下面是往年的试卷，笔者复习时的笔记，若有拙漏还望不弃草昧，希望开卷有益，一些简单的题目就简单介绍一下题意或者给出答案

前言

题型：

一、单项选择题（20 分，每题 2 分或 30 分，每题 2 分）
二、读程序写输出（20 分，每题 4 分）
三、判断正误或者简答题，（20 分，每题 2 分）或（每题 2 分，共 20 分）
四、程序填空题，某个位置挖一小块（20 分，每题 2 分）
五、编程题，稍微整块一些（20 分，每题 10 分）

提示：

- 可读性很重要，请多写注释，多利用函数进行实现。如果老师能很好地理解你的程序，你可以得到更高的分数。
- 考试试卷是英文版！！
- 有些答案是CHATGPT生成，不准确之处请明辨，用VS调试的结果都是准确的，已经标明

计院1

【金山文档】 华南理工大学高级语言程序设计(C _II)试卷及答案
https://kdocs.cn/l/chLuNfdF1MKv

一、有理数类的运算

求最大公约数

- 使用欧几里得算法（辗转相除法）来求两个整数的最大公约数

  基本思想是，两个整数的最大公约数等于其中较小的数和两数相除的余数的最大公约数

```
int gcd(int a, int b) {
    while (b != 0) {
        int temp = b;
        b = a % b;
        a = temp;
    }
    return a;
}
```

分数化简

- 求分子和分母的最大公约数，并将它们约分

1. 可以使用先前提到的欧几里得算法来计算最大公约数
2. 将分子和分母分别除以最大公约数得到化简后的分数。

小数转化成分数

1. 将小数的小数部分乘以适当的倍数，使其成为一个整数
2. 然后将这个整数作为分子，相应倍数的10作为分母
3. 将得到的分数进行化简

在成员函数中重载类型转换符double

```
operator double() {
        return static_cast<double>(numerator) / denominator;
    }

    Fraction frac(3, 4);
    double result = frac; // 使用 double 转换符，将 Fraction 对象转换为 double 类型的值
```

作为成员函数、非成员函数的重载运算符

- 成员函数：单参数

```
Rational& Rational::operator +(const Rational &other)
```

- 非成员函数：

```
Rational operator+(const Rational &, const Rational &)
```

复制构造函数和重新赋值运算符

- 复制构造函数

  复制构造函数用于创建一个新对象，并且该新对象的值与另一个已存在的对象相同。通常在以下情况下会调用复制构造函数：

  - 使用一个对象来初始化另一个对象。
  - 将对象作为参数传递给函数，函数的参数是按值传递的。
  - 在函数返回对象时，会调用复制构造函数来创建返回值的副本。

```
ClassName(const ClassName& other) {
    // 复制其他对象的数据到当前对象
}

```

- 重载赋值运算符

重载赋值运算符函数允许你重载对象的赋值操作符（`=`），从而在对象之间进行赋值操作。通常在以下情况下会调用重载赋值运算符函数：

- 通过赋值操作符（`=`）将一个对象的值赋给另一个对象。
- 在对象初始化后，需要重新分配一个新的值给对象。

```
ClassName& operator=(const ClassName& other) {
    if (this == &other) {
        return *this;
    }
    // 将其他对象的数据赋给当前对象
    return *this;
}

```

区别

1. **执行时机：** 复制构造函数在对象初始化时被调用，而重载赋值运算符函数在对象已经存在并且需要给它赋值时被调用。
2. **使用方式：** 复制构造函数用于创建新对象并将已有对象的值复制到新对象中，而重载赋值运算符用于将已有对象的值赋给另一个已存在的对象。
3. **返回类型：** 复制构造函数不需要返回值，因为它的主要目的是初始化新对象；而重载赋值运算符函数返回一个引用，以便允许链式赋值。

一些陌生的单词

- numerator 分子
- denominator 分母
- maxCommonFactor 最大公因子
- fraction 分数

二、给出类，回答问题

1.考察私有继承

- 将基类的公有和保护成员变为派生类的私有成员，而基类的私有成员在派生类中不可访问

  `getx()` 函数在类 A 中是公有的，但是类 B 私有继承了类 A。私有继承意味着类 A 的公有成员在类 B 中变为私有成员，因此无法在类 B 外部或者类 B 的派生类中直接访问。除非在类 B 中进行重新声明。

- 基类的get函数，返回的是基类对应的数据，注意同名产生的误导
- 派生类在创建时，同样创建了其基类，基类是按照默认构造函数创建的，故派生类调用基类的public成员函数时，不进行相关操作的前提下，保持构造函数中的默认值
- 如果用了委托构造函数

```
B(double px=2):A(px)
```

那么 `B b(3)` 是将3赋给了基类的x，而本身的x参数是默认为0的

2.考察虚继承

(1)介绍

虚继承是 C++ 中的一种继承方式，用于解决菱形继承（diamond inheritance）问题。菱形继承指的是一个派生类同时继承了两个不同路径上的共同基类，导致了同一个基类在派生类中出现了多次，可能引起二义性和冗余的问题。

虚继承通过关键字 `virtual` 来声明基类，使得在派生类中只有一个基类子对象，从而解决了菱形继承问题。

特点：

1. **虚继承声明：** 使用 `virtual` 关键字来声明虚继承，例如 `class Derived : virtual public Base`。
2. **唯一基类：** 虚继承保证了派生类中只有一个基类子对象，即使在多条继承路径中都继承了同一个基类。
3. **共享基类：** 虚继承使得派生类中共同基类的子对象在内存中只有一份，从而避免了二义性和冗余。

用法：

虚继承通常用于以下情况：

- 解决菱形继承问题：当一个派生类继承了多个具有相同基类的基类时，可以使用虚继承来解决二义性问题。
- 实现多重继承时的代码复用：虚继承可以避免在派生类中出现冗余的基类子对象，使得代码更加清晰和可维护。

示例：

```
#include <iostream>

class Base {
public:
    int x;
};

class Derived1 : virtual public Base {
};

class Derived2 : virtual public Base {
};

class Diamond : public Derived1, public Derived2 {
};

int main() {
    Diamond d;
    d.Derived1::x = 10; // 合法
    d.Derived2::x = 20; // 合法
```

```
    std::cout << "x from Derived1: " << d.Derived1::x << std::endl; // 输出 10
    std::cout << "x from Derived2: " << d.Derived2::x << std::endl; // 输出 20

    return 0;
}

```

在上面的示例中，Base 类被虚继承到 Derived1 和 Derived2 中，然后 Derived1 和 Derived2 被普通继承到 Diamond 中。这样做保证了 Diamond 中只有一个 Base 类对象，从而避免了二义性问题。

也就是说，间接派生对象Diamond只会产生一个最基本的基类的数据成员副本，避免了二义性，究竟是要调用哪一个初始对象

（2）题目

- 虚函数与动态绑定
    1. 在派生类中重写基类的虚函数，在基类一定要加`virtual`！！！如果继续往下派生，派生的上一级基类都要加`virtual`，否则就算函数重名，以下步骤基类指针都只是调用基类版本，不过可能派生类会委托构造，基类同一函数输出可能不太一样
    2. 创建基类指针，指向派生类对象
    3. 通过指针调用函数，该过程中编译器会识别对象类型调用相应的虚函数
- 抽象类不可以被实例化，纯虚函数语法

```
virtual returnType functionName( ) = 0;
```

三、模板

1.swap为引入

- 由于传入的两个参数要进行交换，必须传引用
- 返回值为void即可

```
template < typename T   >
void swap(T& a, T& b) {
  T x;
  x = a;
  a = b;
  b = x;
}
```

再看一个例子

```
template <typename T>
T max(T a, T b) {
    return (a > b) ? a : b;
}
```

- **注意事项**

  模板的定义和声明通常放在头文件中，以便在多个源文件中使用。

  在使用模板时，编译器会根据模板参数的实际类型生成特定的函数或类定义。

  模板的编译错误通常会在实例化时报告，因此一些错误可能在实际使用之前不会被发现。

- 基本语法

  声明前要有`template <typename T>`，余下的操作可以对`T`展开

再具体介绍一下：

函数模板

函数模板允许编写单个函数定义，该函数可以用于多种不同类型的参数。例如，下面是一个简单的函数模板，用于计算两个数的最大值：

```
template <typename T>
T max(T a, T b) {
    return (a > b) ? a : b;
}

int main() {
    int result1 = max(5, 10);
    double result2 = max(3.5, 7.2);
    return 0;
}
```

在这个例子中，`max()`函数模板可以接受不同类型的参数，并返回最大值。

类模板

类模板允许创建通用的类定义，其中某些成员的类型是模板参数。例如，下面是一个简单的类模板，用于表示一对值：

```
template <typename T>
class Pair {
private:
    T first;
    T second;
public:
    Pair(T a, T b) : first(a), second(b) {}
    T getFirst() { return first; }
    T getSecond() { return second; }
};

int main() {
```

```
    Pair<int> p1(5, 10);
    Pair<double> p2(3.5, 7.2);
    return 0;
}
```

在这个例子中，`Pair` 类模板可以创建一对任意类型的值。

模板特化和部分特化

模板特化允许为特定类型的模板参数提供定制的实现。部分特化允许为模板参数的某些子集提供定制的实现。例如：

```
template <typename T>
class MyClass {
    // 一般实现
};

template <>
class MyClass<int> {
    // int 类型的特化实现
};

template <typename T>
class MyClass<T*> {
    // 指针类型的部分特化实现
};
```

2.vector

- vector利用模板初始化，第一个参数是一个内置数组，第二个参数是要复制的数据量

```
  const int size=10;
  int a[size]={10,3,17,6,15,8,13,34,25,2};
  vector<int>V(a,a+size);            // 用数组对模板向量赋初值
```

sort函数

- 函数原型

```
template <class RandomAccessIterator>
void sort (RandomAccessIterator first, RandomAccessIterator last);

template <class RandomAccessIterator, class Compare>
void sort (RandomAccessIterator first, RandomAccessIterator last,
Compare comp);
```

`first`：指向容器中第一个要排序的元素的迭代器。

`last`：指向容器中最后一个要排序的元素的下一个位置的迭代器（即不包含在排序范围内）。

`comp`：可选参数，比较函数对象，用于指定排序的比较规则。默认情况下，采用 `<` 运算符进行比较。（升序）

- comp可以传入一个自定义排序方式的函数
- 通常，`comp` 参数是一个函数对象，可以是函数指针、函数对象或者 lambda 表达式。这个函数对象接受两个参数，表示待比较的两个元素，返回一个 `bool` 值，表示第一个元素是否应该排在第二个元素之前
- lamda表达式的形式

```
std::sort(vec.begin(), vec.end(), [](int a, int b) { return a > b;
});
```

- 该题中传入了down函数，实现降序排列

```
bool down(int x,int y)
  {return x > y ;}
```

四、文件操作

- 写入内容，使用`ios::out`
- 对文件完成操作后要关闭文件

软院1

【金山文档 | WPS云文档】 华工C++II试卷及答案 2
https://kdocs.cn/l/cmQDQ0GaGvTJ

一、单选

- 在类定义中，称为接口的成员是**public**类成员
- 友元函数访问类成员

  通过类对象参数，也即通过类对象加点或箭头运算符调用相应成员，private、public、protected的都可以

  静态的成员依旧可以，通过类名加作用域运算符就行

  友元函数本身不是成员函数，无this指针
- 构造函数
  - **构造函数通常是公有的**，这样才能在类的外部创建对象。如果构造函数是私有的，那么只有类的成员函数或友元函数可以创建对象。
  - **构造函数不能是虚函数**。C++ 中的虚函数机制是为了解决多态性问题，主要用于成员函数的动态绑定。而**构造函数在对象创建时就需要调用，无法利用虚函数表进行动态绑定**。析构函数可以是虚函数，以确保在**删除基类指针时调用正确的派生类析构函数。**
- 派生类可以直接调用基类的public和protected成员

  派生类对象和基类对象间不能简单地用赋值运算符，无论顺序如何

  派生类指针不可以指向基类，`is-a`关系不对

- 根据参数个数、类型，调用重载函数是静态绑定，在程序运行前就确定，而if、switch、类的多态是在运行时决定

二、简答题

1.继承

基类成员	公有继承	保护继承	私有继承
`public`	`public`	`protected`	`private`
`protected`	`protected`	`protected`	`private`
`private`	不可访问	不可访问	不可访问

- `Z` 私有继承了 `Y`，这意味着 `Y` 的所有成员（包括 `X` 的 `protected` 成员 `a`）在 `Z` 中都变成了 `private`。因此，`Z` 的成员函数 `funX` 不能访问 `a`，因为 `a` 在 `Z` 中是 `private`。只能在定义它的基类成员中访问
- 继承的方式是改变了继承对象对于派生类而言的数据属性，而不是直接添加该属性的数据到派生类对象中

  因而通过private的继承，派生类无论是在其内部（成员函数）还是外部（其它函数）都无法访问基类的数据，只能通过调用基类的成员函数访问基类数据

3.

带参数的构造函数用于建立对象数据初始化，成员函数用于程序运行时修改数据成员的值。

五、完成程序

1.

- 根据输出的结果，可以确定 `A` 和 `B` 的析构函数需要按照以下方式实现：
  1. `A` 的析构函数需要是虚函数，这样当通过 `A` 类型的指针删除指向 `B` 类型的对象时，会调用 `B` 的析构函数，然后再调用 `A` 的析构函数。
  2. `B` 的析构函数则只需要在析构时输出对应的销毁信息。
- 使用 `virtual` 关键字声明析构函数是为了确保在删除通过 `A` 类型的指针指向的 `B` 类型的对象时，会**正确调用 `B` 的析构函数。**
- **如果 `A` 的析构函数不是虚函数**，删除一个通过 `A` 类型的指针指向的 `B` 类型的对象时，**只会调用 `A` 的析构函数**，而不会调用 `B` 的析构函数。这会导致 `B` **类的析构函数没有被正确调用**，从而可能造成资源泄漏或其他未定义行为。

2.

最小定义，则使用内联函数实现，构造析构默认即可

软院2

【金山文档 | WPS云文档】 C++（II）13-14(2)试题B - 往年试卷
https://kdocs.cn/l/cmxPaRqdrpll

二

3

fstream的对象可以直接用流运算符将double对象写入到文本文件，其以字符串形式存储，而写入二进制文件需要用write函数，将double对象强制转化为二进制字节，大小设置为double的大小

```cpp
#include <iostream>
#include <fstream>

using namespace std;

int main() {
    double PI = 3.1415;

    // 打开文本文件并写入 PI 的值
    ofstream ftxt("d:\\test1.txt", ios::out);
    if (!ftxt.is_open()) {
        cerr << "无法打开文本文件: d:\\test1.txt" << endl;
        return 1;
    }
    ftxt << PI;
    ftxt.close();

    // 打开二进制文件并写入 PI 的值
    ofstream fdat("d:\\test2.dat", ios::out | ios::binary);
    if (!fdat.is_open()) {
        cerr << "无法打开二进制文件: d:\\test2.dat" << endl;
        return 1;
    }
    fdat.write(reinterpret_cast<char*>(&PI), sizeof(PI));
    fdat.close();

    return 0;
}

```

浮点数据在两个文件中各占的字节数相同吗?

不相同。原因如下：

1. **文本文件**:
    - 在文本文件中，数据以可读的字符串形式存储。写入文本文件时，`PI` 的值被转换为字符串 "3.1415"，并存储在文件中。

- 具体占用的字节数取决于字符串表示形式，例如 "3.1415" 占 6 个字节（包括小数点）。

2. **二进制文件**:
   - 在二进制文件中，数据以二进制格式存储。写入二进制文件时，`PI` 的值被直接存储为 `double` 类型的二进制表示。
   - 在大多数现代计算机体系结构中，`double` 类型占 8 个字节（64 位）。

5、虚继承的构造函数调用顺序

虚基类的构造函数：最先调用的是虚基类（即所有直接或间接的虚基类）的构造函数。无论虚基类在继承层次结构中的位置如何，它们的构造函数总是最先执行。虚基类的构造函数只会调用一次，即使有多个派生类从它继承。

非虚基类的构造函数：在调用完所有虚基类的构造函数后，接下来会按照继承关系中出现的顺序调用非虚基类的构造函数，从最基类开始，逐级向下。

直接基类的构造函数：然后是直接基类的构造函数。如果有多个直接基类，它们的构造函数按照它们出现在派生类定义中的顺序调用。

派生类的构造函数：最后调用的是派生类自己的构造函数。

```cpp
#include <iostream>

using namespace std;

class Base {
public:
    Base() { cout << "Constructing Base" << endl; }
};

class Derived1 : virtual public Base {
public:
    Derived1() { cout << "Constructing Derived1" << endl; }
};

class Derived2 : virtual public Base {
public:
    Derived2() { cout << "Constructing Derived2" << endl; }
};

class Derived3 : public Derived1, public Derived2 {
public:
    Derived3() { cout << "Constructing Derived3" << endl; }
};

int main() {
    Derived3 obj;
    return 0;
}

```

输出结果如下：

```
Constructing Base
Constructing Derived1
Constructing Derived2
Constructing Derived3
```

三

1.数组反转

这个程序的功能是对数组 `a` 进行部分反转和整体反转。让我们一步一步解析这个程序的工作原理和最终输出。

首先，程序定义了一个函数 `fun`，用于反转数组的某个部分：

```
cpp复制代码void fun(int *s, int n1, int n2) {
    int i, j, t;
    i = n1;
    j = n2;
    while (i < j) {
        t = *(s + i);
        *(s + i) = *(s + j);
        *(s + j) = t;
        i++;
        j--;
    }
}
```

然后，在 `main` 函数中，定义了一个包含10个整数的数组 `a` 并依次调用 `fun` 函数对数组进行操作：

```
cpp复制代码void main() {
    int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 0}, i, *p = a;
    fun(p, 0, 3);
    fun(p, 4, 9);
    fun(p, 0, 9);
    for (i = 0; i < 10; i++)
        cout << *(a + i);
    cout << endl;
}
```

现在，让我们详细地看一下每一步的数组变换：

1. **初始数组**：

   ```
   a = {1, 2, 3, 4, 5, 6, 7, 8, 9, 0}
   ```

2. **调用** `fun(p, 0, 3)`：

- 反转数组 a的第0到第3个元素：
  - i = 0, j = 3: 交换 `a[0]` 和 `a[3]` -> `4, 2, 3, 1, 5, 6, 7, 8, 9, 0`
  - i = 1, j = 2: 交换 `a[1]` 和 `a[2]` -> `4, 3, 2, 1, 5, 6, 7, 8, 9, 0`

```
css
复制代码
a = {4, 3, 2, 1, 5, 6, 7, 8, 9, 0}
```

3. **调用** `fun(p, 4, 9)`：
   - 反转数组 a 的第4到第9个元素：
     - i = 4, j = 9: 交换 `a[4]` 和 `a[9]` -> `4, 3, 2, 1, 0, 6, 7, 8, 9, 5`
     - i = 5, j = 8: 交换 `a[5]` 和 `a[8]` -> `4, 3, 2, 1, 0, 9, 7, 8, 6, 5`
     - i = 6, j = 7: 交换 `a[6]` 和 `a[7]` -> `4, 3, 2, 1, 0, 9, 8, 7, 6, 5`

```
css
复制代码
a = {4, 3, 2, 1, 0, 9, 8, 7, 6, 5}
```

4. **调用** `fun(p, 0, 9)`：
   - 反转整个数组 a：
     - i = 0, j = 9: 交换 `a[0]` 和 `a[9]` -> `5, 3, 2, 1, 0, 9, 8, 7, 6, 4`
     - i = 1, j = 8: 交换 `a[1]` 和 `a[8]` -> `5, 6, 2, 1, 0, 9, 8, 7, 3, 4`
     - i = 2, j = 7: 交换 `a[2]` 和 `a[7]` -> `5, 6, 7, 1, 0, 9, 8, 2, 3, 4`
     - i = 3, j = 6: 交换 `a[3]` 和 `a[6]` -> `5, 6, 7, 8, 0, 9, 1, 2, 3, 4`
     - i = 4, j = 5: 交换 `a[4]` 和 `a[5]` -> `5, 6, 7, 8, 9, 0, 1, 2, 3, 4`

```
a = {5, 6, 7, 8, 9, 0, 1, 2, 3, 4}
```

最后，程序输出整个数组的元素：

```
5678901234
```

2.文件读写

`fstream` 的输入操作符（>>）会**跳过空白字符（空格、制表符、换行符）**来读取数据。因此在这个程序中，字符读取会是这样的。

详细说明

当使用 `>>` 运算符从文件流中读取数据时，它会自动跳过任何空白字符（空格、制表符、换行符）并读取下一个有效的非空白字符。因此，程序能够正确地解析文件中的数据，即使这些数据是由空白字符分隔的。

示例代码解释

```
cpp复制代码fstream iofile(fname, ios::in);
if (!iofile) return;
iofile >> str >> a >> str >> a >> str >> a;
cout << "area=" << a << endl;
iofile.close();
```

让我们逐步解析这一段代码读取的数据：

1. **文件内容**：

   ```
   yaml
   复制代码
   length: 10  width:  20  area: 200
   ```

2. **读取过程**：
   - `iofile >> str` 读取 `length:`
   - `iofile >> a` 读取 `10`
   - `iofile >> str` 读取 `width:`
   - `iofile >> a` 读取 `20`
   - `iofile >> str` 读取 `area:`
   - `iofile >> a` 读取 `200`

具体步骤

假设文件指针初始位置在文件的开始位置，文件内容为：

```
yaml
复制代码
length: 10  width:  20  area: 200
```

- **第一次读取**：
  - ```
    iofile >> str
    ```
    - 跳过所有空白字符（没有空白字符）
    - 读取字符串 `length:`
    - `str` 现在是 `"length:"`
  - ```
    iofile >> a
    ```
    - 跳过所有空白字符（遇到一个制表符）
    - 读取整数 `10`
    - `a` 现在是 `10`
- **第二次读取**：
  - ```
    iofile >> str
    ```
    - 跳过所有空白字符（遇到一个制表符）
    ```

        - 读取字符串 `width:`
        - `str` 现在是 `"width:"`
    - ```
      iofile >> a
      ```
        - 跳过所有空白字符（遇到一个制表符）
        - 读取整数 `20`
        - `a` 现在是 `20`
- **第三次读取**：
    - ```
      iofile >> str
      ```
        - 跳过所有空白字符（遇到一个制表符）
        - 读取字符串 `area:`
        - `str` 现在是 `"area:"`
    - ```
      iofile >> a
      ```
        - 跳过所有空白字符（遇到一个制表符）
        - 读取整数 `200`
        - `a` 现在是 `200`

3.模板类

关键点说明

1. **模板类 `FF`**：
    - 这是一个模板类，它可以接受任何数据类型（如 `int`，`float`，`double` 等）。
    - 模板参数 `TT` 被用来定义类的成员变量和成员函数的参数类型。
2. **成员变量和构造函数**：
    - 类 `FF` 有三个成员变量 `a1`，`a2`，和 `a3`，它们的类型是模板参数 `TT`。
    - 构造函数使用初始化列表来初始化这三个成员变量。
3. **成员函数 `Sum`**：
    - 这个函数返回三个成员变量的和。因为这些变量的类型是 `TT`，所以这个函数可以适用于任何支持加法运算的类型。

4.派生类构造函数

同二.5中所述

先调用虚基类，然后基类，直接基类，再到本身的构造函数，析构时相反

如果继承多个类要注意用`虚继承`

5.模板函数

注意输出 `" "`是输出了一个空格而不是引号

四、程序填空
      ```

1.继承

```
class bicycle : public virtual vehicle  //自行车类

class motorcar : public virtual vehicle  //机动车类

class motorcycle: public bicycle, public motorcar  //摩托车类
```

`virtual`加在`public`后

2.求和

第一个空没有想到，指针指向数组时要先给其new出空间，不然它没有初始化，无指向的对象

```
T* c = new T[n];
for(int i = 0; i < n; i++)
    c[i] = a[i] + b[i];
return c;

```

```
cout << p[i];
```

```
p[i]
```

4.

```
x=2x square=4
x=3x cube=27
```

五、程序编写

1.

```
template <typename T>
T average(const T target[], int size)//传入内置数组以及它的大小两个参数
{
    T sum = 0;
    for(int i = 0; i < size; i++){
      sum += target[i];
    }
    return sum/size;
}
```

2.

```
void createBinaryFile(char * filetext, char * fileData)
{
  ifstream readFromTxt(“d:\ courseFile.txt”, ios::in);
  if(!readFromTxt.open()){//应该用is_open()函数来检查
    cerr << "文件打开失败";
    //这里应该添加return，光输出错误不会影响操作
  }
  else{
    while(cin.getline())//此处应该改为readFromTxt.getline，因为是从文件中读数据而不是从标准流中
    {
      course c;
      readFromTxt >> c.num >> c.hour >> c.credit;
      ofstream writeToBinary(“d:\ courseFile.dat", ios::binary|ios::out);//此处应该多加一个app模式，且打开文件应该在循环外部，影响性能
      writeToBinary.write(reinterpret_cast<char *>(&c), sizeof(course));
    }
  }
}
```

修改后的代码：

```
void createBinaryFile(char * filetext, char * fileData)
{
    ifstream readFromTxt(filetext, ios::in);
    if(!readFromTxt.is_open()){
        cerr << "文件打开失败" << endl;
        return;
    }

    ofstream writeToBinary(fileData, ios::binary | ios::out | ios::app);
    if(!writeToBinary.is_open()){
        cerr << "二进制文件打开失败" << endl;
        return;
    }

    while(readFromTxt.getline())
    {
        course c;
        readFromTxt >> c.num >> c.hour >> c.credit;
        writeToBinary.write(reinterpret_cast<char *>(&c), sizeof(course));
    }

    readFromTxt.close();
    writeToBinary.close();
```

```
24 }
```

软院3

【金山文档 | WPS云文档】 A卷含答案（往年试卷）
https://kdocs.cn/l/cppLtlBRtw0s

一、单选

1.初始化字符数组

- 初始化列表中的字符个数不能超过数组大小，且尽量要留一位给空字符，A勉强对
- 用字符串初始化时，整个字符串包括最后的空字符 '\0' 都会被复制到数组中，而用单个字符初始化时，只有指定的字符会被放入数组中，其余位置会被自动填充为 '\0'。B有两位空字符，D溢出，C在初始化中指定大小
- C++中，空字符 `'\0'` 表示空字符或者空终止符，通常用于表示字符串的结束。它的 ASCII 值为 0。

  而数字 0 表示整数零。

  尽管它们在ASCII值上是相同的（都是0），但它们的含义不同。空字符通常用于表示字符串的结束，而数字0则表示整数零。

3.强制转换

`(x+y)/2`：先计算 `(3+2)/2 = 5/2 = 2`。

`(int)a`：强制转换为整数，即取整数部分，`2.9` 取整为 `2`。

`(int)b`：同样取整数部分，`3.5` 取整为 `3`。

`(int)a%(int)b`：计算取整后的余数，即 `2%3 = 2`。

6.函数指针

对于给定的函数指针数组 `pfun[2]`，我们有以下声明：

```
typedef double ft ( double );  // 定义函数指针类型 ft
ft * pfun[2];  // 声明函数指针数组 pfun
pfun[0] = fun;  // 将 fun 函数的地址赋给 pfun[0]
```

现在来分析每个选项：

A) `( *pfun[0] )(3.14)`：这是通过指针间接调用函数 `fun`，是正确的写法。

B) `fun(3.14)`：这是直接调用函数 `fun`，是正确的写法。

C) `( pfun[0] )(3.14)`：`pfun[0]` 本身就是一个函数指针，所以直接使用 `(3.14)` 作为参数调用它即可，正确

D) `( &pfun[0] )(3.14)`：这是不正确的调用。`&pfun[0]` 取了 `pfun[0]` 的地址，这将是一个指向函数指针的指针，而不是函数指针本身。你不能直接对这个地址使用函数调用语法。正确的用法应该是 `**pfun` 或者 `(*pfun[0])(3.14)`。

所以选择 D

- `ft`是一个函数指针类型，可以指向满足以下条件的函数：

  1. 函数的返回类型和参数类型与`ft`类型匹配。
  2. 函数的参数列表和返回类型与`ft`类型一致。

  因此，如果函数的签名与ft类型匹配，那么该函数就可以用ft指向它。

7.合法标识符

标识符必须以字母、下划线或中文字符开头，后面可以跟字母、下划线、减号、数字或中文字符。不能以关键字作为标识符

- A) `_0123` 和 `ssiped` 都是合法的标识符。在大多数编程语言中，标识符可以以字母或下划线（*）开头，后跟字母、数字或下划线的任意组合。因此， `_0123` 和 `ssiped` 都满足这一规则。

  B) `del-word` 是合法的，但 `signed` 可能不是合法的用户标识符，因为在某些语言中 "signed" 是一个关键字，用来表示有符号类型，因此不能作为普通标识符使用。

  C) list 是合法的，但 `*jer` 不是，因为大多数编程语言中标识符不能以星号（*）开头。

  D) keep% 和 wind 中，keep% 不是合法的，因为百分号（%）通常不是标识符允许使用的字符，而 wind 是合法的。

  所以正确答案是 A)。

8.逻辑运算符两侧的数据类型

> 逻辑运算符两侧的操作数数据类型可以是(　　)
>
> A) 只能是0或1
>
> B) 只能是正整数或0
>
> C) 只能是整数或字符数据
>
> D) 可以是任何合法数据类型
>
> 答案是 D) 可以是任何合法数据类型。

- 在编程中，逻辑运算符（如 AND、OR、NOT）通常用于布尔逻辑操作，它们可以接受任何可以被解释为真（true）或假（false）的值。
- 这包括但不限于布尔值、数字（非零值通常解释为真）、对象（存在即真）、指针（非空即真）等。因此，选项D最符合实际情况，它涵盖了各种可能被逻辑运算所接纳的数据类型。

9.初始化二维数组

A) `int a[2][]={{1,0,1},{5,2,3}};` 这个声明是错误的，因为C语言不允许在声明数组时省略第二维的大小。

B) `int a[][3]={{1,2,3},{4,5,6}};` 这是正确的声明方式。它定义了一个二维数组，其中每行有3个元素，具体初始化值已给出。第一维的大小可以省略，编译器会根据初始化列表自动计算。

C) `int a[2][4]={{1,2,3},{4,5},{6}};` 这个声明是错误的，因为初始化列表的元素数量与声明的数组大小不匹配。第二行只有两个元素，而数组定义为每行4个元素。

D) `int a[][3]={{1,0},{},{1,1}};` 虽然这个声明在某些编译器下可能被接受，因为它尝试初始化一个二维数组，其中一些子数组为空，但这在标准C语言中是不明确的，并可能导致未定义行为。理想情况下，每一行应具有相同数量的元素，或者至少提供的初始值应与数组声明的结构一致。

因此，选项B是正确的，因为它遵循了C语言中初始化二维数组的规范。

- 也就是说，最高维的可以根据初始化列表而定，较低维的必须显式声明

二、写输出结果

2.函数指针

代码解释

```
#include <iostream>

// 定义一个函数f1，接收两个整数参数a和b，返回a除以b的余数乘以5
int f1(int a, int b) {
    return a % b * 5;
}

// 定义一个函数f2，接收两个整数参数a和b，返回它们的乘积
int f2(int a, int b) {
    return a * b;
}

// 定义一个函数f3，接收一个指向接受两个整数参数并返回整数的函数指针t，
// 以及两个整数a和b作为参数，然后调用指针t指向的函数并返回其结果
int f3(int(*)(int, int), int a, int b) {
    return (*t)(a, b);
}

// 主函数
void main() {
    // 声明一个指向接受两个整数参数并返回整数的函数的指针p
    int (*p)(int, int);

    // 将函数f1的地址赋给指针p
    p = f1;
```

```
    // 调用f3函数，传入p、5和6作为参数，这里实际调用了f1(5, 6)，输出计算结果
    cout << f3(p, 5, 6) << endl;

    // 将函数f2的地址赋给指针p
    p = f2;
    // 调用f3函数，传入p、7和8作为参数，这里实际调用了f2(7, 8)，输出计算结果
    cout << f3(p, 7, 8) << endl;
}
```

解释：

这段C++代码定义了三个函数：`f1`、`f2`和`f3`，以及一个主函数`main`。在`main`函数中，通过函数指针`p`灵活地调用了`f1`和`f2`函数，并通过`f3`函数输出它们的结果。

1. **f1(a, b)**: 计算 `a` 除以 `b` 的余数后乘以 `5`。当调用 `f1(5, 6)` 时，因为 `5` 除以 `6` 的余数是 `5`，所以结果是 `5 * 5 = 25`。
2. **f2(a, b)**: 返回 `a` 和 `b` 的乘积。当调用 `f2(7, 8)` 时，结果是 `7 * 8 = 56`。
3. **f3(t, a, b)**: 是一个高阶函数，它接收一个函数指针 `t` 和两个整数 `a`、`b` 作为参数，然后调用 `t` 指向的函数并返回该函数的结果。
    - 首先，`p` 指向 `f1`，通过 `f3(p, 5, 6)` 调用，输出 `f1(5, 6)` 的结果，即 `25`。
    - 然后，`p` 改为指向 `f2`，通过 `f3(p, 7, 8)` 调用，输出 `f2(7, 8)` 的结果，即 `56`。

最终程序的输出将会是：

```
25
56
```

3.从该数递减到1的累和

4.传引用（通过指针）和传值的区别

5.函数内静态数据以及后置自增

该段代码实际上是错误的，不允许重复声明同一变量，且逻辑比较复杂，不建议多纠结这道题

表达式 `t += a++` 是先将 `a` 的值加到 `t` 上，然后再将 `a` 的值加一。

因此，`a` 的值在这个表达式中会被使用两次，但只会在最后一次使用后加一。

静态数据在每次调用结束后会保存其值，下一次调用会继续运用

- 第一次调用 `fun(0)`：
    - 由于参数 `n` 的值为偶数，因此执行偶数情况下的分支。
    - 在该分支中，静态局部变量 `a` 的初始值为 5，将 `t` 加上 `a` 的值，然后 `a` 自增。
    - `t` 的值为 5，`a` 的值自增为 6。
- 第二次调用 `fun(1)`：

    - 由于参数 `n` 的值为奇数，因此执行奇数情况下的分支。
    - 在该分支中，静态局部变量 `a` 的初始值为 5，将 `t` 加上 `a` 的值，然后 `a` 自增。
    - `t` 的值为 5，`a` 的值自增为 6。
- 第三次调用 `fun(2)`：
    - 由于参数 `n` 的值为偶数，因此执行偶数情况下的分支。
    - 在该分支中，静态局部变量 `a` 的初始值为 5，将 `t` 加上 `a` 的值，然后 `a` 自增。
    - `t` 的值为 5，`a` 的值自增为 6。

最终，循环结束，输出 `s` 的值，即 5 + 5 + 5 = 15。因此，程序的输出结果是:

```
15
```

- 只返回了t的值，而不是表达式

6.字符转数字

几个注意点

- 第二层循环里面 j 每次递增2；
- j == 4时，pch[i] [j]是一个空字符'/0'，不在数字范围内，跳出内层循环
- 每一次内层循环将该数乘以10再加上这一次的字符

故最后输出结果是：

```
2004
```

三、简答题

1.默认参数

2.逻辑运算、递增

1. 首先，定义了两个整型变量 `x` 和 `y`。
2. 然后，使用赋值表达式 `x=y=1;` 同时给 `x` 和 `y` 赋值为 `1`。
3. 接下来是 `++x||++y;`语句。这里涉及到两个操作：前置递增运算符++和逻辑或运算符||
    - 由于 `++x` 是前置递增，所以在 `x` 被用作表达式之前，其值先增加 `1`。因此，`x` 从 `1` 变为 `2`。
    - `||` 是逻辑或运算符，其规则是：如果左侧的操作数为真（true），则不会评估右侧的操作数，因为整个表达式的结果已经确定为真。在这个例子中，由于 `++x` 使得 `x` 的值为 `2`（真值），所以 `++y` 不会被执行。
4. 因此，最终 `x` 的值为 `2`，`y` 的值保持为 `1`。

3.do while循环以及条件判断

- 在第一次进入循环之前，`x` 的值为1，`!x`（x的逻辑非）为0（假），但**由于是“do...while”循环，无论条件是否为真，都会先执行一次循环体内（do内）的代码，这就是一次循环**
- 循环体内，`x++` 执行后，`x` 的值变为2；`i++` 执行后，`i` 的值变为1。
- 第一次循环结束后检查条件 `!x && i <= 3`，此时 `x` 为2（`!x` 为假），即使 `i` 仍满足 `i <= 3`，整个条件表达式也为假（因为逻辑与操作中只要有假则整体为假），因此循环不会再次执行。

综上所述，这个循环只执行**1次。**

4.插入大括号实现相应输出

5.找错

a.要赋地址

b.要用间值运算符*

c.等号左右顺序反了

d.要用间值运算符*

e.指针偏移，是+=2

f.要在等号右侧用间值运算符*

四、程序填空

1.阶乘的循环实现以及多阶乘求和

答案:

```
i += 2;
```

```
f *= j;
```

```
s += f;
```

2.ASCLL码的应用

答案:

函数原型

```
void change(char *, char *)
```

当字符不为空时，循环继续

```
*s1 != '/0'
```

利用ASCLL码，减去字符0再加上字符a便可得到相应小写字母的值

```
*s1 - '0' + 'a'
```

往下递增，处理后一位字符

```
*s1 ++;
*s2 ++;
```

3.打印数组

由题，最后打印了整个数组，前面的if条件是根据格式对数组的一些赋值处理

答案：

```
i + j == 8 || i == j
```

```
a[i][j] = 2
```

```
i < j && i+j > 8
```

五、编程序

1.unique

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

// 函数声明
int deleteSame(int a[], int size);

int main() {
    int n;
    cout << "Enter the number of elements: ";
    cin >> n;

    int *arr = new int[n];
    cout << "Enter the elements: ";
    for (int i = 0; i < n; ++i) {
        cin >> arr[i];
    }

    int newLength = deleteSame(arr, n);

    cout << "After deletion, the output is: ";
    for (int i = 0; i < newLength; ++i) {
```

```
        cout << arr[i] << " ";
    }
    cout << endl;
    cout << "The number of data is: " << newLength << endl;

    delete[] arr;  // 释放动态分配的内存
    return 0;
}

// 删除重复元素并返回新数组的长度
int deleteSame(int a[], int size) {
    if (size == 0) return 0;  // 如果数组为空，直接返回0

    int uniqueIndex = 0;  // 用于存储不重复元素的位置

    for (int i = 1; i < size; ++i) {
        if (a[uniqueIndex] != a[i]) {
            uniqueIndex++;
            a[uniqueIndex] = a[i];
        }
    }
    return uniqueIndex + 1;  // 返回新数组的长度
}

```

2.搜索计数

```
int searchCharacter(string target, char key);
{
  int keyNum = 0;
  for(int i = 0; i < target.size(); i++){
    if(target[i] == key){
      keyNum ++;
    }
  }
  return keyNum;
}
```

软院4

【金山文档 | WPS云文档】《面向对象程序设计C++》期末考试试卷 2
https://kdocs.cn/l/csonCAHDLGAD

一、单选

4.

- A) 错误，运算符重载不能用于定义全新的运算符，只能对已存在的C++运算符进行重新定义，改变其对于特定类型数据的操作方式。
- B) 正确，有些运算符如“.”（成员访问）和“::”（作用域解析）只能作为成员函数重载，因为它们的语义直接与类的结构相关。
- C) 错误，若重载运算符+，相应的运算符函数名不是直接使用"+"，而是采用特殊的函数名形式，如对于非成员函数重载是"**operator+**"，对于成员函数重载不需要显式写出操作符名称在前面。
- D) 错误，重载二元运算符时，不一定必须声明两个形参。如果将运算符**重载为一个类的成员函数**，则编译器会自动将**左侧操作数视为该类的隐含this指针**，因此只需要一个额外的参数来表示右侧操作数。例如，对于表达式`a + b`，如果`+`是类A的成员函数，则重载函数定义可能为`A operator+(const A& rhs)`，这里`rhs`代表右侧操作数，而`a`则是通过this指针隐式传递的。

三、程序输出

27.

1. 当创建 `C` 类的对象 `obj` 时，首先会调用基类 `A` 的构造函数。因此，首先输出 "A"。
2. 接下来，由于 `C` 类中定义了一个 `B` 类的成员变量 `b`，所以在构造 `C` 类的对象时，需要先构造 `b`。因此，接下来会调用 `B` 类的构造函数，输出 "B"。
3. 最后，`C` 类自身的构造函数被调用，输出 "C"。

综上所述，构造顺序遵循以下原则：

- **先调用基类的构造函数。**
- **然后按照成员变量声明的顺序调用成员对象的构造函数。**
- **最后调用派生类自身的构造函数体。、**

软院5（编程模块略难一些）

一、单选

1.

答案解析

- A）结构体可以包含多种数据类型。这是正确的。结构体允许你组合不同类型的数据成员，如整型、浮点型、字符型等。

  B）不同结构体的成员必须有唯一的名称。这是不正确的。结构体内部的成员名称不需要与其它结构体中的成员名称唯一，只要在同一结构体内成员名称唯一即可。**（作用域）**

  C）关键字typedef用于定义新的数据类型。这是正确的。typedef用于为现有的数据类型创建一个新的名称（别名），使得代码更具可读性或便于使用。

D）结构体总是通过引用传递给函数。这是不正确的。结构体可以按值传递也可以按引用传递给函数，具体取决于传递的方式。如果不使用引用(&)或指针(*)，结构体会**默认按值传递，但这通常不是效率高的做法，特别是当结构体很大时。使用引用或指针可以实现按引用传递，提高效率。**加上const声明便可以不更改数据

typedef介绍

`typedef` 是 C++ 中用于定义类型别名的关键字，它可以为现有的类型定义一个新的名称，使代码更简洁和易于理解。在复杂的类型声明中，`typedef` 特别有用，可以提高代码的可读性和可维护性。

使用示例和解释：

1. **简单类型别名**：

```
typedef int Integer;
Integer a, b; // a 和 b 都是 int 类型
```

这里，我们定义了 `Integer` 作为 `int` 类型的别名，所以 `a` 和 `b` 事实上都是 `int` 类型。

2. **指针类型别名**：

```
typedef int* IntPointer;
IntPointer p1, p2; // p1 和 p2 都是 int* 类型的指针
```

定义了 `IntPointer` 作为 `int*` 类型的别名。注意，`IntPointer p1, p2;` 定义了两个 `int*` 类型的指针。如果没有 `typedef`，写作 `int* p1, p2;`，则只有 `p1` 是指针，`p2` 只是一个 `int` 类型。

3. **结构体类型别名**：

```
struct Person {
    char name[50];
    int age;
};

typedef struct Person PersonType;
PersonType person1, person2;
```

这里，我们定义了一个结构体 `Person`，并使用 `typedef` 定义 `PersonType` 作为该结构体类型的别名，使得在声明变量时可以简化书写。

4. **数组类型别名**：

```
typedef int Array10[10];
Array10 arr1, arr2; // arr1 和 arr2 都是长度为 10 的 int 数组
```

使用 `typedef` 定义 `Array10` 作为长度为 10 的 `int` 数组的别名。

5. **函数指针类型别名**：

```
typedef void (*FuncPtr)(int, int);
void exampleFunction(int a, int b) {
    // 函数实现
}

FuncPtr fp = exampleFunction;
fp(1, 2); // 调用 exampleFunction(1, 2)
```

使用 `typedef` 定义 `FuncPtr` 作为返回类型为 `void` 且接受两个 `int` 参数的函数指针的别名。

2.字符数组赋值

- D. 不可以将单个字符 `'5'` 赋值给一个字符数组 `a`
- C. 用字符 `'1'` 和 `'2'` 初始化，剩余部分将填充空字符 `'\0'`。
- B. 用字符串初始化，编译器会自动确定数组大小
- A. 用字符串初始化，其余部分填充空字符

4.结构体的声明

1. 在结构体中，定义了两个具有相同名称的成员变量 `a`，这是错误的。因为结构体中的成员变量名必须唯一。
2. 结构体的声明缺少分号。正确的写法是 `} a,d;` 后面需要有分号来结束声明。
3. 变量 `d` 的声明缺少类型说明。应该是 `struct d d;` 来正确声明一个结构体变量。

因此，总共有3个错误。

正确的应该是：

```
struct d {
    int a;
    double b;
}; // 在这里加上分号来结束结构体的声明

struct d a, d; // 然后继续声明结构体变量

```

5.自增运算符

1. `(i++)`：首先将 `i` 的值（2）用于表达式，然后 `i` 增量为 3。
2. `(++j)`：首先将 `j` 增量为 3，然后将增量后的值（3）用于表达式。

将这两个部分相加：2（来自 `i++`） + 3（来自 `++j`） = 5。

后置自增总是在表达式结束后进行

7.移位运算符

- 执行完 `unsigned x=10; x<<=2;` 后，变量 `x` 的值将会是 40。
- `x <<= 2` 表示对 `x` 进行左移位操作，移动的位数是右边的操作数（2），即将 `x` 的二进制表示向左移动两位。对于无符号整数，左移操作会在右侧补0。所以，初始值为 10 的二进制表示是 `0000 1010`，左移两位后变为 `0010 1000`，对应的十进制为 40。

  必须有一个可修改的左值，如果是字符的话会移动其ASCLL码
- 左移位的介绍：
  - 用于将一个数的二进制表示向左移动指定的位数。左移操作将数的每一位向左移动，并在右侧用零填充。左移操作通常用 << 符号表示。例如，对于整数 x，x << n 表示将 x 的二进制表示向左移动 n 位。
  - 左移位的效果可以用一个简单的例子来说明：假设有一个十进制数 10（二进制表示为 0000 1010），如果对它进行左移 2 位操作，结果将是 40（二进制表示为 0010 1000）。

  左移位的主要应用包括：

  1. 乘法运算的优化：左移位可以用于乘以2的幂的操作，因为左移 1 位相当于乘以 2，左移 n 位相当于乘以 2^n。
  2. 位操作：左移位常用于位操作中，用于在二进制表示中移动和设置位。
  3. 数据编码：左移位也用于数据编码和加密算法中，以实现数据的移位和重组。

12.switch语句，小心没有break

第一个 case 是 1，所以将执行 case 1 的代码 `i++`，这会将 i 的值增加为 2。

然后，由于没有 `break` 语句，控制流将继续向下执行，执行 case 2 的代码 `i++`，这又会将 i 的值增加为 3。

同样，因为没有 `break` 语句，控制流将继续执行 case 3 的代码 `++i`，这将再次将 i 的值增加为 4。

最后，由于有 `break` 语句，switch 语句结束，控制流不会继续执行其他 case。所以最后 i 的值是 4。

13.switch

A）错误。在 switch 语句中，默认情况（default case）是**可选的**，**不是必需的**。如果没有匹配的 case，程序会执行默认情况之外的代码，但如果没有提供默认情况，程序会继续执行下一条语句，也即结束switch

B）错误。在默认情况（default case）中，break 语句不是必需的。它用于跳出 switch 语句，但如果在默认情况中没有 break 语句，程序会继续执行下一条语句。

C）错误。表达式 `(x>y && a<b)` 只有在两个条件都为 true 的情况下才会返回 true。逻辑与运算符 `&&` 表示当所有操作数都为 true 时，整个表达式才为 true。

D）正确。逻辑或运算符 `||` 在表达式中使用时，如果其任一操作数为 true，则整个表达式将被视为 true。

14.数组

A）错误。数组可以存储相同类型的多个值，但不能存储不同类型的值。数组在声明时需要指定数据类型，其元素必须与该类型一致。

B）错误。数组的下标通常应该是整数类型，比如 int。使用浮点数作为数组下标会导致编译错误。

C）错误。如果初始化列表中的初始化器数量少于数组中元素的数量，剩余的元素不会被初始化为初始化列表中的最后一个值。未初始化的元素将保持未定义状态，其值取决于存储在数组所在内存位置的内容，这可能导致不确定的行为。

D）正确。如果初始化器列表包含的初始化器数量超过数组元素的数量，编译器将会报错。这是因为数组的大小已经确定，无法容纳多余的初始化值。

15.指针

A）正确。任何类型的指针都可以被赋值给 void 指针。void 指针是一种通用的指针类型，可以指向任何类型的数据。

B）错误。取地址操作符 & 可以应用于变量，而不仅仅是常量或表达式。它用于获取变量的内存地址。

C）错误。声明为 void 类型的指针不能直接进行解引用操作。void 指针通常用于通用的内存操作，如动态内存分配，但它们不能直接用于访问或修改内存中的值。必须将 void 指针转换为特定类型的指针，然后才能对其进行解引用操作。

D）错误。指针之间的赋值通常是允许的，但类型必须兼容。如果尝试将指向一个类型的指针赋值给另一个类型的指针，且这两种类型之间不存在隐式转换，就需要使用强制类型转换来确保类型的兼容性。

18.二维内置数组初始化

A）正确。这是一个合法的数组初始化方式。在这种情况下，数组中的所有元素都将被初始化为0。

B）错误。列数必须对应，行数编译器会生成

C）正确。数组中的元素将按顺序初始化为提供的值。

D）正确。数组中的元素将按顺序初始化为提供的值。

二、程序输出

1.循环，break与continue

k = 6;

注意点：

- 内层循环结束后的`if(!(j%2))`永远为false，因为内层循环结束后`j`为3
- 内层循环在`j`为偶数时跳过该次循环，`j`为奇数时让k自增，每一次的外层循环让`k`加2，执行三次，结果为6

2.指针输出子串

程序执行结果为：（VS调试结果，准确）

```
I Love C++
Love C++
ve C++
 C++
++
```

> 如何理解cout << p，它不会输出地址吗

- 在C++中，`cout << p`的行为取决于`p`的类型和所指向的数据。在这个特定的例子中，`p`是一个指向字符的指针（`char *p`），并且它被初始化为指向一个以空字符`\0`结尾的字符串（C风格字符串）的首地址，即字符串字面量`"I Love C++"`。
- 当你使用`cout << p`时，如果`p`是一个指向字符的指针，并且它指向的内存中存储的是一个以`\0`结尾的字符串，`cout`会从这个地址开始，一直打印到遇到`\0`停止，而不是直接打印出指针的地址值。这是因为`<<`运算符对`char *`类型的特殊处理，它理解这是一个C风格的字符串，从而执行了这样的打印行为。
- 换句话说，`cout << p`在这种上下文中，相当于输出了从指针`p`所指向的内存地址开始的字符序列，直到遇到空字符`\0`，这与直接打印指针的地址（如使用`cout << static_cast<void*>(p)`）是不同的。如果要输出指针自身的地址值，就需要进行显式的类型转换，告诉`cout`你想要打印的是地址而非字符串内容。

> 为什么p++后是指向L而不是空格

`p++`操作确实是使指针`p`向前移动指向下一个字符，但这里可能有一点理解上的混淆。澄清一下程序的执行流程：

1. 初始化阶段：`char *p, a[]= "I Love C++";`定义了一个字符指针`p`和一个字符数组`a`，后者初始化为字符串`"I Love C++"`。接着`p = a;`使得指针`p`指向字符串的首字符`'I'`。
2. 第一次循环：进入循环时，`*p != '\0'`为真，因为`p`指向的是`'I'`，不是结束符`\0`。此时`cout << p << endl;`输出从`p`开始到字符串结束的整个子串，即`"I Love C++"`。然后`p++`使`p`指向下一个字符，即空格。
3. 第二次循环判断前的`p++`：在第一次循环的末尾，执行了一次`p++`，这使得`p`从指向空格变成了指向`'L'`。接下来的判断`if(*p != '\0')`检查`p`是否指向`\0`，由于现在`p`指向`'L'`，因此条件为真，执行内部的`p++`，这一步实际上让`p`从指向`'L'`变成了指向`'o'`。

所以，“`p++`后是指向`L`而不是空格”，实际上是在第一次循环结束到第二次循环开始之间的操作。循环内部的逻辑中，`p`先是指向了空格，然后经过`p++`操作，才指向了`'L'`。但紧随其后的另一个`p++`（在`if`语句内部）立即将`p`移动到了`'o'`。这就是为什么看起来在某些解释中，从空格直接“跳过”到了`'L'`，但实际上是因为连续执行了两次`p++`操作导致的。

注意点

- 指针每次移动两位，空字符`'\0'`和空格`' '`是不同的概念，空字符出现在未显式初始化区域或者字符数组末尾

3.静态常量

输出为：（VS调试结果，准确）

```
16
52
```

该静态常量在一次声明后，始终保持其操作后的值，第一次操作结束，c为12，第二次调用将略过该静态常量的声明，再将c乘以4，得到48，输出结果

4.加密

输出为：（VS调试结果，准确）

```
Enter a four-digit integer :1234
The orginal number is : 1234
The encrypt number is : 0189
```

解释过程：

1. 程序首先初始化所有变量为0，包括`temp`、`value`、`a`、`b`、`c`、`d`。
2. 程序进入do-while循环，并提示用户输入一个四位数。
3. 用户输入

   ```
   1234
   ```

   后，通过

   ```
   cin.get()
   ```

   逐个**读取字符**并减去48（ASCII码中'0'的值为48，这样可以将字符数字转换为其对应的整数值）：
   - `a=1`，`b=2`，`c=3`，`d=4`。
4. 检查每个数字是否在0到9之间，条件满足，因此计算得到`value=1*1000+2*100+3*10+4=1234`。
5. 进入加密逻辑：
   - 首先，每个数字加上7并取模10进行加密变换：`a=(1+7)%10=8`，`b=(2+7)%10=9`，`c=(3+7)%10=0`，`d=(4+7)%10=1`。

     此时a，b，c，d依次为 8，9，0，1
   - 然后，交换数字对：`temp=a; a=c; c=temp;` 和 `temp=b; b=d; d=temp;`，导致最终`a=0`，`b=1`，`c=8`，`d=9`。
6. 输出结果：
   - 原始数字：`1234`
   - 加密后的数字：由于直接输出数字变量没有使用连接符，这里假设期望的输出格式是每个数字紧挨着输出，即`0189`

注意点：

- 调用cin.get( )函数后，用户可以一直输入，直到敲击换行符，这些输入会保存在流中，但get函数每次只提取一个**字符**，请注意是字符，故要减去相应ASCLL码值才可以得到数字

5.类构造析构

输出为：（VS调试，准确）

```
Initalizing default
Initalizing default
0 0
Desdtructor is active
Desdtructor is active
```

比较简单，无需多言

6.递归

输出为：（VS调试，准确）

```
The result is:16
```

每次递归将b取模后判断，a自乘了4次

三、程序填空

1.矩阵求最大值

完整代码：

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

// 需要定义m和n的具体值，这里以示例值为例
const int m = 3;
const int n = 4;

// 声明findmax函数，需要知道列数n
void findmax(int a[][n]);

int main()
{
    int a[m][n] = { {1,2,3,4}, {5,6,7,8}, {9,10,1,2}};
    findmax(a);
    return 0;
}

void findmax(int b[][n])
{
    int max, maxrow, maxcol;
```

```
21
22      // 初始化max，maxrow，和maxcol
23      max = b[0][0]; // 假设第一个元素是最大值
24      maxrow = 0;
25      maxcol = 0;
26
27      for (int i = 0; i < m; i++)
28          for (int j = 0; j < n; j++)
29              if (b[i][j] > max)
30              {
31                  max = b[i][j]; // 更新最大值
32                  maxrow = i;     // 记录最大值所在的行
33                  maxcol = j;     // 记录最大值所在的列
34              }
35
36      cout << "The maximum in the matrix is " << max << endl;
37      cout << "The maximum position is row:" << maxrow << " column:" <<
    maxcol << endl;
38  }
```

2.字符串降序排列

关键点：

- 循环的时候用选择排序找到最大字符的索引，找到就交换值
- 传入char *的数组，也即字符串数组
- 利用了strcmp函数比较两个字符串的大小

strcmp函数介绍

`strcmp` 函数是 C 标准库 `<cstring>` 中的一个字符串比较函数，用于比较两个字符串的大小关系。

其函数原型如下所示：

```
1  int strcmp(const char *str1, const char *str2);
```

函数接受两个参数，分别是待比较的两个字符串指针 `str1` 和 `str2`。函数返回一个整数值，表示两个字符串的大小关系：

- 如果 `str1` 大于 `str2`，返回一个正数。
- 如果 `str1` 小于 `str2`，返回一个负数。
- 如果 `str1` 等于 `str2`，返回 0。

`strcmp` 函数按字典顺序**逐个比较** `str1` 和 `str2` 指向的字符，直到出现以下情况之一：

1. **出现不同的字符，返回它们之间的差值**（`str1[i] - str2[i]`）。
2. **到达字符串末尾（即 `\0`），两个字符串相等，返回 0。**

例如：

- `strcmp("apple", "banana")` **返回负数**，因为 'a' 的 ASCII 值小于 'b' 的 ASCII 值。
- `strcmp("banana", "apple")` 返回正数，因为 'b' 的 ASCII 值大于 'a' 的 ASCII 值。
- `strcmp("apple", "apple")` 返回 0，因为两个字符串相等。

在上面的程序中，`strcmp` 函数用于比较字符串，并在排序时确定字符串的顺序。

注意

返回的值是根据ASCLL码相减的结果，运用时要注意if的条件，从大写A到小写z，则应该是相减结果小于0时进行交换

完整代码：

```
#include <iostream>
using std::cout;
using std::endl;
#include <cstring>

// 添加参数类型为char*的数组和字符串数组的长度
void sortstr(char *strs[], int n)
{
    int i, j, k;
    char *temp;

    for(i = 0; i < n - 1; i++)
    {
        k = i;
        for(j = i + 1; j < n; j++)
            // 比较条件应该是当前字符串是否小于后面的字符串，如果是则交换索引
            if(strcmp(strs[k], strs[j]) < 0) // 比较字符大小，利用strcmp方法，
                k = j;

        // 将找到的最大字符串与当前位置的字符串交换
        if(k != i)
        {
            temp = strs[i];
            strs[i] = strs[k];
            strs[k] = temp;
        }
    }
}

int main()
{
    int i;
    char *pname[4] = {"pascal", "Visual Basic", "Visual C++", "lisp"};

    // 调用sortstr函数，传入字符串数组和数组的长度
    sortstr(pname, 4);

```

```
    for(i = 0; i < 4; i++)
        cout << pname[i] << endl;

    return 0;
}
```

3.二分查找

关键点：

- 查找前请确保数组已经完成排序，要根据排序情况来确定二分查找的具体实现

完整代码：

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

int binarySearch(const int[], int, int, int, int);

int main()
{
    const int arraySize = 15;
    int a[arraySize], key;
    for (int i = 0; i < arraySize; i++) // create some data
        a[i] = 2 * i;
    cout << "Enter a number between 0 and 28: ";
    cin >> key;
    int result = binarySearch(a, key, 0, arraySize - 1, arraySize);
    if (result != -1)
        cout << '\n' << key << " found in array element " << result <<
endl;
    else
        cout << '\n' << key << " not found" << endl;
    return 0;
}

int binarySearch(const int b[], int searchKey, int low, int high, int
size)
{
    int middle;
    while (low <= high)
    {
        middle = (low + high) / 2;
        if (searchKey == b[middle]) // match
            return middle;
        else if (searchKey < b[middle])
            high = middle - 1;
        else
            low = middle + 1;
    }
```

```
    return -1;
} // end function binarySearch
```

填空内容为：

1. `while (low <= high)`，循环条件为 `low` 小于等于 `high`。
2. `middle = (low + high) / 2;`，计算中间位置的索引值。
3. `if (searchKey == b[middle])`，如果搜索键等于数组中间元素的值，则找到了匹配项。
4. `else if (searchKey < b[middle])`，如果搜索键小于数组中间元素的值，则更新 `high`。
5. `else`，否则，更新 `low`。

这段程序实现了二分查找算法，用于在有序数组中查找指定值的位置。

四、编写程序，矩形类

关键点：

- 利用cmath工具计算了距离
- isRectangle函数，该类的私有成员函数，无需被外部调用，实现了封装

  用于判断给定的四个点是否能构成一个矩形。该函数采用了一个简单的方法来检查矩形的特性：

  1. 它计算了四条边的长度，并检查相邻两条边的长度是否相等。如果相邻的两条边的长度相等，那么这个四边形是一个平行四边形（可能是矩形或菱形），但并不一定是矩形。
  2. 它检查对角线的长度是否相等。对于矩形来说，对角线的长度应该相等，而且对角线互相垂直。

```
private:
    bool isRectangle(double x1, double y1, double x2, double y2, double x3, double y3, double x4, double y4) {
        double distance1 = pow((x2 - x1), 2) + pow((y2 - y1), 2);
        double distance2 = pow((x3 - x2), 2) + pow((y3 - y2), 2);
        double distance3 = pow((x4 - x3), 2) + pow((y4 - y3), 2);
        double distance4 = pow((x1 - x4), 2) + pow((y1 - y4), 2);

        double diagonal1 = pow((x3 - x1), 2) + pow((y3 - y1), 2);
        double diagonal2 = pow((x4 - x2), 2) + pow((y4 - y2), 2);

        return (distance1 == distance3) && (distance2 == distance4) && (diagonal1 == diagonal2);
    }

```

完整代码：

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

class Rectangle {
private:
    double x1, y1, x2, y2, x3, y3, x4, y4;

public:
    Rectangle(double xx1, double yy1, double xx2, double yy2, double xx3, double yy3, double xx4, double yy4) {
        setCoordinates(xx1, yy1, xx2, yy2, xx3, yy3, xx4, yy4);
    }

    void setCoordinates(double xx1, double yy1, double xx2, double yy2, double xx3, double yy3, double xx4, double yy4) {
        if (xx1 >= 0 && xx1 <= 20.0 && yy1 >= 0 && yy1 <= 20.0 &&
            xx2 >= 0 && xx2 <= 20.0 && yy2 >= 0 && yy2 <= 20.0 &&
            xx3 >= 0 && xx3 <= 20.0 && yy3 >= 0 && yy3 <= 20.0 &&
            xx4 >= 0 && xx4 <= 20.0 && yy4 >= 0 && yy4 <= 20.0) {
            if (isRectangle(xx1, yy1, xx2, yy2, xx3, yy3, xx4, yy4)) {
                x1 = xx1; y1 = yy1;
                x2 = xx2; y2 = yy2;
                x3 = xx3; y3 = yy3;
                x4 = xx4; y4 = yy4;
            } else {
                cerr << "Error: Supplied coordinates do not form a rectangle!" << endl;
                exit(1);
            }
        } else {
            cerr << "Error: Coordinates must be in the first quadrant and <= 20.0!" << endl;
            exit(1);
        }
    }

    double getLength() {
        double length1 = sqrt(pow((x2 - x1), 2) + pow((y2 - y1), 2));
        double length2 = sqrt(pow((x4 - x3), 2) + pow((y4 - y3), 2));
        return max(length1, length2);
    }

    double getWidth() {
        double width1 = sqrt(pow((x4 - x1), 2) + pow((y4 - y1), 2));
        double width2 = sqrt(pow((x3 - x2), 2) + pow((y3 - y2), 2));
        return min(width1, width2);
    }

```

```
    double getPerimeter() {
        return 2 * (getLength() + getWidth());
    }

    double getArea() {
        return getLength() * getWidth();
    }

    bool isSquare() {
        return getLength() == getWidth();
    }

private:
private:
    bool isRectangle(double x1, double y1, double x2, double y2, double x3, double y3, double x4, double y4) {
        double distance1 = pow((x2 - x1), 2) + pow((y2 - y1), 2);
        double distance2 = pow((x3 - x2), 2) + pow((y3 - y2), 2);
        double distance3 = pow((x4 - x3), 2) + pow((y4 - y3), 2);
        double distance4 = pow((x1 - x4), 2) + pow((y1 - y4), 2);

        double diagonal1 = pow((x3 - x1), 2) + pow((y3 - y1), 2);
        double diagonal2 = pow((x4 - x2), 2) + pow((y4 - y2), 2);

        return (distance1 == distance3) && (distance2 == distance4) && (diagonal1 == diagonal2);
    }

};

int main() {
    Rectangle rect(1, 1, 6, 1, 1, 5, 6, 5);
    cout << "Length: " << rect.getLength() << endl;
    cout << "Width: " << rect.getWidth() << endl;
    cout << "Perimeter: " << rect.getPerimeter() << endl;
    cout << "Area: " << rect.getArea() << endl;
    cout << "Is square? " << (rect.isSquare() ? "Yes" : "No") << endl;
    return 0;
}

```