数据结构与算法

---

1 数据结构

1.1 链表

链表转置：第一次循环，当前结点为p1结点，要实现逆序，即当前结点的next指针应当指向其前结点，先用next保存其后结点p2，再令p1的next指针指向prev（此时prev为空），之后当前结点移到p2结点，其前结点变为p1，故先用prev保存p1结点。重复以上过程，直到每个结点的next指针都指向它的前结点。

```
typedef struct Node{
    int data;
    struct Node *next;
} *Link;

Link revList(Link head)
{
    Link prev = NULL, next = NULL;  //当前结点的前/后一个结点
    Link cur;   //当前节点
    if(!head || !head->next) return head;
    cur = head->next;
    while(cur != NULL){
        //next结点存储当前结点的下一结点
        next = cur->next;
        //将当前结点的next指针指向prev结点
        cur->next = prev;
        //prev结点存储当前结点
        prev = cur;
        //当前结点后移一个单位
        cur = next;
    }
    //此时prev存储未逆序前的最后一个结点，逆序后变为首元结点
    head->next = prev;
    return head;
}
```

链表排序：插入的过程中实现排序。

```c
typedef struct Node{
    int data;
    struct Node *next;
}Node;
/* 从小到大排序 */
void sortInsert(Node *head, int data){
    Node *tmp = head;  //头结点
    Node *cur = (Node *)malloc(sizeof(Node));  //要插入的结点
    if(!cur) return;
    cur->data = data;
    cur->next = NULL;
    if(!head->next){
        head->next = cur;
        return;
    }
    while(tmp->next){
        if(tmp->next->data > tmp->data){
            cur->next = tmp->next;
            tmp->next = cur;
            return;
        }
        tmp = tmp->next;
    }
    tmp->next = cur;
}
```

链表的优缺点：随机插入/删除快，时间复杂度为O(1)；查找慢，时间复杂度为O(N)。

1.2 二叉树

3.1 结点数

问题：将一颗有111个结点的完全二叉树从根这一层开始，每一层从左到右依次对结点进行编号，根结点编号为1，则编号最大的非叶子结点编号为?

解答：由完全二叉树性质可知该树有7层，前六层为满二叉树，结点数为 63 个。第7层结点数为 111 - 63 = 48 个。故第六层中非叶节点个数为 48 / 2 = 24 个。前五层的结点数为 31 个，故最大非叶结点编号为 31 + 24 = 55 。

3.2 遍历

问题：若一个二叉树的前序遍历结果是abefcgd，下面哪个不可能是它的中序遍历：

A. ebfagcd B. ebafgcd C.bfaegcd D. aebfgcd。

解答：由前序遍历结果可知此二叉树的根为 a ，逐一对选项进行分析，A选项的中序遍历可知 ebf 为左子树，gcd 为右子树，结合前序遍历结果可知左子树的根结点为 b ，右子树的根结点为 c 。故A选项成立。同理B选项成立。而C选项中左子树为 bf 。结合前序遍历可知左子树不可能为 bf 。故答案为C选项。

注意：由前序遍历和中序遍历、中序遍历和后序遍历可以唯一确定一棵树，而由前序遍历和后序遍历不能唯一确定。前序遍历的第一个结点即为根结点，后序遍历的最后一个结点即为根结点。

2 常用算法

2.1 其他算法

二分查找：

```c
/* 循环实现 */
int BinarySearch(const int a[], int x, int N)
{
    int low, mid, high;
    low = 0;
    high = N - 1;
    while(low <= high)
    {
        mid = low + (high - low) / 2;
        if(x < a[mid]) high = mid - 1;
        else if(x > a[mid]) low = mid + 1;
        else return mid;
    }
    return -1;
}
/* 递归实现 */
int BinarySearch(int a[], int low, int high, int x){
    int mid = low + (high - low) / 2;  //重要，注意理解
    if(low <= high){
        if(x < a[mid]) return BinarySearch(a, low, mid, x);
        else if(x > a[mid]) return BinarySearch(a, mid+1, high, x);
        else return mid;
    }else return -1;
}
```

判断大小端：**大端**是指数据的高字节保存在内存的低地址中，而数据的低字节保存在内存的高地址中。**小端**则正好相反。

方法一：通过指针地址判断

```c
int check(){
    int num = 0x12345678;
    //(char*)&num获得num的低8位地址
    return (*((char *)&num) == 0x12);
}
//大端返回1，小端返回0
```

方法二：通过联合体判断

联合体union的存放顺序是所有成员都从低地址开始存放。c.a = 1，即0x00000001，若处理器为大端，则低字节0x01存放在高地址，故b的值为0，若处理器为小端，则低字节0x01存放在低地址，故b的值为1。

```c
int check(){
    union w{
        int a;
        char b;
    }c;
    c.a = 1;
    return (c.b == 1);
}
//大端返回0，小端返回1
```

分解质因数：**质因数**（质因子）是指能整除给定正整数的质数。例如6的质因数是3和2。把一个合数分解成若干个质因数的乘积的形式，即求质因数的过程叫做**分解质因数**。分解质因数的方法为：

- 如果这个质数恰等于n，则说明分解质因数的过程已经结束，打印出即可。
- 如果n>k，但n能被k整除，则应打印出k的值，并用n除以k的商作为新的正整数n，重复执行第一步。
- 如果n不能被k整除，则用k+1作为k的值，重复执行第一步。

```
void prim(int m){
    int n = 2;
    if(m >= n){
        while(m % n) n++;
        m /= n;
        prim(m, n)
    }
}
```

2.2 排序算法

冒泡法：

快速排序：

3 C/C++

3.1 C和C++的区别

C和C++：

机制不同：C是面向过程的（但也可以编写面向对象的程序）；C++是面向对象的，提供了类。

适用领域不同：C适合代码体积小，效率高的场合；C++适合更上层的，复杂的。

侧重点不同：C++侧重于对象而不是过程，侧重于类的设计而不是逻辑。

面向对象与面向过程：

面向对象：把数据及对数据的操作方法放在一起，作为一个相互依存的整体，即对象。对同类对象抽象出其共性，即类，类中的大多数数据，只能被本类的方法进行处理。类通过一些简单的外部接口与外界发生关系，对象与对象之间通过消息进行通信。

面向过程：一种以事件为中心的开发方法，就是自顶向下顺序执行，逐步求精，其程序结构是按功能划分为若干个基本模块，这些模块形成一个树状结构，各模块之间的关系也比较简单，在功能上相对独立，每一模块内部一般都是由顺序、选择和循环三种基本结构组成的，

C和C++的struct：

- C语言的struct不能有函数成员，而C++的struct可以有。
- C语言的struct中数据成员没有private、public和protected访问权限的设定，而C++ 的struct的成员有访问权限设定。
- C语言的struct是没有继承关系的，而C++的struct却有丰富的继承关系。

C语言中的struct没有权限设置，是一些变量的集合体，可以封装数据却不可以隐藏数据，而且成员不可以是函数。

C++中的 struct与class的区别

- 默认继承权限不同。class继承默认是private继承，而struct继承默认是public继承。
- class还用于定义模板参数，就像typename，但关键字struct不用于定义模板参数。

引用和指针：

- 引用只能在定义时被初始化一次，之后不能被改变，具有**从一而终**的特性。而指针却是可变的。
- 引用使用时不需要解引用（*），而指针需要解引用。

- 引用不可以为空，而指针可以为空。
- 对引用进行sizeof操作得到的是所指向的变量（对象）的大小，而对指针进行sizeof操作得到的是指针本身（所指向的变量或对象的地址）的大小。
- 作为参数传递时，两者不同。

malloc/free和new/delete：

- malloc/free 是 C++/C 语言的标准库函数，new/delete 是 C++的运算符。它 们都可用于申请动态内存和释放内存。
- maloc/free 无法满足动态对象的要求，不能执行构造函数和析构函数。对象在创建的同时要自动执行构造函数，对象在消亡之前要自动执行析构函数。

3.2 关键字

static关键字：

- 在函数体内，被声明为静态的变量只初始化一次，以后该函数再被调用，将不会再初始化。
- 在模块内（但在函数体外），如果把一个变量或者函数声明为静态的，那么可以将其作用域被限制在本模块内，起一个“隐藏”的作用，避免命名冲突。
- 默认初始化为0，因为静态变量存储在静态数据区，而静态数据区中的所有字节默认值都是0x00。
- 在C++中，在类中声明static变量或者函数。其初始化时使用作用域运算符来标明它所属类，因此，静态数据成员是类的成员，而不是对象的成员。

const关键字：

- 定义常量，使其值不可被修改，且编译器可以对其进行类型检查。
- 修饰函数形参，防止其值被意外修改，提高程序健壮性。
- 修饰常量指针`const char * p`和指针常量`char * const p`。
- 修饰函数返回值。
- 在C++中，修饰类成员函数，任何不会修改数据成员的函数都应该用const修改。以及修饰类成员数据。

volatile关键字：

volatile是一个类型修饰符，被其修饰的变量，编译器不会对其进行优化。所以每次用到它的时候都是直接从对应的内存当中提取，而不会利用cache(缓存)或寄存器中的原有数值。一般用来修饰多线程间被多个任务共享的变量和并行设备硬件寄存器等。

3.3 字符串

3.4 C++

虚函数及其作用：

指向基类的指针在操作它的多态类对象时，可以根据指向的不同类对象调用其相应的函数，这个函数就是虚函数。

虚函数的作用：在基类定义了虚函数后，可以在派生类中对虚函数进行重新定义，并且可以通过基类指针或引用，在程序的运行阶段动态地选择调用基类和不同派生类中的同名函数。

使用虚函数时，需要注意以下几个方面的内容（重要）：

- 只需要在声明函数的类体中使用关键字virtual将函数声明为虚函数，而定义函数时不需要使用关键字virtual。
- 当将基类中的某一成员函数声明为虚函数后，**派生类中的同名函数自动成为虚函数。**
- 非类的成员函数、全局函数以及类的中静态成员函数和构造函数也不能定义为虚函数，但可以将析构函数定义为虚函数。
- 基类的析构函数应该定义为虚函数，否则会造成内存泄漏。

重载、重写和重定义

重载（overload）：

在同一个作用域内；函数名相同，参数列表不同（参数**个数不同**，或者参数**类型不同**，或者**参数个数和参数类型都不同**），返回值类型可相同也可不同；这种情况叫做c++的重载！

c++函数重载达到的效果：调用函数名相同的函数，会根据实参的类型和实参顺序以及实参个数选择相应的函数。c++函数重载是一种静态多态。

重写（override）：

当在子类中定义了一个与父类**完全相同**的虚函数时，则称子类的这个函数**重写（也称覆盖）**了父类的这个虚函数。

覆盖（重写）达到的效果：在子类中重写了父类的虚函数，那么子类对象调用该重写函数，调用到的是子类内部重写的虚函数，而并不是从父类继承下来的虚函数（这其实就是动态多态的实现）。

覆盖（重写）的两个必要条件：父类函数为虚函数，并且父类和子类函数的函数名、参数个数、参数类型等都必须相同。

重定义（redefine）：

重定义（隐藏）是指派生类的函数屏蔽了与其同名的基类函数，隐藏的**不光是类的成员函数**，还可以是**类的成员变量**；规则如下：

- 如果派生类的函数与基类的函数同名，但是参数不同，则不论有无virtual关键字， 基类的函数都将被隐藏。
- 如果派生类的函数与基类的函数同名，并且参数也相同，但是基类函数没有virtual 关键字，此时基类的函数被隐藏。

3 Linux环境编程

3.1 进程和线程

- 进程是资源分配的最小单位；线程是程序执行的最小单位，也是处理器调度的基本单位。
- 进程有自己的独立地址空间，每启动一个进程，系统就会为它分配地址空间，建立数据表来维护代码段、堆栈段和数据段；线程有自己的堆栈，共享进程中的数据。
- CPU切换和创建一个线程的开销比进程小很多。
- 进程间通信要以IPC方式进行；线程间通信更方便，同一进程下的线程共享全局变量、静态变量等数据。
- 每个独立进程有一个程序运行的入口；线程不能独立执行，必须依存在应用程序中。

互斥：指对于共享的进程系统资源，在各单个线程访问时的排它性。当有 若干个线程都要使用某一共享资源时，任何时刻最多只允许一个线程去使用， 其它要使用该资源的线程必须等待，直到占用资源者释放该资源。

同步：指线程之间所具有的一种制约关系，能实现对资源的有序访问，一个线程的执行依赖另一个线 程的消息，当它没有得到另一个线程的消息时应等待，直到消息到达时才被唤醒。

3.2 进程间通信

管道：一种半双工的通信方式，数据只能单向流动，且只能在具有亲缘关系的进程间使用。使用简单方便。

有名管道(FIFO)：一种半双工的通信方式，但允许无亲缘关系的进程间通信。长期存于系统中，使用不当易出错，缓冲区有限。

信号量：一个计数器，可以用来控制多个线程对共享资源的访问。它不用于交换大批数据, 而用于进程间以及同一个进程内不同线程之间的同步手段。缺点：信号量有限。

消息队列：消息的链表，存放在内核中并由消息队列标识符标识，可以实现任意进程间的通信，并通过系统调用函数来实现消息发送和接收之间的同步，无需考虑同步问题，方便。缺点：信息的复制需

要额外消耗CPU的时间，不适宜于信息量大或操作频繁的场合。

共享内存：映射一段能被其他进程所访问的内存，这段共享内存由一个进程创建，但多个进程都可以访问，共享内存是最快的IPC方式， 数据的共享使进程间的数据不用传送，而是直接访问内存，加快了程序的效率。共享内存没有提供同步的机制。

3.3 信号量与互斥锁

- 互斥量用于线程的互斥，信号量用于线程的同步。**互斥**是指某一资源同时只允许一个访问者对其进行访问，具有唯一性和排它性。但互斥无法限制访问者对资源的访问顺序，即访问是无序的。**同步**是指在互斥的基础上（大多数情况），通过其它机制实现访问者对资源的有序访问。在大多数情况下，同步已经实现了互斥，特别是所有写入资源的情况必定是互斥的。
- 互斥量值只能为0/1，信号量值可以为非负整数。也就是说，一个互斥量只能用于一个资源的互斥访问，它不能实现多个资源的多线程互斥问题。
- 互斥量的加锁和解锁必须由同一线程分别对应使用，信号量可以由一个线程释放，另一个线程得到。

4 网络编程

1.1 TCP/IP

OSI模型：物理层、数据链路层、网络层、传输层、会话层、表示层、应用层。

TCP/IP模型：物理层、数据链路层、网络层(IP)、传输层(TCP UDP)、应用层(HTTP FTP) 。

TCP建立连接(三次握手)：

TCP断开连接(四次挥手)：

TCP和UDP：

- TCP面向连接；UDP是无连接的，即发送数据前无需建立连接。
- TCP提供可靠的服务，通过校验和，丢包时的重传控制，序号标识，滑动窗口、确认应答，次序乱掉的分包进行顺序控制实现可靠传输，通过TCP连接传送的数据无差错、不丢失、不重复且按序到达；UDP不保证可靠交付。
- UDP具有较好的实时性，工作效率高，适用于对高速传输和实时性有较高要求的通信场景。
- TCP连接只能是点到点的；UDP支持一对一、一对多、多对一和多对多的交互通信方式。
- TCP对系统资源要求较多，UDP对系统资源要求较少。

1.2 HTTP