

# C++面向对象程序设计

---

## 第三讲：封装

# 抽象：获取实体的属性和行为

对具体实体（对象）进行概括，抽取一类对象的公共属性和行为。

- 注意本质，围绕重点，抓住共性 ——》 属性和行为
- 属性：数据抽象，某类对象的属性或状态（对象相互区别的依据）
  - 标识属性
  - 确定属性的值域
- 行为：方法抽象，某类对象的行为特征或具有的功能
  - 标识行为
  - 行为的处理对象（有哪些？来源是什么？）
  - 行为的处理结果（有什么？谁接受/收处理结果？）

# 抽象的实例：时间

- 时间的构成要素（应用范围）
  - 时、分、秒、百分秒、毫秒、微秒、纳秒
- 时间的使用
  - 啥时候了？几点了？
    - 6点（上午还是下午？）
  - 30分钟后出发、2小时后去哪儿等等
    - 到底是啥时候呢？
  - .....

# 抽象的实例：时间属性

- 属性
  - 小时
    - 值域：0-12 / 0-23 ?
  - 分钟
    - 值域：0-59
  - 秒
    - 值域：0-59

# 抽象的实例：时间功能

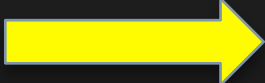
- 设置当前时间
  - 设定时间对象的值(时间)，谁如何给定？
- 显示
  - 将时间对象的值打印在屏幕上
- 时/分/秒
  - 获取时间对象的分量值（时、分、秒）
- 计算
  - 加上时/分/秒后的时间值，是改变对象本身的值，还是生成新的对象
- .....
  - 还能想到什么？

# 封装时间对象——类

整合属性与行为并进行程序语言表示—〉对象泛化为类  
对象泛称—〉类名；属性—〉成员变量；行为—〉成员函数

- 时间
    - 属性
      - 时 (0-23)
      - 分 (0-59)
      - 秒 (0-59)
    - 行为
      - 设定时间 (时、分、秒)
      - 显示时间 ()
      - 增加分量 (时/分/秒)
  - Time
    - 成员变量
      - char hour; 0-23
      - char minute; 0-59
      - char second; 0-59
    - 成员函数
      - void SetTime(char, char ,char)
      - void ShowTime ()
      - void Add(char, char)
- 

# 类定义（封装）（续）

- Time
    - 成员变量
      - char hour; 0-23
      - char minute; 0-59
      - char second; 0-59
    - 成员函数
      - void SetTime(char, char ,char)
      - void ShowTime ()
      - void Add(char, char)
  - CTime
    - 成员变量
      - int hour; 0-23
      - int minute; 0-59
      - int second; 0-59
    - 成员函数
      - void SetTime(int, int, int)
      - void ShowTime ()
      - void add(int, char)
      - void addHour(int)
      - void addSec(int)
      - void addMin(int)
- 

# 封装：定义时间类(头文件)

文件名：mytime

```
#ifndef TIME_H_
#define TIME_H_
namespace nsname {
class CTime {
public:
    int hour;
    int minute;
    int second;
    void SetTime(int h, int m, int s);
    void ShowTime();
    void addHour(int);
    void addMin(int);
    void addSec(int);
};
} //namespace nsname
#endif /* TIME_H_ */
```



# 类的实现——时间类

```
#include "mytime"
#include <iostream>
#include <iomanip>
using namespace std;
namespace nsname {

void CTime::SetTime(int h, int m, int s)
{
    hour = h;
    minute = m;
    second = s;
}

void CTime::ShowTime()
{
    cout << hour << ":" << minute << second;
}

}
```

文件名: mytime.cpp

```
void CTime::addHour(int h)
{
    hour += h;
}

void CTime::addMin(int m)
{
    minute += m;
}

void CTime::addSec(int s)
{
    second += s;
}

}
```

# 类的使用——时间对象

```
#include "mytime"
```

```
using namespace nsname;
```

```
int main() {
```

```
    CTime c1;
```

```
    c1.SetTime(14,20,30);
```

```
    c1.ShowTime();
```

```
    c1.SetTime(8,2,30);
```

```
    c1.ShowTime();
```

```
    c1.SetTime(8,2,3);
```

```
    c1.ShowTime();
```

```
    c1.addHour(5);
```

```
    c1.ShowTime();
```

```
    return 0;
```

```
}
```

# 类定义（封装）

## 类的属性和行为的访问限制

- 限制数据成员和函数成员的访问权限
  - 公有 public
    - 完全公开的属性和行为
  - 私有 private
    - 个体专属的属性和行为
  - 保护 protected
    - 家族私有的属性和行为

# 类的定义

- 类是一种用户自定义的数据类型
- 声明形式
  - Class 类名称
  - {
    - public:
      - 公有成员
    - private:
      - 私有成员
    - protected:
      - 保护成员
  - } ;

注意给定访问限制的方式:

从限制方式开始的后续成员,  
直到变更访问限制

默认的成员访问限制是“私有成员”

## 类的定义（续）

- 成员访问限制符号对其后续成员均有效，直到遇到下一个成员访问限制符号；
- 紧跟在类名称的后面的成员，没有访问限制符号注明的话，均为私有成员；
- 从程序的角度而言
  - 随时随地可以访问的是类的公有成员；
  - 类的私有成员则只能在类的成员函数中被访问；
  - 类的保护成员可以被类及其派生类的成员函数中被访问。

# 成员私有化

- 限制非类成员函数对成员变量的访问
- 限制对象的属性和行为被其他对象访问和使用

```
class CTime
{
private:
    int hour, minute, second;
public:
    int getHour(); //取值函数
    void setHour(int h); //设值函数
    int getMinute(); //取值函数
    void setMinute(int m); //设值函数
    int getSecond(); //取值函数
    void setSecond(int s); //设值函数
    void SetTime(int h, int m, int s);
    void ShowTime();
    void addHour(int);
    void addMin(int);
    void addSec(int);
};
```

```
int CTime::getHour()
{
    return hour;
}
```

```
void CTime::SetHour(int h)
{
    if((h >= 0) && (h < 24)) hour = h;
}
```

```
void CTime::addSec(int s)
{
    second += s;
    addMin(second / 60);
    second %= 60;
}
```

```
void CTime::addMin(int m)
{
    minute += m;
    addHour(minute / 60);
    minute %= 60;
}
```

```
void CTime::addHour(int h)
{
    hour += h;
    hour %= 24;
}
```

# 示例

```
int main()
{
    CTime c1;
    c1.SetTime(24,20,32); //
    c1.ShowTime();
    cout << endl;

    c1.SetTime(23,20,32); //
    c1.addSec(180); //
    c1.ShowTime();
    cout << endl;

    c1.addMin(180); //
    c1.ShowTime();
    cout << endl;
    return 0;
}
```

设值函数: `if((h >= 0) && (h < 24)) hour = h;`

设值函数: `if((h >= 0) && (h < 24)) hour = h;`  
180秒后的时间值会是多少?

180分后的时间值会是多少?

# 课堂作业：

- 定义一个日期类
  - 能想到什么，就做什么！！



# 构造对象

- 思考问题
  - 变量在使用前必须有值
  - 定义类时，并不能其成员变量赋值
  - 如何才能定义对象时，就使对象的属性有效呢？
- 构造函数
  - 作用：确定对象的初始形态
  - 如何定义构造函数
  - 何时如何调用构造函数
    - 定义对象时自动调用的类的成员函数
      - 隐式调用
    - 动态构造对象时调用的成员函数
      - 显式调用

# 构造函数的作用

- **确定对象的初始形态**
  - 属性值有效
    - 对成员变量进行赋值，使对象的属性有效/有意义
  - 分配有效的空间
    - 申请内存空间
  - 确定对象的初始状态
    - 空闲、繁忙
- .....

# 声明和定义构造函数

- 定义时间类的构造函数
- `class CTime {`
  - `.....`
  - `CTime();` //类的构造函数（默认构造函数，没有任何形参！）
    - 函数名与类名相同，但没有返回值的类型
    - 事实上，不声明和定义构造函数（即构造函数缺失时），编译器也会自动生成一个函数体为空的默认构造函数
- `}`
- `CTime::CTime()`
  - `{`
    - `hour = 0;`
    - `minute = 0;`
    - `second = 0;`
  - `}`

# 使用构造函数

- 定义对象时，隐式调用默认构造函数
  - `CTime c1;`
- 动态构造对象时，显式调用构造函数
  - `CTime * pc1 = new CTime();`

# 问题

- 如何让程序员在使用时间对象时，能灵活地赋以合理的时间值
  - 想一下
    - `int x = 5;`
    - `int x = 20;`
    - `double x = 0.2;`
    - `double x = 10;`
    - `int x[] = {31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};`
    - `struct person zhangsan={"zhangsan", 'F', "1995-10-1"};`//假设在结构体定义

程序员在定义变量时可以按需赋值

# 丰富构造函数

- 让程序员自主赋值给对象
  - 存在问题：该调用哪个函数来构造对象，如何确定函数的参数
  - 解决方法：共用函数名称（标识符）
  - 程序员：使用不同的构造函数来构造对象
- 函数重载
  - 函数名相同
  - 形参列表不同
    - 数量不同
    - 顺序不同
    - 类型不同
    - 数量、顺序、类型都不同

# 重载构造函数

- `CTime(int h, int m, int s)`
- `{`
  - `hour = h; minute = m; second = s;`
- `}`
  
- `CTime(const string &atime)`
- `{`
  - `//请完成拆分字符串，分离出时分秒并赋给hour, minute和second`
- `}`
  
- `#include <ctime> //包含C的时间类型函数库time.h`
- `CTime(time_t atime) //time_t 时间类型`
- `{`
  - `//请实现该函数体`
- `}`

# 示例

- `CTime c1("10:23:32");`
- `CTime c2(10, 23, 32);`
- `CTime c3(time(NULL));`
  
- `CTime *p1 = new CTime("10:23:32");`
- `CTime *p2 = new CTime(10, 23, 32);`
- `CTime *p3 = new CTime(time(NULL));`



# 带默认参数的构造函数

- 带有默认参数的函数
  - 形参有默认值（实参可以缺失）
  - 实参匹配规则：从左向右
  - 默认参数：从右向左
    - 所有指定默认值的参数都必须出现在不指定默认值的参数的右边
    - 函数调用时，若某个参数省略，则其后的参数皆应省略而取默认值
    - 函数在调用前必须有完整的原型声明或定义
- 带有默认函数的构造函数
  - `CTime(time_t now=time(NULL));`
  - 参数全具有默认值时，需注意与默认构造函数的冲突！！

# 对象的拷贝（复制/克隆）

- 向函数传递对象
  - 传对象（传值）、传对象指针、传对象的[常]引用
- 拷贝构造函数
  - 函数定义
  - 调用拷贝构造函数
    - 对象定义
    - 向函数传递传递时（形参是对象）
      - 注意不是对象引用，也不是对象指针
    - 函数返回对象时
      - 不是对象引用，也不是对象指针
- 浅拷贝和深拷贝的问题

# 析构函数

- 与构造函数对应
- 处理对象析构时的空间释放等操作
- 原型
  - `~类名(void)`
- 调用
  - 对象释放时，出作用域或执行delete
    - `CTime x;`
    - `CTime * px = new CTime();`
    - `delete px;`

# 使用内联函数

- 声明内联函数
  - inline 标识符
- 内联函数与宏定义的差异
  - #define 符号替换
  - inline 代码替换

# 静态成员

- 创建对象之前设置待创建对象的共有特性
    - 如时区、12小时制/24小时制等。
  - 静态成员变量（必须在类定义之外，使用之前给类的静态成员赋值）
    - `static 类型名称 成员变量名称; // static std::string zone;`
  - 静态成员函数
    - `static 返回值类型 成员函数名称(形参列表); //static void showzone(void);`
  - 类的静态成员遵循类成员的访问限制特性
  - 可以在没有创建对象时，通过类名来访问类的静态成员
  - 类的普通成员函数可以访问类的静态成员
  - 定义类的静态成员函数时，可以访问类的静态成员，但不可以访问类的非静态成员
    - 因为该类的对象并不一定已被创建
  - 类的静态成员与函数的静态变量有何区别？
- 类的非静态成员只能通过对象来访问

# 运算符重载

- 为对象提供合适的运算符
- 时间运算
  - 若干秒/分/时后的时间值
  - 两时间之间的差
- 运算符是特殊的函数
  - `operator * // *` 是操作符 `+`、`-`、`*`、`/`、`++`、`-` 等
  - 不能改变算符操作数的数量
  - 符合算符的基本含义

# 为时间对象添加运算符

- 确定合适的运算符及其含义
  - +、-、/
- 依据算符要求的操作数的个数确定操作数
  - 至少有一个是时间对象
- 确定合适的返回值（类型）
  - 新的时间对象，数值（整型、实型、.....）
- 声明函数原型
- 定义函数体

# 为时间对象添加运算符 +

- 加法运算 operator +
  - n秒后的时间
- 运算的数据
  - 时间对象，增加的秒数
  - 参与运算的时间对象不改变
- 返回值
  - 运算后的时间对象
- 函数原型
  - **CTime operator + (const CTime& t, int s);**

```
CTime operator+(const CTime &t, int s)
{
    CTime r(t);
    r.addSec(s);
    return r;
}
```



# 为时间对象添加运算符 -

- 减法运算 operator -
  - 两时间对象相差的秒数
- 运算的数据
  - 时间对象1, 时间对象2
  - 参与运算的时间对象不改变
- 返回值
  - 两时间对象相差的秒数
- 函数原型
  - `int operator -(const CTime& x, const CTime& y);`

```
int operator -(const CTime& x, const CTime& y)
{
    //计算
    //返回计算结果
}
```

# 为时间对象添加运算符 -=

- 加法运算 operator -
  - 减去n秒后的时间
- 运算的数据
  - 时间对象，减少的秒数
  - 参与运算的时间对象的值发生变化
- 返回值
  - 参与运算的时间对象
- 函数原型
  - `CTime& operator -=(CTime& t, int s);`

```
CTime& operator -=(CTime &t, int s)
{
    //计算时间对象t的值
    //返回t
}
```

//传引用，则实际参与运算的时间对象  
//就是实参(时间对象)

# 友元

- 友元函数
  - 普通函数，而非类的成员函数
  - 可以访问类的非公有成员，破坏了对象“信息隐藏”的特性
  - 使用友元函数实现操作符重载
  - `friend 函数原型; //在类定义中声明`
- 友元类
  - 一个对象是另一个对象的朋友
  - `friend class oneclass; //在类定义中声明`
- 伴随命名空间使用的问题
  - 函数体定义时不能在函数名的前面添加命名空间的名称
  - 需要在命名空间中定义函数体

# 使用友元函数重载操作符

- 在类定义中声明（和定义函数体）
- 在类实现文件中定义函数体
- 与普通函数的差异在于是否能够访问类的私有成员

# 以成员函数的方式重载操作符

- 直接将重载的操作符函数作为类的成员函数
- 参与操作符运算的第一个数据为类实例（对象）
- 只能以成员函数进行重载的操作符
  - =、[]、类型转换函数
- 重载赋值操作符
- 类型转换函数

# 为时间对象添加赋值操作符 =

- 赋值与拷贝构造函数的差异
  - 一个是在定义或构造对象时被调用
  - 一个是在对象赋值时被调用
  - 都可以实现不同类型的数据（或对象）到本类对象的转换
- 只能以成员函数的形式定义
  - `CTime& operator =(const CTime& s);` //同类对象的赋值
  - `CTime& operator =(const std::string& s);` //使用其它对象赋值

# 对象的输入和输出

- 重载输入流操作符 >>
  - 以普通函数的形式重载
    - `istream& operator >>(istream& ins, 类名& obj);` //只能访问对象的公有成员
  - 以友元函数的形式重载
    - `friend istream& operator >>(istream& ins, 类名& obj);` //可以访问对象的所有成员
- 重载输出流操作符 <<
  - 以普通函数的形式重载
    - `ostream& operator <<(ostream& outs, const 类名& obj);` //只能访问对象的公有成员
  - 以友元函数的形式重载
    - `friend ostream& operator <<(ostream& outs, const 类名& obj);` //可以访问对象的所有成员

# 输入时间对象

## 普通函数

```
istream& operator >>(istream& ins, CTime& t)
{
    std::string str;
    ins >> str; //格式要求: "hh:nn:ss"
    t.setTime(str); //调用对象的设值函数为对象的属性（成员变量）赋值
    return ins;
}
```

## 友元函数

```
friend istream& operator >>(istream& ins, CTime& t)
{
    ins >> t.hour >> t.minute >> t.second;
    //直接给对象的属性（成员变量）输入数据，但不会检查数据的有效性
    return ins;
}
```



# 输出时间对象

## 普通函数

```
ostream& operator <<(ostream& outs, const CTime& t)
{
    outs << std::string(t); //调用成员函数（类型转换）后输出
    return outs;
}
```

## 友元函数

```
friend ostream& operator <<(istream& outs, const CTime& t)
{
    outs << t.hour << ":" << t.minute << ":" << t.second;
    //直接使用对象的属性（成员变量）输出
    return outs;
}
```

# 重载下标运算符 []

- 让对象具有类数组的访问形式
- 仅能以成员函数的形式重载
- 按序访问
  - 元素类型名 `operator[](int idx)`; //仅能获取相应属性的值, 只读
  - 元素类型名 `& operator[](int idx)`; //能获取和设置相应属性的值, 读写
- 以名称访问
  - 元素类型名 `operator[](std::string name)`; //仅能获取相应属性的值, 只读
  - 元素类型名 `& operator[](std::string name)`; //能获取和设置相应属性的值, 读写

# 给时间对象属性添加数组形式的访问

```
int operator[](int idx)
{
    int r;
    switch(idx) {
        case 0:
            r = hour;
            break;
        case 1:
            r = minute;
            break;
        case 2:
            r = second;
        }
    return r;
}
```

只能获取属性时、分、秒的值

```
int& operator[](const std::string& name)
{
    int* p;
    if(name == "hour") {
        p = &hour;
    } else if(name == "minute") {
        p = &minute;
    } else {
        p = &second;
    }
    return *p;
}
```

假设t为时间对象，则可：

**t["hour"] = 20; //设置t的时值**

或

**int h = t["hour"]; //取t的秒值**

# ++和--操作符

- 前缀

- 成员函数

- `CTime operator++(); //--`

返回的是对象变更后的值

- 友元函数

- `friend CTime operator++(CTime& t); //--`

- 后缀

- 成员函数

- `CTime operator++(int); //--`

返回的是对象变更前的值

- 友元函数

- `friend CTime operator++(CTime& t, int x); //--`

- 这里的参数int x仅表示是后缀++, 其值不参与运算

# 示例

```
CTime operator++(CTime& t) //友元或普通函数, 前缀++
{
    t.addSec(1);
    return t;
}
```

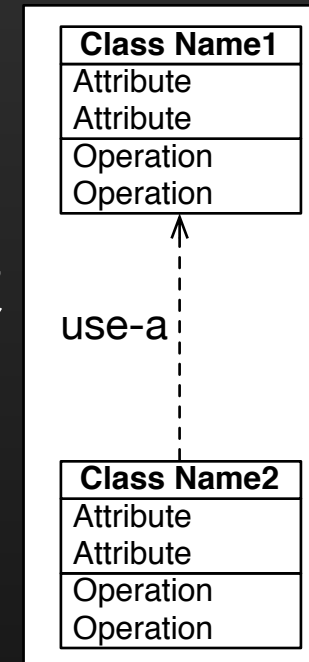
```
CTime operator++(int x) //成员函数, 后缀++
{
    CTime t(*this); //构造一个新的对象, 用于保存对象的原有值
    this->addSec(1); //变更对象的属性值
    return t;      //返回变更前的对象 (前期保存的对象)
}
```

```
CTime operator++(CTime& t, int x) //普通或友元函数, 后缀++
{
    CTime t1(t); //构造一个新的对象, 用于保存对象的原有值
    t.addSec(1); //变更对象的属性值
    return t1;   //返回变更前的对象 (前期保存的对象)
}
```

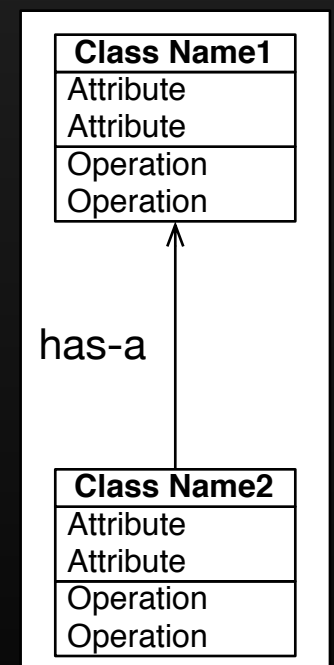
# 类与类的关系

- 对象的成员变量是一个对象
- 依赖、关联、聚合、组合
- 依赖：所谓依赖就是某个对象的功能依赖于另外的某个对象，而被依赖的对象只是作为一种工具在使用，而并不持有对它的引用。
  - 成员函数的参数
  - 成员函数函数体中的局部变量（对象）
  - 成员函数中调用他类的静态方法
- 关联：所谓关联就是某个对象会长期的持有另一个对象的引用，而二者的关联往往也是相互的。关联的两个对象彼此间没有任何强制性的约束，只要二者同意，可以随时解除关系或是进行关联，它们的生命期问题上没有任何约定。被关联的对象还可以再被别的对象关联，所以关联是可以共享的。
  - 朋友间的关系

## 局部使用



## 单向/双向



# 类与类的关系

- 聚合：聚合是强版本的关联。它暗含着一种所属关系以及生命期关系。被聚合的对象还可以再被别的对象关联，所以被聚合对象是可以共享的。虽然是共享的，聚合代表的是一种更亲密的关系。
  - 球队与球员
  - 手机与配件
- 组合：组合是关系当中的最强版本，它直接要求包含对象对被包含对象的拥有以及包含对象与被包含对象生命期的关系。被包含的对象还可以再被别的对象关联，所以被包含对象是可以共享的，然而绝不存在两个包含对象对同一个被包含对象的共享。
  - “有国才有家”
  - 人和生理器官

