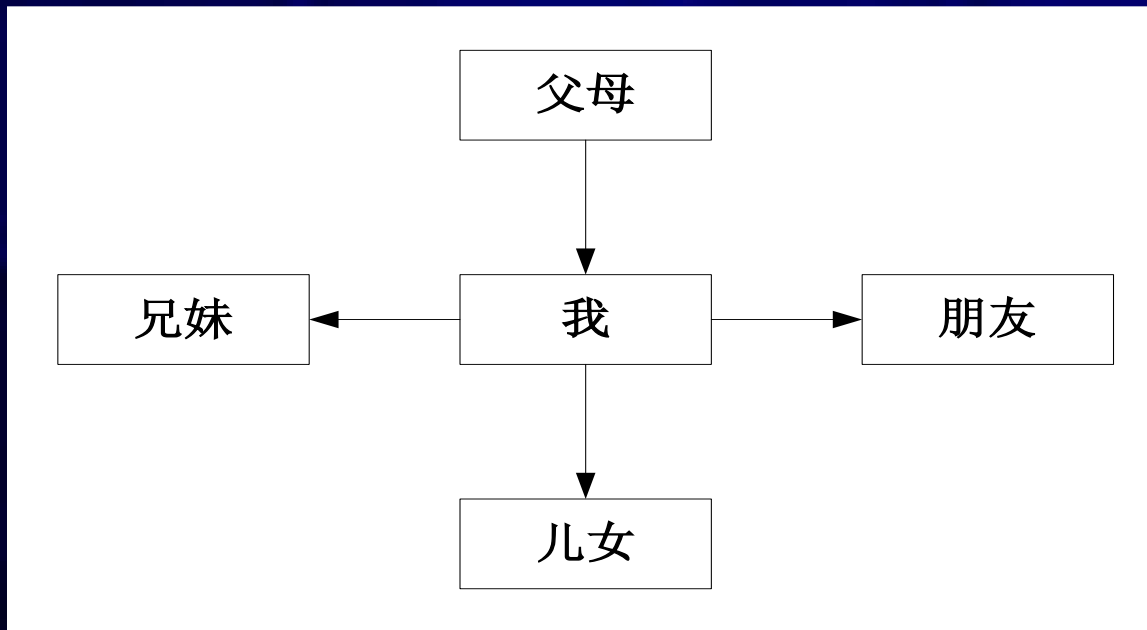
☒ 对象化的表示更接近于对世界的自然描述

➤ 程序员使用面向对象的观点来分析问题

- ❑ 即将所要解决的问题转化为程序中的对象——任何问题在程序中都映射为对象
- ❑ 找出问题的属性（数据描述）与操作方法（通过函数来体现）
- ❑ 然后用计算机语言来描述问题，最后在计算机中加以处理

以“人”为例：

- 每一个“人”的个体都是“人类”的一个实例（对象）
- 每一个“人”有自己的属性
 - ❑ 姓名、年龄、职业、...
- 个体之间存在着各种关系
 - ❑ 个体之间能够相互作用（操作）：交谈、合作、...



面向对象程序设计

➤ 程序设计

- ❑ 程序员在解决问题时应具有高度的概括、抽象能力
- ❑ 根据自然实体（待解决的问题）的属性进行分类和抽象
- ❑ 准确描述实体类的属性和行为
- ❑ 对问题进行分析和抽象，使用程序设计语言中的类和类之间的关系来描述待解决的问题及其相关性
- ❑ 对类进行具体化，产生出对应的问题对象，以消息传递的机制来组织对象之间的相互作用和操作

➤ 程序组成形式：对象 + 消息

- ❑ 对象与对象之间通过消息作为连接相互驱动
- ❑ 对象的行为（操作）体现为对消息的处理方式
- ❑ 对象（问题）之间的关系是编程关心的重点，而对象功能实现细节则处于次要的地位，并且通常被封装

面向对象程序设计

➤ 面向对象程序设计的优越性

- ❑ 提高软件质量：实现数据与方法的封装，通过方法来操作改变数据，提高了数据访问的安全性
- ❑ 易于软件维护
- ❑ 支持软件重用，大大提高软件生产的效率
- ❑ 实现可重用的软件组件，实现软件设计的产业化
 - ✉ 由于程序是类的集合从而可以根据问题的相关性来组装程序
 - ✉ 而面向过程程序设计则是函数的集合，零散不便于代码重用

总结：面向过程与面向对象

➤ 面向过程的程序设计

- Program = Algorithms + Data Structures

- 问题求解的方法：向过程传递数据

➤ 面向对象的程序设计

- 问题求解的方法：向对象发送消息

➤ 面向过程的程序设计语言与面向对象的程序设计语言

- Modula-2、Ada

 - ☒ 支持数据隐藏和数据封装

- C++、Java

 - ☒ 支持数据隐藏和数据封装

 - ☒ 支持继承和多态、支持重用

OOP 基本手段——抽象

- 对具体问题（对象）进行分类概括，提取出这一类对象的共同性质并且加以描述的过程。
- 编程的要求：
 - ❑ 先注意问题的本质及描述，其次是实现过程或细节。它直接决定程序的优劣——类的定义及组成元素；
 - ❑ 所涉及到的主要内容：
 - ⊗ 数据抽象（属性）——描述某类对象的属性或状态（对象相互区别的物理量）；
 - ⊗ 行为抽象（方法）——描述某类对象的共有的行为特征或具有的功能。
 - ❑ 抽象的实现：在 C++ 中通过类 class 来声明

OOP 基本手段——抽象

➤ 例：对现实中钟表类的抽象

□ 属性抽象：所有钟表都具有以下属性

```
int Hour;  
int Minute;  
int Second;
```

□ 行为（操作）抽象：钟表对外（其他对象）体现的行为

```
SetTime(int Hour, int Minute, int Second);  
ShowTime();
```

□ 注：钟表内部的如何保证准确的时间是钟表内部实现的细节，使用钟表功能的外部对象并不关心这些细节
在 OOP 中，这些细节也是次要的

OOP 基本手段——抽象

➤ 例：对人的抽象得到“人类”

□ 属性抽象

```
char *name;  
int age;  
int id;  
...
```

□ 行为的抽象：

```
Eat();  
Speak();  
Work();  
...
```

OOP 基本手段——封装

➤ 将抽象出的属性和行为结合，作为一个整体来对待

- ❑ 使用的工具：在 C++ 中使用类 class 来进行封装
- ❑ 属性表示为类中的数据成员
- ❑ 行为表示为类中的成员函数
- ❑ 封装机制将这二类成员组合在一起，形成问题的类，类实例化就可以产生类的实体——对象

➤ 封装的目的：

- ❑ 增强安全性和简化编程，使用者不必(甚至是不能)了解具体的实现细节，而只需要通过类外部接口，以特定的访问权限，来使用类的成员
- ❑ 高度模块化，易于生产软件组件（例如控件等）

➤ 实现封装：定义 class

数据封装和数据隐藏

➤ 数据封装和数据隐藏

❑ Encapsulation

- ✉ 将基本数据和能够对这些基本数据进行的操作（方法）结合起来
- ✉ 对这些数据的操作只能通过指定的方法（method）来进行
- ✉ 这些操作方法也称为接口（Interface）

❑ Data hiding

- ✉ 将基本数据的内部结构对外隐藏起来，使内部数据结构对外具有不可访问性

❑ 数据封装和数据隐藏的优点：可维护性

- ✉ 将数据及对数据的处理封装起来，对外隐藏实际的数据结构及对数据处理的实现，对外提供一致的接口
- ✉ 当数据及操作的内部实现发生改变时，只要接口保持一致，将不会影响软件的其他部分

数据封装和数据隐藏

➤ C++ 的类是一种用于数据封装和数据隐藏的工具

□ 类 (class) 定义用户的抽象数据结构

- ⊗ 基本数据和抽象数据结构 (对象), 即类的成员变量
- ⊗ 对这些数据的操作方法 (类的成员函数)
- ⊗ 所有的成员函数中, 暴露给外部、可被外部使用者调用的方式 (public 方法) 是类的接口 (interface) 方法

□ 对象 (Object) 是类的实例

- ⊗ 一个对象是一个具有内存映像、符合类结构的实体
- ⊗ 使用者对对象实体的操作, 通过调用该对象的接口方法来实现
- ⊗ 调用对象的接口方法, 可以看作是向该对象发送了一条消息; 接口方法的实现 (对内部数据进行操作), 是对象对该消息进行处理的过程

OOP 基本手段——封装

► 实例：

```
class Clock
{
    public:                                // 外部接口
        void SetTime(int NewHour, int NewMin, int NewSec);
        void ShowTime(void);

    private:                               // 内部属性
        int    Hour;
        int    Minute;
        int    Second;
        void Run(void);                    // 内部行为
};
```

成员变量

成员函数

成员函数

成员函数

封装的特征

➤ 有一定的边界

- ❑ 所有的内部变化都限制在此边界内；

➤ 有外部接口（类中的 public 成员）

- ❑ 此对象利用它与其他对象发生关联，进行操作
- ❑ 向对象发送消息：调用对象的 public 成员函数

➤ 有特定的数据保护或访问权限

- ❑ 例如：类中的 private 成员，在对象外部不允许访问或修改
- ❑ 保护内部细节，内部实现的变化不会影响外部对对象的访问

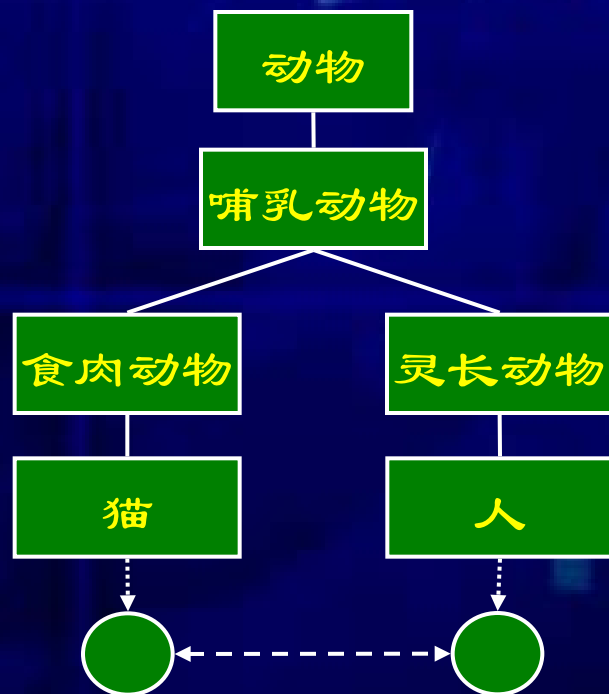
OOP 基本手段——继承与派生

➤ 在 C++ 中，支持分类层次的一种机制

- 允许程序员定义一个新的类，新类在原有类属性和行为的基础上，定义更具体、更详细的属性和行为

➤ 现实意义：

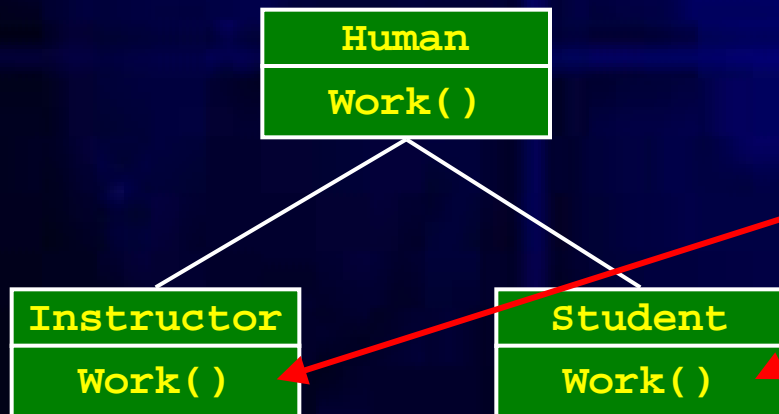
- 现实世界中对象并不是孤立的，而是相互关联的（横向）
- 而每一个对象所属的类在纵向上体现为从属或继承关系，这为 OOP 的继承提供现实基础
- 通过继承可以实现对现有软件的重用、扩展
- 继承通过类的派生来实现



OOP 基本手段——多态性

➤ 在类的派生过程中，允许不同的派生类对同一个操作具有不同的行为实现

- ❑ 多态的体现：不同的派生类中，对同一成员函数采用不同的实现方式，在调用时总是能够保证正确的方法被调用
- ❑ 目的：行为与标识统一
- ❑ 实现：虚函数与重写（override）函数



```
Instructor A;  
Student B;  
Human *p;
```

```
p = &A;  
p->Work();  
p = &B;  
p->Work();
```

C++ 类与对象

C++ 类

- 类是具有相同属性和行为的一组对象的集合，它为属于该类的全部对象提供了统一的抽象描述，其内部包括属性和行为两个主要部分。
- 利用类可以实现数据的封装、隐藏、继承与派生。
- 利用类易于编写大型复杂程序，其模块化程度比C中采用函数更高。

类的声明形式

类是一种用户自定义类型，与 struct 类似

声明形式：

```
class 类名称
{
    public:
        公有成员（外部接口）
    private:
        私有成员
    protected:
        保护型成员
};
```


类成员的访问类型

➤ public

- 声明后面的成员变量或成员函数能够被外部的函数或对象直接访问，用于定义类的外部接口

➤ private

- 声明后面的成员变量或成员函数只允许在本类对象的成员函数中被访问，不允许外部访问
- 用于定义类的内部数据和代码实现
- 省略情况下，若前面没有 public/private/protected 关键字声明，class 中的成员都是 private 类型

➤ protected

- 声明后面的成员变量或成员函数只允许在本类或其派生类对象的成员函数中被访问，不允许外部访问

类定义的例子

```
class Clock
{
    public:
        void SetTime(int NewHour, int NewMin,int NewSec);
        void ShowTime();
    private:
        int Hour, Minute, Second;
};

void Clock::SetTime(int NewHour, int NewMin, int NewSec)
{
    Hour    = NewHour;
    Minute  = NewMin;
    Second  = NewSec;
}

void Clock::ShowTime()
{
    cout << Hour << ":" << Minute << ":" << Second;
}
```

类成员的定义

➤ 数据成员（属性、成员变量）

- ❑ 数据与一般的变量声明相同，但需要放在类的声明体中

➤ 成员函数

- ❑ 在类说明中给出原型，在类说明以外给出函数的实现
- ❑ 在定义成员函数时，需要在函数名前使用类名加以限定
`Clock::SetTime`
- ❑ 也可直接在类说明中给出函数体，形成内联（inline）成员函数
- ❑ 允许声明重载函数和带默认形参值的函数

内联成员函数

➤ 为了提高运行时的效率，对于较简单的成员函数可以声明为内联形式。

- ❑ 将函数体放在类的声明中
- ❑ 使用inline关键字

```
class Point
{
    public:
        void Init(int initX,int initY)
        {
            X = initX;
            Y = initY;
        };
        int GetX() {return X;};
        int GetY() {return Y;};
    private:
        int X,Y;
};
```

内联成员函数

```
class Point
{
public:
    void Init(int initX,int initY);
    int GetX();
    int GetY();
private:
    int X,Y;
};
inline void Point::Init(int initX,int initY)
{
    X=initX;
    Y=initY;
}
inline int Point::GetX()
{
    return X;
}
inline int Point::GetY()
{
    return Y;
}
```

class 与 struct

➤ 在 C++ 中，class 与 struct 作用相同，差别仅在于：

- ❑ 不加访问类型限制（public/private/protected）时，class 成员的缺省的访问限制都是 private
- ❑ 不加访问类型限制（public/private/protected）时，struct 成员的缺省的访问限制都是 public

C++对象

➤ 类的对象是该类的某一特定实例，是类型为该类的一个变量

➤ 声明形式：

类名 对象名；

➤ 例：

Clock myClock;

对象成员的访问方式

➤ 对对象成员的访问，与访问结构成员类似

□ 使用“对象名.成员名”方式访问

MyClock.ShowTime(); // 访问成员函数

MyClock.Hour = 12; // 访问成员变量，注意访问限制

□ 注意：

⊗ 从类的外部访问该类的对象时，需要受到访问类型（public/private/protected）的限制

⊗ 在同一个对象的成员函数中访问本对象的成员，可以省略“对象名.”，直接使用成员名

类成员访问

```
class Clock
{
    public:
        void SetTime(int NewHour, int NewMin,int NewSec);
        void SetTime(Clock OtherClock);
        void ShowTime();
    private:
        int Hour, Minute, Second;
};

...
void Clock::SetTime(Clock OtherClock)
{
    Hour    = OtherClock.Hour;
    Minute  = OtherClock.Minute;
    Second  = OtherClock.Second;
}

main()
{
    Clock Clock1, Clock2;
    Clock1.Hour = 12;
    Clock1.SetTime(12,0,0);
    Clock2.SetTime(Clock2);
}
```



C++类中的自引用

➤ 在类的成员函数的上下文中，存在一个指向被调用的对象的指针（用关键字 `this` 来指出）

- ❑ 调用同一类的不同对象的成员函数时，其上下文的 `this` 指针不同，分别指向不同对象
- ❑ 尽管这些对象采用相同的成员函数代码，但实际操作是不同的对象成员的数据成员
- ❑ `this` 是类成员函数的一个
隐含参数
(除了静态成员函数外)

```
void CCounter::Increase()  
{  
    iCounterValue ++;  
    // 等同于  
    // this->iCounterValue ++;  
}  
...
```

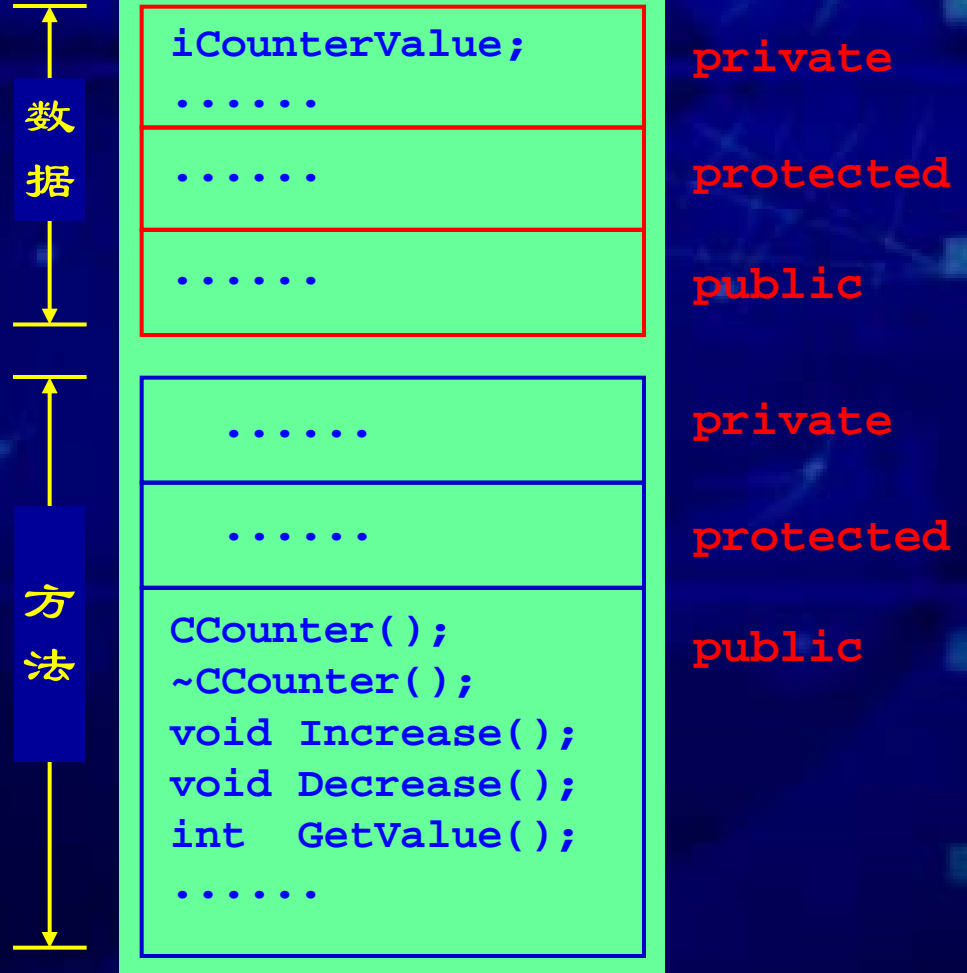
C++ 类的概念结构

➤ C++ 类定义抽象数据结构

```
class CCounter
{
private:
    int  iCounterValue;
    ...
protected:
    ...
public:
    CCounter();
    ~Ccounter();

    void Increase();
    void Decrease();
    int  GetValue();
    ...
};
```

```
CCounter counter;
```



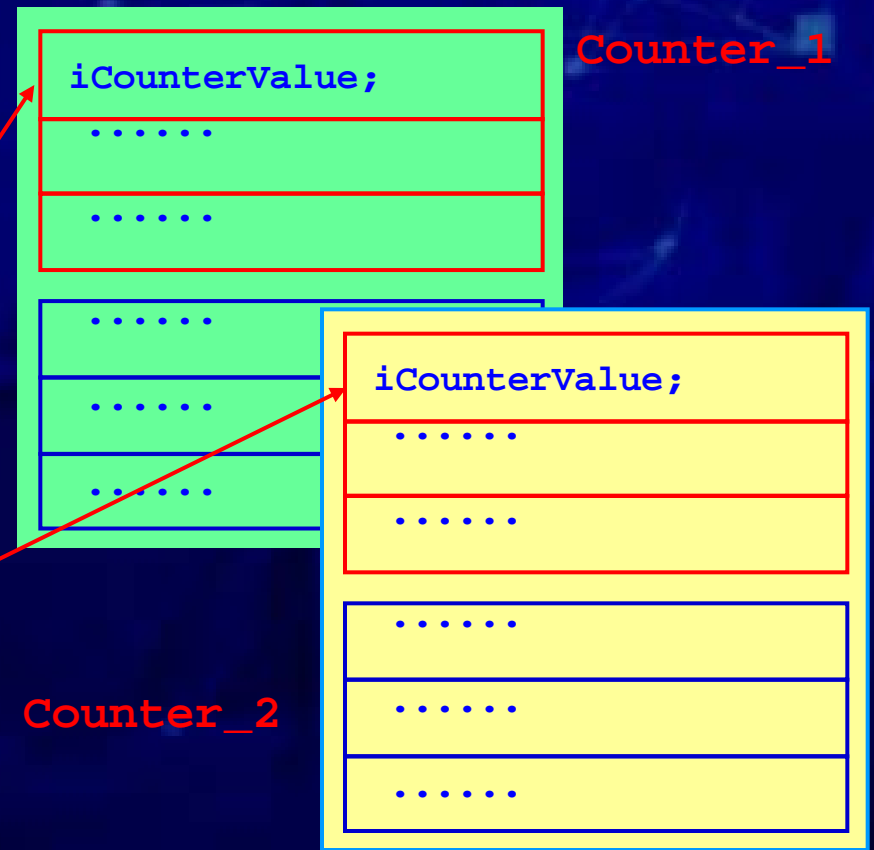
C++类的概念结构

➤ 属于一个类的所有实例（对象）具有相同的结构

□ 当调用一个对象的方法（向该对象发送消息）时，其操作（访问）的目标是本对象中的数据成员，而不对其他对象的数据成员进行操作

☒ 这种方式可以防止程序中对非相关数据的意外修改和访问，提高程序的安全性和健壮性

```
void CCounter::Increase()  
{  
    iCounterValue ++;  
}  
...  
main()  
{  
    CCounter counter_1;  
    CCounter counter_2;  
    ...  
    counter_1.Increase();  
    counter_2.Increase();  
    ...  
}
```



动态存储分配

C++中引入的新算符

new/delete



动态申请内存操作符 new

➤ 语法:

new 类型名T (初值列表)

➤ 功能

- ❑ 在程序执行期间，动态申请一块用于存放类型T的对象的内存空间，并依初值列表赋以初值
- ❑ 动态内存空间是由操作系统管理的，可供所有程序共享申请的内存空间，称为堆 (heap)
- ❑ new 使程序员不必调用库函数，如 malloc 等

➤ 结果值

- ❑ 成功：T类型的指针，指向新分配的内存
- ❑ 失败：0 (NULL)

释放内存操作符 delete

➤ 语法:

delete 指针表达式

➤ 功能:

- ❑ 释放指针表达式所指向的动态内存空间，归还给操作系统
- ❑ new 使程序员不必要调用库函数，如 free 等

例：申请和释放内存

```
int *p;  
p = new int;           // 申请内存存放一个 int 类型, 地址返回到 p 中  
delete p;              // 释放 p 指向的内存空间 (一个 int 类型)  
p = new int(2);        // 申请内存, 初始化为 2  
delete p;  
  
p = new int[100];      // 申请一个 int 类型的数组, 长度为 100, 返回数组首  
    地址  
delete [] p;           // 释放数组
```


类的构造函数

➤ `int a=0;` 或 `int a;`

```
int a;  
int function()  
{  
    int b;  
    a=b;        // a=??  
}
```

➤ 如何为自定义的类对象实现像内建数据类型的初始化功能而且可以避免忘记赋初值发生错误？

➤ 构造函数的作用

- ❑ 在类的对象被创建的时候，对被创建的对象的数据成员的值进行特定的初始化，或者说将对象初始化为一个特定的状态
- ❑ 在对象创建时由系统自动调用类的构造函数创建对象
- ❑ 如果未声明构造函数，则系统自动产生出一个类的默认构造函数
- ❑ 构造函数也允许为内联函数、重载函数、带默认形参值的函数
- ❑ 构造函数没有返回值

✉ 也不是返回 `void`

构造函数举例

```
class Clock
{
public:
    Clock (int Hour, int Min, int Sec);           // 构造函数, 与类名相同
    ...
private:
    int Hour, Minute, Second;
};

Clock::Clock(int Hour, int Min, int Sec)
{
    this->Hour = Hour;
    Minute    = Min;
    Second    = Sec;
}

int main()
{
    Clock  MyClock(0,0,0);                       // 调用构造函数, 初始化 MyClock
    MyClock.ShowTime();
    return 0;
}
```

构造函数的重载

```
class Clock
{
public:
    Clock();
    Clock(int Hour, int Min, int Sec);
    ...
};

Clock::Clock()
{
    Hour = Minute = Second = 0;
}

Clock::Clock(int Hour, int Min, int Sec)
{
    ...
}

int main()
{
    Clock MyClock(0,0,0);
    Clock MyClock2;
    return 0;
}
```

// 无参数的构造函数
// 构造函数

The diagram consists of red lines and arrows. A vertical line on the right side of the code block has two horizontal arrows pointing left. The top arrow points to the `Clock()` constructor declaration in the `public` section of the `Clock` class. The bottom arrow points to the `Clock(int Hour, int Min, int Sec)` constructor declaration. From the bottom of this vertical line, a horizontal line extends to the left, then turns down to a horizontal line that connects to the `Clock MyClock(0,0,0);` line in the `main` function. Another horizontal line extends from the bottom of the vertical line to the left, then turns down to a horizontal line that connects to the `Clock MyClock2;` line in the `main` function.

拷贝构造函数

- `int a=0; int b=a;` 对自定义的类对象如何实现第二条语句？
- 拷贝构造函数是一种特殊的构造函数，其形参为本类的对象引用。
- 当用类的一个对象去初始化（创建）该类的另一个对象时系统自动调用拷贝构造函数实现拷贝赋值。

```
class 类名
{
    public :
        类名 (类名 &对象名) ;    //拷贝构造函数
    ...
};
```

拷贝构造函数

```
#include <iostream>
using namespace std;
class Point
{
    public:
        Point(int xx=0,int yy=0)
            {X=xx; Y=yy;};
        Point(Point& p);
        int GetX() {return X;};
        int GetY() {return Y;};
    private:
        int X,Y;
};

Point::Point (Point& p)
{
    X=p.X;
    Y=p.Y;
    cout << "Copy Constructor"
         << endl;
}
```

```
void func1(Point p)
{
    cout << p.GetX() <<endl;
}

Point func2()
{
    Point A(1,2);
    return A; //调用拷贝构造函数
}

int main()
{
    Point A(1,2);
    Point B(A); //拷贝构造函数被调用
                //或 Point B=A;

    func1(A);    // 调用拷贝构造函数

    Point C = func2();
}
```

拷贝构造函数的调用

- 当用类的一个对象去初始化（创建）该类的另一个对象时，拷贝构造函数被调用，初始化新对象
- 若函数的参数是类对象，调用函数时，系统自动调用拷贝构造函数，将实参赋对象的值复制一份，作为函数调用的形参
- 当函数的返回值是类对象时，用 return 语句返回对象时，系统自动调用拷贝构造函数，复制一份对象的拷贝，作为返回值
- 缺省拷贝构造函数：
 - ❑ 如果程序员没有为类声明拷贝初始化构造函数，则编译器自己生成一个拷贝构造函数
 - ❑ 这个构造函数执行的功能是：用作为初始值的对象的每个数据成员的值，初始化将要建立的对象的对数据成员
 - ❑ 在某些情况下，缺省拷贝构造函数可能是不正确的，例如存在指针的情况

析构函数

- 在C语言编程时，是不是经常有动态申请内存后、忘记释放，打开文件、忘记关闭的情况？尤其是第三方使用时？如何避免？
- 完成对象被删除前的一些清理工作
 - ❑ 关闭对象在生存期间打开的文件
 - ❑ 释放动态申请的资源：内存、...
- 在对象的生存期结束的时刻系统自动调用它，然后系统才能销毁此对象所占用的空间。
- 如果程序中未声明析构函数，编译器将自动产生一个默认的析构函数。
- 形式：~类名（）；

构造函数和析构函数举例

```
#include <iostream>
using namespace std;
#define BUF_SIZE 512
class DataBuf
{
private:
    //FILE * fPtr;
    char * Buf;
public:
    DataBuf();
    DataBuf(DataBuf & DF);
    ~ DataBuf();

    //... 其它函数
    void ReadBuf();
    ...
};
```

```
DataBuf :: DataBuf( )
{
    // 构造函数
    Buf = new char [BUF_SIZE];
}
DataBuf :: DataBuf(DataBuf & DF)
{
    // 拷贝构造函数
    Buf = new char [BUF_SIZE];
}
DataFile::~~DataFile()
{
    // 析构函数
    delete [] Buf;
}
```


类的组合

- 类中的成员是另一个类的对象。
- 用于在已有的抽象的基础上实现更复杂的抽象。

组合类例子

```
class Point
{
    private:
        float x,y;    //点的坐标
    public:
        //构造函数
        Point();
        Point(float h,float v);

        float GetX(void);
        float GetY(void);
        void Draw(void); //画点
};

//...函数的实现
```

```
class Line
{
    private:
        Point p1,p2; //两个端点
    public:
        //构造函数
        Line(Point a,Point b);

        Void Draw(void); //画线段
};

//...函数的实现略
```

组合类的构造函数

➤ 原则：不仅要负责对本类中的基本类型成员数据赋初值，也要对对象成员初始化。

➤ 声明形式：

类名::类名(参数表)

: 成员(初始化参数), 成员(初始化参数),

{

本类初始化

}

需要使用参数进行初始化的成员
(包括对象成员) 在列表中列出;
没有列出的对象成员, 则调用其
默认的构造函数

组合类构造函数

```
class Point
{
    private:
        float x,y;    //点的坐标
    public:
        //构造函数
        Point();
        Point(Point& p);
        Point(float h,float v);

        float GetX(void);
        float GetY(void);
        void Draw(void); //画点
};
```

//...函数的实现

```
class Line
{
    private:
        Point p1,p2; //两个端点
        int width; //线宽
    public:
        //构造函数
        Line();
        Line(int x1, int y1);
        Line(int x1, int y1,
              int x2, int y2
              int w);
        Line(Point a,Point b,int w);
};
```

组合类构造函数

```
Line::Line()
```

```
{  
    ...  
}
```

对 p1、p2 调用默认构造函数 Point()

```
Line::Line(int x1, int y1) : p1(x1,y1)
```

```
{  
    ...  
}
```

对 p1 调用构造函数 Point(h, v),
对 p2 调用默认构造函数 Point()

```
Line::Line(int x1, int y1, int x2, int y2, int w)
```

```
    :p1(x1,y1), p2(x2,y2), width(w)
```

```
{  
    ...  
}
```

用 w 初始化成员 width

对 p1、p2 调用构造函数 Point(h, v)

```
Line::Line(Point a, Point b, int w):p1(a), p2(b), width(w)
```

```
{  
    ...  
}
```

对 p1、p2 调用构造函数 Point(p)

组合类的构造函数调用

- 构造函数调用顺序：先调用内嵌对象的构造函数（按内嵌时的声明顺序，先声明者先构造）。然后执行本类的构造函数。（析构函数的调用顺序相反）
- 若调用默认构造函数（即无形参的），则内嵌对象的初始化也将调用相应的默认构造函数
 - 这时，若成员对象类没有定义默认构造函数，就会出错

总结：构造函数与析构函数

➤ 构造函数

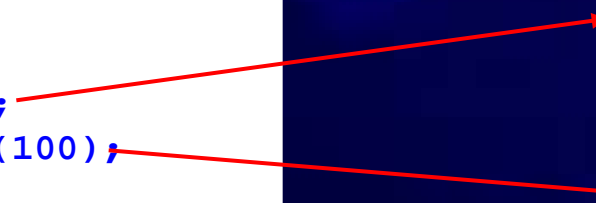
- ❑ 在类的对象被创建时，将会自动调用类的构造函数对类的对象进行初始化
- ❑ 构造函数没有返回值
- ❑ 构造函数可以重载
- ❑ 构造函数可以包含参数

```
main()
{
    ...
    CCounter counter_1;
    CCounter counter_2(100);
    ...
}
```

```
class CCounter
{
private:
    int iCounterValue;
    ...
public:
    CCounter();
    CCounter(int iStart);
    ...
};

CCounter::CCounter()
{
    iCounterValue = 0;
}

CCounter::CCounter(int iStart)
{
    iCounterValue = iStart;
}
```



总结：构造函数与析构函数

➤ 构造函数中数据成员的初始化

❑ 情况1：成员为基本数据类型时

❑ 情况2：成员为对象时

➤ 使用初始化列表

```
class CDbCounter
{
private:
    int iMaxCounterValue;
    CCounter counter_1;
    CCourter counter_2;
    ...
public:
    CDbCounter();
    CDbCounter(int iStart_1,
                int iStart_2,
                int iMaxCntVal);
    ...
};
```

```
CDblCounter::CDblCounter()
{
    iMaxCounterValue = 1000;
    ...
}

CDblCounter::CDblCounter(int iStart_1,
                          int iStart_2,int iMaxCntVal)
: counter_1(iStart_1),
  counter_2(iStart_2),
  iMaxCounterValue(iMaxCntVal)
{
    ...
    // iMaxCounterValue = iMaxCntVal;
    ...
}
```


构造函数与析构函数

➤ 构造函数什么时候被调用？

❑ 全局对象变量

在开始执行 main() 函数代码前，
全局对象变量的构造函数被调用

❑ 局部（自动）对象变量

在定义局部对象变量时，对象的
构造函数被调用

❑ 动态创建的对象

对象申请动态空间成功后被调用

❑ 对象的成员

✉ 在容器对象被创建时，其构
造函数被调用时被执行

对象数组（包括动态的）通常不
能使用带有参数的构造函数

```
class CCounter
{
    ...
};


CCounter g_counter;

main()
{
    CCounter counter_1(200);
    CCounter counter_2;
    CCounter Counter_List[16];

    CCounter * pCounter;
    pCounter = new Ccounter(100);

    Ccounter * pCounterArray;
    pCounterArray = new Ccounter[16];

    ...
}
```



构造函数与析构函数

➤ 析构函数

□ 在对象被销毁时被调用，用于销毁前的准备工作

✉ 如释放对象占用的资源（内存空间、设备...）

```
#define MAX_STR_LEN    1024
class CMyString
{
private:
    int    iStrLen;
    char * pStrBuf;
    ...
public:
    CMyString();
    CMyString(char * pStr);
    ~CMyString();

    CopyString(char * pSource);
};
```

```
CMyString::CMyString():iStrLen(0)
{
    pStrLen = new char[MAX_STR_LEN];
    pStrLen[0] = 0;
}
CMyString::CMyString(char * pStr)
{
    pStrLen = new char[MAX_STR_LEN];
    CopyString(pStr);
}
CMyString::CopyString(char * pSource)
{
    ::strcpy(pStrBuf, pSource);
    iStrLen = ::strlen(pSource);
}
CMyString::~~CMyString()
{
    delete [] pStrLen;
}
```

构造函数与析构函数

➤ 析构函数什么时候被调用？

❑ 全局对象变量

在main() 函数退出前，全局对象变量的析构函数被调用

❑ 局部（自动）对象变量

在局部对象变量的生命期结束之前，对象的析构函数被调用

❑ 动态创建的对象

在释放对象的动态空间之前，其析构函数被调用

❑ 成员对象

☒ 容器对象被销毁时，成员对象的析构函数后被调用

```
class CCounter
{
    ...
};

CCounter g_counter;

main()
{
    ...
}
```

g_counter 的析构
函数被调用

构造函数与析构函数

➤ 析构函数什么时候被调用？

❑ 全局对象变量

在main() 函数退出前，全局对象变量的析构函数被调用

❑ 局部（自动）对象变量

在局部对象变量的生命期结束之前，对象的析构函数被调用

❑ 动态创建的对象

在释放对象的动态空间之前，其析构函数被调用

❑ 成员对象

✉ 容器对象被销毁时，成员对象的析构函数后被调用

析构函数被调用

```
void func()
{
    CCounter counter;
    ...
}

main()
{
    if (...)
    {
        CCounter counter_1(200);
        ...
    }

    CCounter * pCounter;
    pCounter = new Ccounter(100);
    Ccounter * pCounterArray;
    pCounterArray = new Ccounter[16];

    delete pCounter;
    delete [] pCounterArray;
}
```

Counter 的析构函数被调用

Counter_1的析构函数被调用

前向引用声明

- 类应该先声明，后使用
- 如果需要在某个类的声明之前，引用该类，则应进行前向引用声明。
- 前向引用声明只为程序引入一个标识符，但具体声明在其它地方。

前向引用声明举例

```
class B;    //前向引用声明

class A
{   public:
    void f(B b);
};

class B
{   public:
    void g(A a);
};
```

前向引用声明注意事项

- 使用前向引用声明虽然可以解决一些问题，但它并不是万能的。需要注意的是，尽管使用了前向引用声明，但是在提供一个完整的类声明之前，不能定义该类的对象，也不能在内联成员函数中使用该类的对象。请看下面的程序段：

```
class Fred;           //前向引用声明
class Barney
{
    Fred x;           //错误：类Fred的声明尚不完善
};
class Fred
{
    Barney y;
};
```

前向引用声明注意事项

```
class Fred;    //前向引用声明
```

```
class Barney
{
    public:
        void method()
        {
            x->youDo();    //错误：Fred类的对象在定义之前被使用
        };
    private:
        Fred* x;    //正确，经过前向引用声明，可以声明Fred类的对象指针
};
```

```
class Fred
{
    public:
        void youDo();
    private:
        Barney* y;
};
```

使用前向引用声明时，只能使用被声明的符号，而不能涉及该类的任何实现细节，因为该类还没有定义