**C++内存分配机制**

内存主要分为如下5个存储区：

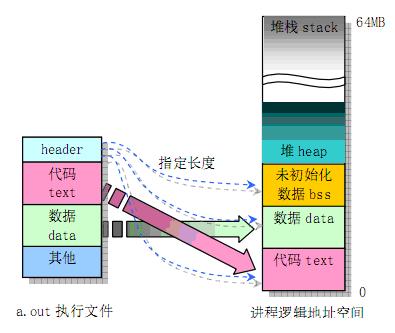
**静态存储区** (Global Static Area)： 在编译时分配，保存全局变量和静态变量。

**栈** (Stack)：保存函数内的局部变量（包括函数实参），函数结束时由编译器负责分配释放。向低地址扩展。

**堆**（Heap)：由new或malloc申请的内存，由程序员负责释放。向高地址扩展。

代码段：存放程序执行代码。

未初始化数据段：存放程序中未初始化的全局变量。



**new、delete、malloc、free**

new和delete对应，malloc和free对应

new和delete是运算符，malloc和free 是库函数

都可用于申请动态内存和释放内存，区别 new 不止是分配内存，而且会调用类的构造函数，delete会调用类的析构函数；而malloc则只分配内存，不会进行初始化类成员的工作，free也不会调用析构函数。

申请内存空间后一定要释放，且置指针为NULL。

**delete与 delete []**

delete调用一次析构函数 ，释放一个对象内存， delete []会调用多次析构函数，释放整个数组的内存。

delete和new对应，delete []和new []对应

**内存泄漏**

程序没有释放已经不再使用的内存，失去对这段内存的控制。

主要是两种类型的内存泄漏：

1. **堆内存泄漏**。 这部分是在程序运行中通过 new、malloc 在堆上申请分配的内存，如果在使用之后没有调用对应的delete、free释放的话，就会产生堆内存的泄漏。
2. **系统资源的泄漏**。程序中使用系统分配的资源，比如文件描述符、socket，用完之后没有用相应的函数释放，也会造成资源的浪费。

解决方法：

1. 良好的编程习惯。申请函数和释放函数成对。
2. 可以考虑使用智能指针。

**内存溢出**

**程序在申请内存的时候，没有足够的内存空间供其使用。**

**虚函数与纯虚函数**

**虚函数**：在基类中用virtual 关键字修饰，并在派生类中重新定义的成员函数。因为函数的调用并不是在编译时刻被确定的，而是在运行时刻被确定的。

作用是借助指针或者引用来实现多态。  
静态成员函数（没有this指针）、内联函数、构造函数不能是虚函数；   
在派生类中重新定义虚函数的参数个数和参数的类型与基类中相同；

虚函数指针：类对象的前四个字节，指向虚函数表。

构造函数和析构函数中调用虚函数：虚函数此时变成普通的函数，没有多态性质。因为派生类初始化时先调用基类构造函数，再初始化派生部分。因此在调用基类构造函数时，派生类对象还没有生成。

**纯虚函数**：在类的成员函数声明的后面加 =0 。

含有纯虚函数的类叫抽象类，作用是定义接口。其纯虚函数的实现由派生类给出。如果派生类没有重新定义纯虚函数，则这个派生类仍然还是一个抽象类。

不能创建一个抽象基类的对象。

**封装、继承**、**多态**

封装：将客观事物抽象成类，每个类对自身的数据和方法实行保护。封装之后，类的接口和实现分离，隐藏了类的实现细节。

**继承**：继承是指这样一种能力：使用现有类的所有功能的同时，实现类功能的扩展。

通过继承联系在一起的类构成一种层次关系，层次关系的根部叫做基类或父类，通过从基类直接或间接继承而来的类叫做派生类或子类。

基类定义所有类共同拥有的成员，每个派生类定义各自特有的成员。

继承的实现方式：

实现继承：同时继承函数的接口和实现。

接口继承：只继承函数的接口，子类提供实现。由虚函数或纯虚函数实现。

继承的方式（对用户代码来说）：

public（公有）：对基类访问控制不变

protected（受保护）：基类public 变成protected，其他不变

private（私有）：全部变成private

**多态**：通俗地讲同一操作作用于不同的对象，可以有不同的解释，产生不同的执行结果。

多态分为两种，一种是编译时的多态，一种是运行时的多态。

动态多态：当使用基类的指针或引用调用一个虚函数，根据所绑定的对象类型的不同调用不同的方法。如果绑定到基类对象，则执行基类的虚函数，如果绑定的是派生类对象，则执行派生类的虚函数。

多态实现有两个要素：第一是虚函数，第二是基类的指针或引用。

静态多态：一般是通过函数和运算符重载实现的，在编译时就确定。

**多态的作用**

隐藏实现细节，使得代码模块化；

扩展代码模块，实现代码重用；

接口重用

**动态绑定和静态绑定比较：**

静态多态

优点：在编译期完成的，效率较高；可以进行优化。

动态多态

优点：实现与接口分离，可复用

缺点：运行时绑定，增加运行时的开销；无法对虚函数进行优化。

**函数重载、函数覆盖与函数隐藏**

**重载**：相同作用域中声明多个名称相同，但参数个数或参数类型不同的函数。  
**覆盖：**指子类重新定义基类的虚函数，形参类型必须和基类函数完全一致。返回类型一般也必须匹配，一个例外是：当返回类型是类本身的指针或引用。  
**隐藏**：是指子类重新定义基类的非虚函数，派生类的名字将隐藏基类的名字。

不能重载的运算符：**.\*** 、 **::** 、**sizeof** 、**? :**

**构造函数**

构造函数不能定义为虚函数：构造一个对象的时候，必须知道对象的实际类型，而虚函数是在运行时确定实际类型的，因此编译器无法知道对象的实际类型。虚函数的调用是通过虚函数指针指向的虚函数表来实现调用的，但是调用构造函数时，虚函数表还没有被初始化，就不能调用虚函数。

**必须通过初始化列表初始化的成员：const成员变量，引用类型，没有默认构造函数的类类型（性能）。**

**派生类构造函数与析构函数**

派生类对象不能直接初始化基类的成员，而是通过调用基类的构造函数来初始化基类的成员。

构造函数的调用次序：先调用基类的构造函数（一般是拷贝构造函数），然后再初始化派生类的成员。

**基类应该定义一个虚析构函数**：基类的指针或引用可以绑定到动态分配的派生类对象，在释放对象时，需要调用派生类的析构函数，否则造成内存泄漏。

析构函数的调用次序：先执行派生类的析构函数，然后是基类的析构函数（会自动执行）

如果构造函数或析构函数中调用虚函数，则应该执行与构造函数或析构函数所属类型相对应的虚函数版本。

**拷贝构造函数**

三种调用拷贝构造函数的情况：

1. 用一个对象去初始化同一个类的另一个新对象；
2. 函数调用时，形参和实参的结合
3. 函数返回值为对象时

浅拷贝和深拷贝

浅拷贝是指在对象赋值时，只对对象中的数据成员进行简单的赋值，如果存在动态成员（指针等），两个对象的动态成员指向了同一个地址。

深拷贝：如果存在动态成员（指针等），则拷贝动态成员所指向的内容。

**拷贝构造函数和赋值操作符的区别：**

拷贝构造函数是构造函数，而赋值操作符属于操作符重载；

拷贝构造函数没有返回值，而赋值操作符一般返回类的引用；

拷贝构造函数是用一个对象初始化一个新对象，而赋值操作符是作用于两个已经初始化的对象（因为是两个已经存在的对象，所以要检查是否是自身赋值）；

**空类的默认生成的成员函数**

构造函数

拷贝构造函数

析构函数

赋值运算符

取地址运算符

**引用**

引用就是某个目标变量的别名,对引用的操作与对变量的直接操作效果完全相同；

申明一个引用的时候，要对其进行初始化；

引用申明完毕后，不能再把该引用名作为其他变量名的别名；

引用不是一种数据类型，不占存储单元。

操作符声明为引用的作用：

流操作符<<和>>，常常希望被连续使用，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。如果返回一个流对象，程序必须重新（拷贝）构造一个新的流对象，实际上是针对不同对象的。如果返回一个流指针则不能连续使用<<操作符。

拷贝构造函数的参数；

赋值操作符=的参数

+ - \* / 四则运算符，不能返回引用，只能返回一个对象。

**引用传递、值传递和指针传递**

引用传递和指针传递的效果是一样的，都可以改变实参。

引用传递没有实参拷贝，它是直接对实参变量操作

指针传递需要分配存储空间，进行实参拷贝；通过地址间接访问指向的变量。

值传递需要分配存储空间，进行实参拷贝；改变形参的值不会影响外部实参的值。

**智能指针shared\_ptr**

shared\_ptr实现技术是使用引用计数。将一个计数器与类指向的对象相关联，引用计数跟踪该类有多少个对象共享同一指针。

每次创建类的新对象时，初始化指针并将引用计数置为1；当调用拷贝构造函数时，拷贝指针并增加与之相应的引用计数；调用赋值操作符时，减少左操作数所指对象的引用计数（如果引用计数为减至0，则删除对象），并增加右操作数所指对象的引用计数；调用析构函数时，减少引用计数（如果引用计数减至0，则删除基础对象）。

shared\_ptr有两个成员变量：

T \* px：用于储存对象的指针；

unsign long \*pn：用于记录有多少个shared\_ptr共享同一个对象。

int \*a = new int(2);

shared\_ptr<int> sp(a);

shared\_ptr<int> sp2(sp1); //两个shared\_ptr指向

int \*b = sp1.get(); //获得shared\_ptr对象

sp1.unique(); //是否是唯一所有者

sp1.use\_count(); //引用计数个数（用于测试或调试）

**空指针和悬垂指针**

空指针是指被赋值为NULL的指针；delete指向动态分配对象的指针将会产生悬垂指针。

（1） 空指针可以被多次delete，而悬垂指针再次删除时程序会变得非常不稳定；

（2） 使用空指针和悬垂指针都是非法的，而且有可能造成程序崩溃，如果指针是空指针，尽管同样是崩溃，但和悬垂指针相比是一种可预料的崩溃。

结构与联合

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(所有成员公用一块地址空间)，而结构的所有成员都存在(不同成员的存放地址不同)

大小：结构体是所有成员相加，有内存对齐；

联合是成员的最大长度，有内存对齐；

枚举只占一个int大小。

**struct 和 class**

都可以用来定义类；

差别：默认访问权限和默认继承的权限不同，struct 的默认访问权限和默认继承权限为public， class 默认访问权限和默认继承权限为private。

struct不能用来申明模板。

**const 用法**

修饰常量

修饰指针或引用

修饰函数返回值

修饰函数：任何不会修改数据成员的函数都应该声明为const类型。（const对象只能访问const成员函数；const成员函数不能访问非const成员函数）

int t = 1;

const int a=1; //修饰常量，需要初始化

const int \*pa = &t; //\*pa为常量，即pa指向的内容不能变

int const \*pb = &t;//同上

int \*const pc = &t; //pc为常量，即pc不能指向其它地址

const int \* const pd = &t; //pd为常量，\*pd也为常量

int fun(const int &a); //顶层const

int fun2(const int a) const; //常成员函数，不能调用非const成员

**static**

面向过程的static，主要是c语言中static的用法：

静态全局变量：在全局变量前加static；只初始化一次；只能在声明它的文件当中可见，不能被其它文件使用。

静态局部变量：在局部变量前加static。保存在静态存储区；只初始化一次。

静态函数：在函数返回类型前加 static。只能在声明它的文件当中可见，不能被其它文件使用。

面向对象的static，主要是static在类中的作用：

静态成员函数：类内成员函数的声明前加static。static成员函数属于整个类所有，不包含this指针，只能访问类的static成员变量和static成员函数。

静态成员变量：类内数据成员的声明前加static。static成员变量属于整个类所有，对类的所有对象只有一份拷贝，且不能在类内初始化。

static 成员变量或函数的优点：不用创建对象而直接调用；

只占用一块内存空间，而非static会创建多个实例。

**explicit**

只能用于修饰只有一个参数的类构造函数。

修饰构造函数，当构造函数只有一个参数时，会有一个缺省的转换操作：将该参数的构造函数对应数据类型的数据转换为该类对象。

防止构造函数的隐式自动转换。

**volatile**

用来指示编译器不要对修饰的对象执行优化。提醒编译器它后面所修饰的变量随时都有可能改变，因此编译后的程序每次需要存储或读取这个变量的时候，都会直接从变量地址中读取数据。如果没有volatile关键字，则编译器可能优化读取和存储，可能暂时使用寄存器中的值。

**extern**

对函数或变量进行声明，表明在其他模块中已经定义过了，在这里要使用该函数或变量。只要包含模块的头文件即可。

extern “C” ：表示编译器在编译时按照C的编译和连接规则来进行链接。因为C++支持函数的重载，多个重载函数编译之后并不是原来的函数名。

作用：解决名字匹配的问题，实现C和C++混合编程。

static 和 extern 不能同时修饰一个变量或函数。

**multable**

const成员函数只能访问成员变量，不能修改成员。如果想在const成员函数中修改变量，则可以使用multable修饰。

mutable和const不能同时用于修饰成员变量。

**auto**

自动通过初值符推断声明的类型占位符。

**全局变量、局部变量和静态变量**

局部变量：在一个函数内部定义的变量，保存在堆栈区，作用域在函数范围内，生存周期函数运行期间。

全局变量：在函数外定义的变量，保存在静态存储区，作用域是整个文件，生存周期是整个程序运行期间。

静态变量：用static修饰，保存在静态存储区。定义在函数内是静态局部变量，定义在函数外是静态全局变量。

**define宏定义 和 const 常量**

define 和 const 都可以用来定义常量，差别：

define宏是在预处理阶段展开；const常量是编译运行阶段使用。

define宏没有类型，只做简单的字符替换，不做任何类型检查；const常量有数据类型，在编译阶段会执行类型检查。

const变量只有一个拷贝，而#define 宏变量有多个拷贝（每替换一次有一个拷贝）。

const 还可以用来修饰变量、指针、函数和函数返回值。

define最好每个变量用一个括号

宏定义表示一年中有多少秒

#define SECONDS\_OF\_YEAR (60 \* 60 \* 24 \* 365)UL

两个参数较小值

#define MIN(A,B) ((A) <= (B) ？ (A) : (B))

交换两个数

#define swap(a,b) (a)=(a)+(b);(b)=(a)-(b);(a)=(a)-(b);

**内联函数和宏定义**  
都是在使用时展开。

内联函数在运行时可调试，而宏定义不可以;  
编译器会对内联函数的参数类型做安全检查，而宏定义不会；   
内联函数可以访问类的成员变量，宏定义则不能；   
在类中声明并定义的成员函数，自动转化为内联函数。

但是每次内联函数的调用都要复制代码，将使程序的总代码量增大，消耗更多的内存空间。

不适合使用内联：函数体内的代码比较长，内存消耗大；函数体内出现循环，时间开销大。

内联函数用 inline 放在函数的定义前（注意不是声明，声明为inline不能成为内联函数）

**sizeof**

返回一个对象或者类型所占的内存字节数。

|  |  |  |  |
| --- | --- | --- | --- |
| 32位编译器 | | 64位编译器 | |
| 类型 | 字节数 | 类型 | 字节数 |
| char | 1 | char | 1 |
| short | 2 | short | 2 |
| int | 4 | int | 4 |
| unsigned int | 4 | unsigned int | 4 |
| float | 4 | float | 4 |
| double | 8 | double | 8 |
| long | 4 | long | 8 |
| unsigned long | 4 | unsigned long | 8 |
| long long | 8 | long long | 8 |
| \*p | 4 | \*p | 8 |

**结构体大小**

结构体的大小不是所有成员大小简单的相加，需要考虑内存对齐问题。一般遵循以下规则：

1. 第一个成员的首地址也即是结构体变量的首地址；
2. 每个成员的首地址相对于结构体的首地址的偏移量是该成员数据类型大小的整数倍；
3. 结构体的总大小必须是最大数据类型大小的整数倍。

如果用 #pragma pack(n) 指定了编译器的对齐系数n ，第2和3条要将对齐系数考虑进去。

char a[] 或 char a[0] 可以放在结构体最后，不占内存空间。

**sizeof 和 strlen**

sizeof 是操作符，strlen 是库函数；

sizeof 返回size\_t，它是unsigned int类型，strlen 返回 int 类型；

sizeof可以用来计算类型、函数、变量的字节数，strlen 只能计算字符数组的长度；

sizeof 编译时计算，strlen要在运行时才计算。

**类大小**

如果类中含有虚函数，则编译器需要为类构建虚函数表，所有的虚函数地址都存在这张表里，类中只需要一个指针指向虚函数表首地址即可。

类中的静态成员是被类所有实例所共享的，它不计入sizeof计算的空间

类中的普通函数或静态普通函数都存储在栈中，不计入sizeof计算的空间

类成员采用字节对齐的方式分配空间

因此只计算：非静态成员变量和虚函数指针（与虚函数个数无关）。

**空类 A 的大小 sizeof(A)= 1**  
空类可以被实例化。为了区分不同的实例，编译器在类中加的一个字节，这样每个实例就有唯一的地址。

**四种类型转换**

static\_cast 静态的，适合任何具有明确定义的类型转换。和C的强制类型转换功能相同。  
dynamic\_cast 动态的，将一个基类的指针或引用安全地转换为派生类的指针或引用。只用于指针或引用以及含有虚函数的类。空指针可以转换为所需类型的空指针。

B\* pb;

D\* pd,md;

pb = &md;

pd=dynamic\_cast<D \*>(pb);

reinterpret\_cast两种不同类型之间的转换，例如整型和指针的转换。  
const\_cast 改变对象的常量属性。const实参传到非const形参；const对象调用非const函数。但是对去掉const属性的对象写操作是未定义的。

**数组和指针**

数组下标和指针的偏移量相同。

函数参数或表达式中时，数组名和指针作用相同。数组会自动退化为同类型的指针。

用字符串字面值初始化，数组内容可以修改，而指针指向常量字符串，不能修改。char str[]=”abc”; //可修改

char \*p=”abc”; //不可修改

sizeof可以计算出数组所占字节大小，而作用于指针只能计算指针本身的大小。

**两个数的较大者**

( ( a + b ) + abs( a - b ) ) / 2

**不能被继承的类**

1. **将构造函数设为私有**

**通过定义公有静态函数来创建和释放类的实例。**

class A {

public:

static A \*GetA() {

return new A();

}

static void deleteA(A \* pA) {

delete pA;

}

private:

A() {}

~A() {}

**};**

1. 虚继承

template<typename T> class A {

friend T;

private:

A() {}

~A() {}

};

class B: virtual public A<B> {

public:

B() {}

~B() {}

};

类B是类A<B> 的友元，在B中可以调用 A<B> 的构造函数和析构函数。如果一个类继承class C: public B，调用类C的构造函数时，会跳过B而直接调用A<B>的构造函数，但是C不是A<B>的友元，不能调用私有构造函数。

**如何如何实现只能动态/静态分配类对象？**

静态定义建立类的对象，就是由编译器在栈上为类对象分配内存，是通过直接调用类的构造函数实现的。

动态定义类的对象，就是用 new 操作符在堆上为类对象申请内存空间，然后调用类的构造函数进行初始化。

如果不能静态定义类的对象，说明不能之间访问类的构造函数，将构造函数设为 private 或者 protected 就可以实现。然后定义一个静态成员函数，用于返回由 new 生成的类对象指针。

如果只能静态定义类的对应，将 new 操作符重载为 private 的就可以了。

**sprintf、strcpy 及 memcpy**

strcpy ：操作对象是两个字符串；

sprintf ：操作对象不限于字符串：源对象可以是任意基本类型的数据，主要用来实现任意基本类型向字符串的转换。

memcpy ：内存拷贝，实现将一个内存块的内容复制到另一个内存块。内存块由其首地址以及长度确定。通常用来对结构或者数组的拷贝。

**实现strcpy 函数**

char \* strcpy( char \*dst, const char \*src ) {

　　assert( (dst != NULL) &&(src != NULL) );

　　char \*ret = dst;

　　while( (\*dst++ = \* src++) != ‘\0’ );

　　return ret;

}

返回目的字符串的理由：能够增加字符串使用的灵活性，跟连续赋值和输入输出流的连续输出一样。

**实现strlen 函数**

int strlen( const char \*str ) {

　　assert( str != NULL ); //断言字符串地址非0

　　int len=0; //一定要初始化。

　　while( (\*str++) != '\0' )

　　 len++;

　　return len;

}

**实现memcpy函数**

void \*memcpy(void \*dst, const void \*src, size\_t count) {

assert((dst!=NULL)&&(src!=NULL));  
     char \*pdst=(char \*)(dst);  
     const char \*psrc=(const char \*)(src);  
    //目的地址和源地址重叠，从源地址的末尾方向开始拷贝  
     if( pdst>psrc && pdst<psrc + count ) {  
         pdst = pdst + count-1;  
         psrc = psrc + count-1;  
         while(count--) {   
             \*pdst-- = \*psrc--;  
        }  
    }

    //目的地址和源地址不重叠，从源地址的开始方向拷贝  
    else {  
        while(count--) {  
             \*pdst++ = \*psrc++;  
        }  
    }  
    return dst;  
}

**实现 String 类**

class String {

public:

       String(const char \*str = NULL);  //构造函数

       String(const String &other);  //拷贝构造函数

       ~ String();  //析构函数

       String & operater =(const String &rhs);  // 赋值操作符重载

private:

       char \*m\_data; // 用于保存字符串

};

String::String(const char \*str) {

       if ( str == NULL )  //当初始化字符串为空，为m\_data申请空间存放 ’\0’

      {

              m\_data = new char[1] ;

              m\_data[0] = '\0' ;

       }

       else { //为m\_data 申请同样大小的空间

              m\_data = new char[strlen(str) + 1];

              strcpy(m\_data, str);

       }

}

String::String(const String &other) {

       m\_data = new char[strlen(other.m\_data) + 1];

       strcpy(m\_data, other.m\_data);

}

String& String::operator =(const String &rhs) {

       if ( this == &rhs) //地址相同

              return \*this ;

       delete []m\_data; //删除原来的数据，新开一块内存

       m\_data = new char[strlen(rhs.m\_data) + 1];

       strcpy(m\_data, rhs.m\_data);

       return \*this ;

}

String::~String() {

       delete []m\_data ;

}

**STL**

**常用的容器**

**vector、deque、list、string**

**map、set**

**stack、queue**

**vector**

内部结构：数组

随机访问每个元素。可以动态增加或删除元素，自动完成内存管理。当元素个数超过容量时，会重新进行内存分配，分配的空间是原容量的两倍。

**deque**

内部数据结构：数组

双向队列，随机访问每个元素。可以方便地在开头和末尾增加、删除元素。

**list**

内部结构：双向循环链表

不支持元素的随机访问。可以双向遍历。可以在常数时间在任何地方增加、删除元素。

**string**

**map**

内部结构：红黑树

键值对，键唯一，默认按照键的升序排列。

**set**

按照键进行排序存储

**stack**

**queue**

**C语言连接mysql**

|  |  |
| --- | --- |
| MYSQL m\_conn | 声明一个MYSQL对象 |
| mysql\_init(&m\_conn) | 初始化mysql |
| mysql\_real\_connect | 连接到数据库 |
| mysql\_query | 执行字符串SQL语句，以‘\0’结尾 |
| mysql\_real\_query | 同上。多一个长度参数，以长度为结尾 |
| MYSQL\_ROW row | 行数据对象 |
| MYSQL\_RES \*res | 查询结果 |
| mysql\_store\_result | 返回查询结果集 |
| row=mysql\_fetch\_row | 返回结果集的下一行 |
| mysql\_close | 关闭连接 |

#include "mysql.h"

MYSQL m\_conn;  
int ret;  
     
 mysql\_init(&m\_conn);  
 if(mysql\_real\_connect(&m\_conn, "localhost","root","\*\*\*\*\*\*","Jack",0,NULL,0)) {  
  //insert  
  ret = mysql\_query(&m\_conn, "insert into t1 values(10,'Tony')");

  //delete  
  char\* Data = "delete from t1 where a=3";  
        res = mysql\_real\_query(&m\_conn, deleteData,(unsigned int)strlen(Data));

//show data  
        MYSQL\_ROW row;  
  MYSQL\_RES \*res;  
  int t;  
  char \*allData = "select \* from t1";  
  ret = mysql\_real\_query(&m\_conn, allData, (unsigned int)strlen(allData));  
  res = mysql\_store\_result(&m\_conn); //  
     while(row=mysql\_fetch\_row(res)) {  
    for(t=0; t<mysql\_num\_fields(res); t++)   
   printf("%s\t",row[t]);  
    printf("\n");

}

}  
mysql\_close(&m\_conn);

**大端模式与小端模式**

大端模式：指数据的低位保存在内存的高地址中，而数据的高位保存在内存的低地址中。网络字节序一般是指大端（对大部分网络传输协议而言）传输。

小端模式：指数据的低位保存在内存的低地址中，而数据的高位保存在内存的高地址中。Intel x86和ARM是小端模式。

**为什么会有大小端模式之分呢？**

在计算机系统中，是以字节为单位的，每个地址对应一个字节。对于多字节的类型，两种模式存放的方式相反。

小端模式 ：强制转换数据不需要调整字节内容。

大端模式 ：符号位的判定固定为第一个字节，容易判断正负。

**如何判断？**

1. 通过short强制类型转换成char单字节，通过判断起始存储位置；

short int x;

    char x0,x1;

    x=0x1122;

    x0=((char \*)&x)[0];  //低地址单元，如果为0x22，小端模式

    x1=((char \*)&x)[1];  //高地址单元

1. union的存放顺序是所有成员都从低地址开始存放。

union NUM {

       int a;

       char b;

   }num;

   num.a = 0x1122;

   if(num.b == 0x22) //低地址单元，如果为0x22，小端模式