1.多态

子类型多态（虚函数）、强制多态（自定义数据类型/基本数据类型强转）、参数多态（类模板、函数模板）、重载多态（函数重载、操作符重载）

2.map底层原理。hash_map的实现?

map底层是基于红黑树实现的，因此map内部元素排列是有序的。unordered_map底层则是基于哈希表实现的，因此其元素的排列顺序是杂乱无序的。

·map底层为红黑树查找大致为logN的时间复杂度；unordered_map底层是闭散列的哈希桶，查找为O(1)，性能更优。

·调用insert操作，map相较于unordered_map操作慢，大致有2到3倍差异；但是map插入更加稳定

·unordered_map的erase操作会缩容，导致元素重新映射，降低性能。

3.sizeof和strlen区别

strlen计算字符串的长度，以'\0'为字符串结束标志

sizeof是分配的数组实际所占的内存空间大小，不受里面存储内容

sizeof()是运算符，由于在编译时计算，因此sizeof不能用来返回动态分配的内存空间的大小。实际上，用sizeof来返回类型以及静态分配的对象、结构或数组的空间，返回值跟这些里面所存储的内容没有关系。

具体而言，当参数分别如下时，sizeof返回的值含义如下：

数组-编译时分配的数组空间大小

指针-存储该指针所用的空间大小

类型-该类型所占的空间的大小

对象-对象的实际占用空间大小

函数-函数返回类型所占空间的大小

4.字符串复制函数的弊端

比方说，字符串的结尾是'\0'，也是占一个字符空间的，那么如果我们在利用strcpy拷贝字符串的时候，应该多加1个字符空间，就是专门留给这个'\0'的。

```
char str[] = "MengLiang";
//此处分配空间没有考虑到'\0'
//应该+1
char* New_str = (char*)malloc(strlen(str));
strcpy(New_str, str);
```

strcpy只是复制字符串，但不限制复制的数量。很容易造成缓冲溢出(多出来将一部分将会覆盖原来的内存单元)，也就是说，不过dest有没有足够的空间来容纳src的字符串，它都会把src指向的字符串全部复制到从dest开始的内存。

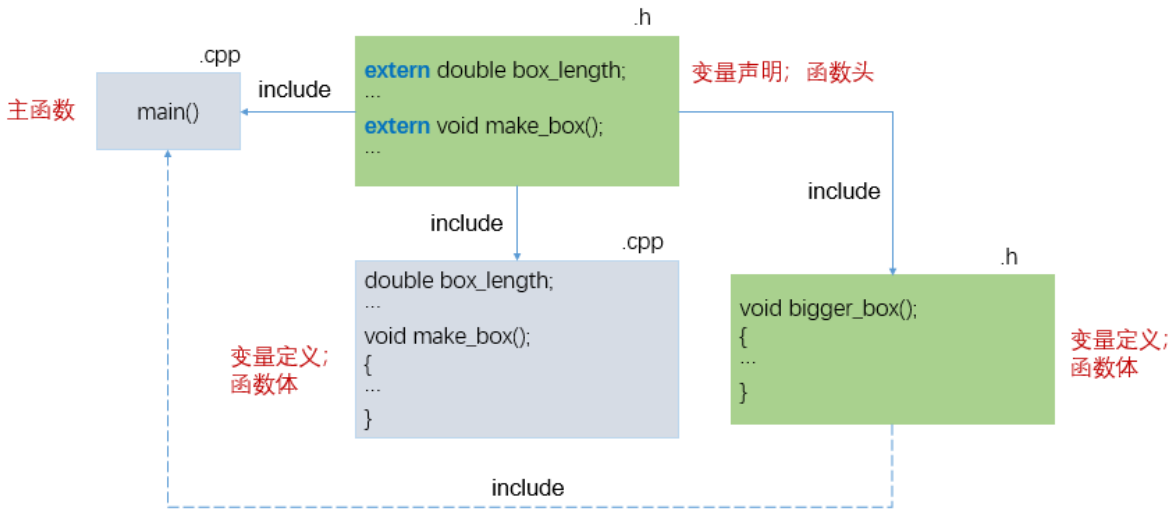
为了避免这个问题，我们可以使用memcpy来完成的字符串的拷贝工作，memcpy是用来在内存中复制数据的，它会把指定长度的内存块复制到另一块内存中而不关内存的中存放的是什么数据，同时也要防止内存的溢出。

```
memcpy(b, a,sizeof(b));//注意如用sizeof(a)，会造成b的内存地址溢出
```

5.extern关键字

extern是一种"**外部声明**"的关键字，字面意思就是**在此处声明**某种变量或函数，**在外部定义。**

extern关键字的主要作用是**扩大变量/函数的作用域**，使得其它源文件和头文件可以复用同样的变量/函数，也起到类似"分块储存"的作用，划分代码。如图所示，在一个头文件里做了外部声明，就能把变量的定义部分和函数体的实现部分转移到其它地方了。



extern "C" {}： 按照C语言的方式来编译。

6.c++怎么实现多人聊天。讲讲listen怎么用?

```
https://www.cnblogs.com/qkqBeer/articles/11012664.html
```

7.vector增长策略，扩容机制

当元素个数超出当前容量时，超出的部分并不是在原有空间后追加，因为无法保证源空间之后尚有可分配的空间，而是申请更大的空间，然后将数据拷贝进新空间并释放原空间（vector会将自身容量扩大为原来的两倍），扩充空间需要经过的步骤

8.快排的最好、最差、平均情况

最好：每次均等划分 O(nlogn)

最差：正序和倒序 O(n^2)

平均：O(nlogn)

9.main执行前会做什么工作

1.设置栈指针

2.初始化static静态和global全局变量，即data段的内容

3.将未初始化部分的赋初值：数值型short，int，long等为0，bool为FALSE，指针为NULL，等等，即.bss段的内容

4.运行全局构造器，类似c++中全局构造函数

5.将main函数的参数，argc，argv等传递给main函数，然后才真正运行main函数

首先`main()`函数只不过是提供了一个函数入口，在`main()`函数中的显示代码执行之前，会由编译器生成_main函数，其中会进行所有全局对象的构造以及初始化工作。简单来说对静态变量、全局变量和全局对象来说的分配是早在`main()`函数之前就完成的，然后传参给main函数执行，所以`C/C++`中并非所有的动作都是由于`main()`函数引起的。

同理在`main()`函数执行后，程序退出，这时候会对全局变量和全局对象进行销毁操作，所以在`main()`函数还会执行相应的代码。

类的构造函数会先执行，再执行main，最后会调用类的析构函数。

10.strcpy与memcpy的区别，讲一下strncpy与memcpy的区别。

strncpy这个函数会出现三种情况：
1、num<source串的长度（包含最后的'\0'字符）：那么该函数将会拷贝source的前num个字符到destination串中（**不会自动为destination串加上结尾的'\0'字符**）；
2、num=source串的长度（包含最后的'\0'字符）：那么该函数将会拷贝source的全部字符到destination串中（包括source串结尾的'\0'字符）；
3、num>source串的长度（包含最后的'\0'字符）：那么该函数将会拷贝source的全部字符到destination串中（包括source串结尾的'\0'字符），**并且在destination串的结尾继续加上'\0'字符**，直到拷贝的字符总个数等于num为止。

11.手写memcpy

```
void *Memcpy(void *dst, const void *src, size_t size)
{
```

```
    if (dst == NULL || src == NULL)
        return NULL;

    char *psrc;
    char *pdst;

    //地址重叠的情况
    if ((src < dst) && (char*)src + size > (char *)dst)
    {
        psrc = (char*)src + size - 1;
        pdst = (char*)dst + size - 1;
        while(size--)
        {
            *pdst-- = *pdst--;
        }
    }else {
        psrc = (char*)src;
        pdst = (char*)dst;
        while(size--)
        {
            *pdst++ = *psrc++;
        }
    }
    return dst;

}
```

12.红黑树为什么查找速度更快？讲讲红黑树是怎么实现自平衡的？

AVL树为了维护平衡要不断调整，付出了更多的代价；而红黑树并不是一味追求平衡因子为1的平衡，而是近似平衡，因此在插入删除时会减少大量左旋右旋操作，性能更加稳定。

红黑树五大原则**(保证了自平衡)**:
(1)节点为红色或黑色
(2)根节点为黑色
(3)红色节点的子节点和父节点不能为红色
(4)从任意节点到所有叶子节点的路径中黑色节点个数相同
(5)叶子节点为黑色

13.比红黑树查找更快的数据结构?

冲突较小的哈希表。

14.函数指针了解吗？如何定义和初始化函数指针？函数指针通常用在什么地方?

定义

每一个函数都占用一段内存单元，它们有一个起始地址，指向函数入口地址的指针称为函数指针。是一个**指针变量。**

```
int (*p)(int a, int b); //p是一个指向函数的指针变量，所指函数的返回值类型为整型

int *p(int a, int b); //p是函数名，此函数的返回值类型为整型指针
```

初始化

```
//如函数max的原型为：int max(int x, int y);
//指针p的定义为：int (*p)(int a, int b);
//则p = max;的作用是将函数max的入口地址赋给指针变量p。这时，p就是指向函数max的指针变量，也就是p和max都指向函数的开头。

在一个程序中，指针变量p可以先后指向不同的函数，但一个函数不能赋给一个不一致的函数指针（即不能让一个函数指针指向与其类型不一致的函数）。

如有如下的函数：int fn1(int x, int y); int fn2(int x);

定义如下的函数指针：int (*p1)(int a, int b); int (*p2)(int a);

则

p1 = fn1; //正确

p2 = fn2; //正确

p1 = fn2; //产生编译错误
```

用途：

```
//将函数作为参数传递给函数，或者回调函数

//Calculate用于计算积分。一共三个参数。第一个为函数指针func，指向待积分函数。二三参数为积分上下限
double Calculate(double(*func)(double x), double a, double b) ;
```

15.用宏实现一个返回两个数最大值的功能?

```
#define MAX(x, y) \
        ((x) > (y) ? (x) : (y))
```

可能会不安全，比如参数类型不一样。

```
#define MAX_4(x, y) ({\
        typeof(x) _x = (x); \
        typeof(y) _y = (y); \
        (void)(&_x == &_y); \
        _x > _y ? _x : _y; \
})
```

//MAX_4宏定义，通过typeof关键字，来获取参数的类型，并保存参数的一份拷贝，防止参数副作用影响对比结果，再通过(void)(&_x == &_y);来对比两个参数类型，如果不是同一种类型，在编译阶段就会报出warning，引起开发者注意，提前消灭隐患。

16.用宏实现的话有什么不安全的地方?

- 宏定义实际上是字面的替换，所以如果忘记加括号的话可能会误用
- 而且宏定义不会进行类型检查，也不安全。

```
#define T(a,b) a+b
...
int x=,x=,x=,x;
x=T(x,x)*x;
```

17.如何防止头文件被重复引用?

```
//方法一：
#ifndef _HEADERNAME_H
#define _HEADERNAME_H


...//(头文件内容)


#endif

//方法二：
#pragma once
...//(头文件内容)
```

18.给定一个字符串，求最长的重复子串？（比如abcdefgabcde的最长重复子串是abcde)

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

```
//给定一个字符串,求出其最长的重复子串
//方法一
string lsubstr_1(const string & str)
{
    vector<string> vs;
    for (int i = 0; i < str.size(); i++)
        vs.push_back(str.substr(i));
    sort(vs.begin(), vs.end());
    int max = 0;
    int flag = 0;
    for (int i = 0; i <( vs.size()-1); i++)
    {
        int j = 0;
        while (vs[i][j] == vs[i + 1][j] && j<vs[i].size() && j<vs[i+1].size())
            j++;
        if (j>max)
        {
            max = j;
            flag = i;
        }
    }
    return vs[flag].substr(0, max);
}

//方法二
string lsubstr_2(const string & str)
{
    string maxstr;
    for (int i = 0; i < str.size();i++)
    for (int j = (str.size() - i); j >=1 ; j--)
    {
        string subs = str.substr(i, j);
        int front = str.find(subs);
        int back = str.rfind(subs);
        if (front != back && subs.size() > maxstr.size())
            maxstr = subs;
    }
    return maxstr;
}

//方法三
string lsubstr_3(const string & str)
{
    string maxstr;
    for (int i = 0; i < str.size(); i++)
        for (int j = 0; j < i; j++)
        {
            string temp;
            int k = j;
            int m = i;
            while (str[m] == str[k] && i<str.size() && k<str.size())
            {
                m++; k++;
            }
            temp = str.substr(j, k - j);
            if (temp.size()>maxstr.size())
                maxstr = temp;
        }
```

```
    return maxstr;
}
```

19.假设有大量的IP地址的访问需要用hash进行存储，如何减少冲突?

多级哈希减少冲突

20.vector是如何实现的？和C的数组有什么区别?

array是静态空间，一旦配置好了就不能改变了，如果程序需要一个更大的array，只能自己再申请一个更大的array，然后将以前的array中的内容全部拷贝到新的array中。

vector是动态空间，是线性的连续空间。随着元素的加入，它的内部机制会自动扩充空间以容纳新的元素。

vector在增加元素时，如果超过自身最大的容量，vector则将自身的容量扩充为原来的两倍。扩充空间需要经过的步骤：重新配置空间，元素移动，释放旧的内存空间。一旦vector空间重新配置，则指向原来vector的所有迭代器都失效了，因为vector的地址改变了。

21.vector在插入新元素时如何申请内存的?

如上

22.C和C++动态申请内存的区别?

c使用malloc和free，c++则是new和delete。

C 语言的malloc() 和free() 并不会调用析构函数和构造函数。C++的 new 和 delete 操作符 是 “类意识”，并且当调用new的时候会调用类的构造函数和当delete 调用的时候会调用析构函数。

最大的区别在于释放内存的方式，C++分配内存的方式的优势在于，它不再需要计算数据类型的大小，也就是不需要再使用sizeof函数，相对来说，要方便一些。

23.C++内存管理?

1、**栈区（stack）**— 由编译器自动分配释放 ，存放函数的参数值，局部变量的值等。其操作方式类似于数据结构中的栈。

2、**堆区（heap）** — 一般由程序员分配释放， 若程序员不释放，程序结束时可能由OS回收 。注意它与数据结构中的堆是两回事，分配方式倒是类似于链表，呵呵。

3、**全局区（静态区）（static）**—，全局变量和静态变量的存储是放在一块的，**初始化的全局变量和静态变量在一块区域（RW），未初始化的全局变量和未初始化的静态变量在相邻的另一块区域（ZI）**。 - 程序结束后有系统释放

4、**文字常量区** —常量字符串就是放在这里的。 程序结束后由系统释放

5、**程序代码区**—存放函数体的二进制代码。

24.malloc最大可以申请多大的空间?

理想的状态下，最大的字节数等于当前剩余的堆空间大小。

主要看有多少是连续的内存，如果是有2g空闲，但是不连续，也申请不到2G。

25.如果使用new却没有进行delete，会发生什么？一定会发生内存泄露吗?

对象并没有释放，会发生内存泄漏。

也会隐式释放，当执行的程序退出时，系统会自动回收相应的内存等程序运行中所使用的资源。

26.初始化过的static变量和未初始化过的static变量有什么区别?

初始化的全局变量和初始化的静态变量在一块区域，未初始化的全局变量和未初始化的静态变量在相邻的另一块区域

27.谈谈你对函数局部变量生存周期的理解?

全局变量，当程序关闭之后生存结束

局部变量，函数方法执行完毕之后生存周期结束

28.静态局部变量在函数初次调用时和调用后的行为?

> 自动变量如果没有赋初值，其存储单元将是随机值，就静态变量而言，如果定义时没有赋初值，系统将自动赋0，并且赋值只在函数第一次调用时有效，以后调用都按照前一次调用时保留的值使用，因为静态局部变量的生存周期始于函数的第一次调用，贯穿整个程序，函数第一次调用时，静态局部变量的内存单元得以分配，赋以初值，再次调用函数时，此静态局部变量单元已经存在，不会再为其分配单元；
> 静态变量赋初值只在函数第一次调用时起作用，若没有赋初值，系统自动赋0

静态局部变量在编译时赋初值，即只赋初值一次；而对自动变量赋初值是在函数调用时 进行，每调用一次函数重新给一次初值，相当于执行一次赋值语句。

29.如果在三次握手结束后，客户端断电了，服务端可以得知吗？服务端会主动断开连接吗?

TCP连接不能自动查别断连的现象。

TCP中KEEPALIVE机制是默认不打开的，打开后会定时向连接对方发送ACK包(linux下默认是7200s 即2小时发生一次发送一次握手信息)，如果在发送ACK包（也叫心跳包）后对方不回应才能检测道对方的断开信息。

30.malloc如果遇到内存不足以申请的情况怎么办？new失败后会出现啥，怎么解决？new出的空间最大多大

- 内存充足情况下malloc失败，很可能是由于指针越界，对未知的内存做了操作，致使malloc不能继续分配内存，解决办法就是查找最近一次malloc的地方，查看这次malloc申请的内存都做了什么，基本就是最近一次malloc申请的空间出问题了。
- new失败后会返回NULL指针，抛出bad_alloc异常。捕获异常，重新分配更小的内存。

- new分配理想的状态下，最大的字节数等于当前剩余的堆空间大小。主要看有多少是连续的内存，如果是有2g空闲，但是不连续，也申请不到2G。

31.define和const的区别?

1.编译器处理方式

define宏是在预处理阶段展开

const常量是在编译阶段使用

2.类型和安全检查不同

define宏没有类型，不做安全检查，是简单的替换

const常量有具体的类型，在编译阶段会这行类型检查

3.存储方式不同

define宏是展开不分配内存

const常量分配内存空间（堆栈可以）

32.set的底层实现？和map的实现有什么区别?

都是基于红黑树，不过结点存放的分别是值和pair<int,int>。

33.extern c{}介绍

用C语言的方式来编译

34.八大基础排序算法（代码）

35.大端存储、小端存储

大端：低字节存放在高地址

小端：低字节存放在低地址，跟我们的逻辑一样。

```
void BigLittleEndian()
{
   int i  = 1;
   char *p = (char *)(&i);
   if( *p == 1)
    printf("LittleEndian");
   else
    printf("BigEndian");
}
```

36.找出101个数中重复的数算法叙述

```
bool duplicate(int numbers[], int length, int* duplication) {
        for(int i=0;i<length;i++){ //2,3,1,0,2,5,3
            int cur = numbers[i];
            if(cur >= length) cur = cur-length;
            if(numbers[cur] >= length){
                *duplication = cur;
                return true;
            }
            numbers[cur] += length;
        }
        return false;
    }
```

37.实现数据结构--hash表（包括：<key>类型为 int 和 string；数据结构（用的链地址法，所以vector<list>）；可避免冲突的hash函数；插入、删除、euqal方法）

```
#include <vector>
#include <list>
using namespace std;

template <typename Hashedfunc>
class HashTable
{
  public:
    explicit HashTable(int size = 101);

    void makeEmpty()
    {
        for(int i = 0; i < theLists.size(); i++)
            theLists[i].clear();
    }

    bool contains(const Hashedfunc & x) const
    {
        const list<Hashedfunc> & whichList = theLists[myhash(x)];
        return find(whichList.begin(), whichList.end(), x) != whichList.end();
    }

    bool remove(const Hashedfunc & x)
    {
        list<Hashedfunc> & whichList = theLists[myhash(x)];
        typename list<Hashedfunc>::iterator itr = find(whichList.begin(),
whichList.end(), x);
        if(itr == whichList.end())
            return false;
        whichList.erase(itr);
        --currentSize;
        return true;
    }

    bool insert(const Hashedfunc & x)
    {
```

```
        list<Hashedfunc> & whichList = theLists[myhash(x)];
        if(find(whichList.begin(), whichList.end(), x) != whichList.end())
            return false;
        whichList.push_back(x);
        if(++currentSize > theLists.size())
            rehash();
        return true;
    }

  private:
    vector<list<Hashedfunc> > theLists;   // The array of Lists
    int  currentSize;

    void rehash()
    {
        vector<list<Hashedfunc> > oldLists = theLists;

        // Create new double-sized, empty table
        theLists.resize(2 * theLists.size());
        for(int j = 0; j < theLists.size(); j++)
            theLists[j].clear();

        // Copy table over
        currentSize = 0;
        for(int i = 0; i < oldLists.size(); i++)
        {
            typename list<Hashedfunc>::iterator itr = oldLists[i].begin();
            while(itr != oldLists[i].end())
                insert(*itr++);
        }
    }

    int myhash(const Hashedfunc & x) const
    {
        int hashVal = hash(x);

        hashVal %= theLists.size();
        if(hashVal < 0)
            hashVal += theLists.size();

        return hashVal;
    }
};
```

38.有一千万个url，要用到什么数据结构存储?判断域名是否为一百万个域名的后缀?

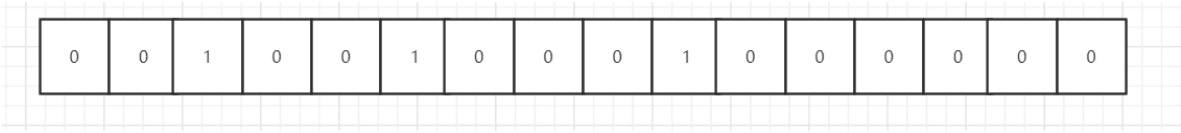
用hashmap分成几千个小文件存储。

用布隆过滤器判断，将100万个域名取它们的后x位（x位域名的长度）映射到布隆过滤器上，然后比较这个域名的映射是否出现。

https://www.cnblogs.com/dong008259/archive/2012/02/17/2355551.html

补充：布隆过滤器

布隆过滤器可以用来判断大量数据中，一个数据是否重复出现，假设有k个hash函数，还有一个长度为m的数组，k个哈希函数可以计算出k个值，分别对应着数组中的k个位置。假如一个元素计算到3个位置分别是1、2、3，那么将数组中这三个位置置位1，一次性读完所有数据后，数组中位置的1就置完了，此时判断一个元素是否在这个海量数据中出现就可以计算它的所有hash的值是否在那个数组中出现。类似如下：



随着数据的增加，误判率随之增加。

39.爬虫遇到登录注册页面怎么办

- 一种方式是手动设置 cookie ，就是先在网站上面登录，复制登陆后的 cookies ，在爬虫程序中手动设置 HTTP 请求中的 Cookie 属性，这种方式适用于采集频次不高、采集周期短，因为 cookie 会失效，如果长期采集的话就需要频繁设置 cookie，这不是一种可行的办法。
- 第二种方式就是使用程序模拟登陆，通过模拟登陆获取到 cookies，这种方式适用于长期采集该网站，因为每次采集都会先登陆，这样就不需要担心 cookie 过期的问题。

40.字节对齐

为了提高效率，计算机从内存中取数据是按照一个固定长度的。以32位机为例，它每次取32个位，也就是4个字节（每字节8个位）。字节对齐有什么好处？以int型数据为例，如果它在内存中存放的位置按4字节对齐，也就是说1个int的数据全部落在计算机一次取数的区间内，那么只需要取一次就可以了.

类型	对齐方式（变量存放的起始地址相对于结构的起始地址的偏移量）
char	偏移量必须为sizeof(char)即1的倍数
int	偏移量必须为sizeof(int)即4的倍数
float	偏移量必须为sizeof(float)即4的倍数
double	偏移量必须为sizeof(double)即8的倍数
short	偏移量必须为sizeof(short)即2的倍数

```
struct MyStruct
{
double dda1;//偏移量为0，占8字节。
char dda;//偏移量为8，占1字节。
int type//偏移量为9，补齐3个字节，占4字节
}; //一共占8+1+3+4=16

struct MyStruct
```

```
{
char dda;    //偏移量为0，满足对齐方式，dda占用1个字节；
double dda1;//偏移量为1，补7个字节，占8个字节。
int type;    //偏移量为16，占4个字节。
};           //总的为1+7+8+4=20，不是结构中占用最大空间的类型所占用的字节数
sizeof(double)=8的倍数，所以需要填充4个字节，以满足结构的大小为8的倍数。一共24。

#pragma pack(push)   //保存对齐状态
#pragma pack(4)     //设定为4字节对齐
struct test
{
char m1;//偏移量为0，满足对齐方式，占用1个字节；
double m4;//偏移量为1，补3个字节，占8个字节。
int m3;//偏移量为12，占4个字节。
};
#pragma pack(pop)    //恢复对齐状态
//一共占1+3+8+4=16

#pragma pack(push)   //保存对齐状态
#pragma pack(8)     //设定为8字节对齐
struct test
{
char m1;//偏移量为0，满足对齐方式，占用1个字节；
double m4;//偏移量为1，补7个字节，占8个字节。
int m3;//偏移量为16，占4个字节。
};
#pragma pack(pop)    //恢复对齐状态
//一共占1+7+8+4=20,补4个，一共24.
//对于结构来说，它的默认对齐方式就是它的所有成员使用的对齐参数中最大的一个，如果设定为16字节对齐，实际上也会也8，因为最大的double占8字节。

#pragma pack(8)
struct S1{
char a;//偏移量为0，满足对齐方式，占用1个字节；
long b;////偏移量为1，补3个字节，占4个字节。
};//一共8个字节。
struct S2 {
char c;//偏移量为0，满足对齐方式，占用1个字节；
struct S1 d;//偏移量为1，补7个字节，占8个字节。
long long e;//偏移量为16，占8个字节。
};
#pragma pack()
//sizeof(S2)结果为24。
```

41.消息队列注意事项

- 生产者过快，处理消息速度过慢；
- 生产者过慢，处理消息速度过快；
- 大量的消息写入磁盘，IO频繁，可能会IO阻塞;
- 消息的顺序性可能出错；（单线程来保证消息顺序，一个消费者一个消息队列；或者给消息编号，比如同一类型的数据对同样的值取hash，然后就会是同一类，那么就会分发给同一个消费者）

42.共享内存读写

共享内存的消息复制只有两次。一是，从输入文件到共享内存；二是，从共享内存到输出文件。

而管道需要四次，一是复制文件到输出缓冲区，二是把输出缓冲区文件复制到管道，三是从管道复制到输入缓冲区，四是从输入缓冲区复制。

43.union和struct

1.struct 可以存储多个成员信息，而Union每个成员会用同一个存储空间。在任何同一时刻，Union只存放了一个被先选中的成员，而结构体的所有成员都存在。

2.对于Union的不同成员赋值，将会对其他成员重写，**原来成员的值就不存在了**，而对于struct 的不同成员赋值 是互不影响的。

44.struct的字节对齐问题，两个int，一个char ，sizeof（）是多大

4+4+1+3=12

45.c语言的函数加上static有什么用

1. 静态函数会被自动分配在一个一直使用的存储区，直到退出应用程序实例，避免了调用函数时压栈出栈，速度快很多。
2. 其他文件中可以定义相同名字的函数，不会发生冲突。
3. 函数的作用域仅局限于本文件，不能被工程内其他文件所用。

46.short类型占几个字节，int 类型占几个字节，short和int相加时会有隐式类型转换，对于补码来说，这个转换是在short的高字节位填充什么？把一个负数转换为补码形式最简单的方法应该是怎么做？

2，4；

把short型数据赋值给int或者unsigned int型数据时，如果short型数据是负的，则扩展的比特位全是1,；如果short型数据是正的，则扩展的比特位全是0.

将它的正数取反，然后末尾加1。

47.带参数的宏定义怎么写？

```
#define func(a,b) (a*b)
//参数绝对不能带括号
```

48.strlen这个函数如果要自己写应该怎么实现？

```
int myStrlen(const char *str)  /* 使用了一个int型变量 */
{
     if(str==NULL) return 0;
     int len = 0;
     for(; *str++ != '\0'; )
     {
         len++;
     }
    return len;
}
```

49.进程调度用来解决什么问题

主要因为很多任务在任务队列中，而处理器有限，需要从中选择任务进入内存运行。引入的目的是决定就绪队列中的哪个进程或内核级线程能获得处理器，并将处理器出让给它进行工作。

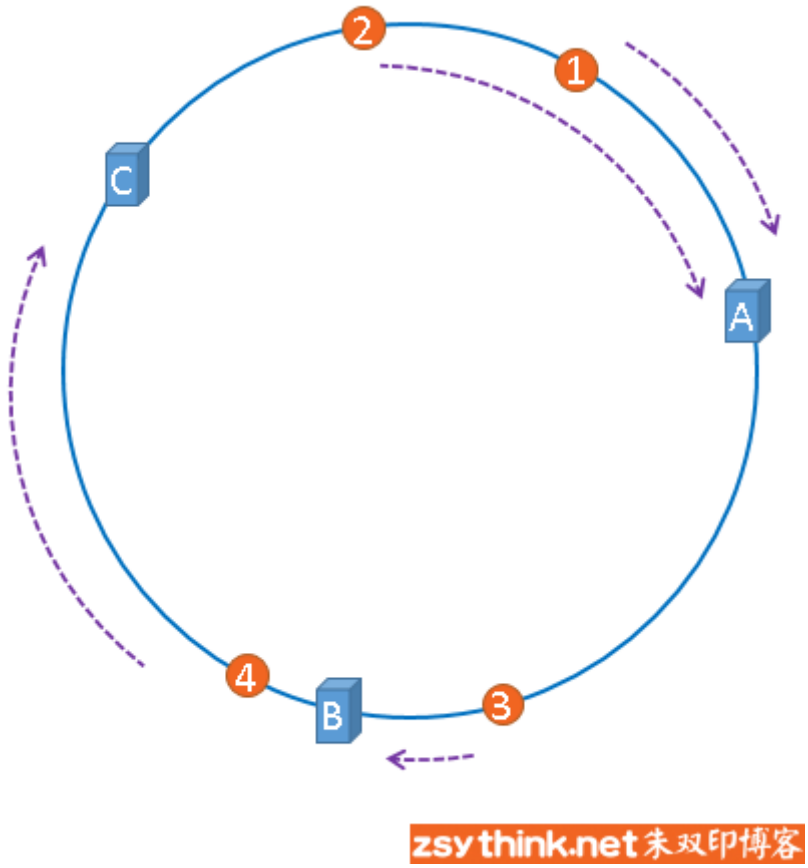
50.多线程、锁

```
//多线程
_beginthread(thread_func,0,&argv);
//多线程2
thread t(func, argv);
t.detach();//main线程和t线程各自独立，如果为join则MainThread在join的子线程没执行完的这段时间里，什么也不做，直到子线程执行完毕，才会继续向下执行。

//手动锁
mutex mutex;
mutex.lock();
......
mutex.unlock();

//自动锁
mutex mutex;
for (int i = 0; i < 100000000; i++) {
    //调用即加锁，离开大括号作用域自动解锁
    lock_guard<mutex> lock (mutex);
    val++;
}
```

51.一致性哈希

假如一个hashmap扩容以后，例如从16->32，那么之前的元素存放位置可能从2变成了18，这样的话，就要重新计算hash地址。一致性hash的出现主要是为了解决分布式缓存的问题，如果当前有三台服务器(key%3)，当服务器满足不了需求增加到了6台（key%6），那么几乎所有缓存的位置都会发生改变。

一致性哈希算法不是对容量进行取余，而是对2^32取余。把一个圆想象成一个一个的点，一共有2^32个，假设当前有三台机器，可以将机器也映射到其中，占据一个点（机器的IP地址% 2^32），此时一个元素进来可以算出它在圆中的位置，如果顺时针寻找，判断这个元素遇到的第一个机器，就是它所存放的机器。



一致性hash算法可以较大的减少位置的改变。