**C++ 学习心得**

1. **const int i = -1, &r = 0; 是合法的,i是一个常量,r是一个常量引用,常量引用可以绑定到字面值常量,而非常量引用则不能。**
2. **const int buf; 是非法的,const对象创建后其值不能改变,所以const对象必须初始化,int const \*const p\_a;也是非法的,而int const \* p\_a;是合法的,int const \* p\_a;不是一个const对象,表示定义了一个不能改变其指向的对象的指针p\_a,但指针p\_a的值可以改变。**
3. **指针指向内存中的某个对象,而引用绑定到内存中的某个对象,它们都实现了对其他对象的间接访问,二者的区别主要有两方面:**
4. **指针本身就是一个对象,允许对指针进行赋值和拷贝,而且在指针的生命周期内可以指向几个不同的对象；引用不是一个对象,无法令引用重新绑定到另外一个对象。**
5. **指针无需在定义时赋值,和其他内置类型一样,在块作用域内定义的指针如果没有被初始化,也将拥有一个不确定的值；引用则必须在定义时赋值。**
6. **如何区分顶层const和底层const:**
7. **常量指针:指向“常量”的指针,代表不能改变其指向的对象的指针,表示指针指向的对象的值不能通过指针的方式去改变。声明时const限定符可以放在类型名前后都可,举例int类型:声明时int const 和const int 是等价的,声明指向“常量”的指针是底层const,**

**int num\_a = 1;**

**int const \*p\_a = &num\_a;//底层const**

**//\*p\_a = 2；//错误,指向“常量”的指针不能改变其所指的对象**

**注意:指向“常量”的指针不代表其所指的对象一定是常量,只是代表不能通过解引用符(操作符\*)来改变它所指向的对象,上述指针p\_a指向的就不是常量,可以通过赋值语句num\_a = 2;来改变它所指向的对象。**

1. **指针常量:代表指针本身是常量,表示指针指向了某个对象之后不能再指向其他的对象,声明时必须初始化,之后它存储的值(地址)不能再改变,声明时const限定符必须放在指针符号\*后面,即\*const,声明指针常量是顶层const,即指针本身添加const限定符是顶层const,顶层const可以进一步地表示任意的对象是常量。**

**int num\_b = 2;**

**int \*const p\_b = &num\_b; //顶层const**

**// p\_b = &num\_a; //错误,指针常量不能改变其存储的值(地址)**

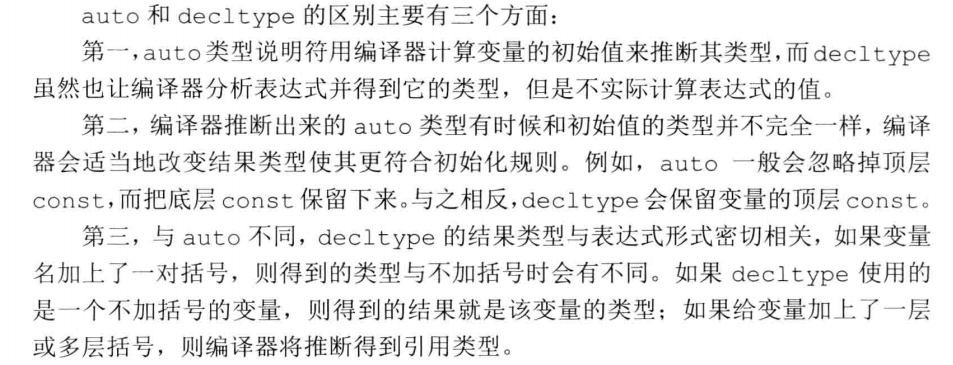
**\*p\_b = 3; //正确,如果指针常量所指的对象不是一个常量的话,可以通过指针常量修改其所指的对象。**

1. **const int v2 = 0; int v1 = v2; int \*p1 = &v1; const int \*p2 = &v2;**

**p1 = p2; 是非法的,p1是普通指针,指向的对象可以是任意值,p2是指向“常量”的指针,不能改变其所指的对象(底层const),令p1指向p2指向的对象,就会改变p2所指的对象。**

**p2 = p1; 是合法的,p2可以指向一个非常量,只不过不能通过p2改变其所指的对象。**

1. **int x = 0; decltype(f()) sum = x; sum的类型就是函数f的返回类型,decltype如果使用的是一个不加括号的变量,则得到的结果就是该变量的类型；如果给变量加上了一层或多层括号,编译器会把它当成一个表达式,从而推断得到引用类型。**



**8. struct Foo{ /\* 此处为空 \*/ }; 类体后面可以紧跟变量名以示对该类型对象的定义,须在类体右侧表示结束的花括号之后写一个分号。**

**9. 首先查看赋值的对象其值是否能够改变(简言之是否是常量),如果赋值的对象是一个指针的话,再查看指针所指的对象是否是常量,最后再查看赋值的类型是否一致,如果没有问题,方可赋值。**

**10. 把0到15之间的十进制数转换成对应的十六进制形式:**

**const string hexdigits = "0123456789ABCDEF";**

**string result;**

**string::size\_type n;**

**cout << hexdigits.size() << endl;**

**while (cin >> n)**

**{**

**if (n < hexdigits.size())**

**result += hexdigits[n];**

**}**

**cout << result << endl;**

**在输入完数据按回车键后,该行数据才被送入键盘缓冲区,形成输入流,提取运算符“>>”才能从中提取数据；当while在处理cin >> string::size\_type类型的对象时,会对输入流中的每个字符进行判断其合法性,(此时输入流已在缓冲区)直到遇到非法输入时退出while循环,该行从最初合法输入至非法输入前的内容会被依次处理。**

**上述如果键入 12 0 c 4回车后的结果为C0**

1. **const 和 constexpr之间的主要区别是const变量的初始化可以延迟到运行时,而constexpr变量必须在编译时进行初始化,使用constexpr时,必须用常量表达式进行初始化。**
2. **指针指向的是什么类型的对象,解引用出来的就是什么类型的对象；**

**int ia[3][4];int cnt = 0;**

**for (auto p = begin(ia); p != end(ia); ++p)**

**{**

**for (auto q = begin(\*p); q != end(\*p); ++q)**

**{**

**\*q = cnt;**

**cnt++;**

**}**

**}**

**auto p = ia;编译器会自动地将数组名ia转换成指向数组ia首元素的指针,即auto p = &ia[0],指针p指向了数组ia的第一个内层数组(ia的首元素ia[0]),\*p的结果是获取数组ia的首元素(ia[0]),由于ia[0]也是一个数组,没有对数组取下标索引,因此auto q = \*p；是把\*p对象的首地址赋值给q,由于\*p是一个一维数组,所以指针q指向的是\*p数组的第一个整型元素ia[0][0]**

**for (auto p = ia; p != end(ia);++p)**

**{**

**for (int q = 0; q < 4; ++q)**

**{**

**cout << (\*p)[q] << “ ”**

**}**

**cout << endl;**

**}**

**输出结果如下:**



**指针p指向的是一维数组ia[0],对p解引用的结果是一维数组,如若访问对象\*p中的元素可以使用对象加索引的方式进行,即(\*p)[q],(\*p)是一个对象,q是一个索引值。**

**int \* ip[3][4];**

**int m = 0;**

**for (auto p = ia; p != end(ia); ++p)**

**{**

**for (auto q = \*p; q != end(\*p) ; q++)**

**{**

**int n = 0;**

**ip[m][n] = q;**

**cout << \*(ip[m][n]) << " ";**

**n++;**

**}**

**cout << endl;**

**m++;**

**}**

**输出结果如下:**



**int (\*p1)[4] = ia;**

**for (int ii = 0 ; ii < 4 ;++ii)**

**{**

**cout << (\*p1)[ii] << " ";**

**}**

**cout << endl;**

**输出结果如下:**



**默认情况下,类型修饰符从右向左依次绑定,对于 int \* ip[3][4]来说,从右向左开始,首先我们定义的是一个大小为3X4的二维数组,它的名字是ip,然后知道数组中存放的是指向int类型的指针。q是一个指向int类型的指针,因此可以赋值给ip[m][n]**

**对于int (\*p1)[4]来说,先从括号内的顺序说起,\*p1意味着是一个指针,再观察右边,可知道p1是一个指向大小为4的数组的指针,最后观察左边,知道数组中元素的类型是int,因此可以这样解释:p1是一个指针,它指向一个int类型数组,数组中包含4个元素。可以通过(\*p1)[ii]来访问数组中的元素。**

1. **若想向vector对象中添加新元素,需要使用push\_back函数;**

**vector<int> vInt; vInt.push\_back(42);**

1. **C++没有定义两个迭代器的加法运算,而是定义了两个迭代器的减法运算,两个迭代器相减的结果是它们之间的距离,也就是说将运算符右侧的迭代器向前移动多少个元素后可以得到左侧的迭代器,参与运算的两个迭代器必须指向同一个容器中的元素或尾后元素。**
2. **string类型的数组无论定义在函数内还是函数外都被默认初始化为空串,而int类型的数组如果定义在所有函数体外,元素则被默认初始化为0,如果定义在函数体内部,则元素拥有一个不确定的值。**
3. **数组与vector的相似之处是都能存放类型相同的对象,且这些对象本身没有名字,需要通过其所在位置访问;不同点是数组的大小固定不变,但运行性能较好。**
4. **与指针相关的属性有3个,分别是指针本身的值(value)、指针所指的对象(content)、以及指针本身在内存中的存储位置(address).指针本身的值是一个内存地址值,表示指针所指对象在内存中的存储地址;指针所指的对象可以通过解引用指针访问;指针也是一个对象,所以指针也存储在内存的某个位置,它有自己的地址。**
5. **C++只允许使用数组初始化vector对象,**

**int a[10];vector<int> vInt(begin(a),end(a));**

1. **char x,\*p = &x;**

**printf("%d\n",sizeof p); //输出8,说明系统为p分配了8字节来存放x的地址**

**printf("%d\n",sizeof \*p); //输出1,说明系统为\*p分配了1字节,也就是为x分配了1字节(1字节(b)=8比特(bit),1千字节(kb)=1024字节(b))**

1. **可以创建对数组的引用,但是不能创建每个元素都是引用的数组;**

**int a[3][4] = {{0,1,2,3},**

**{4,5,6,7},**

**{8,9,10,11}};**

**for (int (&row)[4] : a)**

**{**

**for (int &col : row)**

**{**

**cout << col << " ";**

**}**

**cout << endl;**

**}**

**可以通过int (&row)[4] = a[0];声明一个数组的引用row,但是不能通过**

**int & row[4] =a[0];去声明一个引用的数组,其声明了一个名为row,包含4个整型引用的数组,也就是说数组中的每个元素都是引用,这种声明是非法的,引用必须初始化。**

1. **重载运算符定义运算对象的类型和返回值的类型,但运算对象的个数、运算符的优先级和结合律是无法改变的。**
2. **int ia[] = {0,2,4,6,8};**

**int last = \*(ia + 4); //把last初始化为8,也就是ia[4]的值。**

**last = \*ia + 4; // 等价于ia[0] + 4**

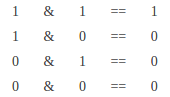
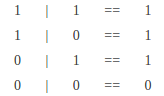
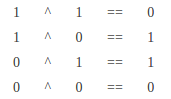
1. **continue语句终止最近的循环中的当前迭代并立即开始下一次迭代。**
2. **break语句负责终止离它最近的while、do while、for、或switch语句,并从这些语句之后的第一条语句开始继续执行。**
3. **求余的规则: m % (-n)等价于m % n,(-m) % n等价于 -(m % n),m和n皆为正数。**
4. **计算机中存储某种类型的内存空间是有限的,因此该类型的表示能力也是有限的,当计算的结果超出该类型所能表示的范围时,产生溢出。**
5. **短路求值策略: 对于逻辑与运算符来说,当且仅当左侧运算对象为真时才计算右侧运算对象;对于逻辑或运算符来说,当且仅当左侧运算对象为假时才计算右侧运算对象。C++没有规定相等性运算符的两个运算对象的求值顺序。**
6. **const char \*cp = “Hello World”; if (cp && \*cp)**

**首先检查指向字符串的指针cp是否有效,如果cp为空指针或无效指针,则条件不满足,如果cp有效,即cp指向了内存中的某个有效地址,继续解引用指针cp并检查cp所指的对象是否为空字符’\0’,如果cp所指的对象不是空字符则条件满足,否则不满足。当前cp所指的对象是字符串”Hello World”的首字符’H’,不是空字符,所以if的条件部分为真。**

1. **C++规定<, <=, >, >= 的优先级高于== 和!= ,i != j < k等价于i != (j < k)**
2. **解引用运算符的优先级低于点运算符**

**vector<string>::iter; \*iter.empty();此语句是非法的,该语句先计算iter.empty(),而迭代器没有定义empty函数。**

1. **与运算(&) 或运算(|)  异或运算(^)**

**非运算(~) : 单目运算,位值取反,置0为1,或置1为0**

1. **switch必须在必要的地方使用break;语句,应该把变量定义在块作用域内,case标签只能有一个值且不能是变量。**
2. **constexpr int size = 5;**

**int ia[size] = {1,2,3,4,5};**

**for (int \*ptr = ia,ix = 0; ix != size && ptr != ia + size; ++ix, ++ptr){}**

**首先定义一个常量表达式size,它的值是5;接着以size作为维度创建一个整型数组ia;for语句头包括三部分:第一部分定义整型指针ptr指向数组ia的首元素,并且定义了一个整数ix,并赋初值为0;第二部分判断循环终止的条件,当ix没有达到size同时指针ptr没有指向ia最后一个元素的下一位置时,执行循环体;第三部分令变量ix和指针ptr分别执行递增操作。**

1. **const string \*ps; void \*pv;**

**pv = (void \*)ps;// c风格的转换,把const string\* 类型转换成void\* 类型**

**pv = static\_cast<void \*>(const\_cast<string \*>(ps));// c++风格的转换,先把const string\* 类型转换成string\* 类型,然后再把string\* 类型转换成void\* 类型**

**利用static\_cast执行强制类型转换,对于底层const则使用const\_cast;**

1. **do {int ival = get\_response();} while(ival);**

**在[第一次](https://www.baidu.com/s?wd=%E7%AC%AC%E4%B8%80%E6%AC%A1&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1dBuWRYuHTsnyFWP1csujRd0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EnWndrHc3n16z" \t "/home/devy/文档\x/_blank)执行do语句的时候虽然创建了ival,但是在遇到"}"的时候,ival会自动消亡,再执行while(ival)语句的时候会找不到这个叫做ival的变量,编译不会通过。**

1. **C++ 参数传递的三种方式**

① **按值传递(pass by value)**

**形参是实参的拷贝,改变形参的值并不会改变外部实参的值。从被调函数的角度来说,值传递是单向的(实参->形参),形参的值只能传入,不能传出,当函数内部需要修改参数,并且不希望这个改变影响外部实参时,采用值传递。**

② **指针传递(pass by pointer)**

**形参为指向外部实参的指针,当对形参所指的对象进行操作时,就相当于对实参本身进行的操作。**

③ **引用传递(pass by reference)**

**形参相当于实参的别名,对形参的操作其实就是对实参的操作,在引用传递过程中,被调函数对形参的任何操作都被处理成间接寻址,即通过栈中存放的地址访问主调函数中的实参变量。正因为如此,被调函数对形参做的任何操作都会影响主调函数中的实参变量。**

**#include <iostream>**

**using namespace std;**

**void change1(int n) {**

**cout << "值传递--被调函数操作地址:" << &n << endl;**

**n++;**

**}**

**void change2(int& n) {**

**cout << "引用传递--被调函数操作地址:" << &n << endl;**

**n++;**

**}**

**void change3(int\* n) {**

**cout << "指针传递--被调函数操作地址:" << n << endl;**

**\*n += 1;**

**}**

**int main() {**

**int n = 0;**

**cout << "实参的地址:" << &n << endl;**

**change1(n);**

**cout << "after change1() n = " << n << endl;**

**change2(n);**

**cout << "after change2() n = " << n << endl;**

**change3(&n);**

**cout << "after change3() n = " << n << endl;**

**return 0;**

**}**

**执行结果:**

**实参的地址:00F5F8C0**

**值传递--被调函数操作地址:00F5F7EC**

**after change1() n = 0**

**引用传递--被调函数操作地址:00F5F8C0**

**after change2() n = 1**

**指针传递--被调函数操作地址:00F5F8C0**

**after change3() n = 2**

**请按任意键继续. . .**

**引用的规则:**

⑴ **引用被创建的同时必须初始化(指针在创建的时候可以不用初始化)**

⑵ **不能有NULL引用,引用必须与合法的存储单元关联(指针则可以是NULL)**

⑶ **一旦引用被初始化,就不能改变引用的关系(指针则可以随时改变其所指的对象)**

**指针传递的实质:**

**指针传递本质上是值传递的方式,它所传递的是一个地址值,被调函数的形式参数(地址)作为被调函数的局部变量处理,即在栈中开辟了内存空间以存放由主调函数放进来的实参(地址)的值,从而成为了实参的一个副本。被调函数对局部变量形式参数的修改,都不会影响主调函数实参(地址)的值,但是可以在被调函数中改变指针所指向的对象,如若在被调函数中修改主调函数中的指针使其指向别的对象,此时需要向被调函数传递二级指针(指向指针的指针),此时指向指针的指针可以对其指向的指针(对象即地址)进行修改,即使修改指向指针的指针,并不会对外部变量产生影响。**

**#include <iostream>**

**using namespace std;**

**void swap(int\*\* p, int\*\* q)**

**{**

**int\* temp;**

**temp = \*p;**

**\*p = \*q;**

**\*q = temp;**

**}**

**int main() {**

**int a = 0, b = 1;**

**int\* p1;**

**int\* p2;**

**p1 = &a;**

**p2 = &b;**

**swap(&p1, &p2);**

**cout << "a is :" << a << endl;**

**cout << "b is :" << b << endl;**

**cout << "\*p1 is :" << \*p1 << endl;**

**cout << "\*p2 is :" << \*p2 << endl;**

**return 0;**

**}**

**执行结果:**

**a is :0**

**b is :1**

**\*p1 is :1**

**\*p2 is :0**

**请按任意键继续. . .**

**可以看出swap完成的功能是通过二级指针交换形式变量p,q各自所指向的对象(指针)。**

**二级指针p,q通过修改&p1,&p2所指向的对象(指针),使其各自所指向的对象(指针)互换,即p1变成了以前的p2,p2变成了以前的p1,从结果可以看出p1以前指向的是变量a,现在指向的是变量b,p2以前指向的是变量b,现在指向的是变量a,但是a和b的值没有发生改变,只是以前分别指向a和b的指针p1,p2发生了改变。**

**p p1 a**

a value

&a

&p1

**q p2 b**

&b

b value

&p2

**函数执行后:**

**p p1 b**

b value

&b

&p1

**q p2 a**

&a

a value

&p2

|  |  |  |  |
| --- | --- | --- | --- |
|  | **名称** | **值** | **类型** |
| **◢** | **q** | **0x0093fd24 {0x0093fd3c {1}}** | **int \* \*** |

**变换前**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **名称** | **值** | **类型** |
| **◢** | **p** | **0x0093fd30 {0x0093fd48 {0}}** | **int \* \*** |

|  |  |  |  |
| --- | --- | --- | --- |
|  | **名称** | **值** | **类型** |
| **◢** | **q** | **0x0093fd24 {0x0093fd48 {0}}** | **int \* \*** |

**变换后**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **名称** | **值** | **类型** |
| **◢** | **p** | **0x0093fd30 {0x0093fd3c {1}}** | **int \* \*** |

**变换前p1的值是0x0093fd48,变换后p1的值是0x0093fd3c**

**变换前p2的值是0x0093fd3c,变换后p2的值是0x0093fd48**

1. **静态类型与动态类型,静态绑定与动态绑定**

**静态类型:程序在编译时确定变量的类型,在程序运行时不会改变。**

**动态类型:程序在运行时才能确定变量的实际类型。是指由一个左值表达式表示的左值所指向的最终派生类对象的类型。例如,如果一个静态类型为类B的指针p指向一个继承于B的类D的对象,则表达式\*p的动态类型为D,引用按照相似规则处理。**

**静态绑定:程序在编译时绑定,通过对象调用实现。**

**动态绑定:程序在运行时绑定,通过地址实现,只有采用“指针->函数()”,或“引用变量.函数()”的方式调用C++类中的虚函数才会执行动态绑定,对于虚函数,实际调用该函数的对象类型为该指针或引用所指向的对象的实际类型。**

**#include <iostream>**

**using namespace std;**

**class Base {**

**public:**

**void func() {**

**cout << "func() in Base." << endl;**

**}**

**virtual void test() {**

**cout << "test() in Base." << endl;**

**}**

**};**

**class Derived : public Base {**

**void func() {**

**cout << "func() in Derived." << endl;**

**}**

**void test() {**

**cout << "test() in Derived." << endl;**

**}**

**};**

**int main() {**

**Base\* b;**

**b = new Derived();**

**b->func();**

**b->test();**

**return 0;**

**}**

**执行结果:**

**func() in Base.**

**test() in Derived.**

**按任意键关闭此窗口...**

**由运行结果可以看出,b是一个基类指针,它指向了一个派生类对象,基类Base里面有两个函数,其中test为虚函数,func为非虚函数,因此,对于test就表现为动态绑定,实际调用的是派生类对象中的test,而func为非虚函数,因此表现为静态绑定,即指针类型是什么类型,就会调用该类型相应的函数。**

1. **虚函数、动态绑定、运行时多态之间的关系**

**虚函数是动态绑定的基础,动态绑定是实现运行时多态的基础。**

**要触发动态绑定,需要满足两个条件:**

① **只有虚函数才能进行动态绑定,非虚函数不进行动态绑定。**

② **必须通过基类类型的指针或引用实现对函数的调用。**

**通过基类指针或基类引用做形参,当实参传入子类(或派生类)的指针或引用时,在函数内部触发动态绑定,从而实现运行时多态。**

**在上述代码中添加两个全局函数分别以基类指针和基类引用做形参来实现运行时多态。**

**#include <iostream>**

**using namespace std;**

**class Base {**

**public:**

**void func() {**

**cout << "func() in Base." << endl;**

**}**

**virtual void test() {**

**cout << "test() in Base." << endl;**

**}**

**};**

**class Derived : public Base {**

**void func() {**

**cout << "func() in Derived." << endl;**

**}**

**void test() {**

**cout << "test() in Derived." << endl;**

**}**

**};**

**void Polymorphic1(Base& b){**

**b.func();**

**b.test();**

**}**

**void Polymorphic2(Base\* b) {**

**b->func();**

**b->test();**

**}**

**int main() {**

**Base b;**

**Derived d;**

**cout << "对通过基类指针作形参实现多态进行测试:" << endl;**

**cout << "Base b by pointer: " << endl;**

**Polymorphic2(&b);**

**cout << "Derived d by pointer: " << endl;**

**Polymorphic2(&d);**

**cout << "对通过基类引用作形参实现多态进行测试:" << endl;**

**cout << "Base b by reference: " << endl;**

**Polymorphic1(b);**

**cout << "Derived d by reference: " << endl;**

**Polymorphic1(d);**

**return 0;**

**}**

**程序执行结果:**

**对通过基类指针作形参实现多态进行测试:**

**Base b by pointer:**

**func() in Base.**

**test() in Base.**

**Derived d by pointer:**

**func() in Base.**

**test() in Derived.**

**对通过基类引用作形参实现多态进行测试:**

**Base b by reference:**

**func() in Base.**

**test() in Base.**

**Derived d by reference:**

**func() in Base.**

**test() in Derived.**

**请按任意键继续. . .**

1. **虚函数、动态绑定**

**#include "pch.h"**

**#include <iostream>**

**using namespace std;**

**class Shape {**

**protected:**

**int width, height;**

**public:**

**Shape(int a = 0, int b = 0)**

**{**

**width = a;**

**height = b;**

**}**

**virtual int area()**

**{**

**cout << "Parent class area :" << endl;**

**return 0;**

**}**

**};**

**class Rectangle : public Shape {**

**public:**

**Rectangle(int a = 0, int b = 0) :Shape(a, b) { }**

**int area()**

**{**

**cout << "Rectangle class area :" << endl;**

**return (width \* height);**

**}**

**};**

**class Triangle : public Shape {**

**public:**

**Triangle(int a = 0, int b = 0) :Shape(a, b) { }**

**int area()**

**{**

**cout << "Triangle class area :" << endl;**

**return (width \* height / 2);**

**}**

**};**

**// 程序的主函数**

**int main()**

**{**

**Shape \*shape;**

**Rectangle rec(10, 7);**

**Triangle tri(10, 5);**

**// 存储矩形的地址**

**shape = &rec;**

**// 调用矩形的求面积函数 area**

**shape->area();**

**// 存储三角形的地址**

**shape = &tri;**

**// 调用三角形的求面积函数 area**

**shape->area();**

**return 0;**

**}**

**多态是指不同的类可以拥有相同名称的成员函数,但具体的实现不同,函数的参数可以是相同或者不相同。**

**虚函数是在基类中使用关键字virtual声明的函数。在派生类中重新定义基类中定义的虚函数时,会告诉编译器不要静态链接(函数调用在程序执行前就已经准备好了)到该函数,函数调用在程序执行的过程中根据所调用的对象的类型来选择调用的函数(确定为派生类的对象分配具体的内存空间),这种操作被称为动态链接或者是动态绑定。基类的析构函数前面需要加上virtual关键字,只要基类的析构函数是虚函数,则能够保证delete基类指针(此时基类指针指向子类)时同时运行基类和派生类的析构函数。**

**纯虚函数是指在基类中不能对虚函数给出有意义的实现,以便在派生类中重新定义虚函数以更好地适用于对象。纯虚函数的写法如下：**

**virtual int area() = 0; “= 0” 告诉编译器纯虚函数没有主体,含有纯虚函数的类是抽象类,抽象类不能生成对象。**

1. **C++静态局部变量**

**普通成员变量每个对象有各自的一份,而静态成员变量一共就一份,为所有对象共享。**

**在静态成员函数中,不能访问非静态成员变量,也不能调用非静态成员函数。**

1. **delete运算符**

**野指针是指不可用内存区域的指针或者是不可访问的内存区域的指针,delete之后的指针会变成一个野指针,**

**int\* i\_p1 = new int(33);**

**cout << "指针i\_p1的地址是：" << &i\_p1 << " 指针i\_p1所指向的值是：" << \*i\_p1 << " 指针i\_p1的值是：" << i\_p1 << endl;**

**delete i\_p1;**

**//cout << "指针i\_p1的地址是：" << &i\_p1 << " 指针i\_p1所指向的值是：" << \*i\_p1 << " 指针i\_p1的值是：" << i\_p1 << endl;**

**int i1 = 44;**

**i\_p1 = &i1;**

**cout << "指针i\_p1的地址是：" << &i\_p1 << " 指针i\_p1所指向的值是：" << \*i\_p1 << " 指针i\_p1的值是：" << i\_p1 << endl;**

**i\_p1 = nullptr;**

**int i2 = 55;**

**i\_p1 = &i2;**

**cout << "指针i\_p1的地址是：" << &i\_p1 << " 指针i\_p1所指向的值是：" << \*i\_p1 << " 指针i\_p1的值是：" << i\_p1 << endl;**

**程序执行结果:**

**指针i\_p1的地址是：00EFFD48 指针i\_p1所指向的值是：33 指针i\_p1的值是：00136DF0**

**指针i\_p1的地址是：00EFFD48 指针i\_p1所指向的值是：44 指针i\_p1的值是：00EFFD3C**

**指针i\_p1的地址是：00EFFD48 指针i\_p1所指向的值是：55 指针i\_p1的值是：00EFFD30如果把**

**//cout << "指针i\_p1的地址是：" << &i\_p1 << " 指针i\_p1所指向的值是：" << \*i\_p1 << " 指针i\_p1的值是：" << i\_p1 << endl;**

**这一语句取消注释,则程序会在这一语句运行时崩溃,原因时delete后的指针i\_p1是一个野指针,程序无法访问i\_p1,因此会崩溃,**

**delete的作用是释放该指针所指向的对象的内存空间并被系统回收,同时解除指针与该指针所指向的对象之间的绑定关系,当使用delete之后,该指针不再指向其所指向的对象,而此时的指针变成一个野指针,不能对其进行访问,如果想继续使用该指针,则首先需对其进行赋值操作(重新指向一个对象),然后才可以对其进行访问。指针执行delete之后再对其进行赋值为nullptr的操作,如果想对其进行访问,也是需要对该指针首先进行赋值操作(重新指向一个对象),才能继续使用该指针。**

1. **定义 声明 赋值 拷贝构造函数**

**class Person**

**{**

**public:**

**Person() {**

**cout << "Default Constructor" << endl;**

**}**

**Person(const Person& p)**

**{**

**cout << "Copy Constructor" << endl;**

**}**

**Person& operator=(const Person& p)**

**{**

**cout << "Assign Operator" << endl;**

**return \*this;**

**}**

**Person(const Person && p)**

**{**

**cout << "Move Constructor" << endl;**

**}**

**private:**

**int age;**

**string name;**

**};**

**void f(Person p)**

**{**

**return;**

**}**

**Person f1()**

**{**

**Person p;**

**return p;**

**}**

**void f2(Person& p)**

**{**

**return;**

**}**

**// 定义：为对象分配内存空间**

**// 声明：告诉编译器有这样一个变量名存在**

**int main()**

**{**

**Person p; //1**

**Person p1 = p; //2**

**Person p2; //3**

**p2 = p; //4**

**f(p2); //5**

**p2 = f1(); //6**

**Person p3 = f1(); //7**

**cout << "f2()" << endl;**

**f2(p2);**

**cout << endl;**

**return 0;**

**}**

**程序执行结果如下:**

**Default Constructor**

**Copy Constructor**

**Default Constructor**

**Move Constructor**

**Assign**

**Default Constructor**

**Move Constructor**

**f2()**

1. **定义对象p调用的是默认构造函数Person()**
2. **定义对象p1调用的是拷贝构造函数Person(const Person& p),因为需要为对象p1分配内存空间**
3. **定义对象p2调用的是默认构造函数Person()**
4. **把对象p的值赋值给p2调用的是Person类的赋值运算符Person& operator=(const Person& p)**
5. **函数f的参数类型是Person,在调用过程中,首先函数f把实参p2传递给形参p,相当于在栈中为Person的对象p分配了一块内存空间,此时对象p调用的是拷贝构造函数Person(const Person& p)**
6. **函数f1没有参数,但有返回值,返回值的类型为Person,在函数f1内部,定义了一个Person对象p,在栈中为p分配了一块内存,此时对象p调用的是默认构造函数Person(),由于f1的返回值类型是Person,对象p在离开函数f1的作用域之前需要把p的值传递给临时变量temp,然后temp再把值传递给main()函数中的变量p2,此时把p的值赋值给temp时,首先要对temp进行内存空间分配,对象temp调用的是拷贝构造函数Person(const Person& p)临时对象 temp再把值赋值给p2,则对象p2调用的是赋值运算符Person& operator=(const Person& p)**
7. **类似于p2 = f1();此过程首先函数f1内部对象p调用了一次默认构造函数Person(),此时不再产生临时变量temp,而是调用移动构造函数Person(const Person && p),因此在为对象p3分配内存空间之后,直接把函数f1内部对象p的值赋值给p3**

**总结:如果把类Person的拷贝构造函数以及赋值运算符的定义实现部分注释掉,则程序运行的时候会调用默认的拷贝构造函数以及默认的赋值运算符,如果函数f(Person p)改成f(Person& p),则在执行语句f(p)的时候不会调用拷贝构造函数Person(const Person& p)**

1. **类的public成员函数可以访问同类对象的私有成员变量**

**#include "pch.h"**

**#include <iostream>**

**using namespace std;**

**class CTest**

**{**

**public:**

**CTest(int i);**

**CTest(const CTest& ct);**

**CTest& operator=(const CTest& ct);**

**void printValue(const CTest& ct);**

**private:**

**int value;**

**};**

**CTest::CTest(int i) :value(i)**

**{**

**cout << "Default Constructor" << endl;**

**}**

**CTest::CTest(const CTest& ct) : value(ct.value)**

**{**

**cout << "Copy Constructor" << endl;**

**}**

**CTest& CTest::operator=(const CTest& ct)**

**{**

**cout << "Assign Operator" << endl;**

**if (this == &ct)**

**return \*this;**

**this->value = ct.value;**

**return \*this;**

**}**

**void CTest::printValue(const CTest& ct)**

**{**

**cout << ct.value << endl;**

**}**

**int main()**

**{**

**CTest t1(1);**

**CTest t2(2);**

**//cout << t1.value << endl; //对象不能直接访问类的私有成员变量,但是可以通过类的//public成员函数可以访问同类的私有成员变量**

**t1.printValue(t2);**

**std::cout << "Hello World!\n";**

**}**

**程序执行结果如下:**

**Default Constructor**

**Default Constructor**

**2**

**Hello World!**

1. **类的友元函数可以访问同类对象的私有成员变量**

**#include "pch.h"**

**#include <iostream>**

**using namespace std;**

**class Coordinate**

**{**

**friend ostream& operator<<(ostream& out, Coordinate& coor);**

**public:**

**Coordinate(int x = 0, int y = 0);**

**void printCoordinate(const Coordinate& coor);**

**Coordinate& operator-();**

**bool operator==(Coordinate& coor);**

**private:**

**int m\_iX;**

**int m\_iY;**

**};**

**Coordinate::Coordinate(int x, int y)**

**{**

**m\_iX = x;**

**m\_iY = y;**

**}**

**void Coordinate::printCoordinate(const Coordinate& coor)**

**{**

**cout << "(" << coor.m\_iX << "," << coor.m\_iY << ")" << endl;**

**}**

**Coordinate& Coordinate::operator-()**

**{**

**m\_iX = -m\_iX;**

**m\_iY = -m\_iY;**

**return \*this;**

**}**

**ostream& operator<<(ostream& out, Coordinate& coor)**

**{**

**out << "(" << coor.m\_iX << "," << coor.m\_iY << ")" << endl;**

**return out;**

**}**

**bool Coordinate::operator==(Coordinate& coor)**

**{**

**return this->m\_iX == coor.m\_iX && this->m\_iY == coor.m\_iY;**

**}**

**int main()**

**{**

**Coordinate coor1(1, 2);**

**Coordinate coor2(3, 4);**

**coor1.printCoordinate(coor2);**

**cout << coor1 << endl; //对运算符<<重载,并且是类Coordinate的友元函数,可以对 //Coordinate的私有成员变量进行访问。**

**std::cout << "Hello World!\n";**

**}**

**程序执行结果如下：**

**(3,4)**

**(1,2)**

**Hello World!**

1. **C++对类的成员函数和成员变量的访问修饰符的默认值是private,成员函数的末尾加上const的作用是指该成员函数不会修改该对象的任何成员变量的值,这样的成员函数也称为常量成员函数。mutable修饰类的成员变量,一旦类的成员变量被mutable修饰了,则表示这个类的成员变量处于可以被修改状态,即使这个成员变量在常量成员函数中被使用,也可以修改。this代表了类的自身对象的引用,编译器把类的对象的地址传递给成员函数中的隐藏的形参(this),任何对类成员的直接访问都被看做是通过this做隐式调用的。在普通成员函数中,this是一个指向非const对象的const指针,在常量(const)成员函数中this指针是一个指向const对象的const指针。**
2. **C语言中static的语义:**
3. **static变量:静态局部变量在函数内定义且只初始化一次,生存期为整个源程序,但作用域与普通变量相同,只能在定义该变量的函数内使用。退出该函数后,尽管该变量还继续存在但不能使用它。对基本类型的静态局部变量若在声明时未赋以初值,则系统自动赋予0值。而对自动变量不赋初值,则其值是不定的。非静态全局变量的作用域是整个源程序(多个源文件可以共同使用);而静态全局变量则限制了其作用域,即只在定义该变量的源文件内有效,在同一源程序的其它源文件中不能使用它。**
4. **static函数:(也叫内部函数)只能被本文件中的函数调用,而不能被同一程序其它文件中的函数调用。**
5. **C++语言中的static成员:**
6. **类的static成员变量属于整个类的成员变量,不属于某个对象,我们一旦在某个对象中修改了类的static成员变量的值,则在其他对象的成员变量中也能看到被修改的结果。对于这种类的static成员变量的调用,我们使用类名::成员变量名。**
7. **类的static成员函数是属于整个类的成员函数,调用时使用类名::成员函数名。**
8. **const成员变量的初始化,在构造函数的初始化列表里进行,不可以通过赋值来进行初始化。**
9. **构造函数:构造函数后面紧跟=default;编译器则自动为构造函数生成函数体。构造函数后面如果紧跟=delete;编译器则禁止生成默认的构造函数。**
10. **RTTI(Run Time Type Identification)运行时类型识别的两个功能:**
11. **dynamic\_cast运算符:能够将基类的指针或引用安全地转换为派生类的指针或引用,dynamic\_cast运算符是为了防止当基类的指针指向派生类的对象时,而此时的对象调用了派生类独有的成员函数(基类不存在此成员函数)而引发的错误。与static\_cast的区别是强制类型转换,使用static\_cast的前提是要保证转换后的类型是正确的。**
12. **typeid运算符:返回指针或引用所指对象的实际类型,主要是为了比较两个指针是否指向同一类型的对象。只有当基类有虚函数时,编译器才会对typeid()中的表达式求值,否则typeid()返回的是表达式的静态类型(定义时的类型)。**
13. **左值引用不能绑定到临时变量上去,临时变量会被系统当作是右值,同时右值引用不能绑定左值,只能绑定临时变量。**
14. **++i是一个左值表达式,表达式++i直接返回的是i变量本身,i++则是右值表达式,因为表达式i++会在系统中返回值为i的一个临时变量,再进行i = i + 1;语句的执行。**

**int i = 1;**

**int &&rrefi = i++; //成功绑定右值,但rrefi的值和i没有关系,因为表达式i++返回的是一个临时变量。**

**i += 5; //此时i的值为7,而rrefi的值为1**

**int &&rrefi = ++i;//语法错误,右值引用不能绑定到左值表达式上**

**int &lrefi = i++;//语法错误,左值引用不能绑定到右值表达式上**

**int &lrefi2 = rrefi;**

**rrefi虽然是右值引用(绑定到了右值),但rrefi本身是左值(变量)**

**string st = “I Love China!”;**

**string def = std::move(st);//std::move()函数调用的是string类型中的移动构造函数把st的内容转移到def中去了,而st的内容为空,地址没有变。**

**sting &&ref = std::move(st);//把一个左值st转换成临时右值,然后赋值给右值引用变量ref,相当于右值引用ref和左值变量st进行了绑定,ref是st的别名。**

1. **模板定义并不会导致编译器生成代码,只有在我们调用这个函数模板时,编译器为我们实例化了一个特定版本的函数之后,编译器才会生成代码,编译器生成代码的时候,需要能够找到函数的函数体,函数模板的定义通常都是在.h的头文件中。**

**template<typename T,int a,int b>**

**int funcaddv3(T c)**

**{**

**return int(c) + a + b;**

**}**

**int a = 1;int b = 2;int c = funcaddv3<int , a , b>(3);//语法错误,模板实参需要在编译时指定**

**funcaddv3<int,1,2>(3);//语法正确**

**<typename T,int a,int b>是模板形式参数列表,其中T代表的是一个类型,int a,int b代表非类型参数,非类型参数在编译器生成代码的时候指定。**

1. **赋值运算符为什么要用引用：**

**并非必须返回引用,返回引用的好处既可以避免拷贝构造函数和析构函数的调用,又可以保证=、 +=等运算符的原始语义清晰。啥叫原始语义清晰呢？如**

**(str3 = str1) = str2;**

**我们的意识里,就是先执行括号内容,即str1赋值给str3,然后str2再赋值给str3,最后str3输出的内容是str2的,即如果运算符重载返回的是对象引用时。**

**//返回的是对象引用的情况**

**#include <iostream>**

**using namespace std;**

**class String**

**{**

**private:**

**char\* str;**

**int len;**

**public:**

**String(const char\* s);//构造函数声明**

**String& operator=(const String& another);//运算符重载,此时返回为引用**

**void show()**

**{**

**cout << "value = " << str << endl;**

**}**

**/\*copy construct\*/**

**String(const String& other)**

**{**

**len = other.len;**

**str = new char[len + 1];**

**strcpy(str, other.str);**

**cout << "copy construct" << endl;**

**}**

**~String()**

**{**

**delete[] str;**

**cout << "deconstruct" << endl;**

**}**

**};**

**String::String(const char\* s)**

**{**

**len = strlen(s);**

**str = new char[len + 1];**

**strcpy(str, s);**

**}**

**String& String::operator=(const String &other)**

**{**

**if (this == &other)**

**return \*this;**

**// return;**

**delete[] str;**

**len = other.len;**

**str = new char[len + 1];**

**strcpy(str, other.str);**

**return \*this;**

**// return;**

**}**

**int main()**

**{**

**String str1("abc");**

**String str2("123");**

**String str3("456");**

**str1.show();**

**str2.show();**

**str3.show();**

**(str3 = str1) = str2;**

**cout << "str3的内容为：" << endl;**

**str3.show();**

**return 0;**

**}**

**程序执行结果如下：**

**value = abc**

**value = 123**

**value = 456**

**str3的内容为：**

**value = 123**

**deconstruct**

**deconstruct**

**deconstruct**

**str3得到了str2的内容,与我们认识的‘=’运算符逻辑相符。**

**而如果运算符重载返回的是对象时,**

**//这是返回类型为对象的情况**

**#include <iostream>**

**using namespace std;**

**class String**

**{**

**private:**

**char\* str;**

**int len;**

**public:**

**String(const char\* s);//构造函数声明**

**String operator=(const String& another);//运算符重载,此时返回为空**

**void show()**

**{**

**cout << "value = " << str << endl;**

**}**

**/\*copy construct\*/**

**String(const String& other)**

**{**

**len = other.len;**

**str = new char[len + 1];**

**strcpy(str, other.str);**

**cout << "copy construct" << endl;**

**}**

**~String()**

**{**

**delete[] str;**

**cout << "deconstruct" << endl;**

**}**

**};**

**String::String(const char\* s)**

**{**

**len = strlen(s);**

**str = new char[len + 1];**

**strcpy(str, s);**

**}**

**String String::operator=(const String &other)**

**{**

**if (this == &other)**

**return \*this;**

**// return;**

**delete[] str;**

**len = other.len;**

**str = new char[len + 1];**

**strcpy(str, other.str);**

**return \*this;**

**// return;**

**}**

**int main()**

**{**

**String str1("abc");**

**String str2("123");**

**String str3("456");**

**str1.show();**

**str2.show();**

**str3.show();**

**(str3 = str1) = str2;**

**cout << "赋值后str3的内容为：" << endl;**

**str3.show();**

**return 0;**

**}**

**程序执行结果如下：**

**value = abc**

**value = 123**

**value = 456**

**copy construct**

**copy construct**

**deconstruct**

**deconstruct**

**赋值后str3的内容为：**

**value = abc**

**deconstruct**

**deconstruct**

**deconstruct**

**str3只得到了str1的内容,并没有得到str2的内容,这是因为执行(str3=str1)后,因为返回的是对象(一个临时对象,str3的一个拷贝),不是引用,所以此时str3不在后面的‘=str2’的操作中,而是str2对一个临时对象赋值,所以str3的内容保持不变(等于str1)。**

1. **深拷贝与浅拷贝（https://blog.csdn.net/caoshangpa/article/details/79226270）**