

# 第二天课堂笔记

## 一、回顾知识点

### 1.1 Qt安装

不要安装在C:\Qt位置，应该放在 C:\Users\{自己的用户名}\Qt 位置，安装路径不能出现中文或空格或特殊符号。

### 1.2 C++的概述

C++是对C扩展，C的语法完全在c++中使用，c++包含C的面向过程编程、新的面向对象编程和模板编程（泛型）。

标准(跨平台):

- ANSI/ISO C++ 98
- C++2003
- C++11(2011)

### 1.3 C++的面向对象

面向过程：程序由数据结构+算法组成的。

面向对象：程序由多个对象组成的，一个对象由数据结构+算法组成的。

面向对象的三大特征：

- 1) 封装性：由客观事物抽象出类，在类中定义数据（属性，特征）和方法（行为），依据访问权限，设置类的外部对数据及方法的可访问性。
- 2) 继承性：类与类之间的关系是父子的继承关系，子类可以继承父类的数据和行为，提高了代码复用性、降低代码冗余。
- 3) 多态性：一个接口，多种实现方法

### 1.4 c++扩展c

1) :: 运算符可以访问全局变量（局部变量与全局变量名冲突时）

2) 命名空间 namespace（提高变量重名）

定义命名空间： namespace 名称{ 成员; }

命名空间内的成员类型： 变量、函数(定义、声明)、类、结构体...

实现命名空间内的声明函数的功能： 返回值类型 命名空间名::函数名() {}

使用命名空间：

1) using声明， using 命名空间名称::成员名；

在声明的区域内，成员名可以直接作为变量使用。

2) using 编译， using namespace 命名空间名称；

命名空间内的成员，可以直接访问，但如果之前已存在相同名的变量时，必须加 x:: 前缀避免访问了当前已声明局部变量

3) 强化数据类型转化

```
char *p = (char *)malloc()
const int a=10;
int *p = (int *)&a;
```

#### 4) 优化了struct

定义结构体类型的变量时，不需要struct关键字

#### 5) 强化变量数据类型的声明

```
void add(int a); // void add(a);
```

#### 6) 三目运算表达式增强

c 的三目表达式返回是变量值，而c++返回 是变量  
(a>b?a:b) = 10; // 给a或b变量赋值为10

#### 7) 新增 bool类型，值为true或false

#### 8) 全局变量的检查

c 中可以先定义变量，然后再声明变量  
c++ 声明变量即定义变量。

#### 9) const不同

C中const变量会在栈区分配空间

c++中const变量默认情况下，不会创建栈区空间，只存放在符号表中。

存在以下几种情况，会创建栈区空间：

- 1) 取const变量地址时，立即分配空间
- 2) const变量使用其它变量作为初始化值时，立即分配空间
- 3) 类中使用时，立即分配空间

const变量与宏的区别：

const变量有数据类型， 无宏没有数据类型

const变量存在作用域， 无宏没有作用域（有效范围是从定义开始到结束）

无宏不能访问类的其它成员

## 二、C++扩展C的II

### 2.1 const和#define的区别

#### 1) const变量有数据类型， 而#define 宏无数据类型

调用重载的函数时，传入const变量是正确的，而传入宏, 是无法确认调用函数的入口。

#### 2) const变量具有作用域， 而#define 宏从定义开始到文件结束都是有效的。

```
void f1(){
    const int a=100;
#define x 10;
    cout << "x=" << x << endl; // 10
}

int main(){
    cout << "x=" << x << endl; // 10
    cout << "a=" << a << endl; // error
}
```

#### 3) #define 宏在命名空间中定义时，是否只属于命名空间

不是，因为宏是没有作用域的。

## 2.2 引用(reference) 【重要】

引用相当于指针，在向函数传递地址时，可以使用指针，也可以使用引用，引用是C++对C的重要的扩充。

### 2.2.1 引用的基本用法

普通变量的用法：为变量起别名，变量的引用名即和变量名操作同一个内存空间。

定义引用变量时，必须给初始化值，另外，定义之后引用不能再引用其它的变量。

初始值不能是NULL。

```
int a=10;
int& b = a;
b+= 20; // a += 20
// a=30, b=30
```

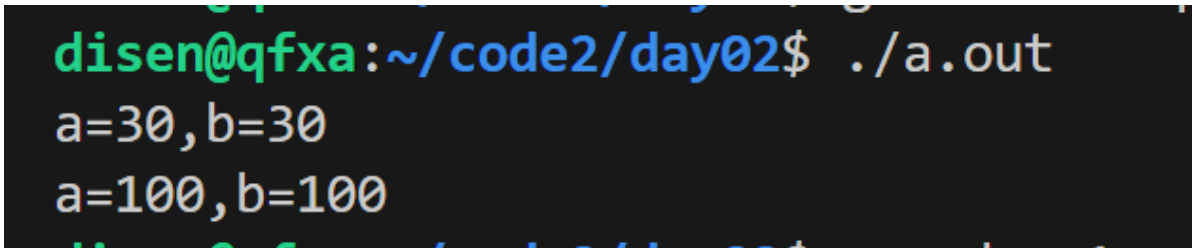
```
int a=10;
int b=20;
int &c = a;
c = b; // 将b变量的值赋值给c引用的空间， 正确的
// &c = b; c引用不能改为其它变量的空间。
```

```
#include <iostream>

using namespace std;

int main(int argc, char const *argv[])
{
    int a = 10;
    int &b = a; // & 是引用符号， 不是取变量的地址
    b += 20;
    cout << "a=" << a << ",b=" << b << endl;

    int c = 100;
    b = c;
    cout << "a=" << a << ",b=" << b << endl;
    return 0;
}
```



```
disen@qfxa:~/code2/day02$ ./a.out
a=30,b=30
a=100,b=100
```

引用数组的正确方式：不能直接引用数组 (int &p = arr)

```
#include <iostream>

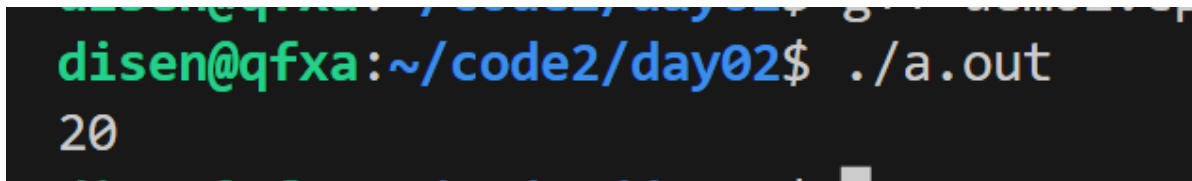
using namespace std;

int main(int argc, char const *argv[])
{
    int arr[5] = {1, 2, 3, 4, 5};

    // 给arr设置一个别名（引用）
    // 1) 先定义 5个元素数组类型的别名
    typedef int FiveArr[5];
    FiveArr &p = arr;

    // 2) 直接定义数组的引用
    int(&q)[5] = arr;

    p[0] = 20;
    cout << q[0] << endl;
    return 0;
}
```



```
disen@qfxa:~/code2/day02$ ./a.out
20
```

## 2.2.2 函数的参数引用

形参可以是引用，声明方式：数据类型 &形参名

函数的返回值也可以是一个引用，返回的变量不能是局部变量。

【扩展】如果函数返回值作为赋值语句的左值，则必须返回引用。

```
#include <iostream>

using namespace std;

void cumsum(int &total, int n)
{
    // total 是一个引用，也可以实参的别名
    total += n;
}

int main(int argc, char const *argv[])
{
    int total = 0;
    for (int i = 1; i <= 10; i++)
    {
        cumsum(total, i);
    }
    cout << "total=" << total << endl;
    return 0;
}
```

如2：返回值为数组引用的使用方法

```

#include <iostream>
#define N 10

using namespace std;
typedef int TenArr[N];

TenArr &new_arr()
{
    static int m[N] = {0}; // 全局静态区存储 m数组
    for (int i = 0; i < N; i++)
    {
        m[i] = i;
    }
    return m;
}

int main(int argc, char const *argv[])
{
    TenArr &p = new_arr();
    for (int i = 0; i < N; i++)
    {
        cout << "p[" << i << "]=" << p[i] << endl;
    }
    return 0;
}

```

```

p[0]=0
p[1]=1
p[2]=2
p[3]=3
p[4]=4
p[5]=5
p[6]=6
p[7]=7
p[8]=8
p[9]=9

```

如3：数组引用作为函数的参数, 数组名本身就是一个地址，无需再按数组的引用定义方式来定义形参。

```

#include <iostream>

using namespace std;

// void sort(int (&arr)[], int size) 编译报错
// 在c或c++数组的传递都是地址，不需要取引用（地址）。
void sort(int arr[], int size)

```

```

{
    for (int i = 0; i < size - 1; i++)
    {
        for (int j = 0; j < size - 1 - i; j++)
        {
            if (arr[j] > arr[j + 1])
            {
                arr[j] = arr[j] ^ arr[j + 1];
                arr[j + 1] = arr[j] ^ arr[j + 1];
                arr[j] = arr[j] ^ arr[j + 1];
            }
        }
    }
}

int main(int argc, char const *argv[])
{
    int nums[] = {1, 5, 2, 0, 9, 10, 7};
    sort(nums, 7);
    for (int i = 0; i < 7; i++)
    {
        cout << nums[i] << "\t";
    }
    cout << endl;
}

```

### 2.2.3 引用的本质

引用的本质是指针常量，初始化时必须给定初始值，因为指针不能修改。

如：

```

int a=10;
int &aa = a; // int * const aa = &a;

```

【扩展】常量指针和指针常量的区别

常量指针：     const int \*p = &a;   const修饰是\*p，\*p只读， p读写  
 指针常量：     int \* const p = &a;   const修饰是p，p只读，\*p读写。  
 常量指针常量：   const int \* const p= &a;     \*p只读， p只读

### 2.2.4 指针的引用

为指针起个别名，表示为指针的指针。

用法：数据类型 \*& 变量名 = 指针变量名。

如：

```

#include <iostream>
#include <cstdlib>

using namespace std;

struct STU
{

```

```

    int sid;
    float score;
};

void set_score(STU **p, int sid, float score)
{
    (*p)->sid = sid;
    (*p)->score = score;
}

int main()
{
    STU *p = (STU *)malloc(sizeof(STU));
    set_score(&p, 1, 99);
    cout << "sid=" << p->sid << ", score=" << p->score << endl;
    return 0;
}

```

修改为指针引用的方式：

```

#include <iostream>
#include <cstdlib>

using namespace std;

struct STU
{
    int sid;
    float score;
};

void set_score(STU *&q, int sid, float score)
{
    q->sid = sid;
    q->score = score;
}

int main()
{
    STU *p = (STU *)malloc(sizeof(STU));
    set_score(p, 2, 98); // 实参是p指针，接收是指针引用，自动取p的地址
    cout << "sid=" << p->sid << ", score=" << p->score << endl;
    return 0;
}

```

## 2.2.5 常量引用

用法： const 数据类型 & 变量名 = 其它变量或常量；

【注意】

- 1) 一般的引用不能赋值常量（字面量），但是const引用可以赋值常量。
- 2) const引用不能修改内容（数据）。

如：

```

#include <iostream>

using namespace std;

int main(int argc, char const *argv[])
{
    int x = 10;
    const int &x1 = x;
    // x1 -= 5; // 报错,assignment of read-only reference 'x1'
    const int &x2 = 50;
    cout << "x2=" << x2 << endl;
    // int &x3 = 'a'; // 报错, 常量不能赋值引用的
    cout << "x3=" << x3 << endl;
}

```

## 2.3 内联函数

定义函数的前面添加了 inline 关键字，则此函数为内联函数。

内联函数只能在当前文件中使用，相当于函数前面加 static。

内联函数一般用于替换 有参的宏，有参宏经常会出错，而且参数是无数据类型。

每一次使用内联函数时，都会像有参宏一样，展开一次（内联函数不入栈，运行效率高）。

内联函数为了继承宏函数的效率，没有函数调用时开销，然后又可以像普通函数那样使用，每次使用时都会进行参数、返回值类型的安全检查，又可以作为成员函数。

内联函数作为类的成员函数时，可以访问私有成员（数据）。

内联函数的声明和定义必须在一起，否则取消内联函数性质。

如：

```

#include <iostream>
#define ADD(a, b) a + b

using namespace std;
inline int add(int a, int b)
{
    return a + b;
}

int main(int argc, char const *argv[])
{
    int ret = add(10, 20) * 100;
    cout << "ret=" << ret << endl;
    int ret2 = ADD(10, 20) * 100;
    cout << "ret2=" << ret2 << endl;
}

```



```
disen@qfxa:~/code2/day02$ ./a.out
ret=3000
ret2=2010
```

【说明】类中所有函数，编译器默认都是内联函数。`inline`修饰的函数是否为内联函数，取决于编译器。对于非`inline`修饰的函数，也有可能转成内联函数（体积小、功能简单的函数）。

内联函数使用时的注意事项：

- 1) 不能存在任何形式的循环语句
- 2) 不能存在过多的条件判断语句
- 3) 函数体不能过于庞大
- 4) 不能对函数进行取址操作

## 2.4 函数的默认值参数

c++的函数在声明时，可以设置形参的默认值，在调用函数时，形参位置没有指定实参数时，则使用默认值。

如：

```
#include <iostream>

using namespace std;

bool isSs(int n = 10)
{ // 判断n是否质数
    int i = 2;
    for (; i < n / 2; i++)
    {
        if (n % i == 0)
            return false;
    }

    return true;
}

double area(double r = 1.0)
{
    return r * r * 3.14;
}

int main(int argc, char const *argv[])
{
    cout << "isSs(7) is " << isSs(7) << endl;
    cout << "isSs() is " << isSs() << endl;
    cout << "area() is " << area() << endl;
    cout << "area(2) is " << area(2) << endl;

    return 0;
}
```

```
}
```

```
disen@qfxa:~/code2/day02$ ./a.out
isSS(7) is 1
isSS() is 0
area() is 3.14
area(2) is 12.56
```

#### 【注意】

函数的默认参数从左向右，如果一个参数设置了默认参数，那么这个参数之后的参数都必须设置默认参数。如果函数声明和函数定义分开写，函数声明和函数定义不能同时设置默认参数。

如：

```
// 定义f函数的是错误的，从c参数开始，后面所有参数都应该有默认值
void f(int a, int b, int c=1, int d){

}

// 定义f2的函数是正确的
void f2(int a, int b=3, int c=4){

}
```

## 2.5 函数的占位参数

c++在声明函数时，可以设置占位参数。占位参数只有参数类型声明，而没有参数名。

一般情况下，在函数体内部无法使用占位参数。

占位参数也可以设置默认值。

【注意】调用函数时，占位参数也需要传值

使用场景：在后面我们要讲的操作符重载的后置++要用到这个。

如：

```
#include <iostream>

using namespace std;

void f(int a, int)
{
    cout << "f(int, int)" << endl;
}

void f(double, int)
{
    cout << "f(double, int)" << endl;
}

void f(int a)
```

```

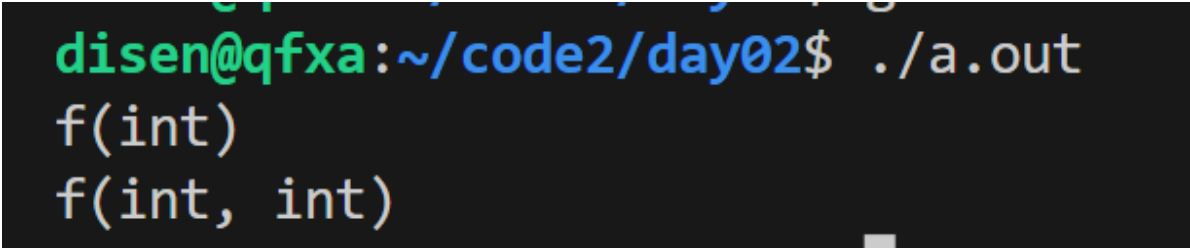
{
    cout << "f(int)" << endl;
}

void f(double a)
{
    cout << "f(double)" << endl;
}

int main(int argc, char const *argv[])
{
    f(1);
    f(2, 5);

    return 0;
}

```



```

disen@qfxa:~/code2/day02$ ./a.out
f(int)
f(int, int)

```

## 2.6 函数重载和extern "c"

c++支持函数的重载特性，函数重载即与函数名和参数列表相关，与函数的返回值无关。

存在多个函数名相同、参数列表不同的情况时，称之为函数重载。

参数列表不同：个数不同，个数相同时类型不同 或 顺序不同。

```

void f(int a);
void f(int a, int b); // f(int,int)
void f(double a,int b);
void f(int a, double b);
// void f(int b, int a); 重载检查按参数类型不同查找， f(int, int) 上面已声明了，不能再
次声明。

```

由于c++可以使用C的模块中函数，当C++整个工程编译时，可能会将使用C语言编写的函数名编译成c++规则的函数名（定义的函数名前随机添加新的名称），链接程序时，可能会找不到目标函数，因此采用 `extern "c"` 解决。

如：demo11/area.h

```

#ifndef __AREA__H
#define __AREA__H

extern double S(double r);
#endif

```

demo11/area.c

```
#include "area.h"

double S(double r)
{
    return r * r * 3.1415926;
}
```

demo11/main.cpp:

```
#include <iostream>
#include "area.h"

using namespace std;

int main(int argc, char const *argv[])
{
    double r = 2.5;
    double s = S(r);
    cout << "S(2.5)=" << s << endl;
}
```

执行如下命令，则会找不到C的S()函数：

```
disen@qfxa:~/code2/day02/demo11$ gcc -c area.c -o area.o
disen@qfxa:~/code2/day02/demo11$ g++ -c main.cpp -o main.o
disen@qfxa:~/code2/day02/demo11$ g++ main.o area.o -o main
main.o: In function `main':
main.cpp:(.text+0x2a): undefined reference to `S(double)'
collect2: error: ld returned 1 exit status
disen@qfxa:~/code2/day02/demo11$
```

`extern "C"` 用法：解决以上的问题，修改area.h头文件

```
#ifndef __AREA__H
#define __AREA__H
#if __cplusplus
extern "C" {
    // 如果在C++使用中，编译时函数名保持C的风格（原函数名）
    extern double S(double r);
}
#endif
#endif
```

再一次对c和cpp的文件进行编译与链接，则成功。

```
disen@qfxa:~/code2/day02/demo11$ gcc -c area.c -o area.o
disen@qfxa:~/code2/day02/demo11$ g++ -c main.cpp -o main.o
disen@qfxa:~/code2/day02/demo11$ g++ main.o area.o -o main
disen@qfxa:~/code2/day02/demo11$ ./main
S(2.5)=19.635
disen@qfxa:~/code2/day02/demo11$
```

## 三、类与对象【重点】

### 3.1 类与对象的概念

从C和C++的struct开始，C的struct结构体只能存在数据变量，而C++的struct体可以函数。

```
#include <iostream>

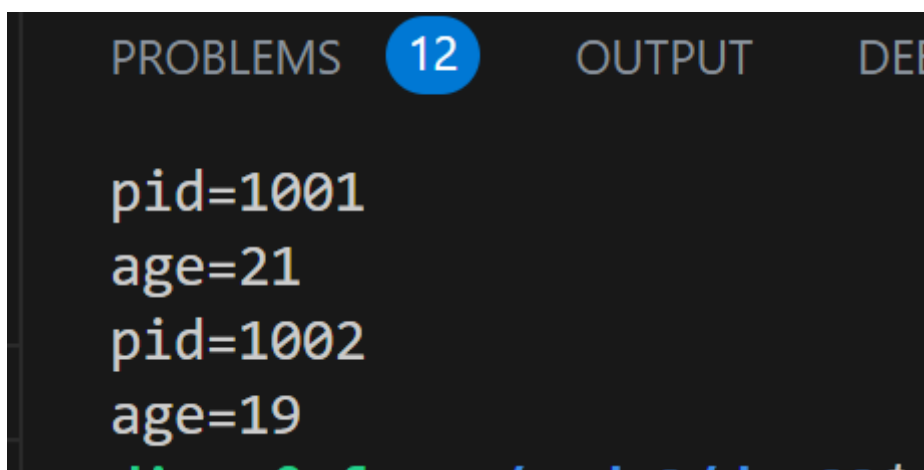
using namespace std;

struct Person // 描述员工的信息
{
    int pid;
    int age;
    void show() // 函数，结构体内的方法
    {
        // 可以访问同一个结构体内的成员变量
        cout << "pid=" << pid << endl;
        cout << "age=" << age << endl;
    }
};

int main(int argc, char const *argv[])
{
    Person p1 = {1001, 21};
    p1.show();

    Person p2 = {1002, 19};
    p2.show();

    return 0;
}
```



The screenshot shows the output of the program. At the top, there are tabs for 'PROBLEMS', '12' (selected), 'OUTPUT', and 'DEB'. The output text is as follows:

```
pid=1001
age=21
pid=1002
age=19
```

### 3.1.1 类的封装

通过类可以封装对象的属性（特征，数据变量或引用）、行为（函数，类方法），可以通过访问权限（公开 public、私有的 private、受保护 protected）设置属性和行为的访问性。私有的或受保护的权限的（数据或方法）可以通过公开的方法（行为）进行操作。

类的封装主要的任务：

1. 把变量（属性）和函数（操作）合成一个整体，封装在一个类中
2. 对变量和函数进行访问控制

访问属性	属性	对象内部	对象外部
public	公有	可访问	可访问
protected	保护	可访问	不可访问
private	私有	可访问	不可访问

public修饰的成员，在任何地方都可以访问。类的内部，所有的成员都可以访问，如果不使用子类的情况下，protected和private 一样的。如果存在子类，则protected成员可以被访问。

定义类的语法：

```
class 类名{
[访问权限:]
    // 封装类成员的属性
    // 封装类成员的方法
};
```

【注意】类的定义是建议在全局区或头文件中。

如：定义学生类，包含学生的学号、姓名、成绩以及显示成绩、添加成绩的方法

```
class Student{
private:
    int sid;
    char name[32];
    float score;
public:
    void showScore(){
        cout << "sid=" << sid << ",score=" << score << endl;
    }
    void addScore(float n); // 只是声明
}

// 实现类中声明的方法（函数）
void Student::addScore(float n){
    score = n;
}
```

### 3.1.2 类的实例

类定义好之后，可以通过类，创建具体的对象，创建的类对象又称之为类的实例。

通过类创建的对象，对象即具有类的属性（数据）和方法。

语法：

```
类名 对象名；
```

如：

```
#include <iostream>
#include <cstring>

using namespace std;

class Student
{
public:
    int sid;
    char name[32];

private:
    // 如果成员变量初始化值时，编译时指定标准为 c++11
    // g++ xxx.cpp -std=c++11
    float score = 0; // 默认值是随机的，成员变量可以给定初始值

public:
    void showScore()
    {
        cout << "sid=" << sid << ",score=" << score << endl;
    }
    void addScore(float n); // 只是声明
};

// 实现类中声明的方法（函数）
void Student::addScore(float n)
{
    score = n;
}

int main(int argc, char const *argv[])
{
    Student s1;

    s1.sid = 1001;
    strcpy(s1.name, "disen");
    // s1.score = 100; 不能直接访问private成员
    s1.addScore(100); // 通过公开的方法修改private成员
    s1.showScore();

    return 0;
}
```

请设计一个 Person 类，Person 类具有 name 和 age 属性，提供初始化函数(Init)，并提供对 name 和 age 的读写函数(set, get)，但必须确保 age 的赋值在有效范围内(0-100),超出有效范围，则拒绝赋值，并提供方法输出姓名和年龄.(10 分钟)

### 3.1.3 课堂练习

请设计一个 Person 类，Person 类具有 name 和 age 属性，提供初始化函数(init)，并提供对 name 和 age 的读写函数(set, get)，但必须确保 age 的赋值在有效范围内(0-100),超出有效范围，则拒绝赋值，并提供方法输出姓名和年龄.(10 分钟)

```
#include <iostream>
#include <cstring>

using namespace std;
class Person
{
private:
    char name[32]; // 类成员变量
    int age;

public:
    void init(const char name[], int age)
    {
        // name是局部变量， 优先类成员变量
        // 为了区分类成员变量和局部变量时，可以使用隐藏的this指针
        // this指针代表是当前类的某一个对象（谁调用此方法，this就代表谁）
        // Person p1; p1.init(...) this代表是p1
        // Person p2; p2.init(...) this代表是p2
        strcpy(this->name, name);
        if (age <= 0 || age >= 100)
        {
            cout << "age invalid" << endl;
            return;
        }
        this->age = age;
    }

    void setName(const char name[])
    {
        strcpy(this->name, name);
    }
    const char *getName()
    {
        return name; // 访问是类成员变量
    }

    void setAge(int age)
    {
        if (age <= 0 || age >= 100)
        {
            cout << "age invalid" << endl;
            return;
        }
        this->age = age;
    }
    int getAge()
```



```

    {
        return age;
    }

    void show()
    {
        cout << "name=" << name << ", age=" << age << endl;
    }
};

int main(int argc, char const *argv[])
{
    Person p1;
    Person p2;

    p1.init("disen", 20);
    p2.init("jack", 19);
    p1.show();
    p2.show();

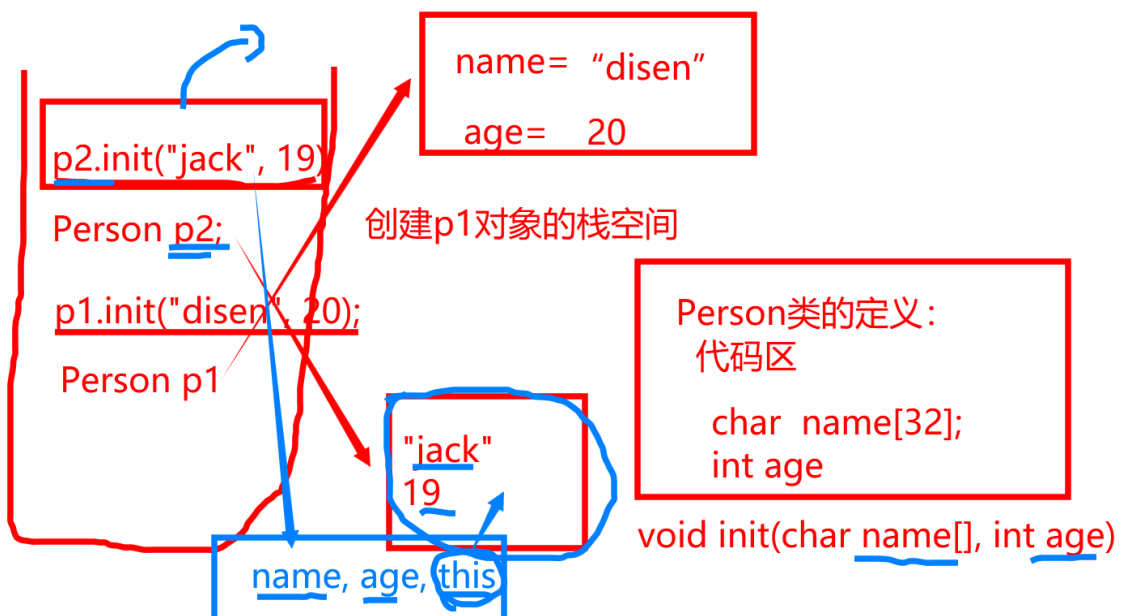
    cout << "-----" << endl;
    p1.setName("小马");
    p2.setName("小李子");
    p1.show();
    p2.show();
    return 0;
}

```

```

disen@qfxa:~/code2/day02$ ./a.out
name=disen, age=20
name=jack, age=19
-----
name=小马, age=20
name=小李子, age=19

```

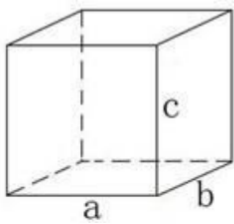


创建类对象时，在栈区分配内存空间，存放对象的属性（成员变量），当对象调用方法时，除了实参的数据之外，默认存在this指针，this指向调用的对象地址（内存空间），在方法中通过this指针可以操作对象自己的属性（成员变量）。

## 3.2 面向对象程序设计案例

### 3.2.1 设计立方体类与应用

设计立方体类(Cube)，求出立方体的面积( $2ab + 2ac + 2bc$ )和体积( $a * b * c$ )，分别用全局函数和成员函数判断两个立方体是否相等。



```
class Cube{
private:
    int a,b,c; // 长，宽，高
public:
    void init(int a, int b, int c){
        // this 代表某一个对象， this->a, 代表某个对象的属性a
        // this->a = a ; 将形参接收外部传入的实际数据，赋值给对象的属性a
        this->a = a;
        this->b = b;
        this->c = c;
    }
    int s(); // 求面积
    int v(); // 求体积
};

int Cube::s(){
    return 2*(a*b+a*c+b*c);
}

int Cube::v(){
    return a*b*c;
}
```

```
int main(){
    Cube c1, c2;
    c1.init(2, 3, 4);
    cout << "c1 s() = " << c1.s() << endl;

    c2.init(3, 4, 5);
    cout << "c2 s() = " << c2.s() << endl;

    return 0;
}
```

比较大小的函数：

## 1) 全局函数

```
bool equalCube(Cube &c1,Cube &c2){  
    // if(c1.a == c2.a && c1.b==c2.b && c1.c == c2.c)  
    return c1.S() == c2.S() && c1.V() == c2.V();  
}
```

## 2) 成员函数

```
class Cube{  
    ....  
public:  
    ...  
    bool eqaul(Cube &other){  
        return other.S() == S() && other.V() == V();  
    }  
}
```

完整的程序代码：

```
#include <iostream>  
  
using namespace std;  
  
class Cube  
{  
private:  
    int a, b, c; // 长, 宽, 高  
public:  
    void init(int a, int b, int c)  
    {  
        // this 代表某一个对象, this->a, 代表某个对象的属性a  
        // this->a = a ; 将形参接收外部传入的实际数据, 赋值给对象的属性a  
        this->a = a;  
        this->b = b;  
        this->c = c;  
    }  
    int S(); // 求面积  
    int V(); // 求体积  
    bool eqaul(Cube &other) // 当前对象和other对象比较  
    {  
        return other.S() == S() && other.V() == V();  
    }  
};  
  
int Cube::S()  
{  
    return 2 * (a * b + a * c + b * c);  
}  
  
int Cube::V()  
{  
    return a * b * c;  
}  
  
bool equalCube(Cube &c1, Cube &c2)
```

```

{
    // if(c1.a == c2.a && c1.b==c2.b && c1.c == c2.c)
    return c1.S() == c2.S() && c1.V() == c2.V();
}

int main(int argc, char const *argv[])
{
    Cube c1, c2;
    c1.init(2, 3, 4);
    cout << "c1 S() = " << c1.S() << endl;
    cout << "c1 V() = " << c1.V() << endl;

    c2.init(3, 4, 5);
    cout << "c2 S() = " << c2.S() << endl;
    cout << "c2 V() = " << c2.V() << endl;

    // cout << "c1 == c2 ? " << equalCube(c1, c2) << endl;
    cout << "c1 == c2 ? " << c1.equal(c2) << endl;
    Cube c3;
    c3.init(4, 5, 3);
    // cout << "c2 == c3 ? " << equalCube(c2, c3) << endl;
    cout << "c2 == c3 ? " << c3.equal(c2) << endl;

    return 0;
}

```

```

disen@qfxa:~/code2/day02$ g++ demo15.cpp
disen@qfxa:~/code2/day02$ ./a.out
c1 S() = 52
c1 V() = 24
c2 S() = 94
c2 V() = 60
c1 == c2 ? 0
c2 == c3 ? 1

```

### 3.2.2 点和圆的关系

设计一个圆形类（AdvCircle），和一个点类（Point），计算点和圆的关系。

假如圆心坐标为  $x_0, y_0$ , 半径为  $r$ , 点的坐标为  $x_1, y_1$ :

- 1) 点在圆上:  $(x_1 - x_0)(x_1 - x_0) + (y_1 - y_0)(y_1 - y_0) == r * r$
- 2) 点在圆内:  $(x_1 - x_0)(x_1 - x_0) + (y_1 - y_0)(y_1 - y_0) < r * r$
- 3) 点在圆外:  $(x_1 - x_0)(x_1 - x_0) + (y_1 - y_0)(y_1 - y_0) > r * r$

设计点类、圆类:

```

class Point{
public:
    int x, y;
}

```

```

class AdvCircle{
private:
    int r;
    Point o;

public:
    int init(int x0, int y0, int r){
        this->r = r;
        o.x=x0;
        o.y=y0;
    }
    int at(Point &point);
};

int AdvCircle::at(Point &point){
    int distance = (o.x - point.x)*(o.x - point.x) + (o.y-point.y)*(o.y-
point.y);
    int rr = r*r;
    if(distance == rr){
        return 1; // 在圆上
    }else if(distance < rr){
        return 2; // 在圆内
    }else{
        return 0; // 在圆外
    }
}

```

主函数:

```

int main(){
    AdvCircle c1;
    c1.init(0, 0, 2);

    Point p1;
    p1.x=2;
    p1.y=0;
    int flag = c1.at(p1);
    const char *ret = flag?(flag==1?"在圆上":"在圆内"):"在圆外";
    cout << "c1.at(p1)" << ret << endl;
}

```

完整的程序:

```

#include <iostream>

using namespace std;
class Point
{
public:
    int x, y;
};

class AdvCircle
{
private:
    int r;

```

```

    Point o;

public:
    int init(int x0, int y0, int r)
    {
        this->r = r;
        o.x = x0;
        o.y = y0;
    }
    int at(Point &point);
};

int AdvCircle::at(Point &point)
{
    int distance = (o.x - point.x) * (o.x - point.x) + (o.y - point.y) * (o.y -
point.y);
    int rr = r * r;
    if (distance == rr)
    {
        return 1; // 在圆上
    }
    else if (distance < rr)
    {
        return 2; // 在圆内
    }
    else
    {
        return 0; // 在圆外
    }
}

int main(int argc, char const *argv[])
{
    AdvCircle c1;
    c1.init(0, 0, 2);

    Point p1;
    p1.x = 2;
    p1.y = 3;
    int flag = c1.at(p1);
    const char *ret = flag ? (flag == 1 ? "在圆上" : "在圆内") : "在圆外";
    cout << "c1.at(p1)" << ret << endl;
    return 0;
}

```

```

disen@qfxa:~/code2/day02$ ./a.out
c1.at(p1)在圆外

```