## memset：

void \*memset(void \*s, int ch, [size\_t](http://baike.baidu.com/view/3236587.htm" \t "_blank) n);

函数解释：将s中当前位置后面的n个字节 （typedef unsigned int size\_t ）用 ch 替换并返回 s 。

Memset的作用：是在一段内存块中填充某个给定的值，它是对较大的[结构体](http://baike.baidu.com/view/204974.htm" \t "_blank)或[数组](http://baike.baidu.com/view/209670.htm" \t "_blank)进行清零操作的一种最快方法[1]

## 二、fill：

原理是把那一块单元赋成指定的值

 int  d[100];  
 fill(d,d+100,1);

结果：

1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

初始化数组一般用fill

## 三、lower\_bound 和 upper\_bound:

ForwardIter lower\_bound(ForwardIter first, ForwardIter last,const \_Tp& val)

利用二分查找算法返回一个非递减序列[first, last)中的第一个大于等于值val的位置。

ForwardIter upper\_bound(ForwardIter first, ForwardIter last, const \_Tp& val) 利用二分查找算法返回一个非递减序列[first, last)中第一个大于val的位置。

## 四、Priority\_queue:

### 1、使用：

导入#include <queue> ， priority\_queue<T> queue;

// top得到的是最大值

// 需要获得最大值的时候可以使用

### 2、原理：

（1） priority\_queue 对于基本类型的使用方法相对简单。他的模板声明带有三个参数:  
priority\_queue<Type, Container, Functional>  
其中Type 为数据类型， Container 为保存数据的容器，Functional 为元素比较方式。  
Container 必须是用数组实现的容器，比如 vector, deque 但不能用 list.  
STL里面默认用的是 vector. 比较方式默认用 operator< , 所以如果你把后面俩个参数缺省的话，  
优先队列就是大顶堆，队头元素最大。

(2) 如果要用到小顶堆，则一般要把模板的三个参数都带进去。  
STL里面定义了一个仿函数 greater<>，对于基本类型可以用这个仿函数声明小顶堆

priority\_queue<int, vector<int>, greater<int> > q;

// VS2013无法使用最小堆

(3) 对于自定义类型，则必须自己重载 operator< 或者自己写仿函数

实例：

假设有自定义类型：

struct Node

{

int x;

int y;

Node(int a, int b) : x(a), y(b) {

};

};

需要定义比较函数(在一个结构体中书写)：

struct cmp

{

bool operator() (Node a, Node b) {

if (a.x == b.x)

{

return (a.y > b.y);

}

return a.x > b.x;

}

};

创建优先队列：

priority\_queue<Node, vector<Node>, cmp> q;

queue.push(Node(rand(), rand()));

(4)内部使用二叉堆实现。

## 五、随机数：

1、添加头文件：#include <stdlib.h>

2、rand() 取随机数

3、srand(time(NULL)); 设置每次选取的随机数都不一样

## 六、Set和map：

1、两种类型的数据结构都是二叉搜索数实现，可以用这两个数据结构，而不用自己写二叉搜索数。

2、set使用：

#include <iostream>

#include <set>

using namespace std;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* set的使用 \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void userSet() {

set<int> s;

// 插入元素

s.insert(1);

s.insert(3);

s.insert(5);

// 查找元素(通过迭代器来查找元素)

set<int>::iterator ite;

ite = s.find(1);

if (ite == s.end()) // find() ，返回给定值值得定位器，如果没找到则返回end()。

{

cout << "NOT FOUND" << endl;

}

else

{

cout << "FOUND" << endl;

}

// 查找元素是否存在

if (s.count(4) == 0) // 返回元素在集合中出现的次数

{

cout << "NOT FOUND" << endl;

}

else

{

cout << "FOUND" << endl;

}

// 删除

s.erase(3);

// 遍历set集合

for (ite = s.begin(); ite != s.end(); ite++)

{

cout << \*ite << " ";

}

}

int main() {

userSet();

system("pause");

return 0;

}

3、map的使用：

#include <iostream>

#include <set>

#include <map>

#include <string>

using namespace std;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* map的使用 \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void userMap(){

map<int, string> myMap;

myMap[0] = "哈哈";

myMap[1] = "傻逼";

map<int, string>::iterator ite;

for (ite = myMap.begin(); ite != myMap.end(); ite++)

{

cout << ite->first<<" ";

cout << ite->second << endl;

}

// 删除(删除键对应的键值对)

myMap.erase(0);

}

## 七、Queue:

queue<int>q1;

queue<double>q2;

queue的基本操作有：

1.入队：如q.push(x):将x元素接到队列的末端；

2.出队：如q.pop() 弹出队列的第一个元素，并不会返回元素的值；

3,访问队首元素：如q.front()

4,访问队尾元素，如q.back();

5,访问队中的元素个数，如q.size();

## 八、delete：

1、void DestroyList(List \*L) { delete L;} 释放指针L指向的内存空间

## 九、排序sort（）：

sort(arr, arr+10); // 导入头文件 #include<algorithm> 这里扫描10个元素。

## 九、位运算符:

### 1、左移、右移：

（1）规则：

左移：按二进制形式把所有的数字向左移动对应的位数，高位移出(舍弃)，低位的空位补零

右移：按二进制形式把所有的数字向右移动对应的位数，低位移出(舍弃)，高位的空位补符号位，即正数补0，负数补1。

(1)左移n位，原本的数乘以2的n次方。

(2)右移n位，原本的数除以2的n次方。

### 2、按位与运算 &

2.1、规则：

两位同时为“1”，结果才为“1”，否则为0

2.2、使用：

(1) 对某些位清0

(2) 保留某些位

### 3、按位或运算 |

3.1、规则：

(1) 有一个为1，结果就为1.

3.2、使用：

### 4、按位异或运算 ^

两个二进制位相同为0，不同为1。

3.1、规则:

(1) 0 ^ 任何数 = 任何数

(2) 1 ^ 任何数 = 任何数取反

(3) 任何数 ^ 任何数 = 0

3.2、使用:

(1) 可以用来使特定的位翻转。比如二进制位中把0和1异或，就能得到1

(2) 实现两个值的交换： a = a ^ b;

b = a ^ b;

a = a ^ b;

### 5、~ NOT 对1取补（位反转）:

## 十、string

1、string类型不可以赋值为nullptr（只有string类型的指针可以）

## 十一、static：

1. 隐藏: 所有未加static前缀的全局变量和函数都具有全局可见性，其它的源文件也能访问（比如通过extern访问）。如果加了static，就会对其它源文件隐藏（无法通过extern访问），同时仅在变量的作用范围内可见。利用这一特性可以在不同的文件中定义同名函数和同名变量，而不必担心命名冲突。
2. Static声明的字段存储在静态数据区，存储在静态数据区的变量在程序刚开始运行的时候，就完成初始化。也是唯一的一次初始化。存储在静态数据区的变量默认初始化为0（0x00）。下一次调用时还可以保存原来的赋值。
3. 函数前加static使得函数成为静态函数。但此处“static”的含义不是指存储方式，而是指对函数的作用域仅局限于本文件(所以又称内部函数)。使用内部函数的好处是：不同的人编写不同的函数时，不用担心自己定义的函数，是否会与其它文件中的函数同名。
4. static函数和普通函数作用域不同，static函数作用域仅在本文件。只在当前源文件中使用的函数应该说明为内部函数（static），内部函数应该在当前源文件中声明和定义。
5. static全局变量与全局变量有什么区别？ Static全局变量只初始化一次，防止在其他单元文件中被引用。防止命名冲突。
6. static局部变量和普通局部变量有什么区别？ Static局部变量只初始化一次。静态局部变量也在全局数据区（静态数据区），定义静态局部变量的函数执行完毕以后，该变量还在，再次调用该函数，静态局部变量的值会以上次函数结束时的值为准。
7. static函数和普通函数的区别？ Static函数在内存中只含有一份，普通函数在每个被调用中维持一份拷贝。
8. 全局变量和静态变量如果没有手工初始化，则由编译器初始化为0，局部变量的值不可知。

## 十二、存储区：

1. 常量存储区: 里面存放的是常量，不允许修改。程序结束后，系统回收内存
2. 全局/静态存储区: 全局变量和静态变量被分配到同一块内存中。存储在静态数据区的变量有静态变量和全局变量。
3. 堆区：由new分配的内存块。这些内存块的释放编译器不去管，由我们的应用程序去控制，一般一个new就要对应一个delete。如果程序员没有释放掉，那么程序结束以后，操作系统会自动回收。

详细：堆的数据结构不是由系统支持的，而是由函数库提供的。基本malloc/realloc/free函数维护了一套内部的堆数据结构。当程序使用这些函数去获取新的内存空间的时候，这套函数首先会到内部堆中寻找可用的内存空间，如果没有可用的内存空间，则试图利通系统调用来动态增加程序数据段的内存大小，新分配得到的内存空间首先被组织进内部堆中去，然后再以适当的形式返回给调用者。当程序释放分配的内存空间时，这片空间被返回内部堆结构中，可能会被适当的处理，比如和其他空间空间合并成更大的空闲空间，以更适合下一次内存分配申请。这套复杂的机制实际上相当于一个内存分配的缓存池。这套机制的原因：

1. 系统调用申请内存可能不支持任意大小的内存分配。有些系统的系统调用只支持固定大小及其倍数的内存请求(按页分配)；这样的话，对于大量的小内存分配来说会造成浪费。
2. 系统调用申请内存可能是代价昂贵的。系统调用可能涉及用户态和核心态的转化。
3. 没有管理的内存分配在大量复杂内存分配释放操作下很容易造成内存碎片。
4. 栈区：由编译器自动分配释放内存，存放函数内部的局部变量， 函数参数等。

详细：现在计算机（串行执行机制），都直接在代码底层支持栈的数据结构。这体现在，有专门的寄存器存放栈所在的地址，有专门的机器指令完成数据入栈出栈操作。这种机制的特点是效率高，支持的数据有限，一般是整数、指针、浮点数等系统支持的数据类型，并不直接支持其他的数据结构。对子程序的调用就是直接利用栈完成的。机器的call指令里隐含了把返回地址推入栈，然后跳转到子程序地址的操作，而子程序中的ret指令则隐含从栈中弹出返回地址并跳转之的操作。

栈和堆的对比：

1. 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，从而造成大量碎片，是程序效率降低。对于栈来讲，则不会存在这个问题，因为栈是先进后出的数据结构。
2. 生长方向：对于堆，生长方向是向上的，向着内存地址增加的方法；对于栈来讲，生长方向是向下的，向着内存地址向下的方向增长。
3. 分配方式：堆都是动态分配的，没有静态分配的堆。栈有2中分配方式: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，当时栈的动态分配和堆是不同的，栈的动态分配的内存是由编译器进行释放，无需我们手工实现。
4. 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支持：分配专门的寄存器存放栈的地址，压栈和出栈都有专门的机器指令，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，比如为了分配一块内存，库函数会按照一定的算法，在堆内存中寻找可用的足够大小的空间，如果没有足够大小的空间(可能由于内存碎片太多)，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的内存，然后进行返回。显然，堆的效率要比栈低得多。
5. 管理方式：对于栈来讲，是由编译器自动管理的，无需我们手工控制，对于堆来讲，释放工作程序员控制，容易产生内存泄漏。
6. 空间大小：栈一般都是有一定的空间大小。堆内存没有什么限制。

// 代码实例：

Char s1[] = “aaaaaaa”; // 在栈上的数组，里面存放aaaaaa字符串

Char\* s2 = “bbbb”; // 在栈内存中存放了一个指针s2，s2指向一块堆内存，堆内存存放”bbbb”;

aaaaaaa是在运行时赋值的。

bbbb是在编译时确定的。

但是在以后的存取中，在栈上面的数组比指针所指向的字符串快。

1. 自由存储区：由malloc分配的内存块，用free结束自己的生命。

### 静态内存分配及动态内存分配：

**静态与动态内存分配的两个主要区别:**

**（1）静态对象是有名字的变量,我们直接对其进行操作，而动态对象是没有名字的变量，我们通过指针间接地对它进行操作。**

**（2）静态对象的分配与释放由编译器自动处理。程序员需要理解这一点，但不需要做任何事情。相反，动态对象的分配与释放，必须由程序员显式地管理，相对来说比较容易出错，它通过new 和delete 两个表达式来完成。**

## 十三、全局变量、静态全局变量、静态局部变量和局部变量的区别

1. 按存储区域划分：全局变量，静态全局变量，静态局部变量都存放在内存的静态存储区域，局部变量存储在内存的栈区。

2、 自动变量

函数中的局部变量，如果不用关键字static加以声明，编译系统对它们是动态地分配存储空间的。函数的形参和在函数中定义的变量(包括在复合语句中定义的变量)都属此类。在调用该函数时，系统给形参和函数中定义的变量分配存储空间，数据存储在动态存储区中。在函数调用结束时就自动释放这些空间。如果是在复合语句中定义的变量，则在变量定义时分配存储空间，在复合语句结束时自动释放空间。因此这类局部变量称为自动变量(auto variable)。自动变量用关键字auto作存储类别的声明。例如：  
int f(int a) //定义f函数，a为形参  
{  
    auto int b, c=3; //定义b和c为整型的自动变量  
}  
  
存储类别auto和数据类型int的顺序任意。关键字auto可以省略，如果不写auto，则系统把它默认为自动存储类别，它属于动态存储方式。程序中大多数变量属于自动变量。本教程前面各章所介绍的例子中，在函数中定义的变量都没有声明为auto，其实都默认指定为自动变量。在函数体中以下两种写法作用相同：  
auto int b, c=3;  
int b, c=3;

## 十四、寄存器变量:

1. register声明的变量称为寄存器变量。
2. 为了提高执行效率，C++运行将局部变量的值存放在CPU的寄存器中，需要时直接从寄存器中取出参与运算，不必再到内存中去存取。

## 十五、extern声明外部变量:

有时需要用extern来声明全局变量，以扩展全局变量的作用域。

1. 提前引用声明：如果外部变量不在文件的开头定义，其有效的作用范围只限于定义处到文件末尾。如果在定义点之前的函数想引用该全局变量，则应该在引用之前用关键字extern对该变量作外部变量声明，表示该变量是一个将在下面定义的全局变量。有了此声明，就可以从声明处起，合法地引用该全局变量，这种声明称为提前引用声明。
2. **带extern的变量说明是变量声明，不是变量定义。不会为变量分配空间**
3. **函数的定义和声明，默认是所有文件可见(自动extern)，在其前面加上static，则函数只能用于这个文件。**

**inline函数默认为static存储类型。**

**效果：**

**它允许其他源文件建立并使用同名函数，而不互相冲突。**

**声明为静态函数不能被其他源文件所调用，因为他的名字不能得到**

4、

## 十六、new、delete、malloc、free：

1. new和delete配对，释放数组需要用到delete[]。New和delete实际上调用了malloc

和free。另外调用了类的构造函数或析构函数。

1. malloc和free配对，malloc返回的是void指针，需要强转。Malloc不会调用类的构造函数，free不会调用类的析构函数。
2. new申请的内存保存在堆中，malloc申请的内存保存在自由存储区。
3. new、delete是操作符，malloc、free是函数其在头文件malloc.h中声明。

## 十七、引用和指针的区别：

1. 指针概念：指针是一个变量，只不过这个变量存储的是一个地址，指向内存的一个存储单元；
2. 引用概念: 引用跟原来的变量实质上是同一个东西（共用同一块内存），只不过是原变量的一个别名而已。
3. const int& a = 8;

如果T类型引用的初始化值不是一个左值，将建立一个T类型的目标并初始化，那个目标的地址变为引用的值

1. 引用作函数的形参，实际上是在被调函数中对实参变量进行操作
2. 函数返回值时，要生成一个值的副本。而用引用返回值时，不生成值的副本**。**

同时可以作为左值。 int& func(int &i) {return i;} int i=10; func(i) = 5;

（4）引用在定义的时候必须要初始化

1. 区别：
2. 指针可以有多级，引用只能是一级
3. 指针可以为空，引用不可以为空，引用在定义的时候必须初始化。
4. 指针的值在初始化以后可以改变，即指向其它存储单元，而引用在进行初始化后就不会再改变。
5. 用引用进行函数参数的传递时，传递的是实参本身，引用对形参的修改就是对实参的修改。

## 十八、const：

1. （常量指针）下面两个声明都指向一个const int类型的指针，指针所指向的内存不能被修改，该指针可以指向另一个内存: (const在\*前面，限定指针指向的内容不可以变化）

const int \*p;

int const\* q;

1. (指针常量)指针指向的内容可以被修改，但指针不能指向另外一个地方：(const在\*后面，限定指针这个变量保存的地址不能变化)

Int\* const p;

1. 指针指向的内容和指针保存的地址都不能改变：

const int \* const p;

1. const在函数声明中的含义：

const int& SetPoint(const int& param) const  
第一个const:  
函数的返回值限定为const,即返回值不能被修改。const int a=SetPoint(...)  a在此之后便不能被修改。  
第二个const:  
指函数的形参为const类型，函数体内不能被修改.  
第三个const:  
表明这个函数不会对这个类对象的成员变量（准确地说是非静态数据成员）作任何改变。

1. 对于static成员变量，如果同时是const的，可以在类定义中初始化，否则只能在类定义外部初始化。

**二十九处还有！！！**

## 十九、union：

1. 概念: “联合”是一种特殊的类，也是一种构造类型的数据结构。在一个“联合”内可以定义多种不同的数据类型， 一个被说明为该“联合”类型的变量中，允许装入该“联合”所定义的任何一种数据，这些数据共享同一段内存，已达到节省空间的目的（还有一个节省空间的类型：位域）。 这是一个非常特殊的地方，也是联合的特征。另外，同struct一样，联合默认访问权限也是公有的，并且，也具有成员函数。一个联合变量的长度等于各成员中最长的长度。

## 二十、指针：

1、int a = 3; int \*p = &a; \*p表示取出p所指向内存的内容。这里是3。

1. int a[3][4];

int (\*p)[4];

p = a;

1. a是一个数组名，类型是指向一维数组的指针，不是指针变量，a的值是指针常量，即不能有a++或者a=p这些操作。a指向指向这块连续空间的首地址，值是&a[0][0]。
2. a[0]是一维数组名，类型是指向整型的指针，值是&a[0][0]，这个值是一个常量。
3. a[1]是一维数组名，类型是指向整型的指针，值是&a[1][0]，这个值是一个常量。
4. p是一个一维指针变量，指向一维数组的指针变量，值是&a[0][0]，可以执行p++，a=p这些操作。
5. 对于二维数组来说，a+1表示指向下一行元素，也可以理解为指向下一个一维数组。
6. \*(a + 1)取出第一行的首地址（从0开始） \*(a+1)相当于a[1], 前面说a是指向一维数组的指针，一维数组本身就是一个指针指向这一行的第一个元素，因此，a相当于一个二级指针。\*(a + 1)就相当与取出指针a所指向内存的所存储的内容，a+1指向的内存就是a[1]这个一级指针所在的内存，这块内存所存储的内容就是&a[1][0]
7. a[0] + 1是指向第0行第1个元素，也可以理解为指向一维数组a[0]的第一个元素。
8. \*(p + 1) 同 \*(a + 1)
9. 虽然a跟a[0]值是一样的，但是类型不一样，表示的意义不一样。
10. 指针是数组的迭代器：二维数组其实是特殊的一维数组
11. 数组名和数组指针变量的区别？

数组名是指针，类型是指向元素类型的指针，但是值是指针常量。声明数组时编译器会为声明所指定的元素数量保留内存空间，数组指针是指向数组的指针，声明指针变量时编译器只为指针本身保留内存空间。

1. **数组名的类型是指向元素类型的指针，值是指针常量。(a+1)的类型是一个指针变量。把数组名作为参数传递的时候实际上传递的是一个指针变量。sizeof对变量和数组名操作时返回的结果会不一样。数组指针是指向数组的指针，其值可以是变量。**
2. 输出sizeof（p）的时候，要判断p是数组还是变量，如果是变量（比如指针），输出变量本身占用的内存，如果是数组名，输出整个数组的占用空间。
3. 函数指针：
4. 函数指针一般形式：

void(\*p)(int a) = fun;

(2） 用typedef可以重定义函数指针：

typedef void(\*FUNC)(int i);

(3） 指向成员函数的函数指针：

假设有类Person，有成员函数void inti(int i);

有函数指针：typedef void(Person::\*FUNC)(int i);

FUNC f;

则：f = &Person::init;

Person\* p = new Person();

(p->\*f)(10); // 取出init这个函数的地址，p指针调用。

1. 常量指针：指针指向的对象是常量。const int \*p;
2. 指针常量：指针指向的地址不能改变，必须定义时给指针赋值

10、指针函数：返回类型为指针的函数。指针函数不能把在它内部说明的具有局部作用域的数据地址为返回值。因为函数返回后，函数内定义的局部变量所占空间将会释放，并在程序的运行过程中进行再分配。

可以返回堆空间的地址、全局变量或静态变量的地址。

11、void指针：

（1）void指针是空类型指针，只是存放地址，不能提供任何类型信息。

（2）不能进行指针运算(无法计算出移动步长)，无法进行间接引用。

（3）将任何类型的指针赋值给void指针都是合法的。

12、null：给指针赋值为null后，

系统保证指针不指向任何地方；表示指针变量的值没有意义

**用途:**

**避免指针变量的非法引用**

**在程序中常作为状态比较**

13、  
// 输出为123

cout << p << endl;

// 输出为1

cout << \*p << endl;

二十、数据类型分类：

按照取值的不同：

1. 原子类型：是不可以再分解的基本类型，包括整型、实型、字符型
2. 结构类型：由若干个类型组成，是可以再分解的。

## 二十一、字符串

1. 常用字符串函数（函数返回值都是目标串）：z
2. char\* strcat(char\* \_Dest, char\* \_Source); //把源串（Source）拼接到目标串（Dest）上
3. char\* strcpy(char\* \_Dest, char\* \_Source); //把源串（Source）复制到目标串（Dest）上
4. char\* strncpy(char\* \_Dest, char\* \_Source, int count); //把源串（Source）的count个字符复制到目标串（Dest）对应位置上
5. int strcmp(char\* one, char\* two); //①字符串1小于字符串2,strcmp函数返回一个负值;

②字符串1等于字符串2,strcmp函数返回零;

③字符串1大于字符串2,strcmp函数返回一个正值;

## 二十一、函数：

1. 重载函数：在同一个作用域内，一组拥有相同函数名，不同参数列表的函数。 不能仅以返回类型来区分重载函数。
2. inline函数：
3. 函数调用需要时间和空间的开销，调用函数实际上将程序的执行流程转移到被调用函数中，被调函数的代码执行完毕以后，再返回到调用的地方。这种操作要求调用前保护现场并记忆执行的地址，返回后恢复现场，并按原来保存的地址继续执行。对于较长的函数，这种开销可以忽略不计，但对于一些函数代码很短，又被频繁的调用的函数，就不能忽略这种开销。内联函数就是为了解决这个问题。
4. 在程序编译时，编译器将程序中出现的内联函数的调用表达式用内联函数的函数体来替换。由于在编译时，将内联函数体的代码替代到程序中，因此会增加目标程序代码量，进而增加空间开销，而在时间上开销不想函数调用那么大。（空间换时间）
5. 内联函数的注意事项：
6. 在一个文件中定义的内联函数，不能在另外一个文件中使用。它们通常头文件中共享
7. 内联函数中不能有循环语句、if语句或switch语句，否则，函数定义时即使用inline关键字，编译器也会把该函数作为非内联函数处理。
8. 内联函数要在函数被调用之前声明。
9. 函数指针：
10. 函数被存放在内存中的代码区（code）的一片连续的存储单元内，其中排在最前面的那个存储单元的地址就是这个函数的地址，也叫函数指针，用函数名表示，它也是一个地址常量。
11. 函数指针指向代码区中某个函数，通过函数指针可以调用相应的函数。

## 二十二、继承、多态:

1. 类的成员没有指定访问域，默认是private
2. 标识符protected和private类似，他们唯一区别就是在继承时才表现出来。当定义一个子类的时候，基类的protected成员可以被子类访问，private成员不可以被子类访问。
3. 继承：
4. 派生类集成基类的数据成员、成员函数，不继承基类的构造函数、析构函数、重载赋值运算符的成员函数。（但是能调用基类的构造函数、析构函数）
5. 继承方式：
6. 继承方法指定了派生类成员及类外对象对继承来的成员的访问权限。
7. 公有继承:

**（1）将基类的protected区成员继承到派生类的protected区，基类的public区的成员继承到派生类的public区；**

**（2）派生类中的成员函数可以直接访问基类中的public和protected成员，但不能访问基类的private成员；**

**（3）外部函数中，派生类的对象只能访问基类的public成员。**

1. 保护集成：

**（1）将基类的protected区和public区的所有成员都继承到派生类的protected区；**

**（2）派生类中的成员函数可以直接访问基类中的public和protected成员，但不能访问基类的private成员；**

**（3）外部函数中，派生类的对象不能访问基类中的任何成员；**

1. 私有继承：

**（1）将基类的protected区和public区的所有成员都继承到派生类的private区；**

**（2）派生类中的成员函数可以直接访问基类中的public和protected成员，但不能访问基类的private成员；**

**（3）外部函数中，派生类的对象不能访问基类中的任何成员；**

**（4）未加注明的情况下都指private继承方式；**

（3）构造函数和析构函数在继承中的调用：

a、执行派生类的构造函数（无论是不是默认的构造函数）的时候，首先执行父类的构造函数，再执行成员对象的构造函数。如果父类的没有默认的构造函数，则需要显示调用父类的构造函数。

比如：B继承A，B(int a, int b) :A(a) {……..}

又比如：Derive(int i) : b(i \* 10), Base(b.bb)

这里b是Derive的成员变量，Base是Derive的父类，所以先调用Base的构造函数，再调用b的构造函数。

(4) 构造函数和析构函数在组合中的调用：

* **构造函数的调用顺序：**
  + **调用基类构造函数；**
  + **调用各成员对象的构造函数，调用顺序按照它们在类中声明的顺序。**
  + **执行派生类的构造函数体中的内容；**

**析构函数的调用次序，和构造函数相反。**

1. 多态性：

概念：不同的对象接收相同的消息时，产生不同的动作

1. 编译时多态性：

通过函数名重载或运算符重载或者模板实现多态性。

1. 运行时多态性：

（2.1）在程序的运行过程中，根据具体的执行环境来动态确定调用哪一个函数。通过虚函数实现。

虚函数：

A、虚函数的作用是实现[动态联编](http://baike.baidu.com/view/535555.htm" \t "_blank)，也就是在程序的运行阶段动态地选择合适的[成员函数](http://baike.baidu.com/view/2258939.htm" \t "_blank)，在定义了虚函数后，可以在[基类](http://baike.baidu.com/view/535539.htm" \t "_blank)的[派生类](http://baike.baidu.com/view/535532.htm" \t "_blank)中对虚函数重新定义，在派生类中重新定义的函数应与虚函数具有相同的[形参](http://baike.baidu.com/view/1158689.htm" \t "_blank)个数和形参类型。以实现统一的接口，不同定义过程。如果在[派生类](http://baike.baidu.com/view/535532.htm" \t "_blank)中没有对虚函数重新定义，则它继承其[基类](http://baike.baidu.com/view/535539.htm" \t "_blank)的虚函数。

B、当程序发现虚函数名前的关键字virtual后，会自动将其作为[动态联编](http://baike.baidu.com/view/535555.htm" \t "_blank)处理，即在程序运行时动态地选择合适的成员函数。虚函数是C++[多态](http://baike.baidu.com/view/126521.htm" \t "_blank)的一种表现。

C、虚函数的目的是为了实现多态性。多态性是将接口与实现进行分离；用形象的语言来解释就是实现以共同的方法，但因个体差异而采用不同的策略。

D、如果在父类中不采用虚函数，子类直接重写父类的方法，子类对象赋值给父类指针，则无论如何，都是调用父类的对应方法。

E、virtual只能用在类定义中的原型（函数声明），不能用在函数实现时。

F、虚函数在基类与派生类中的声明（包括函数名，返回值类型和参数表列）必须一样，才能实现运行时的多态

G、基类中定义了虚函数，派生类中无论是否说明，同原型函数都自动为虚函数；

H、只有类的非静态成员函数才能声明为虚函数；

I、内联函数不能是虚函数；

J、构造函数不能是虚函数；

K、析构函数通常是虚函数；

L、通过类名::可以抑制虚函数的调用。（调用类名对应的那个虚函数）

(2.2) 成员函数调用中地址的确定：

静态绑定：

非虚成员函数是根据指向对象的指针的类型静态的选择（编译时）

动态绑定：

虚成员函数是动态（在运行时）解析的。根据对象的类型而不是根据只指向对象的指针的类型来动态的选择。使用虚函数，系统将使用动态绑定；在程序设计中提供更强的功能和灵活性；处理过程中占用一些开销。

1. 成员覆盖：

（1）概念：在派生类中新增加的成员可以与基类的成员相同，这种同名不会产生冲突。此时，当没有对这类成员指定作用域时，在派生类中所使用的这类成员为在派生类中定义的成员，因此这种关系称为成员覆盖(overridden)，或称为支配规则。

成员覆盖只是隐藏了从基类继承得到的同名成员，但在派生类中还是可以利用作用域运算符引用这种隐藏了的成员。

1. 编译器为一个空类提供的默认函数？
2. 构造函数、析构函数、拷贝构造函数、拷贝赋值操作符

解释：当我们定义一个class而没有明确定义构造函数的时候，编译器会自动假设两个重载的构造函数 (默认构造函数"default constructor" 和复制构造函数"copy constructor")。拷贝构造函数是一个只有一个参数的构造函数（原型：ClassName(ClassName &cn){};），该参数是这个class的一个对象，这个函数的功能是将被传入的对象（object）的所有非静态（non-static）成员变量的值都复制给自身这个object。  
必须注意：这两个默认构造函数（empty construction 和 copy constructor ）只有在没有其它构造函数被明确定义的情况下才存在。

一个类包含一个对赋值操作符assignation operator (=)的默认定义，该操作符用于两个同类对象之间。这个操作符将其参数对象(符号右边的对象) 的所有非静态 (non-static) 数据成员复制给其左边的对象。

7、sizeof一个空类等于多少？

sizeof一个空类返回1。所谓类的实例化就是在内存中分配一块地址，每个实例在内存中都有独一无二的地址。同样空类也会被实例化，所以编译器会给空类隐含的添加一个字节，这样空类实例化之后就有了独一无二的地址了。所以空类的sizeof为1。C++编译器不允许对象为零长度。试想一个长度为0的对象在内存中怎么存放？怎么获取它的地址？为了避免这种情况，C++强制给这种类插入一个缺省成员，长度为1。如果有自定义的变量，变量将取代这个缺省成员。

## 二十三、重载运算符：

1. Person& operator + (Person &one) //返回类型 operator 符号 (参数列表)

## 二十四、模板:

1. 函数模板：

template <class T>

T fun(T a) {

return a;

}

使用： fun<int>(200) 或 fun(200)

1. 函数模板只是说明，不能直接执行，需要实例化为模板函数后才能执行

在说明了一个函数模板后，当编译系统发现有一个对应的函数调用时，将根据实参中的类型来确认是否匹配函数模板中对应的形参，然后生成一个重载函数。该重载函数的定义体与函数模板的函数定义体相同，它称之为模板函数

1. 类模板：

与函数模板类似，支持多种数据类型

类模板的成员函数定义也是模板

作用域解析操作符前的类名为CLASSNAME<T>

类与其成员函数的定义方式不变

调用时由编译器处理类型替换的问题

## 二十五、类型转换和C++高级类型转化：

1、

## 二十六、预处理指令：

1、#define：

命令#define定义了一个标识符及一个串。在源程序中每次遇到该标识符时，均以定义的串代换它。ANSI标准将标识符定义为宏名，将替换过程称为宏替换。

2、#error：

命令error强迫编译器停止编译，主要用于程序调试。#error指令使处理器发出一条错误信息，该消息包含指令中的文本，这条指令的目的就是程序在崩溃之前能够给出一定的信息。

实例：

#ifndef \_\_GNUC\_\_ // 没有定义 \_\_GNUC\_\_ 宏，表示使用的不是gcc编译器

#error 代码中使用了 gcc 特有的扩展，必须使用 gcc 编译器编译

#endif

3、#if 、#else、#elif、#endif

#if的一般含义是如果#if后面的常量表达式为true，则编译它与#endif之间的代码，否则跳过这些代码。#endif标识一个#if块的结束。

#if Me > 0

cout << "大于0";

#elif Me == 0

cout << "等于0";

#else

cout << "小于0";

跟在#if、#elif后面的表达式在编译时求值，因此只能包含常量及已经定义过的标识符，不可以使用变量（变量在编译时不能算出来）。表达式不许含有操作符sizeof

4、#ifdef、#ifndef

如果有定义 、 如果没有定义

实例：

#ifdef Me

cout << "Me定义了";

#endif

5、#undef

取消前面定义过的宏名定义。

6、#line

改变当前行号\_\_FILE\_\_，及\_\_FILE\_\_的值

#include <stdio.h>

#line 50 "美丽"

void main()

{ /\* 行号101 \*/

printf("%d\n", \_\_LINE\_\_); /\* 行号102 \*/

printf("%s\n", \_\_FILE\_\_);

getchar();

}

7、#pragma

(1) 作用：指示编译器完成一些特定的动作

(2) 格式：#pragma para

A、message参数：能够在编译信息输出窗口中输出相应的信息。

#pragma message(“消息文本”)  
当编译器遇到这条指令时就在编译输出窗口中将消息文本打印出来。

B、once

#pragma once 在头文件最开始加入这条指令，就能保证头文件只被编译一次。

C、hdrstop

#pragma hdrstop 预编译头文件到此为止，后面的头文件不进行预编译。

## 二十七、预定义宏

1. \_\_LINE\_\_ 整数值 表示当前正在编译的行在源文件中的行数
2. \_\_FILE\_\_ 字符串 表示当前被编译的源文件的文件名
3. \_\_DATE\_\_ 字符串(” mm dd yyyy “）， 存储编译开始的日期
4. \_\_TIME\_\_  "hh:mm:ss" 的字符串，存储编译开始的时间

## 二十八、char\*及char[]:

1. char\* s1="abc" 这个指针变量指向“abc”字符串常量，这里字符串存在于常量区。因此不能修改。如果再使用一个指针 char\*s2 = “abc”; 则s2和s1指向的内存地址是一样的。（系统查看常量区里面有没有”abc”这个字符串，有的话，直接返回这个字符串的地址。）
2. char arr[] = “abc”; 这个数组的内存在栈中分配，存放的值是abc，可以修改。arr是一个指针常量，存储&arr[0]，不能修改。如果再定义一个数组，char arr2[] = “abc”; 则在栈中重新开辟一个内存空间，存储”abc”的值。

## 二十九、const：

1、const char str3[] = "abc";

const char str4[] = "abc";

str3数组和str4数组分别在栈上被分配了内存空间，只不过存放的内容一样。因此直接用==比较，是比较内存地址，两者地址不一样，返回false。加入了const，表明数组内容是一个常量，无法被修改。

1. const定义的变量在内存中只有一份，节省内存空间。

如：const doulbe  Pi=3.14159;  //此时并未将Pi放入ROM中

double i=Pi;   //此时为Pi分配内存，以后不再分配！

double j=Pi;  //没有内存分配

使用这个常量的时候，编译器应该查看这个常量是否存在内存中，如果不存在，就为这个常量分配内存。下次再使用这个常量时，就直接返回这个常量在内存中的地址

1. 编译器通常不为普通const常量分配存储空间，而是将它们保存在符号表中，这使得它成为一个编译期间的常量，没有了存储与读内存的操作，使得它的效率也很高。只在需要的时候分配一次内存。
2. const修饰成员变量  
   const修饰类的成员函数，表示成员常量，不能被修改，同时它只能在初始化列表中赋值。  
       class A  
       {   
           …  
           const int nValue;         //成员常量不能被修改  
           …  
           A(int x): nValue(x) { } ; //只能在初始化列表中赋值  
        }
3. const修饰成员函数  
   const修饰类的成员函数，则该成员函数不能修改类中任何非const成员函数。一般写在函数的最后来修饰。  
       class A  
       {   
           …  
          void function()const; //常成员函数, 它不改变对象的成员变量.

//也不能调用类中任何非const成员函数。  
}

对于const类对象/指针/引用，只能调用类的const成员函数，因此，const修饰成员函数的最重要作用就是限制对于const对象的使用。

a. const成员函数不被允许修改它所在对象的任何一个数据成员。

b. const成员函数能够访问对象的const成员，而其他成员函数不可以。

1. (3)const修饰类对象/对象指针/对象引用

             const修饰类对象表示该对象为常量对象，其中的任何成员都不能被修改。对于对象指针和对象引用也是一样。

             const修饰的对象，该对象的任何非const成员函数都不能被调用，因为任何非const成员函数会有修改成员变量的企图。  
例如：  
class AAA  
{   
    void func1();   
void func2() const;   
}   
const AAA aObj;   
aObj.func1(); ×  
aObj.func2(); 正确  
  
const AAA\* aObj = new AAA();   
aObj-> func1(); ×  
aObj-> func2(); 正确

1. 类内部定义const变量。
2. 类外部初始化：

class A{

public:

static const int a;

A(int i) {

// a = i;

}

int getA() {

return a;

}

};

const int A::a = 10;

1. 类内部初始化：

class A{

public:

const int a;

A(int i) : a(i){

}

int getA() {

return a;

}

};

## 三十、volatile：

1. 概念：volatile 关键字是一种类型修饰符，用它声明的类型变量表示可以被某些编译器未知的因素更改，比如：操作系统、硬件或者其它线程等。遇到这个关键字声明的变量，编译器对访问该变量的代码就不再进行优化，从而可以提供对特殊地址的稳定访问。声明时语法：**int volatile vInt;** 当要求使用 volatile 声明的变量的值的时候，系统总是重新从它所在的内存读取数据，即使它前面的指令刚刚从该处读取过数据。而且读取的数据立刻被保存。
2. 多线程下volatile：

有些变量是用volatile关键字声明的。当两个线程都要用到某一个变量且该变量的值会被改变时，应该用volatile声明，该关键字的作用是防止优化编译器把变量从内存装入CPU寄存器中。如果变量被装入寄存器，那么两个线程有可能一个使用内存中的变量，一个使用寄存器中的变量，这会造成程序的错误执行。volatile的意思是让编译器每次操作该变量时一定要从内存中真正取出，而不是使用已经存在寄存器中的值。

## 三十一、拷贝构造函数：

1、在对象赋值给对象时调用， A\* a = new A(10); A b = \*a;

如果是A \*c = a;则不调用，这时，指针c也指向a所指向的内存区域。

### 深拷贝和浅拷贝：

1. 默认的拷贝构造函数是将传入拷贝构造函数引用对象中非静态成员赋值给当前对象的非静态成员，当非静态成员是指针时，可能存在两个对象的成员变量指向同一块资源的情况。这种称为浅拷贝。
2. 要实现深拷贝，需要重写拷贝构造函数，**当一个对象创建时，分配了资源，这时，就需要定义自己的拷贝构造函数，使之不但拷贝成员，也拷贝资源**
3. 如果你的类需要析构函数来析构资源，则它也需要一个拷贝构造函数。因为通常对象是自动被析构的。如果需要一个自定义的析构函数，那就意味着有额外资源要在对象被析构之前释放。此时，对象的拷贝就不是浅拷贝了。

## 三十二、基础数据类型：

1. 程序是针对数据进行处理的，对各种数据进行处理之前都要对其类型预先加以说明，其目的是?
2. 更好的为其分配存储空间
3. 说明了程序处理这些数据的时候应该采用何种运算方法。
4. signed和unsigned只能修饰char和int，默认char和int为有符号，实型数总是有符号的，不用用unsigned修饰。
5. short只能修饰整型int，long只能修饰int和double。
6. 实型：实型数据一般按照指数形式存储。系统把实型数据分成小数部分和指数部分，分别存放。

float（浮点型，4字节，有效数字7位）

double（双精度，8字节，有效数字15位）

long double（长双精度，10字节，有效数字19位）

1. 除法 /对于整型数则为除法取整操作。

如：5/2得到结果2

除法 /对于浮点数则为通常意义的除法。

如：5.0/2.0得到结果为2.5

取余％ 只能对整数取余数，不能对浮点数取余数。

如5％2得到结果为1 -27%8 = -3

6、逻辑与&& 和 &的区别?

&&：有一个条件不满足，则后面的判断不执行了。利用这个优点，可以产生高效代码

& : 有一个条件不满足，则后面的判断依旧执行

## 三十三、预处理、编译、生成目标文件、链接、生成可执行文件：

### 1、预处理：

概念：C++的预处理是指在c++源程序被编译之前，由预处理器对c++程序源代码进行的处理。这个过程并不对程序的源代码进行解析。

这里的预处理器是指真正的编译开始之前由编译器调用的一个独立程序。

预处理器的工作：

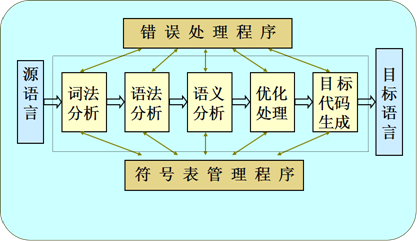
1. 宏的替换
2. 删除注释
3. 处理预处理指令，如#include，#ifdef，将程序引用的头文件包含进源代码中。

### 编译和优化:

编译过程将用户可识别的语言翻译成一组处理器可识别的操作码。

通常翻译成操作码。

**进行语法检查**、函数分配空间，将各个函数编译成二进制码



1. 词法分析：识别单词，确认词类，比如int i；知道int是一个类型，i是一个关键字，及判断i的名字是否合法
2. 语法分析：识别短语和句型的语法属性
3. 语义分析：确认单词，短语和句型的语义特征
4. 优化处理：代码优化
5. 目标代码生成：生成译文

内联函数的替换在编译这个阶段。

### 3、生成目标文件：

目标文件是指编译器生成的CPU可以识别的二进制文件。目标文件一般不能执行，因为一些函数过程没有相关的说明。目标文件中通常仅解析了文件内部的变量和函数，没有解析引用的变量和函数，这就需要将其他编写好的目标文件引用进来，来解析引用的变量和函数。编译器把一个cpp文件编译成目标文件的时候，除了要在目标文件里面写入cpp里面包含的数据和代码，还要至少提供3个表: 未解决符号表，导出符号表，地址重定向表。

1. 未解决符号表：提供了所有在编译单元里引用但是定义并不在本编译单元里的符号及其出现的地址。
2. 导出符号表：提供了本编译单元具有定义，并且愿意提供给其他编译单元使用的符号及其地址。
3. 地址重定向表：提供了本编译单元所有对自身地址的引用的记录

### 4、链接：

由汇编程序生成的目标文件并不能立即就被执行，其中可能还有许多没有解决的问题。例如，某个源文件中的函数可能引用了另一个源文件中定义的某个符号（如变量或者函数调用等）；在程序中可能调用了某个库文件中的函数，等等。所有的这些问题，都需要经链接程序的处理方能得以解决。

目标文件一般不能执行，因为一些函数过程没有相关的说明。目标文件中通常仅解析了文件内部的变量和函数，没有解析引用的变量和函数，这就需要将其他编写好的目标文件引用进来，来解析引用的变量和函数。通常引用的是库，链接完成以后生成可执行文件。

### 5、生成可执行文件：

目标文件和相关的库文件连接以后，生成的可执行文件。

## 三十四、0和null和’\0’

1、0和null和’\0’是一样的，值都是0，不过他们的表现形式不一样。

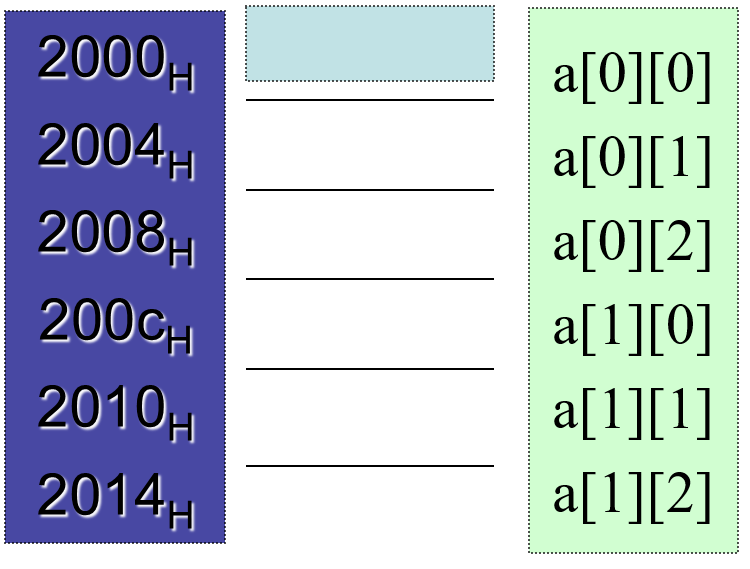
2、null在c++中直接和0一样。

3、’\0’是一个“空字符”常量，它表示一个字符串的结束，它的ASCII码值为0

4、’ ’空格的ASCII码为32， ‘0’的ASCII为48

## 三十五、数组:

1. 数组在内存中按一行一行存放，如图： int a[2][3];



1. 字符数组可以用来存放字符串，在定义时可以利用字符串常量为字符数组变量赋初值。这是其它类型数组所不具备的。其它类型数组变量在赋值时必须使用初值列表，而字符数组却可以使用字符串常量
2. **不论何种类型的指针，其在内存中占在的内存单元数是一致的。（4字节）**
3. 不能用自动变量（auto）的地址来初始化static类型的指针。
4. **指针的加减运算是以其指向的类型的字长为单位的**

## 三十六、extern c:

1. 在模本本身声明后，能被其他模块访问
2. 避免C++编译器按照C++的方式去编译C函数

## 三十七、虚函数：

1. 虚函数：用virtual修饰的函数
2. 虚函数的作用：实现多态。多态性是接口与实现分离，实现相同的方法，因个体不同而采用不同的策略实现方法。
3. 为什么析构函数常常声明为虚函数？ 因为如果析构函数不声明为虚函数，在父类指针指向子类对象的时候，我们delete掉这个基类的指针，会调用基类的析构方法，派生类的析构函数得不到调用。
4. 用对象指针来调用一个函数？
5. 如果是虚函数，会调用派生类中的实现版本
6. 如果是非虚函数，会调用指针所属类型的实现版本

5、声明为虚函数，编译器会给类添加一个虚函数表，里面存放虚函数的指针，会增加类的存储空间，因此，只有一个类被用来做基类的时候，才声明为虚函数。

## 三十八、智能指针：

1、

## 常识：

1、最长公共子串（Longest Common Substirng）和最长公共子序列（Longest Common Subsequence，LCS）的区别为：子串是串的一个连续的部分，子序列则是从不改变序列的顺序，而从序列中去掉任意，的元素而获得新的序列；也就是说，子串中字符的位置必须是连续的，子序列则可以不必连续。

1. 函数可以不显式的写返回值
2. printf("%\*s%s%\*s", (width-strlen(s)-2)/2,"",s,(width-strlen(s)-2)/2,""); //填空

（1）printf中使用%\*s，\*也是一个获取值，类似于%d，

可以表示为(width-strlen(s)-2)/2个””

1. scanf("%\*s")表示跳至下一空白字符，这里主要是中间的\*字符起的作用，

scanf("%\*d %\*d %d", &n);  
如果输入2004 2005 2006  
那么n=2006

1. strlen(s) 计算字符串s的长度。在<string.h>文件中
2. 有数组char buf[1000]; 若buf[100]=0; 则是在100坐标处，赋值为’\0’，字符串提前终止。
3. inline和#define：

（1）#define称为‘宏’，在预处理阶段简单的文本替换，容易产生二义性，不能访问对象的私有成员

（2）inline内联函数，内联函数在编译阶段，可以直接嵌入目标代码中，减少了普通函数调用的内存消耗。

（3）不同：

A、宏是简单的文本替代，内联函数是将代码插入调用调用处

B、宏不是函数，内联函数是函数，但是在编译阶段不单独产生代码，而是将代码插入到调用处。

1. 如何解除const的限制？

使用const\_cast。比如：有const char data[10] = “123”;

想改变data的值为“233“，可以const\_cast<char\*>(data)[0]=’2’;

7、C++中， ？：运算符不能重载