1.C++中有哪4个和类型转换相关的关键字?这些关键字都有什么特点?应该在哪些场合下使用.

C语言中的强制类型转换可以随意的转换我们想要的类型了,格式如下(类型) 变量名;

为什么c++还要引入新的4种类型转换呢?

这是因为新的类型转换控制符可以很好的控制类型转换的过程,允许控制各种类型不同的转换.

还有一点好处是C++的类型转换控制符能告诉程序员或读者我们这个转换的目的是什么,我们只要看一下代码的类型转换控制符,就能明白我们想要达到什么样的目的.

1)static_cast <T*> (content) 静态转换.在编译期间处理

它主要用于C++中内置的基本数据类型之间的转换.但是没有运行时类型的检测来保证转换的安全性.

为什么需要static_cast类型的转换?

a.用于基类和子类之间的指针或引用的转换。这种转换把子类的指针或引用转换为基类表示是安全的;

进行下行转换,把基类的指针或引用转换为子类表示时,由于没有进行动态类型检测,所以是不安全的;

b.把void类型的指针转换成目标类型的指针(不安全).

c.用于内置的基本的数据类型之间的转换.

d.把任何类型的表达式转换成void类型的.

注意:static_cast不会转换掉content的const,volatile,__unaligned属性

2)static_const<T*>(content):去常转换;编译时执行;

它主要作用同一个类型之间的去常和添加常属性之间的转换.不能用做不同的类型之间的转换.

它可以把一个不是常属性的转换成常属性的,同时它也可以对一个本是常属性的类型进行去常.

3)dynamic_cast<T*>(content) 动态类型转换;也是向下安全转型;是在运行的时候执行;

通常用于基类和派生类之间的转换.转换时会进行类型安全检查。

a.不能用于内置的基本数据类型之间的转换.

b.dynamic_cast转换成功的话返回的是类的指针或引用,失败返回null;

c.dynamic_cast进行的转换的时候基类中一定要有虚函数,因为只有类中有了虚函数,

才说明它希望让基类指针或引用指向其派生类对象的情况,这样才有意义.

这是由于运行时类型检查需要运行时类型的信息,而这些信息存储在虚函数表中.

d.在类的转换时,在类层次间进行转换的时候,dynamic_cast和static_cast进行上行转换的时候效果是一样的;但是在进行下行转换的时候,dynamic_cast会进行类型检查所以它更安全.它可以让指向基类的指针转换为指向其子类的指针或是其兄弟类的指针;

4)reinterpret_cast<T*>(content)重解释类型转换;

它有着和C风格强制类型转换同样的功能;它可以转化任何的内置数据类型为其他的类型,同时它也可以把任何类型的指针转化为其他的类型;它的机理是对二进制数据进行重新的的解释,不会改变原来的格式,而static_cast会改变原来的格式;

2.c++中的一个空类,进行sizeof运算的结果是多少?c++中一个空类包含哪些成员?

C++中的一个空类,sizeof的运算结果是1.为什么不是0呢? 空类型不包含任何的信息,但是当我们声明该类型的实例的时候,它必须在内存中占用有一定的空间,否则无法使用这些实例,至于占用多少的内存,由编译器决定一般是1.

一个空类一般包括如下的成员函数?

默认的缺省构造函数	Empty(){}
默认的析构函数	~Empty(){}
拷贝构造函数	Empty(const Empty& empty){}
赋值运算符函数	Empty& operator=(const Empty& empty){}
取址运算符函数	Empty* operator&(){}
const属性的取址运算符函数	const Empty* operator&() const{}

3.构造函数,缺省构造函数;拷贝构造函数,缺省的拷贝构造函数;析构函数,缺省的析构函数;之间的区别?

1)构造函数存在的必要性: 因为C++程序设计希望像使用基本的数据类型那样使用类;而一般的时候类的数据成员都是私有的,在外部不能访问;所以它要定义一个构造函数来确保它实例化的对象进行

2)什么时候会调用构造函数?

创建对象的时候会被自动调用,且仅被调用一次.对象定义语句,new操作符.

作用:为成员变量赋初值,分配资源,设置对象的初始状态;

对象的创建过程:

step1:为整个对象分配内存空间

step2:以构造实参调用构造函数包括下面活动

构造基类部分(先构造基类的部分)

构造成员变量(如果有类类型成员变量的时候会先去执行类类型的构造函数)

执行构造代码(这个是自己写的构造函数)

3)缺省构造函数都做什么事情?什么时候才会使用缺省构造函数?

当我们没有提供任何的构造函数的时候,系统会提供一个缺省构造函数.

缺省的构造函数做的事情:

a.对基本的成员变量,不做初始化.

b.对类类型的成员变量和基类子对象,调用相应类型的缺省构造函数初始化

c.对于已经定义至少一个构造函数的类,无论其构造函数是否带有参数,编译器都不会为其提供缺省构造函数;

缺省的构造函数也叫无参构造函数,但是未必真的没有任何参数,为一个有参构造函数的每个参数提供一个缺省值,同样可以达到无参构造函数的效果;

4)什么是拷贝构造函数?默认的缺省拷贝构造函数都做什么事情?拷贝构造函数的时机?

a.对于普通的内置类型进行复制的时候,是很简单的.例如int a = 4;int b = a;

由于用户自己定义的类类型复制的时候,就要调用拷贝构造函数.它的参数必须是const 类名& that的构造函数,用于从一个已经定义的对象构造其同类型的副本,即对象的克隆;

b.对于缺省的拷贝构造函数它所做的就是浅拷贝;(一般没有指针成员的时候可以使用)

它主要做的事情就是对基本的成员变量,它会逐字节的复制;

对类类型成员变量和基类子对象,调用相应类型的拷贝构造函数.

c.拷贝构造函数的时机:

1.用已经定义的对象作为同类型对象的构造实参;(S s1 = s2这是一个特例.)

2.以对象的值的形式向函数传递参数.

3.从函数中返回一个对象的值.

4.某些拷贝构造过程会因编译优化而被省略.

5)什么是赋值运算符函数?什么时候会用到赋值运算符函数?赋值运算符编写的时候要要注意的事项?

a.A& operator=(const A& a);

b.当用一个已经存在的对象赋值给另外一个已经存在的对象时,会调用赋值运算符函数

c.缺省方式的拷贝构造和拷贝赋值,对包括指针在内的基本类型成员变量按字节赋值,导致浅拷贝问题(只拷贝地址,没有拷贝内容);为了获得完整意义上的对象的副本,必须自己定义拷贝构造函数和拷贝赋值,针对指针型成员变量做深拷贝;

d.相对于拷贝构造,拷贝赋值需要做更多的工作;

避免自赋值

分配新资源

拷贝新内容

释放旧资源

返回自引用

尽量复用拷贝构造函数和析构函数中的代码;

->拷贝构造:分配新资源,拷贝新内容;

->析构函数:释放旧资源;

6)拷贝构造和拷贝赋值总结;

a.无论是拷贝构造还是拷贝赋值,其缺省实现对类类型成员变量和基类子对象,对会调用相应类型的拷贝构造函数和拷贝赋值运算符函数,而不是简单的按字节复制,因此应当尽量避免使用指针型成员变量;

也能减少拷贝构造的机会.
c.处于具体的原因的考虑,确实无法实现完整意义上的拷贝构造和拷贝赋值的时候,
将它们私有化,以防误用.
d.如果一个类提供了自定义的拷贝构造函数,就没有理由不提供实现相同逻辑
的拷贝赋值运算符函数.
7)析构函数;
析构函数每个类成员只有一个,值在主函数执行完之后再执行的.析构函数
不能重载;
在销毁对象时自动被调用,且仅被调用一次;
对象离开所用域的时候会调用析构
delete操作符也会调用析构
释放在对象的构造过程或声明周期内获得的资源
如果一个类没有定义析构函数的时候,那么编译器会为其提供
一个缺省的析构函数;功能如下:
对基本类型的成员变量,什么也不做;
对类类型的成员变量和基类子对象,调用相应类型的析构函数;
析构函数的执行调用顺序:(对象的销毁过程)
a->执行析构代码;
b->析构成员变量
c->析构基类部分
d->释放整个对象所占用的内存空间
析构和构建的过程是完全相反的,构造的时候一定是先构造
基类的部分再来构造自己的部分;
4.已知String类的原型是:

```
class String
{
        public:
          String(const char* str = NULL);    //普通的构造函数
              String(const String& that);   //拷贝构造函数
              ~String(void);//析构函数
              String& operator=(const String& that);//赋值函数
              const char* c_str(void) const;  //完成和C的字符串兼容;
       private:
              char*    m_str;
};
```

```
#include <iostream>
#include <cstring>
using namespace std;
class String
{
public:
    /*一个简写的String(const char* str == NULL)
                {
                    m_str(strcpy((new
char[strlen(str?str:"")+1]),str?str:""));
                }
     */
```

```
{
    if(str == NULL)
    {
        m_str = new char[1];
        *m_str = '\0';
    }
    else
    {
        int length = strlen(str);
        m_str = new char[length+1];
        strcpy(m_str,str);
    }
}
~String(void)
{
    if(m_str)
    {
        delete[] m_str;
        m_str = NULL;
    }
}
/*拷贝构造函数*/


/*简单版本String(const String& that):m_str
  (strcpy(new char[strlen(that.str)+1],that.m_str))
 */
String(const String& that)
{
    m_str = new char[strlen(that.m_str) + 1];
    strcpy(m_str,that.m_str);
}
/*简单版本的赋值运算符函数
  String& operator=(const String& that)
  {
        if(&that != this)
        {
            String& tmep(that);
            swap(m_str,tmp.m_str);
        }
        return *this;
  }
 */
String& operator=(const String& that)
{
    //检查自赋值
    if(this == &that)
```

```
        //释放原有的内存资源
        delete[] m_str;
        //分配新的内存资源,并复制内容
        m_str = new char[strlen(that.m_str + 1)];
        strcpy(m_str,that.m_str);
        //返回本对象的引用
        return *this;
    }
    //保证向C语言字符串的兼容
    const char* c_str(void) const
    {
        return m_str;
    }
private:
    char* m_str;
};
int main(void)
{
    String s1 = "Hello,World!";
    String s2 = s1;
    cout << s1.c_str()<< endl;
    cout << s2.c_str()<< endl;
    String s3("hello,C++!");
    s2 = s3;
    cout << s1.c_str() << endl;
    cout << s2.c_str() << endl;
}
```

当写一个赋值运算符函数的时候,它会考察你下面几个方面的内容;
1)是否把返回值类型声明为该类型的引用,并在函数结束前返回实例自身的引用(即*this).只有返回一个引用,才可以
允许连续赋值.
2)是否把传入的参数的类型声明为常量引用.如果传入的参数不是引用而是实例对象,那么从行参到实参会调用
一次复制构造函数.把参数声明为引用可以避免这样的无谓的消耗,能提高代码的效率.同时,我们在赋值运算符
函数内不会改变传入的实例的状态,因此应该传入的引用参数加上const关键字.
3)是否释放实例自身已有的内存,如果我们忘记在分配新内存之前释放自身已有的空间.程序将内存泄漏.
4)是否判断传入的参数和当前的实例(*this)是不是同一个实例.如果是同一个,则不进行操作直接返回.
如果事先不判断就进行赋值,那么释放实例自身的内存的时候就会导致严重的问题;当*this和传入的参数是
同一个实例时,那么一番释放了自身的内存,传入参数的内存也同时被释放了,因此再也找不到需要赋值的
内容了

5.联合以及大小端模式:分析下段代码的打印结果?已知采用的是大端模式;

```
#include <stdio.h>
union
{
    int i;
    char x[4];
} a;
void main(void)
{

    a.x[0] = 10;//0000 1010 低位低地址

    a.x[1] = 1;//0000 0001 高位高地址

    a.x[2] = 0;
    a.x[3] = 0;
    printf("%#0x",a.i);  //0000 0000 0000 0000 0000 0001 0000 1010
}
```

分析:大端模式：低字节高地址；小端模式：低字节低地址;
而数组的首地址是低地址的,由于是大端模式,所以a.x[0]实际上是i的高位,值为a;
所以打印结果是0xa010000;
如果是小端模式的话:a.x[0]实际上是i的低位,值为a;
这个时候打印结果就是0x10a;

6>写一个判断函数,来判断系统是大端模式还是小端模式?

```
#include <stdio.h>
int checkSystem(void)
{
    union check
    {
        int i;
        char ch;
    }check;
    check.i = 1;
    return (check.ch == 1);
}
int main(void)
{
    int flag = 0;
    flag = checkSystem();
    if(!flag)
    {

        printf("系统是大端模式!\n");
    }
```

```
        printf("系统是小端模式!\n");
    return 0;
}
```

运用char和int的联合来判断,让i=1,如果这个时候取ch的值为1的话
就证明是小端模式,因为联合取的时候肯定取的低位,而低8位的值肯定是1
如果ch的值不是1就证明它是大端模式,其实这个时候它的结果是0;
7.关于strlen和sizeof的区别?
1.sizeof是一个运算符,而strlen是一个库函数,需要包含头文件<string.h>
2.sizeof的参数可以数据类型或变量,而strlen的参数只能是一个以'\0'结束
  的字符串.这里注意如果是字符串或字符数组里面有0字符的时候,它计算
  的是到第一个字符0为止的长度,并且不包括'\0'或字符0;
3.编译器在编译的时候就计算出了sizeof()的结果,而strlen函数必须是在运行
  的时候才计算出来.并且sizeof加算的是数据类型或数组所占用的内存的字节数,
  而strlen计算的是一个字符串到遇到0字符或'\0'的长度.
4.当sizeof以数组名当参数的时候,不会退化.而数组名作为strlen
  的参数的时候会退化为一个字符指针,指向的是字符串的首地址.
看下面的打印结果?

```
#include <stdio.h>
#include <string.h>

/*strlen在计算数组的长度的时候遇到0就会停止计算它的长度*/

int main(void)
{
    char arr[] = {1,2,0,3,4,0};
    printf("sizeof(arr) = %d,strlen(arr) = %d\n",sizeof(arr),strlen(arr));
    return 0;
}
```

这里的sizeof(arr) = 7没有疑问?唯一的疑点是strlen(arr)的值结果不是6而是2;
而如果写成 strlen("10111")的结果依然是5,从而可以得出结论是strlen只有在计算
字符数组的时候才会把字符0当成是'\0',而在字符串中好像不是.
再看一下下面的例子?这个例子也是典型的strlen(str)的例子.

```
#include <stdio.h>
#include <string.h>

int main(void)
{
    char buf[1000];
    int i = 0;
    for(i = 0;i < 1000;i++)
    {
        buf[i] = -1 - i;
```

```
    }
    printf("\n");
    printf("%d\n",strlen(buf));
    return 0;
}
```

求buf的长度,这里的buf[0] = -1;
当i = 127的时候buf[127] = -128; 而无符号的字符的取值范围是-128 到 127;
所以 buf[127 + 128] = 0;即是当i = 255的时候,buf[255] = 0;即是字符0
所以字符串的长度是 0-254 即是 255;

strcat/strcmp/strlen/strcpy 的实现

1.编写一个函数实现strlen以及strcpy函数.
strcpy函数.
后面的字符串拷贝到一个字符数组中,要求拷贝好的字符串在字符数组的首地址,并且只拷贝到'\0'的位置.原型是
char* my_strcpy(char* dest[],const char* src);

```
 1#include <stdio.h>
 2 #include <assert.h>
 3 char* my_strcpy(char *dest,const char* src)
 4 {
 5     assert(dest != NULL && src != NULL);
 6     char* p = dest;
 7     while(*src != '\0')
 8     {
 9         *p++ = *src++;
10     }12     return dest;
13 }
14 int my_strlen(const char* str)
15 {
16     int res = 0;
17     if(NULL == str)
18         return -1;
19     else
20         while(*str++ != '\0')  //注意这里的'\0'和0是相同的.
21             res++;
22     return res;
23 }
```

这两个函数的实现主要是判断空指针,其他的注意细节就是strcpy不要改变第一个参数的指向,还有就是返回字符串的
首地址以实现级联式应用.
下面写一个简化版本的更好.

```
char* other_strcpy(char* dest,const char* src)
{
    assert(dest != NULL && src != NULL);
    char *p = dest;
    while((*p++ = *src++) != '\0')
        NULL;
    return dest;
}
```

2.实现strcat和strcmp函数.
原型是:char* my_strcat(char* dest,const char* src);
int my_strcmp(const char* s1,const char* s2);

```
#include <stdio.h>
#include <assert.h>
int my_strcmp(const char* s1,const char* s2)
{
    assert(s1 != NULL && s2 != NULL);
    for(;*s1 == *s2;s1++,s2++);
    if(*s1 == '\0' && *s2 == '\0')
        return 0;
    return ((unsigned char)*s1 < (unsigned char)*s2 ? -1: 1);
}
```

```
char* my_strcat(char* s1,const char* s2)
{
    assert(s1 != NULL && s2 != NULL);
    char* dest = s1;
    while(*dest != '\0')
        dest++;
    while(*dest++ = *s2++);
    return s1;
}
```

字符串翻转/打印任意进制格式/类型转换

1.字符串的翻转,这里一般是字符数组.不包括字符串字面值.
char* reversal_str(char* str,size_t size);
翻转之后的字符串是原来的字符串的翻转.

```
#include <stdio.h>
#include <string.h>
char* reversal(char* str,size_t len)
{
    if(str != NULL)
    {
```

```
        char* end   = str + len - 1;
        char ch;

        while(start < end) //注意这里的条件不能是start != end;因为大小如果是偶数的
```

话,它们永远不会相等.

```
        {
            ch = *start;
            *start++ = *end;
            *end-- = ch;
        }
    }
    return str;
}
int main(void)
{
    char str[] = "abcdefg";
    size_t len = strlen(str);
    reversal(str,len);
    printf("%s\n",str);
    return 0;
}
```

这里有一定的局限性,因为这样只能是在原来的地址上进行字符串的翻转;如果是字符串字面值的情况,其就不能在原来的地址上
进行翻转.下面给出一个更一般的实现,不过它的思想不是在原来的字符串的地址上进行修改,而是重新分配了一块内存来进行
字符串的翻转;它的思想是得到一个原来的字符串的翻转之后的字符串,它返回的结果是一个新的地址.

```
#include <stdio.h>
#include <string.h>
#include <assert.h>
char* reversal_str(char* const dest,const char* src,size_t len)
{
    assert(dest != NULL && src != NULL);
    char* start = dest;
    char*   end = dest + len - 1;
    char ch;

    strcpy(dest,src);//把它拷贝到dest内存中.

    while(start < end) //注意这里不能写成start != end;
    {
        ch = *start;      //交换首尾字符
        *start++ = *end;
        *end-- = ch;
    }
```

```
}

int main(void)
{
    char *str1 = "abcdefg";
    char str2[] = "1234567";
    char str[100] = {};
    size_t len = strlen(str1);
    reversal_str(str,str1,len);
    printf("%s\n",str);
    len = strlen(str2);
    reversal_str(str,str2,len);
    printf("%s\n",str);
    return 0;
}
```

2.写一个函数,由用户输入一个十进制数和它想要打印这个数的进制数.函数的功能是把这个十进制数按照用户的要求打印出来,例如如果用户输入一个数100并且想要的进制是16进制则打印的结果是64;如果是100，进制要求是8,则结果是144

算法分析:

主要是先对进制求余数,然后是再除于这个base,直到num/base==0为止.

然后再反向打印即可.

```
#include <stdio.h>

void shift_base(int num,size_t base)
{
    int arr[32] = {},i = 0;
    do
    {
        arr[i++] = num % base;
    }while(num /= base);

    printf("它的%d进制的格式是:\n",base);

    for(num = i - 1;num >= 0;num--)
        arr[num] < 10?printf("%d",arr[num]):
            printf("%c",arr[num]-10+'A');
    printf("\n");
}
int main(void)
{
    int num = 0;
    size_t base = 0;

    printf("请输入一个整数:");

    scanf("%d",&num);
```

```
    scanf("%d",&base);
    shift_base(num,base);
    return 0;
}
```

3.设计一个函数,求1! + 2! + 3! + 100!的阶乘之和!
这个题有很多中解法,但是这里有一个巧用static的诀窍,我感觉很强大.
在这里和大家分享下.

```
#include <stdio.h>

long factial(int n)
{
    static long fac = 1;
    int i = 0;
    fac *= n;
    return fac;
}
long sum_fac(int n)
{
    int i;
    static long sum = 0;
    for(i = 1;i <= n;i++)
    {
        sum += factial(i);
    }
    return sum;
}
int main(void)
{
    int n;

    printf("请输入你要求的阶乘数:");

    scanf("%d",&n);
    printf("1!+2!+...%d! = %ld\n",n,sum_fac(n));
    return 0;
}
```

4.类型转换问题?先看题

```
#include <iostream>
#include <cstdio>
#include <cstring>
using namespace std;
int main(void)
{
```

```
    cout << (int)a << endl;
    cout << &a << endl;
    cout << (int&)a << endl;
    cout << boolalpha << ((int)a == (int&)a) << endl;
    float b = 0.0f;
    cout << (int)b << endl;
    cout << &b << endl;
    cout << (int&)b << endl;
    cout << boolalpha << ((int)b == (int&)b) << endl;
    return 0;
}
```

1)(int)a这里注意它和*(int*)&a是不同的,(int)a这里是相当于是取浮点数a的整数部分
所以结果是1.
2)&a打印的是a的地址.
3)(int&)a这里要注意了,这里相当于是打印*(int*)&a;这里的结果肯定不是1了,因为这里
相当于把地址&a里面的内容由浮点类型解释成一个整型输出了,因为float和整型的存
储的方式不同,所以打印的结果不是1.
4)根据上面的分析这里的打印结果是false;
5)由于0.0比较特殊,所以后面的打印结果是0,b的地址,0和true.

5.int和char以及char*之间的转换问题?
下面程序的打印结果.

```
#include <stdio.h>
int main(void)
{
    unsigned int a  = 0xFFFFFFF7;
    unsigned char i = (unsigned char)a;
    char* b = (char*)&a;
    printf("%08x,%08x\n",i,*b);
    return 0;
}
```

分析:
把一个unsigned int 赋值给unsigned char的时候高字节会被截断,只留下低8位.
即f7,打印的时候又是以整型数打印的所以结果是000000f7;
而char* b = (char*)&a;这里向当于让b指向了a的地址,但是a的内容并没有变化,
最后打印的时候又是按整型打印的所以结果是fffffff7;

6.用一个表达式,判断一个数X是否是2的N次方(2,4,8,16,...),不能使用循环语句.
分析:2的n次方就说明这个数的二进制表示只有一个1,如果这个数减去1之后,
则其前面的其他位的结果都是1.所以如果x&(x-1)的结果是0就表示它是2的N
次方.所以可以写成下面的表达式 !(X&(X-1)) 如果返回1表示它是,否则不是.

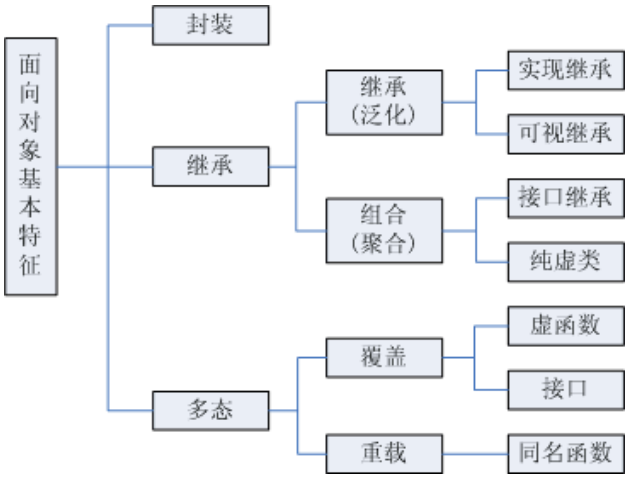
7.下面代码的结果

```
{
        return (x&y) + ((x^y)>>1)
}
fun(144,296) = _____?
```

这里主要考察一种整型数据的用位于求表达式的问题.
(x&y)相于表示的是x和y相同位的一半.(主要是有1的位)
(x^y)表示的是(x和y)不同位的值,右移一位相当于除以2.(主要是一个是1一个是0的位)
因为用二进制求数,其实我们关注的是位数是 1 的地方,其他是 0 的地方只是在填位.

c++重要的概念详解

1.C++面向对象的三大特征?



1)封装:将客观事物封装成抽象的类,并且设计者可以对类的成员进行访问控制权限控制.
这样一方面可以做到数据的隐藏,保护数据安全;另一方面,封装可以修改类的内部实现而不用修改调用了该类的用户的代码.同时封装还有利于代码的方便复用;
2)继承:a.继承具有这样一种功能,它可以使用现有类的所有功能;并且可以在不重新编写原有类的情况下对类的功能进行扩展.
继承的过程是一般到特殊的过程,即是它们是is-a的关系;
基类或父类是一般,而子类或派生类是基类的特殊表现;
要实现继承可以通过继承和组合来实现;
b.广义上的继承分成三大类:
实现继承:使用基类的属性和方法而无需额外编码的能力;
接口继承:接口继承是指仅使用基类的属性和方法的名称,而具体的实现子类必须自己完成的能力;
可视继承:子窗体(类)使用父窗体(类)的外观和实现代码的能力;
3)多态:a.多态的实现分成两种,一种是编译时的多态,主要是通过函数重载和运算符重载来实现的,是通过静态联编实现的;
另外一种是运行时多态,主要是通过函数覆盖来实现的,它需要满足3个条件:基类函数必须是虚函数,并且基类的指针
或引用指向子类的时候,当子类中对原有的虚函数进行重新定义之后形成一个更加严格的重载版本的时候,就会形成
多态;它是通过动态联编实现的;
b.运行时的多态可以让基类的指针或引用指向不同的对象的时候表现出来不同的特性;
2.简述C/C++程序编译时的内存分配情况
1)一般一个c/c++程序编译的时候内存布局如下(地址从低到高的顺序)
a.代码区:存放程序的二进制代码.
b.常量区:这个区和代码区的距离很近,主要存放一些非局部常量值和字符串字面值,一般不允许修改,程序结束由系统释放;
具有常属性并且初始化的全局和静态变量也放在这个区

c.数据区:赋过初值的且不具有常属性的静态和全局变量在数据区.它和BSS段统称为静态区;程序结束后由系统释放;
d.BSS段:没有初始化的静态和全局变量;进程一旦被加载这个区所有的数据都被清0;
e.堆区: 动态分配的内存;由程序员分配和释放,程序结束的时候如果没有释放,则由OS回收;
f.栈区: 由编译器自动分配和释放,不使用的时候会自动的释放.主要用来存放非静态的局部变量,函数的参数和返回值,
        临时变量等.
g.命令行参数和环境变量区;
下面是对应一段经典的代码:

```
#include <stdio.h>
#include <stdlib.h>

#include <unistd.h>  //unix下的头文件

const int const_global = 10; //常全局变量

int init_global = 10;        //初始化的全局变量

int uninit_global;           //未初始化的全局白能量


int main(int argc,char * argv[])
{
    const static int const_static = 10;//常属性的静态变量，不可以被赋值，初始化

    static int init_static = 10;    //初始化静态变量

    static int uninit_static;       //未初始化静态变量

    const int const_local = 10;    // 常属性的局部变量

    int  prev_local = 1;//前局部变量

    int  next_local = 5;//后局部变量

    int* prev_heap = malloc(sizeof(int));//前面分配的堆变量

    int* next_heap = malloc(sizeof(int));//后面分配的堆变量

    const char* literal = "literal";      //字符串字面值，字面值常量

    extern char** environ;              // 环境变量


    printf("----地址最高断命令行参数和环境变量-------\n");

    printf("        环境变量:%p\n",environ);
```

```
    printf("---------------栈区----------------------\n");
    printf("      常局部变量:%p\n",&const_local);
    printf("      前局部变量:%p\n",&prev_local);
    printf("      后局部变量:%p\n",&next_local);
    printf("--------------------堆--------------------\n");
    printf("       前堆变量:%p\n",prev_heap);
    printf("       后堆变量:%p\n",next_heap);
    printf("--------------------BSS--------------------\n");
    printf("未初始化全局变量:%p\n",&uninit_global);
    printf("未初始化静态变量:%p\n",&uninit_static);
    printf("----------------数据----------------------\n");
    printf("  初始化全局变量:%p\n",&init_static);
    printf("  初始化全局变量:%p\n",&init_global);
    printf("----------------代码区---------------------\n");
    printf("     常静态变量:%p\n",&const_static);
    printf("     字面值常量:%p\n",&literal);
    printf("     常全局变量:%p\n",&const_global);
    printf("          函数:%p\n",main);
    return 0;
}
```

2)从上面可以看出c/c++的内存分配方式主要有三种?

a.从静态存储区域分配:

内存在程序编译时已经分配好,这块内存在程序的整个运行期间都存在.

速度快,不容易出错.因为由系统会善后.

b.在栈上分配内存:

在执行函数的时候,函数内非静态局部变量的存储单元都是在栈上创建,函数执行结束的时候这些存储单元自动被释放.栈内存分配内置于处理器的指令集中,效率很高但是分配的内容有限.

c.从堆中分配内存:

即是动态分配内存.程序在运行的时候使用malloc/new申请任意大小的内存,程序员自己负责在何时用free/delete释放内存.动态内存的生存期由程序员决定,使用非常的

频繁的分配和释放不同大小的堆空间将会产生堆内碎片.不易管理;

3)堆和栈之间的主要的区别是什么?

a.管理方式不同:栈是由编译器自动分配和释放,使用方便;而对于堆来说,
分配和释放都必须由程序员来手动完成,不易管理,容易
造成内存泄漏和内存碎片.

b.可用内存空间不同:对于栈来说,它可用的内存空间比较小;而对于堆来说它可以使用的空间比栈要大的多.

c.能否产生碎片不同:由于栈采用的是后进先出的机制,所以栈空间没有内存碎片的产生;
而对于堆来说,由于频繁的使用new/delete势必会造成内存空间分配的不连续,从而
造成大量的碎片,使程序的效率降低.

d.生长方向不同: 对于堆来说,它一般是向上的;即是向着地址增加的方向增长;对于栈来说,它一般是向下的,
即向着地址减小的方向增长.

e.分配的方式不同:对于堆来说,它只能是动态分配的;而对于栈来说,它分为静态分配和动态分配;
静态分配由编译器来进行管理,而动态分配的栈和堆也是不一样的,动态分配的
栈由编译器进行释放,无需我们程序员来释放.

f.分配的效率不同: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:为栈分配专门的寄存器.
压栈和出栈都由专门的指令进行.因此它的效率会很高;
而堆则是由c/c++库函数实现的,机制是非常的负责的;例如要分配一块内存的时候,库
函数会利用特定的算法在堆内存中搜索可用大小的内存空间;如果没有足够大的内存
空间,就会调用系统功能去增加数据段的内存空间.这样才能得到足够大的可用的内存
空间,因此堆内存的分配的效率比栈要低得多.

4)new/malloc以及free/delete之间的区别?

a.new/delete是运算符,只能在C++中使用,它可以重载;mallloc/free是C的标准库函数,在C/C++中都可以使用.

b.对于非内部的数据类型的对象而言,光用malloc/free是无法满足动态对象的要求的.对象在创建的时候需要
执行构造函数,对象在消亡之前需要执行析构函数.而molloc/free是库函数而不是运算符,不在编译器的控制
范围之内,编译器不能将执行构造函数和析构函数的任务强加给malloc/free.因此C++需要一个能够完成动态
分配内存和初始化的new,以及一个能够完成清理和释放内存的运算符delete.

c.new的返回值是指定类型的指针,可以自动的计算所需要分配的内存大小.而malloc的返回值是一个void类型
的指针,我们在使用的时候要进行强制类型转换,并且分配的大小也要程序员手动的计算.

d.new/delete完全覆盖了malloc/free的功能,只所以还要保留malloc/free,是因为我们的C++程序有时要调用用C编写的
而C中又没有new/delete,只能使用malloc/free.

3.指针和引用之间的区别和联系?

联系:

a.指针和引用本质上都是地址的概念,引用在内部其实是用const指针来实现的.

b.给函数传递参数的时候,一级指针和引用作为函数参数的时候可以达到相同的效果.

c.指针的大部分效果都可以通过引用来实现

的引用可以达到同样的效果.
区别:
a.定义引用的时候必须初始化,定义指针的时候可以不初始化.
b.引用不能引用空,但是指针可以指向空.
c.引用的关系一旦确定,就无法改变;引用永远指向的是用来对它初始化的对象;而非常属性的指针是可以改变指向的.
d.指针是一个实体变量,在32位操作系统上面都是4个字节.而引用只是一个别名,其大小和其应用的对象的类型有关系.
e.有指向指针的指针,但是没有引用引用的引用;因为引用一旦建立,它就表示初始化它的对象.
f.有引用指针的引用,但是没有指向引用的指针;
g.有指针数组,但是没有引用数组,但是有数组的引用.
下面是一段代码非常的全面:

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

void foo(int a[3])/*这个地方传递的是数组的首地址*/

{
    cout << sizeof(a)/sizeof(a[0]) << endl;
}

void bar(int (&a)[3])/*这里传参的时候就是数组的整体*/

{
    cout << sizeof(a)/sizeof(a[0]) << endl;
}
int main(void)
{
    int a;
    int* p = &a;

    int** pp = &p;/*存在指向指针的指针*/

    int& r  = a;

    int&& rr = r;/*error没有引用引用的引用*/

    int*&  rp = p; /*有引用指针的引用(指针引用)*/

    int&* pr = &r; /*没有指向引用的指针(引用指针)*/

    int x, y, z;

    int* pa[] = {&x,&y,&z};/*指针数组*/

    int& ra[] = {x,y,z};/*引用数组是不存在的因为引用不是一个实体*/

    int arr[3] = {0};

    int (&ar)[3] = arr;/*数组引用(先近后远,先右后左)*/

    foo(arr);/*这里传递的是数组的第一个元素的首地址*/

    cout << sizeof(arr)/sizeof(arr[0]) << endl;
```

```
    /*这里的数组名代表的是真个数组*/

    int (*parr)[3] = &arr;/*对数组名取地址得到的是一个数组指针

                          这个时候arr代表的是数组的整体;*/

    bar(arr);/*这里传递的就是数组的整体*/

    return 0;
}
```

运算符重载详解

1.操作符函数:

在特定条件下,编译器有能力把一个由操作数和操作符共同组成的表达式,解释为对一个全局或成员函数的调用,该全局或成员函数被称为操作符函数.该全局或成员函数被称为操作符函数.通过定义操作符函数,可以实现针对自定义类型的运算法则,并使之与内置类型一样参与各种表达式运算.

2.首先我们先介绍下左值和右值,因为我们在运用运算符的时候要尽量和内置类型的一致性.

左值:有名的可以直接取地址的我们称之为左值,左值的特性是可以修改的.

右值:右值主要是一些临时变量,匿名变量,字符串字面值常量,字符常量;

表达式有的是左值有的是右值,例如+-*/运算返回的是右值,而赋值运算,复合赋值运算返回的是左值.前++返回的是左值,后++返回的是右值.

类型转换(无论是显示的还是隐式的)都伴随着临时变量的产生.函数的返回值当返回的不是引用的时候也是一个右值,但是一个引用的时候返回的是一个左值.

下面给出一个经典的全面左值和右值的示例:

```
#include <iostream>
int g = 0;
using namespace std;
int foo(void)
{
    return g;
}

int& bar(void)
{
    return g;
}

void func(int& i)
{
    cout << "func1" << endl;
}
void func(const int& i)
{
    cout << "func2" << endl;
}
```

```
int main(void)
{
    int i;  //左值,有名字可以取地址.
    int* p = &i;
    i = 10; //可修改
    //p = &10;这里的10是个右值,上面10赋值给i的时候产生的一个临时变量,是右值.所以10++也是错误的.

    // foo() = 10; 这里返回值也是临时变量是右值,是只读的属性,不能修改;++foo();也是错误的.

    bar() = 10; //这里返回的是一个左值引用,所以它是可以赋值的
    cout << g << endl;
    int a = 10,b = 20,c;
    c = a + b;//a + b是右值.
    //a + b = 10;是非法的,因为表达式的值也是用一个临时变量来存储的.是右值.
    //无论是显示的还是隐式的类型转换,都会产生临时变量.
    (a = b) = 10; //赋值表达式返回的是左值.
    ++a = 1000;   //前++表达式返回的是左值.
    //a++ = 2000;  这里是不行的,后++运算符的表达式的值是右值.
    //a++++;这也是不行的,对右值做++是不行的.
    ++++a;//这个是可以的.++a的返回值还是a本身.
    cout << a << endl; //1000
    char ch = 0;
    i = ch; //这里是可以的.
    //int& r = ch;它会先把ch转换成一个临时变量,然后把这个临时;变量赋值给r,而引用是不能引用一个临时变量的.
    //非常引用只能引用一个左值,不能是右值.
    const int& r = ch;  //常左引用又叫万能引用,它可以引用左值和右值.
```

```
    //所谓类型转换,无论是显示还是隐式的.都不是改变了原来的变量的类型,只能产生一个临时变量.
    return 0;
}
```

3.操作符重载的一般规则
a.C++不允许用户自己定义新的运算符,只能对已经存在的操作符进行重载.
b.C++大部分的运算符都可以重载,但是有一部分运算符是不能重载的主要有下面几种
.成员访问运算符; ::作用域解析运算符; .* 成员指针访问运算符; sizeof运算符; 三目运算符;
.成员访问运算符和.*成员指针访问运算符不能重载的原因是为了保证成员访问的功能不能被改变;
::和sizeof运算符操作的对象是类型而不是一般的变量和表达式,不具备重载的特征.
c.重载不能改变操作的对象操作数的个数;
d.重载不能改变运算符的优先级;
e.重载函数的参数不能有默认的缺省参数值,因为它会改变运算符的操作数和前面的规则矛盾;
f.重载的参数不能全部都是C++的基本类型,因为这样会改变原有的用于标准的运算符的性质.
g.应当尽量使自定义的重载操作符和系统用于标准类型的运算符具有相似的功能;
h.运算符重载可以是类的成员函数,还可以是类的友元函数,还可以是普通的全局函数;
4.运算类双目操作符:+ - * /等 a.左右操作数均可以左值或右值;
b.表达式的值为右值.
c.成员函数形式:

```
class LEFT
 {
        const RESULT operator#(const RIGHT& right)const {}
 };
```

全局函数的形式:

```
const RESULT operator#(const LEFT& left,const RIGHT& right){...}
```

```
#include <iostream>
using namespace std;
class Complex
{
public:
    Complex(int r = 0,int i = 0):m_r(r),m_i(i){}
    //运算符+-/*表达式的结果是右值,不能是引用.
    /*参数用常引用的好处:
      即避免了拷贝构造的开销,它又能接收常右操作数
      且可以保证右操作数不被修改.
      第三个const是可以支持常属性的左操作数.
      因为非常对象依然可以调用常函数.
```

```
    第一const为了追求语义上的一致性,使其返回值是一个

    临时变量.不能被赋值.
   */
   /*从做到右的三个const依次表示:

    1.第一个const:返回常对象,禁止对加号表达式进行赋值.

    2.第二个const:支持常右操作数,并且可以避免其被修改.

    3.第三个const:支持常左操作数.
    */
   const Complex operator+(const Complex& c) const
   {
       return Complex(m_r + c.m_r,m_i + c.m_i);
   }
   const Complex operator*(const Complex& c) const
   {
       return Complex(m_r*c.m_r,m_i*c.m_i);
   }
   const Complex operator/(const Complex& c) const
   {
       return Complex(m_r/c.m_r,m_i/c.m_i);
   }
   void print(void) const
   {
       cout << m_r << '+' << m_i << "i" << endl;
   }
   /*用全局函数来实现(友元)
   const Complex operator-(const Complex& c) const
   {
       return Complex(m_r - c.m_r,m_i - c.m_i);
   }
   */
private:
   int m_r;
   int m_i;
   friend const Complex operator-(const Complex&,const Complex&);
};
const Complex operator-(const Complex& left,
       const Complex& right)//这里没有const了,因为

   //const是修饰this指针的,而全局函数是没有this指针的;

{
   return Complex(left.m_r-right.m_r,left.m_i - right.m_i);
}
```

```
{
    Complex c1(1,2),c2(3,4);
    Complex c3 = c1 + c2;

    //编译器翻译成Complex c3 = c1.operator+(c2);

    c3.print();
    Complex c4 = c1 + c2 + c3;
    //Complex c4 = c1.operator+(c2).operator+(c3);
    c4.print();

    //(c1 + c2) = c3;如果加号运算符没有第一个const这里编译会通过.

    c4 = c3 - c1;

    //c4 = ::operator-(c3,c1);也可以处理成这样

    c4.print();
    c4 = c1*c2;
    c4.print();
    c4 = c2/c1;
    c4.print();
    return 0;
}
```

5.赋值类操作运算符:=,+=,-=,*=,/=,^=,&=,|=等

a.左操作数必须是左值,右操作数可以是左值或者右值.

b.表达式的值为左值,返回自引用.是操作数本身,而非副本.

成员函数形式:

```
class LEFT
{
        LEFT& operator#(const RIGHT& right){...}
}
```

全局函数形式:

```
LEFT& operator#(LEFT& left,const RIGHT& right){...}
```

```
#include <iostream>
using namespace std;
class Complex
{
public:
    Complex(int r = 0,int i = 0):m_r(r),m_i(i){}
    void print(void)const
    {
        cout << m_r << '+' << m_i << 'i' << endl;
    }

    //这里不能是const函数,因为左操作数要改变的.

    Complex& operator+=(const Complex& c) //这里不能带const
    {
```

```
        m_i += c.m_i;
        return *this;
    }
    friend Complex& operator-=(Complex& left,const Complex& right)//友元函数
```

没有this指针,也就没有常函数之说.

```
    {
        left.m_r -= right.m_r;
        left.m_i -= right.m_i;
        return left;
    }
    Complex& operator*=(const Complex& c)
    {
        m_r *= c.m_r;
        m_i *= c.m_i;
        return *this;
    }
    Complex& operator/=(const Complex& c)
    {
        m_r /= c.m_r;
        m_i /= c.m_i;
    }
private:
    int m_r;
    int m_i;
};
int main(void)
{
    Complex c1(1,2),c2(3,4),c3(5,6);
    (c1 += c2) = c3;
    //c1.operator+=(c2).operator=(c3);
    c1.print();
    c3 -= c1;
    c3.print();
    //::operator-=(c3,c1);
    c3 *= c1;
    c3.print();
    c3 /= c1;
    c3.print();
    return 0;
}
```

6.运算类单目操作符:-,~,!等

a.操作数为左值或右值.

b.表达式的值为右值.

const RESULT operator#(void)const{...}

const RESULT operator#(const OPERAND& operand){...}

```
#include <iostream>
using namespace std;
class Integer
{
public:
    Integer(int i = 0):m_i(i){}
    void print(void) const
    {
        cout << m_i << endl;
    }
    const Integer operator-(void) const
    {
        return Integer(-m_i);
    }
    friend const Integer operator~(const Integer& i)
    {
        return Integer(i.m_i * i.m_i);
    }
    const Integer operator!(void) const
    {
        return m_i?Integer(0):Integer(1);
    }
private:
    int m_i;
};
int main(void)
{
    Integer i(100);
    Integer j = -i;
    //Integer j = i.operator-();
    j.print();
    j = ~i;

    //j = ::operator~(i)求其平方.用~完成一个平方的效果.

    j.print();

    j = !i;   //取反,表示真假之间的转换.

    j.print();
    return 0;
}
```

7.前++前--后++后--运算符:

a.前自增减单目操作符:操作数为左值,表达式的值为左值,且为操作数本身(而非副本)

成员函数形式:

OPERAND& operator#(void){...}

全局函数形式:

OPERAND& operator#(OPERAND& operand){...}

b.后自增减单目操作符:操作数为左值,表达式的值为右值,且为自增减以前的值.

成员函数形式:

全局函数形式:
const OPERAND operator#(OPERAND& operand,int){...}

8.输出操作符:<<
a.左操作数为左值形式的输出流(ostream)对象里面的成员,右操作数为左值或右值.
b.表达式的值为左值,且为左操作数本身(而非副本)
c.左操作数的类型为ostream,若以成员函数重载该操作符,就应该将其定义为ostream类的成员,该类为标准库提供,无法添加新的成员,因此只能以全局函数的形式重载该操作符
ostream& operator<<(ostream& os,const RIGHT& right){...}
输入操作符:>>
a.左操作数为左值形式的输入流(istream)对象,右操作数为左值.
b.表达式的值为左值,且为左操作数本身(而非副本)
c.左操作数的类型为istream,若以成员函数形式重载该操作符,就应该将其定义为istream类的成员,
该类为标准库提供,无法添加新的成员,因此只能以全局函数的形式重载该操作符
istream& operator>>(istream& is,RIGHT& right){...}

```
#include <iostream>
using namespace std;
class Complex
{
public:
    Complex(int r = 0,int i = 0):m_r(r),m_i(i){}
    friend ostream& operator<<(ostream& os,const Complex& c)
    {
        return os << c.m_r << '+' << c.m_i << 'i';
    }
    friend istream& operator>>(istream& is,Complex& c)
    {
        return is >> c.m_r >> c.m_i;
    }
private:
    int m_r;
    int m_i;
};
int main(void)
{
    Complex c1(12,23),c2(2,4);
    cout << c1 << ", " << c2 << endl;
    cout << "Please input :" << endl;
    cin >> c1 >> c2;
    cout << c1 << ", " << c2 << endl;
    return 0;
}
```

9.下标运算符:[]
a.常用于在容器类型中以下标方式获取数据元素.
b.非常容器元素为左值,常容器的元素为右值.一个是非const成员,一个是const成员.并且必须定义为成员函数

```
#include <iostream>
using namespace std;
class Array
{
public:
    Array(size_t size):m_data(new int[size]),m_size(size){}
    ~Array(void)
    {
        if(m_data != NULL)
            delete[] m_data;
        m_data = NULL;
    }
    int& operator[] (size_t i)
    {
        return *(m_data + i);
    }
    //常版本
    const int& operator[] (size_t i) const
    {
        return const_cast<Array&>(*this)[i];
    }
private:
    int*   m_data;
    size_t m_size;
};
int main(void)
{
    Array a(10);
    a[0] = 13; //a.operator[](0) = 13;
    a[2] = 26;
    a[9] = 58;
    cout << a[0] << ' ' << a[2] << ' ' << a[9] << endl;
    const Array& r = a;
    cout << r[0] << ' ' << r[2] << ' ' << endl;
    return 0;
}
```

10.小括号函数操作符()

a.如果一个类重载了函数操作符,那么该类的对象就可以被做函数来调用,其参数和返回值就是函数操作符函数的参数和返回值.

b.参数的个数,类型以及返回值的类型,没有限制

c.唯一可以带有缺省参数的操作符函数

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

```
{
public:
    double operator()(double x) const
    {
        return x * x;
    }
    int operator()(int a,int b,int c = 9) const
    {
        return a + b - c;
    }
};
int main(void)
{
    Square square;
    cout << square(13.) << endl;
    //cout << square.operator()(13.) << endl;
    cout << square(10,40,20) << endl;
    cout << square(10,40) << endl;
    return 0;
}
```

11.解引用和间接成员访问操作符:* 以及 ->

a.如果一个类重载了解引用和间接成员访问操作符,那么该类的对象就可以被当作指针来使用.

零碎知识总结

1.变量的声明和定义有什么区别?

声明:变量的声明做了两件事情

a.告诉编译器这个变量已经匹配到一块内存上了,下面的代码用到的变量或对象是在别处定义的.

声明可以出现很多次.

b.告诉编译器这个变量名已经被我占用了,其他的的变量将不能再使用.

定义:告诉编译器创建一个对象,为这个对象分配一块内存并给它取一个名字,这个名字就是常说的变量名或对象名.同一变量或对象的定义只能出现一次.

本质区别:声明没有分配内存,而定义则是创建了对象并为这个对象分配了一块内存.

2.sizeof和strlen之间的区别?

a.sizeof是一个操作符,而strlen是一个库函数,使用需要包含<string.h>头文件.

b.sizeof的参数可以是数据类型或变量,而strlen函数只能以结尾为'\0'的字符串的作为参数.

c.编译器在编译的时候就计算出了sizeof的结果.而strlen函数必须在运行的时候才能计算出来.

d.sizeof计算的是数据类型占用的内存的大小,而strlen计算的字符串实际的长度,不包括'\0';

注意strlen在计算字符数组的时候,遇到0的时候和遇到'\0'的效果一样.

e.数组作为sizeof的参数不会退化,而作为strlen的参数的时候会退化为一个字符指针.

3.简要的说下static的用途.C语言关键字static 和 C++的关键字static之间的区别?

a.在C语言中static可以修饰局部变量,全局变量和函数.

static修饰的局部变量,改变了它的存储方式,由原来的栈区改变成静态区.实际上就是影响了它的生命周期,作用域并没有改变.好处是具有继承特性,每次调用时候都要创建的开销.

static修饰的全局变量,改变了它的声明周期,但是存储方式并没有改变,普通的全局变量和静态的全局变量都是在静态存储区域,只是改变了它的作用域.由原来的在整个源程序都可以使用到只有定义它的那个源文件使用.

static修饰的函数也是改变它的作用域.static修饰的函数又叫内部函数,只能在本模块内使用

b.C++中除了上述的用途之外,static还可以定义类的成员变量和函数.
static定义的成员变量和函函数是隶属于类,而不是对象.所有的用类实例化的对象都共享一份,访问static成员的时候有两种方式:一是通过类域解析::一个是通过实例化的对象.
C++的静态成员可以在多个对象间进行通信,传递信息.

4.C中的malloc/free和C++中new/delete之间的区别?

a.maloc/free是C的标准库函数,可以覆盖,但是不能重载,C和C++都可以使用.而new/delete是操作符,
可以重载，只能在C++中使用.

b.对于非内部的数据类型的对象而言,光用malloc/free是无法满足对象的要求的.对象在创建的时候需要
执行构造函数,对象在消亡之前需要执行析构函数.而malloc/free是库函数而不是运算符,不在编译器
的控制范围之内,编译器不能将执行构造函数和析构函数的任务强加给malloc/free.因此C++需要有一
个能够完成动态分配内存和初始化的new,以及一个能够完成清理和释放内存的运算符delete.

c.new的返回值是指定类型的指针,可以自动的计算所需要分配的内存的大小.而malloc的返回值是一个
void类型的指针,使用的使用要进行强制类型转换,并且分配的大小也要程序员的手动进行计算.

d.new/delete完全覆盖了malloc/free的功能,只所以还保留malloc/free,是因为我们在写C++程序的时候有
时会调用用C编写的代码,而C中又没有new/delete操作符.

e.new的时候做两件事情:内存被分配,为被分配的内存调用一个或多个构造函数构建对象.
delete的时候也是一样:为将要被释放的内存调用一个或多个析构函数,释放内存.

4.写一个标准的宏MIN,并说明下一个宏的一些用的注意事项?

```
#define   MIN(a,b)   ((a) < (b) ? (a) :(b))
```

调用的时候要注意它的副作用,例如
((++*p)<=(x)?(++*p):(x))这种三目操作符最好不要用来比较带有自增或自减的表达式,因为在运算的
过程式会多改变一次,就违背了原来的本意.

5.一个指针可以是volatile吗?

可以.因为指针和普通的变量一样也是一个变量,有时也会有变化程序的不可控制性.常见的例子:
子中断服务子程序修改一个指向一个buffer的指针时,必须用volatile来修饰这个指针.
volatile修饰的指针通常是共享指针,通常是这个指针被多个服务共享的时候,这个时候就有可能
在编译器的检测能力之外改变它的值.

6.简述strcpy,sprintf,memcpy的区别?

```
char* strcpy(char* dest,const char* src);
int      sprintf(char* str,cosnt char* format,...);
void* memcpy(void* dest,void* src,size_t n);
```

1.操作的对象不同,strcpy的两个操作对象均为字符串,sprintf的操作源对象可以是多种数据类型,
目的操作对象是字符串,memcpy的两个对象是两个任意类型的可以操作的内存地址,不限制
任何的类型.
2.执行的效率不同,memcpy最高,strcpy次之,sprintf效率最低.
3.实现的功能不同,strcpy主要实现字符串变量间的拷贝,它不检测边界.sprintf主要实现其他数据格式
到字符串之间的转换.memcpy主要是内存块间的拷贝.

7.设置地址为0x67a9的整型变量的值为0xaa66;

```
int* p = (int*)0x67a9;
*p = 0xaa66;
```

无论是什么平台地址长度和整型数据的长度都是一样的,即一个整型数据可以强制类型转换成地址
指针类型的只要有意义即可

a.重载(overload)的特征:函数名相同,同一作用域下参数列表不同的函数才形成重载.它对于返回类型
和是否是virtual函数没有关系.重载的功能就是同一函数名具有不同的行为.
b.覆盖(override)的特征:它指的是派生类的函数覆盖基类的函数,因此作用域不同.并且覆盖的要求
函数名相同,参数列表相同并且返回类型都必须相同.基类的成员函数必须是虚函数.
所谓的覆盖指的是:用基类对象的指针或引用访问虚函数根据实际的指向来实际的决定所调用的函数
c.隐藏(hide)的特征:隐藏指的是派生类的函数隐藏(屏蔽)了与其同名的基类的函数.在调用一个类成员函数的
时候,编译器会沿着类的继承链逐级的向上查找函数的定义,如果找到了,那么就停止查找了.所以如果一个
派生类和它的基类都有同一个同名的函数,编译器最终选择派生类中的函数,那么就说派生类中的成员函数
隐藏了基类中的成员函数,也就是说它阻止了编译器继续向上查找的行为.