**目 录**

[设计篇](#_Toc6120)

[第二章 综合项目初体验](#_Toc31099)

[2.1 项目下载地址及安装操作说明](#_Toc18013)

[2.2 综合项目介绍](#_Toc6036)

[2.2.1 综合实例1---LostCraft游戏项目介绍](#_Toc5704)

[2.2.2 综合实例2---YYY系统项目介绍](#_Toc13655)

[2.2.3 综合实例3---ZZZ系统项目介绍](#_Toc28695)

[2.3 项目角色功能分析](#_Toc31678)

[2.3.1 综合实例1---LostCraft游戏项目角色功能分析](#_Toc23040)

[2.3.2 综合实例2---YYY系统项目角色功能分析](#_Toc23751)

[2.3.3 综合实例3---ZZZ系统项目角色功能分析](#_Toc22590)

[第三章 面向对象思维建立](#_Toc1321)

[3.1 面向对象概念引入](#_Toc24682)

[3.1.1 软件行业现状](#_Toc10448)

[3.1.2 何为面向对象](#_Toc145)

[3.2 类与对象](#_Toc26656)

[3.2.1 综合实例1---LostCraft游戏项目之对象间行为关系](#_Toc8175)

[3.2.2 综合实例1---LostCraft游戏项目之类的抽象](#_Toc15428)

[3.2.3 类与对象的概念与关系](#_Toc11887)

[第四章 面向对象思维建摸](#_Toc2846)

[4.1 UML漫谈之类图与时序图](#_Toc367)

[4.1.1 类图](#_Toc18323)

[4.1.2 时序图](#_Toc20306)

[4.2 项目实现之UML设计](#_Toc26945)

[4.2.1 综合实例1---**LostCraft**游戏项目之UML设计](#_Toc2274)

[4.2.2 综合实例2---YYY系统项目之UML设计](#_Toc7908)

[4.2.3 综合实例3---ZZZ系统项目之UML设计](#_Toc1690)

[实践篇](#_Toc30971)

[第五章 程序设计语言C++初识](#_Toc28992)

[5.1 C++概述](#_Toc30308)

[5.2 第一个C++程序](#_Toc18572)

[5.2.1 VS2010工具介绍](#_Toc27403)

[5.2.2 编写第一个C++程序helloworld](#_Toc16287)

[5.2.3 代码解释](#_Toc14558)

[5.2.4 程序注释](#_Toc9882)

[5.3 C++编译工作原理](#_Toc21931)

[5.4 C++编程逻辑](#_Toc4507)

[5.4.1 程序中简单数据存储](#_Toc11405)

[5.4.2 输入与输出](#_Toc12516)

[5.4.3 控制语句](#_Toc8203)

[5.4.4 一维数组](#_Toc11003)

[5.4.5 函数](#_Toc23086)

[5.4.6 指针与引用](#_Toc1682)

[第六章 类与对象的实现](#_Toc27224)

[6.1 综合实例1---LostCraft游戏](#_Toc16886)

[6.1.1 类的构成](#_Toc11252)

[6.1.2 类的声明与实现](#_Toc28258)

[6.1.3 对象实例化](#_Toc27920)

[6.1.4 对象间的协作实现](#_Toc2447)

[6.2 综合实例2---YYY系统项目](#_Toc27433)

[6.2.1 类的实现](#_Toc24328)

[6.2.2 对象实例化之new关键字](#_Toc16281)

[6.2.3 对象资源回收之delete关键字](#_Toc5840)

[6.2.4 对象间的协作实现](#_Toc24092)

[6.3 综合实例3---ZZZ系统项目](#_Toc14932)

[6.3.1 类的实现](#_Toc1161)

[6.3.2 this指针](#_Toc27073)

[6.3.3 对象实例化及对象间协作实现](#_Toc3394)

[6.4 类与对象概念的强化](#_Toc3383)

[6.4.1 类与对象的概念](#_Toc25122)

[6.4.2 类与对象的关系](#_Toc26739)

# 设计篇

# 第二章 综合项目初体验

## 2.1 项目下载地址及安装操作说明

## 2.2 综合项目介绍

### 2.2.1 综合实例1---LostCraft游戏项目介绍

现在是2222年的某一天，地球忽然遭受来自外太空生物的攻击，世界正面临着巨大的危机，而你，恰恰是被世界人民选中的具有超能力的未来英雄！

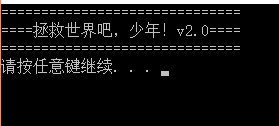
为了拯救地球，打败怪兽，你需要通过不断的战斗来提升自己的等级，并不停的学会新的技能，时间不等人，抓紧时间操练起来吧！

为了世界和平！加油啊，少年！



废话不多说，下面我们就来了解一下整个游戏的规则！

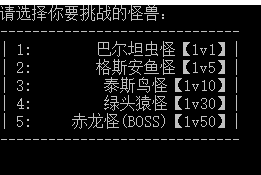
1. 游戏开始后，玩家会看到游戏开始的欢迎界面。



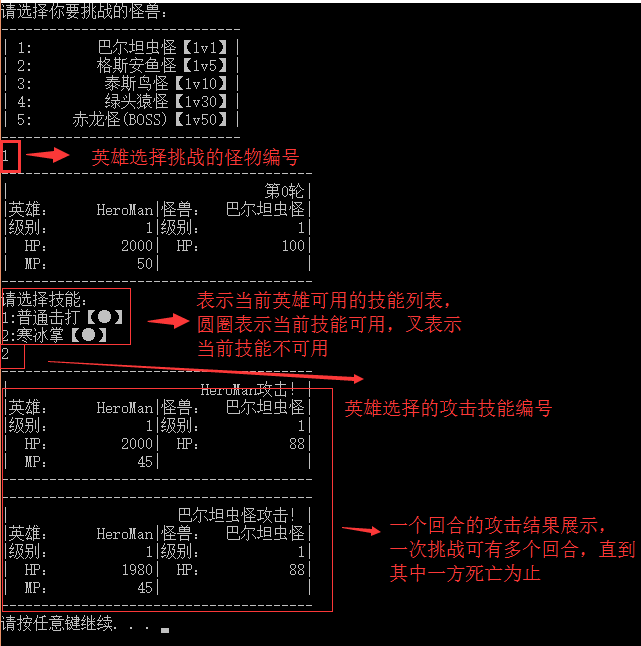
1. 即默认玩家为该游戏英雄，玩家首先通过键盘输入的方式给自己取名，假设玩家给自己取名为HeroMan，当玩家取名成功后，即可进入游戏场景。



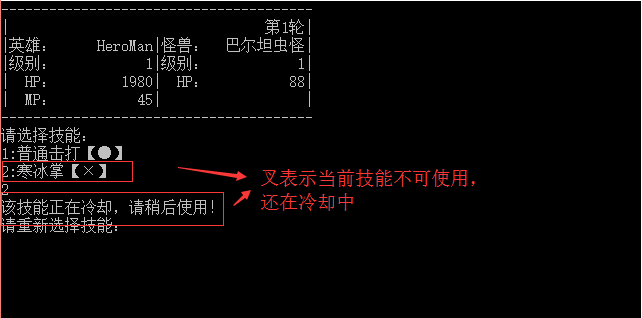
1. 通过游戏场景说明，你会发现地球正在被外太空生物侵袭，而外太空一共有5种怪物，每个怪物的等级分别从低到高（lv1， lv5，lv10， lv30， lv50），其中等级为50的怪物为最终大BOSS，英雄只需要战胜大BOSS，即认为游戏胜利！



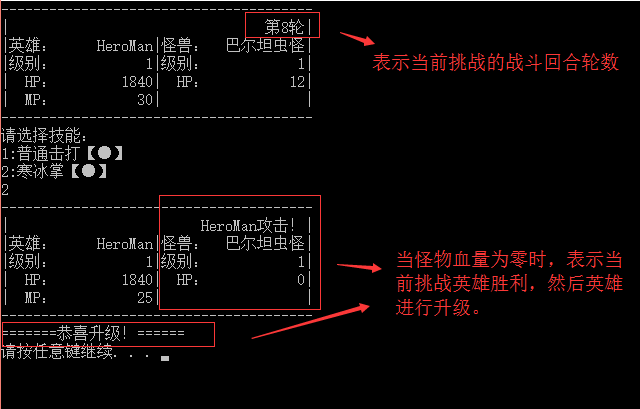
1. 该游戏为挑战类游戏，即每次战斗英雄都可以根据自己的实力从外太空5种怪物中进行选择挑战，按照游戏规则，英雄被创建开始等级默认为1，英雄可通过挑战战斗来提升自己的等级，同时，随着英雄等级的提升或降低，英雄的攻击力也随之提升或降低，且选择怪物后英雄可以看到自己目前所能使用的技能列表，英雄等级不同时，所能使用的技能列表也相对不同，英雄玩家根据自己的战术进行技能选择攻击。



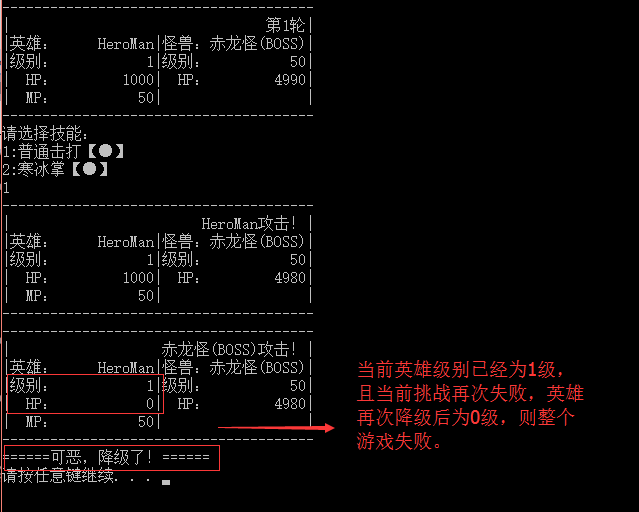
1. 不同的技能有不同的冷却轮数，如果技能未冷却结束，则暂时不可使用该技能。

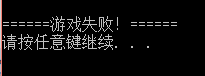


1. 英雄和怪兽在每一局的比赛中都有各自的生命值，该生命值和自身的等级相关，无论是英雄还是怪兽在当局比赛中如果生命值小于等于0，则认为该挑战回合死亡。如果回合挑战胜利，则英雄增加的等级数和怪物的等级数持平，例如英雄当前等级为7级，挑战10级怪兽成功后，则英雄等级更新为17级；反之，如果英雄挑战失败，则等级数降1。

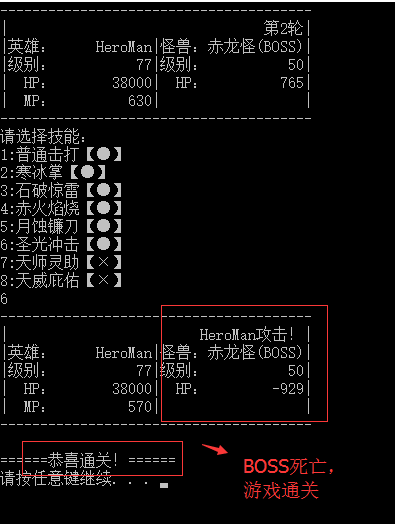


1. 当英雄等级数降为0，则认为英雄死亡，游戏失败！





1. 最终无论英雄等级数为多少，只要英雄成功挑战一次最终极怪兽后，则游戏成功！



### 2.2.2 综合实例2---YYY系统项目介绍

### 2.2.3 综合实例3---ZZZ系统项目介绍

## 2.3 项目角色功能分析

### 2.3.1 综合实例1---LostCraft游戏项目角色功能分析

**英雄角色**

通过游戏规则的描述我们发现，游戏中有且仅有一个英雄，即玩家自己，并且通过观察我们发现英雄这个角色在游戏中具有一定的角色特征，例如：英雄的名字、英雄的等级、攻击力等等；同时我们还发现英雄角色有一定的动作，例如：攻击怪兽、提升等级等等。

下面我们就来具体分析一下英雄这个角色到底有哪些特征以及行为。

**角色特征：**

1. 作为英雄首先要有一个霸气的名字！我们简称为**姓名**；
2. 随着游戏的进行，英雄的级别在不停的变化，我们简称为**等级**；
3. 每一个回合开始，英雄都有自己的生命值，我们简称为**HP**；
4. 每一个回合开始，英雄都有自己的能量值，用来释放特殊技能，我们简称为**MP**；
5. 英雄在不同的等级的时候，所拥有的技能是不一样的，我们简称为**技能列表**。

**角色行为：**

1. 游戏一开始，我们的英雄就应该出场啦，并且英雄出场的时候相关数据就应该被生成出来，例如英雄的姓名、当前等级等等，我们简称为**初始化英雄；**
2. 游戏开始，英雄选择挑战的怪物，然后进入战斗模式，这时英雄需要选择可用的攻击技能，我们简称为**选择技能；**
3. 英雄对怪物有攻击行为，我们简称为**英雄攻击**；
4. 每一个攻击回合结束后，我们都需要判断一下当前英雄血量是否为零，我们简称为**判断英雄是否存活；**
5. 当一次挑战结束时，如果英雄胜利，则英雄的等级也会随之提升，我们简称为**英雄等级提升；**
6. 当一次挑战结束时，如果英雄失败，则英雄的等级也会随之下降，我们简称为**英雄等级降低；**
7. 每一回合的战斗结束时，我们都需要显示一下当前英雄的状态，我们简称为**结算英雄当前回合状态。**

**怪物角色**

随话说，知己知彼，百战不殆，上述我们已经对英雄角色进行了分析，下面我们再来看一下游戏中的怪兽角色，在这里我们要能够理解一点，即虽然游戏中有5种怪兽，可是因为他们所拥有的角色特征和行为都是一样的，都属于怪兽角色，因此，我们可以直接针对怪物来进行分析，看看怪物角色分别有哪些角色特征和角色行为。

**角色特征：**

1. 作为反派角色，无论是哪种怪兽，也应该有一个霸气的名字！我们简称为**怪物名称**；
2. 在游戏过程中，不同的怪兽有不同的级别，我们简称为**怪物等级**；
3. 每一个回合开始，怪兽都有自己的生命值，我们简称为**怪物HP**；
4. 怪兽不同于英雄，怪兽的等级是固定的，且同一个等级的怪兽他的攻击力也是固定的，我们简称为**怪物攻击力**；

**角色行为：**

1. 当英雄选好需要挑战的怪兽后，怪兽就应该被创建啦，我们简称为**初始化所选怪物；**
2. 战斗回合开始，当英雄攻击后，怪兽会反击，即对英雄有攻击行为，我们简称为**怪兽攻击；**
3. 每一个攻击回合结束后，我们都需要判断一下当前怪兽血量是否为零，我们简称为**判断怪兽是否存活。**

**场景角色**

在整个游戏中，最显而易见的角色一定是英雄和怪兽，因此通常这个时候我们往往就会忽略了另外一个很重要的角色，即场景角色。毕竟每一次的战斗都需要一个单独的场景来实现呀！和上述方法一样，我们来看一下场景角色里面有哪些角色特征和角色行为吧。

**角色特征：**

1. 每次战斗的场景里都应该有我们的大英雄存在哦，我们简称为**英雄对象；**
2. 这个时候怎么能缺少我们的大反派呢？所以每次战斗场景中，还有挑战的怪兽存在，我们简称为**怪兽对象；**
3. 每一场的战斗中，我们每一个回合的战斗都应该被记录，并累计统计出来，用于观察英雄的技能是否已经完成冷却继续使用，我们简称为**当前回合**；
4. 最后，我们还应该有一个标识符，用来记录当前游戏是否结束，即是英雄打败BOSS或者英雄挂掉的状态，我们简称为**游戏当前状态**。

**角色行为：**

1. 当游戏开始后，我们应该把给我们英雄战斗的地方准备好，我们简称为**初始化场景；**
2. 在战斗场景中，英雄要选择挑战的怪物，我们简称为**选择怪物；**
3. 英雄想要拯救世界，仅仅光靠一次战斗是不可能的，所以我们要不停的挑战怪物，直到最终打败Boss为止，我们简称为**循环挑战怪物；**
4. 在每一轮挑战开始后，英雄和怪兽之间会进行互相攻击，而每次攻击结束后，我们都应该显示当前英雄和怪物战斗结束时的状态，并且根据场景不同，我们需要显示的状态也不同，我们简称为**显示信息；**

### 2.3.2 综合实例2---YYY系统项目角色功能分析

### 2.3.3 综合实例3---ZZZ系统项目角色功能分析

# 面向对象思维建立

## 3.1 面向对象概念引入

### 3.1.1 软件行业现状

软件对今天的我们来说，不再是个陌生的名词，从第一代计算机的出现到现在的神八天宫交会，从大型的Dota到小型的手游，可以说无处不在，深入到我们生活的每一个角落。在计算机刚刚出世的年代，受限于计算机的计算效能和存储空间，软件的规模相对较小，业务逻辑也不算复杂，代码量相对较小。如今，随着计算机的飞速发展，硬件的更新更是如日中天，处理器的处理速度可以说是飞跃式递增，存储空间大到海量，与之对应的软件需求也是几何级数的增长，**程序代码量庞大且逻辑复杂度高。**

同时随着科技的进步，往日的软件规模远远不能满足当前发展需求，目前软件开发的现状还存在着很多问题，比如**软件规模较大、高级程序语言也越来越多。**

而正是由于软件规模的扩大，导致我们的开发方式也发生了转变，主要体现在两个方面：

1. 个人无法单独完成一项工程，需要团队共同完成；
2. 要按照一定的规范做事，不能随心所欲。

因此各类的高级程序语言也随之应运而生，既然是高级程序语言，那么我们首先应解决下面两个问题。

1. 什么是高级语言？

高级编程语言就是高于机器语言以上的开发语言，刚开始的语言是汇编，后来慢慢发展出了C语言，然后出现更多高级语言，现在的高级语言有多，例如C++、Java，Python等。

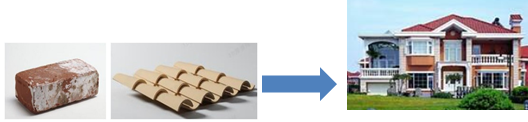
1. “高级”体现在什么地方？

这里以一个例子来说明。

如果把混泥土一类基本建筑材料看作汇编的指令，例如水泥、土堆、石子等。



那么高级语言就是将这些基本材料组合成一个人类能够理解并使用的成套的物品，如砖头、瓦片等，并且最终使用这些砖头瓦片建成一套经过独立设计的房子。



所以语言越低级，描写程序就越复杂，指令越难懂，越靠近机器；语言越高级就越靠近人的表达与理解。

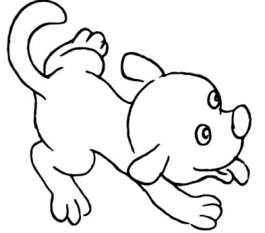
### 3.1.2 何为面向对象

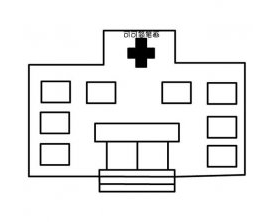
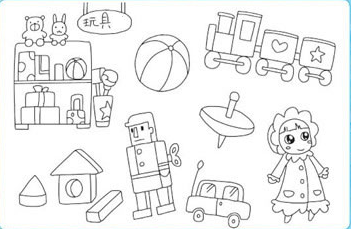
面对这样一个软件现状，我们该如何做呢？我们以一个生活场景入手来看。

现在生活中很流行养宠物，要养好一只宠物，我们该怎么做？

1. 我们要去宠物店买一只自己心爱的宠物
2. 我们要带宠物去医院体检，打预防针
3. 我们要去超市给宠物买食物、玩具、房子
4. 我们要给宠物喂养食物
5. 我们要与它增加感情，陪它一起玩耍
6. 我们要带它美容店美容，带它去与它们同类相亲
7. ……

要养好一只宠物很复杂，并不是我们一个人就能够完成的，**我们需要与不同的人，不同的事物进行协助共同完成。**

那么要想完成这样一件事情，我们先来分析一下：

1. 识别出有哪些对象

* 对象有人、宠物、宠物店，医院、超市……

1. 这些对象独自做什么事情或有什么行为

* 人买宠物，人买玩具，人喂宠物食物，宠物吃食物，医院看病，超市卖东西……

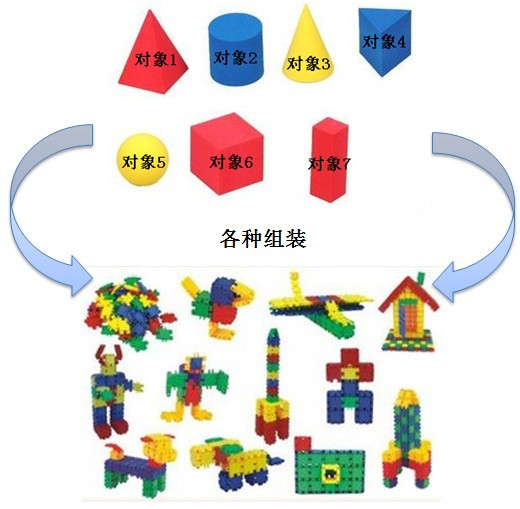
1. 对象间如何协作，如何把每个对象各自完成的事情组装成一件完整的事情。

* 人去宠物店买宠物，人带宠物去医院看病，人喂宠物吃食物……

通过上述分析我们发现，无论我们所涉及的场景有多复杂或庞大，我们的切入点都是场景或案例中所出现的每一个**对象个体**，然后再去分析每一个对象个体之间的关联关系，最后我们只需要借助编程工具，实现这种关联关系即可，我们整个这样的思维过程称之为**面向对象的思维方式**。

用面向对象的思维方式进行分析，那对应的开发方式也应该是面向对象的，因此，我们不得不提出一种一个概念：**OOP(Object-Oriented Programming)：面向对象程序设计。**

它是通过为数据和代码建立分块内存区域的方法来对程序进行模块化设计的一种程序设计方法，这些模块可以被用作样板，在需要时建立其副本。它使得大型、复杂软件项目可以被程序员所管理。



通过上图直观地展示，结合之前我们所说的软件开发现状，思考一下为什么OOP开发方法能够满足当前的软件开发现状？

1. 模块化：把大问题分解成更小的子问题，这样就可以单独解决每一个子问题，当所有的子问题被解决后，与之产生的大问题也随之被解决。
2. 抽象化：定义了能够体现在软件解决方案问题域的术语，这样各个模块设计思路都便于开发者的理解。
3. 封装(细节隐藏)：

* “外面世界” 看不到，也不能依赖它。
* 对象与“外部世界”之间，对象是一个“防火墙”，提高安全性能。
* 隐藏的“数据”及“方法”可以被改变，但不会影响“外部世界”。

1. 可组合：程序员可以根据需要，自由的组合模块开发出新系统。
2. 继承：便于小而简单到大而复杂的模块增量开发。
3. 软件持续使用时间更长：软件的维护与改变只在少数几个模块中发生，不影响软件体系结构。

## 3.2 类与对象

到目前为止，我们学习了面向对象的思维方式：

1. 分析需求，识别出有哪些对象
2. 它们分别做什么事情或有什么行为
3. 每个对象如何协作，各自完成的事情如何组装完成一件完整的事情
4. 考虑程序设计语言如何实现

根据上述的方法，将**面向对象思维分为两步**：**一是分析需求，识别出有哪些对象，它们分别做什么事情****(****who && what)；二是如何利用一个对象协作完成一件完整的事情****(how)。**

可是具体我们应该怎么做呢？

接下来我们以案例一LostCraft游戏项目来进行说明。

### 3.2.1 综合实例1---LostCraft游戏项目之对象间行为关系

回顾之前我们所分析的综合案例一（LostCraft游戏），通过整个游戏运行的流程以及规则的讲解，让我们来看一下整个游戏过程中各个对象之间有哪些行为关系。

1. 游戏开始后，首先进入**游戏初始场景**，玩家首先通过键盘输入的方式给自己取名，假设玩家给自己取名为HeroMan，当玩家取名成功后，即创建出一个**英雄玩家**，这时再进入**游戏战斗场景**。

**对象(who && what)：**游戏初始场景、英雄玩家、战斗游戏场景

**核心行为关系(how)：**通过游戏初始场景创建一个英雄玩家

1. 通过**游戏战斗场景**说明，你会发现地球正在被外太空生物侵袭，而外太空一共有5种怪物，每个怪物的等级分别从低到高（lv1， lv5，lv10， lv30， lv50），其中等级为50的怪物为最终大BOSS，**英雄玩家**需要从这5种**展示怪物**中选择一只**被挑战的怪物**，选中具体怪物后，进入战斗模式，最终英雄玩家只需要战胜大BOSS，即认为游戏胜利！

**对象(who && what)：**游戏战斗场景、英雄玩家、展示怪物、被挑战怪物

**核心行为关系(how)：**英雄玩家选择怪物、游戏战斗场景创建被挑战的怪物

1. 战斗过程中如果**英雄玩家**挑战胜利，则英雄增加的等级数和**被挑战怪物**的等级数持平，例如英雄当前等级为7级，挑战10级怪兽成功后，则英雄等级更新为17级；反之，如果英雄挑战失败，则等级数降1，如果英雄等级数降为0，则认为英雄死亡，游戏失败！

**对象(who && what)：**英雄玩家、被挑战怪物

**核心行为关系(how)：**英雄玩家根据战斗结果提升或降低自身等级

1. 按照游戏规则，随着**英雄玩家**等级的提升或降低，英雄的攻击力也随之提升或降低，且英雄有一个**技能列表**，英雄的等级不同，对应可使用的战斗技能也不同，例如**普通击打**是整个攻击技能里攻击力最弱的一种，英雄玩家可以通过释放技能来攻击**怪兽**，但是每个技能都有冷却次数，例如**寒冰掌**的冷却次数即为2。

**对象(who && what)：**英雄玩家、技能列表、普通击打、怪兽、寒冰掌

**核心行为关系(how)：**英雄玩家释放技能攻击怪兽

1. **英雄玩家**和**被挑战怪兽**在每一局的比赛中都有各自的生命值，该生命值和自身的等级相关，无论是英雄还是怪兽在当局比赛中如果生命值小于等于0，则认为该局死亡。如果当局英雄生命值为0，则英雄当局挑战失败，等级降1级；反之如果当局怪兽生命值为0，则英雄挑战成功，英雄等级增加怪兽的等级数，每一次战斗结束后，在**游戏场景**中需要显示当前英雄玩家的信息。

**对象(who && what)：**英雄玩家、被挑战怪兽、游戏场景

**核心行为关系(how)：**战斗结束后游戏场景显示英雄玩家信息

### 3.2.2 综合实例1---LostCraft游戏项目之类的抽象

通过上述对综合案例一（LostCraft游戏）再次进行分析后，相信大家应该隐约发现每一段行为描述中所存在的一些共同点了吧？

下面我们就按照面向对象的思考模式进行分析和抽象总结。

首先我们来分析整个游戏中所出现过的所有对象，他们分别是：游戏初始场景、英雄玩家、游戏战斗场景、展示怪物、被挑战怪物、技能列表、普通击打、寒冰掌。

其次我们针对上述具体对象进行整理总结，我们发现以下特点：

1. 整个游戏中虽然只有英雄玩家一个对象，但无论英雄玩家由谁创建，他在游戏中的特征和行为都是不变的，例如英雄玩家都有名称，即使不同的人给英雄的取名不同，可是它都可以统称为名称，以此类推，我们发现英雄玩家还有许多其他的特征和行为。

|  |
| --- |
| **共同特征：**名称、级别、生命值HP、能量值MP、技能列表  **共同行为：**初始化英雄、选择技能、英雄进行攻击、判断英雄是否存活、英雄等级提升、英雄等级下降、结算英雄当前回合状态 |

英雄玩家有各式各样的，但是无论它们叫什么、战斗过程如何，它们都拥有共同的特征和行为，**所以我们把这种拥有共同特征和行为的英雄玩家们都归为一类：****英雄类**。

1. 无论是普通击打还是寒冰掌，以及其他剩余的技能，即使他们的名称不同、攻击力不同，但是所拥有的特征和行为都是相同的，例如所有的技能都有编号，即使技能的编号不同，可是它都可以统称为编号；又例如作为技能本身，无论哪一种技能都没有自身行为，以此类推，我们发现所有的技能都还有拥有许多共同的特征和行为。

|  |
| --- |
| **共同特征：**编号、名称、消耗MP数、攻击力、技能规定冷却轮数、技能所需英雄等级数、技能冷却时间  **共同行为：**无行为 |

技能有各式各样的，但是无论它们叫什么、消耗多少MP数值，它们都拥有共同的特征和行为，**所以我们把这种拥有共同特征和行为的技能都归为一类：技能类**。

1. 无论是展示怪物还是被挑战怪物，无论他们叫什么、攻击力如何，但是它们所拥有的特征和行为都是相同的，例如所有的怪物都有攻击力，即使攻击力不同，可是它都可以统称为攻击力；又例如无论是哪一种怪物，它都具有攻击行为，以此类推，我们发现所有的怪物都拥有许多共同的特征和行为。

|  |
| --- |
| **共同特征：**名称、级别、攻击力、生命值HP  **共同行为：**初始化怪物、怪物进行攻击、判断怪物是否存活 |

怪物有各式各样的，但是无论它们叫什么、攻击力是多少，它们都拥有许多共同的特征和行为，**所以我们把这种拥有共同特征和行为的怪兽们都归为一类：怪兽类**。

1. 无论是游戏初始场景还是游戏战斗场景，他们都具有一些相同的特征和行为，例如无论在什么场景中都包含英雄玩家；又例如在游戏中，场景要不停的提供战斗的环境等等，以此类推，我们发现在整个游戏中场景是一个为所有的行为关系提供基本环境的所在。

|  |
| --- |
| **共同特征：**英雄对象、怪物对象、游戏当前状态、当前回合轮次  **共同行为：**初始化场景、选择怪物、挑战环节开始、显示每一轮战斗回合结束后英雄和怪物的信息、战斗结束后显示所有的信息 |

场景有各式各样的，但是无论它们是什么场景、提供什么样的环境，它们都拥有许多共同的特征和行为，**所以我们把这种拥有共同特征和行为的场景都归为一类：场景类**。

**最终，整个游戏共设计为四个类，分别是：英雄类、技能类、怪物类以及场景类。**那么，到底什么是类、类有哪些特征等等一些列问题我们将在下一节中为大家详细讲解。

### 3.2.3 类与对象的概念与关系

在我们上节介绍过的游戏世界中，一切事物都是对象。对象可以是有形的，比如英雄、怪物或者汽车、飞机；也可以是无形的，比如技能、场景或者一项计划。对象可以是一个简单的个体，比如一个英雄；也可以由诸多其他对象组合而成，比如一个英雄有多个技能。对类似的对象进行抽象，找出其共同属性，便构成一种类型。这些都是我们在游戏世界中所熟悉的概念和方法。编写程序的目的是描述和解决游戏世界中的问题，第一步就是要将游戏世界中的对象和类如实地反映在程序中。作为一种面向对象的程序设计语言，C++支持这种抽象。将抽象后的数据和函数封装在一起，便构成了C++的“类”。

**1. 类是面向对象程序设计方法的核心，利用类可以实现对数据的封装和隐蔽。**

在面向过程的结构化程序设计中，程序的模块是由函数构成的，函数将逻辑上相关的语句与数据封装，用于完成特定的功能。**在面向对象程序设计中，程序模块是由类构成的。类是对逻辑上相关的函数与数据的封装，它是对问题的抽象描述。**因此，后者的集成程度更高，也就更适合用于大型复杂程序的开发。

上一节中从抽象和封装的角度引出了类的概念。对于初学者来说，不妨再从另一个更简单的角度来理解类。首先让我们回顾一下基本数据类型，比如int，double，bool等。当声明一个基本类型的变量时，究竟声明了什么呢?请看下面的语句：

int i;

double b;

显然这声明了变量i是用于存储int型数据的，变量b是用来存放double型数据的。但是变量声明的意义不只是这个，另一个同样重要的意义常被我们忽略了，这就是限定对变量的操作。例如对i可以进行算术运算、比较运算等，对b也可以进行算术运算、比较运算。这说明每一种数据类型都包括了数据本身的属性以及对数据的操作。

无论哪一种程序语言，其基本数据类型都是有限的，C++的基本数据类型也远不能满足描述游戏世界中各种对象的需要。于是C++的语法提供了对自定义类型的支持，这就是类。类实际上相当于一种用户自定义的类型，原则上我们可以自定义无限多种新类型。因此不仅可以用int类型的变量表示整数，也可以用自定义类的变量表示“英雄”、 “怪物”、“技能”或者“场景”等对象。**正如基本数据类型隐含包括了数据和操作，在定义一个类时也要说明数据和操作。**

当定义了一个类之后，便可以定义该类的变量，这个变量就称为类的对象（或实例），这个定义的过程也称为类的实例化。

1. **类实际上是一种抽象机制，它描述了一类问题的属性和行为，即：类描述了一类对象的数据存储和操作特性。**在C++中，类的对象就是具有该类类型的某一特定实体（也成实例）。例如，将怪物们看作一个类，那么每一个怪物就是该类的一个特定实体，也就是一个对象，并且每一个怪物都有一些共同的**静态特性**，我们把这些共同的特性抽象到类中，统称为该类的**属性**，一般在代码中，**属性通过声明普通变量来实现**；同时每一个怪物都有一些共同的**动态行为**，我们把这些共同的行为抽象到类中，统称为该类的**操作**，一般在代码中，**操作通过声明函数来实现**；

例如：**英雄类的伪类图：**

英雄 类名

｛ 边界开始

|  |  |
| --- | --- |
| 名称；  级别；  生命值HP；  能量值MP；  技能列表； | 属性 |
| 初始化英雄；  选择技能；  英雄进行攻击；  判断英雄是否存活；  英雄等级提升；  英雄等级下降；  结算英雄当前回合状态； | 操作 |

｝ 边界结束

**技能类的伪类图**：

技能 类名

｛ 边界开始

|  |  |
| --- | --- |
| 编号；  名称；  消耗MP数；  攻击力；  技能规定冷却轮数；  技能所需英雄等级数；  技能冷却时间； | 属性 |

｝ 边界结束

**注：通过技能类我们发现，在一个类中，并不一定要同时存在属性和操作，甚至一个类中可以没有任何属性或操作，****类是灵活的，是由设计者自己设计的类型，因此设计者可根据不同的项目需求进行不同的设计。**

**怪物类的伪类图：**

怪物 类名

｛ 边界开始

|  |  |
| --- | --- |
| 名称；  级别；  攻击力；  生命值HP； | 属性 |
| 初始化怪物；  怪物进行攻击；  判断怪物是否存活； | 操作 |

｝ 边界结束

**场景伪类图：**

场景 类名

｛ 边界开始

|  |  |
| --- | --- |
| 英雄对象；  怪物对象；  游戏当前状态；  当前回合轮次； | 属性 |
| 初始化场景；  选择怪物；  挑战环节开始；  显示每一轮战斗回合结束；  后英雄和怪物的信息；  战斗结束后显示所有的信息； | 操作 |

｝ 边界结束

实际上，每一种数据类型都是对一类数据的抽象，在程序中生命的每一个变量都是其所属数据类型的一个实例。若把类看作是自定义的数据类型，那么，类的对象就可以看成是该类型的一个变量。

注意：对象所占的内存空间只是用于存放数据成员，函数成员不在每一个对象中存储副本，每个函数的代码在内存中只占据一份空间。

1. **类和对象的关系**

类是抽象的，它描述的是一类事物所应具有的属性，而对象则是类的具体化，是类的一个实例，一个类可以创建无数个对象。例如假设技能是一个类，那么寒冰掌、石破惊雷就是它的对象。

类是对象实例的（可能的）无限集合，同一类中的所有对象都具有相同的属性描述和操作集、不同的对象标识符。例如无论是寒冰掌、石破惊雷，还是其他的技能，他们所拥有的属性都是一样的，例如都有名称、攻击力等等。

对象可以被创建也可以被销毁。例如大BOSS赤龙怪，它就是怪物类所创建的一个具体对象，同时在英雄挑战成功时，该对象被销毁。

# 第四章 面向对象思维建摸

## 4.1 UML漫谈之类图与时序图

Unified Modeling Language (UML)又称**统一建模语言**或[**标准建模语言**](http://baike.baidu.com/view/3458435.htm)，它是一个支持模型化和软件系统开发的图形化语言，为软件开发的所有阶段提供模型化和可视化支持，包括由需求分析到规格再到构造和配置。

本书利用UML来表现设计模式中类和对象实例的关系，所以在此先列举一些阅读本书时所必备的UML常识，由于UML的规格相当庞大，本书中所解说的范围仅仅为其中一小部分。如果各位读者有兴趣进一步了解UML，可到下列网站取得UML的详细规格说明。

1. UML Resourse Page <http://www.omg.org/uml/>
2. UML Resourse Center [http://www.retional.com/uml/](http://www.omg.org/uml/)

UML从考虑系统的不同角度出发，定义了用例图、类图、对象图、状态图、活动图、序列图、协作图、构件图、部署图等9种图，这些图从不同的侧面对系统进行描述，系统模型将这些不同的侧面综合成一致的整体，便于系统的分析和构造。而在本书中，我们只解说类图与时序图（又名序列图、顺序图）这两种图。

### 4.1.1 类图

类图是最常用的UML图，在做C++的项目中，画类图是必不可少的工作之一。它**显示了模型的静态结构，特别是模型中存在的类、类的内部结构以及它们与其他类的关系等**。

在画类图之前我们要确定几点：

1. 研究分析问题领域确定系统需求。
2. 确定类，明确类的含义和职责，确定属性和操作。
3. 确定类之间的关系。

类图主要分为三个部分，三个部分从上到下，分别写类的“类名”，“该类所包含的所有属性”和“该类所包含的所有操作”。具体类图的绘制格式如下图所示：



以案例一LostCraft游戏中的英雄类和技能类为例，下面我们将画出UML类图与之前我们用文字语言画出的伪类图一一对应。

怪物

|  |  |  |
| --- | --- | --- |
| ｛ |  | 怪物 |
| 名称；  级别；  攻击力；  生命值HP； | 伪类图转化为UML类图 | 名称；  级别；  攻击力；  生命值HP； |
| 初始化怪物；  怪物进行攻击；  判断怪物是否存活； |  | 初始化怪物；  怪物进行攻击；  判断怪物是否存活； |
| ｝ |  |  |

技能

|  |  |  |
| --- | --- | --- |
| ｛ |  | 技能 |
| 编号；  名称；  消耗MP数；  攻击力；  技能规定冷却轮数；  技能所需英雄等级数；  技能冷却时间； | 伪类图转化为UML类图 | 编号；  名称；  消耗MP数；  攻击力；  技能规定冷却轮数；  技能所需英雄等级数；  技能冷却时间； |
| ｝ |  |  |

通过上述对应想必大家应该很快的了解到类图的绘制格式了吧！

**类与类之间的常见关系**

回顾之前我们对案例一的分析，每个类并不一定是完全独立的，他有可能和其他类之间发生某种联系，下面我们就来说明几种常见的类与类之间的关系以及对应类图的表示方法。

类与类之间常见的关系有：继承关系、关联关系、聚合关系、组合关系以及依赖关系等。而我们在本书中仅讲解依赖和继承两种关系，所解说的范围也是很小一部分，仅仅能够满足我们本书中案例所需的关系。

1. 依赖关系

依赖就是一个类A使用到了另一个类B，而这种使用关系是具有偶然性的、临时性的、非常弱的，但是类B的变化会影响到类A。 因此类A依赖类B，在UML类图设计中，依赖关系用由类A指向类B的带箭头虚线表示。

|  |  |  |
| --- | --- | --- |
| 人 |  | 船 |
| 姓名； |  |  |
| 使用（船）； |  |  |

如上图所示，假设当前某人要过河，需要借用一条船，此时人与船之间的关系就是依赖。首先如果没有船，人无法过河，因此是**人依赖船**，同时人并不是每天都在使用船过河，因此在该关系中体现出依赖关系的偶然性和临时性，作为依赖方，人有一个使用行为，而船则作为一个函数参数被人使用。

1. 继承关系

继承指的是一个类（称为子类）继承另外的一个类（称为父类）的功能，并可以增加它自己的新功能的能力。在UML类图设计中，继承用一条带空心三角箭头的实线表示，从子类指向父类。

|  |  |  |
| --- | --- | --- |
| 父亲 |  | 儿子 |
| 姓名；  年龄； |  | 学历； |
|  |  |  |

如上图所示，上图所表示的含义即为**儿子类继承父亲类**，由图可知，父亲类中有2个属性，分别是：姓名、年龄，由于儿子类继承父亲类，因此儿子类中不仅把父亲类中所有的属性和行为继承下来，还能拓展自己的新功能，因此儿子类中的属性有3个，分别是：姓名、年龄以及学历。

### 4.1.2 时序图

**时序图，亦称为**[**序列图**](http://baike.baidu.com/view/2226296.htm)**或循序图或顺序图，是一种UML交互图，它通过描述对象之间发送消息的时间顺序显示多个对象之间的动态协作。**它可以表示[用例](http://baike.baidu.com/view/706238.htm)的行为顺序，当执行一个用例行为时，时序图中的每条消息对应了一个类操作或[状态机](http://baike.baidu.com/view/1906565.htm)中引起转换的触发事件。

**类图表示“不因时间而变化的部分（静态关系）”，而时序图则表示“随时间而变化的部分（动态行为）”。**

因此，在画时序图之前我们要确定几点：

1. 确定交互过程的上下文；
2. 识别参与过程的交互对象；
3. 为每个对象设置生命线；
4. 从初始消息开始，依次画出随后消息；

时序图主要是由对象（Object）、生命线（Lifeline）、激活（Activation）、消息（Messages）、分支与从属流等元素构成的。以案例一LostCraft游戏中的英雄玩家攻击巴尔坦虫怪为例，下面我们来分析一下时序图的元素组成和绘制流程。

1. 对象

对象就是指类的实例，在序列图中对象大致分为三种状态：激活、运行（存在）和销毁。我们用矩形表示一个对象。

英雄玩家

巴尔坦虫怪

1. 生命线

生命线是一条垂直的虚线，用来表示时序图中的对象在一段时间内的存在。

英雄玩家

巴尔坦虫怪

1. 激活

时序图可以描述对象的激活，激活是对象操作的执行，它表示一个对象直接或通过从属操作完成操作的过程。在UML图中通过一个窄长的矩形来表示，矩形的高度表示对象存在的过程。

英雄玩家

巴尔坦虫怪

1. 消息

消息是对象间的一种通信机制。由发送对象向另一个或其他几个接收对象发送信号，或由一个对象（发送者或调用者）调用另一个对象（接收者）的操作。

英雄玩家

巴尔坦虫怪

攻击（）

怪物死亡

等级提升（）

**同步消息（调用消息）：**消息的发送者把控制传递给消息的接收者，然后停止活动，等待消息的接收者放弃或者返回控制，用来表示同步的意义。

**异步消息：**消息发送者通过消息把信号传递给消息的接收者，然后继续自己的活动，不等待接受者返回消息或者控制。异步消息的接收者和发送者是并发工作的。

**返回消息：**返回消息表示从过程调用返回。

## 4.2 项目实现之UML设计

### 4.2.1 综合实例1---**LostCraft**游戏项目之UML设计

**类图设计**

下面我们就来完成整个游戏的UML类图的设计。相信聪明的你，一定能够很快的看懂下面类图的含义！

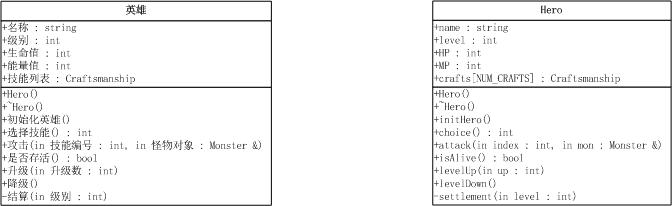
1. **英雄类**

类名：英雄

该类可生成的对象：玩家在游戏中所创建的具体英雄，即玩家本人

属性：名称、级别、生命值HP、能量值MP、技能列表

行为：初始化英雄、选择技能、英雄进行攻击、判断英雄是否存活、英雄等级提升、英雄等级下降、结算英雄当前回合状态



**伪类图转化为类图**

**注：在英雄类图中，其行为的部分，有两个函数，函数名为类名，分别是Hero()和~Hero()，注意这两个函数分别被称为构造函数和析构函数，是程序创建或删除对象时自动调用的两个函数，该部分内容我们将在实践篇中进行讲解，当前同学们在类图中可画可不画，以下其他类图同理。**

**英雄类相关属性说明：**

1. 英雄自身等级默认从1级开始；
2. 英雄最低血量HP为2000，每升一级，血量增加500；
3. 英雄最低魔法值MP为50，每升一级，魔法值增加10；

具体数据由下表所示：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 姓名 | 等级 | HP | MP | 技能列表 |
| 该名称由用户自己取名 | 1 | 2000 | 50 | 该部分参考技能列表 |
| 2 | 2000+1\*500 | 50+1\*10 |
| 3 | 2000+2\*500 | 50+2\*10 |
| 4 | 2000+3\*500 | 50+3\*10 |
| … |  |  |
| n | 2000+(n-1)\*500 | 50+(n-1)\*10 |

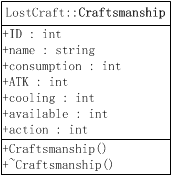
1. **技能类**

类名：技能

该类可生成的对象：英雄在游戏过程中具体可使用的特殊技能

属性：编号、名称、消耗MP数、攻击力、技能规定冷却轮数、技能所需英雄等级数、技能冷却时间

行为：无行为

**转 化**

**技能类相关属性说明：**

1. 技能编号用于表示某一个技能在技能列表里的序号；
2. 英雄必须达到某一定的等级之后，才可能开启可让其使用的相关技能；
3. 不同的等级所消耗的MP数也不同；
4. 技能的攻击力与英雄当前的级别相关；
5. 不同的技能有规定的冷却轮数；

具体数据由下表所示：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| ID | 名称 | 技能所需的英雄等级数 | 消耗MP数 | 攻击力 | 冷却轮数 |
| 1 | 普通击打 | 1 | 0 | 10\*当前级别 | 0 |
| 2 | 寒冰掌 | 1 | 5 | 12\*当前级别 | 1 |
| 3 | 石破惊雷 | 5 | 10 | 15\*当前级别 | 3 |
| 4 | 赤火焰烧 | 10 | 20 | 18\*当前级别 | 3 |
| 5 | 月蚀镰刀 | 20 | 40 | 20\*当前级别 | 4 |
| 6 | 圣光冲击 | 30 | 60 | 22\*当前级别 | 4 |
| 7 | 天师灵助 | 35 | 80 | 25\*当前级别 | 5 |
| 8 | 天威庇佑 | 45 | 100 | 30\*当前级别 | 5 |

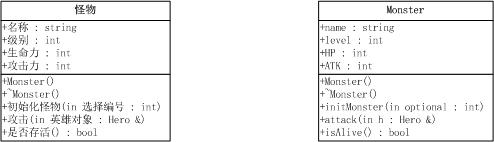
1. **怪物类**

类名：怪物

该类可生成的对象：游戏中每一次英雄挑战的某只具体怪物

属性：名称、级别、攻击力、生命值HP

行为：初始化怪物、怪物进行攻击、判断怪物是否存活



**伪类图转化为实际类图**

**怪物类属性说明：**

系统会自动生成5种怪物，每个怪物的等级分别从低到高（lv1， lv5，lv10， lv30， lv50），其中等级为50的怪物为最终大BOSS，英雄只需要战胜一次大BOSS，即认为游戏胜利！

具体数据由下表所示：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ID | 怪物名称 | 怪物等级 | HP | 攻击力 |
| 1 | 巴尔坦虫怪 | 1 | 100 | 50 |
| 2 | 格斯安鱼怪 | 5 | 500 | 100 |
| 3 | 泰斯鸟怪 | 10 | 1000 | 200 |
| 4 | 绿头猿怪 | 30 | 2000 | 500 |
| 5 | 赤龙怪(BOSS) | 50 | 5000 | 1000 |

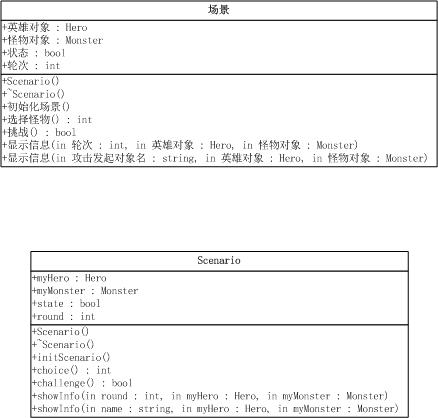
1. **场景类**

类名：场景

该类可生成的对象：游戏开始后生成的具体战斗场景

属性：英雄对象、怪物对象、游戏当前状态、当前回合轮次

行为：初始化场景、选择怪物、挑战环节开始、显示每一轮战斗回合结束后英雄和怪物的信息、战斗结束后显示所有的信息



**转化**

**时序图设计**

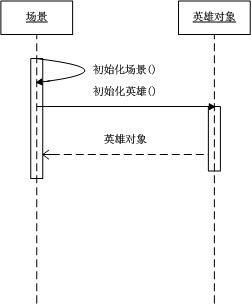
既然类图已经确定，下面我们就使用UML来画出对象之间是如何协作的。回顾之前所设计的四个类：英雄类、技能类、怪物类和场景类，我们发现当游戏开始之后，随着时间的变化，每个对象角色之间的行为关系也在发生着不同的变化，下面我们就按照游戏时间的发展顺序，简单的对角色之间发生的主要行为关系进行描述。

1. **游戏开始，场景初始化英雄**

首先游戏开始，我们需要为游戏创建一个操作环境，即生成我们所需要的游戏场景，这时我们就需要进行游戏初始化。

通过分析可知该行为包括两个对象，一个是游戏场景，另一个是英雄对象，游戏开始后首先进行场景初始化，即创建出可操作的游戏环境，其次进行英雄初始化，即创建出该游戏里的主人翁玩家，然后将该玩家信息返回给当前游戏场景。

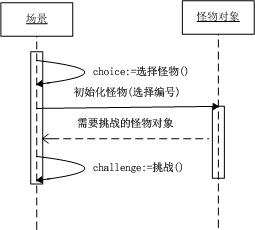
相关时序图如下图所示：



1. **英雄通过场景选择怪物并怪物初始化**

随着游戏开始，玩家首先要从游戏场景中选择需要挑战的怪物；其次，场景会根据玩家选择的怪物编号进行怪物初始化，即创建出玩家需要挑战的这个怪物对象，并将该对象返回给当前游戏场景。

相关时序图如下图所示：



1. **每一个战斗回合英雄攻击怪物，其次怪物反击**

战斗打响后，每一场战斗中每一个回合开始时，首先显示当前英雄玩家以及选择挑战的怪兽对象所有信息；其次英雄选择攻击技能；英雄根据所选的技能进行攻击行为，其次怪物进行反击，此时一个战斗回合结束；每当战斗回合结束后再次显示最新的英雄和怪物的所有信息。

相关时序图如下图所示：

**《该图暂定》**

### 4.2.2 综合实例2---YYY系统项目之UML设计

### 4.2.3 综合实例3---ZZZ系统项目之UML设计

# 实践篇

# 第五章 程序设计语言C++初识

## 5.1 C++概述

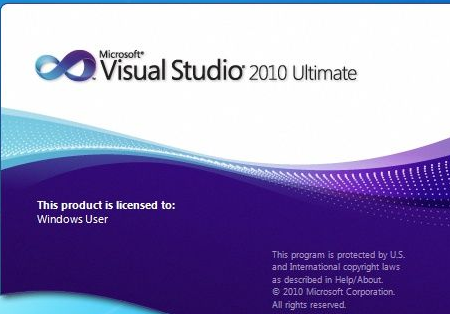
1983年，[贝尔实验室](http://baike.baidu.com/view/37115.htm)的[Bjarne Stroustrup](http://baike.baidu.com/view/687922.htm)在C语言基础上在C语言的基础上成功开发了C++语言， 该语言进一步扩充和完善了C语言，几乎保留了C语言原有的所有优点，并在此基础上增加了面向对象的机制，是一种[面向对象](http://baike.baidu.com/view/125370.htm)的程序设计语言。

在计算机科学中，C++仍被称为C语言的上层结构。总得来说，C++语言的主要特点表现在两个方面，一是尽量兼容C；二是支持面向对象的方法。它操持了C的简洁、高效的接近汇编语言等特点，对C的类型系统进行了改革的扩充，并且C++的编译系统能检查出更多的类型错误，因此C++比C更安全。出于保证语言的简洁和运行高效等方面的考虑，C++的很多特性都是以库（如STL）或其他的形式提供的，而没有直接添加到语言本身里。

C++语言最有意义的方面是支持面向对象的特征，使得开发人机交互类型的应用程序更为简单、快捷。虽然与C的兼容使得C++具有双重特点，但他在概念上完全与C不同，更具面向对象的特征。

## 5.2 第一个C++程序

### 5.2.1 VS2010工具介绍

 要想学好C++语言，选择一个好的C++语言开发环境是很有必要的。在本节中，我们主要来看看C++语言的开发过程，并教大家学习Windows平台上主流的C++语言开发工具Visual Studio的基本使用方法。

[Visual Studio](http://baike.baidu.com/view/28727.htm)是微软公司推出的开发环境，

是目前最流行的[Windows](http://baike.baidu.com/view/4821.htm)平台应用程序开发环

境。Visual Studio 2010版本于2010年4月12

日上市，其[集成开发环境](http://baike.baidu.com/view/14867.htm)（[IDE](http://baike.baidu.com/view/5775.htm)）的界面被重新

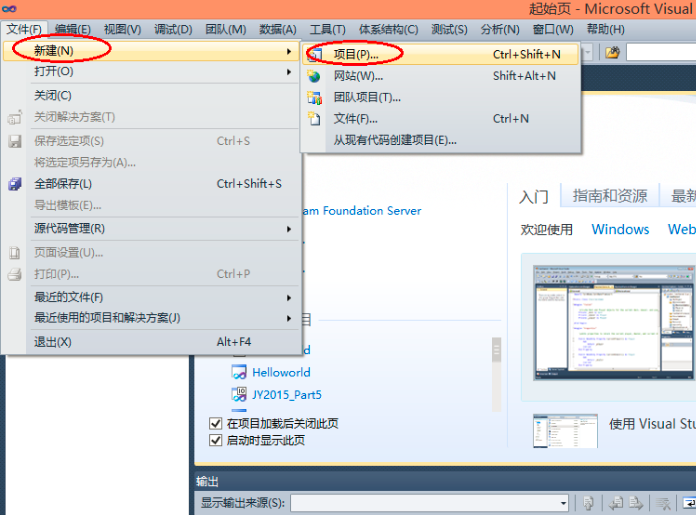
设计和组织，变得更加简单明了。

Visual Studio的全称为Microsoft Visual Studio，意思是“微软可视化工作室”，简称为VS。它包含VB、VC、VF、Delphi、控件、数据库ODBC等开发工具，其中VC就是用来进行C和C++开发的，表示Visual C/C++。VS是由微软公司开发的，用于开发Windows程序的工具，要追溯它的版本信息，那就久远了。不过主要被使用的版本有VS 6.0、VS 2003、VS 2005、VS 2008、VS 2010。每一个版本中都会包含一个VC开发组件或工具，用于C和C++的开发，我们在学习C++语言的时候，使用的就是VC开发工具。

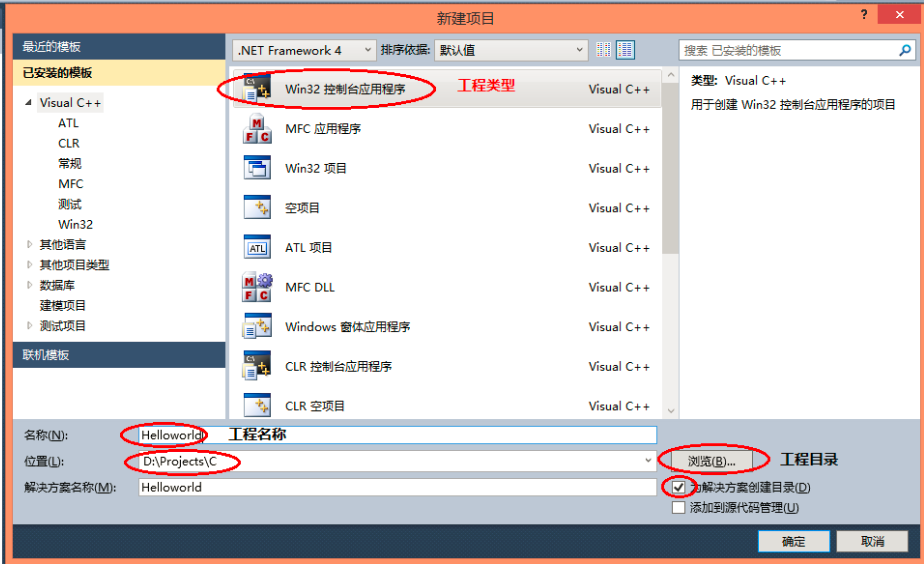
对于众多版本的VS，你可以任选一个进行安装使用，它们之间的差别不是很大，不过，还是建议尽量选择较新的版本，因为更新的版本对于标准的支持往往比较好，而且功能会更强大，所以本门课程我们就将使用VS2010版本来进行开发。

### 5.2.2 编写第一个C++程序helloworld

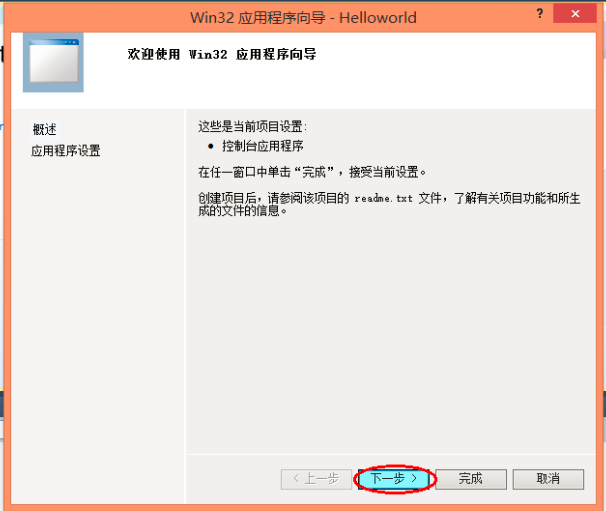
1. 选择新建工程



1. 选择工程类型为win32控制台，确定工程存放位置（目录）以及工程名称



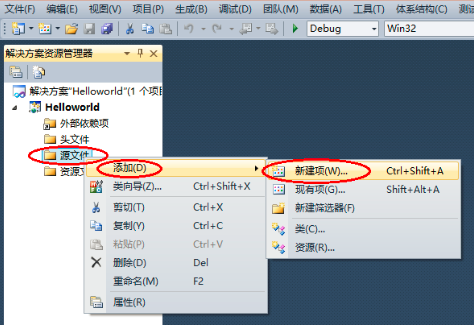
1. 选择下一步



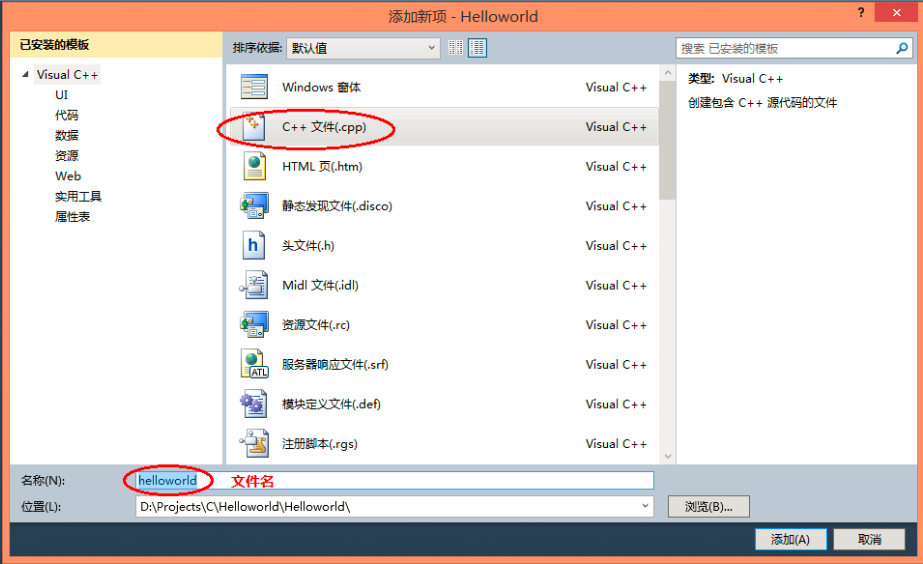
1. 在额外选项中选中“空工程”，点击完成按键完成创建工程。



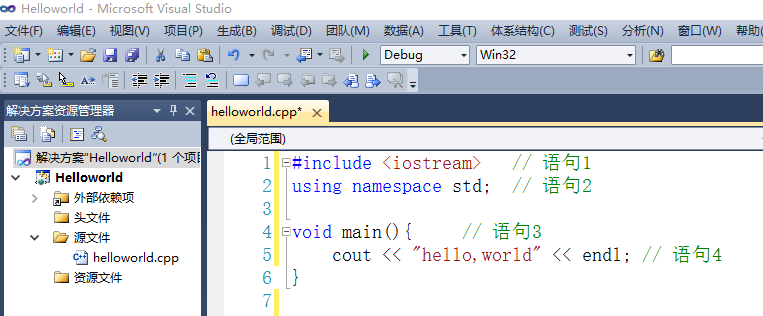
1. 向工程中添加源文件helloworld.cpp



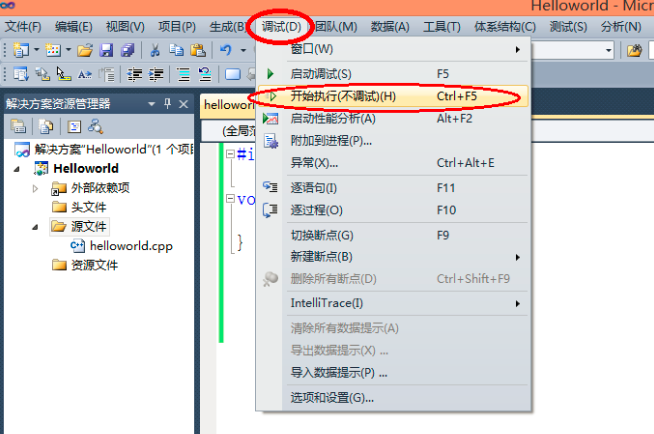
1. 选择文件类型为“C++文件”，输入文件名“helloworld”



1. 工程和源文件都建好了，开始输入代码



1. 输入完代码，点击工具栏快捷键或者菜单栏选择运行程序，观看运行结果



### 5.2.3 代码解释

#include <iostream> // 语句1

using namespace std; // 语句2

void main() // 语句3

{

cout << “hello world” << endl; // 语句4

}

1. 语句1包含两部分知识 #include与iostream

* 在使用 C语言库函数时，要用**预编译命令“#include”将有关的“头文件”包括到用户源文件中**。在头文件中包含了与用到的函数有关的信息。例如使用标准输入输出库函数时，要用到<iostream>文件。#include命令都是放在程序的开头，因此这类文件被称为“头文件”。
* **iostream是一个文件库**，**其代表意思是输入输出流**。直接点说就是in（输入）、out（输出）、stream（流），取in、out的首字母与stream合成，用于处理C++程序中的输入、输出。程序的语句4中使用的输出流cout是在该文件中“创建”的。C++规定程序要使用某个“事物”，例如cout，需要将“创建”它的文件，例如iostream导入到程序中。故这里要用语句1将iostream文件导入到程序中。

1. 语句2意思是“使用命名空间std”

std是什么呢？**它是一种命名空间，属于iostream库，**在不同的命名空间下的“事物”可以重名，这样我们就不要绞尽脑汁为很多类似的“事物”起不同的名字了，所以如果在程序中去掉语句2，那么程序就搞不清cout是称呼哪个“事物”了。

例如：在现实生活中，很多城市都有滨湖区，比如合肥和苏州，怎么区别它们呢，我们可以这样说：合肥的滨湖区和苏州的滨湖区，或者在引用它们的文章的开始处注明：

**下面使用的都是苏州的名字；(该部分等价于定义一个命名空间“苏州”)**

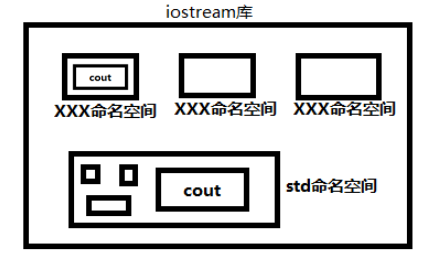
…滨湖区…建设…

…滨湖区…发展…

这样我们在省略号处描述得滨湖区就都指的是苏州市的，这就是语句2使用命名空间的思想。

同理，语句2使程序知道cout是命名空间std下的cout，大家可以尝试去掉语句2，那样要使用cout就需要称呼它为std::cout。**“::”是一个引用符号**，我们以后会用到，暂时可不做理解，

下面我们就直接使用一幅图来表示iostream，std，cout的关系吧！



1. 语句3定义函数main

**C++程序必须有一个且仅有一个函数main作为程序的入口**，无论main函数在工程中的什么位置，程序都将从main函数里的代码开始运行，main函数里面的代码执行完毕，则整个程序就执行完毕，在这里我们定义该函数，并把希望程序运行的语句放在函数内的一对大括号”{”和”}”之间。

1. 语句4是这个小程序的核心功能语句

**cout是**在iostream中定义的负责**输出**的对象，**”cout<<”后面接字符串的含义是将该字符串输出到控制台上。”<<endl”的作用是继续输出一个回车**，这样内容输出完成后则会换行。

现在我们不要求大家能准确的理解程序中每条语句的含义。随着后续章节的学习，我们会达到那样的阅读水平。现在我们只要**记住**：

* 每个C++程序都要定义一个main函数作为程序的入口。程序要执行的语句写在main函数内的大括号{}内。每条语句都以分号结尾。为方便程序阅读，一般一条语句占一行。
* 如果程序有输入或输出的需要，必须使用”#include <iostream>”命令以提供必要的信息，同时要用”using namespace std;”使其中的元素可见。

附：用#include导入的文件的可能存放于几种位置，比如当前工程的目录下，以及例如与VS2010安装位置有关的目录下等等其它可大体概括为“安装目录”下的各类目录中。C++语言使用#include “文件名”和#include <文件名>这两种方法来减少寻找这些文件的时间。

**#include “文件名” 🡺** 先寻找工程当前目录，找不到再在“安装目录”下寻找。**适用于导入用户自定义的、处在工程当前目录下的文件。**

**#include <文件名> 🡺** 不寻找工程当前目录，直接在“安装目录”下寻找。**适用于导入类似iostream等那些“标准库”中的文件。**

### 5.2.4 程序注释

程序注释是以特定格式出现在源文件中的文本，它不是程序的执行语句的一部分，而是程序员解释执行语句的文字性说明。

C++程序中有两种注释：

1. **单行注释**

//…

以连续两个双斜杠注释标识开始的一段文本，表示双斜杠到行尾的文字都是注释，不是程序语句。例如下述代码所示：

cout << “hi” << endl; // 我是一句单行注释哦~我这部分程序是不会执行的~

1. **多行注释**

/\*…

…

…

\*/

注释标识（/\*，\*/）对内的内容都是注释，并且其中的内容可以跨越多行。

注释常用的两个作用：

* 阐明一段比较难懂的程序代码的含义、流程或者在代码中的承接作用。
* 对若干条程序语句做注释，达到在不删除这些语句的前提下让程序不执行这些语句，这样以后在需要用到这些代码时，可以通过删除这些注释标识，来恢复程序语句的执行，而不需要再写一遍。

例如下述代码所示：

/\*

我是多行注释哦~

我这部分程序是不会执行的~

所以我可以写很多很多行~

都不会影响程序本身~

\*/

cout << “hi” << endl;

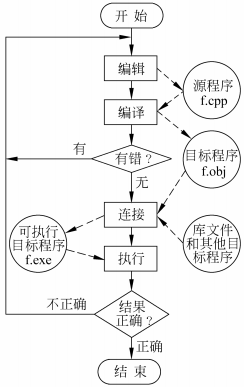
## 5.3 C++编译工作原理

一个程序从编写到最后得到运行结果要经历以下一些步骤：

1. 用C++语言编写程序。用高级语言编写的程序称为“源程序”，C++源程序是以“.cpp”作为文件名后缀的。
2. 对源程序进行编译。为了使计算机能执行高级语言源程序，必须先用一种称为“编译器（compiler）”的软件把源程序翻译成二进制形式的“目标程序”。

编译是以源程序文件为单位分别编译的。目标程序一般以.obj或者.o作为后缀。编译的作用是对源程序进行词法检查和语法检查。编译时对文件中的全部内容进行检查，编译结束后会显示出所有的编译出错信息。**一般编译系统给出的出错信息分为错误（error）和警告（warning）**。

1. 将目标文件连接。改正所有的错误并全部通过编译后，得到一个或多个目标文件。此时要用系统提供的“连接程序（linker）”将一个程序的所有目标程序和系统的库文件以及系统提供的其他信息连接起来，最终形成一个可执行的二进制文件，它的后缀是.exe，是可以直接执行的。
2. 执行程序。运行最终形成的可执行的二进制文件（.exe文件），得到运行结果。
3. 分析运行结果。如果运行结果不正确，应检查程序或算法是否有问题。



## 5.4 C++编程逻辑

### 5.4.1 程序中简单数据存储

1. **程序数据在内存中存取**

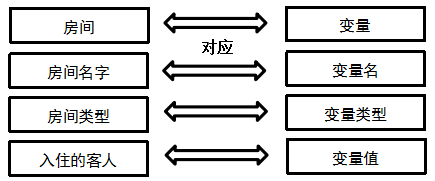
计算机使用内存来存储所使用的数据，那么内存又是如何来存储数据的呢？

举个简单的例子，客户M去旅馆开房间。

* 首先M要在前台选择要入住的是单人间，双人间还是套房，相当于选择房间类型。
* 前台根据房间类型从旅馆所有房间中为它随机打开一间同类型的房间，假设该房间名为room1。
* 客户M入住room1。
* 客户在入住room1期间有任何问题呼叫前台寻求帮助时，都会产生如下对话：“您好，请问您是哪间房间的客人？”“我是room1的客人”。
* 入住期间客户M可以随时退房离开。
* 客户M不再住宿后，该房间仍有可能住宿其他客户，即房间的客人可以随时发生变化。

通过这个例子，我们可以做以下**几点总结**：

* 内存和旅馆一样，会根据客户的需求类型随机开辟一块内存空间（即一间房间）。
* 内存中的数据可以随时发生变化，例如room1里面客人是时时在变化的一样，我们把这种在程序运行过程中可能发生变化的数据称之为**变量**。
* 每个房间都有一个房间名，同样每个变量都有一个变量名，例如room1就可以是一个变量名，但是房间必须取名room1吗？当然不是！如果客户喜欢，它可以给自己住的房间取任何名字，同理，**变量名也是由客户取得，只要遵守变量的命名规范，至于想取什么变量名，随客户开心就好。**
* 就像客户必须先说明房间类型，前台打开房间后才能入住一样，**变量也必须先声明后赋值。**
* **程序在找寻变量数值时是根据变量名来寻找的**，例如上述案例中前台并不关心room1房间里住的是谁，她只需要知道房间名一样。



在C++中，如何定义一个变量呢？

1. **变量的命名规则**

* 变量必须以字母，或下划线开头。
* 变量可以包括数字，但不能以数字开头
* 除了”\_” 符号以外，变量名不能包含任何特殊字符
* 不能使用C++语言的关键字，比如：int，class，public等
* 不含空格

另外，变量命名还必须**注意**：

Jim , jim123, jim\_123 √

123, ji#m123, 1\_m ×

* 变量必须先定义，后使用。
* 变量名是区分大小写的。例如intage和intAge是两个不同的变量。
* 变量名的命名最好要具有一定的含义，以便让阅读者做到见名知意。
* 建议变量名的长度最好不要超过8个字符。

1. **数据类型**

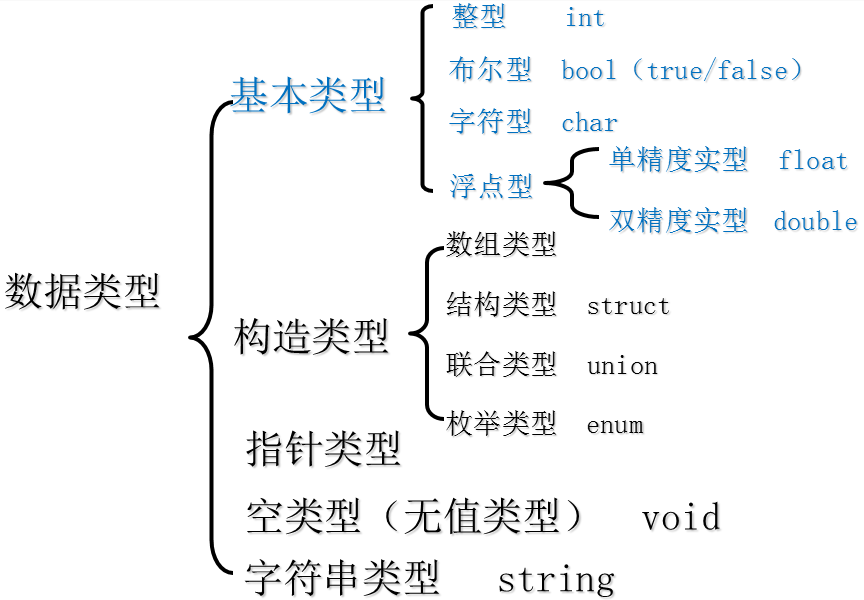
如同同一个旅馆却有多种不同房间类型一样，生活中这样的案例很多，例如同样是数字组成的数据，账号、密码和余额有什么区别？

答：账号、密码不能进行加减等算术运算，而余额可以。甚至对于同样由6位阿拉伯数字组成的密码和余额，我们的读法都不一样…

为什么？

因为在我们的大脑里已默默的赋予账号、密码和余额不同的类型。同样的道理，在计算机编程语言的世界里，每一个数据都有它的类型，具有相同类型的数据才能彼此操作，比如你不会对账号这种数据进行加减乘除等操作。

在C++中，有以下常见的数据类型：



**注：C++标准库定义了一种string类，定义在头文件<string>中，因此如果在代码中我们需要使用string，就必须先进行预编译，即#include <string> ，而string则表示是C++中的字符串，专门用来操作字符序列，并且所有的字符串类型数据必须用双引号“”来表示。**

如同不同的房间类型房间占地面积数也不同一样，不同的数据类型占的存储空间也不同，在C++语言中，要求对所有用到的变量作强制定义，也就是必须“先定义，后使用”，**定义变量的一般形式是：变量类型 变量名；**

如上述案例中的密码定义：

string password; //定义一个字符串类型的变量，变量名为password

然后我们给这个变量进行赋值，在C++中，**赋值符号为一个等号“=”，并且在赋值的过程中，是右边的值赋值给左边。**

password = “abc123”; // 给变量password赋值一个字符串abc123

**同样，我们也可以根据需求不同，只定义变量而不赋值。**

**同时，我们也可以将定义和赋值通过一句话直接实现。**

存款定义并赋值: float money = 100; //定义一个浮点型变量money，并赋值100

旅馆房间定义并赋值: string room1 = “M”;//房间1当前入住的是客人M

**并且，我们也可以一次性定义多个变量，定义格式：变量类型 变量名表列；**

变量名表列指的是一个或多个变量名的序列。如：

float a，b，c，d，e; // 定义a，b，c，d，e为单精度型变量。

1. **常量**

与变量相反，在生活中有一种值，一旦被创建便永远不会被改变，例如一个人的出生日期，总这个人出生开始，这个时间就永远不会发生改变。同样，C++中有一种标识符，它的值在运行期间恒定不变，专门用来存放程序运行过程中永不可变之值。我们把这样的数据称之为**常量**。

**常量定义格式：const (数据类型) 常量名 = 常量值；**

例：const int num = 7 ; // num是一个常量，在整个程序运行过程中都不能被改变。

### 5.4.2 输入与输出

C++并没有直接定义进行输入或输出(IO)的任何语句，这种功能是由标准库提提供的，IO库提供了大量的设施，然而对许多应用，包括本书的例子而言，编程者只需要了解一些基本概念和操作。本书的大多数例子都使用了处理格式化输入和输出的iostream库。

**标准库中处理输入时使用命名为cin(读作see-in)的istream类型对象，这个对象也称为标准输入。处理输出时使用命名为cout(读作see-out)的ostream类型对象，这个对象也称为标准输出。**一般情况下，系统将这些对象与执行程序的窗口联系起来，这样，当我们从cin读入时，数据从执行程序的窗口读入，当写到cout时，输出写至同一窗口。

下面我们来看一段代码。

程序首先在用户屏幕上显示提示语：

#include <iostream>

void main(){

std::cout << "输入两个数字:" << std::endl；

int num1，num2;

std::cin >> num1 >> num2；

std::cout << "两个数分别是：" << num1 << "和" << num2 <<std::endl；

std::cout << "它们的和是：" << num1+num2 << std::endl；

}

输入两个数字:

然后程序等待用户输入，如果用户输入：

1 3

接着敲回车键，则程序产生下面的输出：

两个数分别是：1和3

它们的和是：4

1. **输出流**

main函数体中第一条语句执行了一个表达式。该语句的表达式使用输出操作符在标准输出上输出提示语：std::cout << "输入两个数字:" << std::endl；

这个语句用了两次**输出操作符(<<操作符)**。每个输出操作符实例都接受两个操作数:左操作数必须是ostream对象(std::cout)；右操作数是要输出的值(需要输出的内容)。操作符将其右操作数写到作为其左操作数的ostream对象。

c++中，每个表达式都会产生一个结果，通常是将操作符作用到其操作数所产生的值。当操作符是输出操作符时，结果是左操作数的值，也就是说，输出操作返回的值是输出流本身。

既然输出操作符返回的是其左操作数， 那么我们就可以将输出请求链接在一起。输出提示语的那条语句等价于（std::cout << "输入两个数字:"） << std::endl；因此该部分等价于拆分成以下两段代码。

std::cout << "输入两个数字:"；

std::cout << std::endl；

**endl是一个特殊值，称为操纵符，具有输出换行的效果。**

1. **输入流**

在输出提示语后，先定义两个名为num1和num2的变量来保存输入，然后读取输入：std::cin >> num1 >> num2；**用户在实际输入两个数字时，中间用空格或回车键隔开。**

**输入操作符(>>操作符)**行为与输出操作符相似。它接受一个istream对象作为其左操作数(std::cin)；接受一个对象作为其右操作数(num1或num2)，它从istream操作数读取数据并保存到右操作数中。像输出操作符一样，输入操作符返回其左操作数作为结果。

由于输入操作符返回其左操作数，我们可以将输入请求序列合并成单个语句，换句话说，这个输入操作等价于：

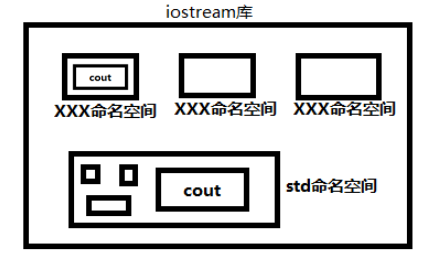
std::cin >> num1；

std::cin >> num2；

输入操作的效果是从标准输入读取两个值，将第一个存放在num1中，第二个存放在num2中。

1. **命名空间的使用**

相信细心的你们一定注意到这个程序中使用的是 std::cout和std::cin以及std::endl，而不是cout和endl。其实，之前我们在写第一个C++程序时就提到过，std是一个命名空间。现在我们回顾一下之前所画的关系图。



由上图可知，iostream文件库很大，也包含很多个cout，既然如此，那么同样在iostream库中的两个cout，我们怎么进行区分呢？这时命名空间的功能就体现出来了，前缀std::则表明cout和cin以及endl是定义在命名空间std中的。使用命名空问的目的就是在于避免与库中定义的同名关键字引起冲突。

不过标准库使用命名空间的副作用是：当我们使用标准库中的名字时，必须显式地表达出使用的是命名空间std下的名字。 std::cout的写法使用了作用域操作符(::操作符)，表示使用的是定义在命名空间std中的cout。

其实我们可以直接在代码编写开始就标注出来我们所使用的cout，cin以及endl均是来自于iostream文件内的std命名空间，**命名空间定义的格式为：using namespace 空间名称；**

最终修改代码如下：

#include <iostream>

using namespace std；

void main(){

cout << "输入两个数字:" << endl；

int num1，num2;

cin >> num1 >> num2；

cout << "两个数分别是：" << num1 << "和" << num2 <<endl；

cout << "它们的和是：" << num1+num2 << endl；

}

### 5.4.3 控制语句

控制语句用于控制程序的流程， 以实现程序的各种结构方式，它们由特定的语句定义符组成，可分成以下三类：(1)条件判断语句：[if语句](http://baike.baidu.com/subview/1338434/1338434.htm)、[switch](http://baike.baidu.com/subview/600161/18274660.htm)语句；(2)循环执行语句：do while语句、[while](http://baike.baidu.com/subview/1455003/18970859.htm)语句、for语句；(3)[转向语句](http://baike.baidu.com/subview/14014926/14589600.htm)：[break语句](http://baike.baidu.com/subview/1097051/1097051.htm)、continue语句、return语句、[goto语句](http://baike.baidu.com/subview/3209842/3209842.htm)（此语句如果滥用它会使程序流程无规律、可读性差，因此在本书中我们不做介绍）

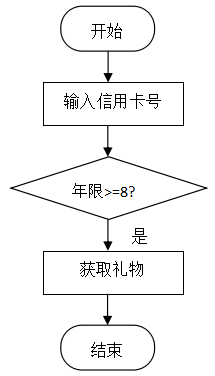
1. **条件判断语句**

* **if语句**

**场景1：**X银行在其官方站点提供了活动页面，持卡人如果持卡已满8年，便可以获得“越久越尊贵”礼品资格。

**问题一：**如何将业务处理的过程用流程图表示出来？

**流程图：**



**问题二：**如何将流程图中的逻辑用程序代码表示？

**分析：**假设持卡年限使用变量year表示，那么year>=8表示持卡人拥有了获取礼物的资格，这样就能得到招行赠与的礼物。

**特点：**i.有条件：持卡年限大于等于8年（year>=8） ii.只考虑一种情况

**综上**：使用if分支语句当中的单分支。

**语法：if单分支语句语法**

|  |
| --- |
| if (条件A) {  当满足条件A（即A为 true 时）执行的代码;  } |

注：if表示假如，即一个判断的开始;且if中的条件A只有true和false两种可能，因为是判断语句，if只识别真或假，即true/false（又可用1/0表示），而不识别其他结果。

**代码示例：**

**场景2：**X银行的持卡人由于使用年限的不同，兑换美的电饭煲使用的积分数量不同。

#include <iostream>

using namespace std;

void main(){

int year; // 用户持卡的年份

cout << “请输入持卡年份：” << endl;

cin >> year; // 将用户输入的数字存入year变量中

if(year >= 8){

cout << “恭喜您，拥有获取礼品资格!” << endl;

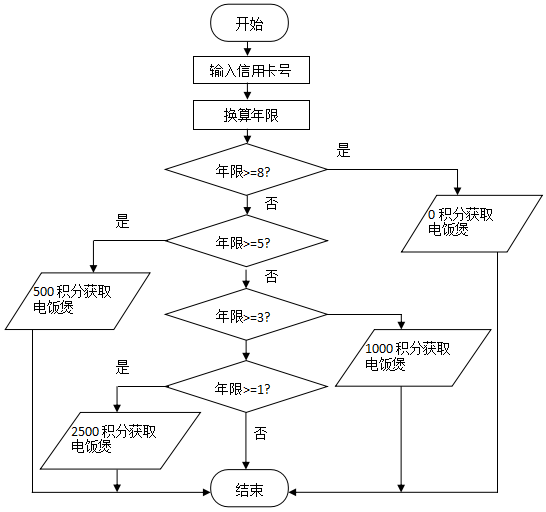
}

}

这时我们可以通过if多分支语句实现。

**问题一：**如何把X银行的工作人员完成上述业务过程用流程图表示出来？

**流程图：**



**问题二：**如何将流程图中的逻辑用程序代码表示？

**分析：**假设持卡人的使用年限值使用变量year表示，那么1<=year<3，2500积分换购美的电饭煲；3<=year<5，1000积分换购美的电饭煲；5<=year<8，500积分换购美的电饭煲；year>=8，0积分换购美的电饭煲。

**特点：**i.有多个条件 ii.考虑了多种情况

**综上：**使用分支语句当中的多分支

**语法：if多分支语句语法**

|  |
| --- |
| if (条件1) {  条件1为 true 时执行的代码;  } else if (条件2) {  条件1为false，但条件2为 true 时执行的代码;  } else if (条件3) {  条件1和条件2均为false，但条件3为 true 时执行的代码;  }  ……  else {  上述条件1、2、3……均为false的情况下所执行的代码;  } |

注：多分支判断语句时，顺序是置顶而下判断的，例如，判断条件2 时，必须是先判断条件1，如果条件1为false，按照顺序，再判断条件2，如果条件1为true，则直接运行条件1下的代码，运行结束后，则直接跳出判断部分，不再执行条件2的判断。

**代码示例：**

#include <iostream>

using namespace std;

void main(){

int year; // 用户持卡的年份

cout << “请输入持卡年份：” << endl;

cin >> year; // 将用户输入的数字存入year变量中

if(year >= 8){

cout << “0积分换购美的电饭煲” << endl;

}else if(year>=5){

cout << “500积分换购美的电饭煲” << endl;

}else if(year>=3){

cout << “1000积分换购美的电饭煲” << endl;

}else if(year>=1){

cout << “2500积分换购美的电饭煲” << endl;

}else{

cout << “对不起，您未达到活动条件” << endl;

}

}

* **switch..case语句**

**场景3：**X银行根据用户持卡年限的不同，把用户等级进行划分，并根据不同用户级别节日发放红包，红包金额如下：

普通用户，刷卡金100元；

银牌用户，刷卡金500元；

金牌用户，刷卡金1000元；

钻石用户，刷卡金2000元。

**分析：**在编程的世界里，用户等级通常使用数字来表示，例如：1表示普通用户，2表示银牌用户，3表示金牌用户，4表示钻石用户。使用grade来表示级别，那么上面的业务过程表示为：

grade==1：发放刷卡金100元

grade==2：发放刷卡金500元

grade==3：发放刷卡金1000元

grade ==4：发放刷卡金2000元

**特点：**

i.也属于分支结构

ii.全部是等值比较

**综上：**C++中对于完全等值比较的分支结构通常使用switch…case结构。

**语法：switch语句语法**

|  |
| --- |
| switch (表达式)  {  case 常量表达式1:  满足常量表达式1时执行的语句;  case常量表达式2:  满足常量表达式2时执行的语句;  ……  default:  所有常量表达式均不满足时执行的语句;  } |

**代码示例：**

注：1、switch语句和if语句不同，它会执行每一个case语句，例如判断case 1语句成立后，程序将执行case 1以内的语句，执行完后再次判断case 2，以此类推，因此，为避免不必要的重复浪费时间，我们通常会在每一个case语句结束后添加一个break语句，这样在执行完当前case语句后，就会跳出判断，不再执行其他case语句。2、default可有可无，但是为了程序的严谨性，一般加上default。

#include <iostream>

using namespace std;

void main(){

int grade; // 用户持卡的级别

cout << “请输入持卡级别：” << endl;

cin >> grade;

switch(grade){

case 1:

cout << “普通用户，发放刷卡金100元” <<endl;

break;

case 2:

cout << “普通用户，发放刷卡金500元” <<endl;

break;

case 3:

cout << “普通用户，发放刷卡金1000元” <<endl;

break;

case 4:

cout << “普通用户，发放刷卡金2000元” <<endl;

break;

default:

cout << “对不起，输入有误” <<endl;

break;

}

}

1. **循环执行语句**

* **for语句**

**场景4：**要求控制台输出100遍“Hello world!”。

**分析：**对于具有规律性的重复操作，在程序中可通过重复执行某些语句来实现。

**特点：**根据循环条件是否满足要求，决定是否重复执行同样的事情

**语法：for语句语法**

|  |
| --- |
| for (表达式1； 表达式2； 表达式3) {  循环体；  } |

**参数：**表达式1：循环变量的初始化（初始值）

表达式2：循环条件（终止值）

表达式3：循环变量的变化

**for语句的执行过程：**

1、计算表达式1；

2、计算表达式2；如果表达式2条件成立，即循环条件成立，就执行一次循环体；

3、计算表达式3；为下一次判断循环条件是否成立做准备，到此完成一次循环。

4、第一次循环结束以后，每次都是从计算表达式2开始，进入下一次循环，直到表达式2不成立时结束循环。

**代码示例：**

#include <iostream>

using namespace std;

void main(){

for(int i = 0; i < 100; i++){

cout << “hello world” << endl;

}

}

* **while语句**

**场景4：**要求控制台输出100遍“Hello world!”，要求用另一种语法来实现。

**语法：while语句语法**

|  |
| --- |
| while (循环条件A) {  当循环条件A为true时要执行的代码;  } |

注：while语句中，首先判断循环条件A是否成立，如果成立则执行循环体，并且执行完后再次判断循环条件A，如此反复，直到条件A不成立为止，结束循环。

**代码示例：**

* **do...while语句**

#include <iostream>

using namespace std;

void main(){

int i = 0;

while(i < 100){

cout << “hello world” << endl;

i++;

}

}

**场景4：**要求控制台输出100遍“Hello world!”，要求再换另一种语法来实现。

**语法：do...while语句语法**

|  |
| --- |
| do{  第一次无条件执行一次循环体，  后期当循环条件A为true时执行;  }while (循环条件A) |

注：do..while语句中，首先无条件执行一次循环体，然后再判断循环条件A是否成立，如果成立则再次执行循环体，如此反复，直到条件A不成立为止，结束循环。

**代码示例：**

**While和do...while区别：**while是先判断，再执行；do...while是先执行，再判断。

#include <iostream>

using namespace std;

void main(){

int i = 0;

do{

cout << “hello world” << endl;

i++;

}while(i < 100)

}

1. **转向语句**

* **break语句**

**场景4：**要求控制台在输出100遍“Hello world!”的循环过程中，在输出68个后停止所有输出。

**特点：**在条件明确的情况下终止整个循环。

**语法：**break关键字一般用于控制语句当中，表示直接终止当前循环。

**代码示例：**

* **continue语句**

#include <iostream>

using namespace std;

void main(){

for(int i = 1; i <= 100; i++){

if(i == 68){ break; }

cout << i << “循环:” << endl;

cout << “hello world” << endl;

}

}

**场景4：**要求控制台在输出100遍“Hello world!”的循环过程中，在输出第68时暂停一次，然后从第69个开始继续输出。

**特点：**在条件明确的情况下暂停当前循环，并重新进行下次循环。

**语法：**continue关键字一般用于控制语句当中，表示中断当前循环，直接继续下次迭代。

**代码示例：**

### 5.4.4 一维数组

#include <iostream>

using namespace std;

void main(){

for(int i = 1; i <= 100; i++){

if(i == 68){ continue; }

cout << i << “循环:” << endl;

cout << “hello world” << endl;

}

}

如果一个班级有10个学生，进行了一次考试，我们需要获取每个学生的成绩，如何处理？在C++中需要在一个变量中存储多个值，解决方法是创建数组。**数组是构造数据类型的一种，是用于描述类型相同且彼此有某种联系的一批数据的数据结构。**

1. **一维数组的定义**

一维数组用于描述一行或者一列数据。

定义一维数组的一般格式为：**类型标识符 数组名[常量表达式];**

例如：int stu\_scores[10];

它表示数组名为stu\_scores，此数组存放的数据为int型，有10个元素。

关于一维数组的**几点说明**：

* 数组名命名规则和变量名相同，遵循标识符定名规则。
* 用方括号括起来的常量表达式表示下标值，如下面的写法都是合法的：

int stu\_scores[10];

int stu\_scores[2\*5];

int stu\_scores[n\*2]; // 假设前面已定义了n为常变量

* 常量表达式的值表示元素的个数，即数组长度。数组长度一旦确定，就不允许改动。

例如，在“int stu\_scores[10];”中，10表示stu\_scores数组有10个元素，下标从0开始，这10个元素是：stu\_scores[0]，stu\_scores[1]，stu\_scores[2]，stu\_scores[3]，stu\_scores[4]，stu\_scores[5]，stu\_scores[6]，stu\_scores[7]，stu\_scores[8]，stu\_scores[9]。注意最后一个元素是stu\_scores[9]而不是stu\_scores[10]。

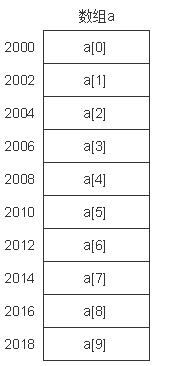
* 常量表达式中可以包括常量、常变量和符号常量，但不能包含变量。也就是说，C++数组的大小一般情况下不允许是变量。例如，下面这样定义数组是错误的：

int n;

cin>>n; // 输入stu\_scores数组的长度

int stu\_scores[n]; // 错误！企图根据n的值决定数组的长度

* 数组是有类型属性的。同一数组中的每一个元素都必须属于同一数据类型。一个数组在内存中占一片连续的存储单元。如果有一个整型数组a，假设数组的起始地址为2000，则该数组在内存中的存储情况如图所示。



1. **一维数组的初始化**

* 在定义数组时分别对数组元素赋予初值。例如：
* int arr[10]={0，1，2，3，4，5，6，7，8，9};
* 可以只给一部分元素赋值。例如：
* int arr[10]={0，1，2，3，4};
* 如果想使一个数组中全部元素值为1，可以写成：
* int arr[10]={1，1，1，1，1，1，1，1，1，1};
* **不能写成：int arr[10]={1\*10}; //** 不能给数组整体赋初值。
* 在对全部数组元素赋初值时，可以不指定数组长度。例如：
* int arr[5]={1，2，3，4，5};

可以写成

* int arr[]={1，2，3，4，5};

1. **一维数组的长度**

如果已知某数组的名字和类型，如何查知其元素的数量呢？

在C++中可以使用sizeof()方法求数组的长度：**sizeof(数组名) / sizeof(数组类型)**

int score[]={78，90，77，60，70};

int len = sizeof(score)/sizeof(int); // len = 5

1. **一维数组的遍历**

班上的学生成绩都储存在数组score中了，如何得到班级所有学生的成绩呢？得到所有学生的成绩也就是要求遍历数组score。访问数组元素的方式是“数组名+下标”，如score[0]、score[1]、score[2]、score[3]、score[4]，其中，只有下标在有规律的变化。因此，可以将下标作为控制变量，使用 for循环遍历并输出索引数组的所有值。

**代码示例：**

#include <iostream>

using namespace std;

void main(){

int score[] = {78, 90, 77,60, 70};

int len = sizeof(score)/sizeof(int); // 求出数组长度

for(int i = 0; i < len; i++){

cout << “当前输出的值是:” << score[i] << endl;

}

}

1. **一维数组的其他操作**

班上的学生成绩都储存在数组score中了，这时如果想要查找一个数据，或者更改一个数据时，该如何操作呢？

**代码示例：**

### 5.4.5 函数

#include <iostream>

using namespace std;

void main(){

int score[] = {78, 90, 77,60, 70};

int len = sizeof(score)/sizeof(int);

cout << “该数组中一共有” <<len<<”个数字,”;

cout << “请输入你要查找的数的索引(索引从0开始):”;

int num = 0;

cin >> num;

if(num < 0 || num >= len){

cout << “对不起，您的序号有误!” <<endl;

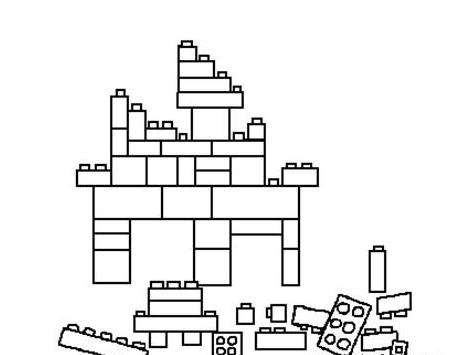
}else{

cout << “您要查找的数字是:” << score[num] << endl;

}

}

程序将变得越来越大，越来越复杂，程序中部分功能可能会反复用到，并且程序中的某些功能一旦写好，就不需要改变，及时在修改整个程序时也希望不要被干扰。这如同需要建造一个城堡，而给我们的却全是一些原始木材一样，复杂度和困难度都是可想而知的。那么如果这样呢？



是的，**积木**，如果原材料已经被加工成各式各样的积木，那“建造城堡”就会显得相当的简单，哪怕是小朋友也可以轻松的搞定了，甚至同一块积木在不同的地方都可以发挥作用。如同积木在系统中相对独立一样，我们将单独某一功能的代码封装起来，也就是将这部分功能从系统中“分离”出来，做成一个“积木”——**函数**。

C++语言对于“积木”的制作，有些严格的规定，只有满足这些语法规定的“积木”，才可以使用。

1. **函数的声明与定义**

C++语言对封装函数时，要对函数进行声明与定义。

封装一个函数，首先，我们要告诉电脑，我们自己编写了一个函数，即这个函数是存在的，这叫作函数的声明（Declaration）。其次，我们要告诉电脑这个函数是怎么运作的，这叫作函数的定义（Definition）。

* **函数的声明的格式**

|  |
| --- |
| **返回值类型 函数名(形式参数列表)；** |

函数声明如同变量声明一样，是一条语句，所以在语句结束后要加上分号，函数名的命名规则与变量名的命名规则一致。

例如： void Sum(int a，int b)；

在上述函数声明中，**Sum**是函数名，也就是这一块“积木”的名字，**void**表示函数返回值为空，**Sum**后面的括号里是填写参数表的地方，整个函数一共有两个参数，并且两个参数都是int类型，分别名为a和b，关于函数返回值，参数，在后面会进一步讲解。

* **函数定义的格式**

|  |
| --- |
| **返回值类型 函数名(形式参数列表) {**  **//这里编写函数的主体**  **}** |

函数定义的格式和声明格式基本类似，仅仅是将声明中的分号改成大括号，并且在大括号内编写需要实现的代码即可，例如：

这个求和函数Sum函数的功能就编写完成了，那么怎么去使用这个函数？

void Sum(int a，int b){

int c = a + b；

cout << c << endl；

}

1. **函数调用**

首先我们来看一段代码：

void main(){

Sum(1，2)；

Sum(2，4)；

Sum(3，6)；

}

当用户运行程序时，运行结果为：

3

6

9

从中我们可以看出函数调用的一般形式：**函数名（实际参数）；**

通过上述调用函数的代码我们发现**函数以下几点特点**：

* 函数调用可以重复多次，并可根据功能需求的不同任意在系统中其他地方调用。
* 函数可以传递不同的实际参数，但参数的个数和类型必须和函数定义时保持一致。
* C++语言的函数调用遵循先定义、后调用的原则。如果对某函数的调用出现在该函数的定义之前，必须用说明语句对函数进行声明。
* 一般把调用者称为主函数，把被调用者称为被调函数。在上述案例中，由main函数调用Sum函数，那么调用者main函数称为主调函数，被调用者Sum函数称为被调函数。

1. **函数的参数**

从参数角度分析，函数可分为有参函数和无参函数，顾名思义，有参函数即为有参数的函数，无参函数为没有参数的函数。上述求和函数就是一个典型的有参函数。

回顾之前我们所编写的Sum函数的声明、定义与调用：

Sum函数的声明： void Sum(int a, int b);

Sum函数的定义： void Sum(int a, int b){ ........... }

Sum函数的调用： Sum(1, 2);

首先在函数的声明与定义中我们可以发现均出现了int a, int b这类参数的写法，并且我们发现无论是参数a还是参数b，都仅仅是知道他们的数据类型和名字，却并不知道他们具体的数值是多少，我们把这样写在函数里的只能知道个数和类型，却不知道具体数值的参数称之为**形式参数，简称形参**。

其次在函数的调用中我们发现函数小括号内不再写的是a和b，而是1和2,这样具体的数值，我们把这样的参数称之为**实际参数，简称实参**。

**关于函数的形参和实参，有几点值得注意：**

* 在定义函数中指定的形参在未出现函数调用时，它们并不占用内存中的存储单元。只有在发生函数调用时，形参才被分配到内存单元。在调用结束后，形参所占用的内存单元也被释放。因此，形参只在函数内部有效，函数调用结束不能再使用该形参变量。
* 实参可以使常量、变量或者表达式，但要求它们必须有确定的值，在调用时按顺序将实参的值赋给形参，即1赋值给a，2赋值给b。
* 在被定义的函数中，必须指定形参的个数以及类型，每个形参之间以逗号隔开。
* 形参和实参在数量、类型、顺序上应该严格保持一致。
* c语言规定，实参变量对形参变量的数据传递是“值传递”，即单向传递，只由实参传给形参，而不能由形参传回来给实参，C++延续这个规定。
* 在调用函数时，给形参分配存储单元并将实参对应的值传递给形参（实际上是对实参的拷贝），调用结束后，形参单元被释放，实参单元容保留并维持原值。因此，在执行一个被调用函数时，形参的值如果发生改变，并不会改变主调函数实参的值。

1. **函数的返回值**

理解函数的声明实现调用之后，我们现在来思考一个问题，在之前的Sum函数中，它的功能是将函数调用时传入的具体数值进行求和并输出，可是往往在真实项目中，我们所需要的功能不仅仅是将结果输出，而是利用计算结果再次进行其他功能，例如现在需要用程序来进行求和运算：1+2+3=？怎么办？

通过分析，我们可以调用两次Sum函数：第一次是Sum(1，2)；求出1+2的和；第二次Sum(【上一次1+2的和】，3)；求出最终结果。

这时我们发现，我们需要获取到Sum(1，2)的计算结果，而不是仅仅输出功能，换句话说，**需要函数在计算结束之后返回计算结果，我们把这样的数值称之为函数的返回值**。

回顾之前函数的定义格式部分：

|  |
| --- |
| **返回值类型 函数名(形式参数列表) {**  **//这里编写函数的主体**  **}** |

按照返回值的角度来分，函数可分为有返回值的函数和返回值为空的函数，刚刚我们实现的Sum函数就是一个典型的返回值为空的函数，即返回值类型为void，那么有返回值的函数怎么修改呢？

首先我们来看下面一段代码：

从上述代码中可以看出，返回值类型部分不再是void而是变成了int，这就表明该函数返回的是一个int类型的结果数据，而在newSum函数定义中最后一行看到return c，这就表明整个函数返回的计算结果就是变量c所存的数值。

newSum函数的声明： int newSum(int a, int b);

newSum函数的定义：

int newSum(int a, int b){

int c = a + b;

return c;

}

现在我们再来实现一下1+2+3这道题。

首先我们获得1+2的结果，即调用newSum函数，调用后我们需要声明一个变量来接收函数返回的值，由于函数返回的是一个int值，因此接受它的变量也应该是int类型。

int result1 = newSum(1,2)；// 通过函数求出1+2的和，然后将结果返回给result1

然后我们将已获得的result1再次作为求和的第一个数进行再次进行计算，并将结果保存起来。

int result2 = newSum(result1,3)；// 求出rsult1+3的结果并返回至result2中

最后，只需要输出result2即是所求的1+2+3的结果。

**关于函数的返回值，有几点值得注意：**

* return 表示从被调函数返回到主调函数继续执行，返回时有且只能有一个返回值。
* 对于返回值类型为void的函数而言，可以没有return语句，但任何一个返回类型不是void的函数必须返回一个值，而且返回值类型需与函数声明以及函数定义时的函数返回类型一致。
* 函数体内可以编写多个return语句，然后有效的可执行的return语句只有一个，当程序运行时遇到第一个return语句时，就会直接结束当前函数并返回结果。

### 5.4.6 指针与引用

什么是指针？假设你要我借给你一本书，我拿着书到了你宿舍，但是你人正巧不在，于是我把书放在你的2层3号书架上，并写了一张纸条放在你的桌上。纸条上写着：你要的书在第2层3号书架上。当你回来时，看到这张纸条就知道书的位置。

想想看这张纸条的作用，纸条本身不是书，它上面也没有放着书。那么你又如何知道书的位置呢？因为纸条上写着书的位置嘛！其实这张纸条就是一个指针了。它上面的内容不是书本身，而是书的地址，你通过纸条这个指针找到了我借给你的本书。

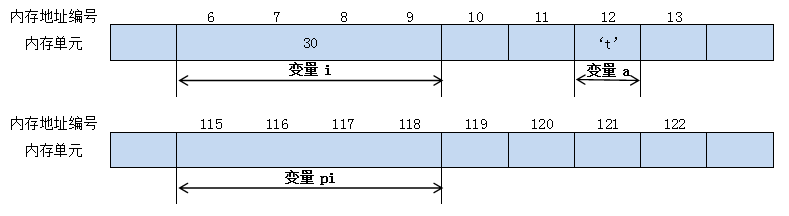
1. **指针的概念与声明**

那么C++ 中的指针又是什么呢？我们来看一个C++中整型数据指针的声明：

|  |
| --- |
| int \* pi； |

**pi是一个整形变量的指针**，但是你以为pi一定是个多么特别的东西吗？其实，pi也只过是一个变量而已，与之前所说的变量并没有实质的区别。

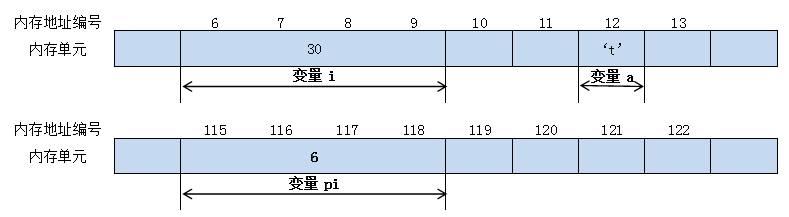
不信你看下图所表示的变量在内存中的存储：int i = 30； char a = ‘t’； int \*pi；



由图中可以看出，我们使用int \*pi声明指针变量，其实就是在内存的某处声明一个一定宽度的内存空间，并把它命名为pi，你能在图中看出pi与前面的i，a 变量有什么本质区别吗？pi也只不过是一个变量而已嘛！那么它又为什么会被称为指针？关键是我们要让这个变量所存储的内容是什么，现在我要让pi成为真正有意义上的指针。

|  |
| --- |
| pi = &i； |

上述代码中，**&是取地址符**，**&i是返回i变量的地址编号**。整句代码的意思就是把i地址的编号赋值给pi，也就是你在pi上写上i的地址编号。结果如下图所示：



你看，执行完pi=&i后，在图示的系统中pi的值是6。这个6就是i变量的地址编号，这样pi就指向变量i了。现在想想pi与那张纸条有什么关系？很明显pi就是那张纸条嘛！上面写着i的地址，而i就是那个本书。你现在看懂了吗？

因此，我们把pi称为指针。**指针变量所存的内容就是内存的地址编号**！但是内存单元是有限定的，pi是一个整型指针，因此pi所指向的内存单元必须也保存的是一个整形数据。

最后只需要通过指针pi来访问到i这个变量即可。

|  |
| --- |
| cout<<\*pi<<endl; |

那么此处\*pi什么意思呢？你只要这样读它：**pi内容所指的地址的内容**，就pi这张“ 纸条”上所写的位置上的那本“书”---变量i。pi所存的数据是为6的地址编号，也就是说pi指向内存编号为6的地址。\*pi就是它所指地址的内容，即地址编号6上的内容了，当然就是30这个值了，所以这条语句会输出30。

换句话说

|  |
| --- |
| cout<<\*pi<<endl; 完全等价于 cout<<i<<endl; |

从上面的例子也能看出，与指针有关的“\*”有两种，**第一种“\*”是定义一个变量为指针变量，仅仅出现在定义语句中；第二种“\*”是访问指针所指向内存空间的值。**那么怎么区别当前的\*号是第一种还是第二种呢？很简单，只需要观察\*号前面是否存在数据类型就可以啦！例如 char \*x；这就是指针变量；\*x这就是取x所指向的值。

**思考题：你能直接看出输出的结果是什么吗？**

|  |
| --- |
| int a = 10；  int \*pa=&a；  \*pa=20；  cout<<a<<endl； |

1. **引用的概念及使用**

什么是引用？假设302班中有一名同学名叫张三，而恰好他也是这个班的班长，并且由于张三从小就弹得一手好钢琴，因此班级的人又都称他为“钢琴王子”。某一天放学302教室门口围着一群女生正在叽叽喳喳，其中有一名女生就问道：“听说这个班有一位钢琴王子，不知道是谁呢？”，这时坐在教室第一排的同学听到后便回头大喊：“班长！有女生打听你呢！”这时，班上所有的同学都坏笑的看向张三同学。

好了，故事到这就结束了，大家仔细想想上述案例，明明是同一个人，却有三种名字，虽然三个名字均不相同，但所有人都可以准确的找到该名同学，这是为什么？其实我们可以将“班长”和“钢琴王子”理解为“张三”同学的别名，所以无论名字有多少，但所指向的却都是同一个人。而起别名这种行为，就是我们C++程序中所说的引用。

下面我们就来看看C++是如何“起别名”的？

从语句1和语句2中我们可以看出分别给zhangsan所存的值起了两个别名，而在新的别名前面仅仅是添加了一个&符号，可以可能有同学会问，&不是取地址吗？怎么又变成起别名了呢？

string zhangsan = “张三”;

string &banzhang = zhangsan; // 语句1

string &qingewangzi = zhangsan; // 语句2

cout << zhangsan << endl; // 输出结果为：张三

cout << banzhang << endl; // 输出结果为：张三

cout << qingewangzi << endl; // 输出结果为：张三

其实“&”有两种含义，**第一种“&”是取地址符；第二种“&”表示引用，即起别名。**那么怎么区别当前的&号是第一种还是第二种呢？很简单，只需要观察&号前面是否存在数据类型就可以啦！例如 char &x；这就是引用；而&x就是取x的地址。

**思考题：****该段代码的输出结果是多少以及原因？**

|  |
| --- |
| int a = 10；  int &b=a；  b = 20；  cout<<a<<endl; |

**终极思考魔王题：该段代码的输出结果是多少以及原因？**

int a = 10；

int \*pa = &a；

int &b = a；

int \*pb = b；

a = 20；

cout<< a <<endl；

cout<< b <<endl；

cout<< \*pa <<endl；

cout<< \*pb <<endl；

# 第六章 类与对象的实现

## 6.1 综合实例1---LostCraft游戏

### 6.1.1 类的构成

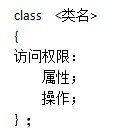
在设计篇中我们针对LostCraft游戏案例进行了分析，识别出了无数个对象，在这些对象的基础上进行了抽象，构建了四个类：英雄类、技能类、怪物类、场景类，并针对这四个类进行了类图设计。

然后通过类图的设计，我们知道：一个类有**属性**和**操作**两个部分组成。

* 具有相同特性（属性）和行为（操作）的[对象](http://baike.baidu.com/view/2387.htm)的[抽象](http://baike.baidu.com/view/5293.htm)就是类；
* 类具有属性，它是[对象](http://baike.baidu.com/view/2387.htm)状态的[抽象](http://baike.baidu.com/view/5293.htm)，用[数据结构](http://baike.baidu.com/view/9900.htm)（普通变量）来描述类的属性。
* 类具有操作，它是[对象](http://baike.baidu.com/view/2387.htm)行为的[抽象](http://baike.baidu.com/view/5293.htm)，用操作名和实现该操作的方法（函数）来描述。

在实际项目设计过程中，根据项目的具体需求来设计一个类，因此允许所设计的类不包含属性或者操作，甚至可以两者都不包含，仅仅是一个空类！

**定义类的一般格式如下图所示：**



* **class是定义类的关键字，表示现在要定义一个类。**
* **<类名>**为一个合法的标识符，**是程序设计者为所定义类的命名，该类的命名规范遵循变量的命名规范**；
* **一对大括号则表示类的边界，**边界内是类的说明部分，称为**类体**；
* **访问权限：**用于标出以下使用的属性和方法的相关权限；
* 在类体的后面有**一个分号**，表示类定义的结束。

### 6.1.2 类的声明与实现

下面我们就分别来定义英雄、技能、怪物和场景这四个类吧！

**英雄类**：

|  |  |  |
| --- | --- | --- |
| 1 |  | class Hero{  public：  string name；  int level；  int HP；  int MP； Craftsmanship crafts[NUM\_CRAFTS]；  void initHero()；  int choice()；  void attack(int，Monster&)；  bool isAlive()；  void levelUp(int)；  void levelDown()；  private：  void settlement(int level)；  public：  Hero(void)；  ~Hero(void)；  }; |

* class表示定义了一个类；
* Hero表示类名，该名称由用户自己命名，类名建议使用英文字母，而不要使用中文或者任何带有特殊字符的类名，会引起不必要的字体乱码的错误；
* {XXX}；大括号内所有的内容XXX均表示该类的说明部分，而分号“；”表示整个类定义完成；
* **public****:是访问权限的一种，表示公共的；private:也是访问权限的一种，表示私有的；**访问权限一共有三种，该部分内容我们会在后几章进行介绍，现在大家可以暂时全部使public权限。
* **NUM\_CRAFTS是一个常量**，const int NUM\_CRAFTS = 8；
* **Hero(void)；是声明一个构造函数，**该函数较为特别，该函数的名字与类同名，且函数本身没有返回值，针对该函数将会在第七章进行介绍，在此大家只需要按照书中的样式编写出来即可。
* **~Hero(void)；是声明一个析构函数，**该函数也较为特别，该函数的名字与类同名，且函数名前一个小波浪号，函数本身没有返回值，同样，该函数将会在第七章进行介绍，在此大家只需要按照书中的样式编写出来即可。

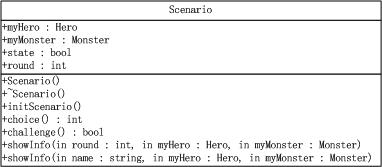
**技能类：**

|  |  |  |
| --- | --- | --- |
| C:\Users\Administrator\Desktop\类图\QQ图片20160701160255.png |  | class Craftsmanship  {  public：  string name；  int consumption；  int ATK；  int cooling；  int available；  int action；  Craftsmanship(void)；  ~Craftsmanship(void)；  }； |

**怪物类：**

|  |  |  |
| --- | --- | --- |
| 2 |  | class Monster  {  public:  string name；  int level；  int HP；  int ATK；  void initMonster(int)；  void attack(Hero&)；  bool isAlive()；  Monster(void)；  ~Monster(void)；  }； |

**技能类：**



|  |
| --- |
| class Scenario  {  public:  Hero myHero；  Monster myMonster；  bool state；  int round；  void initScenario()；  int choice()；  bool challenge()；  void showInfo(int, Hero, Monster)；  void showInfo(string, Hero, Monster)；  Scenario(void)；  ~Scenario(void)；  }； |

### 6.1.3 对象实例化

### 6.1.4 对象间的协作实现

## 6.2 综合实例2---YYY系统项目

### 6.2.1 类的实现

### 6.2.2 对象实例化之new关键字

### 6.2.3 对象资源回收之delete关键字

### 6.2.4 对象间的协作实现

## 6.3 综合实例3---ZZZ系统项目

### 6.3.1 类的实现

### 6.3.2 this指针

### 6.3.3 对象实例化及对象间协作实现

## 6.4 类与对象概念的强化

### 6.4.1 类与对象的概念

### 6.4.2 类与对象的关系

# 第七章 面向对象基础

## 7.1 综合实例1---LostCraft游戏

# 第八章 继承

## 8.1 综合实例1---LostCraft游戏

编写LostCraft程序，在很大程度上是为了描述和解决游戏世界中的现实问题。C++中的类很好的采取了人类思维中的抽象和分类的方法，**类与对象的关系恰当的反映了个体与同类群体共同特征之间的关系。**进一步观察现实世界可以看到，不同事物之间往往不是独立的，很多事物之间都有着复杂的联系。**继承**便是众多联系中的一种。

### 8.1.1 继承的概念

继承就是将公共的东西提取出来，封装成一个类，我们称之为**父类或基类**，其他类继承此类，获得与此类相同的特性，此外还可以在此基础之上，编写其他属性和方法，这些类我们称为**子类或者派生类**。

1. 继承是对自然界中继承关系的一个**抽象**；
2. 继承可以分为**单继承、多继承和公有继承**，其中单继承是继承中较常见的一种，公有继承是默认的继承方式；
3. 继承中我们把被继承者称为**父类或基类**，继承者称为**子类或派生类**；
4. 实现继承时，我们应先抽取出父类和子类，然后借助C++实现继承；

比如，汽车有小汽车和卡车等。汽车和火车、飞机又构成了交通工具这一大类。我们可以将交通工具抽象为**父类**，然后火车、汽车和飞机等**继承**交通工具类。并且还可以继续由飞机**派生**出波音飞机，由汽车**派生**出小汽车和卡车等等。这些派生出的类又称之为**子类**。

图 8-1单继承关系

面向对象的程序设计中提供了类的继承机制，允许程序员在保持原有类特性的基础上，进行更具体、更详细的类的定义。以原有的类为基础产生新的类，我们就说新类继承了原有类的特征，也可以说是从原有类派生出新类。类的派生机制有什么好处？**好处在于代码的重用性和可扩充性。**通过继承可以**充分利用别人做过的一些类似的研究和已经有的一些分析、解决方案。**派生新类的过程一般包括**吸收已有类的成员、调整已有类的成员和添加新的成员三个步骤。**

在LostCraft游戏中，之前我们定义了Hero英雄类和Monster怪物类。我们可以通过比较发现在这两个类中，存在一部分同样的属性。为了后续程序便于重构和可扩展，我们在此为这两个类声明一个共同的父类Role角色类。构成如下图所示的一种继承关系：

从该图中我们可以看出，角色和英雄、怪兽构成了这样的一个层次结构。这可以说是人对自然界中的事物进行分类、分析和认知过程在程序中的体现。在游戏中英雄和怪物是相互联系、相互作用的，根据他们的实际特性，抓住其共同特性和细小差别，利用分类的方法进行分析和描述。在这个层次结构中，由上到下，是一个具体化、特殊化的过程；由下到上，是一个抽象化的过程。上下层之间的关系就可以看作是基类与派生类的关系。

图 8-2继承关系



### 单继承的声明与实现

下面我们就分别来重新定义角色、英雄和怪物类吧！

|  |  |
| --- | --- |
| 图 8-3角色类 | class Role  {  public:  Role(void);  ~Role(void);  string name;  int level;  int HP;  bool isAlive();  }; |

将Hero类和Monster类中的共有变量和函数提取出来，包括：名称、级别、生命值和是否存活函数。

|  |
| --- |
| #include "Role.h"  …… // 此处省略  bool Role::isAlive()  {  return HP>0;  } |

如何实现Role类呢？我们可以从上边的定义可以看出，Role类中只包含一个“是否存活”的成员函数，因此在Role.cpp中除了默认构造函数和析构函数以外，我们来实现这个成员函数。通过前几章的分析，我们知道如何来判断英雄或者怪物存活的方法，因此在Role类中，也是同样的方法来实现该函数：

|  |  |
| --- | --- |
| 图 8-4英雄类 | #include "role.h"  #include "Craftsmanship.h"  class Monster;  **class Hero :**  **public Role**  {  public:  Hero(void);  ~Hero(void);  int MP;  Craftsmanship crafts[NUM\_CRAFTS];  void setName();  void choice();  void attack(Monster&);  void levelUp(int);  void levelDown();  private:  int iAttack;  void settlement(int level);  }; |

**class Hero : public Role** 表明英雄类是从角色类继承而来，或者说角色类派生出英雄类。注意这里表示继承关系的符号用英文的冒号（:）。

由于有了继承关系，这里的共有成员变量并未在Hero类里边声明。比如名称，级别等。这些信息都由继承关系使得Hero类从Role类中继承了进来。同样的，共有的是否存活成员函数也被继承了进来。所以这些共有的成员均不用在Hero类中再次声明。这样一来，就可以在复杂的项目中能够轻松的重构代码，更好更快的完成项目。

另外，在Hero类中也有一些和怪物不一样的细小差别。比如Hero类具有能量值来完成技能攻击。Hero类当然也具有若干技能来发挥更高的攻击力。此外Hero类也包含了一些和怪物不一样的功能（成员函数）。因此，在继承关系中，派生类独有的成员变量和成员函数都需要独立声明。

如何实现派生出来的Hero类呢？通过分析我们可以发现Role类中的所有公有成员我们已经继承得到，只需要再此实现上述声明的成员函数，代码如下：

|  |
| --- |
| #include "Hero.h"  #include "Monster.h"  Hero::Hero(void)  {  …… // 所有函数的实现均和前述内容相同，这里就不再复述，下同。  }  Hero::~Hero(void)  {  }  void Hero::setName()  {  ……  }  void Hero::choice()  {  ……  }  void Hero::attack(Monster &mon)  {  ……  }  void Hero::levelUp(int up)  {  ……  }  void Hero::levelDown()  {  ……  }  void Hero::settlement(int level)  {  ……  } |

|  |  |
| --- | --- |
| 图 8‑5怪物类 | #include "role.h"  class Hero;  **class Monster :**  **public Role**  {  public:  Monster(void);  Monster(int);  ~Monster(void);  int ATK;  void attack(Hero&);  }; |

同Hero类一样，此处Monster类也继承于Role类。

因此，共同的成员此处就不需要再次声明，只需要声明不同的特有成员。

**注意：**Hero类和Monster类均有一个攻击（attack）的成员方法，但他们的参数并不相同，因此此处是两个不同的成员函数。当然我们也可以使得这两个成员函数的参数一致，从而提取出这个方法到Role类中，具体实现方法会在第九章中讲解。

最后，如何实现Monster类？和Hero类的实现类似的，Monster类也从Role继承了所有的公有成员，只需要实现上述声明的函数，代码如下：

|  |
| --- |
| #include "Monster.h"  #include "Hero.h"  Monster::Monster(void)  {  }  Monster::Monster(int optional)  {  …… // 所有函数的实现均和前述内容相同，这里就不再复述，下同。  }  Monster::~Monster(void)  {  }  void Monster::attack(Hero &h)  {  ……  } |

注意：这里我们可以看出Hero类和Monster类的派生的过程主要包括以下三点：

* 吸收基类成员

在Role类中我们声明了共有成员，所在在继承关系中派生类就从基类中吸收了基类的成员。即使在Hero类和Monster类中我们没有声明name、level和isAlive等这样成员，但他们都可以被继承到派生类中。

* 改造基类成员

对基类成员的改造包括两个方面，一个是基类成员的访问控制问题。另一个是对基类变量或者函数成员的覆盖或隐藏。例如在7.1.2节所介绍的showInfo函数。本节我们介绍了已有类的情况下，我们如何向上抽象出基类，为后续加入新的类提供便利。在下节我们还会介绍如何从已有类的情况下，如何派生出新的类来完成程序的重构，通过重构来给大家介绍如何改造基类成员。

* 添加新成员

派生类新成员的加入是继承与派生机制的核心，是保证派生类在功能上有所发展的关键。可以根据实际情况的需要，给派生类添加成员变量和成员函数，来实现必要的新增功能。比如在Hero类中，我们增加了技能列表，选择怪兽，升级降级等成员。使得Hero类的功能更加完善。

## 8.2 综合实例2---YYY系统项目

# 第九章 多态

## 9.1 综合实例1---LostCraft游戏

多态（Polymorphism）按字面的意思就是“多种状态”。在面向对象程序设计中真正的优势不仅仅在于继承，还在于将派生类对象当作基类对象一样处理的功能。支持这种功能的机制就是多态和动态绑定。

### 9.1.1 多态的概念

多态指同一个实体同时具有多种形式。它是面向对象程序设计的一个重要特征。如果一个语言只支持类而不支持多态，只能说明它是基于对象的，而不是面向对象的。**同一个消息被不同类型的对象接收时产生不同的行为就称为多态性，其特点就是一个接口，多个实现。**C++中的多态性具体体现在运行和编译两个方面。编译时多态是**静态多态**，在编译时就可以确定对象使用的形式。运行时多态是**动态多态**，其具体引用的对象在运行时才能确定。

现实世界中，多态的例子有很多，比如：机电学院的校长向社会发布一个消息：9月1日新学年开学。不同的对象则会作出不同的响应：学生要准备好课本准时到校上课；家长要筹集学费；教师要备好课；后勤部门要准备好教室、宿舍和食堂……

同样的道理，在我们的游戏世界中也存在着不少适用于多态的情况。简单的比如说在场景类中显示不同的信息我们使用函数重载的方法来实现，这是一种静态多态。再比如说上一章提到的英雄和怪物类中的攻击成员函数，其核心思想就是攻击，只是攻击的对象不同，实现的方法不同而已，这里就可以采用**虚函数**来完成。这就是一种动态多态。

### 9.1.2 静态多态之函数多态

如同我们在7.1.2节所讲，在Scenario类中，我们实现了三个同叫showInfo的成员函数。这三个函数的实现方法我们称之为重载。那么函数重载就是静态多态的表现形式之一。

在同一个类定义多个同名函数，参数列表不同，这种现象是多态的一个典型的表现形式，在程序执行的过程中，系统会自动根据传入的参数的类型自动的匹配与之对应的函数，从而实现函数的调用。

静态多态的优点就是：由于静多态是在编译期完成的，因此效率较高，编译器也可以进行优化。能够使接口和实现相分离，采用模板定义接口，类型参数定义实现。

### 9.1.3 动态多态之虚函数实现

那么接下来要研究的问题是：当一个基类被继承为不同的派生类时，各派生类可以使用与基类成员相同的成员名，如果在运行时用同一个成员名调用类对象的成员，会调用哪个对象的成员？也就是说，通过继承而产生了相关的不同的派生类，与基类成员同名的成员在不同的派生类中有不同的含义。也可以说，多态性是“一个接口，多种方法”。所以动态多态的设计思想就是对于相关的对象类型，确定它们之间的一个共同功能集，然后在基类中，把这些共同的功能声明为多个公共的虚函数接口。各个子类重写这些虚函数，以完成具体的功能。操作函数通过指向基类的引用或指针来操作这些对象，对虚函数的调用会自动绑定到你实际提供的子类对象上去。

回顾我们的LostCraft游戏当中，在上一章我们提到了Hero类和Monster类均有一个攻击（attack）的成员方法。但他们的参数并不相同。所以我们可以采用动态多态的方式，在Role类中实现一个虚函数。C++中定义虚函数通过在函数签名之前添加virtual。接下来我们分享如何通过“多态”来解决上述问题。

根据“多态”的定义结合“虚函数”的特征，我们首先在父类Role中定义一个虚成员函数attack来定义攻击行为，如下代码所示：

|  |  |
| --- | --- |
| // Role.h  class Role  {  public:  Role(void);  ~Role(void);  string name;  int level;  int HP;  **virtual void attack(Role&);**  bool isAlive();  }; | // Role.cpp  #include "Role.h"  Role::Role(void)  {  }  Role::~Role(void)  {  }  **void Role::attack(Role& r)**  {  **return;**  }  bool Role::isAlive()  {  return HP>0;  } |

* 一般虚函数成员的声明语法是：

**virtual 函数类型 函数名(形参表);**

* 虚函数声明只能出现在类定义中的函数原型声明中，而不能在成员函数实现的时候。
* 攻击函数的参数是Role&。因为Role类是Hero类和Monster类的公共父类，所以类之间满足赋值兼容规则，也就是说将来在实际传递参数时，不管传递的是Hero类还是Monster类，都会向上转换为Role类。
* 虚函数也必须有相应的实现。

同时，我们分析一下这里的attack函数，它的函数内容是return;并没有做任何操作就return返回了。为什么这么做呢？因为在基类中，我们现在并不知道如何来进行攻击。因此在实际项目中我们会在声明中加上“**=0**”来达到这个目的。我们把这样的函数称之为“**纯虚函数**”，修改代码如下：

**virtual void attack(Role&)=0;**

这样一来对应的cpp实现代码中就不需要再写任何代码在实现这个方法了。

注意：**当C++的类中包含了纯虚函数这个类就成了*抽象类*，抽象类规定，继承了抽象的子类必须实现抽象类中定义的抽象方法，*且抽象类不能被实例化*。**

那么相应的Hero类和Monster类中也需要做相应的修改，来继承并覆盖上述定义的虚函数。

|  |  |
| --- | --- |
| // Hero.h  #include "role.h"  #include "Craftsmanship.h"  class Monster;  class Hero :  public Role  {  public:  Hero(void);  ~Hero(void);  int MP;  Craftsmanship crafts[NUM\_CRAFTS];  void setName();  void choice();  **void attack(Role&);**  void levelUp(int);  void levelDown();  private:  int iAttack;  void settlement(int level);  }; | // Hero.cpp  #include "Hero.h"  #include "Monster.h"  Hero::Hero(void)  {  ……  }  Hero::~Hero(void)  {  }  void Hero::setName()  {  ……  }  void Hero::choice()  {  ……  }  **void Hero::attack(Role &r)**  {  **r.HP -= crafts[iAttack].ATK;**  MP -= crafts[iAttack].consumption;  crafts[iAttack].action = crafts[iAttack].cooling + 1;  }  void Hero::levelUp(int up)  {  ……  }  void Hero::levelDown()  {  ……  }  void Hero::settlement(int level)  {  ……  } |
| // Monster.h  #include "role.h"  class Hero;  class Monster :  public Role  {  public:  Monster(void);  Monster(int);  ~Monster(void);  int ATK;    **void attack(Role&);**  }; | //Monster.cpp  #include "Monster.h"  #include "Hero.h"  Monster::Monster(void)  {  }  Monster::Monster(int optional)  {  ……  }  Monster::~Monster(void)  {  }  **void Monster::attack(Role &r)**  {  **r.HP -= ATK;**  } |

* 在派生类中实现（覆盖）基类中的虚成员函数时，派生类的声明与实现不需要添加virtual关键字。
* 派生类在执行代码时，由成员函数来调用或者是通过指针、引用来访问虚函数。

接下来我们来实现操作函数。由于在原来的Scenario类中我们声明了一个Hero对象和一个Monster对象，并未使用指针或者引用的方式来实现，所以我们对原有的Scenario类进行重构，使之能成为采用多态来完成游戏的场景。



那么在上一章中我们讲到了继承，采用继承的方法可以向上抽象类的共同特征，减少扩展类的时候的代码量。同样的，我们也可以采用继承的方法来重构现有类。类图如图所示：

我们在派生出的AdvScenario类中新增两个指针对象rHero和rMonster指针。并且修改choice函数的返回值为void。从而使得在新的场景类中我们可以使用基类指针的方式来表示英雄对象和怪物对象。这样的方式我们称之为类型兼容。

**类型兼容规则是指在需要基类对象的任何地方，都可以使用公有派生类的对象来替代。**通过公有继承，派生类得到了基类中除构造函数、析构函数之外的所有成员。这样，公有派生类实际就具备了基类的所有功能，凡是基类能解决的问题，公有派生类都可以解决。类型兼容规则中所指的替代包括以下情况：

* 派生类的对象可以赋值给基类对象；
* 派生类的对象可以初始化基类的引用；
* 派生类对象的地址可以赋给指向基类的指针。

**在替代之后，派生类对象就可以作为基类的对象使用，但是只能使用从基类继承的成员。**类型兼容规则是多态性的重要基础之一。

那么如何声明和实现AdvScenario类呢？代码如下：

|  |  |
| --- | --- |
| #include "scenario.h"  class AdvScenario :  public Scenario  {  public:  AdvScenario(void);  ~AdvScenario(void);  Role \*rHero; // 英雄的对象指针  Role \*rMonster; // 怪物的对象指针  void choice();  bool challenge();  void showInfo();  void showInfo(string);  }; | #include "AdvScenario.h"  AdvScenario::AdvScenario(void)  {  **rHero = new Hero();**  **rMonster = nullptr;**  }  AdvScenario::~AdvScenario(void)  {  **delete rHero;**  **delete rMonster;**  }  void AdvScenario::choice()  {  int optional;  do  {  CLS;  cout <<right << "请选择你要挑战的怪兽：" << endl;  cout << setfill('-') << setw(30) << "" << endl;  cout << "| 1:" << setfill(' ') << setw(26) << "巴尔坦虫怪【lv1】|" << endl;  cout << "| 2:" << setw(26) << "格斯安鱼怪【lv5】|" << endl;  cout << "| 3:" << setw(26) << "泰斯鸟怪【lv10】|" << endl;  cout << "| 4:" << setw(26) << "绿头猿怪【lv30】|" << endl;  cout << "| 5:" << setw(26) << "赤龙怪(BOSS)【lv50】|" << endl;  cout << setfill('-') << setw(30) << "" << endl;  cin >> optional;  } while (optional<1 || optional>5);  **if (rMonster!=nullptr)**  **{**  **delete rMonster;**  **}**  **rMonster = new Monster(optional-1);**  **return;**  }  bool AdvScenario::challenge()  {  while(true)  {  showInfo();  **((Hero \*)rHero)->choice();**  **rHero->attack(\*rMonster)**;  showInfo(**rHero->name**);  if(**rMonster->isAlive()**)  {  **rMonster->attack(\*rHero);**  showInfo(**rMonster->name**);  if (!**rHero->isAlive()**)  {  **((Hero \*)rHero)->levelDown();**  if (**rHero->level**<=0)  {  cout << endl << "======游戏失败！======" << endl;  return false;  }  return true;  }  else  {  round++;  for (int i=0;i!=NUM\_CRAFTS;i++)  {  **((Hero \*)rHero)->crafts[i].action = ((Hero \*)rHero)->crafts[i].action <= 0 ? 0 : ((Hero \*)rHero)->crafts[i].action - 1;**  }  PC;  continue;  }  }  else  {  if (**rMonster->level** == 50)  {  cout << endl << "======恭喜通关！======" << endl;  return false;  }  **((Hero \*)rHero)->levelUp(rMonster->level);**  return true;  }  }  }  void AdvScenario::showInfo()  {  …… // 修改myHero和myMonster对象为rHero和rMonster指针，下同  }  void AdvScenario::showInfo(string name)  {  ……  } |

### 9.1.4 LostCraft游戏项目最终实现之核心代码展示

那么最终我们的LostCraft游戏最终的代码实现如下：

|  |  |
| --- | --- |
| Craftsmanship.h | Craftsmanship.cpp |
| #pragma once  const string craftnames[NUM\_CRAFTS]={"普通击打","寒冰掌","石破惊雷","赤火焰烧","月蚀镰刀","圣光冲击","天师灵助","天威庇佑"};  const int availables[NUM\_CRAFTS]={1,1,5,10,20,30,35,45};  const int mps[NUM\_CRAFTS] = {0,5,10,20,40,60,80,100};  const int atks[NUM\_CRAFTS] = {10,12,15,18,20,22,25,30};  const int coolings[NUM\_CRAFTS] = {0,1,3,3,4,4,5,5};  class Craftsmanship  {  public:  Craftsmanship(void);  ~Craftsmanship(void);  string name; // 名称  int consumption;// 所消耗MP的值  int ATK; // 攻击力  int cooling; // 技能规定冷却时间  int available; // 技能所需英雄等级  int action; // 技能冷却回合  }; | #include "StdAfx.h"  #include "Craftsmanship.h"  Craftsmanship::Craftsmanship(void)  {  }  Craftsmanship::~Craftsmanship(void)  {  } |

|  |  |
| --- | --- |
| Role.h | Role.cpp |
| #pragma once  class Role  {  public:  Role(void);  ~Role(void);  string name;  int level;  int HP;  virtual void attack(Role&)=0;  bool isAlive();  }; | #include "StdAfx.h"  #include "Role.h"  Role::Role(void)  {  }  Role::~Role(void)  {  }  bool Role::isAlive()  {  return HP>0;  } |

|  |  |
| --- | --- |
| Hero.h | Hero.cpp |
| #pragma once  #include "role.h"  #include "Craftsmanship.h"  class Monster;  class Hero :  public Role  {  public:  Hero(void);  ~Hero(void);  int MP;  Craftsmanship crafts[NUM\_CRAFTS];  void setName();  void choice();  void attack(Role&);  void levelUp(int);  void levelDown();  private:  int iAttack;  void settlement(int level);  }; | #include "StdAfx.h"  #include "Hero.h"  #include "Monster.h"  Hero::Hero(void)  {  level = 1;  HP = 2000;  MP = 50;    for (int i=0;i!=NUM\_CRAFTS;i++)  {  crafts[i].name = craftnames[i];  crafts[i].available = availables[i];  crafts[i].consumption = mps[i];  crafts[i].ATK = atks[i]\*level;  crafts[i].cooling = coolings[i];  crafts[i].action = 0;  }  }  Hero::~Hero(void)  {  }  void Hero::setName()  {  cout << "请输入英雄的名字：";  cin >> name;  cout << "英雄：" << name << "初始化成功！" << endl;  PC;  }  void Hero::choice()  {  cout << "请选择技能：" << endl;  for (int i=0;i!=NUM\_CRAFTS;i++)  {  if (crafts[i].available<=level)  {  cout << i+1 << ":" << crafts[i].name << "【" << (crafts[i].action==0?"●":"×") << "】" << endl;  }  }  do  {  int attack;  cin >> attack;  if (attack>0 && attack <=NUM\_CRAFTS && crafts[attack-1].available<=level)  {  if (crafts[attack-1].action)  {  cout << "该技能正在冷却，请稍后使用！" << endl;  cout << "请重新选择技能：" << endl;  continue;  }  if (MP-crafts[attack].consumption<0)  {  cout << "魔法值不足，请选择其他方式攻击！" << endl;  cout << "请重新选择技能：" << endl;  continue;  }  iAttack = attack-1;  return;  }  else  {  cout << "选择有误！" << endl;  cout << "请重新选择技能：" << endl;  continue;  }  } while (true);  }  void Hero::attack(Role &r)  {  r.HP -= crafts[iAttack].ATK;  MP -= crafts[iAttack].consumption;  crafts[iAttack].action = crafts[iAttack].cooling + 1;  }  void Hero::levelUp(int up)  {  cout << "=======恭喜升级！======" << endl;  level += up;  settlement(level);  PC;  }  void Hero::levelDown()  {  cout << "======可恶，降级了！======" << endl;  level--;  settlement(level);  PC;  }  void Hero::settlement(int level)  {  HP = 2000 + (level-1)\*500;  MP = 50 + (level-1)\*10;  int atks[NUM\_CRAFTS] = {10,12,15,18,20,22,25,30};  for (int i=0;i!=NUM\_CRAFTS;i++)  {  crafts[i].ATK = atks[i]\*level;  crafts[i].action = 0;  }  } |

|  |  |
| --- | --- |
| Monster.h | Monster.cpp |
| #pragma once  #include "role.h"  class Hero;  class Monster :  public Role  {  public:  Monster(void);  Monster(int);  ~Monster(void);  int ATK;    void attack(Role&);  }; | #include "StdAfx.h"  #include "Monster.h"  #include "Hero.h"  Monster::Monster(void)  {  }  Monster::Monster(int optional)  {  string names[5] = {"巴尔坦虫怪","格斯安鱼怪","泰斯鸟怪","绿头猿怪","赤龙怪(BOSS)"};  int levels[5] = {1,5,10,30,50};  name = names[optional];  level = levels[optional];  HP = 100 \* level;  ATK = 20 \* level;  }  Monster::~Monster(void)  {  }  void Monster::attack(Role &r)  {  r.HP -= ATK;  } |

|  |  |
| --- | --- |
| Scenario.h | Scenario.cpp |
| #pragma once  #include "Hero.h"  #include "Monster.h"  class Scenario  {  public:  Scenario(void);  ~Scenario(void);  Hero myHero; // 英雄对象  Monster myMonster; // 怪物对象  bool state; // GO（GameOver）标志  int round; // 当前回合  Monster choice();  bool challenge();  void showInfo();  void showInfo(string);  }; | #include "StdAfx.h"  #include "Scenario.h"  Scenario::Scenario(void)  {  cout << "==============================" << endl;  cout << "====拯救世界吧，少年！v2.0====" << endl;  cout << "==============================" << endl;  PC;  }  Scenario::~Scenario(void)  {  }  Monster Scenario::choice()  {  int optional;  do  {  CLS;  cout <<right << "请选择你要挑战的怪兽：" << endl;  cout << setfill('-') << setw(30) << "" << endl;  cout << "| 1:" << setfill(' ') << setw(26) << "巴尔坦虫怪【lv1】|" << endl;  cout << "| 2:" << setw(26) << "格斯安鱼怪【lv5】|" << endl;  cout << "| 3:" << setw(26) << "泰斯鸟怪【lv10】|" << endl;  cout << "| 4:" << setw(26) << "绿头猿怪【lv30】|" << endl;  cout << "| 5:" << setw(26) << "赤龙怪(BOSS)【lv50】|" << endl;  cout << setfill('-') << setw(30) << "" << endl;  cin >> optional;  } while (optional<1 || optional>5);  Monster mon(optional-1);  return mon;  }  bool Scenario::challenge()  {  while(true)  {  showInfo();  myHero.choice();  myHero.attack(myMonster);  showInfo(myHero.name);  if(myMonster.isAlive())  {  myMonster.attack(myHero);  showInfo(myMonster.name);  if (!myHero.isAlive())  {  myHero.levelDown();  if (myHero.level<=0)  {  cout << endl << "======游戏失败！======" << endl;  return false;  }  return true;  }  else  {  round++;  for (int i=0;i!=NUM\_CRAFTS;i++)  {  myHero.crafts[i].action = myHero.crafts[i].action <= 0 ? 0 : myHero.crafts[i].action - 1;  }  PC;  continue;  }  }  else  {  if (myMonster.level == 50)  {  cout << endl << "======恭喜通关！======" << endl;  return false;  }  myHero.levelUp(myMonster.level);  return true;  }  }  }  void Scenario::showInfo()  {  cout << left << setfill('-') << setw(39) << "" << endl; // 表格宽度共计40个半角字符  cout << "|" << setfill(' ') << "第" << setw(3) << round << "轮" <<setw(30) << "" << "|" << endl;  cout << setfill('-') << setw(39) << "" << endl;  cout << "|" << setfill(' ') << setw(6) << "英雄：" << setw(12) << myHero.name << "|" << setw(6) << "怪兽：" << setw(12) << myMonster.name << "|" << endl;  cout << "|" << setw(6) << "级别：" << setw(12) << myHero.level << "|" << setw(6) << "级别：" << setw(12) << myMonster.level << "|" << endl;  cout << "|" << setw(6) << "HP：" << setw(12) << myHero.HP << "|" << setw(6) << "HP：" << setw(12) << myMonster.HP << "|" << endl;  cout << "|" << setw(6) << "MP：" << setw(12) << myHero.MP << "|" << setw(18) << "" << "|" << endl;  cout << setfill('-') << setw(39) << "" << endl;  }  void Scenario::showInfo(string name)  {  cout << left << setfill('-') << setw(39) << "" << endl; // 表格宽度共计40个半角字符  cout << "|" << setfill(' ') << setw(12) << name << "攻击！" << setw(19) << "" << "|" << endl;  cout << setfill('-') << setw(39) << "" << endl;  cout << "|" << setfill(' ') << setw(6) << "英雄：" << setw(12) << myHero.name << "|" << setw(6) << "怪兽：" << setw(12) << myMonster.name << "|" << endl;  cout << "|" << setw(6) << "级别：" << setw(12) << myHero.level << "|" << setw(6) << "级别：" << setw(12) << myMonster.level << "|" << endl;  cout << "|" << setw(6) << "HP：" << setw(12) << myHero.HP << "|" << setw(6) << "HP：" << setw(12) << myMonster.HP << "|" << endl;  cout << "|" << setw(6) << "MP：" << setw(12) << myHero.MP << "|" << setw(18) << "" << "|" << endl;  cout << setfill('-') << setw(39) << "" << endl;  } |

|  |  |
| --- | --- |
| AdvScenario.h | AdvScenario.cpp |
| #pragma once  #include "scenario.h"  class AdvScenario :  public Scenario  {  public:  AdvScenario(void);  ~AdvScenario(void);  Role \*rHero; // 英雄的对象指针  Role \*rMonster; // 怪物的对象指针  void choice();  bool challenge();  void showInfo();  void showInfo(string);  }; | #include "StdAfx.h"  #include "AdvScenario.h"  AdvScenario::AdvScenario(void)  {  rHero = new Hero();  rMonster = nullptr;  }  AdvScenario::~AdvScenario(void)  {  delete rHero;  delete rMonster;  }  void AdvScenario::choice()  {  int optional;  do  {  CLS;  cout <<right << "请选择你要挑战的怪兽：" << endl;  cout << setfill('-') << setw(30) << "" << endl;  cout << "| 1:" << setfill(' ') << setw(26) << "巴尔坦虫怪【lv1】|" << endl;  cout << "| 2:" << setw(26) << "格斯安鱼怪【lv5】|" << endl;  cout << "| 3:" << setw(26) << "泰斯鸟怪【lv10】|" << endl;  cout << "| 4:" << setw(26) << "绿头猿怪【lv30】|" << endl;  cout << "| 5:" << setw(26) << "赤龙怪(BOSS)【lv50】|" << endl;  cout << setfill('-') << setw(30) << "" << endl;  cin >> optional;  } while (optional<1 || optional>5);  if (rMonster!=nullptr)  {  delete rMonster;  }  rMonster = new Monster(optional-1);  return;  }  bool AdvScenario::challenge()  {  while(true)  {  showInfo();  ((Hero \*)rHero)->choice();  rHero->attack(\*rMonster);  showInfo(rHero->name);  if(rMonster->isAlive())  {  rMonster->attack(\*rHero);  showInfo(rMonster->name);  if (!rHero->isAlive())  {  ((Hero \*)rHero)->levelDown();  if (rHero->level<=0)  {  cout << endl << "======游戏失败！======" << endl;  return false;  }  return true;  }  else  {  round++;  for (int i=0;i!=NUM\_CRAFTS;i++)  {  ((Hero \*)rHero)->crafts[i].action = ((Hero \*)rHero)->crafts[i].action <= 0 ? 0 : ((Hero \*)rHero)->crafts[i].action - 1;  }  PC;  continue;  }  }  else  {  if (rMonster->level == 50)  {  cout << endl << "======恭喜通关！======" << endl;  return false;  }  ((Hero \*)rHero)->levelUp(rMonster->level);  return true;  }  }  }  void AdvScenario::showInfo()  {  cout << left << setfill('-') << setw(39) << "" << endl; // 表格宽度共计40个半角字符  cout << "|" << setfill(' ') << "第" << setw(3) << round << "轮" <<setw(30) << "" << "|" << endl;  cout << setfill('-') << setw(39) << "" << endl;  cout << "|" << setfill(' ') << setw(6) << "英雄：" << setw(12) << rHero->name << "|" << setw(6) << "怪兽：" << setw(12) << rMonster->name << "|" << endl;  cout << "|" << setw(6) << "级别：" << setw(12) << rHero->level << "|" << setw(6) << "级别：" << setw(12) << rMonster->level << "|" << endl;  cout << "|" << setw(6) << "HP：" << setw(12) << rHero->HP << "|" << setw(6) << "HP：" << setw(12) << rMonster->HP << "|" << endl;  cout << "|" << setw(6) << "MP：" << setw(12) << ((Hero \*)rHero)->MP << "|" << setw(18) << "" << "|" << endl;  cout << setfill('-') << setw(39) << "" << endl;  }  void AdvScenario::showInfo(string name)  {  cout << left << setfill('-') << setw(39) << "" << endl; // 表格宽度共计40个半角字符  cout << "|" << setfill(' ') << setw(12) << name << "攻击！" << setw(19) << "" << "|" << endl;  cout << setfill('-') << setw(39) << "" << endl;  cout << "|" << setfill(' ') << setw(6) << "英雄：" << setw(12) << rHero->name << "|" << setw(6) << "怪兽：" << setw(12) << rMonster->name << "|" << endl;  cout << "|" << setw(6) << "级别：" << setw(12) << rHero->level << "|" << setw(6) << "级别：" << setw(12) << rMonster->level << "|" << endl;  cout << "|" << setw(6) << "HP：" << setw(12) << rHero->HP << "|" << setw(6) << "HP：" << setw(12) << rMonster->HP << "|" << endl;  cout << "|" << setw(6) << "MP：" << setw(12) << ((Hero \*)rHero)->MP << "|" << setw(18) << "" << "|" << endl;  cout << setfill('-') << setw(39) << "" << endl;  } |

## 9.2 综合实例2---YYY系统项目