1.new、delete、malloc、free关系
2.delete与 delete []区别
3.C++有哪些性质（面向对象特点）
4.子类析构时要调用父类的析构函数吗?
5.多态，虚函数，纯虚函数
6.求下面函数的返回值（微软）
7.什么是“引用”？申明和使用“引用”要注意哪些问题?
8.将“引用”作为函数参数有哪些特点?
9.在什么时候需要使用“常引用”?
10.将“引用”作为函数返回值类型的格式、好处和需要遵守的规则?
11、结构与联合有和区别?
12、试写出程序结果：
13.重载（overload)和重写(overried，有的书也叫做“覆盖”）的区别?
14.有哪几种情况只能用intialization list 而不能用assignment?
15. C++是不是类型安全的?
16. main 函数执行以前，还会执行什么代码?
17. 描述内存分配方式以及它们的区别?
18.分别写出BOOL,int,float,指针类型的变量a 与“零”的比较语句。
19.请说出const与#define 相比，有何优点?
20.简述数组与指针的区别?
21. int (*s[10])(int) 表示的是什么?
22. 栈内存与文字常量区
23. 将程序跳转到指定内存地址
24. int id[sizeof(unsigned long)];这个对吗？为什么?
25. 引用与指针有什么区别?
26. const 与 #define 的比较 ，const有什么优点?
27. 复杂声明
28. 内存的分配方式有几种?
29. 基类的析构函数不是虚函数，会带来什么问题?
30. 全局变量和局部变量有什么区别？是怎么实现的？操作系统和编译器是怎么知道的?
31. C++ 内存管理
    - 内存分配方式
        - 简介
        - 明确区分堆与栈
        - 堆和栈究竟有什么区别
    - 控制C++的内存分配
        - 重载全局的new和delete操作符
        - 为单个的类重载new[]和delete[]
    - 常见的内存错误及其对策
    - 针与数组的对比
        - 修改内容
        - 内容复制与比较
        - 计算内存容量
    - 指针参数是如何传递内存的
    - 杜绝“野指针”
    - 有了malloc/free为什么还要new/delete
    - 内存耗尽怎么办

malloc/free的使用要点
new/delete的使用要点
Conclusion
32. 四种强制转换
1、static_cast
2、dynamic_cast
3、reinterpret_cast
4、const_cast
5、比较
（1）dynamic_cast vs static_cast
（2）static_cast vs reinterpret_cast

1.new、delete、malloc、free关系

delete会调用对象的析构函数,和new对应free只会释放内存，new调用构造函数。malloc与free是C++/C语言的标准库函数，new/delete是C++的运算符。它们都可用于申请动态内存和释放内存。对于非内部数据类型的对象而言，光用maloc/free无法满足动态对象的要求。对象在创建的同时要自动执行构造函数，对象在消亡之前要自动执行析构函数。由于malloc/free是库函数而不是运算符，不在编译器控制权限之内，不能够把执行构造函数和析构函数的任务强加于malloc/free。因此C++语言需要一个能完成动态内存分配和初始化工作的运算符new，以及一个能完成清理与释放内存工作的运算符delete。注意new/delete不是库函数。

2.delete与 delete []区别

delete只会调用一次析构函数，而delete[]会调用每一个成员的析构函数。在More Effective C++中有更为详细的解释："当delete操作符用于数组时，它为每个数组元素调用析构函数，然后调用operator delete来释放内存。"delete与new配套，delete []与new []配套

```
MemTest *mTest1=new MemTest[10];
MemTest *mTest2=new MemTest;
Int *pInt1=new int [10];
Int *pInt2=new int;
delete[]pInt1; //-1-
delete[]pInt2; //-2-
delete[]mTest1;//-3-
delete[]mTest2;//-4-
```

在-4-处报错。

这就说明：对于内建简单数据类型，delete和delete[]功能是相同的。对于自定义的复杂数据类型，delete和delete[]不能互用。delete[]删除一个数组，delete删除一个指针。简单来说，用new分配的内存用delete删除；用new[]分配的内存用delete[]删除。delete[]会调用数组元素的析构函数。内部数据类型没有析构函数，所以问题不大。如果你在用delete时没用括号，delete就会认为指向的是单个对象，否则，它就会认为指向的是一个数组。

3.C++有哪些性质（面向对象特点）

封装，继承和多态。

4.子类析构时要调用父类的析构函数吗?

析构函数调用的次序是先派生类的析构后基类的析构，也就是说在基类的的析构调用的时候,派生类的信息已经全部销毁了。定义一个对象时先调用基类的构造函数、然后调用派生类的构造函数；析构的时候恰好相反：先调用派生类的析构函数、然后调用基类的析构函数。

5.多态，虚函数，纯虚函数

多态：是对于不同对象接收相同消息时产生不同的动作。C++的多态性具体体现在运行和编译两个方面：在程序运行时的多态性通过继承和虚函数来体现；

在程序编译时多态性体现在函数和运算符的重载上；

虚函数：在基类中冠以关键字 virtual 的成员函数。 它提供了一种接口界面。允许在派生类中对基类的虚函数重新定义。

纯虚函数的作用：在基类中为其派生类保留一个函数的名字，以便派生类根据需要对它进行定义。作为接口而存在 纯虚函数不具备函数的功能，一般不能直接被调用。

从基类继承来的纯虚函数，在派生类中仍是虚函数。如果一个类中至少有一个纯虚函数，那么这个类被称为抽象类（abstract class）。

抽象类中不仅包括纯虚函数，也可包括虚函数。抽象类必须用作派生其他类的基类，而不能用于直接创建对象实例。但仍可使用指向抽象类的指针支持运行时多态性。

6.求下面函数的返回值（微软）

```
int func(x){
  int countx = 0;
  while(x) {
    countx ++;
    x = x&(x-1);
  }
  return countx;
}
```

假定x = 9999。 答案：8

思路：将x转化为2进制，看含有的1的个数。

7.什么是“引用”？申明和使用“引用”要注意哪些问题?

答：引用就是某个目标变量的“别名”(alias)，对应用的操作与对变量直接操作效果完全相同。申明一个引用的时候，切记要对其进行初始化。引用声明完毕后，相当于目标变量名有两个名称，即该目标原名称和引用名，不能再把该引用名作为其他变量名的别名。声明一个引用，不是新定义了一个变量，它只表示该引用名是目标变量名的一个别名，它本身不是一种数据类型，因此引用本身不占存储单元，系统也不给引用分配存储单元。不能建立数组的引用。

8.将“引用”作为函数参数有哪些特点?

（1）传递引用给函数与传递指针的效果是一样的。这时，被调函数的形参就成为原来主调函数中的实参变量或对象的一个别名来使用，所以在被调函数中对形参变量的操作就是对其相应的目标对象（在主调函数中）的操作。

（2）使用引用传递函数的参数，在内存中并没有产生实参的副本，它是直接对实参操作；而使用一般变量传递函数的参数，当发生函数调用时，需要给形参分配存储单元，形参变量是实参变量的副本；如果传递的是对象，还将调用拷贝构造函数。因此，当参数传递的数据较大时，用引用比用一般变量传递参数的效率和所占空间都好。

（3）使用指针作为函数的参数虽然也能达到与使用引用的效果，但是，在被调函数中同样要给形参分配存储单元，且需要重复使用"*指针变量名"的形式进行运算，这很容易产生错误且程序的阅读性较差；另一方面，在主调函数的调用点处，必须用变量的地址作为实参。而引用更容易使用，更清晰。

9.在什么时候需要使用“常引用”?

如果既要利用引用提高程序的效率，又要保护传递给函数的数据不在函数中被改变，就应使用常引用。常引用声明方式：const 类型标识符 &引用名=目标变量名；

例1

```
int a ;

const int &ra=a;

ra=1; //错误

a=1; //正确
```

例2

```
string foo( );

void bar(string & s);
```

那么下面的表达式将是非法的：

```
bar(foo( ));

bar("hello world");
```

原因在于foo( )和"hello world"串都会产生一个临时对象，而在C++中，这些临时对象都是const类型的。因此上面的表达式就是试图将一个const类型的对象转换为非const类型，这是非法的。引用型参数应该在能被定义为const的情况下，尽量定义为const 。

10.将“引用”作为函数返回值类型的格式、好处和需要遵守的规则?

格式：类型标识符 &函数名（形参列表及类型说明）{ //函数体 }

好处：在内存中不产生被返回值的副本；（注意：正是因为这点原因，所以返回一个局部变量的引用是不可取的。因为随着该局部变量生存期的结束，相应的引用也会失效，产生runtime error!

注意事项：

（1）不能返回局部变量的引用。这条可以参照Effective C++[1]的Item 31。主要原因是局部变量会在函数返回后被销毁，因此被返回的引用就成为了"无所指"的引用，程序会进入未知状态。

（2）不能返回函数内部new分配的内存的引用。这条可以参照Effective C++[1]的Item 31。虽然不存在局部变量的被动销毁问题，可对于这种情况（返回函数内部new分配内存的引用），又面临其它尴尬局面。例如，被函数返回的引用只是作为一个临时变量出现，而没有被赋予一个实际的变量，那么这个引用所指向的空间（由new分配）就无法释放，造成memory leak。

（3）可以返回类成员的引用，但最好是const。这条原则可以参照Effective C++[1]的Item 30。主要原因是当对象的属性是与某种业务规则（business rule）相关联的时候，其赋值常常与某些其它属性或者对象的状态有关，因此有必要将赋值操作封装在一个业务规则当中。如果其它对象可以获得该属性的非常量引用（或指针），那么对该属性的单纯赋值就会破坏业务规则的完整性。

（4）流操作符重载返回值申明为“引用”的作用：

流操作符<<和>>，这两个操作符常常希望被连续使用，例如：cout << "hello" << endl; 因此这两个操作符的返回值应该是一个仍然支持这两个操作符的流引用。可选的其它方案包括：返回一个流对象和返回一个流对象指针。但是对于返回一个流对象，程序必须重新（拷贝）构造一个新的流对象，也就是说，连续的两个<<操作符实际上是针对不同对象的！这无法让人接受。对于返回一个流指针则不能连续使用<<操作符。因此，返回一个流对象引用是惟一选择。这个唯一选择很关键，它说明了引用的重要性以及无可替代性，也许这就是C++语言中引入引用这个概念的原因吧。

赋值操作符=。这个操作符象流操作符一样，是可以连续使用的，例如：x = j = 10;或者(x=10)=100;赋值操作符的返回值必须是一个左值，以便可以被继续赋值。因此引用成了这个操作符的惟一返回值选择。

```
#include<iostream.h>

int &put(int n);
int vals[10];
int error=-1;

void main() {
  put(0)=10; //以put(0)函数值作为左值，等价于vals[0]=10;
  put(9)=20; //以put(9)函数值作为左值，等价于vals[9]=20;
  cout<<vals[0];
  cout<<vals[9];
}

int &put(int n) {
  if (n>=0 && n<=9 )
```

```
    return vals[n];
  else {
    cout<<"subscript error";
    return error;
  }
}
```

（5）在另外的一些操作符中，却千万不能返回引用：+-*/ 四则运算符。它们不能返回引用，Effective C++[1]的Item23详细的讨论了这个问题。主要原因是这四个操作符没有side effect，因此，它们必须构造一个对象作为返回值，可选的方案包括：返回一个对象、返回一个局部变量的引用，返回一个new分配的对象的引用、返回一个静态对象引用。根据前面提到的引用作为返回值的三个规则，第2、3两个方案都被否决了。静态对象的引用又因为((a+b) == (c+d))会永远为true而导致错误。所以可选的只剩下返回一个对象了。

11、结构与联合有和区别?

(1). 结构和联合都是由多个不同的数据类型成员组成, 但在任何同一时刻, 联合中只存放了一个被选中的成员（所有成员共用一块地址空间）, 而结构的所有成员都存在（不同成员的存放地址不同）。

(2). 对于联合的不同成员赋值, 将会对其它成员重写, 原来成员的值就不存在了, 而对于结构的不同成员赋值是互不影响的。

12、试写出程序结果：

```
int  a=4;

int  &f(int  x) {
  a=a+x;
  return  a;
}

int main(void) {
  int   t=5;
  cout<<f(t)<<endl;  a = 9
  f(t)=20;             a = 20
  cout<<f(t)<<endl;     t = 5,a = 20  a = 25
  t=f(t);                a = 30 t = 30
  cout<<f(t)<<endl;  }    t = 60
}
```

13.重载（overload)和重写(overried，有的书也叫做“覆盖”）的区别?

常考的题目。从定义上来说：

重载：是指允许存在多个同名函数，而这些函数的参数表不同（或许参数个数不同，或许参数类型不同，或许两者都不同）。

重写：是指子类重新定义父类虚函数的方法。

从实现原理上来说：

重载：编译器根据函数不同的参数表，对同名函数的名称做修饰，然后这些同名函数就成了不同的函数（至少对于编译器来说是这样的）。如，有两个同名函数：function func(p:integer):integer;和function func(p:string):integer;。那么编译器做过修饰后的函数名称可能是这样的：int_func、str_func。对于这两个函数的调用，在编译器间就已经确定了，是静态的。也就是说，它们的地址在编译期就绑定了（早绑定），因此，重载和多态无关！

重写：和多态真正相关。当子类重新定义了父类的虚函数后，父类指针根据赋给它的不同的子类指针，动态的调用属于子类的该函数，这样的函数调用在编译期间是无法确定的（调用的子类的虚函数的地址无法给出）。因此，这样的函数地址是在运行期绑定的（晚绑定）。

14.有哪几种情况只能用intialization list 而不能用assignment?

答案：当类中含有const、reference 成员变量；基类的构造函数都需要初始化表。

15. C++是不是类型安全的?

答案：不是。两个不同类型的指针之间可以强制转换（用reinterpret cast)。C#是类型安全的。

16. main 函数执行以前，还会执行什么代码?

答案：全局对象的构造函数会在main 函数之前执行。

17. 描述内存分配方式以及它们的区别?

1） 从静态存储区域分配。内存在程序编译的时候就已经分配好，这块内存在程序的整个运行期间都存在。例如全局变量，static 变量。

2） 在栈上创建。在执行函数时，函数内局部变量的存储单元都可以在栈上创建，函数执行结束时这些存储单元自动被释放。栈内存分配运算内置于处理器的指令集。

3） 从堆上分配，亦称动态内存分配。程序在运行的时候用malloc 或new 申请任意多少的内存，程序员自己负责在何时用free 或delete 释放内存。动态内存的生存期由程序员决定，使用非常灵活，但问题也最多。

18.分别写出BOOL,int,float,指针类型的变量 a 与“零”的比较语句。

答案：

BOOL : if (!a) or if(a)

int : if (a == 0)

float : const EXPRESSION EXP = 0.000001

if (a < EXP && a >-EXP)

pointer : if (a != NULL) or if(a == NULL)

19.请说出const与#define 相比，有何优点?

答案：

const作用：定义常量、修饰函数参数、修饰函数返回值三个作用。被Const修饰的东西都受到强制保护，可以预防意外的变动，能提高程序的健壮性。

1） const 常量有数据类型，而宏常量没有数据类型。编译器可以对前者进行类型安全检查。而对后者只进行字符替换，没有类型安全检查，并且在字符替换可能会产生意料不到的错误。

2） 有些集成化的调试工具可以对const 常量进行调试，但是不能对宏常量进行调试。

20.简述数组与指针的区别?

数组要么在静态存储区被创建（如全局数组），要么在栈上被创建。指针可以随时指向任意类型的内存块。

(1)修改内容上的差别

char a[] = “hello”;

a[0] = ‘X’;

char *p = “world”; // 注意p 指向常量字符串

p[0] = ‘X’; // 编译器不能发现该错误，运行时错误

(2) 用运算符sizeof 可以计算出数组的容量（字节数）。sizeof(p),p 为指针得到的是一个指针变量的字节数，而不是p 所指的内存容量。C++/C 语言没有办法知道指针所指的内存容量，除非在申请内存时记住它。注意当数组作为函数的参数进行传递时，该数组自动退化为同类型的指针。

char a[] = "hello world";

char *p = a;

cout<< sizeof(a) << endl; // 12 字节

cout<< sizeof(p) << endl; // 4 字节

计算数组和指针的内存容量

void Func(char a[100])

{

cout<< sizeof(a) << endl; // 4 字节而不是100 字节

}

21. int (*s[10])(int) 表示的是什么?

int (*s[10])(int) 函数指针数组，每个指针指向一个int func(int param)的函数。

22. 栈内存与文字常量区

```
char str1[] = "abc";
char str2[] = "abc";
const char str3[] = "abc";
const char str4[] = "abc";
const char *str5 = "abc";
const char *str6 = "abc";
char *str7 = "abc";
char *str8 = "abc";
cout << ( str1 == str2 ) << endl;//0  分别指向各自的栈内存
cout << ( str3 == str4 ) << endl;//0  分别指向各自的栈内存
cout << ( str5 == str6 ) << endl;//1指向文字常量区地址相同
cout << ( str7 == str8 ) << endl;//1指向文字常量区地址相同
```

结果是：0 0 1 1

解答：str1,str2,str3,str4是数组变量，它们有各自的内存空间；而str5,str6,str7,str8是指针，它们指向相同的常量区域。

23. 将程序跳转到指定内存地址

要对绝对地址0x100000赋值，我们可以用(unsigned int*)0x100000 = 1234;那么要是想让程序跳转到绝对地址是0x100000去执行，应该怎么做? *((void ()( ))0x100000) ( );* 首先要将0x100000强制转换成函数指针,即: (void (*)())0x100000 然后再调用它: *((void ()())0x100000)();* 用typedef可以看得更直观些: typedef void(*)() voidFuncPtr; *((voidFuncPtr)0x100000)();

24. int id[sizeof(unsigned long)];这个对吗？为什么?

答案:正确 这个 sizeof是编译时运算符，编译时就确定了 ,可以看成和机器有关的常量。

25. 引用与指针有什么区别?

【参考答案】
1) 引用必须被初始化，指针不必。

2) 引用初始化以后不能被改变，指针可以改变所指的对象。

3) 不存在指向空值的引用，但是存在指向空值的指针。

26. const 与 #define 的比较， const有什么优点?

【参考答案】

（1） const 常量有数据类型，而宏常量没有数据类型。编译器可以对前者进行类型安全检查。而对后者只进行字符替换，没有类型安全检查，并且在字符替换可能会产生意料不到的错误（边际效应） 。

（2） 有些集成化的调试工具可以对 const 常量进行调试，但是不能对宏常量进行调试。

27. 复杂声明

void * (* (*fp1)(int))[10];

float ((fp2)(int,int,int))(int);

int (* (* fp3)())10;

分别表示什么意思？ 【标准答案】

1.void * (* (*fp1)(int))[10]; **fp1是一个指针，指向一个函数，这个函数的参数为int型，函数的返回值是一个指针，这个指针指向一个数组，这个数组有10个元素，每个元素是一个void型指针。

2.float ((fp2)(int,int,int))(int); fp2是一个指针，指向一个函数，这个函数的参数为3个int型，函数的返回值是一个指针，这个指针指向一个函数，这个函数的参数为int型，函数的返回值是float型。 3.int (* (* fp3)())10; fp3是一个指针，指向一个函数，这个函数的参数为空，函数的返回值是一个指针，这个指针指向一个数组，这个数组有10个元素，每个元素是一个指针，指向一个函数，这个函数的参数为空，函数的返回值是int型。

28. 内存的分配方式有几种?

【参考答案】

一、从静态存储区域分配。内存在程序编译的时候就已经分配好，这块内存在程序的整个运行期间都存在。例如全局变量。

二、在栈上创建。在执行函数时，函数内局部变量的存储单元都可以在栈上创建，函数执行结束时这些存储单元自动被释放。栈内存分配运算内置于处理器的指令集中，效率很高，但是分配的内存容量有限。

三、从堆上分配，亦称动态内存分配。程序在运行的时候用malloc或new申请任意多少的内存，程序员自己负责在何时用free或delete释放内存。动态内存的生存期由我们决定，使用非常灵活，但问题也最多。

29. 基类的析构函数不是虚函数，会带来什么问题?

【参考答案】派生类的析构函数用不上，会造成资源的泄漏。

30. 全局变量和局部变量有什么区别？是怎么实现的？操作系统和编译器是怎么知道的?

【参考答案】

生命周期不同：

全局变量随主程序创建和创建，随主程序销毁而销毁；局部变量在局部函数内部，甚至局部循环体等内部存在，退出就不存在；

使用方式不同：通过声明后全局变量程序的各个部分都可以用到；局部变量只能在局部使用；分配在栈区。

操作系统和编译器通过内存分配的位置来知道的，全局变量分配在全局数据段并且在程序开始运行的时候被加载。局部变量则分配在堆栈里面 。

31. C++ 内存管理

C/C++内存管理详解

By ShinChan

Published Sep 25 2014

Contents1. 内存分配方式1.1. 简介1.2. 明确区分堆与栈1.3. 堆和栈究竟有什么区别2. 控制C++的内存分配2.1. 重载全局的new和delete操作符2.2. 为单个的类重载new[]和delete[]3. 常见的内存错误及其对策4. 针与数组的对比4.1. 修改内容4.2. 内容复制与比较4.3. 计算内存容量5. 指针参数是如何传递内存的6. 杜绝“野指针”7. 有了malloc/free为什么还要new/delete8. 内存耗尽怎么办9. malloc/free的使用要点10. new/delete的使用要点11. Conclusion

内存管理是C++最令人切齿痛恨的问题，也是C++最有争议的问题，C++高手从中获得了更好的性能，更大的自由，C++菜鸟的收获则是一遍一遍的检查代码和对C++的痛恨，但内存管理在C++中无处不在，内存泄漏几乎在每个C++程序中都会发生，因此要想成为C++高手，内存管理一关是必须要过的，除非放弃C++，转到Java或者.NET，他们的内存管理基本是自动的，当然你也放弃了自由和对内存的支配权，还放弃了C++超绝的性能。　　伟大的Bill Gates 曾经失言：

> *640K ought to be enough for everybody* — Bill Gates 1981

程序员们经常编写内存管理程序，往往提心吊胆。如果不想触雷，唯一的解决办法就是发现所有潜伏的地雷并且排除它们，躲是躲不了的。

内存分配方式

简介

在C++中，内存分成5个区，他们分别是堆、栈、自由存储区、全局/静态存储区和常量存储区。

栈：在执行函数时，函数内局部变量的存储单元都可以在栈上创建，函数执行结束时这些存储单元自动被释放。栈内存分配运算内置于处理器的指令集中，效率很高，但是分配的内存容量有限。

堆：就是那些由`new`分配的内存块，他们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，那么在程序结束后，操作系统会自动回收。

自由存储区：就是那些由`malloc`等分配的内存块，他和堆是十分相似的，不过它是用`free`来结束自己的生命的。　　**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，在以前的C语言中，全局变量又分为初始化的和未初始化的，在C++里面没有这个区分了，他们共同占用同一块内存区。　　**常量存储区**：这是一块比较特殊的存储区，他们里面存放的是常量，不允许修改。

明确区分堆与栈

堆与栈的区分问题，似乎是一个永恒的话题，由此可见，初学者对此往往是混淆不清的，所以我决定拿他第一个开刀。　　首先，我们举一个例子：

```
void f() { int* p=new int[5]; }
```

这条短短的一句话就包含了堆与栈，看到`new`，我们首先就应该想到，我们分配了一块堆内存，那么指针`p`呢？他分配的是一块栈内存，所以这句话的意思就是：在栈内存中存放了一个指向一块堆内存的指针`p`。在程序会先确定在堆中分配内存的大小，然后调用`operator new`分配内存，然后返回这块内存的首地址，放入栈中，他在VC6下的汇编代码如下：

```
00401028 push 14h
0040102A call operator new (00401060)
0040102F add esp,4
00401032 mov dword ptr [ebp-8],eax
00401035 mov eax,dword ptr [ebp-8]
00401038 mov dword ptr [ebp-4],eax
```

这里，我们为了简单并没有释放内存，那么该怎么去释放呢？是`delete p`么？澳，错了，应该是`delete []p`，这是为了告诉编译器：我删除的是一个数组，编译器就会根据相应的`Cookie`信息去进行释放内存的工作。

堆和栈究竟有什么区别

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：）

控制C++的内存分配

在嵌入式系统中使用C++的一个常见问题是内存分配，即对`new`和`delete`操作符的失控。具有讽刺意味的是，问题的根源却是C++对内存的管理非常的容易而且安全。具体地说，当一个对象被消除时，它的析构函数能够安全的释放所分配的内存。 这当然是个好事情，但是这种使用的简单性使得程序员们过度使用`new`和`delete`，而不注意在嵌入式C++环境中的因果关系。并且，在嵌入式系统中，由于内存的限制，频繁的动态分配不定大小的内存会引起很大的问题以及堆破碎的风险。 作为忠告，保守的使用内存分配是嵌入式环境中的第一原则。 但当你必须要使用`new`和`delete`时，你不得不控制C++中的内存分配。你需要用一个全局的`new`和`delete`来代替系统的内存分配符，并且一个类一个类的重载`new`和`delete`。 一个防止堆破碎的通用方法是从不同固定大小的内存持中分配不同类型的对象。对每个类重载`new`和`delete`就提供了这样的控制。

重载全局的new和delete操作符

可以很容易地重载new 和 delete 操作符，如下所示:

```
void * operator new(size_t size){
    void *p = malloc(size);
    return (p);
}

void operator delete(void *p){
    free(p);
}
```

这段代码可以代替默认的操作符来满足内存分配的请求。出于解释C++的目的，我们也可以直接调用`malloc()`和`free()`。 也可以对单个类的`new`和`delete`操作符重载。这是你能灵活的控制对象的内存分配。

```
class TestClass {
    public:
    void * operator new(size_t size);
    void operator delete(void *p);
    // .. other members here ...
};

void *TestClass::operator new(size_t size){
    void *p = malloc(size); // Replace this with alternative allocator
    return (p);
}

void TestClass::operator delete(void *p){
    free(p); // Replace this with alternative de-allocator
}
```

所有`TestClass`对象的内存分配都采用这段代码。更进一步，任何从`TestClass`继承的类也都采用这一方式，除非它自己也重载了`new`和`delete`操作符。通过重载`new`和`delete`操作符的方法，你可以自由地采用不同的分配策略，从不同的内存池中分配不同的类对象。

为单个的类重载new[]和delete[]

必须小心对象数组的分配。你可能希望调用到被你重载过的`new`和`delete`操作符，但并不如此。内存的请求被定向到全局的`new[]`和`delete[]`操作符，而这些内存来自于系统堆。 C++将对象数组的内存分配作为一个单独的操作，而不同于单个对象的内存分配。为了改变这种方式，你同样需要重载`new[]`和`delete[]`操作符。

```
class TestClass {
    public:
    void * operator new[ ](size_t size);
```

```
    void operator delete[ ](void *p);
    // .. other members here ..
};

void *TestClass::operator new[ ](size_t size){
    void *p = malloc(size);
    return (p);
}

void TestClass::operator delete[ ](void *p){
    free(p);
}

int main(void){
    TestClass *p = new TestClass[10];
    // ... etc ...
    delete[ ] p;
}
```

但是**注意**：对于多数C++的实现，`new[]`操作符中的个数参数是数组的大小加上额外的存储对象数目的一些字节。在你的内存分配机制重要考虑的这一点。你应该尽量避免分配对象数组，从而使你的内存分配策略简单。

常见的内存错误及其对策

发生内存错误是件非常麻烦的事情。编译器不能自动发现这些错误，通常是在程序运行时才能捕捉到。而这些错误大多没有明显的症状，时隐时现，增加了改错的难度。有时用户怒气冲冲地把你找来，程序却没有发生任何问题，你一走，错误又发作了。 常见的内存错误及其对策如下：

- 内存分配未成功，却使用了它。编程新手常犯这种错误，因为他们没有意识到内存分配会不成功。常用解决办法是，在使用内存之前检查指针是否为`NULL`。如果指针`p`是函数的参数，那么在函数的入口处用`assert(p!=NULL)`进行检查。如果是用`malloc`或`new`来申请内存，应该用`if(p==NULL)` 或`if(p!=NULL)`进行防错处理。
- 内存分配虽然成功，但是尚未初始化就引用它。犯这种错误主要有两个起因：一是没有初始化的观念；二是误以为内存的缺省初值全为零，导致引用初值错误（例如数组）。内存的缺省初值究竟是什么并没有统一的标准，尽管有些时候为零值，我们宁可信其无不可信其有。所以无论用何种方式创建数组，都别忘了赋初值，即便是赋零值也不可省略，不要嫌麻烦。
- 内存分配成功并且已经初始化，但操作越过了内存的边界。例如在使用数组时经常发生下标“多1”或者“少1”的操作。特别是在`for`循环语句中，循环次数很容易搞错，导致数组操作越界。
- 忘记了释放内存，造成内存泄露。含有这种错误的函数每被调用一次就丢失一块内存。刚开始时系统的内存充足，你看不到错误。终有一次程序突然死掉，系统出现提示：内存耗尽。动态内存的申请与释放必须配对，程序中`malloc`与`free`的使用次数一定要相同，否则肯定有错误（`new/delete`同理）。
- 释放了内存却继续使用它。

有三种情况：　　(1). 程序中的对象调用关系过于复杂，实在难以搞清楚某个对象究竟是否已经释放了内存，此时应该重新设计数据结构，从根本上解决对象管理的混乱局面。　　(2). 函数的`return`语句写错了，注意不要返回指向“栈内存”的“指针”或者“引用”，因为该内存在函数体结束时被自动销毁。

(3). 使用`free`或`delete`释放了内存后，没有将指针设置为`NULL`。**导致产生“野指针”。　　那么如何避免产生野指针呢**？这里列出了5条规则，平常写程序时多注意一下，养成良好的习惯。

> 规则1：用`malloc`或`new`申请内存之后，应该立即检查指针值是否为`NULL`。防止使用指针值为`NULL`的内存。 规则2：不要忘记为数组和动态内存赋初值。防止将未被初始化的内存作为右值使用。 规则3：避免数组或指针的下标越界，特别要当心发生“多1”或者“少1”操作。 规则4：动态内存的申请与释放必须配对，防止内存泄漏。 规则5：用`free`或`delete`释放了内存之后，立即将指针设置为`NULL`，防止产生“野指针”。

针与数组的对比

C++/C程序中，指针和数组在不少地方可以相互替换着用，让人产生一种错觉，以为两者是等价的。　　数组要么在静态存储区被创建（如全局数组），要么在栈上被创建。数组名对应着（而不是指向）一块内存，其地址与容量在生命期内保持不变，只有数组的内容可以改变。　　指针可以随时指向任意类型的内存块，它的特征是“可变”，所以我们常用指针来操作动态内存。指针远比数组灵活，但也更危险。　　下面以字符串为例比较指针与数组的特性。

修改内容

下面示例中，字符数组a的容量是6个字符，其内容为 hello。a的内容可以改变，如`a[0]= ‘X’`。指针p指向常量字符串“world”（位于静态存储区，内容为world），常量字符串的内容是不可以被修改的。从语法上看，编译器并不觉得语句`p[0]= ‘X’`有什么不妥，但是该语句企图修改常量字符串的内容而导致运行错误。

```
char a[] = “hello”;
a[0] = ‘X’;
cout << a << endl;
char *p = “world”; // 注意p指向常量字符串
p[0] = ‘X’; // 编译器不能发现该错误
cout << p << endl;
```

内容复制与比较

不能对数组名进行直接复制与比较。若想把数组a的内容复制给数组b，不能用语句`b = a`，否则将产生编译错误。应该用标准库函数`strcpy`进行复制。同理，比较b和a的内容是否相同，不能用`if(b==a)`来判断，应该用标准库函数`strcmp`进行比较。　　语句`p = a`并不能把a的内容复制指针p，而是把a的地址赋给了p。要想复制a的内容，可以先用库函数`malloc`为p申请一块容量为`strlen(a)+1`个字符的内存，再用`strcpy`进行字符串复制。同理，语句`if(p==a)`比较的不是内容而是地址，应该用库函数`strcmp`来比较。

```
// 数组…
char a[] = "hello";
char b[10];
strcpy(b, a); // 不能用 b = a;
if(strcmp(b, a) == 0) // 不能用 if (b == a)
…

// 指针…
int len = strlen(a);
char *p = (char *)malloc(sizeof(char)*(len+1));
strcpy(p,a); // 不要用 p = a;
if(strcmp(p, a) == 0) // 不要用 if (p == a)
…
```

计算内存容量

用运算符`sizeof`可以计算出数组的容量（字节数）。如下示例中，`sizeof(a)`的值是12（注意别忘了”）。指针p指向a，但是`sizeof(p)`的值却是4。这是因为`sizeof(p)`得到的是一个指针变量的字节数，相当于`sizeof(char*)`，而不是p所指的内存容量。C++/C语言没有办法知道指针所指的内存容量，除非在申请内存时记住它。

```
char a[] = "hello world";
char *p = a;
cout<< sizeof(a) << endl; // 12字节
cout<< sizeof(p) << endl; // 4字节
```

注意当数组作为函数的参数进行传递时，该数组自动退化为同类型的指针。如下示例中，不论数组a的容量是多少，`sizeof(a)`始终等于`sizeof(char *)`。

```
void Func(char a[100]){
    cout<< sizeof(a) << endl; // 4字节而不是100字节
}
```

指针参数是如何传递内存的

如果函数的参数是一个指针，不要指望用该指针去申请动态内存。如下示例中，Test函数的语句`GetMemory(str, 200)`并没有使`str`获得期望的内存，`str`依旧是`NULL`，为什么?

```
void GetMemory(char *p, int num){
    p = (char *)malloc(sizeof(char) * num);
}

void Test(void){
    char *str = NULL;
    GetMemory(str, 100); // str 仍然为 NULL

    strcpy(str, "hello"); // 运行错误
}
```

毛病出在函数`GetMemory`中。编译器总是要为函数的每个参数制作临时副本，指针参数p的副本是`_p`，编译器使`_p=p`。如果函数体内的程序修改了`_p`的内容，就导致参数p的内容作相应的修改。这就是指针可以用作输出参数的原因。在本例中，`_p`申请了新的内存，只是把`_p`所指的内存地址改变了，但是p丝毫未变。所以函数`GetMemory`并不能输出任何东西。事实上，每执行一次`GetMemory`就会泄露一块内存，因为没有用`free`释放内存。　　如果非得要用指针参数去申请内存，那么应该改用“指向指针的指针”，见示例：

```
void GetMemory2(char **p, int num){
    *p = (char *)malloc(sizeof(char) * num);
}

void Test2(void){
    char *str = NULL;
    GetMemory2(&str, 100); // 注意参数是 &str，而不是str
    strcpy(str, "hello");
    cout<< str << endl;

    free(str);
}
```

由于“指向指针的指针”这个概念不容易理解，我们可以用函数返回值来传递动态内存。这种方法更加简单，见示例：

```
char *GetMemory3(int num){
    char *p = (char *)malloc(sizeof(char) * num);
    return p;
}

void Test3(void){
    char *str = NULL;
    str = GetMemory3(100);

    strcpy(str, "hello");
    cout<< str << endl;
    free(str);
}
```

用函数返回值来传递动态内存这种方法虽然好用，但是常常有人把`return`语句用错了。这里强调不要用`return`语句返回指向“栈内存”的指针，因为该内存在函数结束时自动消亡，见示例：

```
char *GetString(void){
    char p[] = "hello world";
    return p; // 编译器将提出警告
}

void Test4(void){
    char *str = NULL;
    str = GetString(); // str 的内容是垃圾
    cout<< str << endl;
}
```

用调试器逐步跟踪`Test4`，发现执行`str = GetString`语句后`str`不再是`NULL`指针，但是`str`的内容不是`“hello world”`而是垃圾。　　如果把上述示例改写成如下示例，会怎么样?

```
char *GetString2(void){
    char *p = "hello world";
    return p;
}

void Test5(void){
    char *str = NULL;
    str = GetString2();
    cout<< str << endl;
}
```

函数`Test5`运行虽然不会出错，但是函数`GetString2`的设计概念却是错误的。因为`GetString2`内的`“hello world”`是常量字符串，位于静态存储区，它在程序生命期内恒定不变。无论什么时候调用`GetString2`，它返回的始终是同一个“只读”的内存块。

杜绝“野指针”

“野指针”不是`NULL`指针，是指向“垃圾”内存的指针。人们一般不会错用`NULL`指针，因为用`if`语句很容易判断。但是“野指针”是很危险的，`if`语句对它不起作用。 “野指针”的成因主要有三种：

(1). 指针变量没有被初始化。任何指针变量刚被创建时不会自动成为NULL指针，它的缺省值是随机的，它会乱指一气。所以，指针变量在创建的同时应当被初始化，要么将指针设置为NULL，要么让它指向合法的内存。例如：

```
char *p = NULL;
char *str = (char *) malloc(100);
```

(2). 指针p被free或者delete之后，没有置为NULL，让人误以为p是个合法的指针。

(3). 指针操作超越了变量的作用域范围。这种情况让人防不胜防，示例程序如下：

```
class A{
    public:
    void Func(void){ cout << “Func of class A” << endl; }
};

void Test(void){
    A *p;
    {
    A a;
    p = &a; // 注意 a 的生命期
    }
    p->Func(); // p是“野指针”
}
```

函数`Test`在执行语句`p->Func()`时，对象a已经消失，而p是指向a的，所以p就成了“野指针”。但奇怪的是我运行这个程序时居然没有出错，这可能与编译器有关。

有了malloc/free为什么还要new/delete

`malloc`与`free`是C++/C语言的标准库函数，`new/delete`是C++的运算符。它们都可用于申请动态内存和释放内存。　　对于非内部数据类型的对象而言，光用`maloc/free`无法满足动态对象的要求。对象在创建的同时要自动执行构造函数，对象在消亡之前要自动执行析构函数。由于`malloc/free`是库函数而不是运算符，不在编译器控制权限之内，不能够把执行构造函数和析构函数的任务强加于`malloc/free`。　　因此C++语言需要一个能完成动态内存分配和初始化工作的运算符`new`，以及一个能完成清理与释放内存工作的运算符`delet`e。注意`new/delete`不是库函数。我们先看一看`malloc/free`和`new/delete`如何实现对象的动态内存管理，见示例：

```
class Obj{
    public :
   Obj(void){ cout << “Initialization” << endl; }
   ~Obj(void){ cout << “Destroy” << endl; }
   void Initialize(void){ cout << “Initialization” << endl; }
   void Destroy(void){ cout << “Destroy” << endl; }
};

void UseMallocFree(void){
    Obj *a = (obj *)malloc(sizeof(obj)); // 申请动态内存
    a->Initialize(); // 初始化
    //…

    a->Destroy(); // 清除工作
    free(a); // 释放内存
}
void UseNewDelete(void){
    Obj *a = new Obj; // 申请动态内存并且初始化
    //…
```

```
    delete a; // 清除并且释放内存
}
```

类Obj的函数Initialize模拟了构造函数的功能，函数Destroy模拟了析构函数的功能。函数UseMallocFree中，由于malloc/free不能执行构造函数与析构函数，必须调用成员函数Initialize和Destroy来完成初始化与清除工作。函数UseNewDelete则简单得多。　　所以我们不要企图用malloc/free来完成动态对象的内存管理，应该用new/delete。由于内部数据类型的“对象”没有构造与析构的过程，对它们而言malloc/free和new/delete是等价的。　　既然new/delete的功能完全覆盖了malloc/free，为什么C++不把malloc/free淘汰出局呢？这是因为C++程序经常要调用C函数，而C程序只能用malloc/free管理动态内存。　　如果用free释放“new创建的动态对象”，那么该对象因无法执行析构函数而可能导致程序出错。如果用delete释放“malloc申请的动态内存”，结果也会导致程序出错，但是该程序的可读性很差。所以new/delete必须配对使用，malloc/free也一样。

内存耗尽怎么办

如果在申请动态内存时找不到足够大的内存块，malloc和new将返回NULL指针，宣告内存申请失败。通常有三种方式处理“内存耗尽”问题。　　(1). 判断指针是否为NULL，如果是则马上用return语句终止本函数。例如：

```
void Func(void){
    A *a = new A;
    if(a == NULL)
        return;
    …
}
```

(2). 判断指针是否为NULL，如果是则马上用exit(1)终止整个程序的运行。例如：

```
void Func(void){
    A *a = new A;
    if(a == NULL){
        cout << “Memory Exhausted” << endl;
        exit(1);
    }
    …
}
```

(3). 为new和malloc设置异常处理函数。例如Visual C++可以用_set_new_hander函数为new设置用户自己定义的异常处理函数，也可以让malloc享用与new相同的异常处理函数。详细内容请参考C++使用手册。　　上述 (1)、(2) 方式使用最普遍。如果一个函数内有多处需要申请动态内存，那么方式 (1) 就显得力不从心（释放内存很麻烦），应该用方式 (2) 来处理。　　很多人不忍心用exit(1)，问：“不编写出错处理程序，让操作系统自己解决行不行？”　　不行。如果发生“内存耗尽”这样的事情，一般说来应用程序已经无药可救。如果不用exit(1)把坏程序杀死，它可能会害死操作系统。道理如同：如果不把歹徒击毙，歹徒在老死之前会犯下更多的罪。　　有一个很重要的现象要告诉大家。对于32位以上的应用程序而言，无论怎样使用malloc与new，几乎不可能导致“内存耗尽”。对于32位以

上的应用程序，“内存耗尽”错误处理程序毫无用处。这下可把Unix和Windows程序员们乐坏了：反正错误处理程序不起作用，我就不写了，省了很多麻烦。　　必须强调：不加错误处理将导致程序的质量很差，千万不可因小失大。

```
void main(void){
    float *p = NULL;
    while(TRUE){
        p = new float[1000000];
        cout << “eat memory” << endl;
        if(p==NULL)
            exit(1);
    }
}
```

malloc/free的使用要点

函数`malloc`的原型如下：

```
void * malloc(size_t size);
```

用`malloc`申请一块长度为`length`的整数类型的内存，程序如下：

```
int *p = (int *) malloc(sizeof(int) * length);
```

我们应当把注意力集中在两个要素上：“类型转换”和“sizeof”。　　`* malloc`返回值的类型是`void*`，所以在调用`malloc`时要显式地进行类型转换，将`void *`转换成所需要的指针类型。

`* malloc`函数本身并不识别要申请的内存是什么类型，它只关心内存的总字节数。我们通常记不住`int`，`float`等数据类型的变量的确切字节数。例如`int`变量在16位系统下是2个字节，在32位下是4个字节；而`float`变量在16位系统下是4个字节，在32位下也是4个字节。最好用以下程序作一次测试：

```
cout << sizeof(char) << endl;
cout << sizeof(int) << endl;
cout << sizeof(unsigned int) << endl;
cout << sizeof(long) << endl;
cout << sizeof(unsigned long) << endl;
cout << sizeof(float) << endl;
cout << sizeof(double) << endl;
cout << sizeof(void *) << endl;
```

在`malloc`的“()”中使用`sizeof`运算符是良好的风格，但要当心有时我们会昏了头，写出`p = malloc(sizeof(p))`这样的程序来。　　函数`free`的原型如下：

```
void free( void * memblock );
```

为什么free函数不象malloc函数那样复杂呢？这是因为指针p的类型以及它所指的内存的容量事先都是知道的，语句free(p)能正确地释放内存。如果p是NULL指针，那么free对p无论操作多少次都不会出问题。如果p不是NULL指针，那么free对p连续操作两次就会导致程序运行错误。

new/delete的使用要点

运算符new使用起来要比函数malloc简单得多，例如：

```
int *p1 = (int *)malloc(sizeof(int) * length);
int *p2 = new int[length];
```

这是因为new内置了sizeof、类型转换和类型安全检查功能。对于非内部数据类型的对象而言，new在创建动态对象的同时完成了初始化工作。如果对象有多个构造函数，那么new的语句也可以有多种形式。例如：

```
class Obj{
    public :
    Obj(void); // 无参数的构造函数
    Obj(int x); // 带一个参数的构造函数
    …
}

void Test(void){
    Obj *a = new Obj;
    Obj *b = new Obj(1); // 初值为1
    …
    delete a;
    delete b;
}
```

如果用new创建对象数组，那么只能使用对象的无参数构造函数。例如：

```
Obj *objects = new Obj[100]; // 创建100个动态对象
```

不能写成：

```
Obj *objects = new Obj[100](1);// 创建100个动态对象的同时赋初值1
```

在用delete释放对象数组时，留意不要丢了符号'[]'。例如：

```
delete []objects; // 正确的用法
delete objects; // 错误的用法
```

后者有可能引起程序崩溃和内存泄漏。

Conclusion

- 越是怕指针，就越要使用指针。不会正确使用指针，肯定算不上是合格的程序员。
- 必须养成使用“调试器逐步跟踪程序”的习惯，只有这样才能发现问题的本质。

32. 四种强制转换

使用标准C++的类型转换符：static_cast、dynamic_cast、reinterpret_cast和const_cast。

1、static_cast

用法：static_cast (expression)

该运算符把expression转换为type-id类型，但没有运行时类型检查来保证转换的安全性。它主要有如下几种用法：

（1）用于类层次结构中基类和派生类之间指针或引用的转换

进行上行转换（把派生类的指针或引用转换成基类表示）是安全的

进行下行转换（把基类的指针或引用转换为派生类表示），由于没有动态类型检查，所以是不安全的

（2）用于基本数据类型之间的转换，如把int转换成char。这种转换的安全也要开发人员来保证

（3）把空指针转换成目标类型的空指针

（4）把任何类型的表达式转换为void类型

注意：static_cast不能转换掉expression的const、volitale或者__unaligned属性。

2、dynamic_cast

用法：dynamic_cast (expression)

该运算符把expression转换成type_id类型的对象。type_id必须是类的指针、引用或者void*；

如果type_id是类指针类型，那么expression也必须是一个指针，如果type_id是一个引用，那么expression也必须是一个引用。

dynamic_cast主要用于类层次间的上行转换和下行转换，还可以用于类之间的交叉转换。

在类层次间进行上行转换时，dynamic_cast和static_cast的效果是一样的；

在进行下行转换时，dynamic_cast具有类型检查的功能，比static_cast更安全。

```
class B{
public:
  int m_iNum;
  virtual void foo();
};
class D:public B{
public:
```

```
  char *m_szName[100];
};

void func(B *pb){
  D *pd1 = static_cast<D *>(pb);
  D *pd2 = dynamic_cast<D *>(pb);
}
```

在上面的代码段中，如果pb指向一个D类型的对象，pd1和pd2是一样的，并且对这两个指针执行D类型的任何操作都是安全的； 但是，如果pb指向的是一个B类型的对象，那么pd1将是一个指向该对象的指针，对它进行D类型的操作将是不安全的（如访问m_szName），而pd2将是一个空指针。

另外要注意：B要有虚函数，否则会编译出错；static_cast则没有这个限制。这是由于运行时类型检查需要运行时类型信息，而这个信息存储在类的虚函数表（关于虚函数表的概念，详细可见）中，只有定义了虚函数的类才有虚函数表，没有定义虚函数的类是没有虚函数表的。

另外，dynamic_cast还支持交叉转换，如下所示。

```
class A{
public:
  int m_iNum;

  virtual void f(){}
};
class B:public A{
};
class D:public A{
};

void foo(){
  B *pb = new B;
  pb->m_iNum = 100;
  D *pd1 = static_cast<D *>(pb); //compile error
  D *pd2 = dynamic_cast<D *>(pb); //pd2 is NULL
  delete pb;
}
```

在函数foo中，使用static_cast进行转换是不被允许的，将在编译时出错，而使用dynamic_cast转换则是允许的，结果是空指针。

3、reinterpret_cast

用法：reinterpret_cast (expression)

type-id必须是一个指针、引用、算术类型、函数指针或者成员指针。

它可以把一个指针转换成一个整数，也可以把一个整数转换成一个指针（先把一个指针转换成一个整数，在把该整数转换成原类型的指针，还可以得到原先的指针值）。

该运算符的用法比较多。

(static_cast .与. reinterpret_cast比较，见下面)

该运算符平台移植性比价差。

4、const_cast

用法：const_cast (expression)

该运算符用来修改类型的const或volatile属性。除了const 或volatile修饰之外， type_id和expression的类型是一样的。

常量指针被转化成非常量指针，并且仍然指向原来的对象；

常量引用被转换成非常量引用，并且仍然指向原来的对象；常量对象被转换成非常量对象。

volatile和const类型，举例如下所示。

```
class B{
public:
  int m_iNum;
}
void foo(){
  const B b1;
  b1.m_iNum = 100; //comile error
  B b2 = const_cast<B>(b1);
  b2. m_iNum = 200; //fine
}
```

上面的代码编译时会报错，因为b1是一个常量对象，不能对它进行改变； 使用const_cast把它转换成一个非常量对象，就可以对它的数据成员任意改变。注意：b1和b2是两个不同的对象。

5、比较

（1）dynamic_cast vs static_cast

```
class B {     ... };
class D : public B {     ...};
void f(B* pb){
  D* pd1 = dynamic_cast<D*>(pb);
  D* pd2 = static_cast<D*>(pb);
}
```

If pb really points to an object of type D, then pd1 and pd2 will get the same value. They will also get the same value if pb == 0. If pb points to an object of type B and not to the complete D class, then dynamic_cast will know enough to return zero. However, static_cast relies on the programmer's assertion that pb points to an object of type D and simply returns a pointer to that supposed D object.

即dynamic_cast可用于继承体系中的向下转型，即将基类指针转换为派生类指针，比static_cast更严格更安全。dynamic_cast在执行效率上比static_cast要差一些，但static_cast在更宽上范围内可以完成映射，这种不加限制的映射伴随着不安全性。static_cast覆盖的变换类型除类层次的静态导航以外，还包括无映射变换、窄化变换(这种变换会导致对象切片,丢失信息)、用VOID*的强制变换、隐式类型变换等...

（2）static_cast vs reinterpret_cast

reinterpret_cast是为了映射到一个完全不同类型的意思，这个关键词在我们需要把类型映射回原有类型时用到它。我们映射到的类型仅仅是为了故弄玄虚和其他目的，这是所有映射中最危险的。(这句话是C++编程思想中的原话) static_cast 和 reinterpret_cast 操作符修改了操作数类型。它们不是互逆的；static_cast 在编译时使用类型信息执行转换，在转换执行必要的检测(诸如指针越界计算, 类型检查). 其操作数相对是安全的。另一方面；reinterpret_cast 仅仅是重新解释了给出的对象的比特模型而没有进行二进制转换， 例子如下：

```
int n=9;
double d=static_cast < double > (n);
```

上面的例子中, 我们将一个变量从 int 转换到 double。 这些类型的二进制表达式是不同的。 要将整数 9 转换到 双精度整数 9，static_cast 需要正确地为双精度整数 d 补足比特位。其结果为 9.0。

而reinterpret_cast 的行为却不同:

```
int n=9;
double d=reinterpret_cast<double & > (n);
```

这次, 结果有所不同. 在进行计算以后, d 包含无用值. 这是因为 reinterpret_cast 仅仅是复制 n 的比特位到 d, 没有进行必要的分析. 因此, 你需要谨慎使用 reinterpret_cast.

引自：http://www.cppblog.com/lapcca/archive/2010/11/30/135081.aspx

补充：

（1）static_cast：在功能上基本上与C风格的类型转换一样强大，含义也一样。它有功能上的限制。例如，你不能用static_cast像用C风格转换一样把struct转换成int类型或者把double类型转换成指针类型。另外，static_cast不能从表达式中去除const属性，因为另一个新的类型转换符const_cast有这样的功能。

可以静态决议出类型的转换可能性，即使是在继承体系中，即使包括了多重继承和虚继承，只要可以进行静态决议就可以转换成功

（2）const_cast：用于类型转换掉表达式的const或volatile属性。通过使用const_cast，你向人们和编译器强调你通过类型转换想做的只是改变一些东西的constness或者volatieness属性。这个含义被编译器所约束。如果你试图使用const_cast来完成修改constness或者volatileness属性之外的事情，你的类型转换将被拒绝。

（3）dynamic_cast：它被用于安全地沿着类的继承关系向下进行类型转换。这就是说，你能用dynamic_cast把指向基类的指针或引用转换成指向其派生类或其兄弟类的指针或引用，而且你能知道转换是否成功。失败的转换将返回空指针（当对指针进行类型转换时）或者抛出异常（当对引用进行类型转换时）。

（4）reinterpret_cast：使用这个操作符的类型转换，其转换结果几乎都是执行期定义。因此，使用reinterpret_cast的代码很难移植。reinterpret_casts的最普通的用途就是在函数指针类型之间进行转换。

举例如下：

```
#include <iostream>
using namespace std;
class A {
public:
  virtual void foo()     {     }
};
class B {
public:
  virtual void foo()     {     }
};
class C : public A , public B {
public:
  virtual void foo() { }
};

void bar1(A pa) {
  B *pc = dynamic_cast<B>(pa);
}
void bar2(A pa) {
  B *pc = static_cast<B>(pa); //error
}
void bar3() {
  C c;
  A pa = &c;
  B *pb = static_cast<B>(static_cast<C*>(pa));
}
int main(){
  return 0;
}
```

A、bar1无法通过编译 B、bar2无法通过编译 C、bar3无法通过编译 D、bar1可以正常运行，但是采用了错误的cast方法 解答： 选B。 dynamic_cast是在运行时遍历继承树，所以，在编译时不会报错。但是因为A和B没啥关系，所以运行时报错(所以A和D都是错误的)。 static_cast：编译器隐式执行的任何类型转换都可由它显示完成。其中对于：（1）基本类型。如可以将int转换为double(编译器会执行隐式转换)，但是不能将int*用它转换到double*（没有此隐式转换）。（2）对于用户自定义类型，如果两个类无关，则会出错（所以B正确），如果存在继承关系，则可以在基类和派生类之间进行任何转型，在编译期间不会出错。所以bar3可以通过编译（C选项是错误的)。