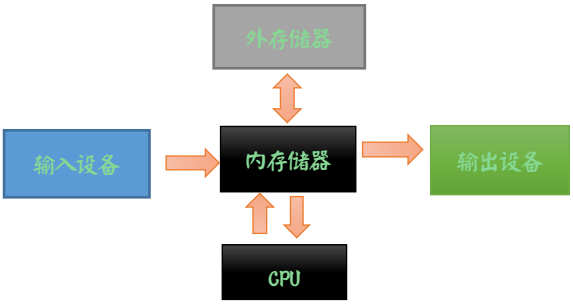
第 1 章　绪论

1.1　计算机系统简介

计算机硬件结构：



一个只有硬件的计算机称为裸机。
计算机能识别的是机器语言，机器语言指令由二进制 0 和 1 组成。

计算机指令系统：硬件能直接识别的语言(机器语言)集合。它是软件和硬件的主要界面。软件最终被转换成指令系统里的指令序列。

计算机软件：
1) 应用软件：常用的软件大部分是应用软件，如 IE、PS、QQ...
2) 系统软件：Windows、Linux/Unix、MacOS 等。
3) 中间件：提供系统软件和应用软件之间链接的软件。

软件由程序和文档文档构成。
计算机程序由由指令构成，是指令的序列，描述解决问题的方法和数据

1.2　计算机语言和程序设计方法的发展

1.2.1　计算机语言

1) 机器语言：由二进制指令构成，能被硬件识别，可以表示简单的操作，如加法、减法、数据移动等。
机器语言对于人类非常不友好，和人类语言之间存在巨大鸿沟。

2) 汇编语言：将机器指令映射为一些助记符，如 ADD、SUB、mov 等。其抽象层次低，需要考虑机器细节。

3) 高级语言：关键字、语句容易被理解；有含义的数据命名和算式；屏蔽了机器细节。

1.2.2　程序设计方法的发展

1) 面向过程：机器语言、汇编语言、高级语言
大型复杂的软件难以用面向过程的方式编写。
2) 面向对象：由面向对象的高级语言支持
一个系统由对象构成；对象之间通过消息进行通信。

1.3　面向对象的基本概念

对象与类

面向对象三特点：
1) 封装：屏蔽对象内部细节，只保留对外接口，安全性好；
2) 继承：代码复用，改造、扩展已有类形成新的类；
3) 多态：同样的消息作用于不同对象上有可能引起不同行为。

🕮 1.4 程序的开发过程

源程序：高级语言编写的待翻译的程序。
目标程序：源程序通过翻译程序加工后生成的机器语言程序。
可执行程序：连接目标程序及库中某些文件，生成的一个可执行文件。如 Windows 的.exe 文件。

三种不同类型的翻译程序：
1) 汇编程序：将汇编语言源程序翻译成目标程序。
2) 编译程序：将高级语言源程序翻译成目标程序。
3) 解释程序：将高级语言源程序翻译成机器指令，翻译边执行。
Java 是半编译半解释型语言，目的是为了跨平台。
C++是直接编译为本地机器语言代码。

C++程序的开发过程：
1) 算法与数据结构设计
2) 源程序编写
3) 编译
4) 连接
5) 测试
6) 调试

🕮 1.5 计算机中信息的表示和储存

计算机中信息：
1) 控制信息：一些指令
2) 数据信息：
   a) 数值信息：整数、浮点数
   b) 非数值信息：字符数据、逻辑数据

信息存储单位：比特(bit, b)、字节(byte, B)

补码的优点：
1) 0 的表示唯一
2) 符号位可作为数值参加运算
3) 补码运算结果仍是补码

补码计算规则：
1) 正整数原码、反码、补码都是自己。

2) 负整数补码 = 反码 +1
负整数反码：符号位 1 不变，其余各位取反。反码作为计算补码的中间码，本身没有什么用。
补码符号位不变，剩余取反 +1，就是原码。

小数的表示：
定点：小数点固定，约定在某个分界，两边分别是整数部分和小数部分，过时
浮点：计算机采用浮点方式表示小数。

$$N = M \times 2^E$$

E：称数 N 的阶码，反映该浮点数所表示的数据范围。
M：N 的尾数，其位数反映数据的精度。

字符常用编码：ASCII 码、汉字编码、Unicode、UTF-8 等。

🕮 1.6 C++开发工具
Visual Studio、Eclipse、Dev C++、GCC 等。

第 2 章 C++简单程序设计
🕮 2.1 C++语言概述
由 C 语言发展而来，最初被称为"带类的 C"。
1998 年被 ISO 批准为国际标准。

C++的特点：
1) 兼容 C，支持面向过程
2) 支持面向对象
3) 支持泛型编程

```cpp
#include <iostream> // 包含头文件
using namespace std;
int main(int argc, char const *argv[]){
    // cout: 标准输出流; <<: 插入运算符; cout 将后面字符串送到显示器上
    cout << "hello world!" << endl; // endl: 行结束
    cout << "C++大坑!" << endl;
    return 0;
}
```

结果：

```
hikari@ubuntu:~/cpp_test$ g++ hello.cpp -o hello.out
hikari@ubuntu:~/cpp_test$ ./hello.out
hello world!
C++大坑!
```

1) `include`：编译预处理命令，在编译之前找到 iostream 文件，将其内容全部粘贴到 `include` 语句所在之处。iostream 头文件包含了 cout 的声明。
2) `namespace`：命名空间可以避免重名。std 是标准库命名空间。直接使用 cout 应该 `std::cout`，添加 `using namespace std;`使用 std 中的对象可以不带 std 了。

🕮 2.2 基本数据类型、常量、变量
① 整数类型

1) 基本整数类型：**int**
2) 按符号分：**signed**、**unsigned**
3) 按数据范围分：**short**、**long**、**long long**

② 字符类型(**char**)：容纳单个字符的编码，实质存储也是整数。
③ 浮点数类型：**float**、**double**、**long double**。
④ 字符串类型：有字符串常量，基本类型没有字符串变量。
1) 采用字符数组存储字符串(C 风格的字符串)，不建议使用
2) 标准库的 String 类(C++风格的字符串)
⑤ 布尔类型(**bool**)：**true**、**false**

```cpp
#include <iostream>
using namespace std;
int main(int argc, char const *argv[]){
    const double PI(3.14159);
    int r;
    cout << "r = " << r << endl;
    cout << "请输入半径：";
    cin >> r;
    cout << "r = " << r << endl;
    cout << "半径为" << r << "的圆面积为:" << (int)(PI*r*r*100)/100.0 << endl;
    return 0;
}
```

结果：

```
hikari@ubuntu:~/cpp_test$ g++ 2.cpp -o 2.out
hikari@ubuntu:~/cpp_test$ ./2.out
r = 0
请输入半径：2
r = 2
半径为2的圆面积为:12.56
```

🕮 2.3 运算与表达式

① 算术运算符
② 赋值运算符，复合赋值运算符
③ 关系运算符
④ 逻辑运算符
⑤ 逗号运算符
多个表达式可以用逗号分开，其中用逗号分开的表达式的值分别计算，但整个表达式的值是最后一个表达式的值。
逗号运算符优先级比赋值运算符还要低。

```cpp
    int n;
    cout << (n = 3 + 4, 5 + 6) << endl;
    cout << "n = " << n << endl;

    int a, b, c;
    a = (b = 3, (c = b + 4) + 5);
    cout << "a = " << a << ", b = " << b << ", c = " << c << endl;
```

结果：

```
11
n = 7
a = 12, b = 3, c = 7
```

⑥ 三目运算符，条件表达式

```
int a = 10, b = 15;
int c = a > b ? a : b;
cout << "max(" << a << ", " << b <<") = " << c << endl;
```

⑦ sizeof 运算

后面可加类型名、变量、表达式等，结果是该类型所占多少字节。

⑧ 位运算符

按位与&、按位或|、按位异或^、取反~、左移<<、右移>>

混合运算时的类型转换：

1) 隐式转换：范围小的数据转换为范围大的。

范围大的数据赋值给范围小的变量，会造成精度损失。

2) 显式转换：强制类型转换

```
char a = 'a';
// 3 种强制类型转换完全等价
cout << int(a) << endl;
cout << (int)a << endl;
cout << static_cast<int>(a) << endl;
```

类型转换操作符有：const_cast、dynamic_cast、reinterpret_cast、static_cast。

🕮 2.4 数据的输入和输出

① IO 流

C++中将数据从一个对象到另一个对象的流动称为流(stream)，流在使用前要被建立，使用后要被删除。

数据的输入输出通过 IO 流实现。cin 和 cout 是预定义的流类对象。cin 处理标准输入，即键盘输入；cout 处理标准输出，即屏幕输出。

② 预定义的插入符和提取符

<<是预定义的插入符，作用在流类对象 cout 上可实现向标准输出设备输出。

提取符>>可连续写多个，每个后面跟一个表达式，该表达式通常是用于存放输入值的变量：cin >> a >> b;

③ 常用 IO 流类库操纵符(manipulator)

dec	数值采用十进制表示
hex	十六进制
oct	八进制
ws	提取空白符
endl	插入换行符，并刷新流
ends	插入空字符
setprecision(int)	设置浮点数的小数位数(包括小数点)
setw(int)	设置域宽

```
#include <iostream>
#include <iomanip> // io 操纵符
using namespace std;
int main(int argc, char const *argv[]){
```

```
    cout << "***" <<setw(5) << setprecision(3) << 3.14159265358 << "***" << endl;
    return 0;
}
```

结果：

```
*** 3.14***
```

🕮 2.5 选择结构

🕮 2.6 循环结构

🕮 2.7 自定义类型

❁ 2.7.1 类型别名：为已有类型另外命名

1) typedef

```
    typedef int Length;
    Length a = 10;
```

2) using

```
    using length = int; // C++11 特性
    length l = 1;
    cout << l << endl;
```

❁ 2.7.2 枚举类型

① 不限定作用域枚举类型：enum 枚举类型名字 {变量值列表};

```
enum Week {SUN, MON, TUE, WED, THU, FRI, SAT};
enum Color {RED, GREEN=8, BLUE};
```

1) 枚举元素是常量，不能对其赋值
2) 枚举元素有默认值，依次为 0,1,2,...
3) 也可以在声明时另行指定枚举元素的值
4) 枚举值可以进行关系运算
5) 整数值不能直接赋值给枚举类型，需要强制类型转换
6) 枚举值可以赋值给整型变量

② 限定作用域的 enum 类

❁ 2.7.3 auto 类型和 decltype 类型

1) auto：编译器通过初始值自动推断变量的类型
2) decltype：定义一个变量与某一表达式的类型相同，但不使用其值

```
#include <iostream>
using namespace std;

int main(int argc, char const *argv[]){
    // auto 与 decltype 是 C++11 的新关键字，编译时需要指明 std=c++11
    double a = 1;
    decltype(a) b = 2; // 声明 b 类型与 a 一样,值为 2
    cout << "b = " << b << ", size = " << sizeof b << endl;
    auto c = a + b; // c 的类型由 a+b 决定
    cout << "c = " << c << ", size = " << sizeof(c) << endl;
    return 0;
}
```

结果：

```
hikari@ubuntu:~/cpp_test$ g++ -std=c++11 3.cpp && ./a.out
b = 2, size = 8
c = 3, size = 8
```

第 3 章　函数

🕮　3.1　函数定义

函数：定义好的功能模块
定义函数：将一个模块的算法用程序语言描述
函数的参数与返回值

🕮　3.2　函数调用

调用函数前需要先声明函数原型，因为函数的定义和调用往往不在一个程序，或定义在调用之后。

例 1：定义 pow()函数

```cpp
#include <iostream>
using namespace std;

// 求 x 的 n 次方
double pow(double x, int n){
    double ret = 1;
    for (int i = 1; i <= n; i++){
        ret *= x;
    }
    return ret;
}

int main(int argc, char const *argv[]){
    cout << "5^8 = " << pow(5, 8) << endl;
    return 0;
}
```

结果：

```
5^8 = 390625
```

例 2：计算 π

$$\pi = 16\arctan\left(\frac{1}{5}\right) - 4\arctan\left(\frac{1}{239}\right)$$

$$\arctan x = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \cdots$$

```cpp
double abs(double x){
    return x >= 0 ? x : -x;
}

double arctan(double x){
    int i = 1;
    double square = pow(x, 2);
    double ret = 0;
    double xi = x;
    double e = x;
    while (abs(e) > 1e-15){ // 如果某项绝对值足够小，后面直接忽略
        ret += e;
        xi *= -square;
        i += 2;
        e = xi/i;
    }
    return ret;
```

```
}

double cal_pi(){
    return 16 * arctan(1/5.0) - 4 * arctan(1/239.0);
}

int main(int argc, char const *argv[]){
    cout << "pi = " << cal_pi() << endl;
    return 0;
}
```

结果：

```
pi = 3.14159
```

例 3：输出 11~99 之间的数 m，满足 m、m^2、m^3 都是回文数

```
#include <iostream>
using namespace std;

// 判断 n 是否是回文数
bool is_palindrome(unsigned n){
    unsigned n0 = n;
    unsigned ret = 0;
    while (n0){
        ret = ret*10 + n0 % 10;
        n0 /= 10;
    }
    return ret == n;
}

int main(int argc, char const *argv[]){
    for (unsigned i = 11; i < 1000; i++){
        unsigned i2 = i*i;
        unsigned i3 = i2*i;
        if (is_palindrome(i) && is_palindrome(i2) && is_palindrome(i3)){
            cout << i << ", " << i2 << ", " << i3 << endl;
        }
    }
    return 0;
}
```

结果：

```
11, 121, 1331
101, 10201, 1030301
111, 12321, 1367631
```

例 4：cstdlib 两个用于生成伪随机数的函数

```
void srand(unsigned seed);
```

参数 seed 是 rand()函数的种子，初始化 rand()起始值

```
int rand(void);
```

从指定的 seed 开始，返回一个[seed, RAND_MAX)间的随机整数

指定相同的 seed，每次随机数序列都一样。

可以指定 seed 为当前系统流逝时间(时间戳，单位秒)

```
#include <iostream>
#include <cstdlib>
#include <ctime>
```

```cpp
using namespace std;

int randint(int a, int b){
    return (rand() % (b - a)) + a ;
}

int main(int argc, char const *argv[]){
    time_t t = time(0); // 时间戳
    cout << t << endl;
    // srand()函数为rand()函数生成随机数种子
    srand((unsigned)t);
    for(int i = 0; i < 10; i++){
        cout << randint(-10, 10) <<", ";
    }
    return 0;
}
```

结果：

```
1540381102
3, 0, 9, -4, 7, 8, -4, -10, 7, 5,
```

🕮 3.3 嵌套与递归

例：汉诺塔

三根针 A、B、C，A 上有 N 个盘子，大的在下，小的在上。要求把 N 个盘子从 A 借助 B 移动到 C，每次只能移动一个盘子，移动过程中需要保持大盘在下，小盘在上。

```cpp
#include <iostream>
using namespace std;

void move(int n, char src, char dest){
    cout << "move " << n << " : " << src << " --> " << dest << endl;
}

void hanoi(int n, char src, char middle, char dest){
    if (n == 1){
        move(n, src, dest);
    }else{
        hanoi(n-1, src, dest, middle);
        move(n, src, dest);
        hanoi(n-1, middle, src, dest);
    }
}

int main(int argc, char const *argv[]){
    hanoi(3, 'A', 'B', 'C');
    return 0;
}
```

结果：

```
move 1 : A --> C
move 2 : A --> B
move 1 : C --> B
move 3 : A --> C
move 1 : B --> A
move 2 : B --> C
move 1 : A --> C
```

🕮 3.4 函数的参数传递

函数在调用时才分配形参存储单元；实参可以是常量、变量或表达式。

实参类型需要和形参类型一致。若不一致，尝试隐式转换，不行编译器就报错。
值传递是传递参数值，即*单向传递*。
引用传递可以实现*双向传递*。
类的实例对象可能很大，如果直接传对象，开销会很大，此时选择传引用。
传引用作参数可以保障实参数据的安全。

🕮 3.5 引用类型

引用&是标识符的别名。
定义一个引用时，必须同时对它初始化，使它指向一个已存在的对象。
一个引用被初始化后，不能改为指向其他对象。

引用可以作为形参。
例：交换两个整数(引用传递)

```cpp
#include <iostream>
using namespace std;

void swap(int &a, int &b){ // 传入 a 和 b 的引用
    int t = a;
    a = b;
    b = t;
}

int main(int argc, char const *argv[]){
    int x = 5, y = 8;
    cout << "x = " << x << ", y = " << y << endl;
    swap(x, y);
    cout << "x = " << x << ", y = " << y << endl;
    return 0;
}
```

结果：

```
x = 5, y = 8
x = 8, y = 5
```

🕮 3.6 含有可变参数的函数

C++新标准提供两种主要方法：
1) 如果所有实参类型相同，可以传递一个名为 initializer_list 的标准库类型
initializer_list 是一种标准库类型，用于表示某种特定类型值的数组，其定义在同名的头文件中。
其中元素永远是常量值，无法改变其中元素的值。
2) 如果实参类型不同，可以编写可变参数的模板。

🕮 3.7 内联函数

关键字 `inline`
对于简单的函数，编译时在调用处用函数体进行替换，节省参数传递、控制转移(转子函数再返回)等开销，提高运行效率。

注意：
1) 内联函数体内不能有循环语句和 `switch` 语句。
2) 内联函数的定义必须出现在内联函数第一次被调用之前。

3) 对内联函数不能进行异常接口声明。

`inline` 是用户对编译器的建议，一些好的编译器也会自己判断要不要将某个函数定义成内联函数。

🕮 3.8 constexpr 函数

`constexpr` 修饰的函数在其所有参赛都是 `constexpr` 时一定返回 `constexpr`。

```
constexpr int get_num(){
    return 10;
}
int main(int argc, char const *argv[]){
    constexpr int a = get_num();
    cout << "a = " << a << endl;
    return 0;
}
```

🕮 3.9 带默认参数值的函数

和 Python 的默认参数几乎一样。

🕮 3.10 函数重载

和 Java 的函数重载几乎一样。

🕮 3.11 C++系统函数

C++系统库提供了几百个函数供直接调用，如 sqrt()、abs()等。
需要包含相应的头文件，如 cmath

第 4 章 类与对象

🕮 4.1 面向对象基本特点

1) 抽象：对同一类对象的共同属性和行为进行概括，形成类。
2) 封装：将抽象出的数据、代码封装在一起，形成类。
增加安全性，简化编程，使用时不必了解实现细节。只需要通过外部接口，以特定的访问权限，使用类的成员。
3) 继承：在已有类的基础上，进行扩展形成新的类。
4) 多态：同一名称，不同的功能实现方式。达到行为标识统一，减少程序中标识符的个数。

🕮 4.2 类和对象

例：

```
#include <iostream>
#include <iomanip>
using namespace std;

class Clock{
    public:
        void setTime(int h=0, int m=0, int s=0);
        void showTime();
    private:
        int hour, minute, second;
};
```

```cpp
// 成员函数可以写在类外,但类内部必须有函数声明;也可以把简单函数作为内联函数写在类内部
void Clock::setTime(int h, int m, int s){
    hour = h;
    minute = m;
    second = s;
}

void Clock::showTime(){
    cout << setfill('0') << setw(2) << hour << ":" << setfill('0') << setw(2) <<
minute << ":" << setfill('0') << setw(2) << second << endl;
}

int main(int argc, char const *argv[]){
    Clock c;
    c.setTime(7, 30);
    c.showTime();
    return 0;
}
```

结果：

```
07:30:00
```

🕮 4.3 构造函数

例 1：有参构造和无参构造

```cpp
#include <iostream>
#include <iomanip>
using namespace std;

class Clock{
    public:
        Clock(int h, int m, int s); // 有参数的构造函数
        Clock(); // 无参构造
        void setTime(int h=0, int m=0, int s=0);
        void showTime();
    private:
        int hour, minute, second;
};
// 构造函数的实现，初始化列表
Clock::Clock(int h, int m, int s):hour(h), minute(m), second(s){}
// 默认构造函数，初始化全部设为 0
Clock::Clock():hour(0), minute(0), second(0){}
// 两个成员函数与之前相同

int main(int argc, char const *argv[]){
    Clock c(12, 30, 22);
    c.showTime();
    c.setTime(7, 30);
    c.showTime();
    cout << "--------" << endl;
    Clock c2;
    c2.showTime();
    return 0;
}
```

结果：

```
12:30:22
07:30:00
--------
00:00:00
```

委托构造函数(C++11)

```
// 构造函数的实现，初始化列表
Clock::Clock(int h, int m, int s):hour(h), minute(m), second(s){}
// 无参默认构造函数，调用有参构造，初始化全部设为 0
Clock::Clock():Clock(0, 0, 0){} // 委托构造函数
```

复制构造函数

复制构造函数是特殊的构造函数，形参是本类对象的引用，作用是用一个已存在的对象初始化同类型的新对象。

复制构造函数被调用的 3 种情况：

1) 定义一个对象时，以本类另一个对象作为初始值；
2) 如果函数形参是类的对象，调用函数时，将使用实参对象初始化形参对象；
3) 如果函数的返回值是类的对象，函数 return 语句的对象初始化一个临时无名对象，传递给主调函数。

如果没有声明复制构造函数，编译器自己生成一个默认的复杂构造函数，其功能是用初始值对象的每个数据成员，初始化新对象。
默认的复制构造函数使用的是浅层复制。

如果不希望对象被复制构造：

1) C++98：复制构造函数声明为 `private`，并不提供函数实现
2) C++11：用`=delete` 指示编译器不生成默认复制构造函数。

例 2：复制构造函数

```
#include <iostream>
using namespace std;

class Point{
    private:
        int x, y;
    public:
        Point(int x=0, int y=0){
            this->x = x;
            this->y = y;
        }
        Point(const Point& p){
            x = p.x;
            y = p.y;
            cout << "calling copy constructor..." <<endl;
        }
        void setX(int x){
            this->x = x;
        }
        void setY(int y){
            this->y = y;
        }
        void showPoint(){
            cout << "(" << x << ", " << y << ")" << endl;
        }
};
```

```cpp
void f1(Point a){
    a.showPoint();
}
Point f2(){
    Point a(1);
    return a;
}

int main(int argc, char const *argv[]){
    Point a(12, 34);
    Point b = a; // 1. 用 a 初始化 b
    b.showPoint();
    f1(b); // 2. 对象 b 作为 f1()的实参
    b = f2(); // 3. 函数 f2()返回值是 Point 对象,赋值给 b
    b.showPoint();
    return 0;
}
```

结果：

```
hikari@ubuntu:~/cpp_test$ g++ 10.cpp && ./a.out
calling copy constructor...
(12, 34)
calling copy constructor...
(12, 34)
(1, 0)
```

预期应该调用 3 次复制构造函数，实际只调用了两次。函数返回类对象没有调用，原因是 G++使用了返回值优化 RVO(return value optimization)。

🕮 4.4 析构函数

完成对象被删除前的一些清理工作。

如果没有声明析构函数，编译器自动生成默认的析构函数，其函数体为空。

上例中添加析构函数：

```cpp
class Point{
    private:
        int x, y;
    public:
        Point(int x=0, int y=0){
            this->x = x;
            this->y = y;
        }
        ~Point(){ // 析构函数
            cout << "(" << x << ", " << y << ") delete..." << endl;
        }
        // ...
};
```

结果：

```
calling copy constructor...
(12, 34)
calling copy constructor...
(12, 34)
(12, 34) delete...
(1, 0) delete...
(1, 0)
(1, 0) delete...
(12, 34) delete...
```

🕮 4.5 类的组合

类中的成员是其他类的对象，可以在已有抽象的基础上实现更复杂的抽象。

例：

```
#include <iostream>
#include <cmath>
using namespace std;

class Point{
    private:
        int x, y;

    public: // 全写成内联的了...写在外面看着蛋疼...
        Point(int x=0, int y=0){
            this->x = x;
            this->y = y;
        }
        Point(const Point& p){
            x = p.x;
            y = p.y;
            cout << "calling copy constructor of Point..." <<endl;
        }
        void setX(int x){this->x = x;}
        int getX(){return x;}
        void setY(int y){this->y = y;}
        int getY(){return y;}
        void showPoint(){
            cout << "(" << x << ", " << y << ")" << endl;
        }
};

class Line{
    private:
        Point a, b;
        double length;
    public:
        Line(){};
        Line(Point a, Point b){
            this->a = a;
            this->b = b;
            double x = static_cast<double> (a.getX() - b.getX());
            double y = static_cast<double> (a.getY() - b.getY());
            this->length = sqrt(x*x + y*y);
            cout << "calling constructor of Line..." << endl;
        }
        Line(Line& line){
            this->a = line.a;
            this->b = line.b;
            this->length = line.length;
            cout << "calling copy constructor of Line..." << endl;
        }
        double getLength(){return length;}
};

int main(int argc, char const *argv[]){
    Point a(1, 1), b(4, 5);
```

```
    Line line(a, b);
    cout << "----------" << endl;
    Line l2(line); // 复制构造函数建立新对象
    cout << "line = " << line.getLength() << "\nl2 = " << l2.getLength() << endl;
    return 0;
}
```

结果：

```
calling copy constructor of Point...
calling copy constructor of Point...
calling constructor of Line...
----------
calling copy constructor of Line...
line = 5
l2 = 5
```

前向引用声明

类应该先声明后使用。如果需要在某个类的声明之前引用该类，则应进行前向引用声明。它只是为程序引入一个标识符，但具体声明在其他地方。

```
class B; // 前向引用声明
class A{
    public:
        void f(B b);
};

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

使用注意：

1) 在提供一个完整的类声明前，不能声明该类的对象，也不能在内联成员函数中使用该类对象。

2) 当使用前向引用声明时，只能使用被声明的符号，不能涉及类的任何细节。

// 意思就是没什么卵用吧...

🕮 4.6 UML

UML 是可视化、面向对象的建模语言，此处只使用 UML 一些符号。

UML 三个基本部分：

1) 事物 (Things)

2) 关系 (Relationships)

3) 图 (Diagrams)

🕮 4.7 结构体和联合体

❁ 4.7.1 结构体

C++的结构体已经和 C 语言的结构体不一样了，是一种特殊形态的类。

与类的唯一区别：

类的默认访问权限是 `private`；结构体的默认访问权限是 `public`

结构体用处：

1) 定义主要用来保存数据、而没有什么操作的类型

2) 将数据成员设为公有，使用用结构体更方便
3) 与 C 语言保持兼容

使用 struct 关键字定义结构体。

```
#include <iostream>
#include <iomanip>
#include <string>
using namespace std;

struct Student{
    int num;
    string name;
    int age;
    char gender;
};

int main(int argc, char const *argv[]){
    Student hikari = {1, "hikari", 26, 'M'};
    cout << hikari.name << endl;
    return 0;
}
```

4.7.2 联合体

使用 union 关键字联合体。特点是所有成员共用同一组内存单元，任何两个成员不会同时有效。

```
union Score{ // 共用内存空间，4 字节
    char grade; // 分数等级
    bool is_pass; // 是否及格
    int percent; // 百分制分数
};
```

无名联合

```
static union{ // 定义两个变量共用内存空间
    int i;
    double d;
};

int main(int argc, char const *argv[]){
    i = 12;
    cout << "i = " << i << "\nd = " <<d << endl;
    d = 34;
    cout << "i = " << i << "\nd = " <<d << endl;
    return 0;
}
```

结果：

```
i = 12
d = 5.92879e-323
i = 0
d = 34
```

后面给 d 赋值，原先 i 的值就不见了。

例：

```
#include <iostream>
#include <string>
```

```cpp
using namespace std;

class ExamInfo {
private:
    string name; // 课程名称
    enum { GRADE, PASS, PERCENTAGE } mode; // 计分方式
    union {
        char grade; // 等级制的成绩
        bool pass; // 只记是否通过课程的成绩
        int percent; // 百分制的成绩
    };
public:
    // 三种构造函数，分别用等级、是否通过和百分初始化
    ExamInfo(string name, char grade):name(name), mode(GRADE), grade(grade){}
    ExamInfo(string name, bool pass):name(name), mode(PASS), pass(pass){}
    ExamInfo(string name, int percent):name(name), mode(PERCENTAGE),
percent(percent){}
    void show();
};

void ExamInfo::show() {
    cout << name << ": ";
    switch (mode) {
      case GRADE: cout << grade;  break;
      case PASS: cout << (pass ? "PASS" : "FAIL"); break;
      case PERCENTAGE: cout << percent; break;
    }
    cout << endl;
}

int main() {
    ExamInfo course1("English", 'B');
    ExamInfo course2("Calculus", true);
    ExamInfo course3("C++ Programming", 85);
    course1.show();
    course2.show();
    course3.show();
    return 0;
}
```

结果：

```
English: B
Calculus: PASS
C++ Programming: 85
```

🕮 4.8 枚举类

从 C 语言继承来的枚举类型，可以自动隐式转为整数类型，类型定义不严格。C++11 推出的枚举类，也叫强类型枚举。

```cpp
enum class 枚举类型名: 底层类型 {枚举值列表}
```

不指定底层类型默认 int

枚举类优势：

1) 强作用域，其作用域限制在枚举类中。使用 Type 的枚举值 General：

Type::General

因此不同枚举类的枚举值可以重名。

2) 转换限制，枚举类对象不可以与整数隐式转换。
3) 可以指定底层类型。

```
enum class Side{ Right, Left };
enum class Thing{ Wrong, Right };  // 不冲突

int main() {
    Side s = Side::Right;
    Thing w = Thing::Wrong;
    // comparison of two values with different enumeration types
    cout << (s == w) << endl;  // 编译错误，无法直接比较不同枚举类
    return 0;
}
```

第 5 章　数据共享与保护

🕮　5.1　标识符的作用域与可见性

作用域分类:
1) 函数原型作用域：函数的形参表，函数原型声明：int f(int n);
2) 局部作用域(块作用域)：函数的形参，在块中声明的标识符{int n;}
3) 类作用域：范围包括类体和成员函数体
4) 文件作用域：始于声明点，终于文件尾
5) 命名空间作用域

可见性：表示从内层作用域向外层作用域能看见什么
如果内层作用域定义了和外层同名的标识符，则外层作用域的同名标识符在内层不可见。

🕮　5.2　对象的生存期

静态生存期：和程序的运行期相同，文件作用域中声明的对象具有静态生存期。
函数内部使用 **static** 声明静态生存期对象。

动态生存期：始于程序执行到声明点，终于其作用域结束处。
块作用域声明的，没有 **static** 修饰的对象。

```
#include <iostream>
using namespace std;

int A = 5;
void f(){
    static int b = 2;
    static int c; // 静态局部变量,只初始化 1 次，默认初始化为 0，具有全局生存期
    int d = 10;
    cout <<"A = " << A << ", b = " << b << ", c = " << c << ", d = " << d << endl;
    A *= 2;
    c = b;
    b += A;
    d += A;
}

int main(int argc, char const *argv[]){
```

```
    for(int i = 0; i < 3; i++){
        A++;
        f();
    }
    return 0;
}
```

结果：

```
A = 6, b = 2, c = 0, d = 10
A = 13, b = 14, c = 2, d = 10
A = 27, b = 40, c = 14, d = 10
```

🕮 5.3 类的静态成员

`static` 修饰类成员，为该类所有对象共享，静态数据成员具有静态生存期。必须在类外定义和初始化，用`::`指明所属的类。

静态属性和方法。

```
#include <iostream>
using namespace std;

class Point{
    private:
    int x, y;
    static int COUNT; // 非 const 静态值初始化必须在类外

    public:
    Point(int x=0, int y=0){ // 构造函数
        this->x = x;
        this->y = y;
        COUNT++;
    }
    ~Point(){ // 析构函数
        cout << "(" << x << ", " << y << ") delete..." << endl;
        COUNT--;
    }
    Point(const Point& p){ // copy 构造函数
        x = p.x;
        y = p.y;
        cout << "calling copy constructor of Point..." <<endl;
        COUNT++;
    }
    // ...
    static int showCount(){ // 静态方法
        cout << "Point Object Count: " << COUNT << endl;
    }
};
int Point::COUNT = 0; // 静态属性初始化在外部
int main(int argc, char const *argv[]){
    Point a(12, 34);
    a.showCount();
    Point b = a;
    b.showPoint();
    b.showCount(); // 对象调用静态方法
    Point::showCount(); // 类名调用静态方法
    return 0;
}
```

结果：

```
Point Object Count: 1
calling copy constructor of Point...
(12, 34)
Point Object Count: 2
Point Object Count: 2
(12, 34) delete...
(12, 34) delete...
```

🕮 5.4 类的友元

友元是 C++提供的破坏数据封装和数据隐藏的机制。通过一个模块声明为另一个模块的友元，一个模块能引用到另一模块本是隐藏的信息。**慎用！**

*友元函数*是在类中使用 **friend** 声明的非成员函数，可以通过对象访问 **private** 和 **protected** 成员。
友元类：若 A 类为 B 类的友元，则 A 类所有成员都可以访问对方的私有成员。友元关系是单向的。

```cpp
#include <iostream>
#include <cmath>
using namespace std;

class Point{
    public:
        // ...
        // 友元函数，使用 const 设为只读
        friend double distance(const Point& a, const Point& b);
};

double distance(const Point& a, const Point& b){
    double x = static_cast<double> (a.x - b.x);
    double y = static_cast<double> (a.y - b.y);
    return sqrt(x*x + y*y);;
}

int main(int argc, char const *argv[]){
    Point a(12, 34);
    Point b(15, 38);
    cout << "distance is " << distance(a, b) << endl;
    return 0;
}
```

结果：

```
distance is 5
(15, 38) delete...
(12, 34) delete...
```

🕮 5.5 共享数据的保护

常类型(**const** 修饰)
1) 常对象：必须进行初始化，不能被更新

2) 常成员：常数据成员和函数成员
常成员函数不更新对象的属性。`const` 可被用于**函数重载**。
3) 常引用：被引用的对象不能更新
友元函数中用**常引用**做参数，既能获得较高执行效率，又能保证实参的安全性。
4) 常数组
5) 常指针

🕮 5.6 多文件结构和预编译命令

C++程序一般组织结构：
一个工程可分为多个源文件，如：
1) 类声明文件(.h 文件)
2) 类实现文件(.cpp 文件)
3) 类的使用文件(main()所在的.cpp 文件)
利用工程组合各个文件。

比如之前写的 Point 类，其定义放在 Point.h 文件；具体类的实现放在 Point.cpp 文件，需要在 Point.cpp 文件#include "Point.h"头文件。

主函数所在文件，也要#include "Point.h"。
主函数所在文件是使用者创建，而类实现可能由他人完成。

外部变量：文件作用域定义的变量，默认都是外部变量。
其他文件如需使用，需要用 `extern` 关键字声明。

外部函数：类外部定义的函数，都具有文件作用域，可以在不同的编译单元被调用，只要在调用之前进行引用性声明(声明函数原型)即可。

匿名空间定义的变量和函数，都不会暴露给其他编译单元。

```
namespace { // 匿名命名空间
    int n;
    void f(){
        cout << "hello!" << endl;
    }
}
```

标准 C++库是一个极为灵活并可扩展的可重用的软件模块的集合。
标准 C++类与组件在逻辑上分为 6 类：
1) 输入/输出类
2) 容器类和抽象数据类型
3) 存储管理类
4) 算法
5) 错误处理
6) 运行环境支持

编译预处理
1) `#include` 包含命令

将一个源文件嵌入到当前文件中该点处

`#include<xxx>`：按标准方式搜索，位于 C++系统目录的 include 子目录下

`#include "xxx"`：首先在当前目录搜索，若没有，再按标准方式搜索

2) `#define` 宏定义指令

定义符号常量，很多情况被 `const` 取代

定义带参数的宏，已被内联函数取代

3) `#undef`

删除由`#define` 定义的宏，使之不再起作用

4) 条件编译指令：`#if`、`#elif`、`#else`、`#endif`

```
#if 常量表达式 1
    程序正文 1 // 当常量表达式 1 非零时编译
#elif 常量表达式 2
    程序正文 2 // 当常量表达式 2 非零时编译
#else
    程序正文 3 // 其他情况时编译
#endif
```

```
// 如果标识符经#define 定义过,且未删除,编译程序段 1
#ifdef 标识符 // #ifndef...#define...更常用
    程序段 1
#else
    程序段 2
#endif
```

`#ifndef...#define...`避免重复 `include` 包含某个头文件，造成某些变量被重复定义的问题。

第 6 章　数组、指针、字符串

🕮　6.1　数组的定义与初始化

❁　6.1.1　一维数组的存储

数组元素在内存中顺次存放，它们的地址是连续的。

数组名字是数组首元素内存地址；数组名是一个**地址类型**常量，不能被赋值

❁　6.1.2　一维数组初始化

1) 列出全部元素的初始值

```
static int arr[5] = {1, 2, 3, 4, 5};
```

2) 可以只给部分元素指定初始值，其余默认为 0

```
static int arr[10] = {1, 2, 3, 4, 5};
```

3) 列出全部数组元素初始值，可以不指定数组长度

```
static int arr[] = {1, 2, 3, 4, 5};
```

❁　6.1.3　二维数组的存储

如 int a[3][4];

$$a\begin{cases} a[0]--a[0][0],a[0][1],a[0][2],a[0][3] \\ a[1]--a[1][0],a[1][1],a[1][2],a[1][3] \\ a[2]--a[2][0],a[2][1],a[2][2],a[2][3] \end{cases}$$

a, a[0], a[1], a[2]都是地址，且 3 个一维数组(一行)连续存储。

6.1.4 二维数组初始化

1) 将所有元素写在{}中，按顺序初始化

```
static int arr[3][4] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};
```

2) 分行列出二维数组元素的初始值

```
static int arr[3][4] = {(1, 2, 3, 4), (5, 6, 7, 8), (9, 10, 11, 12)};
```

3) 只对部分元素初始化

```
static int arr[3][4] = {(1), (7, 8), (9, 10, 11)};
```

4) 列出全部初始值，第 1 维下标个数可以省略

```
static int arr[][4] = {(1, 2, 3, 4), (5, 6, 7, 8), (9, 10, 11, 12)};
```

如果不作任何初始化，局部作用域的非静态数组中会存在垃圾数据，static 数组中的数据默认初始化为 0。
如果只对部分元素初始化，剩余未显式初始化的元素自动初始化为 0。

例：斐波那契数列

```
#include <iostream>
using namespace std;

int main(int argc, char const *argv[]){
    const int num = 20;
    int fib[num] = {0, 1};
    for (int i = 2; i < num; i++)
        fib[i] = fib[i - 1] + fib[i - 2];
    for (int i = 0; i < num; i++){
        if (i % 5 == 0) //每行 5 个数
            cout << endl;
        cout.width(12); //输出宽度 12
        cout << fib[i];
    }
    return 0;
}
```

结果：

```
           0           1           1           2           3
           5           8          13          21          34
          55          89         144         233         377
         610         987        1597        2584        4181
```

6.2 数组作为函数的参数

数组名作为参数，传的是数组的首地址，因此函数可能会修改数组。使用 `const` 修饰可以保证数组传入不被函数修改。

6.3 对象数组

6.4 基于范围的 for 循环

基于指针：

```
    int arr[] = {1, 2, 3, 4, 5};
    for (int *p = arr; p < arr + sizeof(arr) / sizeof(int); p++)
        cout << *p << " ";
```

基于范围：

```
int arr[] = {1, 2, 3, 4, 5};
for (int &e : arr)
    cout << e << " ";
```

🕮 6.5 指针的定义与运算

❀ 6.5.1 指针的定义

内存空间的访问方式：通过变量名；通过地址。

指针：内存地址，用于间接访问内存单元

指针变量：存放地址的变量

```
int i;
cout << "i=" << i << endl; // i=0
int *p = &i;
*p = 12;
cout << "i=" << i << endl; // i=12
```

注意：

1) 使用变量地址作为指针初始值，该变量必须在指针初始化前声明过，且变量类型与指针类型一致。

2) 可以用一个**合法**的指针初始化另一个指针变量。

3) 不要用一个内部非静态变量去初始化 `static` 指针。

4) 地址运算符`&`与指针运算符`*`互为逆运算。

5) 合法地址，如：通过`&`求得已定义变量或对象的起始地址，动态内存分配成功时返回的地址。

6) 整数 0 可以赋值给指针，表示空指针。C++11 使用 `nullptr` 关键字，是表达更准确、类型安全的空指针。

7) 允许定义 `void*`类型的指针，该指针可以被赋予任何类型对象的地址。
它只能存放地址，不能访问指向的对象，需要类型转换为其他指针。

```
void *pv; // void 类型指针
int i = 10;
pv = &i;                          // void 类型指针指向 int 变量
int *pi = static_cast<int *>(pv); //void 指针转为 int 指针
cout << "*pi = " << *pi << endl;  // *pi = 10
```

❀ 6.5.2 常量指针与指针常量

① 指向常量的指针(const 指针)

特点：不能改变所指对象的值，但指针本身可以改变，即可以指向其他对象

```
int a = 10;
const int *p = &a;
int b = 22;
p = &b; //ok, p 可以指向其他 int
// error: assignment of read-only location '* p'
*p = 8; // 编译出错,不能通过 p 改变对象的值
```

② 指针类型的常量：指针本身的值不能改变

```
int a = 10;
int *const p = &a;
*p = 8; // ok, 可以修改指针指向对象的值
cout << "a = " << a << endl; // a = 8
```

```
int b = 22;
// error: assignment of read-only variable 'p'
p = &b; // 错误,p 是指针常量,不能改为指向其他对象
```

❁ 6.5.3 指针的运算

① 算术运算

指针±n：指针当前位置的前方或后方第 n 个数据的起始位置。

指针++、--运算：指向下一个或上一个完整的数据的起始。

运算的结果值取决于指针指向的数据类型，总是指向一个完整数据的起始

一般当指针指向连续存储的同类型数据时，指针与整数的运算才有意义。

```
int arr[3] = {1, 2, 3};
int *p = arr;
cout << "*p = " << *p << endl; // arr[0]
cout << "*(p+1) = " << *(p+1) << endl; // arr[1]
cout << "*(p+2) = " << *(p+2) << endl; //arr[2]
```

② 关系运算

1) 指向同类型数据的指针之间可以进行各种关系运算。
2) 指向不同类型的指针、指针与一般整数变量的关系运算是无意义的。
3) 指针可以和 0 进行等于或不等于关系运算，判断是否空指针。

🕮 6.6 指针与数组

❁ 6.6.1 使用指针访问数组

❁ 6.6.2 指针数组

```
int row1[] = {1, 2, 3};
int row2[] = {2, 3, 4};
int row3[] = {3, 4, 5};
int *matrix[] = {row1, row2, row3};
for (int i = 0; i < 3; i++){
    for (int j = 0; j < 3; j++){
        cout << matrix[i][j] << "\t";
    }
    cout << endl;
}
```

二维数组按行连续依次存放；对于指针数组，3 个指针 matrix[0]、matrix[1]、matrix[2]连续，但它们指向的 3 个数组之间并不一定连续存放。

🕮 6.7 指针与函数

❁ 6.7.1 指针作为函数参数

需要数据双向传递时(引用也可以达到此效果)

需要传递一组数据，只传首地址运行效率更高

```
void print(int const *arr, int n){ // 打印 int 数组
    if (n < 1){
        cout << "[]" << endl;
        return;
    }
    cout << "[";
    for (int *p = arr;; p++){
```

```
        if (p >= arr + n - 1){
            cout << *p << "]" << endl;
            return;
        }
        cout << *p << ", ";
    }
}

int main(int argc, char const *argv[]){
    int a[] = {1, 2, 3, 4, 5};
    print(a, sizeof(a) / sizeof(int)); // [1, 2, 3, 4, 5]
}
```

6.7.2 指针类型的函数

不要将非静态局部地址作为函数返回值，离开函数体该地址已经无效。

1) 应该返回在主调函数有效、合法的地址

2) 函数中通过动态内存分配 `new` 操作获得的内存地址返回主调函数是合法有效的，但内存分配和释放不在同一级别，要注意不能忘记释放，避免内存泄露

6.7.3 指向函数的指针

函数指针指向的是程序代码存储区首地址。

函数指针典型用途：实现函数回调

1) 通过函数指针调用函数，使得处理相似事件时可灵活使用不同的方法

2) 调用者不关心谁是被调用者，只要知道存在一个具有特定原型和限制条件的被调用函数。

```
int compute(int a, int b, int (*f)(int, int)) { return f(a, b); }
int max(int a, int b) { return (a > b) ? a : b; }
int min(int a, int b) { return (a < b) ? a : b; }
int sum(int a, int b) { return a + b; }

int main(int argc, char const *argv[]){
    int a = 12, b = 23;
    // 函数名就是指针，写不写&都可以
    cout << "max(" << a << ", " << b << ") = " << compute(a, b, max) << endl;
    cout << "min(" << a << ", " << b << ") = " << compute(a, b, min) << endl;
    cout << "sum(" << a << ", " << b << ") = " << compute(a, b, sum) << endl;
}
```

结果：

```
max(12, 23) = 23
min(12, 23) = 12
sum(12, 23) = 35
```

6.8 对象指针

用指针访问对象成员：p->get()相当于(*p).get()

`this` 指针：指向当前对象自己的指针，隐含于类的每一个非静态成员函数中，指出成员函数所操作的对象。

当通过一个对象调用成员函数时，系统先将该对象的地址赋值给 `this` 指针，然后调用成员函数。成员函数对数据进行操作时，就隐含使用了 `this` 指针。

```cpp
class A{
  private:
    int a;
  public:
    A(int a = 0) { this->a = a; }
    void setA(int a) { this->a = a; }
    int getA() { return this->a; }
};

int main(int argc, char const *argv[]){
    A a(12);
    cout << "a = " << a.getA() << endl;
    A *p = &a; // 对象指针
    p->setA(23); // 通过指针调用成员函数
    cout << "a = " << a.getA() << endl;
    return 0;
}
```

🕮 6.9 动态分配内存

① 动态申请内存操作符 new

new T(初始化参数列表)

功能：在程序执行期间，申请用于存放 T 类型对象的内存空间

成功返回 T*，指向新分配的内存；失败，抛出异常。

② 释放内存操作符 delete

delete 指针 p

释放指针 p 所指向的内存，必须是 new 操作申请的地址。

🏵 6.9.1 动态数组

分配：new T[长度]

释放：delete[] 数组名 p

多维数组

new T[第 1 维长度][第 2 维长度]...

🕮 6.10 智能指针

C++11 的智能指针

unique_ptr：不允许多个指针共享资源，可以用标准库的 move 函数转移指针

shared_ptr：多个指针共享资源

weak_ptr：可复制 shared_ptr，但其构造或释放对资源不产生影响

🕮 6.11 vector 对象

封装任何类型的动态数组，自动创建和删除

数组下标越界检查

```cpp
double avg(vector<int> v){
    double s = 0;
    for (int i = 0; i < v.size(); i++)
        s += v[i];
```

```
    return s / v.size();
}

int main(int argc, char const *argv[]){
    vector<int> arr = {34, 76, 23, 45, 12, 56};
    cout << "avg(arr) = " << avg(arr) << endl;
    return 0;
}
```

除了使用下标遍历 vector，还可以：

```
    for (auto i = arr.begin(); i != arr.end(); i++) // 迭代器
        cout << *i << endl;

    for (auto e : arr) // for-each
        cout << e << endl;
```

🕮 6.12 对象复制与移动

❀ 6.12.1 浅层复制与深层复制

① 浅层复制：实现对象间数据元素的一一对应复制。
默认的复制构造函数

② 深层复制：当被复制对象数据成员的指针类型时，不是复制该指针成员本身，而是将指针指向的对象进行复制。

例：

1) 默认复制构造函数是浅层复制

```
#include <iostream>
using namespace std;

void print(int const *arr, int n); // P26

class MyArray{
  private:
    int *arr;
    int size;
    void copy(int const *arr, int low, int high){
        size = 0;
        this->arr = new int[high - low];
        while (low < high)
            this->arr[size++] = arr[low++];
    }

  public:
     MyArray(int const *arr, int low, int high){
        copy(arr, low, high);
        cout << "calling constructor..." << endl;
    }
    ~MyArray(){
        delete[] arr;
        cout << "calling deconstructor..." << endl;
    }
    void show(){print(arr, size);}
    void set(int n, int e){
        if (n < 0 || n >= size)
```

```
            return;
        arr[n] = e;
    }
};

int main(int argc, char const *argv[]){
    int arr[] = {1, 2, 3, 4, 5};
    int length = sizeof(arr) / sizeof(int);
    MyArray ma1(arr, 0, length);
    MyArray ma2(ma1);
    ma1.show(); // [1, 2, 3, 4, 5]
    ma2.set(2, 23);
    ma1.show(); // [1, 2, 23, 4, 5]
    return 0;
}
```

2) 自定义深层复制构造函数

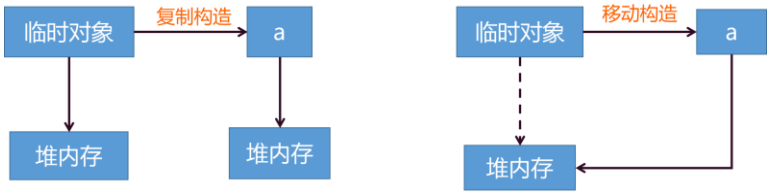
```
    MyArray(MyArray const &a){
        copy(a.arr, 0, a.size);
        cout << "calling copy constructor..." << endl;
    }
```

6.12.2 移动构造

C++11 提供的一种新的构造方法。C++11 之前，如果要将源对象的状态转移到目标对象，只能通过复制。而在某些情况，没必要复制对象，只需要移动它们。

*移动*语义：源对象资源的控制权全部交给目标对象

当临时对象在被复制后，就不再被利用了。完全可以把临时对象的资源直接移动，就避免了多余的复制操作。



函数返回含有指针成员的对象

① 使用深层复制构造函数

返回时构造临时对象，动态分配将临时对象返回到主调函数，再删除临时对象

② 使用移动构造函数

将要返回的局部对象转移到主调函数，省去构造和删除临时对象的过程。

```
    MyArray(MyArray &&a){ // 移动构造函数, &&是右值引用
        arr = a.arr;
        a.arr = nullptr;
        cout << "calling move constructor..." << endl;
    }
```

然而实际运行没有调用移动构造函数，应该与编译器优化有关。

6.13 字符串

6.13.1 C 风格字符串

字符串常量
各字符连续存储，每个字符 1 个字节，以'\0'结尾，相当于一个隐含创建的字符常量数组。首地址可以赋给 char 常量指针：

```
const char *STR = "hikari";
```

字符数组存储字符串

```
char str[] = "hikari";
```

用字符数组表示字符串的缺点：
1) 执行连接、拷贝、比较等操作，都要显示调用库函数
2) 当字符串长度不确定时，需要 new 动态创建字符数组，最后 delete 释放
3) 字符串实际长度大于为它分配的空间时，产生数组下标越界的错误

6.13.2 string 类

常用构造函数：

```
string(); //默认构造函数, 建立一个长度为 0 的字符串
string(const char* s); // 用指针指向的字符串常量初始化
string(const string& s); // 复制构造函数
```

输入整行字符串

getline()可以输入整行字符串(string 头文件)

```
getline(cin, s);
```

输入字符串时，可以使用其他分隔符作为字符串结束标志：

```
getline(cin, s, ','); //逗号为分隔符
```

```
#include <iostream>
#include <string>
using namespace std;

int main(int argc, char const *argv[]){
    for (int i = 0; i < 2; i++){
        string country, city;
        getline(cin, country, ',');
        getline(cin, city);
        cout << "country: " << country << ", city: " << city << endl;
    }
    return 0;
}
```

例：字符串拼接

```
#include <iostream>
#include <string>
using namespace std;
char *concat(char const *a, char const *b){
    //计算 a 和 b 的长度
    int la, lb = 0;
    while (a[la] != '\0')
        la++;
    while (b[lb] != '\0')
        lb++;
    char *s = new char[la + lb + 1];
    // 将 a 和 b 的每个字符填入新的字符数组,结尾加上'\0'
```

```
    for (int i = 0; i < la; i++)
        s[i] = a[i];
    for (int i = 0; i < lb; i++)
        s[la + i] = b[i];
    s[la + lb] = '\0';
    return s;
}
int main(int argc, char const *argv[]){
    char a[] = "hoshizora";
    char b[] = "rin";
    cout << concat(concat(a, " "), b) << endl; //hoshizora rin
    string sa = a, sb = b; //string 类
    cout << sa + " " + sb << endl; //hoshizora rin
    return 0;
}
```