c++智能指针：

智能指针(smart pointer)其实不是一个指针。它就是用来帮助我们管理指针，维护其生命周期的类。

利用引用计数的策略来释放智能指针的内存， java python 的方法类似。

引用计数：对一个指针所指向的内存，目前有多少个对象在使用它

- 当引用计数为 0 时，删除对象
- 多个智能指针对象共享同一一个引用计数类
- 在进行赋值等操作时，动态维护引用计数

如何动态维护引用计数？引用计数改变发生在如下时刻：

1. 调用构造函数时：SmartPointer p(new Object());
2. 赋值构造函数时: SmartPointer p(const SmartPointer &p);
3. 赋值时: SmartPointer p1(new Object());
   SmartPointer p2 = p1;

```
class Object
{
public:
    int a;
    int b;
};
```

```
class Counter
{
    friend class SmartPointerPro;
public:
    Counter()
    {
        ptr = NULL;
        cnt = 0;
    }
    Counter(Object* p)
    {
        ptr = p;
        cnt = 1;
    }
    ~Counter()
    {
        delete ptr;
    }
private:
    Object* ptr;
    int cnt;
};
```

```
class SmartPointerPro
{
public:
    SmartPointerPro(Object* p)
    {
        ptr_counter = new Counter(p);
    }
    SmartPointerPro(const SmartPointerPro &sp)
    {
        ptr_counter = sp.ptr_counter;
        ++ptr_counter->cnt;
    }
    SmartPointerPro& operator=(const SmartPointerPro &sp)
    {
        ++sp.ptr_counter->cnt;
        --ptr_counter.cnt;

        if (ptr_counter.cnt == 0)
        {
            delete ptr_counter;
        }
        ptr_counter = sp.ptr_counter;
    }
    ~SmartPointerPro()
    {
        --ptr_counter->cnt;
        if (ptr_counter->cnt == 0)
        {
            delete ptr_counter;
        }
    }
private:
    Counter* ptr_counter;
};
```

最后一个问题：怎么样获取智能指针所包装的指针呢？

方法 1：写 GetPtr(), GetObject()等函数

```
class SmartPointerPro
{
public:
    // 前面维护智能指针的部分，空间所限，这里省略
    Object* GetPtr()
    {
        return ptr_counter->ptr;
    }
    Object& GetObject()
    {
        return *(ptr_counter->ptr)
    }
private:
    Counter* ptr_counter;
};
```

方法 2：更自然的写法，重载指针的操作符，使得 SmartPointerPro 类的对象可以像指针一样被使用。

```cpp
class SmartPointerPro
{
public:
    // 前面维护智能指针的部分，空间所限，这里省略
    Object* operator->()
    {
        return ptr_counter->ptr;
    }
    Object& operator*()
    {
        return *(ptr_counter->ptr)
    }
private:
    Counter* ptr_counter;
};

int main()
{
    SmartPointerPro p(new Object());

    p->a = 10;
    p->b = 20;

    int a_val = (*p).a;
    int b_val = (*p).b;
    return 0;
}
```

C++库中的智能指针：

std::auto_ptr，包含头文件#include<memory> 即可使用，使用时注意以下问题：

1. 不是基于引用计数的实现，一个指针在同一时刻只能被一个对象拥有
2. 尽量不要赋值，如果使用了，请不要再使用之前的对象
3. 不要当成参数传递
4. 不能放入 vector 等容器中

基于引用计数：boost::shared_ptr，包含头文件<boost/smart_ptr.hpp>即可使用

引用计数带来的问题：

如何处理循环引用问题？

```cpp
class parent;
class child;

typedef boost::shared_ptr<parent> parent_ptr;
typedef boost::shared_ptr<children> children_ptr;

class parent
{
public:
    ~parent() { std::cout << "destroying parent" << endl;}
    children_ptr children;
};

class children
{
public:
    ~children() { std::cout << "destroying children" << endl;}
    parent_ptr parent;
};

int main()
{
    parent_ptr father(new parent());
    children_ptr son(new children());

    father->children = son;
    son->parent = father;

    return 0;
}
```

使用 boost::weak_ptr 来解决循环引用问题

http://www.tuicool.com/articles/6j2yy2z

引申问题：Java 的垃圾回收机制

引用计数、分代回收、Mark & Sweep, Copy &Sweep…

http://www.cnblogs.com/dolphin0520/p/3783345.html

http://www.cnblogs.com/laoyangHJ/articles/java_gc.html

http://jefferent.iteye.com/blog/1123677

http://blog.csdn.net/initphp/article/details/30487407

快速排序：

```
void qsort(int begin, int end)
{
    if (begin >= end)return;

    int left = begin;
    int right = end;
    int key = a[left];
    while (left < right)
    {
        while ((left < right) && (a[right] <= key))right--;
        a[left] = a[right];
        while ((left < right) && (a[left] >= key))left++;
```

```
            a[right] = a[left];
        }
        a[left] = key;
        qsort(begin, left-1);
        qsort(left+1, end);
}
```

动态规划解决最长公共子串问题

```
int longestCommonSubstring(string &A, string &B) {
        // write your code here
        int m = A.size();
        int n = B.size();
        int max = 0;
        vector<vector<int>> f(m+1,vector<int>(n+1,0));
        for (int i = 1; i <= m; i++ ){
            for (int j = 1; j <= n; j++){
                if (A[i-1] == B[j-1]){
                    f[i][j] = f[i-1][j-1] + 1;
                    if (max < f[i][j]){
                        max = f[i][j];
                    }
                }else{
                    f[i][j] = 0;
                }
            }
        }
        return max;
}
```

动态规划法解决最长公共子序列问题：

```
int longestCommonSubsequence(string A, string B) {
        // write your code here
        int m = A.size();
        int n = B.size();
        vector<vector<int>> f(m+1,vector<int>(n+1,0));
        int max = 0;
        for (int i = 1; i <= m; i++){
            for (int j = 1; j <= n; j++){
                if (A[i-1] == B[j-1]){
                    f[i][j] = f[i-1][j-1] + 1;
                }else{
                    f[i][j] = (f[i-1][j] > f[i][j-1]) ? f[i-1][j] : f[i][j-1];

                }
                if (f[i][j] > max){
```

```
                    max = f[i][j];
                }
            }
        }
        return max;
}
```

链表基本操作：

有序链表插入一个节点

```
ListNode *insertionSortList(ListNode *head,ListNode *cur) {
        ListNode *dummy = new ListNode(INT_MIN);
        dummy->next = head;
        ListNode *pre = dummy;
        ListNode *beh = dummy->next;
        while(beh != NULL && beh->val < cur->val ){
            pre = beh;
            beh = beh->next;
        }
        if (beh != NULL){
            pre->next = cur;
            cur->next = beh;
        }else{
            pre->next = cur;
            cur->next = NULL;
        }
        return dummy->next;
}
```

链表反转

```
ListNode *reverse(ListNode *head) {
            ListNode *prev = NULL;
            while (head != NULL) {
                ListNode *temp = head->next;
                head->next = prev;
                prev = head;
                head = temp;
            }
            return prev;
        }
```

寻找链表的中间位置的前一个位置

```
ListNode *findmiddle(ListNode *head){
            if (head == NULL){
                return NULL;
            }
            ListNode *slow = head;
            ListNode *fast = head->next;
```

```
        if(fast != NULL && fast->next != NULL && fast->next->next == NULL){
            return slow;//三个节点的情况下会有 bug
        }
        while(fast != NULL && fast->next != NULL){
            slow = slow->next;
            fast = fast->next->next;
        }
        return slow;
    }
链表合并
ListNode *merge(ListNode *left, ListNode *right){
        ListNode *dummy = new ListNode(-1);
        ListNode *head = dummy;
        while(left != NULL && right != NULL){
            if(left->val <= right->val){
                head->next = left;
                left = left->next;
            }else{
                head->next = right;
                right = right->next;
            }
             head = head->next;
        }
        if(left != NULL){
            head->next = left;
        }else{
            head->next = right;
        }
        return dummy->next;
    }
    链表归并排序
    ListNode *sortList(ListNode *head) {
            if(head == NULL || head->next == NULL){
                return head;
            }
            ListNode *middle = findmiddle(head);
            ListNode *right = sortList(middle->next);
            middle->next = NULL;
            ListNode *left = sortList(head);
            return merge(left,right);
    }
#include <numeric>          // std::iota 用来初始化数组或 vector
int main () {
  int numbers[10];
```

```
  std::iota (numbers,numbers+10,100);
  std::cout << "numbers:";
  for (int& i:numbers) std::cout << ' ' << i;
  std::cout << '\n';
numbers: 100 101 102 103 104 105 106 107 108 109
vector<int> values(n);
iota(values.begin(), values.end(), 1);
string path;
path.substr(beg,end-beg);//提取 path 中从 beg 开始到 end（不包括）结束的字串。
用递归的方式实现二叉树的广度优先遍历
void traverse(TreeNode *root, size_t level, vector<vector<int>> &result) {
        if (!root) return;
        if (level > result.size())
        result.push_back(vector<int>());     //vector 预先加入空的元素。
        result[level-1].push_back(root->val);
        traverse(root->left, level+1, result);
        traverse(root->right, level+1, result);
}
非递归方式实现二叉树的广度优先遍历
vector<vector<int> > zigzagLevelOrder(TreeNode *root) {
          vector<vector<int>> result;
          vector<int> temp;
          queue<TreeNode*> qu;
          int size;
          if (root == NULL)
              return result;
          qu.push(root);
          while(!qu.empty()){
              size = qu.size();
              for(int i = 0; i < size; i++){
                  temp.push_back(qu.front()->val);
                  if(qu.front()->left != NULL)
                      qu.push(qu.front()->left);
                  if(qu.front()->right != NULL)
                      qu.push(qu.front()->right);
                  qu.pop();
              }
              result.push_back(temp);
              temp.clear();
          }
          return result;
    }
非递归前序遍历二叉树：
/**
```

```
 * Definition of TreeNode:
 * class TreeNode {
 * public:
 *         int val;
 *         TreeNode *left, *right;
 *         TreeNode(int val) {
 *                 this->val = val;
 *                 this->left = this->right = NULL;
 *         }
 * }
 */

class Solution {
public:
    /**
     * @param root: The root of binary tree.
     * @return: Preorder in vector which contains node values.
     */
    vector<int> preorderTraversal(TreeNode *root) {
        // write your code here
        vector<int > result;
        stack<TreeNode*> astack;
        TreeNode *pointer = root;
        while(!astack.empty() || pointer != NULL){
            if(pointer != NULL){
                result.push_back(pointer->val);
                astack.push(pointer);
                pointer = pointer->left;
            }else{
                pointer = astack.top();
                astack.pop();
                pointer = pointer->right;
            }
        }
        return result;
    }
};
```

非递归中序遍历二叉树：

```
/**
 * Definition of TreeNode:
 * class TreeNode {
 * public:
 *         int val;
 *         TreeNode *left, *right;
```

```
 *          TreeNode(int val) {
 *              this->val = val;
 *              this->left = this->right = NULL;
 *          }
 * }
 */
class Solution {
    /**
     * @param root: The root of binary tree.
     * @return: Inorder in vector which contains node values.
     */
public:
    vector<int> inorderTraversal(TreeNode *root) {
        // write your code here
        vector<int> result;
        stack<TreeNode*> astack;
        TreeNode *pointer = root;
        while(!astack.empty() || pointer != NULL){
            if(pointer != NULL){
                astack.push(pointer);
                pointer = pointer->left;
            }else{
                pointer = astack.top();
                astack.pop();
                result.push_back(pointer->val);
                pointer = pointer->right;
            }
        }
        return result;
    }
};
```

更多笔试题

http://www.nowcoder.com/contestRoom

谷歌笔试题

http://www.nowcoder.com/companyCenterTerminal_144

微软笔试题

http://www.nowcoder.com/companyCenterTerminal_146

阿里笔试题

http://www.nowcoder.com/companyCenterTerminal_134

腾讯笔试题

**http://www.nowcoder.com/companyCenterTerminal_138**

百度笔试题

http://www.nowcoder.com/companyCenterTerminal_139

二叉树节点的读入

(11,LL) (7,LLL) (8,R) (5,) (4,L) (13,RL)

```
for(;;) {
    if(scanf("%s", s) != 1) return 0;
    if(!strcmp(s, "()")) break;
    int v;
    sscanf(&s[1], "%d", &v); // s 是(11,LL)  则&s[1]  是”11,LL”
    addnode(v, strchr(s, ',')+1);// strchr(s, ',')+1 对应的字符串就是”LL”
  }
```

字符串反转  用于大整数求和。

```
string a;
reverse(a.begin(), a.end());
```

单链表归并排序

```
ListNode *sortList(ListNode *head) {
    if (head == NULL || head->next == NULL)return head;
    ListNode *fast = head, *slow = head;
    while (fast->next != NULL && fast->next->next != NULL) {
        fast = fast->next->next;
        slow = slow->next;
    }
    fast = slow;
    slow = slow->next;
    fast->next = NULL;
    ListNode *l1 = sortList(head);
    ListNode *l2 = sortList(slow);
    return mergeTwoLists(l1, l2);
}
ListNode *mergeTwoLists(ListNode *l1, ListNode *l2) {
    ListNode dummy(-1);
    for (ListNode* p = &dummy; l1 != nullptr || l2 != nullptr; p = p->next) {
        int val1 = l1 == nullptr ? INT_MAX : l1->val;
        int val2 = l2 == nullptr ? INT_MAX : l2->val;
        if (val1 <= val2) {
            p->next = l1;
            l1 = l1->next;
        } else {
            p->next = l2;
            l2 = l2->next;
        }
    }
    return dummy.next;
}
```

折半查找的迭代写法：

```
int bsearch(int *a,int x,int y,int t){
    int m;
```

```
        while(x<y){
            m = x+(y-x)/2;
            if(a[m] == t)
                return m;
            else if(a[m] > t)
                y=m;
            else
                x=m+1;
        }
        return -1;
}
// string::find 用法
#include <iostream>            // std::cout
#include <string>              // std::string
int main ()
{
  std::string str ("There are two needles in this haystack with needles.");
  std::string str2 ("needle");

  // different member versions of find in the same order as above:
  std::size_t found = str.find(str2);
  if (found!=std::string::npos)
    std::cout << "first 'needle' found at: " << found << '\n';

  found=str.find("needles are small",found+1,6);
  if (found!=std::string::npos)
    std::cout << "second 'needle' found at: " << found << '\n';

  found=str.find("haystack");
  if (found!=std::string::npos)
    std::cout << "'haystack' also found at: " << found << '\n';

  found=str.find('.');
  if (found!=std::string::npos)
    std::cout << "Period found at: " << found << '\n';

  // let's replace the first needle:
  str.replace(str.find(str2),str2.length(),"preposition");
  std::cout << str << '\n';

  return 0;
}
```

http://xem.github.io/miniCodeEditor/?utm_campaign=Manong_Weekly_Issue_13&utm_medium=

EDM&utm_source=Manong_Weekly#custom 在线文本编辑器的代码。

注意：单精度浮点型（float）和双精度浮点型（double）都以%f 的格式输出。

在%与格式字符之间还可以加上一些说明符以对输出格式作进一步的限定。

-：输出时左对齐，默认是右对齐，如 printf（"%-10d",i）。

dd：指定输出的参数所占的最小宽度，如果数据的长度小于最小宽度则以空格来填补。例如，printf（"%5d",i），若 i 的值的长度大于等于 5 位，则原样输出；若小于 5 位则左边以空格补齐；若是%-5d，则右边以空格补齐。

四皇后问题回溯法求解程序：

```
#include<iostream>
using namespace std;
int C[4];
int tot=0;
void search(int cur)
{
   int i,j;
   if(cur == 4)
   {
         tot++;                          //解的数量
         for(int k=0;k<4;k++)        //打印所有的解
               cout<<C[k]<<' ';
         cout<<endl;
   }
   else for(i=0;i<4;i++)
   {
        int ok=1;
        C[cur]=i;
        for(j=0;j<cur;j++)
        {
          if(C[cur]==C[j] || cur-C[cur]==j-C[j] ||    cur+C[cur]==j+C[j]     )
          {
             ok=0;
             break;
          }
        }
        if(ok)
              search(cur+1);
   }
}
int main()
{
      search(0);
      cout<<tot<<endl;
      system("pause");
      return 0;
```

}

C++ 的一个常见面试题是让你实现一个 String 类，限于时间，不可能要求具备 std::string 的功能，但至少要求能正确管理资源。具体来说：

能像 int 类型那样定义变量，并且支持赋值、复制。

能用作函数的参数类型及返回类型。

能用作标准库容器的元素类型，即 vector/list/deque 的 value_type。（用作 std::map 的 key_type 是更进一步的要求，本文从略）。

换言之，你的 String 能让以下代码编译运行通过，并且没有内存方面的错误。

```
void foo(String x)
{
}

void bar(const String& x)
{
}

String baz()
{
   String ret("world");
   return ret;
}

int main()
{
   String s0;
   String s1("hello");
   String s2(s0);
   String s3 = s1;
   s2 = s1;

   foo(s1);
   bar(s1);
   foo("temporary");
   bar("temporary");
   String s4 = baz();

   std::vector<String> svec;
   svec.push_back(s0);
   svec.push_back(s1);
   svec.push_back(baz());
   svec.push_back("good job");
}
```

本文给出我认为适合面试的答案，强调正确性及易实现（白板上写也不会错），不强调效率。某种意义上可以说是以时间（运行快慢）换空间（代码简洁）。

首先选择数据成员，最简单的 String 只有一个 char* 成员变量。好处是容易实现，坏处是某些操作的复杂度较高（例如 size() 会是线性时间）。为了面试时写代码不出错，本文设计的 String 只有一个 char* data_成员。而且规定 invariant 如下：一个 valid 的 string 对象的 data_ 保证不为 NULL，data_ 以 " 结尾，以方便配合 C 语言的 str*() 系列函数。
其次决定支持哪些操作，构造、析构、拷贝构造、赋值这几样是肯定要有的（以前合称 big three，现在叫 copy control）。如果钻得深一点，C++11 的移动构造和移动赋值也可以有。为了突出重点，本文就不考虑 operator[] 之类的重载了。

这样代码基本上就定型了：

```
#include <utility>
#include <string.h>
class String
{
 public:
  String() : data_(new char[1])
  {
    *data_ = '';
  }

  String(const char* str) : data_(new char[strlen(str) + 1])
  {
    strcpy(data_, str);
  }

  String(const String& rhs) : data_(new char[rhs.size() + 1])
  {
    strcpy(data_, rhs.c_str());
  }
  /* Delegate constructor in C++11
  String(const String& rhs): String(rhs.data_)
  {
  }
  */

  ~String()
  {
    delete[] data_;
  }

  /* Traditional:
  String& operator=(const String& rhs)
  {
    String tmp(rhs);
    swap(tmp);
```

```
    return *this;
  }
  */
  String& operator=(String rhs) // yes, pass-by-value
  {
    swap(rhs);
    return *this;
  }

  // C++ 11
  String(String&& rhs)
    : data_(rhs.data_)
  {
    rhs.data_ = nullptr;
  }

  String& operator=(String&& rhs)
  {
    swap(rhs);
    return *this;
  }

  // Accessors

  size_t size() const
  {
    return strlen(data_);
  }

  const char* c_str() const
  {
    return data_;
  }

  void swap(String& rhs)
  {
    std::swap(data_, rhs.data_);
  }

 private:
  char* data_;
};
```

注意代码的几个要点：

只在构造函数里调用 new char[]，只在析构函数里调用 delete[]。
赋值操作符采用了《C++编程规范》推荐的现代写法。
每个函数都只有一两行代码，没有条件判断。
析构函数不必检查 data_ 是否为 NULL。
构造函数 String(const char* str) 没有检查 str 的合法性，这是一个永无止境的争论话题。这里在初始化列表里就用到了 str，因此在函数体内用 assert() 是无意义的。
这恐怕是最简洁的 String 实现了。

练习 1：增加 operator==、operator<、operator[] 等操作符重载。

练习 2：实现一个带 int size_; 成员的版本，以空间换时间。

练习 3：受益于右值引用及移动语意，在 C++11 中对 String 实施直接插入排序的性能比 C++98/03 要高，试编程验证之。（g++的标准库也用到了此技术。）
前言

上个周末在和我的同学爬香山闲聊时，同学说到 STL 中的 string 类曾经让他备受折磨，几年前他开发一个系统前对 string 类还比较清楚，然后随着程序的复杂度的加深，到了后期，他几乎对 string 类失去了信心和信任，他觉得他对 string 类一头雾水。老实说，我几年前也有同样的痛苦（就是当我写下《标准 C＋＋类 string 的 Copy-On-Write 技术》之前的一段时间）。那时，我不得不研究那根本不是给人看的 SGI 出品的 string 类的源码，代码的可读性几乎为零，而且随着了解越深入，就越觉得 C++的世界中到处都是陷阱和缺陷。越来越觉得有时候那些类并不像自己所想象的那样工作。

为什么会发生这样的情况呢？string 类只是一个“简单”的类，如果是一些比较复杂的类呢？而这几年来，C++阵营声讨标准模板库中的标准 string 类愈演愈烈。C++阵营对这个“小子”的争讨就没有停止过。相信在下一个 C++的标准出台时，string 类会有一个大的变化。

了解 string 类

在我们研究 string 类犯了什么毛病之前，还让我先说一下如何了解一个 C++的类。我们要了解一个 C++的类，一般来说，要从三个方面入手。

一、 意图（Intention）。知其然还要知所以然，string 类的意图是什么？只有了解了意图，才知道它的思路。这是了解一个事物最重要最根本的部分。不然，你会发现它的行为并不会像你所期望的那样。string 类的意义有两个，第一个是为了处理 char 类型的数组，并封装了标准 C 中的一些字符串处理的函数。而当 string 类进入了 C++标准后，它的第二个意义就是一个容器。这两件事并不矛盾，我们要需理解 string 的机制，需要从这两个方面考虑。

二、 规格（Specification）。我们注意到 string 类有太多的接口函数。这是目前 C++阵营中声讨其最重的话题。一个小小的 string 类居然有 106 个成员接口函数。居然，C++标准委员会会容忍这种“ugly”的事情的发生？目前的认为导致“C++标准委员会脑子进水”

的主流原因有两点，一个是为了提高效率，另一个是为了常用的操作。

1）让我们先来看效率，看看 string 类中的“==”操作符重载接口：
bool operator==(const string& lhs, const string& rhs);

bool operator==(const string& lhs, const char* rhs);
bool operator==(const char* lhs, const string& rhs);

头一个很标准，而后两个似乎就显得没有必要了。如果我们调用：（Str == “string”）如果没有后面两个接口，string 的构造函数会把 char*的 ”string”转成 string 对象，然后再调用第一个接口，也就是 operator==(str, string(“string”))。如此“多余”的设计只能说是为了追求效率，为了省去调用构造/析构函数和分配/释放内存的时间（这会节省很多的时间）。在后面两个接口中，直接使用了 C 的 strcmp 函数。如此看来，这点设计还是很有必要的。string 类中有很多为了追求效率的算法和设计，比如：Copy-on-Write（参看我的《标准 C＋＋类 string 的 Copy-On-Write 技术》）等。这些东西让我们的 string 变得很有效率，但也带来了陷阱。如果不知道这些东西，那么当你使用它的时候发生不可意料的问题，就会让你感到迷茫和不知所措。

2）另一个让 string 类拥有这么庞大的接口的原因是常用的操作。比如 string 类的 substr()，这是一个截取子字符串的函数。其实这个函数并不需要，因为 string 有一个构造函数可以从别的 string 类中指定其起始和长度构造自己，从而实现这一功能。还有就是 copy()函数，这也是一个没有必要的函数，copy 这个函数把 string 类中的内容拷贝到一个内存 buffer 中，这个方法实践证明很少有人使用。可能，1）为了安全起见，需要有这样一个成员把内容复制出去；2）设计者觉得 copy 要比 strcpy 或是 memcpy 好写也漂亮很多吧。copy()比起 substr()更没有必要存在。

三、 实现（Implementation）。C++标准并没有过多的干预实现。不同的产商会有不同的实现。不同的产商会考虑标准中的一件事情是否符合市场的需要，并要考虑自己的编译器是否有能够编译标准的功能。于是，他们总是会轻微或是颠覆地修改着标准。C++在编译器的差异是令人痛苦和绝望的，如果不了解具体的实现，在你使用 C++的时候，你也会发现它并不像你所想象的那样工作。

只有从上述三个方面入手，你才能真正了解一个 C++类，而你也才能用好 C++。C++高手们都是从这样的三个方面剖析着 C++现实中的各种类，并以此来验证 C++类的设计。
String 类犯了什么错？
string 类其实挺好的。它的设计很有技术含量。它有高的效率、运行速度快、容易使用。它有很充足的接口可以满足各式各样的法，使用起来也很灵活。
然而，这个 string 类似乎有点没有与时俱进，它现在的设计还保持着 10 年以前的样子，10 年来，整个技术环境都出现很多变革，人们也在使用 C++的过程中得到了许许多的经验。比如：现在的几乎所有的程序都运行在一个多进/线程的环境中，而 10 年前主流还只是单进/线程的应用，这是一个巨大的变化。近几年来，C++阵营也在实践中取得了很多的经验，特别是模板技术，而我们的 STL 显然没能跟上脚步。

首当其冲的是 string 类，目前 C++阵营对 string 类的声讨主要集中在下面几个方面。对于下面的这些问题，C++阵营还是争论不休。不过，作为一个好的程序员，我们应该在我

们的设计和编程中注意一下这些方面。
1）目前的标 string 类有 106 个接口函数（包括构造和析构函数），如果考虑上默认参数，那么就一共有 134 不同的接口。其中有 5 个函数模板还会产生无穷多个各种各样的函数。还有各种各样的性能上的优化。在这么从多的成员函数中，很多都是冗余不必要的。最糟糕的是，众多程序员们并不了解它们，导到要么浪费了它的优势，要么踩中了其中的陷阱。
2）很多人认为 string 类提供的功能中，该有的没有，已有的又很冗余。string 类在同一个功能上实现了多次，而有一些功能却没有实现。如：大小写不区分的比较，宽行字符（w_char）的支持，和字符 char 型数据的接口等等。
3）作为一个 STL 的容器，string 类和的设计和其它容器基本一样。这些 STL 都不鼓励被继承。因为 STL 容器的设计者们的几乎都遗忘了虚函数的使用，这样阻止了多态性，也许，这也是一个为了考虑效率和性能的设计。对于 STL 容易这样的设计，大多数人表示接受。但对于 string 类，很多人认为，string 类是一个特殊的类，考虑到它被使用的频率，string 类应该得到特殊的照顾。
4）还有很多人认为标准的 strng 类强行进行动态内存分配（malloc），那怕是一个很短的字符串。这会导致内存碎片问题（我们知道内存碎片问题会让 malloc 很长时间才能返回，由于降低了整个程序的性能。而关于内存碎片问题，这是一个很严重的问题，目前除了重启应用程序，还没有什么好的方法）。他们认为，string 类的设计加剧了内存碎片问题。他们希望 string 类能够拥有自己的栈上内存来存放一些短字符串，而不需要总是去堆上分配内存。（因为用于 string 类的字符串长度几乎都不会很长）
5）很多 string 类的实现，都采用了 Copy-On-Write（COW）技术。虽然这是一个很有效率的技术。但是因为内存的共享，导致了程序在“多线程”环境中及容易发生错误，如果分别在两个线程中的 string 实例共享着同一块内存，很有可能发生潜在的内存问题（参看我的《标准 C＋＋类 string 的 Copy-On-Write 技术》最后一节示例）。目前这一技术很有可能从下一版本的标准中移去。（而一些新版本的 STL 都不支持 COW 了，如 Microsoft VC8.0 下的 STL）

6）标准的 string 类不支持 policy-base 技术的错误处理。string 遇到错误时，只是简单地抛出异常。虽然这是一个标准，但有一些情况下不会使用异常（GCC －fno-exception）。另外，不可能要求所有的程序都要在使用 string 操作的时候 try catch，一个比较好的方法是 string 类封装有自己的 error-handling 函数，并且可以让用户来定义需要使用哪一种错误处理机制。由于 string 类的的种种不如人意，特别是 106 个接口函数让许多人难以接受，有很多人都在写适合自己的 string 类。但总体来说，标准的 string 类是一个好坏难辨的类。无论你是否是一个高级咨深的程序员，你都会用到它。它和整个 C++一样，都是一把双刃剑，在大多数情况下，它还是值得我们信赖。
目前，对 string 类的争论有很多很多。C++标准委员会也说：“The C++ Standard is the best we could make it. If we could have agreed on how to make it better, then we would have made it better.”在 C++这么一个混乱的领地，虽然 STL 并不完美，但对比起来说，他还是显得那么地漂亮和精致辞。
后记
又到了关于 C++文章结束的时候，我还是要老调重弹。C++这个世界是很复杂很危险的。单单本文章中的一个小小的 string 类就能引起这么多的讨论，何况是别的类？
让我们大胆地怀疑一下， C++是否真是一种高级的语言？目前的 C++需要使用他的人有相当的专业技术水平，而寄希望于人的高水平看来是和技术的发展背道而驰的。好的一门开发语言是让人可以容易方便地把更多的精力集中在业务逻辑性上。而 C++这门语言却需要技术人员比以住的 C 有着更为高深和专业的技术知识和能力。

是我们对 string 的“无知”搞乱了我们的程序，还是 string 这个类的设计把我们的程序搞乱了？是 C++自己把软件开发搞得一团混乱？还是使用 C++的人把其搞得一团混乱？好像兼而有之，目前我们无法定论，但我们知道，无论是 C++标准委员会，还是开发人员，大家的 C++未来之路都还有很长。

iostream 是 cstdio 基础上的封装，所以 cout 的线程安全依赖于 printf 之类。而符合 posix 标准的 printf 是可重入的，所以一个<<是线程安全的，多次调用就没有保证了。std::cout<<”hello world ”<<std::endl; 不如 std::cout<<”hello world \n”;安全。
g++编译加调试。编译 g++ -c hello.cc 产生目标文件 hello.o 连接 g++ hello.o 产生可执行文件 gcc 编译生成 gdb 调试 使用 gdb 调试前用-g 选项 gcc –g summary.c 整数运算时结果只保留整数，10/20=0。左移<<右边补 0，右移>>左边如果是无符号数和非负数则补 0，如果为负数则补 1。左移相当于*2 的 n 次方，一般情况右移相当于/2 的 n 次方。-1 和 0 右移不变。
a=20; 3<a<5 为真，因为 3<20 为真，真<5 为真。
不用中间变量交换两个数的值：a=a^b;b=a^b;a=a^b;或 a=a+b;b=a-b;a=a-b;
异或和或只有同为 1 时结果不一样，c=a^b；d=a|b；e=c^d；可以求出 a,b 都为 1 的位。
在一个式子中反复修改一个变量的值，结果是不确定的。
优先级前三名：括号，成员，单目。后三名：三目，赋值，逗号。
enum color{black,white,blue,green};默认从 0-3，若指定 blue=100,则 green=101。
Switch 语句中 case 后的量是字面量，会自动转为整型，可以是 int,char,bool,enum;
Switch 不要定义新变量。
#include<ctime> time(NULL);取得系统时间，从 1970 年算起。

```
#include<ctime>
#include<cstdlib>
int main()
{
    srand(time(NULL));
    rand()%100;
    可以产生 0-100 内的随机数。

}
```

闰年：能被 4 整除，且不能被 100 整除。能被 400 整除。

```
if((y%4==0&&y%100!=0)|| (y%400==0))
```

g++编译器的 bug，每个函数必须有返回值，没有则混乱。
只要返回类型不是 void，一定要用 return 返回。
堆栈后进先出的特点便于实现函数递归，f(3)-f(2)-f(1)-f(0)-f(0)- f(1)- f(2)- f(3) 堆栈可以保存现场，恢复现场。

```
int main()
{
    Cout<<f1()<<endl;
}
void f1()
{
    Cout<<””<<endl;
}
```

有错，因为 f1()无返回值。

void order (int n,int m=1)指定 m 默认值，有默认值的参数靠右摆放,声明带默认值，定义不带。
函数调用过程：1 保存现场 2 传递参数 3 执行函数 4 带回结果 5 恢复现场。
递归解决问题的方法：逐级化简，化到最简后直接解决。
str[]=”kobe” sizeof(str)=5;strlen(str)=4。sizeof(str++)*str 不会变成 o,strlen(str++)会变。
C++中自己写的头文件要用“ ”来引用。如#include“func.h”
同一个项目中有头文件 func.h，实现文件 func.cc（要包含#include<iostream>和 using namespace std; #include“func.h”），主文件 main.cc（在开头要包含#include<iostream>和 using namespace std; #include“func.h”），可以一起编译连接：g++ *.cc。
在头文件中写：

```
#ifndef _FUNC_H_
#define _FUNC_H_//头文件名大写,前后写_
void func(int n);
void const_func(int n);
#endif//可以避免头文件被重复分析，提高编译执行效率。
```

程序最后加一个空行可以防止编译器的警告。
在头文件中声明全局变量 extern int g;，在其他地方定义，最好不用。
只在本文件中使用的变量可以用 static 来定义。

1）全局变量是不显式用 static 修饰的全局变量，但全局变量默认是动态的，作用域是整个工程，在一个文件内定义的全局变量，在另一个文件中，通过 extern 全局变量名的声明，就可以使用全局变量。

2）全局静态变量是显式用 static 修饰的全局变量，作用域是声明此变量所在的文件，其他的文件即使用 extern 声明也不能使用。

静态局部变量有以下特点：

该变量在全局数据区分配内存；

静态局部变量在程序执行到该对象的声明处时被首次初始化，即以后的函数调用不再进行初始化；

静态局部变量一般在声明处初始化，如果没有显式初始化，会被程序自动初始化为 0；

它始终驻留在全局数据区，直到程序运行结束。但其作用域为局部作用域，当定义它的函数或语句块结束时，其作用域随之结束；

```
int counter()
{
    static int cnt=0;//加上 static 后每次调用不会将 cnt 清零，会继续累加。
    ++cnt;
    return cnt;
}
int main()
{
    cout<<counter()<<endl;
    cout<<counter()<<endl;
    cout<<counter()<<endl;
}
```

enum Color{BLACK,WHITE,BLUE=100,GERRN}//自动从 0 依次递增，指定后按指定的值输出，GERRN 的值是 101。
sizeof()中是类型。int a[5]的大小：sizeof(a)或 sizeof(int [5])把名字去掉，只留下 int [5]编译器

会自己找到类型，输出 20。
数组名表示第一个元素的地址(只有字符型除外)。c++为了与 c 兼容，在输出字符变量地址的时候，改成输出从这个内存单元开始存放的一系列字符，到向后遇到第一个'\0'为止。
要输出字符地址可以：cout<<hex<<"0x"<<(int)&ch;初始数据不够自动补 0。不写元素个数时，编译器自己数。数组初始化可以 a[5]={},数组元素会自动初始化为 0;
数组元素越界赋值会影响到其他变量，如 int a[5];a[5]=10;
　　把数组名当数值用的时候，它仅仅是一个地址。

```
double max (double score[])
{
    cout<<"sizeof (score)="<<sizeof (score)<<endl;
    return 100.0;
}
int main()
{
    double s[3]={90,95,94};
    cout<<"sizeof(s)="<<sizeof(s)<<endl;
    cout<<"max="<<max(s)<<endl;
}
```

输出：sizeof(s)=24,max=sizeof(score)=4,100。
调用函数 max 时，把 s 的首地址传递给 max，因此 score 为地址，当把数组名当数值用时它仅仅是一个地址。因此 sizeof(score)=4。
swap(a,b)用于交换数据，如不能通过编译，则加#include<algorithm>
选择排序：

```
for(i=0;i<N;i++)
    for(j=i+1;j<N;j++)
        if(a[j]<a[i])
            swap(a[i],a[j]);
```

主目录下，/exrc ab .i #include <iostream>
ab .u　using namespace std;
ab .m int main()
ab .f #include<fstream>
ab .s #include <string>
ab .r return
ab #i #include
strlen()求字符串长度，sizeof()求数组长度。

```
struct  Student{
    char name[20];
    bool gender;
    int age;
    char addr[120];
    double score;
};//输出长度是 160，不是 153。short 占 2 个字节，内存对齐是一次用 4 个字节，不够的会对齐。
```

首先，每种类型的变量的默认对齐长度都是自己的变量长度，比如：char 占一个字节，那么对齐长度就是一个字节，int 占四个字节，对齐长度就是四个字节，double 占八个字节，对齐长度就是 8。int 的对齐长度为 4 的实际意义是，int 变量必须存储在四的倍数的地址上。

那么对于 struct{char b; int a}，其长度是 8，因为 b 虽然只占用 1 个字节，但是 a 必须从 4 的倍数开始存储，因此 b 后面的 3 个字节都废掉了。因此一共需要 8 个字节才能把 b 和 a 存下来。

那么对于 struct{int a; char b}，其长度还是 8！晕菜了！原因如下：

字节对齐的细节和编译器实现相关，但一般而言，满足三个准则：

1) 结构体变量的首地址能够被其最宽基本类型成员的大小所整除。

2) 结构体每个成员相对于结构体首地址的偏移量（offset）都是成员大小的整数倍，如有需要编译器会在成员之间加上填充字节（internal adding）；

3) 结构体的总大小为结构体最宽基本类型成员大小的整数倍，如有需要编译器会在最末一个成员之后加上填充字节（trailing padding）。

规则 1 是控制结构体变量的首地址的，与结构体变量的长度没关系。

规则 2 是控制结构体内每个变量的相对地址的，与结构体变量的长度有关系。

规则 3 是控制结构体总体长度的，与结构体变量的长度有关系。

如果结构体里面嵌套结构体就要注意了，结构体变量的起始地址只是其内部最宽的基本类型的整数倍，而非结构体自身的整数倍，外面结构体的长度，也仅仅是里面最宽的基本类型的长度倍数。

比如：

```
struct S1 {
  char c;   // 1 个字节
  int i;    // 前面空 3 个字节，占用 4 个字节
}; // 刚好 8 个字节，是 4 的倍数
struct S2 {
  char c1; // 1 个字节
  S1 s;   // 前面空 3 个字节，而不是空 7 个字节，占用 8 个字节
  char c2; // 占用 1 个字节
}; // 一共 13 个字节，要成为 4 的倍数，后面增加 3 个字节，成为 16 个字节
```

对于简单的能计算就行，因为有的时候笔试会出这个。

1 :数据成员对齐规则：结构(struct)(或联合(union))的数据成员，第一个数据成员放在 offset 为 0 的地方，以后每个数据成员存储的起始位置要从该成员大小的整数倍开始(比如 int 在 32 位机为 4 字节,则要从 4 的整数倍地址开始存储。

2 :结构体作为成员:如果一个结构里有某些结构体成员,则结构体成员要从其内部最大元素大小的整数倍地址开始存储.(struct a 里存有 struct b,b 里有 char,int ,double 等元素,那 b 应该从 8 的整数倍开始存储.)

3 :收尾工作:结构体的总大小,也就是 sizeof 的结果,.必须是其内部最大成员的整数倍.不足的要补齐.

4) 结构体内类型相同的连续元素将在连续的空间内，和数组一样。

为了节省空间，可以把几个数据压缩到少数的几个类型空间上，比如需要表示二个 3 位二进制的数，一个 2 位二进制的数，则可以用一个 8 位的字符表示之。

使用位域的主要目的是压缩存储，其大致规则为：

1) 如果相邻位域字段的类型相同，且其位宽之和小于类型的 sizeof 大小，则后面的字段将紧邻前一个字段存储，直到不能容纳为止；

2) 如果相邻位域字段的类型相同，但其位宽之和大于类型的 sizeof 大小，则后面的字段将从新的存储单元开始，其偏移量为其类型大小的整数倍；

3) 如果相邻的位域字段的类型不同，则各编译器的具体实现有差异，VC6 采取不压缩方式，

Dev-C++采取压缩方式；
4)　如果位域字段之间穿插着非位域字段，则不进行压缩；
5)　整个结构体的总大小为最宽基本类型成员大小的整数倍。
结构体中，字符数组不能直接赋值，字符串操作 #include<cstring> strcpy(字符数组名，字符串)。strlen(字符串)，strcmp (字符串 1，字符串 2) 对应字符比 ASCii 码，结果为整数，1>2 结果大于 0，1<2 结果小于 0，1=2 结果等于 0。strcat(字符数组名，字符串)将字符串追加到字符数组中。strlen()会根据编译器来决定是否算上‘\0’。
cin>>字符变量地址，表示从这个地址开始依次摆放输入的字符串，最后追加\0。
c++风格字符串:#include<string>　string name; cin>> name;name=”kobe”;使用name中的字符用name[下标]。c++风格字符串不以\0作结束标志。　(不能读空格，用getline(cin,str)可以读一整行。) 求长度：name.size()（string 是指针类型，string str;sizeof(str)结果为4）赋值：name=字符串，追加：name+=字符串,比较：str1==str2;str1>str2;str1<str2,转换为c风格字符串：name.c_str(); c风格转换为c++风格：string(char [])
c风格字符串输入一行用：cin.getline(地址，长度)。
a[5],*a=a[0];

```
char* input (char* p)
{
	cout<<”input your name:”;
	cin>>p;//字符型变量地址特殊用法。
	return p;/ /字符型变量地址特殊用法。
}
int main()
{
	char name [20];
	cout <<input(name)<<endl;
}
```

栈区给程序用，堆区给程序员用。
指针的作用：1 访问数组元素 2 为函数传递参数 3 把数组和字符串传递给函数 4 从系统获得内存空间 5 创建数据结构。
int * a[5]表示 5 个指针，可以写 a[i]=new int;，int (*a)[5](最好不用)表示指向数组的指针。p=&a（a 为数组名，&a 代表整个数组）;*p=a;(*p)[3]=a[3];
int * p=a;　p[i]与 a[i]等价。*（a+i）与* &a[i]都表示 a[i]。int *p=&n; *p 就是 n。
*p++先算 p++,后算*。
const int*p=&n;表示*p 是整型常量,不能通过 p 来改变 n 但 n 自身可改变，p 可以改变指向。
int* const p 表示 p 是常量（常指针），指向不变，内容可变。
struct date; date *p;　(*p).year 等价于 p->year
double * p=NULL;　p=new double;delete p;
cin>> n; p=new double [n];

```
for (int i=0;i<n;i++)
{
	cin>>p[i];
}
```

delete[] p;//归还数组空间。删除后 p 没有变化，最好 p=NULL。如果 p++了，删除会出现段错误。

指针都占 4 字节内存空间。
void* p=NULL;int n=100;
p=&n;cout<<*(int*)p<<endl;否则会出错。
int main(int argc,char*argv[] ) argc 是命令行中的字符串个数（包括命令名），以空格为间隔。
argv 是一组字符指针，依次指向每个一个字符串。cout<<argv[i]可以输出对应字符串。
argc,argv 可以使任何合法的名字。使用时在 a.out 后加参数。
#include <cstdlib>  atof(str1)将 str1 转化为浮点数; atoi(str1)将 str1 转化为整型数。
int &rn=n;引用只能用于变量，不能用于常量，终身制（一个引用对应一个变量，不能像指针一样修改引用。），rn 与 n 的地址相同。
能用引用解决的问题不要用指针。
const int& n 常量引用，系统会临时找一个空间存储，作用:1 接收常量初始化，2 禁止修改这个引用。
使用引用是注意：尽量使用引用传递参数，尽量应 const 来限制修改引用的参数。
全局变量和静态变量默认初始化为 0,其他变量是垃圾数据。
代码区（存放函数）
数据区（全局变量和静态变量）
常量区（字面量）
堆（程序员用）
栈
像声明函数一样定义函数指针，不同在于，要把函数名改成（*指针名）的形式。
void sort (int a[],int n);
void (*fp) (int a[],int n)=NULL;
fp=sort;//函数名就是函数地址。
sort(a,5);
fp(a,5);//不允许输出函数指针的值。
在行命令模式下输入：!man 函数名，可以显示所要加的头文件名。
memset(地址，数字，字节数)将指定地址开始的内存单元赋值指定数字。头文件为：
#include<cstring>。
int*p=new int (v);用 v 来初始化*p//类和结构也可以这样初始化。
sizeof();只关心类型，不关心内容。
没有 cout<<endl;会等待缓冲区满后再输出，可以加上<<flush 来解决。输出前加上 cout<<'\r'会重新回到一行开头输出。
在调用成员函数时，程序会自动把结构变量的地址传过来，可以通过 this 来使用这个地址。
6,$s/p/this/g 从第 6 行到最后一行，将 p 替换为 this，全部替换。
%s/this->//g 把 this->去掉。
在成员函数中，把用来调用这个成员函数的变量成为当前变量。
延时 1 秒：time_t  t=time(NULL);while(time(NULL) ==t);
ctrl +c 结束程序运行。
构造函数与类名一样，不用写返回类型，自动调用，不能主动调用。clock  c(0,3,5);创建对象时初始化。结构与类的区别：结构是公开的，类的数据是私有的。
私有类型只能在自己的成员函数中使用。创建对象是如果不需要参数的话不能写成 A  a3();系统会认为是声明函数。
创建对象时传递的参数是传递给构造函数的形参，而不是赋值给成员变量。
在构造函数的函数头和函数体之间加：表示初始化列表。A():n(6){…};6 为初始化数据，多

项用逗号隔开：A():n(6)，d(6.0){…};初始化列表只能用于构造函数。数组和结构不能初始化。
常量只能初始化，不能赋值，加 const 修饰的成员变量只能初始化。
class A{} A (int n=0,double d=0.0):n(n),d(d){};带默认值的构造函数，可以传参数，也可不传。
两个构造函数，一个无参数，一个默认参数，会出现编译错误。
~类名，析构函数。
static 会延长生命期,将局部变量的生命期延长至程序结束。析构函数没有形参，不可重载。
构造时先创建成员的，后创建自己的。
全局变量在 main 前创建。全局对象的构造函数在 main 之前调用。
构造函数使用默认参数的好处是可以满足无参调用，如：A* p=new A[3];
使用 A* p=new A[3];时必须用 delete[] p;否则会不能调用析构函数。
无参的对象声明：Book A;A 后不要加() 。
使用对象：对象.成员函数。
继承：1 单重继承（java 支持）2 多重继承 3 虚继承（2、3 两种极少使用）。
多态：1 虚函数 2 类型识别 3 纯虚函数 4 抽象类（2、3、4 了解）。
class Student : public Person{};Student 类继承 Person 类。
私有成员(private)：子类可以继承父类的私有成员，但子类中不能直接访问父类的私有成员，可以通过函数访问。
保护成员（protected）：子类可以直接访问父类中的保护成员，类外（main 中）不能直接访问保护成员。
private 成员只能被该类的成员函数和友元函数使用。
protected 成员只能被该类的成员函数和友元函数以及派生类的成员函数和友元函数使用。
公有继承访问权限如下：

| | | | |
|---|---|---|---|
| 父类 | 私有 | 保护 | 公有 |
| 父类 | 能 | 能 | 能 |
| 子类 | 不能 | 能 | 能//公有和保护都不变 |
| 类外 | 不能 | 不能 | 能 |

私有继承访问权限如下（一律私有）：

| | | | |
|---|---|---|---|
| 父类 | 私有 | 保护 | 公有 |
| 父类 | 能 | 能 | 能 |
| 子类 | 不能 | 不能 | 不能//公有和保护都变成私有成员 |
| 类外 | 不能 | 不能 | 不能 |

保护继承访问权限如下（公开变保护）：

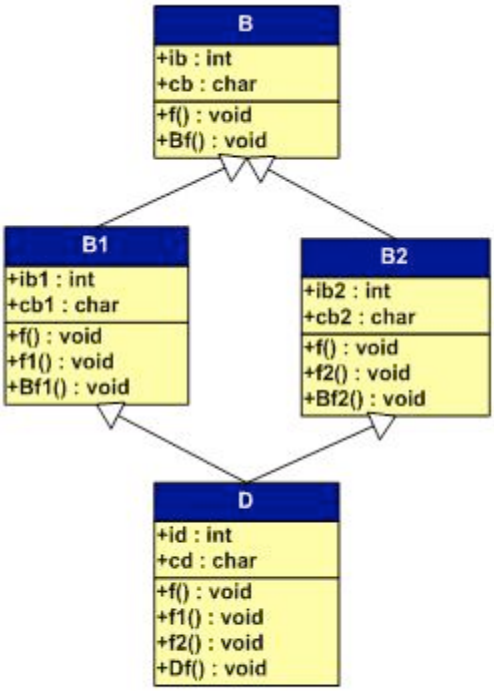
| | | | |
|---|---|---|---|
| 父类 | 私有 | 保护 | 公有 |
| 父类 | 能 | 能 | 能 |
| 子类 | 不能 | 能 | 能//在子类中，公有和保护都变成保护成员 |
| 类外 | 不能 | 不能 | 不能 |

子类中调用父类的函数可以直接调用 getname(), main 中要用对象调用。
子类定义新的函数会覆盖父类的同名函数,不存在重载，要指明调用父类的函数则用 Animal::show(true);或在子类中再写一个与父类同名的函数，在其中调用父类的函数，用法如下：void show(bool newline){//与父类的同名。

```
    Animal::show(newline);
}
```

子类构造函数会首先调用父类构造函数，再执行自己的语句。
子类析构函数在执行完自己的语句后会调用父类的析构函数。

创建子类对象的时候，默认总是调用父类的无参构造函数，在子类的构造函数的初始化列表中用父类类名来给父类传递参数：Child(int d):mc(d),Parent(50){};。
class A ; A(5)是没有名字的对象也是对象(能使用的前提是要有一个包含一个形参的构造函数)，使用完后立即释放，相当于强制类型转换,可以是（A）5，也也可以是 A(5)。
const A a1(5)是常量对象，不能调用普通成员函数，如果成员函数在函数名和函数体之间加上了 const 则表示不会修改当前对象的数据成员（void show() const {};），这种函数可以由常量对象调用，与没有 const 的同名函数构成重载。
普通成员（非常量对象）也可以调用有 const 的函数，优先调用没有 const 的函数。
多重继承中，子类会依次调用父类的构造函数（class A:public B,public C,public D）。
虚继承：
多重继承中，来自同一个虚基类的成员会合并成一份。class Fruit :virtual public Production{};
为了解决从不同途径继承来的同名的数据成员在内存中有不同的拷贝造成数据不一致问题，将共同基类设置为虚基类。这时从不同的路径继承过来的同名数据成员在内存中就只有一个拷贝，同一个函数名也只有一个映射。这样不仅就解决了二义性问题，也节省了内存，避免了数据不一致的问题。




钻石型虚拟多重继承

```
include "stdafx.h"
include <iostream>
using namespace std;

int gFlag = 0;

class Base
{
public:
```

```
Base(){cout << "Base called : " << gFlag++ << endl;}
void print(){cout << "Base print" <<endl;}
};

class Mid1 : virtual public Base
{
public:
Mid1(){cout << "Mid1 called" << endl;}
private:
};

class Mid2 : virtual public Base
{
public:
Mid2(){cout << "Mid2 called" << endl;}
};

class Child:public Mid1, public Mid2
{
public:
Child(){cout << "Child called" << endl;}
};

int main(int argc, char* argv[])
{
Child d;

//这里可以这样使用
d.print();

//也可以这样使用
d.Mid1::print();
d.Mid2::print();

system("pause");
return 0;
}
```

输出：

```
1: Base called : 0
  2: Mid1 called
  3: Mid2 called
  4: Child called
  5: Base print
  6: Base print
```

7: Base print
8: 请按任意键继续. . .

1.在多继承情况下，虚基类关键字的作用范围和继承方式关键字相同，只对紧跟其后的基类起作用。

2.声明了虚基类之后，虚基类在进一步派生过程中始终和派生类一起，维护同一个基类子对象的拷贝。

3.观察类构造函数的构造顺序，拷贝也只有一份。

如果函数声明和定义分开，形参默认值写在函数声明时，初始化列表写在构造函数定义时。

约分函数：deduce()

```
{
    if(deno<0){deno=-deno;nume=-nume;}
    if(deno==0)cout<<”ERROR”;
    int absn=(nume<0?-nume:nume);
    for(int i=(absn<deno?absn:deno);i>1;i--)
    if(nume%i==0&&deno%i==0)
    {
        nume/=i;
        deno/=I;
        break;
    }
}
```

Dai(in tai,int an,int ad):i(ai),Fract(an,ad){};子类 Dai 构造函数给父类 Fract 传递参数。

子类对象总可以看成父类对象，调用的函数是父类函数:

```
Dai f2(2,12,16);
Fract* p=NULL;
p=&f2;
```

p->show();//用父类指针指向子类对象会调用父类的函数。

父类中的函数前加上 virtual（即虚函数）（只在声明时写）表示使用多态，即 p->show();会调用子类的 show()函数(virtual void show()),子类和父类的函数的返回类型和参数表必须一样。

指针调用和引用都可以使用多态：Fract & f=f2;f.show();会调用子类的 show()。

多态：根据对象的真实类型，调用对象所属类中的相应函数。

增加虚函数后，系统会增加一个指向虚函数的指针（虚函数表），会增加存储空间（4 个字节，与虚函数数量无关）。

C++的编译器应该是保证虚函数表的指针存在于对象实例中最前面的位置（这是为了保证取到虚函数表的有最高的性能——如果有多层继承或是多重继承的情况下）。这意味着我们通过对象实例的地址得到这张虚函数表，然后就可以遍历其中函数指针，并调用相应的函数。

http://blog.csdn.net/haoel/article/details/3081328

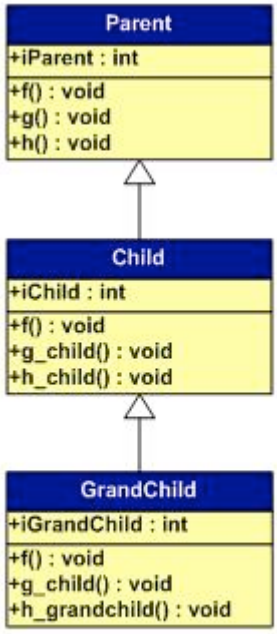
```
class Base {
    public:
        virtual void f() { cout << "Base::f" << endl; }
        virtual void g() { cout << "Base::g" << endl; }
        virtual void h() { cout << "Base::h" << endl; }
```

```
};
```
按照上面的说法，我们可以通过 Base 的实例来得到虚函数表。 下面是实际例程：
```
 typedef void(*Fun)(void);
Base b;
Fun pFun = NULL;
cout << "虚函数表地址：" << (int*)(&b) << endl;
cout << "虚函数表 — 第一个函数地址：" << (int*)*(int*)(&b) << endl;

// Invoke the first virtual function
pFun = (Fun)*((int*)*(int*)(&b));
pFun();
```
实际运行经果如下：(Windows XP+VS2003， Linux 2.6.22 + GCC 4.1.3)

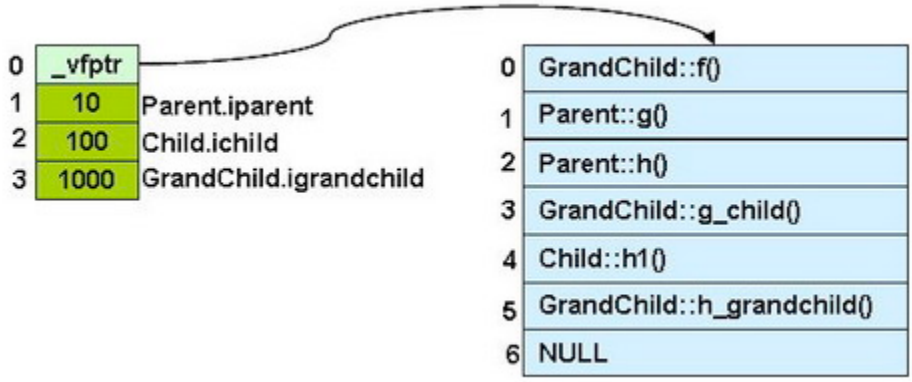
```
虚函数表地址：0012FED4
虚函数表 — 第一个函数地址：0044F148
Base::f
```
一般继承（无虚函数覆盖）对于实例：Derive d; 的虚函数表如下：
```
class Parent {
public:
	int iparent;
	Parent ():iparent (10) {}
	virtual void f() { cout << " Parent::f()" << endl; }
	virtual void g() { cout << " Parent::g()" << endl; }
	virtual void h() { cout << " Parent::h()" << endl; }

};

class Child : public Parent {
public:
	int ichild;
	Child():ichild(100) {}
	virtual void f() { cout << "Child::f()" << endl; }
	virtual void g_child() { cout << "Child::g_child()" << endl; }
	virtual void h_child() { cout << "Child::h_child()" << endl; }
};

class GrandChild : public Child{
public:
	int igrandchild;
	GrandChild():igrandchild(1000) {}
	virtual void f() { cout << "GrandChild::f()" << endl; }
	virtual void g_child() { cout << "GrandChild::g_child()" << endl; }
	virtual void h_grandchild() { cout << "GrandChild::h_grandchild()" << endl; }
};
```

我们使用以下程序作为测试程序：（下面程序中，我使用了一个 int** pVtab 来作为遍历对象内存布局的指针，这样，我就可以方便地像使用数组一样来遍历所有的成员包括其虚函数表了，在后面的程序中，我也是用这样的方法的，请不必感到奇怪，）

```
typedef void(*Fun)(void);
    GrandChild gc;
    int** pVtab = (int**)&gc;
    cout << "[0] GrandChild::_vptr->" << endl;
    for (int i=0; (Fun)pVtab[0][i]!=NULL; i++){
                    pFun = (Fun)pVtab[0][i];
                    cout << "       ["<<i<<"] ";
                    pFun();
    }
    cout << "[1] Parent.iparent = " << (int)pVtab[1] << endl;
    cout << "[2] Child.ichild = " << (int)pVtab[2] << endl;
    cout << "[3] GrandChild.igrandchild = " << (int)pVtab[3] << endl;
```




单一继承



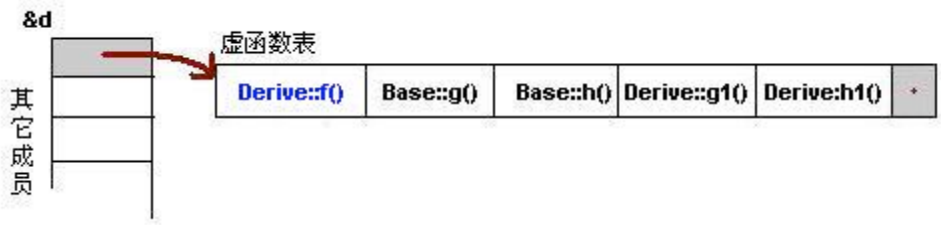
可见以下几个方面：

1）虚函数表在最前面的位置。

2）成员变量根据其继承和声明顺序依次放在后面。

3）在单一的继承中，被 overwrite 的虚函数在虚函数表中得到了更新。

一般继承（有虚函数覆盖）




我们从表中可以看到下面几点，

1）覆盖的 f()函数被放到了虚表中原来父类虚函数的位置。

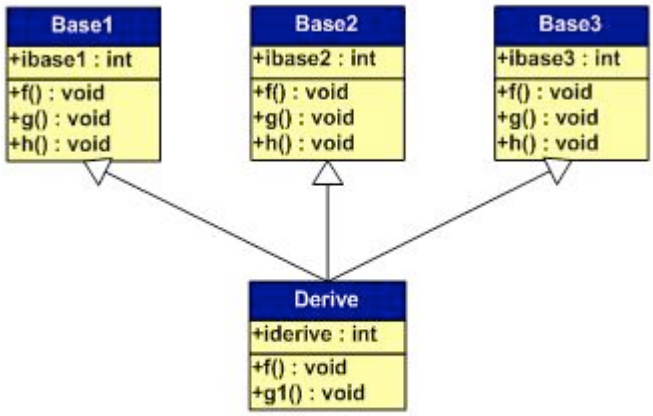
2）没有被覆盖的函数依旧。

这样，我们就可以看到对于下面这样的程序，

```
Base *b = new Derive();
b->f();
```

由 b 所指的内存中的虚函数表的 f()的位置已经被 Derive::f()函数地址所取代，于是在实际调用发生时，是 Derive::f()被调用了。这就实现了多态。

多重继承




多重继承

```
class Base1 {
public:
    int ibase1;
    Base1():ibase1(10) {}
    virtual void f() { cout << "Base1::f()" << endl; }
    virtual void g() { cout << "Base1::g()" << endl; }
    virtual void h() { cout << "Base1::h()" << endl; }

};

class Base2 {
```

```
public:
    int ibase2;
    Base2():ibase2(20) {}
    virtual void f() { cout << "Base2::f()" << endl; }
    virtual void g() { cout << "Base2::g()" << endl; }
    virtual void h() { cout << "Base2::h()" << endl; }
};

class Base3 {
public:
    int ibase3;
    Base3():ibase3(30) {}
    virtual void f() { cout << "Base3::f()" << endl; }
    virtual void g() { cout << "Base3::g()" << endl; }
    virtual void h() { cout << "Base3::h()" << endl; }
};


class Derive : public Base1, public Base2, public Base3 {
public:
    int iderive;
    Derive():iderive(100) {}
    virtual void f() { cout << "Derive::f()" << endl; }
    virtual void g1() { cout << "Derive::g1()" << endl; }
};
```

测试程序

```
typedef void(*Fun)(void);
Derive d;
int** pVtab = (int**)&d;

cout << "[0] Base1::_vptr->" << endl;
pFun = (Fun)pVtab[0][0];
cout << "        [0] ";
pFun();

pFun = (Fun)pVtab[0][1];
cout << "        [1] ";pFun();

pFun = (Fun)pVtab[0][2];
cout << "        [2] ";pFun();

pFun = (Fun)pVtab[0][3];
cout << "        [3] "; pFun();
```

```
pFun = (Fun)pVtab[0][4];
cout << "          [4] "; cout<<pFun<<endl;

cout << "[1] Base1.ibase1 = " << (int)pVtab[1] << endl;


int s = sizeof(Base1)/4;

cout << "[" << s << "] Base2::_vptr->"<<endl;
pFun = (Fun)pVtab[s][0];
cout << "          [0] "; pFun();

Fun = (Fun)pVtab[s][1];
cout << "          [1] "; pFun();

pFun = (Fun)pVtab[s][2];
cout << "          [2] "; pFun();

pFun = (Fun)pVtab[s][3];
out << "          [3] ";
cout<<pFun<<endl;

cout << "["<< s+1 <<"] Base2.ibase2 = " << (int)pVtab[s+1] << endl;

s = s + sizeof(Base2)/4;

cout << "[" << s << "] Base3::_vptr->"<<endl;
pFun = (Fun)pVtab[s][0];
cout << "          [0] "; pFun();

pFun = (Fun)pVtab[s][1];
cout << "          [1] "; pFun();

pFun = (Fun)pVtab[s][2];
cout << "          [2] "; pFun();

pFun = (Fun)pVtab[s][3];
  cout << "          [3] ";
cout<<pFun<<endl;

s++;
cout << "["<< s <<"] Base3.ibase3 = " << (int)pVtab[s] << endl;
s++;
cout << "["<< s <<"] Derive.iderive = " << (int)pVtab[s] << endl;
```

输出：

[0] Base1::_vptr->

[0] Derive::f()

[1] Base1::g()

[2] Base1::h()

[3] Driver::g1()

[4] 00000000 ç 注意：在 GCC 下，这里是 1

[1] Base1.ibase1 = 10

[2] Base2::_vptr->

[0] Derive::f()

[1] Base2::g()

[2] Base2::h()

[3] 00000000 ç 注意：在 GCC 下，这里是 1

[3] Base2.ibase2 = 20

[4] Base3::_vptr->

[0] Derive::f()

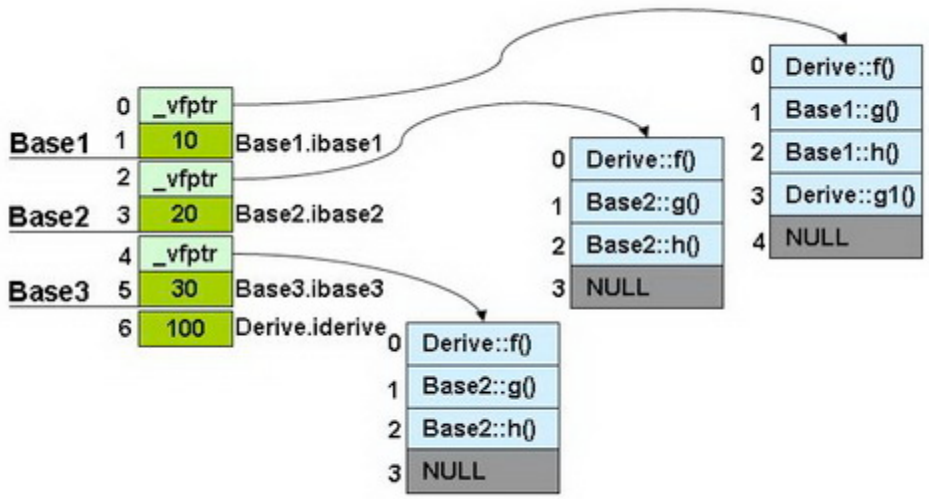
[1] Base3::g()

[2] Base3::h()

[3] 00000000

[5] Base3.ibase3 = 30

[6] Derive.iderive = 100




我们可以看到：

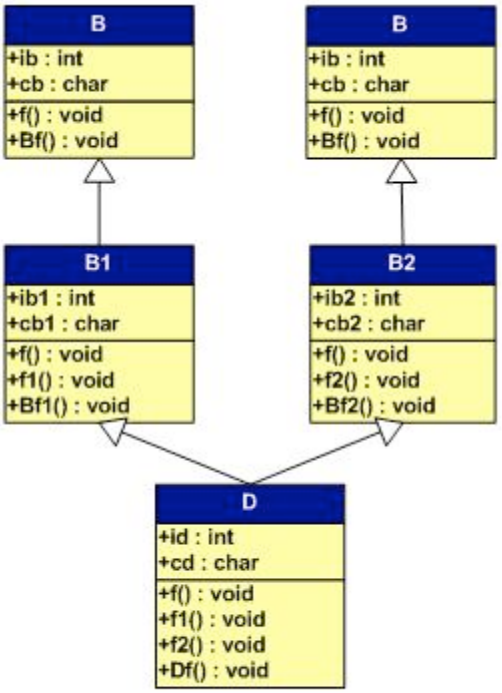
1） 每个父类都有自己的虚表。

2） 子类的成员函数被放到了第一个父类的表中。

3） 内存布局中，其父类布局依次按声明顺序排列。

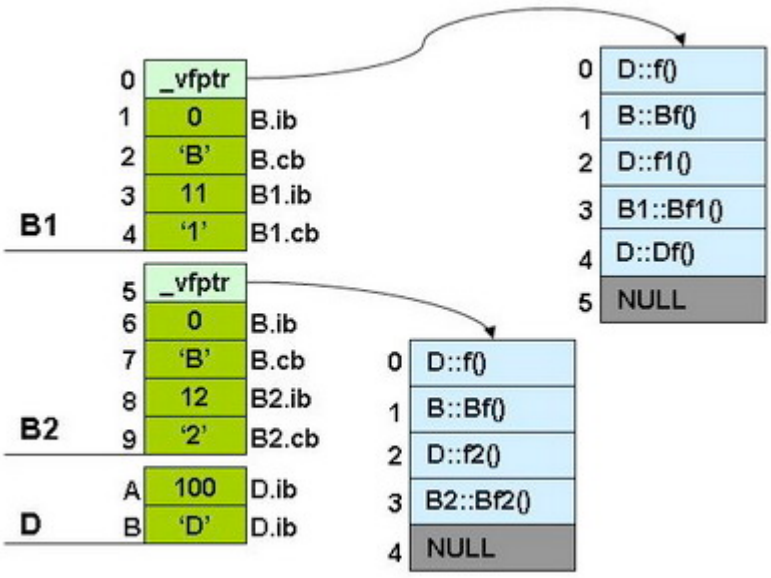
4） 每个父类的虚表中的 f()函数都被 overwrite 成了子类的 f()。这样做就是为了解决不同的父类类型的指针指向同一个子类实例，而能够调用到实际的函数。

重复继承



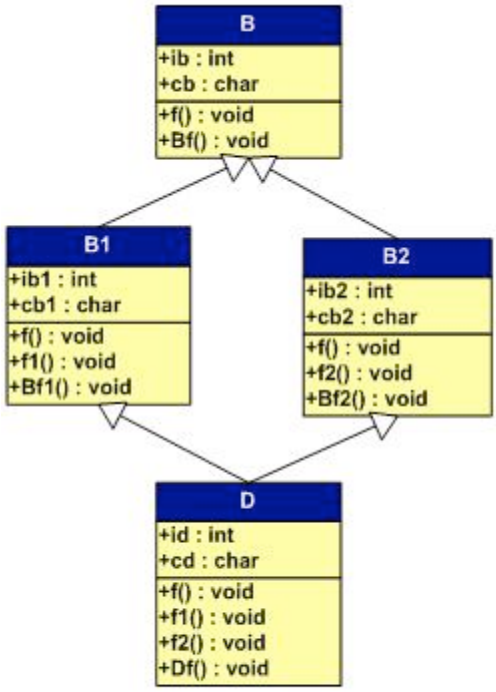

钻石型--重复继承

其类继承的源代码如下所示。其中，每个类都有两个变量，一个是整形（4 字节），一个是字符（1 字节），而且还有自己的虚函数，自己 overwrite 父类的虚函数。如子类 D 中，f()覆盖了超类的函数， f1() 和 f2() 覆盖了其父类的虚函数，Df()为自己的虚函数。




钻石型多重虚拟继承

虚拟继承的出现就是为了解决重复继承中多个间接父类的问题的。钻石型的结构是其最经典的结构。也是我们在这里要讨论的结构：




钻石型虚拟多重继承

GCC 3.4.4

[0] B1::_vptr ->
    [0] : D::f()
    [1] : D::f1()
    [2] : B1::Bf1()
    [3] : D::f2()
    [4] : D::Df()
    [5] : 1
[1] B1::ib1 : 11
[2] B1::cb1 : 1
[3] B2::_vptr ->
    [0] : D::f()
    [1] : D::f2()
    [2] : B2::Bf2()
    [3] : 0
[4] B2::ib2 : 12
[5] B2::cb2 : 2
[6] D::id : 100
[7] D::cd : D
[8] B::_vptr ->
    [0] : D::f()
    [1] : B::Bf()
    [2] : 0
[9] B::ib : 0

[10] B::cb : B
[11] NULL : 0
VC++ 2003
[0] D::B1::_vptr->
    [0] D::f1()
    [1] B1::Bf1()
    [2] D::Df()
    [3] 00000000
[1] = 0x0013FDC4  ç 该地址取值后是-4
[2] B1::ib1 = 11
[3] B1::cb1 = 1
[4] D::B2::_vptr->
    [0] D::f2()
    [1] B2::Bf2()
    [2] 00000000
[5] = 0x4539260  ç 该地址取值后是-4
[6] B2::ib2 = 12
[7] B2::cb2 = 2
[8] D::id = 100
[9] D::cd = D
[10]  = 0x00000000
[11] D::B::_vptr->
    [0] D::f()
    [1] B::Bf()
    [2] 00000000
[12] B::ib = 0
[13] B::cb = B
virtual void eat()=0;=0 表示不会真正调用，这样的虚函数是纯虚函数，有纯虚函数的类是抽象类。
抽象类不允许直接创建对象，只能用来指向或引用子类对象。
vi 下输入/ _cast 可以找到所有的类型转换。
类型转换算子：***_cast<要转换的类型>(要转换的数据)。
static_cast 支持的转换：1 数值类型之间，2 有类型的指针与 void*之间（不能用于两个有类型的指针之间）3 无名对象支持的转换如 static_cast<A>(100)。
const_cast 把常量转成变量，const_cast<类型&>（数据）&表示引用，不能将非指针的常变量转换为普通变量，如 const_cast<int > i。。
reinterpret_cast 重新解释内存（最危险的转换）用于不同类型指针之间转换，不能将 const 类型的指针转换为 void*类型，如 const int* pci=0; void* pv=reinterpret_cast<void*>(pci)。
dynamic_cast 从父类向子类转换时把关用，dynamic_cast<子类*>(父类地址)要求父类至少有一个虚函数，因为要用多态，成功（父类地址指向的是子类对象）返回子类地址，失败（父类地址指向的不是子类对象）返回 NULL。//以上三种是编译时转换，dynamic_cast 是运行时转换。
typedef unsigned int  UI 给类型其别名。
虚函数使用注意事项：1 构造函数不能是虚函数。2 如果类中任何一个成员函数是虚函数，

那么析构函数应为虚函数（用父类指针创建子类对象时，在删除指针时只调用父类的析构，将父类析构声明为虚函数则可以调用子类的析构函数）。

3 如果一个类可定被用作其他派生类的基类，尽可能使用虚函数，甚至纯虚函数。

友元（在类的内部用 friend 声明）是直接访问某个类中私有成员的桥梁（友元函数、友元类）。

```
class A{
    int data;
public:
    A(int d=0):data(d){}
    friend   A   add (const   A & a1,const A & a2)//声明友元，将 friend 加在函数声明的前面。
    friend   class   B;//声明友元类。
}//友元不是成员，可以放在类内的任何位置，友元中不能用 this 指针。
class B{
public:
    void twice(A& a1)//用对象.成员的方式访问 A 类私有成员
    {
        a1.data*=2;//数据×2。
    }

}
```

```
#include<iostream>
using namespace std;
class A{
    int data;
public:
    A(int d):data(d){
    cout<<"A()"<<endl;
    }
    void show()
    {
        cout<<"data ="<<data<<endl;
    }
    ~A(){cout<<"~A()"<<endl;}
       friend class B;
};
class B{
public:
    void twice(A& a1)
    {
        a1.data*=2;
    }
};
int main()
{
    A oa(50);
```

```
    B ob;
    ob.twice(oa);
    oa.show();
}
```

凡是向函数中传递对象时，几乎都会用引用，而且会加 const。

友元不会反转（A 是 B 的友元，但 B 不一定是 A 的友元），不能传递，不会继承。

凡是所有对象共用一份的数据，都要声明为静态数据成员（又称类变量）。

静态数据成员像全局变量一样在所有函数之外初始化。

```
class sd0705{
public:
    string name;
    static string teacher;//放在私有中就只能在本类中访问。
    sd0705(){}
    static void newteacher(const string* t)
    {
        teacher=t;
    }
};
string sd0705::teacher=”kobe”;//类外初始化。
sd0705::newteacher(“bryant”);//用类名来调用。
```

公共的静态成员函数属于整个类（没有 this 指针，没有当前变量），是一个全局函数，可以直接调用，不需要（可以但不提倡）通过任何对象(使用类名来调用)，静态成员函数中不得使用非静态数据成员。

cp day10/friend.cc day11

用友元函数 friend A add (const A & a1,const A & a2)时，要在类中加一个复制构造函数：A(const A& obj){cout<<””<<endl; 。

A(bool t):data(t?123:456){ cout<<"A(bool t)"<<endl; } A(char c):data((int)c){ cout<<"A(char c)"<<endl; }//根据对象初始化参数类型调用相应的构造函数。

凡是用旧对象初始化新对象一定会调用复制构造函数。

当对象中有指针成员指向动态内存时，用默认复制构造函数会出问题，两个指针会指向同一个地方。这是会出现混乱。最好自己写复制构造函数。

默认复制构造函数是浅拷贝，自己写复制构造函数是深拷贝。

```
#include<iostream>
using namespace std;
class A{
   int data;
public:
    A():data(100){
    cout<<"A()"<<endl;
    }
    A(int d):data(d){
    cout<<"A(int d)"<<endl;
    }
    A(bool t):data(t?123:456){
```

```
    cout<<"A(bool t)"<<endl;
    }
    A(char c):data((int)c){
    cout<<"A(char c)"<<endl;
    }
    A(const A& o):data(o.data){
    cout<<"A(const A& o)"<<endl;
    }
      void show()
    {
        cout<<"data = "<<data<<endl;
    }
    virtual ~A(){
            cout<<"~A()"<<endl;
              }
};
int main()
{
    A a1;
    A a2(50);
    A a3(false);
    A a4('v');
    A a5(a2);
    a1.show();
    a2.show();
    a3.show();
    a4.show();
    a5.show();
}
```

c++中允许使用 operator+来命名函数。可以直接用+来调用函数。

vc++中运算符重载时声明和定义最好不要分开。

运算符重载就是自己写运算符函数，来规定运算符如何工作。

双目运算符重载两种方式：

一友元：1 在类中声明 friend，2 定义函数，返回类型 operator 运算符（形参 1，形参 2），返回值作为运算结果。使用方法：operator+(obj1,obj2) 或 obj1+obj2。

二成员：1 直接在类中声明，2 定义函数，返回类型 operator 运算符 （形参）返回值作为运算结果。使用方法： obj1+obj2。

ostream & operator<<(ostream& os,const F& o)//cout 的重载只能用友元形式，cout 前不能用 const，因为输出时会改变 cout 中的很多状态成员。

```
ostream & operator<<(ostream& os,const F& o){
        os<<o.nu<<o.de<<endl;
        return os;
}
istream & operator>>(istream& is, F& o){
```

```
        is>>o.nu>>o.de;
        return is;
}
```
单目运算符重载两种方式：

一友元：返回类型 operator 运算符（形参）。

二成员：返回类型 operator 运算符（）{return *this;}。

运算符重载时，前++，前—看成单目运算符，后++后—看成双目运算符。

```
friend A& operator++(A & o){     //前++重载。
        o.data+=10;
        return o;
}
friend A   operator++(A & o,int ){     //后++重载。
        A old(o);
        o.data+=10;
        return old;
}
A& operator--(){       //前--重载。
        o.data-=10;
        return *this;
}
A operator--(int ){           //后--重载。
        A old(*this);
        data-=10;
        return old;
}
class A{
    int data;
public:
    A(int d=0):data(d){
    }
    friend istream& operator>>(istream& is,A& o){
            is>>o.data;
            return is;
    }
    friend ostream& operator<<(ostream& os,const A& o){
            os<<o.data;
            return os;
    }
    friend A& operator++(A& o){
            o.data+=10;
            return o;
    }
    friend A operator--(A& o,int ){
            A old(o);
```

```
            o.data-=10;
            return old;
    }
    A operator++(int ){
        A old(*this);
        data+=10;
        return old;
    }
    A& operator--(){
        data-=10;
        return *this;
    }
};
int main()
{
    A o1,o2;
    cout<<"please enter two object:"<<endl;
    cin>>o1>>o2;
    cout<<++o1<<endl;
    cout<<++o2<<endl;
    cout<<o1--<<endl;
    cout<<o2--<<endl;
}
```

运算符重载要求至少有一个操作数是自定义类型的。

强制类型转换重载：不需要写返回类型，只能写成成员形式 operator 类型（）{}

三目运算符不能重载。

= （） [ ] -> ->*只能用成员重载。

Complex 构造 +- * >> << (double) ~交换实部虚部。

文件和控制台的 I/O 是一样的。输入是从键盘读取，输出是写到屏幕。

输入是从控制台（文件）读取，输出是写到控制台（文件）。数据的传输叫流。

输入过程：键盘--->键盘缓冲区--->回车--->输入缓冲区--->程序读取。

如果两次输入两个字符，第一次输入一个字符时，如果多输入几个，则第二次的会不用输入，直接输出。因为第一次输入的存入了缓冲区。

cerr 和 clog(一般不用) 输出不用缓冲，也不能重定向。用法跟 cout 完全一样。先写 cout<<"hello";后写 cerr<<"world";会先输出 world。因为不需要缓冲。cerr 和 clog 都是 ostream 类的对象。

istream 类和 ostream 类都是 ios 类的子类。iostream 类是 istream 类和 ostream 类的子类。ifstream 类是 istream 类的子类。ofstream 类是 ostream 类的子类。

创建文件对象：头文件中加入 #include <fstream> ofstream fout("a.txt");(fout 是自定义的文件对象)。ifstream fin("a.txt");(fin 是自定义的文件对象)。

使用文件对象：fout<< ;fin>> ;

释放文件对象：fout.close(); fin.close();

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

```
#include<string>
using namespace std;
int main()
{
    ofstream fout("a.txt");
    fout<<"kobe"<<endl;
    fout<<'a'<<endl;
    fout<<5<<endl;
    fout.close();
    string str1=" ";
    char ch='\0';
    int n=0;
    ifstream fin("a.txt");
    getline(fin,str1);
    fin>>ch>>n;
    fin.close();
}
```

cout<<cout<<endl;cout<<cin<<endl;输出结果是地址。istream 和 ostream 都重载了一个类型转换运算符：(void *)会将 cin cout 转为地址输出。

一个 I/O 流对象在没有出错时可以当成真，在出错时可以当成假（if(!fin /cin) break;）。

fin("")中是 c 语言风格的字符串，使用时可以：string name;cin>>name;

ifstream fin(name.c_str());

>>又叫格式化输入运算符。可以将输入的字符转化为需要的类型。<< 是格式化输出，将输出的内容转化为字符。

非格式化的输入：get() getline() read()。

get()用于从输入流中读取一个字符，返回字符的 ascII 码。cout<<cin.get()<<endl;输入 a 后，显示 97。每调用一次都是新的，cin.get();cin.get()会读入两个字符。

get(char&)从输入流中读取一个字符，保存到形参中。cin.get(b)返回 cin。可以这样写：cin.get(b).get(c)。cin.get(char &)可以读入包括空格，制表符，回车在内的字符。

cin.getline(字符数组名，长度（大于要输入的一行的长度）)//c 风格的。getline(cin/fin,str)//c++风格的。两种风格的 getline 有第三个参数：结束字符，默认为 '\n'，会读走结束字符，但不会出现在字符串中（day12 password.cc）。

输入流一旦处于错误状态，就不再执行任何输入。

```
if(!cin){
    cout<<"error"<<endl;
    cin.clear();//回复错误状态，不是清空缓冲区。
}
while(cin.get()!='\n') ;//清空输入缓冲区。
cin.ignore(长度,'\n');//清空输入缓冲区。
void istream::ignore(int len=1,char ch=-1)
{
    for(int i=0;i<len;i++){
        if(get()==c) break;
    }
```

}
head -1 /etc/passwd //只看第一行。
id root //查看 root 用户的 id。
getline(cin,str,'$');
good
afternoon
$//会读入两行。
char ch=cin.peek();//peek 可以偷看输入流中下一个字符，返回 ascii 码。与 get 不一样，不会读走字符。int t=cin.peek();也可。
cin.putback(ch);//可以将读出的字符（在 ch 中）退回到缓存。
cin.width(10);//限制只输入 10 个字符。
cin>>wS;//跳过空白字符。
peek 与 putback 不要都用。
put（char）输出一个字符，cout.put(cin.get());//会输出字符。
fin 会把文件中的字符转化为 ascii 码存在内存中,fout 会把内存中的整形（ascii 码）数据转化为字符存入文件中，用 fout.write 和 fin.read 可以避免转换。
fout.write((char*)地址，字节数)会直接将内存(只支持 char 类型指针，要用类型转换)中的数据写到文件（不经过转换）//不要用 string 类型的地址。
fin.read(地址，字节数)会直接将文件的内容读到内存中//不要用 string 类型的地址//read 不会自动加'\0'。

```
1 #include<iostream>
  2 using namespace std;
  3 #include<fstream>
  4
  5 int main()
  6 {
  7         int n=123;
  8         double d=23.34;
  9         struct S{
 10                 char ch;
 11                 bool b;
 12                 char addr[100];
 13         };
 14         S s{'S',true,"kobe"};
 15         class A{
 16                 int data;
 17         public:
 18                 A(int t=0):data(t){}
 19                 void show() const{
 20                         cout<<"data =:"<<data<<endl;
 21                 }
 22         };
 23         A a(100);
 24         int n2;
```

```
25      double d2;
26      S s2;
27      A a2;
28      ofstream fout("var.dat");
29      fout.write((char*)&n,sizeof(n));
30      fout.write((char*)&d,sizeof(d));
31      fout.write((char*)&s,sizeof(s));
32      fout.write((char*)&a,sizeof(a));
33      fout.close();
34      ifstream fin("var.dat");
35      fin.read((char*)&n2,sizeof(n2));
36      fin.read((char*)&d2,sizeof(d2));
37      fin.read((char*)&s2,sizeof(s2));
38      fin.read((char*)&a2,sizeof(a2));
39      cout<<"n2=: "<<n2<<endl;
40      cout<<"d2=: "<<d2<<endl;
41      cout<<"s2.ch=: "<<s2.ch<<endl;
42      cout<<"s2.b=: "<<s2.b<<endl;
43      cout<<"s2.addr=: "<<s2.addr<<endl;
44      a2.show();
45      fin.close();
46 }
```

输入输出中，-1 有一个专门的名字：EOF。

cin/fin.gcount()//读取到的字节数。

fin.eof()//判断是否读到文件末尾。

cout.width(长度)//制定宽度输出，如果宽度小于实际长度，不会影响输出。如果宽度大，则前面是空格，是一次性的，只对下一次输出有效。

cout.fill('填充字符')//制定填充字符（不是一次性的）。

```
cout.width(10);
cout.fill('*');
cout<<"123"<<endl;//123 前用*填充。
```

用 cout.width(10);cout.fill('*');来重载<<运算符是输出格式控制符。

cout.precision(8)//设定输出 double 类型时输出 8 位有效数字（有四舍五入）。

cout.setf(ios::left|ios::hex)//输出左对齐同时是 16 进制。|表示同时使用两个标志。

cout.unsetf(ios::left)//清除标志。

ios::left ios::right(默认右对齐) ios::dec（默认十进制） ios::oct ios::hex ios::showbase（输出带进制前缀 0x 等）ios::showpoint（总是带小数点） ios::uppercase（十六进制的 0x 和 ABCD 以及科学计数法时的 e 大写） ios::showpos（带正负号输出） ios::scientific（科学计数法 ）

cout.precision(2);cout.setf(ios::fixed);//表示小数点后输出两位，要先去掉科学计数法再用。

输出十六进制数时，要将八进制和十进制清掉：cout.setf(ios::hex);cout.unsetf(ios::dec|ios::oct);

输出格式控制符：flush endl oct hex dec （cout<<hex<<;）。

setf 中的都可以用 cout 来实现//cout<<left;

cout.precision(8)等价于 cout<<setprecision(8)//要包含头文件<iomanip>。

cout.fill('#')等价于 cout<<setfill('#') //要包含头文件<iomanip>。

cout.width(10)等价于 cout<<setw(10) //要包含头文件<iomanip>。
fout 默认会清除文件中原有内容。fout(文件名，ios::app)//在原文件基础上追加内容。
ios::binary//二进制形式，linux 中文件都是二进制，windows 中有文本和二进制两种形式。文本形式中，'\n'在写文件时会写成'\r''\n'。读文件时会将'\r''\n'读成'\n'。不加 ios::binary 默认是 windows 模式。
fout<<"hello"<<endl; fout.seekp(0);//改写第 0 个字符。fout.put('H');
ifstream fin("文件名"，ios::in|ios::binary);fin.seekg(2)//读取低 3 个字符。
fstream fio(文件名，ios::in|ios::out)//既读又写。
内部类：成员内部类和局部内部类。
A::B objb;//B 在 A 内定义，是成员内部类//C 类中也可以同时定义内部类 B 类。
内部类的用处：1 隐藏类名，避免类名冲突。2 通过提供统一的内部类名来统一各个类的操作方法。
一个健壮的程序有 1/3 的错误处理代码。
处理异常：

```
try{
    if(!fin)
    throw 100.0;
}
catch (double e){
}
```

cout<<"what a good day!"<<endl;//100.0 与 double 一致则执行 catch 中的语句(执行完后会执行 what a good day!不会再返回到 try)，否则会中断程序。
catch(…){}//会匹配任何类型，一般放在末尾。
异常处理后，catch 后的语句继续执行。
如果当前没有捕获语句或者捕获语句中没有匹配的异常。则程序会跳出当前的函数。
在函数的调用处，如果没有捕获住异常，则直接跳转到更高一层的调用者。
main 中没有处理的异常会导致程序异常终止。
有继承的情况下一定要先 catch 子类，再 catch 父类。
catch 中如果不能处理异常，可以用 throw 抛出。
catch(int e ){throw;}
int(),char()等基本类型的无名变量是 0。
throw "stack overflow";要用 catch(const char*){}捕获，类型要严格匹配，不能只用 char*。
链表：

```
#include<iostream>
using namespace std;
typedef int T;
class List{
    struct Node{
        T data;
        Node* next;
        Node(const T& d):data(d){
            next=NULL;
        }
    };
```

```
    Node* head;
public:
    List():head(NULL){}
    void clear(){
        while(head!=NULL){
            Node* q=head->next;
            delete head;
            head=q;
        }
    }
    ~List(){clear();}
    void insert_ahead(const    T& n){
            Node* p=new Node(n);
            p->next=head;
            head=p;
    }
    void travel(){
        Node* p=head;
        while(p!=NULL){
            cout<<p->data<<' ';
            p=p->next;
        }
        cout<<endl;
    }
    int size(){
        int cnt=0;
        Node*p=head;
        while(p!=NULL){
            cnt++;
            p=p->next;
        }
        return cnt;
    }
    T gethead(){
        if(head==NULL)
            throw "no head";
        return head->data;
    }
    T gettail(){
        if(head==NULL)
            throw "no tail";
        Node* p=head;
        while(p->next!=NULL){
            p=p->next;
```

```
            }
            return p->data;
        }
        bool empty(){
            return head==NULL;
        }
        int find(const T& t){
            if(empty())
                throw "list is empty";
            Node* p=head;
            int time=0;
            while(p!=NULL){
                if(p->data==t){
                    return time;
                }
                p=p->next;
                time++;
            }
            return -1;
        }
        bool change(const T& o,const T& a){
                int t=find(o);
                if(t==-1)
                return false;
                Node* p=getpointer(t);
                p->data=a;
                return true;
        }
    private:
        Node* getpointer(const T& n){
                Node* p=head;
                for(int i=0;i<n;i++)
                    p=p->next;
                return p;
        }
    public:
        bool eraser(const T& t){
            int pos=find(t);
            if(pos==-1)
            return false;
            if(pos==0){
                Node*p=head->next;
                head=p;
            }
```

```
            else{
                Node*pre=getpointer(pos-1);
                Node*cur=pre->next;
                pre->next=cur->next;
                delete cur;
            }
            return true;
        }
};


int main()
{
        List obj;
        obj.insert_ahead(1);
        obj.insert_ahead(2);
        obj.insert_ahead(3);
        obj.insert_ahead(4);
        obj.insert_ahead(5);
        obj.travel();
        cout<<"The size of the list is :"<<obj.size()<<endl;
        cout<<"find 3 "<<obj.find(3)<<endl;
        cout<<"find 8 "<<obj.find(8)<<endl;
        obj.change(2,50);
        obj.travel();
        obj.eraser(3);
        obj.eraser(5);
        obj.travel();
}
```

链表：

```
#include<iostream>
using namespace std;
typedef int T;
class List{
        struct Node{
                T data;
                Node* next;
                Node(const T& d):data(d){
                        next=NULL;
                }
        };
        Node* head;
public:
        List():head(NULL){}
```

```
void clear(){
	while(head!=NULL){
		Node* q=head->next;
		delete head;
		head=q;
	}
}
~List(){clear();}
void insert_ahead(const    T& n){
		Node* p=new Node(n);
		p->next=head;
		head=p;
}
void insert_back(const T& t){
	Node* q=new Node(t);
	if(head==NULL)
	head=q;
	else{
		getpointer(size()-1)->next=q;
	}
}
void travel(){
	Node* p=head;
	while(p!=NULL){
		cout<<p->data<<' ';
		p=p->next;
	}
	cout<<endl;
}
int size(){
	int cnt=0;
	Node*p=head;
	while(p!=NULL){
		cnt++;
		p=p->next;
	}
	return cnt;
}
T gethead(){
	if(head==NULL)
		throw "no head";
	return head->data;
}
T gettail(){
```

```
            if(head==NULL)
                throw "no tail";
            Node* p=head;
            while(p->next!=NULL){
                p=p->next;
            }
            return p->data;
        }
        bool empty(){
            return head==NULL;
        }
        int find(const T& t){
            if(empty())
                throw "list is empty";
            Node* p=head;
            int time=0;
            while(p!=NULL){
                if(p->data==t){
                    return time;
                }
                p=p->next;
                time++;
            }
            return -1;
        }
        bool change(const T& o,const T& a){
                int t=find(o);
                if(t==-1)
                return false;
                Node* p=getpointer(t);
                p->data=a;
                return true;
        }
    private:
        Node* getpointer(const T& n){
                Node* p=head;
                for(int i=0;i<n;i++)
                    p=p->next;
                return p;
        }
    public:
        bool eraser(const T& t){
            int pos=find(t);
            if(pos==-1)
```

```
            return false;
            if(pos==0){
                Node*p=head->next;
                head=p;
            }
            else{
                Node*pre=getpointer(pos-1);
                Node*cur=pre->next;
                pre->next=cur->next;
                delete cur;
            }
            return true;
        }
};


int main()
{
   List obj;
   obj.insert_back(5);
   obj.travel();
}
```

链表逆置：

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

struct node
{
   int d;
   node* next;

};
void insert(node* &head,int t)
{
       node* p=new node;
       p->d=t;
       p->next=NULL;
       node* q;
       if(head==NULL){
            head=p;
       }
       else{
             q=head;
             while(head->next!=NULL)
```

```
                head=head->next;
            head->next=p;
            head=q;
        }
}
void reverse(node* &head)
{
        node* p=head;
        node* q=p->next;
        node* r=NULL;
        while(q!=NULL){
                r=q->next;
                q->next=p;
                p=q;
                q=r;
        }
        head->next=NULL;
        head=p;
}
void show(node* &head)
{
        node* p=head;
        while(p!=NULL){
                cout<<p->d<<' ';
                p=p->next;
        }
        cout<<endl;
}
int main()
{
        node* head=NULL;
        int t;
        int i=0;
        while(i<7){
                cin>>t;
                insert(head,t);
                i++;
        }
        show(head);
        reverse(head);
        show(head);
}
```

队列：

```
#include<iostream>
```

```
using namespace std;
typedef double T;
class List{
	struct Node{
		T data;
		Node* next;
		Node(const T& d):data(d){
			next=NULL;
		}
	};
	Node* head;
public:
	List():head(NULL){}
	void clear(){
		while(head!=NULL){
			Node* q=head->next;
			delete head;
			head=q;
		}
	}
	~List(){clear();}
	void insert_ahead(const	T& n){
			Node* p=new Node(n);
			p->next=head;
			head=p;
	}
	void insert_back(const T& t){
		Node* q=new Node(t);
		if(head==NULL)
		head=q;
		else{
			getpointer(size()-1)->next=q;
		}
	}
	void travel(){
		Node* p=head;
		while(p!=NULL){
			cout<<p->data<<' ';
			p=p->next;
		}
		cout<<endl;
	}
	int size(){
		int cnt=0;
```

```
        Node*p=head;
        while(p!=NULL){
                cnt++;
                p=p->next;
        }
        return cnt;
}
T gethead(){
        if(head==NULL)
                throw "no head";
        return head->data;
}
T gettail(){
        if(head==NULL)
                throw "no tail";
        Node* p=head;
        while(p->next!=NULL){
                p=p->next;
        }
        return p->data;
}
bool empty(){
        return head==NULL;
}
int find(const T& t){
        if(empty())
                throw "list is empty";
        Node* p=head;
        int time=0;
        while(p!=NULL){
                if(p->data==t){
                        return time;
                }
                p=p->next;
                time++;
        }
        return -1;
}
bool change(const T& o,const T& a){
                int t=find(o);
                if(t==-1)
                return false;
                Node* p=getpointer(t);
                p->data=a;
```

```
                return true;
        }
private:
        Node* getpointer(int n){
                Node* p=head;
                for(int i=0;i<n;i++)
                        p=p->next;
                return p;
        }
public:
        bool eraser(const T& t){
                int pos=find(t);
                if(pos==-1)
                return false;
                if(pos==0){
                        Node*p=head->next;
                        head=p;
                }
                else{
                        Node*pre=getpointer(pos-1);
                        Node*cur=pre->next;
                        pre->next=cur->next;
                        delete cur;
                }
                return true;
        }
};
class Queue{
        List l;
public:
        void push(const T& t){l.insert_back(t);}
        void pop(){l.eraser(l.gethead());}
        T front(){return l.gethead();}
        T back(){return l.gettail();}
        bool empty(){return l.empty();}
        int size(){return l.size();}
        void clear(){l.clear();}
};

int main()
{
        Queue q;
        q.push(1.0);
        q.push(2.0);
```

```
    q.push(3.0);
    q.push(4.0);
    q.push(5.0);
    cout.setf(ios::showpoint);
    while(!q.empty()){
        cout<<q.front()<<' ';
        q.pop();
    }
    cout<<endl;
}
```

数组实现的队列：

```
#include<iostream>
using namespace std;
typedef int T;
class Queue{
    T a[10];
    int num;
public:
    void push(const T& t){
        if(full())
            throw "stack overflow";
        a[num++]=t;
    }
    void pop(){
        if(empty())
            throw "empty stack";
        num--;
        for(int i=0;i<num;i++)
            a[i]=a[i+1];
    }
    T top(){
        if(empty())
            throw "no top queue";
        return a[0];
    }
    bool empty(){return num==0;}
    bool full(){return num==10;}
    int size(){return num;}
    int capacity(){return 10;}
    void clear(){num=0;}
    Queue():num(0){}

};
int main()
```

```
{
    try{
    Queue q;
    for(int i=0;i<10;i++)
        q.push(i*5);
    while(!q.empty()){
        cout<<q.top()<<' ';
        q.pop();
    }
    cout<<endl;
    for(int i=0;i<10;i++)
        q.push(i);
    cout<<"full ?"<<q.full()<<endl;
    }
    catch(const char*){
        cout<<"exception"<<endl;
    }catch(...){
        cout<<"unknow exception"<<endl;
    }
};
```

栈：

```
#include<iostream>
#include<string>
using namespace std;
typedef string T;
class List{
    struct Node{
        T data;
        Node* next;
        Node(const T& d):data(d){
            next=NULL;
        }
    };
    Node* head;
public:
    List():head(NULL){}
    void clear(){
        while(head!=NULL){
            Node* q=head->next;
            delete head;
            head=q;
        }
    }
    ~List(){clear();}
```

```
void insert_ahead(const    T& n){
        Node* p=new Node(n);
        p->next=head;
        head=p;
}
void insert_back(const T& t){
    Node* q=new Node(t);
    if(head==NULL)
    head=q;
    else{
        getpointer(size()-1)->next=q;
    }
}
void travel(){
    Node* p=head;
    while(p!=NULL){
        cout<<p->data<<' ';
        p=p->next;
    }
    cout<<endl;
}
int size(){
    int cnt=0;
    Node*p=head;
    while(p!=NULL){
        cnt++;
        p=p->next;
    }
    return cnt;
}
T gethead(){
    if(head==NULL)
        throw "no head";
    return head->data;
}
T gettail(){
    if(head==NULL)
        throw "no tail";
    Node* p=head;
    while(p->next!=NULL){
        p=p->next;
    }
    return p->data;
}
```

```
        bool empty(){
                return head==NULL;
        }
        int find(const T& t){
                if(empty())
                        throw "list is empty";
                Node* p=head;
                int time=0;
                while(p!=NULL){
                        if(p->data==t){
                                return time;
                        }
                        p=p->next;
                        time++;
                }
                return -1;
        }
        bool change(const T& o,const T& a){
                        int t=find(o);
                        if(t==-1)
                        return false;
                        Node* p=getpointer(t);
                        p->data=a;
                        return true;
        }
private:
        Node* getpointer(int n){
                        Node* p=head;
                        for(int i=0;i<n;i++)
                                p=p->next;
                        return p;
        }
public:
        bool eraser(const T& t){
                int pos=find(t);
                if(pos==-1)
                return false;
                if(pos==0){
                        Node*p=head->next;
                        head=p;
                }
                else{
                        Node*pre=getpointer(pos-1);
                        Node*cur=pre->next;
```

```
                pre->next=cur->next;
                delete cur;
            }
            return true;
        }
};
class Stack{
        List l;
public:
        void push(const T& t){
            l.insert_ahead(t);
        }
        void pop(){
            l.eraser(top());
        }
        T top(){
            return l.gethead();
        }
        bool empty(){
            return l.empty();
        }
        int size(){
            return l.size();
        }
        void clear(){
            l.clear();
        }
};

int main()
{
        Stack s;
        s.push("hello");
        s.push("students");
        s.push("we");
        s.push("are");
        s.push("kobe");
        s.push("fans");
        while(!s.empty()){
            cout<<s.top()<<' ';
            s.pop();
            }
        cout<<endl;
}
```

数组实现的栈：

```
#include<iostream>
using namespace std;
typedef int T;
class Stack{
	T a[10];
	int num;
public:
	void push(const T& t){
		if(full())
			throw "stack overflow";
		a[num++]=t;
	}
	void pop(){
		if(empty())
			throw "empty stack";
		num--;
	}
	T top(){
		if(empty())
			throw "no top stack";
		return a[num-1];
	}
	bool empty(){return num==0;}
	bool full(){return num==10;}
	int size(){return num;}
	int capacity(){return 10;}
	void clear(){num=0;}
	Stack():num(0){}

};
int main()
{
	try{
	Stack s;
	for(int i=0;i<10;i++)
		s.push(i*5);
	while(!s.empty()){
		cout<<s.top()<<' ';
		s.pop();
	}
	cout<<endl;
	for(int i=0;i<10;i++)
		s.push(i);
```

```
        cout<<"full ?"<<s.full()<<endl;
        s.push(100);
        cout<<"can't see this"<<endl;
        }
        catch(const char*){
                cout<<"exception"<<endl;
        }catch(...){
                cout<<"unknow exception"<<endl;
        }
}
```

栈实现的队列：

```
#include<iostream>
using namespace std;
typedef int T;
class Stack{
        T a[10];
        int num;
public:
        void push(const T& t){
                if(full())
                        throw "stack overflow";
                a[num++]=t;
        }
        void pop(){
                if(empty())
                        throw "empty stack";
                num--;
        }
        T top(){
                if(empty())
                        throw "no top stack";
                return a[num-1];
        }
        bool empty(){return num==0;}
        bool full(){return num==10;}
        int size(){return num;}
        int capacity(){return 10;}
        void clear(){num=0;}
        Stack():num(0){}

};
class Queue{
        Stack s1,s2;
public:
```

```
        void push(const T& t){
                if(full())
                        throw "queue is full";
                s1.push(t);
        }
        void pop(){
                if(empty())
                        throw "queue is empty";
        }
        T top(){
                T temp,top;
                while(!s1.empty()){
                        temp=s1.top();
                        s1.pop();
                        s2.push(temp);
                }
                top=s2.top();
                s2.pop();
                while(!s2.empty()){
                        temp=s2.top();
                        s2.pop();
                        s1.push(temp);
                }
                return top;
        }
        bool empty(){return s1.empty();}
        bool full(){return s1.full();}
        int size(){return s1.size();}
        int capacity(){return 10;}
        void clear(){s1.clear();}
};
int main()
{
        try{
        Queue q;
        for(int i=0;i<10;i++)
                q.push(i*5);
        while(!q.empty()){
                cout<<q.top()<<' ';
                //q.pop();
        }
        cout<<endl;
        for(int i=0;i<10;i++)
                q.push(i);
```

```
        cout<<"full ?"<<q.full()<<endl;
        while(!q.empty()){
                cout<<q.top()<<' ';
                //q.pop();
        }
        cout<<endl;
        }
        catch(const char*){
                cout<<"exception"<<endl;
        }catch(...){
                cout<<"unknow exception"<<endl;
        }
}
```

二叉树：

```
#include<iostream>
using namespace std;
typedef int T;
class Tree{
        struct Node{
                T data;
                Node* right;
                Node* left;
                Node(const T& t):data(t){left=NULL;right=NULL;}
        };
        Node* root;
public:
        Tree(){root=NULL;}
        void travel(Node* tree){
                if(tree==NULL)
                        return;
                travel(tree->left);
                cout<<tree->data<<' ';
                travel(tree->right);
        }
        void travel(){
                travel(root);
        }
        void clear(Node* tree){
                if(tree=NULL)
                        return;
                clear(tree->left);
                clear(tree->right);
                delete tree;
                tree=NULL;
```

```
}
void clear(){
	clear(root);
}
int size(Node* tree){
	if(tree==NULL)
		return 0;
	return size(tree->left)+size(tree->right)+1;
}
int size(){
	return size(root);
}
void insert(Node*& tree,Node* p){
	if(tree==NULL)
		tree=p;
	else if(p==NULL)
		return;
	else if(tree->data>p->data)
		insert(tree->left,p);
	else
		insert(tree->right,p);
}
void insert(const T& t){     // 要与 const 对应：Node(const T& t):
	Node*    p=new Node(t);
	insert(root,p);
}
Node*& find(Node*& tree,const T& t){
	if(tree==NULL)
		return tree;
	if(tree->data==t)
		return tree;
	else if(tree->data>t)
		return find(tree->left,t);
	else
		return find(tree->right,t);
}
void erase(Node*&tree,const T& t){
	Node*& p=find(tree,t);
	if(p==NULL) return;
	insert(p->right,p->left);
	Node* q=p;
	p=p->right;
	delete q;
}
```

```
    void erase(const T& t){
        erase(root,t);
    }
    void update(const T& o,const T& n){
        Node* p=find(root,o);
        if(p==NULL)
            return;
        erase(root,o);
        p=new Node(n);
        insert(root,p);
    }
    int high(Node* tree){
        if(tree==NULL)
            return 0;
        return ((high(tree->left)>high(tree->right))?high(tree->left):high(tree->right))+1;
    }
    int high(){
        return high(root);
    }
};
int main()
{
    Tree t;
    t.insert(2);
    t.insert(4);
    t.insert(6);
    t.insert(8);
    t.insert(10);
    t.insert(12);
    t.insert(14);
    t.insert(16);
    t.insert(18);
    t.insert(20);
    cout<<"the tree has    "<<t.size()<<" nodes"<<endl;
    cout<<"the high is "<<t.high()<<endl;
    t.travel();
    cout<<endl;
    t.erase(18);
    t.erase(14);
    t.update(12,24);
    t.update(10,30);
    cout<<"the high is "<<t.high()<<endl;
    t.travel();
    cout<<endl;
```

```
}
```
在二叉树中将所有的叶子用链表连接起来：
```
void link_ins(link* &head,link* p)
    {
        link* q=head;
        if(head==NULL)
            head=p;
        else if(p==NULL)
            return;
        else {
            while(q->next!=NULL)
                q=q->next;
            q->next=p;
            //head=q;
        }
    }
    void link_ins(const int& t)
    {
        link* p=new link(t);
        p->next=NULL;
        link_ins(head,p);
    }
void leef(node* tree)
    {
        if(tree==NULL)
            return;
        else{
            leef(tree->left);
            if((tree->left==NULL)&&(tree->right==NULL))
                link_ins(tree->data);
            leef(tree->right);
        }
    }
    void leef()
    {
        leef(root);
    }
    void show()
    {
        link* p=head;
        while(p!=NULL){
            cout<<p->da<<' ';
            p=p->next;
        }
```

```
            cout<<endl;
        }
    tree.leef();//main()中
    tree.show();
```

二叉排序树整合了两种结构的优点：有序数组（查找快）和链表（插入和删除快）。

先根，再左，后右是前序遍历（先根遍历），先左，再根，后右是中序遍历（中根遍历，使用最多），先左，再右，后根是后续遍历（后根遍历）。

二叉树：

```
class Tree{
    struct Node{
        T data;
        Node* left;
        Node*right;
        Node(const T & t):data(t){left=NULL;right=NULL;}
    };
    Node* root=NULL;
public:
    //Tree(){tree=NULL;}
    void travel(Node * tree){
        if(tree==NULL)
            return;
        travel(tree->left);
        cout<<tree->data<<' ';
        travel(tree->right);
    }
    void clear(Node* & tree){//要用引用。
        if(tree==NULL)
        return;
        clear(tree->left);
        clear(tree->right);
        delete tree;
        tree=NULL;
    }
    int size(Node* tree){
        if(tree==NULL)
        return 0;
        return size(tree->left)+size(tree->right)+1;
    }
    void insert(Node*& tree,Node* p){
         if(tree==NULL)
            tree=p;
        else if(p==NULL)
            return;
        else if(p->data<tree->data)
```

```
            insert(tree->left,p);
        else
            insert(tree->right,p);
    }
    Node*& find(Node*& tree,const T& t){
        if(tree==NULL)
            return tree;
        if(tree->data==t)
            return tree;
        else if(t<tree->data)
            find(tree->left,t);
        else
            find(tree->right,t);
    }
```

删除节点：1 合并左右分支，2 让指针指向合并后的分支，3 释放要删除的节点。

```
    void erase(Node *&tree,const T& t){
        Node*&p=find(tree,t);
        if(p==NULL)    return;
        insert(p->right,p->left);//合并左右分支。
        Node* q=p;
        p=p->right;//指针指向合并后的分支。
        delete q;
    }
    void update(const T & o,const T & n){
        Node* p=find(root,o);
        if(p==NULL) return;
        erase(root,o);
        p=new Node(n);
        insert(root,p);
    }
    int   high(Node* tree){
        if(tree==NULL)
            return 0;
        return (high(tree->left)>high(tree->right)?high(tree->left):high(tree->right))+1;
};//达内论坛有源程序。
```

表达式树：把优先级最低的运算符放在根节点，然后把运算符左右两部分分别产生两棵树来组成左右子树。

算法是在有限步骤内求解某一问题所使用的一组定义明确的规则。

算法特征：有穷性，确切性，输入，输出，可行性。

c++程序不可靠的原因：1 野指针 2 没有错误检查 3 垃圾数据 4 局部变量的引用和地址 5 使用已经 delete 的空间 6 越界访问数组。

设计出复杂性尽可能低的算法是我们在设计算法时追求的一个重要目标。

线性查找：o(n),二分查找 o（logN）选择排序，冒泡排序，插入排序 o（N*N）快速排序 O（N*logN）

直接排序：每次选择一个最大（最小）的元素放到应该的位置。
1 i 从 0-N-1，2 找从[i]到[N-1]的数据中最小元素的位置 pos。3 把它和[i]交换。

```
void sort(T a[],int n)
{
    T   pos,temp;
    for(int i=0;i<n -1;i++){
        pos=i;
        for(int k=i+1;k<n ;k++)
            if(a[pos]>a[k])
                pos=k;
            temp=a[i];
            a[i]=a[pos];
            a[pos]=temp;
    }
}
```

冒泡排序：每次只比相邻的两个元素。顺序不合适就交换。适合于只有少量数据顺序不对的场合。

```
void sort(T a[],int n)
{
    T swap,temp,turn=0;
    do{
        swap=0;
        for(int i=0;i<n-turn-1;i++){
            if(a[i]>a[i+1]){
                temp=a[i];
                a[i]=a[i+1];
                a[i+1]=temp;
                swap++;
            }
        }
        turn++;
    }while(swap);
}
```

插入排序：对每个新元素（除第一个），1 找合适的插入位置（先把数据存一份）。2 把从那个位置开始的所有数据右移。3 把新元素放到插入位置。

```
void sort(T a[],int n)
{
    T temp;
    int p;
    for(int i=1;i<n;i++){
        temp=a[i];
        for(p=i;p>0&&a[p-1]>temp;p--)
            a[p]=a[p-1];
        a[p]=temp;
```

```
    }
}
```
选择排序优于插入排序优于冒泡排序。

快速排序：选出一个分界值，把数组分成两个部分。大于等于分界值的元素集中到数组的某一部分，小于分界值的部分集中到数组的另一部分。对分成的两部分分别再做同样的操作。

```
#include<algorithm>
typedef int T;
using namespace std;
void sort(int a[],int n)
{
    if(n<=1) return;//不加会出段错误。
    swap(*a,a[n>>1]);
    T* l=a+1;
    T* r=a+n-1;
    T v=*a;
    while(l<r){
        while(*l<v&&l<r) l++;
        while(*r>=v&&r>a) r--;
        if(l<r) swap(*l,*r);
    }
    if(v>*r) swap(*a,*r);
    sort(a,r-a);
    sort(r+1,n-(r-a)-1);
}
```
合并排序：

```
#include<iostream>
#include<cstdlib>
using namespace std;
void merge_sort(int* a,int x,int y,int* t)
{
    if(y-x>1)
    {
        int m=x+(y-x)/2;
        int p=x,q=m,i=x;
        merge_sort(a,x, m, t);
        merge_sort( a, m, y, t);
        while(p<m||q<y)
        {
            if(q>=y||(p<m&&a[p]<=a[q])) t[i++]=a[p++];
            else t[i++]=a[q++];
        }
        for(i=x;i<y;i++) a[i]=t[i];
    }
}
```

```
int main()
{
	int a[10]={5,4,6,7,8,9,2,3,1,10};
	int t[10];
	memset(t,0,10);
	merge_sort(a,0,9,t);
	for(int i=0;i<10;i++)
		cout<<a[i]<<' ';
	cout<<endl;
}
```

模板（《c++标准程序库》）template<typename　T>(模板头)//表示 T 是任意类型，要写在定义的函数的前面，每个函数写一次。编译器优先选用非模板函数。

模板的声明和定义不要分开。模板要保存在头文件中。把函数写成模板是函数模板，把类写成模板是类模板。typename T 模板形参，确定模板形参的过程叫实例化。类模板要实例化后才能创建对象。Stack <int> st//实例化。Stack<int>共同构成类名。

template<typename　T，typename　U>//可同时定义多个。实例化：pair<int,double> p1;

```
#include<iostream>
#include<algorithm>
using namespace std;
template<typename T>
void sort (T* a,int n){
	int pos;
	for(int i=0;i<n-1;i++){
		pos=i;
		for(int j=1;j<n;j++)
			if(a[pos]>a[j])
				pos=j;
		swap(a[i],a[pos]);
	}
}
template<typename T>
void disp (T* a,int n){
	for(int i=0;i<n;i++)
		cout<<a[i]<<' ';
		cout<<endl;
}
int main()
{
	int a[5]={4,6,2,7,8};
	double b[6]={1.1, 2.3, 4.2, 5.3, 6.7, 8.2};
	sort(a,5);
	disp(a,5);
	sort(b,6);
	disp(b,6);
```

}

STL 标准模板库：1 类模板（容器），2 函数模板（通用算法）。

容器都有一个内部类：iterator(迭代器，模拟指针的功能，又称智能指针)。几乎所有的函数模板的形参是 iterator 类型的。标准模板库不直接用指针，而是用迭代器来代替。

标准模板库中的区间包括开始指针指向的元素，不包括结束指针指向的元素。

容器分为：1 序列式 2 关联式。序列式包括：1vector 动态数组，长度可变 2deque 双端队列，两端都可插入数据 3list 双向链表。关联式，所有的关联式容器都是二叉查找树模板，包括：1map 映射，不允许重复 2multimap 多映射，允许重复 3set 数据集，不允许重复 4multiset 多数据集，允许重复。用什么容器就要包含相应的头文件，如<vector>

每一个容器包含四个迭代器：begin() 指向第一个元素。end()指向最后一个元素之后（end 指向的元素不属于这个容器）。

```
1 #include<iostream>
2 using namespace std;
3 #include<list>
4
5 int main()
6 {
7       list<int> l1;
8       int a[5]={3,4,5,6,7};
9       list<int> l2(a,a+5);
10      cout<<"l1.size() :"<<l1.size()<<endl;//调用函数。
11      cout<<"l2.size() :"<<l2.size()<<endl;
12      list<int>::iterator it ;// list<int>是类名。
13      for(it=l2.begin();it!=l2.end();it++)//迭代器用法同指针。
14            cout<<*it<<' ';
15      cout<<endl;
16      it=l2.begin();
17      it++;
18      l2.erase(it); //模板内部函数
19      l2.insert(l2.begin(),100);//模板内部函数
20      l2.insert(l2.end(),200);
21      for(it=l2.begin();it!=l2.end();it++)
22            cout<<*it<<' ';
23      cout<<endl;
  }
```

容器构造函数：

C<T>c 创建一个名为 c 的空容器，T 是元素类型。

C c(c2)创建容器 c2 的副本 c;c 和 c2 必须具有相同的容器类型。

C c(b,e) 创建 c,其元素是迭代器 b 和 e 标示的范围内的元素的副本，适用于所有容器。

C c(n,t) 用 n 个值为 t 的元素创建容器 c,其中值 t 必须是容器类型 C 的元素类型的值，只是用于顺序容器。

C c(n) 创建有 n 个值初始化元素的容器 c。只适用于顺序容器。

将一个容器初始化为另一个容器的副本：

vector<int> ivec;

vector<int>  ivec2(ivec);

初始化为一段元素的副本：

list<string> slist(svec.begin(),svec.end());

vector<string>::iterator mid=svec.begin()+svec.size()/2;

deque<string> front(svec.begin(),mid);

deque<string> back(mid,svec.end());

分配和初始化指定数目的元素：

const list<int>::size_type list_size=64;

list<string> slist(list_size,"eh");//初始化为 64 字符串，每个是 eh。

迭代器为所有标准库容器类型提供的运算：

*iter//返回迭代器所指向的元素的引用。

iter->mem//对 iter 进行解引用，获取指定元素中名为 mem 的成员，等效于(*iter).mem。

++iter//给 iter 加 1。

iter++

--iter//给 iter 减 1。

iter--

iter1==iter2  iter1!=iter2//比较两个迭代器是否相等（或不等）当两个迭代器指向同一个容器中的同一个元素或者它们都指向同一个容器的超出末端的下一位置是，两个迭代器相等。

vector 和 deque 容器的迭代器提供的额外运算：

iter+n

iter-n//产生指向容器中前面（后面）第 n 个元素的迭代器。

iter1+=iter2

iter1-=iter2//将 iter1 加上或减去 iter2 的运算结果赋给 iter1。

>,>=,<,<=//当一个迭代器指向的元素在容器中位于另一个迭代器指向的元素之前，则前一个迭代器小于后一个迭代器。

vector<int>::iterator iter=vec.begin()+vec.size()/2;

list 容器不支持迭代器的算术运算和关系运算。

迭代器的范围：begin 和 end，end 指向最后一个元素的下一个位置。但 begin 和 end 相等时迭代器范围为空，当不相等时，迭代器范围内至少有一个元素。

容器定义的类型别名：

size_type//无符号整型，存储容器长度。

iterator//迭代器类型。

const_iterator//只读迭代器类型。

reverse_iterator//按逆序寻址元素的迭代器。

const_reverse_iterator//只读逆序迭代器。

difference_type//足够存储两个迭代器差值的有符号整型，可为负数。

value_type//元素类型。

reference//元素的左值类型。

const_reference//元素的常量左值类型。

向顺序容器(list vector deque)中添加元素，在尾部：

while(cin>>text_word)

container.push_back(text_word);// container 的类型是 list vector 和 deque。

c.push_back(t)//在容器 c 的尾部添加值为 t 的元素,返回 void 类型,只适用于 list 和 deque

c.push_front(t)//在容器 c 的前端添加值为 t 的元素，返回 void 类型，只适用于 list 和 deque

for(size_t ix=0;ix!=4;++ix)

ilist.push_front(ix);//容器中的元素是 3,2,1,0。

c.insert(p,t)//在 p 所指向的元素前面插入值为 t 的新元素，返回指向新元素的迭代器。

list<string> lst;

list<string>::iterator iter=lst.begin();

while(cin>>word)

iter=lst.insert(iter,word);//相当于 push_front 函数。

c.insert(p,n,t)// 在 p 所指向的元素前面插入 n 个值为 t 的新元素，返回 void 类型。

svec.insert(svec.end(),10,"anna");

c.insert(p,b,e)//在 p 所指向的元素前面插入有迭代器 b 和 e 标记范围内的元素，返回 void 类型。

任何 insert 和 push 操作都可能导致迭代器失效，必须确保迭代器在每次循环后得到更新。

避免存储 end 操作返回的迭代器:

last=v.end();

while(first!=last){//错误，应改成 while(first!=v.end()){

first=v.insert(first,42);

++first;//添加运算使存储在 last 中的迭代器失效，该迭代器既没有指向容器 v 的元素，也不再指向 v 的超出末端的下一位置。

}

容器中的元素都是副本，系统将元素值复制到容器里，此后，容器内的元素值发生变化时，被复制的原值不会受到影响。

man –k dir//对要查询功能模糊时用-k。

UC（unix c）：1 文件系统（1 目录操作 2 文件操作）2 多进程（1 进程间关系 2 进程间通信）3 网络通信（1 基础知识 2 实践：TCP UDP）。

图形开发在 linux 下：Qt 和 KDE。

用户配置文件在自己的主目录下。

用户信息在/etc/passwd 中，组信息在/etc/group 中。

env 查看环境变量。echo $ PWD//显示具体环境变量。

int main(int argc,char* argv[],char* env[])//第三个参数接收环境变量。

编程习惯：向函数传递指针数组的时候，如果没有说明元素个数，总是用一个 NULL 放在末尾，以 NULL 作为结束标志。

char* name="NAME";char* value=NULL;value=getenv(name);//输出指定环境变量的值,要包含头文件<stdlib.h>c++中要写成<cstdlib>。

export 变量名=值//设定环境变量，只对本进程及其子进程有用。

bash exit//打开关闭终端。

编程习惯 2：返回指针的函数，正常返回一个非空指针，出错返回 NULL。

编程习惯 3：返回整数的函数，正常返回一个非负整数（通常是 0），出错返回一个负数（通常是-1）。

perror("提示")会输出提示：错误信息的文字描述（加头文件<stdio.h>）。

fg 前台运行，bg 后台运行。

进程四种状态：1O 正在执行 2S 等待 cpu 调度 3R 已经准备好，还没开始执行 4T 挂起，不会有 cpu 执行时间。day17

cout<<getlogin()<<endl;//输出登陆名,要加头文件<unistd.h>。
cout<<getuid()<<endl;//取得用户自己的 id, 要加头文件<unistd.h>。
cout<<geteuid()<<endl;//取得有效 id，执行别人建的文件，结果是别人的 id, 要加头文件<unistd.h>。
chmod u+s a.out//设置-用户权限。
用户密码保存在/etc/shadow 中。
cout<<getgid()<<endl;//取得用户自己的组 id, 要加头文件<unistd.h>。
cout<<getegid()<<endl;//取得有效组 id，执行别人建的文件，结果是别人的 id, 要加头文件<unistd.h>。
编程习惯 4：unix 中，所有的结构的所有的成员，都是以结构名缩写加下划线开头。
编程习惯 5：unix 中，以_t 结尾的都是整数。
passwd* p=NULL;
p=getpwuid(uid);//获得指定用户的信息，用用户 id 查询。
string name;
p=getpwnam(name.c_str());//获得指定用户的信息，用用户名查询。
*(argv[1])和 argv[1][0]都表示 argv[1]中的第一个字符，可用来判断是数字还是字符。
char* getcwd(char* buf,size_t size)//获得当前目录，要加头文件<unistd.h>,第一个参数指明放目录名的字符串的地方，一般是一个字符数组名。第二个参数是空间大小，一般是数组长度。返回值是 buf，可以直接输出，cout<<getcwd(buf,256)<<endl;
unix 中凡是要自己准备字符数组来存储字符串的时候，总是要把字符数组的长度传递过去。
DIR* opendir(const char* dirname)//打开目录，返回目录的指针, 要包含头文件<dirent.h>和<sys/types.h>。//dir=argv[1];DIR* p=opendir(dir);
struct dirent* readdir(DIR* dirp)//读目录, 要包含头文件<dirent.h>和<sys/types.h>
其中有一个成员是 d_name，是数组名，是相对路径文件名，要使用必须变成 string 类型加到目录名之后。
dir=argv[1];
dirent* pd=NULL;//不用野指针。
while((pd=readdir(p))!=NULL){
string path=dir;
path+="/";
path+=pd->d_name;
ifstream fin(path.c_str());//unix 只支持 c 风格。
}
int closedir(DIR* dirp)//关闭目录，要包含头文件<dirent.h>和<sys/types.h>
ls ~ 显示主目录下文件。

```
1 #include<iostream>
  2 #include<dirent.h>
  3 #include<list>
  4 #include<string>
  5 #include<sys/types.h>
  6 #include<algorithm>
  8 using namespace std;
 10 int main(int argc,char* argv[])
 11 {
```

```
12      char* dir=NULL;
13      if(argc==1)
14          dir=".";
15      else
16          dir=argv[1];
17      DIR* p=opendir(dir);
18      if(p==NULL){
19          cout<<"not a directory"<<endl;
20          return -1;
21      }
22      dirent* pd=NULL;
23      list<string> ls;
24      while((pd=readdir(p))!=NULL){
25          //cout<<pd->d_name<<endl;
26          if(pd->d_name[0]=='.') continue;
27          ls.push_back(pd->d_name);
28      }
29      closedir(p);
30      ls.sort();//利用容器排序。
31      list<string>::iterator it;//定义迭代器。
32      it=ls.begin();
33      while(it!=ls.end())
34          cout<<*it++<<endl;
35      //copy(ls.begin(),ls.end(),ostream_iterator<string>(cout,"\n"));//高级输出法。
36 }
```

查询文件信息用，stat（argv[1],&s）//头文件：#include<sys/stat.h> #include<fcntl.h> #include<sys/types.h>;

```
 1 #include<iostream>
 2 using namespace std;
 3 #include<sys/stat.h>
 4 #include<fcntl.h>
 5 #include<sys/types.h>
 6
 7 int main(int argc,char* argv[])
 8 {
 9      if(argc==1){
10          cout<<*argv<<"filename"<<endl;
11          return 0;
12      }
13      int res;
14    struct      stat s;
15      res=stat(argv[1],&s);//   文件信息放在 s 结构中。
16      if(res<0){
17          cout<<"no this file"<<endl;
```

```
18              return -1;
19      }
20      cout<<s.st_uid<<", size: "<<s.st_size<<endl;//输出文件的 id 和大小。
21      if(S_ISDIR(s.st_mode))
22              cout<<"directory"<<endl;
23      else
24              cout<<"file"<<endl;
25 }
```

S_ISREG(s.st_mode)//测试是不是常规文件。S_ISCHR(s.st_mode)//测试是不是字符设备文件。S_ISBLK(s.st_mode)//测试是不是块设备文件。S_ISFIFO(s.st_mode)//测试是不是管道文件。

如果文件是符号连接文件，stat()读取的是源文件的属性，如果要读取连接文件自身的属性，则需要调用 lstat()函数。读取已打开的文件的属性用 fstat()函数。

改变文件的权限用 chmod(_const　char* _file, _mode_t　_mode)

chmod("test03",S_IRUSR|S_IWUSR|S_IWGRP)//S_IRUSR 拥有者可读 S_IWUSR 拥有者可写， S_IXUSR 拥有者可执行， S_IRWXU 拥有者可读可写可执行。S_IRGRP 同组用户可读， S_IWGRP 同组用户可写，S_IXGRP 同组用户可执行， S_IRWXG 同组用户可读可写可执行。S_IROTH 其他用户可读， S_IWOTH 其他用户可写， S_IXOTH 其他用户可执行， S_IRWXO 其他用户可读可写可执行。

改变符号连接文件本身的属性用 lchmod(_const　char* _file, _mode_t　_mode)

改变已经打开的文件的属性用 fchmod(_const　char* _file, _mode_t　_mode)

目录操作:创建目录用 mkdir("新目录名"，权限),删除目录用 rmdir(“目录名”)改变当前目录用 chdir(“目录名”)要包含头文件<sys/stat.h>和<unistd.h>

ls –l –d　a//查看 a 目录的权限，不加-d 查看目录内的权限。

nmask 命令可以屏蔽权限，0022 可以屏蔽其他用户的写权限。

rename("旧名"，“新名”)//给目录或文件改名。

```
1 #include<iostream>
2 using namespace std;
3 #include<sys/stat.h>
4 #include<unistd.h>
5 #include<string>
6
7 int main()
8 {
9       string cmd;
10      string dirname;
11      for(; ;){
12              cout<<"cmd:";
13              cin>>cmd;
14              if(cmd=="mkdir"){
15                      cin>>dirname;
16                      mkdir(dirname.c_str(),0777);
17              }
18              else if(cmd=="rmdir"){
19                      cin>>dirname;
```

```
20                  rmdir(dirname.c_str());
21              }
22              else if(cmd=="chdir"){
23                  cin>>dirname;
24                  chdir(dirname.c_str());
25              }
26              else if(cmd=="pwd"){
27                  char buf[256];
28                  getcwd(buf,256);
29                  cout<<buf<<endl;
30              }
31              else if(cmd=="bye"){
32                  cout<<"C U"<<endl;
33                  break;
34              }
35              else
36                  cout<<"unknown command "<<endl;
37      }
38 }
```

int access(“文件名”,int amode)//文件权限判别。amode：R_OK W_OK X_OK(是否可执行) F_OK（是否存在）如果有制定权限，返回 0，没有返回<0。

if(! access())//返回 0，要用!来使用。

打开文件用 int open(“文件名”，int oflag,…（前面用 O_CREAT 就要加上权限，0x644,0600）)//oflag 文件描述符：O_RDONLY O_WRONLY O_RDWR(三选一，必选且只能选一)O_APPEND (追加) O_CREAT （如果文件不存在就新建一个） O_EXCL（与 O_CREAT 用|合用表示必须创建新文件） O_TRUNC（清空原文件）//unix 中写入数据后，只覆盖原文件前面部分，不是直接清空。

关闭文件用 close(fd);//fd 是 open 的返回值。

```
1 #include<iostream>
2 using namespace std;
3 #include<fcntl.h>
5 int main()
6 {
7       char* name="/home/kobe/day17/tim.txt";
8       int fd=open(name,O_WRONLY|O_CREAT,0644);
9       if(fd<0){
10          cout<<"Error!"<<endl;
11      }
12      else {
13          cout<<"ok ,"<<fd<<endl;//fd 输出为 3。0,1,2 为 shell 占用。
14      }
15      close(fd);
16 }//fd 中 0 对应 cin,1 对应 cout,2 对应 cerr。
```

ulink(“文件名”)，remove(“文件名”)//删除文件。

size_t read(fd,内存地址，字节数);//返回值为读取到的字节数，不能直接输出读到的数据，因为不会自动加上'\0'。 size_t write(fd,内存地址，字节数)
如果读到的字节数小于要求的字节数，意味着读完了。
读和写是顺序的，如果读了前 4 个字节，再写就会从第 5 个字节往后写（day17 read_write.cc）。
off_t lseek(int fd,off_t offset（偏移量 ）,int whence)//whence 是参考位置，有三种：SEEK_SET(文件头，只能是正偏移) SEEK_CUR（当前位置，可以有正负偏移） SEEK_END（文件尾，只能是负偏移）

```
1 #include<iostream>
  2 using namespace std;
  3 #include<fcntl.h>
  4 #include<unistd.h>
  5
  6 int main()
  7 {
  8         int fd=open("rw.txt",O_RDWR);
  9         if(fd<0){
 10                 cout<<"rw.txt not exit"<<endl;
 11                 return -1;
 12         }
 13         char ch;
 14         for(int i=0;i<3;i++){
 15                 read(fd,&ch,1);
 16                 cout<<ch;
 17                 lseek(fd,1,SEEK_CUR);
 18         }
 19         cout<<endl;
 20         lseek(fd,0,SEEK_SET);
 21         char buf[9];
 22         read(fd,buf,8);
 23         buf[8]='\0';
 24         cout<<buf<<endl;
 25         lseek(fd,0,SEEK_END);
 26         write(fd,"bye\n",4);
 27         close(fd);
 28 }
```

获得主机名：int gethostname(地址，长度);//要加头文件<unistd.h>

```
  1 #include<iostream>
  2 using namespace std;
  3 #include<unistd.h>
  5 int main()
  6 {
  7         char name[256];
  8         int res=gethostname(name,256);
  9         if(res<0)
```

```
10              cout<<"error"<<endl;
11      else
12              cout<<name<<endl;
13 }
```

系统提供直接访问时间的函数只有 time 一个。

由 time_t 格式的时间转化为 struct tm 格式的时间用 gmtime()（格林尼治时间）和 localtime()（本地时间）两个函数来完成。由 tm 格式转化为 time_t 格式用 mktime()来完成。

tm 格式的时间成员有 tm_sec[0-60] tm_min[0-59] tm_hour[0-23] tm_mday(0-31 一月中的天数) tm_mon（月份数 0-11） tm_year (年数，从 1900 年开始) tm_wday （星期几，0-6，0 是星期日）tm_yday(一年中第几天，0-365)

localtime()返回值是 tm*，参数是 time_t 型地址。

```
1 #include<iostream>
2 using namespace std;
3 #include<time.h>
5 int main()
6 {
7       time_t t=time(NULL);
8       tm* p=localtime(&t);
9       cout<<p->tm_year+1900<<"year "<<p->tm_mon+1<<"month "<<p->tm_mday<<"day
week: "<<p->tm_wday<<" "<<p->tm_hour<<": "<<p->tm_min<<": "<<p->tm_sec<<endl;
10 }
```

strftime()用于转换成制定的格式，%Y 年 4 位 %H 小时 2 位 %M 分钟 2 位 %S 秒 2 位 %m 月份 2 位 %d 日 2 位。%F 表示 %Y -%m-%d。%T 表示 %H:%M:%S。输出时%Y 等会被替换，其他保留。

```
time_t t=time(NULL);
tm* p=localtime(&t);
char buf[100];
strftime(buf,100,"%F %w   %T",p);
cout<<buf<<endl;
```

进程：一份正在执行的程序。ps –ef 查看进程。ps –ef|wc 统计进程数量。

sleep(1);//定时 1 秒，unix 中的函数。

ps –u 252 查看 252 用户的所有进程。

每个进程有自己独立的 4G 地址空间，在不同控制台执行 a.out 变量的地址一样，但不是一个，其中一个改变不影响另一个。

登记退出处理函数：atexit(函数名)；程序退出时会调用函数，函数必须无参，返回值为 void，头文件为<stdlib.h>，调用顺序与登记顺序相反。

atexit(f1);atexit(f2);//先执行 f2 后执行 f1

非正常结束：exit(整数)。exit(0)相当于 return 0，当不会调用局部对象的析构函数。

exit()不会析构局部对象。

只有在 main 函数中 return 整数才是正常结束。

ps –l 查看进程状态。

vi 中行命令模式 :后输入！可以执行命令，vi 不会挂起。

system(命令)可以执行命令//头文件<stdlib.h>//system(“cd ”)无效，因为子进程 sh 的环境参数不会影响父进程。

```
 1 #include<iostream>
 2 using namespace std;
 3 #include<stdlib.h>
 4 #include<string>
 5
 6
 7 int main()
 8 {
 9         cout<<"wecomme "<<endl;
10         for(;;){
11                 string cmd;
12                 cout<<"Kobe > ";
13                 getline(cin,cmd);
14                 if(cmd=="bye") break;
15                 system(cmd.c_str());
16         }
17 }
```

ps –lp 2871//查询进程号为 2871 的进程。

init 进程的 id 是 1。unix 中有调度进程 schedule,进程号是 0，linux 中没有。

fork 是 unix c 的标志性函数，会把当前进程复制一份但进程 ID 不同，是多进程函数。

fork（头文件<unistd.h>）在父进程中的返回值是子进程 ID（大于 0），在子进程的返回值是 0，出错则返回<0。

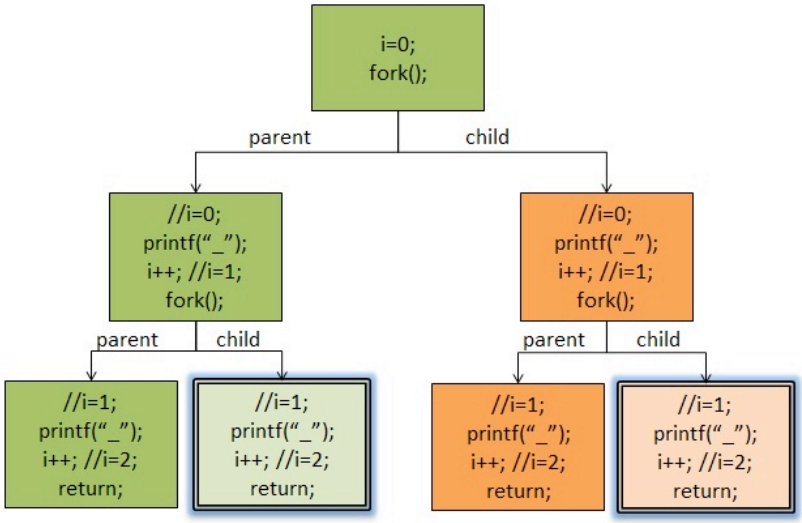
```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main(void)
{
int i;
for(i=0; i<2; i++){
    fork();
    printf("-");
}
return 0;
}
```

以上程序输出 8 个-。

- fork()系统调用是 Unix 下以自身进程创建子进程的系统调用，一次调用，两次返回，如果返回是 0，则是子进程，如果返回值>0，则是父进程（返回值是子进程的 pid），这是众为周知的。
- 还有一个很重要的东西是，在 fork()的调用处，整个父进程空间会原模原样地复制到子进程中，包括指令，变量值，程序调用栈，环境变量，缓冲区，等等。

所以，上面的那个程序为什么会输入 8 个“-”，这是因为 printf(“-”);语句有 buffer，所以，对于上述程序，printf(“-”);把“-”放到了缓存中，并没有真正的输出（参看《C 语言的迷题》中的第一题），**在 fork 的时候，缓存被复制到了子进程空间**，所以，就多了两个，就成了 8 个，而不是 6 个。

另外，多说一下，我们知道，Unix 下的设备有“块设备”和“字符设备”的概念，所谓块设备，就是以一块一块的数据存取的设备，字符设备是一次存取一个字符的设备。磁盘、内存都是块设备，字符设备如键盘和串口。**块设备一般都有缓存，而字符设备一般都没有缓存**。图中方框圈起来的两个进程复制了父进程的缓冲区。




getpid()//取得进程的 ID。getppid()//取得父进程号。

```
1 #include<iostream>
2 #include<sys/types.h>
3 #include<unistd.h>
4 using namespace std;
5
6 int main()
7 {
8       cout<<"old pid:"<<getpid()<<endl;
9       cout<<"try to use fork()..."<<endl;
10      pid_t id=fork();
11      if(id<0){
12            cout<<"fork error"<<endl;
13            return -1;
14      }
15      if(id>0){
16            for(int i=0;i<5;i++){
17                  cout<<"parent: "<<getpid()<<endl;
18                  sleep(3);
19            }
20      }
21      else {
22            for(int i=0;i<5;i++){
23                  cout<<"child : "<<getpid()<<endl;
24                  sleep(3);
25            }
```

```
 26         }
 27         cout<<"-----bye-----"<<endl;//会输出两次，父子进程都会执行。
 28 }
```

```
1 #include<iostream>
  2 #include<unistd.h>
  3 using namespace std;
  4
  5 int main()
  6 {
  7         cout<<"before fork()"<<endl;
  8         fork();
            //fork();//加上会变成四个进程。
  9         cout<<"after fork()"<<endl;//会输出两次，父子进程各一次。
 10 }
```

父进程结束的进程称为孤儿进程。unix 中子进程的资源必须由父进程回收，所以必须要有父进程。所以孤儿进程的父进程是一号进程 init。子进程先结束，父进程未结束，则子进程变为僵死进程，占用资源但不执行。

pid_t wait(NULL);//回收子进程资源，会等待子进程结束，返回子进程 id，头文件<sys/wait.h>。int s ;wait(&s);s 带回子进程结束状态。if(WIFEXITED(s))//判断是否正常结束。cout<<WEXITSTATUS(s);//取得返回值，

```
  1 #include<iostream>
  2 using namespace std;
  3 #include<unistd.h>
  4 #include<sys/wait.h>
  5 #include<sys/types.h>
  6
  7 int main()
  8 {
  9         if(fork()==0){
 10                 for(int i=0;i<5;i++){
 11                 cout<<"child :"<<getpid()<<endl;
 12                 sleep(1);
 13                 }
 14                 cout<<"------child end-----"<<endl;
 15                 return 123;
 16         }
 17         else{
 18                 cout<<"parent :"<<getpid()<<endl;
 19                 pid_t cid=wait(NULL);
 20                 cout<<"wait "<<cid<<endl;
 21                 cout<<"sleep 10 "<<endl;
 22                 sleep(10);
 23                 return 0;
 24         }
```

```
25 }
```
java 中没有多进程，只有线程。
exec 系列函数在进程空间中装入新程序来覆盖旧程序。新程序从头开始执行，execvp()。
execvp(“程序名”，argv )//argv 会传给 main 的 argv[]的第二个形参，头文件<stdlib.h>。
execvp()后的程序只有在失败时才会执行，因为会被 execvp()覆盖。
g++ cmdline –o cmdline//给 a.out 改名。

```
1 #include<iostream>
2 using namespace std;
3 #include<stdlib.h>
4 #include<unistd.h>
5
6 int main()
7 {
8       if(fork()==0){
9             char* argv[5]={
10                  "cmdline",
11                  "aa","bb","cc",
12                  NULL};
13            execvp("cmdline",argv);
              //execlp("cmdline","cmdline","aa","bb","cc",NULL);用 execlp 要将参数展开。
14            cout<<"Not found cmdline!"<<endl;//被覆盖不执行。
15            return -1;
16      }
17      else{
18            sleep(4);
19      }
20 }//只在子进程中调用 exec 系列函数。
```

execlp(“ls”,”ls”,”-l”,”/etc”,NULL);//用 execlp（）执行 ls –l /etc。
execlp 中的 p 代表会通过 PATH 环境变量寻找可执行文件。没有 p 表示在当前目录中找。
pkill 程序名//可以结束程序。system("pkill printa");pkill a 表示 kill 所有进程。
精灵进程(daemon) 进程间通信：1 信号机制，2FIFO（文件队列，命名管道），3 消息队列。

```
1 #include<iostream>
2 using namespace std;
3 #include<unistd.h>
4
5 int main()
6 {
7       if(fork()!=0){
8             cout<<"return to shell"<<endl;
              //write(1,"return to shell\n",16);//文件描述符 1 是输出，可以使用。
9             return 0;
10      }
11      for(;;){//运行时 shell 可以输入命令，父进程结束，但子进程没有结束，要用 pkill a.out
```
来结束。

```
12              cout<<'.'<<flush;
                //write(2,".",1);//文件描述符 2 是错误输出，可以使用。
13              sleep(3);
14      }
15 }
4 void child()
 5 {
 6      close(0);//关闭文件 012 后，子进程会在后台运行，叫精灵进程。
 7      close(1);
 8      close(2);
 9      for(;;){
10              write(2,".",1);
11              sleep(3);
12      }
13
14 }
15 int main()
16 {
17      cout<<"input a string:"<<flush;
18      char buf[100]={};
19      read(0,buf,5);
20      cout<<"buf ="<<buf<<endl;
21      if(fork()!=0){
22              write(1,"return to shell\n",16);
23              return 0;
24      }
25      else
26              child();
27 }
```

后台运行的进程叫精灵进程或守护进程。用 ps 看不到，要用 ps –u 用户 id。

精灵进程规范写法：1fork 后让父进程结束，变成孤儿进程。2 调用 setsid()新建一个会话，与原来的组和会话彻底脱离关系。3 设置新的权限屏蔽，umask(0077)//屏蔽其他人的一切权限。0022 只屏蔽别人的写权限。4 关闭所有文件描述符 close0-255。

精灵进程用于写后台服务程序。

makefile 文件描述依赖关系和编译命令（一定要以 tab 键开头）。

```
1 a.out: daemon.o make_daemon.o work.o//左边是目的文件，右边是依赖文件。
2       g++ daemon.o make_daemon.o work.o//一定要以 tab 键开头。
3 daemon.o: daemon.cc daemon.h
4       g++ -c    daemon.cc
5 make_daemon.o:make_daemon.cc daemon.h
6       g++ -c make_daemon.cc
7 work.o: work.cc
8       g++ -c work.cc
```

a. out : daemon.o make_daemon.o 其中:左边的叫目的文件，:右边的叫依赖文件。

建立 makefile 文件后，编译时输入 make 命令就可以完成编译，提高了编译效率，降低了编译难度。

man sleep 没有结果，可以用 man –a sleep。

重要信号（整型）头文件<signal.h>：掌握红色信号

SIGABRT 调用 abort 函数产生此信号。

SIGALRM 超时信号（由 alarm 设置时钟）

SIGCHLD 当子进程终止时调用此信号（向父进程发送，回收子进程资源）。

SIGINFO （ctrl+T）状态键产生此信号（用此信号终止失控程序）

SIGINT （按中断键 Delete or Ctrl+C 时产生）（用此信号终止失控程序）

SIGIO 此信号用于表示一个异步 IO 事件

SIGKILL 强制杀死进程信号（无法捕获）

SIGPIPE 当读进程终止，写入管道时会产生此信号

SIGQUIT 当用户在终端退出时，产生此信号

SIGSTOP 作业控制信号，停止一个进程，不能被捕捉或忽略

SIGSTP 挂起后台进程（Ctrl+Z），终端产生此进程

SIGUSR1 用户自定义

SIGUSR2 用户自定义

SIGTERM kill 进程时会收到信号。

man signal 可以查询信号。

用 旧函数名 signal(信号（int），函数名（函数指针）)登记信号处理函数，

kill -USR1 2074 向 2074 进程发送信号 SIGUSR1。

在有些系统中 signal()对某个符号的处理一次有效。

```
1 #include<iostream>
2 #include<signal.h>
3 using namespace std;
4 void func(int sig)
5 {
6       signal(sig,func);// 在有些系统中 signal()对某个符号的处理一次有效,在第一次调用时再登记一次。
7       if(sig==SIGINT)
8             cout<<"Ctrl+C"<<endl;
9       else if(sig==SIGUSR1)
10            cout<<"user signal1"<<endl;
11      else if(sig==SIGUSR2)
12            cout<<"user signal2"<<endl;
13      else
14            cout<<"unknown??"<<endl;
15 }
16
17 int main()
18 {
19      cout<<"==========1============"<<endl;
20      signal(SIGINT,func);
21      signal(SIGUSR1,func);
```

```
22        signal(SIGUSR2,func);
23        cout<<"======2==========="<<endl;
24        for(int i=0;i<200;i++){
25              sleep(10);
26              cerr<<'.';
27        }
28        cout<<"==================3============"<<endl;
29 }// 用 kill   -USR1   2074 向 2074 进程发送信号 SIGUSR1。
```

信号在有些系统中会取消正在进行的阻塞（等待）。阻塞中的函数调用会以出错方式返回。
signal(信号，SIG_IGN)忽略信号。signal(信号，SIG_DFL)对信号采取缺省处理。
signal()返回 SIG_ERR 表示登记失败。
用 kill 发信号仅限于给自己的进程发。kill(pid,信号)。
raise(信号)可以给自己发信号。
alarm(秒数)可以用来定时,时间一到会发一个信号。
产生随机数:srand(time(NULL))用头文件是<stdlib.h>，srand 的种子一样会产生相同的随机数序列，所以用时间做种子，使用时用 rand()就可以取得随机数。

```
 1 #include<iostream>
 2 using namespace std;
 3 #include<signal.h>
 4 #include<stdlib.h>
 5 #include<time.h>
 6
 7 int v=0;
 8 int x=0;
 9 void func(int sig)
10 {
11        cout<<"time out!"<<endl;
12        cout<<"V: "<<v<<endl;
13        cout<<"X: "<<x<<endl;
14        exit(0);
15 }
16 int main()
17 {
18        signal(SIGALRM,func);
19        srand(time(NULL));
20        alarm(20);
21        for(int i=0;i<10;i++){
22              int n1= rand()%10;
23              int n2= rand()%10;
24              cout<<n1<<'+'<<n2<<'=';
25              int sum;
26              cin>>sum;
27              if(sum==n1+n2)
28                    v++;
```

```
29              else
30                      x++;
31      }
32      cout<<"so quick"<<endl;
33      raise(SIGALRM);//给自己发信号。
34 }
```

```
1 #include<iostream>
 2 using namespace std;
 3 #include<signal.h>
 4 #include<unistd.h>
 5 #include<sys/wait.h>
 6 #include<sys/types.h>
 7 #include<stdlib.h>
 8
 9 void func(int sig)
10 {
11      signal(sig,func);
12      wait(NULL);
13      cout<<"waitde one child"<<endl;
14 }
15
16 void child(int len,char ch)
17 {
18      if(fork()!=0)
19              return;
20      for(int i=0;i<len;i++){
21              cerr<<ch; //用 cerr 是因为要马上显示。
22              sleep(3);
23      }
24      exit(0);//不加会出现 4 个进程。不能让第一个子进程返回。
25 }
26 int main()
27 {
28      signal(SIGCHLD,func);
29      child(10,'.');
30      child(20,'^');
31      for(int i=0;i<100;i++){
32              cerr<<'$';//用 cerr 是因为要马上显示。
33              sleep(1);
34      }
35 }
```

创建管道命令：mkfifo　a.f　创建管道函数是：mkfifo (文件名，权限) mkfifo(“a.fifo”,0600)

头文件是<sys/stat.h>

管道必须有读写双方才能打开。

```
#include<iostream>//写管道
using namespace std;
#include<sys/stat.h>
#include<string>
#include<sys/types.h>
#include<fcntl.h>
#include<unistd.h>
int main()
{
        mkfifo("a.fifo",0600);
        int fd=open("a.fifo",O_WRONLY);
        if(fd<0){
                cout<<"error!"<<endl;
                return -1;
        }
        cout<<"pipe ready!"<<endl;
        for(;;){
                cout<<"input text :";
                string str;
                getline(cin,str);
                write(fd,str.c_str(),str.size());
                if(str=="bye")
                        break;
        }
        close(fd);
}
```

```
#include<iostream>//读管道
using namespace std;
#include<sys/stat.h>
#include<string.h>
#include<sys/types.h>
#include<fcntl.h>
#include<unistd.h>
int main()
{
        mkfifo("a.fifo",0600);
        int fd=open("a.fifo",O_RDONLY);
        if(fd<0){
                cout<<"error!"<<endl;
                return -1;
        }
        cout<<"pipe ready!"<<endl;
        for(;;){
                char buf[1000];
```

```
20              int n;
21              n=read(fd,buf,1000);
22              buf[n]='\0';
23              cout<<"info :"<<buf<<endl;
24              if(strcmp(buf,"bye")==0)
25                      break;
26      }
27      close(fd);
28 }//在两个终端运行两个程序后，在写管道终端写入数据，在读管道终端会显示，数据量不能太大，读写双方关闭后，fifo 中的数据全部消失，向 fifo 中写的数据总是会追加在后面，不会覆盖。
```

不能将 fifo 移动，因为必须有读写双方才能有效。

消息队列在内存中传递数据。

ipcs 命令显示消息队列的状态。

ipcrm –q id 命令删除消息队列。

消息队列头文件：<sys/ipc.h>和<sys/msg.h>

创建消息队列：msgget (key,创建方式和权限) msgget(key,IPC_CREAT|0600)，返回值为 qid（整型），消息队列是一方发，另一方收，信息存在队列中。

消息队列中有不同的通道，每个消息队列有 2G 个通道。向消息队列中发送数据时一个结构，第一个成员是 long 型，表示通道号，要>0。后面可跟任意多个任意类型的成员。

发送数据：msgsnd(消息队列号，结构体地址，大小，0) msgsnd(qid,&m,sizeof(m),0)

接受数据：msgrcv(消息队列号，结构体地址，大小，通道号，0) msgsnd(qid,&m,sizeof(m),no,0)

接受数据时 0 号通道表示接受所有通道信息。

删除消息队列：msgctl(qid,IPC_RMID,NULL);

```
1 #include<iostream>
  2 using namespace std;
  3 #include<sys/ipc.h>
  4 #include<sys/msg.h>
  5 #include<stdlib.h>
  6
  7 int main(int argc,char* argv[])
  8 {
  9       if(argc==1){
 10              cout<<*argv<<"key"<<endl;
 11              return 0;
 12      }
 13      int key=atoi(argv[1]);
 14      int qid;
 15      qid=msgget(key,IPC_CREAT|0600);
 16      if(qid<0){
 17              cout<<"cannot creat "<<endl;
 18              return -1;
 19      }
 20      cout<<"OK!"<<endl;
```

```
 21         cout<<"key= 0x"<<hex<<key<<endl;
 22         cout<<"qid= "<<dec<<qid<<endl;
 23 }
1 #include<iostream>
  2 using namespace std;
  3 #include<sys/ipc.h>
  4 #include<sys/msg.h>
  5 #include<stdlib.h>
  6 struct Msg{
  7         long tongdao;
  8         char name[20];
  9         char classname[10];
 10         int age;
 11 };
 12
 13 int main(int argc,char* argv[])
 14 {
 15         if(argc==1){
 16                 cout<<*argv<<"key"<<endl;
 17                 return 0;
 18         }
 19         int key=atoi(argv[1]);
 20         int qid;
 21         qid=msgget(key,IPC_CREAT|0600);
 22         if(qid<0){
 23                 cout<<"cannot creat "<<endl;
 24                 return -1;
 25         }
 26         cout<<"OK!"<<endl;
 27         cout<<"key= 0x"<<hex<<key<<endl;
 28         cout<<"qid= "<<dec<<qid<<endl;
 29         Msg m;
 30         cout<<"input tong dao hao :"<<endl;
 31         cin>>m.tongdao>>m.name>>m.age;
 32         msgsnd(qid,&m,sizeof(m),0);
 33 }
  1 #include<iostream>
  2 using namespace std;
  3 #include<sys/ipc.h>
  4 #include<sys/msg.h>
  5 #include<stdlib.h>
  6 struct Msg{
  7         long tongdao;
  8         char name[20];
```

```
 9          char classname[10];
10          int age;
11 };
12
13 int main(int argc,char* argv[])
14 {
15          if(argc==1){
16                  cout<<*argv<<"key"<<endl;
17                  return 0;
18          }
19          int key=atoi(argv[1]);
20          int qid;
21          qid=msgget(key,IPC_CREAT|0600);
22          if(qid<0){
23                  cout<<"cannot creat "<<endl;
24                  return -1;
25          }
26          cout<<"OK!"<<endl;
27          cout<<"key= 0x"<<hex<<key<<endl;
28          cout<<"qid= "<<dec<<qid<<endl;
29          Msg m;
30          cout<<"input tong dao hao :"<<endl;
31          int no;
32          cin>>no;
33          msgrcv(qid,&m,sizeof(m),no,0);
34          cout<<m.tongdao<<endl;
35          cout<<m.name<<endl;
36          cout<<m.age<<endl;
37 }
1 #include<iostream>
 2 using namespace std;
 3 #include<sys/ipc.h>
 4 #include<sys/msg.h>
 5 #include<stdlib.h>
 6
 7 int main(int argc,char* argv[])
 8 {
 9          if(argc==1){
10                  cout<<*argv<<"key"<<endl;
11                  return 0;
12          }
13          int key=atoi(argv[1]);
14          int qid;
15          qid=msgget(key,IPC_CREAT|0600);
```

```
16        if(qid<0){
17              cout<<"cannot creat "<<endl;
18              return -1;
19        }
20        cout<<"OK!"<<endl;
21        cout<<"key= 0x"<<hex<<key<<endl;
22        cout<<"qid= "<<dec<<qid<<endl;
23        if(msgctl(qid,IPC_RMID,NULL)<0)
24              cout<<"error remove"<<endl;
25        else
26              cout<<"remove ok!"<<endl;
27 }
```

把当前目录设为主目录： 1 chdir(getenv(“HOME”)) 2 getpwuid(getuid()) 3getpwnam(getlogin())

linux 下 按 q 可以查看多个 man 帮助文档，unix 下用:n，在 man 中按 h 可以继续帮助。

动态链接库：windows 下是.dll linux 下是 lib***.so

编译动态链接库：g++ add.cc -shared –o libadd.so

```
1 #include<iostream>//add.cc
2 using namespace std;//不能有 main 函数。
3 int add(int a,int b,int c)
4 {
5       return (a+b-c);
6 }
```

编译 add.cc：g++ add.cc –share –o libadd.so

mv libadd.so so/lib 将连接移动到 lib 目录中。

```
1 #include<iostream>//testadd.cc
2 using namespace std;
3 #include"include/add.h"//链接库头文件
4
5 int main()
6 {
7       int r=add(123,456,78);
8       cout<<"r= "<<r<<endl;
9 }
```

编译 testadd.cc：g++ testadd.cc –l add -Llib

修改环境变量：export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:lib

执行 a.out。

调试程序：vi debug.cc g++ -g debug.cc //加-g 产生调试信息。

gdb a.out 进入 gdb 后，list(l) 15//列出以第 15 行为中心的十行代码。break(b) 6 在第六行设断点。info break 查看断点（有编号）。delete 断点编号//删除断点。运行用：run 运行后查看变量的值： print 变量名。print argc=20 修改变量 argc 的值。next 执行下一行。continue 一直继续运行。单步执行：step（遇到函数会进去，next 不会）finish 运行到函数结束。quit 退出调试。

网络：OSI(开放系统互连)7 层模型：物理层(Hub/Repeater) 数据链路层（一般有驱动程序组

成,Bridge/Switcher　IEEE802.x/DHCP ）网络层 (IP/ICMP/ARP　Router/Switcher)
传输层 （有端口，TCP/UDP）会话层（会话建立维护，虚的） 表示层（统一数据表示格式） 应用层（产生数据，gateway SNMP/telnet/ftp/HTTP/SMTP）
ifconfig　–a 查看 ip 地址。
网络层的 ip 地址能实现跨网段的传输。
系统给每个服务提供一个端口（port）。
TCP/IP 网络模型：物理层，网络接口层，互联网层，传输层，应用层。
IP 协议作用：寻址、路由、流量控制。是无连接的，用虚拟 ip 包。
IP 地址：D 类是多播地址，E 类保留，A 类 0 开头的，1~126,127.0.0.1 是自己，10 开头的是局域网。
B 类：1 0 开头的，128~191 。
C 类：1 1 0 开头的，192~223 192 一般用于局域网，有 253 台主机。
主机地址全为 0 的表示这个网络,128.211.0.0.，出现于路由表中。
主机地址全为 1 的是广播地址，128.211.255.255。
向自己所在的网络发广播用 255.255.255.255。
环回地址：127.0.0.1。
TCP transmission control protocal，面向连接，完全可靠，点对点，全双工，流接口，三次握手建立连接，。
c++网络应用模式是 C/S 模式（客户端服务器模式），java 网络应用模式是 B/S 模式（浏览器服务器模式）。
TCP 断开连接 4 次握手。
套接字：socket 是操作系统用来网络操作的接口。
套接地址：ip 种类，对方 ip，对方 port。
使用套接地址永远有：(sockaddr*)因为要实际使用的是 sockaddr_in 或 sockaddr_in6,要强制类型转换。

```
struct　in_addr{
    in_addr_t　s_addr;//32bits
};
struct　sockaddr_in{
    sa_family_t　sin_family;//AF_INET(ipv4),AF_INET6
    in_port_t　　sin_port; //16bits TCP/UDP　port number
    struct in_addr sin_addr;//32bits address
    char　　sin_zero[8];//没使用。
};
```

上传网络的数据要转换成网络字节顺序，转换函数有四个，头文件<netinet/in.h>。

```
uint32_t　htonl(uint32_t　hostlong)//从 host 到 net 转长整型
uint16_t　htons(uint16_t　hostshort) //从 host 到 net 转短整型

uint16_t　ntohs(uint16_t　netshort) //从 net 到 host 转短整型
uint32_t　ntohl(uint32_t　netlong) //从 net 到 host 转长整型
```

socket 编程头文件<sys/socket.h><netinet/in.h><arpa/inet.h>
避免使用 1024 以下的端口号。
建立套接字：int　ss=socket(协议家族名，协议，0)socket(AF_INET,SOCK_STREAM,0)
AF_INET 表示使用 ipv4，SOCK_STREAM 表示使用 TCP 协议。

绑定套接字：socklen　len=sizeof(si); int r=bind(ss,(sockaddr*) &si,len);必须用(sockaddr*)强转。
r=listen(ss,20);//建立 20 个分机。
ip 地址由点分十进制到内部格式的转换：inet_pton()
由内部格式向点分十进制的转换：inet_ntop(AF_INET,&c.sin_addr.s_sddr,ip,sizeof(ip))//ip 是自定义的 char 数组。
在 unix 系统中编译需要：g++　ipserver.cc　–lsocket　-lnsl。 linux 中不需要。

```
 1 #include<iostream>
 2 using namespace std;
 3 #include<sys/socket.h>
 4 #include<netinet/in.h>
 5 #include<arpa/inet.h>
 6 #include<unistd.h>
 7 #include<fcntl.h>
 8 #include<string>
 9 #include<stdlib.h>
10
11 int main(int argc,char* argv[])
12 {
13         short port=12345;
14         if(argc>1)
15                 port=atoi(argv[1]);
16         int ss=socket(AF_INET,SOCK_STREAM,0);// AF_INET ipv4    SOCK_STREAM TCP
17         if(ss<0)    return -1;
18         sockaddr_in si;//建立套接地址。
19         si.sin_family=AF_INET;//ip v4 格式地址。
20         si.sin_port=htons(port);//端口转成网络格式。
21         si.sin_addr.s_addr=0; //INADDR_ANY，非 0 ip 地址也要转。
22         socklen_t    len=sizeof(si);//取得套接地址长度。
23         int r=bind(ss,(sockaddr*)&si,len);//绑定套接字，相当于给电话分配号码。
24         if(r<0) return -1;
25         r=listen(ss,20);//转中继,把套接字变成被动套接字。相当于电话接分机。
26         if(r<0) return -1;
27         sockaddr_in c;//记录对方的套接地址。
28         for(;;){
29                 len=sizeof(c);//记录对方的套接地址的长度。
30                 int cs=accept(ss,(sockaddr*)&c,&len);//等待连接请求到来，到来时记录对方的套接地址，accept 返回的是可与对方通信的套接字，是新建的套接字。
31                 if(cs<0) continue;
32                 char ip[100];
33                 inet_ntop(AF_INET,&c.sin_addr.s_addr,ip,sizeof(ip));//将对方的 ip 地址由内部格式转成点分十进制格式（人习惯的格式）。
34                 cout<<ip<<"is coming "<<endl;
35                 string    msg="your ip:";
```

```
36              msg+=ip;
37              msg+='\n';
38              write(cs,msg.c_str(),msg.size());//将要发送的数据写到新建的套接字。
39              close(cs);//关闭套接字。
            }
    }
```

客户端程序：

```
 1 #include<iostream>
 2 using namespace std;
 3 #include<sys/socket.h>
 4 #include<netinet/in.h>
 5 #include<arpa/inet.h>
 6 #include<unistd.h>
 7 #include<fcntl.h>
 8 #include<stdlib.h>
 9
10 int main(int argc,char* argv[])
11 {
12      if(argc!=3){
13          cout<<*argv<<"ip port"<<endl;
14          return 0;
15      }
16      int s=socket(AF_INET,SOCK_STREAM,0);
17      if(s<0) return -1;
18      sockaddr_in    si;
19      short port=atoi(argv[2]);
20      char* ip=argv[1];
21      si.sin_family=AF_INET;
22      si.sin_port=htons(port);
23      inet_pton(AF_INET,ip,&si.sin_addr.s_addr);
24      int r=connect(s,(sockaddr*)&si,sizeof(si));//连接指定的 socket。
25      if(r<0) return -1;
26      char    msg[200];
27      int n=read(s,msg,sizeof(msg));
28      if(n<0) return -1;
29      msg[n]='\0';
30      cout<<msg<<endl;
31      close(s);
32 }
```

char msg[200]; strlen(msg),是数组中实际存储的字符的个数，sizeof(msg)一直是 200。

复杂客户端：

```
1 #include<iostream>
 2 using namespace std;
 3 #include<sys/socket.h>
```

```
 4 #include<netinet/in.h>
 5 #include<arpa/inet.h>
 6 #include<unistd.h>
 7 #include<fcntl.h>
 8 #include<stdlib.h>
 9 #include<cstring>
10
11 int main(int argc,char* argv[])
12 {
13         if(argc!=3){
14                 cout<<*argv<<"ip port"<<endl;
15                 return 0;
16         }
17         int s=socket(AF_INET,SOCK_STREAM,0);
18         if(s<0) return -1;
19         sockaddr_in    si;
20         short port=atoi(argv[2]);
21         char* ip=argv[1];
22         si.sin_family=AF_INET;
23         si.sin_port=htons(port);
24         inet_pton(AF_INET,ip,&si.sin_addr.s_addr);
25         int r=connect(s,(sockaddr*)&si,sizeof(si));
26         if(r<0) return -1;
27         char    msg[200];
28         int n=read(s,msg,sizeof(msg));
29         if(n<0) return -1;
30         msg[n]='\0';
31         cout<<msg<<endl;
32
33         for(;;){
34                 cout<<"msg:";
35                 cin.getline(msg,200);
36                 int n=write(s,msg,strlen(msg));//向服务器发数据。
37                 if(n<=0) break;
38                 if(msg[0]=='q') break;
39                 n=read(s,msg,sizeof(msg));//接收服务器发来的数据。
40                 msg[n]='\0';//read 不是字符串，不能直接输出。
41                 cout<<msg<<endl;
42                 if(n<=0) break;
43         }
44         close(s);
45 }
```

复杂服务器程序：

```
 1 #include<iostream>
```

```
 2 using namespace std;
 3 #include<sys/socket.h>
 4 #include<netinet/in.h>
 5 #include<arpa/inet.h>
 6 #include<unistd.h>
 7 #include<fcntl.h>
 8 #include<string>
 9 #include<stdlib.h>
10 #include<signal.h>
11 #include<sys/wait.h>
12
13 void func(int sig)
14 {
15         signal(sig,func);
16         wait(NULL);
17 }
18 int main(int argc,char* argv[])
19 {
20         signal(SIGCHLD,func);//防止出现僵死进程。
21         short port=12345;
22         if(argc>1)
23                 port=atoi(argv[1]);
24         int ss=socket(AF_INET,SOCK_STREAM,0);
25         if(ss<0)    return -1;
26         sockaddr_in si;
27         si.sin_family=AF_INET;
28         si.sin_port=htons(port);
29         si.sin_addr.s_addr=0; //INADDR_ANY
30         socklen_t    len=sizeof(si);
31         int r=bind(ss,(sockaddr*)&si,len);
32         if(r<0) return -1;
33         r=listen(ss,20);
34         if(r<0) return -1;
35         sockaddr_in c;
36         for(;;){
37                 len=sizeof(c);
38                 int cs=accept(ss,(sockaddr*)&c,&len);
39                if(cs<0) continue;
40                 if(fork()!=0) {
41                         close(cs);//父进程结束 cs，因为不需要。
42                         continue;
43                 }
44                 close(ss);//子进程结束 ss，因为不需要。
45                 char ip[100];
```

```
46              inet_ntop(AF_INET,&c.sin_addr.s_addr,ip,sizeof(ip));
47              cout<<ip<<"is coming "<<endl;
48              string    msg="your ip:";
49              msg+=ip;
50              msg+='\n';
51              write(cs,msg.c_str(),msg.size());
52              char buf[1000];
53              for(;;){
54                      int n=read(cs,buf,1000);
55                      if(n<=0) break;
56                      buf[n]='\0';
57                      msg=ip;
58                      msg+=':';
59                      msg+=buf;
60                      cout<<msg<<endl;//显示客户端的数据。
61                      msg="server: ";
62                      msg+=buf;
63                      write(cs,msg.c_str(),msg.size());//将客户端的数据发回去。
64                      if(buf[0]=='q') break;
65              }
66              close(cs);
67              cout<<ip<<"exited "<<endl;
68              exit(0);
69      }
70 }
```

shutdown()函数可以替代 close()终止网络连接，close()有两个限制：1 把描述符的引用计数减 1，仅在该计数变为 0 时才关闭套接字。shutdown 可以不管引用计数就激发 TCP 的正常连接终止序列。2close 终止读写两个方向的数据传送，有时候我们需要告知对端我们已经完成了数据的传送，即使对端仍有数据要发送给我们。

int shutdown(int sockfd,int howto)//howto 的值：SHUT_RD 关闭连接的读这一半，缓冲区中的数据全部丢弃。SHUT_WR 关闭连接的写这一半，缓冲区中的数据将被发送掉。SHUT_RDWR 连接的读半部和写半部都关闭。

发送大量的数据用 UDP 发送，QQ 用的是 UDP。

UDP 过程：1socket 2 bind 3 收数据 recvfrom 4 发数据 sendto 5 close()

TCP 和 UDP 的端口不冲突，可同时用一个。

int s=socket(AF_INET,SOCK_DGRAM,0);// SOCK_DGRAM 是 UDP 数据报。

recvfrom(s,msg,1000,0,(sockaddr*)&c,&len)//s 接受的套接字，msg 是放信息的数组，1000 是数组大小，第四个参数是 0，c 是对方的套接字地址，len 是对方套接字的长度。返回值是收到的字节数。

```
1 #include<iostream>
2 using namespace std;
3 #include<sys/socket.h>
4 #include<netinet/in.h>
5 #include<arpa/inet.h>
```

```
 6 #include<unistd.h>
 7 #include<stdlib.h>
 8 #include<string.h>
 9
10 int main(int argc,char* argv[])
11 {
12         short port=12345;
13         if(argc>1)
14                 port=atoi(argv[1]);
15         int s=socket(AF_INET,SOCK_DGRAM,0);
16         sockaddr_in si;
17         si.sin_family=AF_INET;
18         si.sin_addr.s_addr=0;
19         si.sin_port=htons(port);
20         socklen_t len=sizeof(si);
21         int r=bind(s,(sockaddr*)&si,len);
22         if(r<0) return -1;
23         char cmd;
24         char ip[100];
25         char msg[1000];
26         sockaddr_in c;
27         for(;;){
28                 cout<<"cmd: r/a/s/q "<<endl;
29                 cin>>cmd;
30                 switch(cmd){
31                 case 'r'://接收命令
32                         r=recvfrom(s,msg,1000,0,(sockaddr*)&c,&len);//接收函数
33                         msg[r]='\0';
34                         port=ntohs(c.sin_port);
35                         inet_ntop(AF_INET,&c.sin_addr.s_addr,ip,sizeof(ip));
36                         cout<<ip<<'@'<<port<<':'<<msg<<endl;
37                         break;
38                 case 'a':
39                         cin.getline(msg,1000);
40                         sendto(s,msg,strlen(msg),0,(sockaddr*)&c,len);//发送函数
41                         break;
42                 case 's':
43                         cin>>ip>>port;
44                         cin.getline(msg,1000);
45                         c.sin_family=AF_INET;
46                         c.sin_port=htons(port);
47                         inet_pton(AF_INET,ip,&c.sin_addr.s_addr);
48                         sendto(s,msg,strlen(msg),0,(sockaddr*)&c,len);
49                         break;
```

```
50              case 'q':
51                  exit(0);
52              default:
53                  cout<<"invalid cmd\n";
54                  cin.ignore(1000,'\n');
55              }
56      }
57 }
```

C 语言用调用栈来描述函数之间的调用关系。调用栈有栈帧组成，gcc ***.c -g
gdb a.out 进入 gdb 调试界面 bt 查看调用栈。 p a 打印变量 a 的值。
up 返回上一个栈帧。
段错误：段是指二进制文件内的区域。每次递归调用都会往调用栈里增加一个栈帧，栈帧多了就会发生栈溢出。在 Linux 系统中，用系统命令 ulimit 指定栈的大小
ulimit -s 32768 指定栈为 32M 。在 windows 栈的大小存储在可执行文件中，使用 gcc 可以指定大小，gcc -Wl,---stack=16777216 指定栈为 16M。
读入若干个单词或一句话可以用 while(cin.peek()!='\n') 然后用 cin.get(ch)来读入字符逐个处理，比如数字符，跳过空格等。
判断字符串的最小周期：

```
#include<stdio.h>
#include<string.h>
int main(){
    char word[100];
    scanf("%s",word);
    int len=strlen(word);
for(int i=1;i<=len;i++) if(len%i==0){
    int ok=1;
    for(int j=i;j<len;j++)
        if(word[j]!=word[j%i])  {ok=0;break;}
    if(ok) {printf("%d\n",i); break;}
}
return 0;
}
```

栈是由编译器在程序运行时分配的空间区域，有操作系统维护，里面的变量通常是局部变量、函数参数等。栈是向低地址扩展的数据结构，是一块连续的内存区域。能从栈中获得的空间较小。堆是向高地址扩展的数据结构，是不连续的内存区域，堆获得的空间较灵活，也较大。堆的效率低于栈。

程序中不能返回局部变量的地址，能返回值。

线程是进程中的一个动态对象，是一组独立的指令流，进程中的所有线程将共享进程中的资源，但是各个线程也拥有独立的程序计数器、堆栈和寄存器上下文。

可以使用 pstree 命令来查看系统中运行进程间的关系。

系统将进程的进程描述符通过双向循环链表组成一个队列，任务队列 tasks 即用于连接所有的进程。

进程 0 是所有进程的祖先，进程 1 由进程 0 创建，创建进程 1 之后，进程 0 会执行 cpu_idle 函数进入无限循环状态，因此进程 0 又被称之为 idle 进程，它是唯一一个不通过 fork 创建

的进程。进程 0 与进程 1 共享地址空间资源。

进程从用户态切换到内核态后，需要栈空间用于函数调用。因此内核为每个进程分配了一个固定大小的内核栈，用于保存进程在内核态中的函数调用链以及进程描述符。

shell 中用户进程的执行过程：1 用户键入命令。2 shell 在用户指定的路径中搜索该命令对应的可执行文件。3 shell 调用 fork 创建子进程。4 shell 的子进程调用 exec()装入该命令的可执行文件并执行。5 shell 等待命令结束，或者说子进程退出。可以通过 ctrl+Z 发送一个 SIGSTOP 信号给子进程，把子进程停止并放到后台，让 shell 重新运行。exec()会读取以一个外部程序来取代当前进程，所以不能在父进程里调用 exec()，否则父进程会结束。

Linux 内核采用了写时复制技术(copy-on-write)对 fork()创建进程的过程进行优化，此时子进程以只读的方式共享父进程的资源（父进程的页表还是被复制了），只有在子进程试图修改进程地址空间上的一页时，才进行该页的复制，这种技术可以避免进行大量数据的复制。

子进程拥有自己的数据段和堆栈段，它与父进程之间不共享任何的数据。

Linux 启动后，执行 ps –ef 后，以 d 结尾的进程就是内核进程，只能使用 3G 以上的空间，普通进程可以使用全部的 4G 空间。

进程的退出可以调用两个函数:exit(),exit_group()，exit()在内核中的服务例程是 sys_exit(), 然后 sys_exit()调用 do_exit()。

僵死进程的父进程如果结束，则系统会为他找一个新的父进程，新的父进程会清除僵死进程，如果父进程一直不结束，则子进程会一直保持僵死状态，即使以 root 用户使用 kill -9 也不能将其杀死。在父进程创建子进程时调用 signal(SIGCHLD,SIG_IGN)，将子进程的退出信号忽略，可以避免出现僵死进程。

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

void sort(string &str)
{
    int len=str.size();
    int i=0,j=0,k=0;
    char temp;
    for ( i=0;i<len-1;i++){
        k=i;
        for(j=i+1;j<len;j++ )
            if(str[j]<=str[k])
                k=j;
        temp=str[k];
        str[k]=str[i];
        str[i]=temp;

    }
}

int main()
{
    int count=0;
```

```
    int tag=0;
    string dic[2000],sorted[2000];
    string s="***";
    string word;
    while(cin.peek()!='\n'){
        cin>>dic[count];
        sorted[count]=dic[count];
        count++;
    }
    count=0;
    while(!(sorted[count]==s)){
        sort(sorted[count]);
        count++;

    }
    while(cin.peek()!='@'){
        cout<<"input the words:";
        cin>>word;
        tag=0;
        sort(word);
        while(!(sorted[tag]==s)){
            if(word==sorted[tag])
                break;
            tag++;
        }
        if(sorted[tag]==s)
            cout<<"No this word!"<<endl;
        else
            cout<<dic[tag]<<" "<<endl;
    }
    /*count=0;
    while(!(dic[count]==s)){

        cout<<dic[count]<<" ";
        count++;
    }*/
    cout<<endl;
    return 1;
}
```

内核中所有的线程共享 1G 的地址空间，而每个应用进程都是有独立私有的 3G 地址空间，互相之间不干扰。

USB 设备有四种传输方式:控制传输,中断传输,批量传输,等时传输.

Linux 系统为设备建立一个全局设备树来管理设备，可以实现动态电源管理和热插拔。

设备模型可以粗略的划分为两个层次：1 总线，设备，驱动 2kobject、kset、kobj_type

三个数据结构。kobject 是构成设备模型的核心结构，比如引用计数、设备上锁、设备名称、用来形成树状结构的链表指针等都放在 kobject 中。kobject 在 include/linux/kobject.h 中定义。

进程可以分为三类：交互式进程，批处理进程，实时进程。交互式进程比如文本编辑器，批处理进程比如编译器，实时进程比如视频播放器。

Nice 值是每个进程都会有的属性，ps –el 可以查看进程的优先级和 nice 值， NI 显示的是 nice 值，而 PRI 显示的是优先级。nice 值是-20~19 之间的值，默认取值是 0。较高的 nice 值使进程运行在较低的优先级上，较低的 nice 值使进程运行在较高的优先级上。普通用户可以修改进程的 nice 值比默认的值更高，只有超级用户才能修改进程的 nice 值比默认值更低，执行命令 nice –n 5 top 使 top 命令运行在更低的优先级上。

查看进程的内存地址空间：pmap 2268 //2268 是 bash 的进程号。第一列是内存区域起始地址，第二列为内存区域大小，第三列为属性，第四列为内存映射的文件。

Linux 使用 struct mm_struct 来描述一个进程的地址空间信息，定义在 include/linux/mm_types.h

进程的代码段、数据段等到内存的映射是通过系统调用 mmap()实现的，

void *mmap(void *addr ,size_t length, int prot ,int flags, int fd ,off_t offset);将文件从偏移 offset 的位置开始的长度为 length 的块映射到内存中，addr 是参考虚拟地址，一般为 NULL，这样内核会自动生成一个合理的虚拟地址。prot 是映射参数，可以是 PROT_EXEC （可执行）PROT_READ PROT_WRITE PROT_NONE 也可以是他们比特或的结果。fd 是文件的标志。

为了兼容各种构架的 CPU，Linux 采用四层页表来实现虚拟内存，页全局目录(PGD)，页上级目录(PUD)，页中间目录(PMD)，页表项(PTE)。在 x86 构架下只使用 PGD 和 PTE。他们各占 10bit，每级页表都是 1024 项。

虚拟内存分配后，并没有分配相应的物理页面，内核通过缺页错误处理在对虚拟内存地址进行一次访问时创建页表和分配物理页面。

内存工具：proc 接口查看内存信息：/proc/meminfo 查看系统整体内存的使用情况,在/home 下 vi /proc/meminfo。/proc/zoneinfo 查看内存各个 ZONE 的分布和使用情况。/proc/slabinfo 查看高速缓存 slab 的使用情况。/proc/vmstat 虚拟内存的实时统计值。/proc/swaps 查看交换分区的信息。/proc/sys/vm/*查看和配置虚拟内存管理的一些参数。/proc/<pid>/maps 查看进程的内存段映射信息。

ps vmstat 统计虚拟内存的使用情况。 pmap

虚拟文件系统 VFS，Linux 支持数十种文件系统，VFS 是在各种文件系统之间进行沟通交流的桥梁，它提供了一个通用的文件系统模型。VFS 使用面向对象的思路，主要有四种对象类型：超级块，索引节点，目录项，文件。超级块代表一个已经安装的文件系统，存储该文件系统的有关信息，比如文件系统的类型、大小、状态等。索引节点代表存储设备上的一个实际的物理文件，存储该文件的有关信息，比如访问权限、大小、创建时间等。目录项描述文件系统的层次结构，一个路径的各个组成部分，不管是目录还是普通文件都是一个目录对象。文件代表已经被进程打开的文件，主要用于建立进程和文件之间的对应关系。它由 open()系统调用创建，由 close()系统调用销毁。

ps 主要选项如下。

-a：显示系统中所有进程的信息。

-e：显示所有进程的信息。

-f：显示进程的所有信息。

-l：以长格式显示进程信息。

-r：只显示正在运行的进程。

-u：显示面向用户的格式（包括用户名、CPU 及内存使用情况等信息）。
-x：显示所有非控制终端上的进程信息。
-p：显示由进程 ID 指定的进程的信息。
-t：显示指定终端上的进程的信息。

**pstree 命令显示进程的树状关系。**

pstree 命令列出当前的进程，以及它们的树状结构。
格式：pstree [选项] [pid|user]
主要选项如下。
-a：显示执行程序的命令与完整参数。
-c：取消同名程序，合并显示。
-h：对输出结果进行处理，高亮显示正在执行的程序。
-l：长格式显示。
-n：以 PID 大小排序。
-p：显示 PID。
-u：显示 UID 信息。
-G：使用 VT100 终端编码显示。
-U：使用 UTF-8（Unicode）编码显示。

**top 命令显示进程**

top 命令用来显示系统当前的进程状况。
格式：top [选项]
主要选项如下。
d：指定更新的间隔，以秒计算。
q：没有任何延迟的更新。如果使用者有超级用户，则 top 命令将会以最高的优先序执行。
c：显示进程完整的路径与名称。
S：累积模式，会将已完成或消失的子进程的 CPU 时间累积起来。
s：安全模式。
i：不显示任何闲置（Idle）或无用（Zombie）的进程。
n：显示更新的次数，完成后将会退出 top。
说明：top 命令和 ps 命令的基本作用是相同的，都显示系统当前的进程状况。但是 top 是一个动态显示过程，即可以通过用户按键来不断刷新当前状态。
好了，我们回到我们的主题：堆和栈究竟有什么区别？

主要的区别由以下几点：

1、管理方式不同；
2、空间大小不同；
3、能否产生碎片不同；
4、生长方向不同；
5、分配方式不同；
6、分配效率不同；

管理方式：对于栈来讲，是由编译器自动管理，无需我们手工控制；对于堆来说，释放工作由程序员控制，容易产生 memory leak。

空间大小：一般来讲在 32 位系统下，堆内存可以达到 4G 的空间，从这个角度来看堆内存几乎是没有什么限制的。但是对于栈来讲，一般都是有一定的空间大小的，例如，在 VC6 下面，默认的栈空间大小是 1M（好像是，记不清楚了）。当然，我们可以修改：

打开工程，依次操作菜单如下：Project->Setting->Link，在 Category 中选中 Output，然

后在 Reserve 中设定堆栈的最大值和 commit。

注意：reserve 最小值为 4Byte；commit 是保留在虚拟内存的页文件里面，它设置的较大会使栈开辟较大的值，可能增加内存的开销和启动时间。

碎片问题：对于堆来讲，频繁的 new/delete 势必会造成内存空间的不连续，从而造成大量的碎片，使程序效率降低。对于栈来讲，则不会存在这个问题，因为栈是先进后出的队列，他们是如此的一一对应，以至于永远都不可能有一个内存块从栈中间弹出，在他弹出之前，在他上面的后进的栈内容已经被弹出，详细的可以参考数据结构，这里我们就不再一一讨论了。

生长方向：对于堆来讲，生长方向是向上的，也就是向着内存地址增加的方向；对于栈来讲，它的生长方向是向下的，是向着内存地址减小的方向增长。

分配方式：堆都是动态分配的，没有静态分配的堆。栈有 2 种分配方式：静态分配和动态分配。静态分配是编译器完成的，比如局部变量的分配。动态分配由 alloca 函数进行分配，但是栈的动态分配和堆是不同的，他的动态分配是由编译器进行释放，无需我们手工实现。

分配效率：栈是机器系统提供的数据结构，计算机会在底层对栈提供支持：分配专门的寄存器存放栈的地址，压栈出栈都有专门的指令执行，这就决定了栈的效率比较高。堆则是 C/C++函数库提供的，它的机制是很复杂的，例如为了分配一块内存，库函数会按照一定的算法（具体的算法可以参考数据结构/操作系统）在堆内存中搜索可用的足够大小的空间，如果没有足够大小的空间（可能是由于内存碎片太多），就有可能调用系统功能去增加程序数据段的内存空间，这样就有机会分到足够大小的内存，然后进行返回。显然，堆的效率比栈要低得多。

从这里我们可以看到，堆和栈相比，由于大量 new/delete 的使用，容易造成大量的内存碎片；由于没有专门的系统支持，效率很低；由于可能引发用户态和核心态的切换，内存的申请，代价变得更加昂贵。所以栈在程序中是应用最广泛的，就算是函数的调用也利用栈去完成，函数调用过程中的参数，返回地址，EBP 和局部变量都采用栈的方式存放。所以，我们推荐大家尽量用栈，而不是用堆。

虽然栈有如此众多的好处，但是由于和堆相比不是那么灵活，有时候分配大量的内存空间，还是用堆好一些。

无论是堆还是栈，都要防止越界现象的发生（除非你是故意使其越界），因为越界的结果要么是程序崩溃，要么是摧毁程序的堆、栈结构，产生以想不到的结果,就算是在你的程序运行过程中，没有发生上面的问题，你还是要小心，说不定什么时候就崩掉，那时候 debug 可是相当困难的：）