**C++**

目录

[1 关键字 2](#_Toc482813777)

[1.1 volatile 2](#_Toc482813778)

[1.2 auto 2](#_Toc482813779)

[1.3 static 3](#_Toc482813780)

[1.4 强制类型转换 3](#_Toc482813781)

[1.5 Inline 4](#_Toc482813782)

[1.6 Const 5](#_Toc482813783)

[1.7 explicit禁止隐式转换 5](#_Toc482813784)

[2 存储 5](#_Toc482813785)

[2.1 Struct/classs变量空间对齐 5](#_Toc482813786)

[2.2 存储区划分 6](#_Toc482813787)

[3 函数 6](#_Toc482813788)

[3.1 传参：值/指针/引用 6](#_Toc482813789)

[3.2 缺省参数 6](#_Toc482813790)

[3.3 函数指针 6](#_Toc482813791)

[3.4 函数重载 6](#_Toc482813792)

[3.5 函数模板 6](#_Toc482813793)

[4 类 7](#_Toc482813794)

[4.1 构造/析构 7](#_Toc482813795)

[4.2 运算符重载 7](#_Toc482813796)

[4.3 友元 8](#_Toc482813797)

[4.4 类模板 8](#_Toc482813798)

[4.5 继承 9](#_Toc482813799)

[4.6 虚函数与纯虚函数 10](#_Toc482813800)

[5 标准模板库STL 10](#_Toc482813801)

[[1]](#endnote-1)

1. 关键字
   1. volatile

作为指令关键字，确保本条指令不会因编译器的优化而省略，且要求每次直接读值。

什么情况下需要使用：

1) 并行设备的硬件寄存器；

2) 一个中断服务子程序中会访问到的非自动变量；

3) 多线程应用中被几个任务共享的变量。

* 1. auto

**1)自动数据类型**

auto num = 10； //int类型

auto num = 10.9； //double类型

**2) 作为函数的返回值**

template<class T1,class T2>

auto add(T1 a,T2 b){

return a+b;

}

**3) for循环中用auto**

**4) 注意事项：**

①我们可以使用valatile，pointer（\*），reference（&），rvalue reference（&&） 来修饰auto

1. auto k = 5;
2. auto\* pK = **new** auto(k);
3. auto\*\* ppK = **new** auto(&k);
4. **const** auto n = 6;

②用auto声明的变量必须初始化

1. auto m; // m should be intialized

③auto不能与其他类型组合连用

1. auto int p; // 这是旧auto的做法。

④函数和模板参数不能被声明为auto

1. **void** MyFunction(auto parameter){} // no auto as method argument
2. **template**<auto T> // utter nonsense - not allowed
3. **void** Fun(T t){}

⑤定义在堆上的变量，使用了auto的表达式必须被初始化

1. **int**\* p = **new** auto(0); //fine
2. **int**\* pp = **new** auto(); // should be initialized
3. auto x = **new** auto(); //  no intializer
4. auto\* y = **new** auto(9); // Fine. Here y is a int\*
5. auto z = **new** auto(9); //Fine. Here z is a int\* (It is not just an int)

⑥以为auto是一个占位符，并不是一个他自己的类型，因此不能用于类型转换或其他一些操作，如sizeof和typeid

1. **int** value = 123;
2. auto x2 = (auto)value; // no casting using auto
3. auto x3 = **static\_cast**<auto>(value); // same as above

⑦定义在一个auto序列的变量必须始终推导成同一类型

1. auto x1 = 5, x2 = 5.0, x3='r';  // This is too much....we cannot combine like this

⑧auto不能自动推导成CV-qualifiers（constant & volatile qualifiers），除非被声明为引用类型

1. **const** **int** i = 99;
2. auto j = i;       // j is int, rather than const int
3. j = 100           // Fine. As j is not constant
4. // Now let us try to have reference
5. auto& k = i;      // Now k is const int&
6. k = 100;          // Error. k is constant
7. // Similarly with volatile qualifer

⑨auto会退化成指向数组的指针，除非被声明为引用

1. **int** a[9];
2. auto j = a;
3. cout<<**typeid**(j).name()<<endl; // This will print int\*
4. auto& k = a;
5. cout<<**typeid**(k).name()<<endl; // This will print int [9]
   1. static

**全局静态变量**

在全局变量之前加上关键字static，全局变量就被定义成为一个全局静态变量。

1）内存中的位置：静态存储区（静态存储区在整个程序运行期间都存在）

2）初始化：未经初始化的全局静态变量会被程序自动初始化为0（自动对象的值是任意的，除非他被显示初始化）

3）作用域：全局静态变量在声明他的文件之外是不可见的。准确地讲从定义之处开始到文件结尾。

好处：

定义全局静态变量的好处：

（1）不会被其他文件所访问，修改

（2）其他文件中可以使用相同名字的变量，不会发生冲突。

**局部静态变量**

在局部变量之前加上关键字static，局部变量就被定义成为一个局部静态变量。

1）内存中的位置：静态存储区

2）初始化：未经初始化的全局静态变量会被程序自动初始化为0（自动对象的值是任意的，除非他被显示初始化）

3）作用域：作用域仍为局部作用域，当定义它的函数或者语句块结束的时候，作用域随之结束。

注：当static用来修饰局部变量的时候，它就改变了局部变量的存储位置，从原来的栈中存放改为静态存储区。但是局部静态变量在离开作用域之后，并没有被销毁，而是仍然驻留在内存当中，直到程序结束，只不过我们不能再对他进行访问。

当static用来修饰全局变量的时候，它就改变了全局变量的作用域（在声明他的文件之外是不可见的），但是没有改变它的存放位置，还是在静态存储区中。

**静态函数**

在函数的返回类型前加上关键字static，函数就被定义成为静态函数。

函数的定义和声明默认情况下是extern的，但静态函数只是在声明他的文件当中可见，不能被其他文件所用。

定义静态函数的好处：

（1）其他文件中可以定义相同名字的函数，不会发生冲突

（2）静态函数不能被其他文件所用。 存储说明符auto，register，extern，static，对应两种存储期：自动存储期和静态存储期。 auto和register对应自动存储期。具有自动存储期的变量在进入声明该变量的程序块时被建立，它在该程序块活动时存在，退出该程序块时撤销。

关键字extern和static用来说明具有静态存储期的变量和函数。用static声明的局部变量具有静态存储持续期（static storage duration），或静态范围（static extent）。虽然他的值在函数调用之间保持有效，但是其名字的可视性仍限制在其局部域内。静态局部对象在程序执行到该对象的声明处时被首次初始化。

**类内部的静态变量**

在类内部的变量之前加上关键字static。

1）内存中的位置：静态存储区

2）初始化：要在类的外部初始化

3）作用域：类的公有静态变量，在类的外部也可以通过 类名::变量名 调用。

4）应用changhe ：用类名不实例化对象也可以调用。被类的所有对象所共享。（通常用于估计某类和它的派生类所有对象的个数）

注：类中不能对static数据成员进行初始化，要初始化的话必须在类外进行定义，static数据成员不是通过类构造函数进行初始化的。

**类内部的静态函数**

无法使用this指针

* 1. 强制类型转换

1. **const\_cast**：去掉类型的const或volatile属性。
2. **static\_cast：**

无条件转换，静态类型转换。用于：

基类和子类之间转换：其中子类指针转换成父类指针是安全的;但父类指针转换成子类指针是不安全的。(基类和子类之间的动态类型转换建议用dynamic\_cast)；

基本数据类型转换。enum, struct, int, char, float等。static\_cast不能进行无关类型(如非基类和子类)指针之间的转换；

把空指针转换成目标类型的空指针；

把任何类型的表达式转换成void类型；

static\_cast不能去掉类型的const、volitale属性(用const\_cast)。

1. **dynamic\_cast：**

有条件转换，动态类型转换，运行时类型安全检查(转换失败返回NULL)：

安全的基类和子类之间转换；

必须要有虚函数；

相同基类不同子类之间的交叉转换。但结果是NULL。

例：

BaseClass\* pb2 = new BaseClass();

DerivedClass \*pd21 = static\_cast<DerivedClass \*>(pb2);

//父类->子类，静态类型转换，危险！访问子类m\_szName成员越界

DerivedClass \*pd22 = dynamic\_cast<DerivedClass \*>(pb2);

//父类->子类，动态类型转换，安全的。结果是NULL

1. **reinterpreter\_cast**

仅仅重新解释类型，但没有进行二进制的转换：

转换的类型必须是一个指针、引用、算术类型、函数指针或者成员指针；

在比特位级别上进行转换。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)。但不能将非32bit的实例转成指针；

最普通的用途就是在函数指针类型之间进行转换；

很难保证移植性。

例：

int doSomething(){return 0;};

typedef void(\*FuncPtr)();

//FuncPtr 是一个指向函数的指针，该函数没有参数，返回值类型为 void

FuncPtr funcPtrArray[10];

//10个FuncPtrs指针的数组 让我们假设你希望（因为某些莫名其妙的原因）把一个指向下面函数的指针存入funcPtrArray数组：

funcPtrArray[0] = &doSomething;

// 编译错误！类型不匹配，reinterpret\_cast可以让编译器以你的方法去看待它们：funcPtrArray

funcPtrArray[0] = reinterpret\_castFuncPtr>(&doSomething);

//不同函数指针类型之间进行转换

1. **总结**

去const属性用const\_cast。

基本类型转换用static\_cast。

多态类之间的类型转换用daynamic\_cast。

不同类型的指针类型转换用reinterpreter\_cast。

* 1. Inline

**作用：**代替宏定义，编译展开，减少调用开销，效率高；编译器调用时类型检查，更安全。

**应用场合：**简单函数，有while/switch/递归等都不可。

**注意事项：**

1. 类内定义的简单函数，编译器自动优化为内联函数，无需inline；
2. 类外定义简单函数，需用inline修饰；

class A

{

public:void Foo(int x, int y) { } // 自动地成为内联函数

}

将成员函数的定义体放在类声明之中虽然能带来书写上的方便，但不是一种良好的编程风格，上例应该改成：

// 头文件

class A

{

public:

void Foo(int x, int y);

}

// 定义文件

inline void A::Foo(int x, int y){}

1. Inline是对编译器的建议，编译器可选择忽略；
2. 定义在头文件中。因为内联函数要在调用点展开，所以**编译器必须随处可见内联函数的定义**，要不然就成了非内联函数的调用了。所以，这要求每个调用了内联函数的文件都出现了该**内联函数的定义。**
   1. Const
3. **const成员函数**

任何不会修改数据成员的函数都应该声明为const类型。

const要放在成员函数参数列表的括号()的后面，大括号之前。

int GetValue() const

{

return value;

}

注：const成员函数不能修改类内部成员，但是mutable关键字修饰的成员可以修改。

**2) const成员变量**

必须在构造时初始化。（一旦初始化后不读内存，从代码区的符号表读取）

只能用初始化成员列表的方式赋值。(static变量必须类外定义)

注：不同的对象其const数据成员的值可以不同。所以不能在类声明中初始化const数据成员，因为类的对象未被创建时，编译器不知道const 数据成员的值是什么。

**3）修饰类**

(1)const修饰类对象表示该对象为常量对象，其中的任何成员都不能被修改。对于对象指针和对象引用也是一样。   
(2)const修饰的对象，该对象的任何非const成员函数都不能被调用，因为任何非const成员函数会有修改成员变量的企图。

**4）修饰变量和指针**

(1)只可读不可写：const int num = 5; int a[num]; //cpp中可以

(2)const int num = 5; int \*p = (int\*)(&num); \*p=4; //num的值依然为5，C中的强制转换方式不可行

更改：int \*p = const\_cast<int\*>(&num); \*p = 4;

//强制去掉const属性，仍然不可以改num的值，内存中的值改为4，但是读取const值时直接从常量列表没有从真实内存中读取类似define读取（c++编译器做的优化）

define在内存中没有实体，在常量表（在代码区通过寄存器产生常量）中

(3) int const \*p 或 const int \*p（指向常量的指针变量：指向常量的地址，指向的地址可以改；可以指向常量或变量，但不可以更改指针指向的值，只读不可写强制类型不起作用）

int \* const p（指向变量的常量指针：指针是常量，指向的地址不能变；只能指向变量，指向的值**可**以改，类型要匹配，可读可写的权限，强制类型发生作用）const int \* const p （指向常量的常量指针，可以指向常量或变量，但不可更改地址和值）

* 1. explicit禁止隐式转换

class A{

public:

A(int b):a(b){}

private:

int a;

}

A test = 1; //是可以的，默认调用A::A(int a)这个构造函数，explicit加在构造函数之前，编译器就不自作聪明的自动调用就必须是A test(1)才可以。

存储

* 1. Struct/classs变量空间对齐

首先，默认的块对齐长度是4字节，此外也可以通过#pragma pack（x）修改，同时，自己修改完之后记得使用#pragma pack（）恢复默认值。此外，又分了几种情况：

（1）如果有虚函数，则会存储虚函数表的地址（仅一个地址，4字节或8字节），该地址是存放在类内存的最开始的地方。

（2）如果**出现比x大的非静态成员变量**（静态成员变量不算在类内存里面，而是和函数放在一起的），则**对齐长度取较大类型的尺寸值**。

（3）所有成员变量之间的先后位置不受虚函数影响，而且存储的先后位置和定义变量时的先后顺序是一样的（这样似乎就可以通过**调整成员变量定义的先后顺序来节省内存**啦）。

（4）所有变量的存储地址都是该变量类型的尺寸的整数倍。

（5）定义空类型，没有成员函数和成员变量，sizeof为1，在内存中占位，否则无法使用。

* 1. 存储区划分

**栈 -** 就是那些由编译器在需要的时候分配，在不需要的时候自动清楚的变量的存储区。里面的变量通常是局部变量、函数参数等。

**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。

**全局区（静态区）**, 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。程序结束释放。

**常量存储区，**他们里面存放的是常量，不允许修改（当然，你要通过非正当手段也可以修改）- 程序结束释放。

函数

* 1. 传参：值/指针/引用

值：复制值传给函数的栈，使用后释放

指针：将变量的地址入栈

引用：不存在进栈出栈操作，效率高，const int &a;//常量引用，效率高，但指定不能改变原数值。适用场合：大数组/对象作为函数参数时。

* 1. 缺省参数

int sum(int x=0, int y=100, int z=0)

{

return x+y+z;

}

某一个参数一旦设置了缺省值，其后而的参数也必须设置缺省值。

* 1. 函数指针

1. **定义：**

Double pam(int);//函数原型

Double (\*pd)(int);//\*pd与pam类似，pam为函数，\*pd也为函数，则pd为这类函数的指针。

2）int(\* z( int x, int(\*y)(int)))(int)

--z为函数，其参数为x和函数指针，返回值为同类型的函数指针 Y z(int x, Y)--Y为函数指针

int(\* & z( int x, int(\* &y)(int)))(int)-- 返回函数指针的引用

3)

int(\*p)(int a, int b)(add);

相当于：

int(\*p)(int a, int b);

p = add;//用函数add初始化函数指针

* 1. 函数重载

函数重载只能通过参数重载，不可通过返回值重载（即不能只有返回值不同）

* 1. 函数模板

**定义模板**

template<typename T>

T const& max(T const& a, T const& b)

{

return a < b ? b : a;

}

或者

template<typename T>

auto const& max(T const& a, T const& b) //函数的参数不允许自动类型

{

return a < b ? b : a;

}

或者

template<typename T1, typename T2>

auto max(T1 const& a, T2 const& b)

{

return a < b ? b : a;

}

通用函数可变参数模板： //通用可变参数模板 处理不限定个数的参数，处理不同类型

template<typename T, typename...Args>

void showall(const T &value, const Args &...args)

{

std::cout << value << std::endl;

showall(args...); //继续传递

}

可变参数模板:

template <typename T>

void show(T t, ...)

{

std::cout << t << std::endl;

}

**函数模板的使用**

add(T a，T b);

**模板的重载**

模板的重载会根据数据类型自动匹配

//函数模板可以对类型进行优化重载，根据类型会覆盖

//如果仍然要使用模板函数，需要实例化

如：

一个类中：

int add(int a,int b);

template<class T>

T add(T a,T b);

//这种情况下如果ab是int类型，则会默认调用普通函数，要用模板函数则要实例化模板函数：add<int>(a,b);

**函数模板覆盖**

函数模板可以被覆盖，覆盖时候可以加上类型覆盖

如：

父类中：

template<typename T>

void swap(T &a, T &b)

子类中：

//template<> //这里可有可无

void swap(int &int1,int &int2)

类

* 1. 构造/析构

1. **析构函数：**对象从内存消失时自动调用析构函数，只有一个，没有参数。
2. **构造函数：**

**应用：**实例化类时自动调用; 常用来初始化成员变量; 初始化成员列表（const 常量， 引用必须用成员列表赋初值）; 默认参数在头文件中声明;

注：用new给指针分配内存要统一new或new[]，在析构中释放内存；构造函数没有执行之前，没有this指针。

**初始化成员列表**：按变量的声明顺序初始化，而不是初始化列表的顺序。

**拷贝构造函数：**类名(const 类名 &a);

**浅拷贝：**两个对象之间成员变量简单的赋值，默认的拷贝构造函数，成员变量有指针使用浅拷贝时，只是简单的指针赋值，不存在内存地址分配，则两个对象的指针指向同一个内存区域，析构释放内存时释放两次，出错。

**深拷贝：**不同对象的指针成员指向不同的内存地址，拷贝时不是指针赋值，而是将内存的内容拷贝过来。如果类成员有指针，需要自己实现拷贝构造函数，不然存在浅拷贝的风险。

只需在函数声明后加上“**=default**;”，就可将该函数声明为 defaulted 函数，编译器将为显式声明的 defaulted 函数自动生成函数体。

只需在函数声明后加上“**=delete**;”，就可将该函数禁用。

**3）拷贝赋值运算符：**

A& A::operator=(const A& a); //返回A的引用

* 1. 运算符重载

**1）类成员操作符：**

A operator+(A&){return A(data+a.data);};

A1 = A2 + A3;

A1：+的返回值； A2：调用运算符+的对象； A3：+的参数

**2）友元操作符：**

friend A operator+(A&,A&){ return A(a1.data+a2.data);};//友元函数

A1 = A2 + A3;

A1：+的返回值； A2： +的第一个参数； A3：+的第二个参数；

class A{

public:

A(int d):data(d){}

A operator+(A&){return A(data+a.data);};//成员函数

A operator-(A&){return A(data-a.data);};

A operator\*(A&){return A(data\*a.data);};

A operator/(A&){return A(data/a.data);};

A operator%(A&){ return A(data%a.data);};

friend A operator+(A&,A&){ return A(a1.data+a2.data);};//友元函数

friend A operator-(A&,A&){return A(a1.data-a2.data);};

friend A operator\*(A&,A&){return A(a1.data\*a2.data);};

friend A operator/(A&,A&){return A(a1.data/a2.data);};

friend A operator%(A&,A&){return A(a1.data%a2.data);};

private:

int data;

};

//关系运算符重载

bool operator == (const A& );

bool operator != (const A& );

bool operator < (const A& );

bool operator <= (const A& );

bool operator > (const A& );

bool operator >= (const A& );

//逻辑运算符重载

bool operator || (const A& );

bool operator && (const A& );

bool operator ! ();

//单目运算符重载

A& operator + ();

A& operator - ();

A\* operator & ();

A& operator \* ();

//自增减运算符重载

A& operator ++ ();//前置++

A operator ++ (int);//后置++

A& operator --();//前置--

A operator -- (int);//后置—

//位运算符重载

A operator | (const A& );

A operator & (const A& );

A operator ^ (const A& );

A operator << (int i);

A operator >> (int i);

A operator ~ ();

//赋值运算符重载

A& operator += (const A& );

A& operator -= (const A& );

A& operator \*= (const A& );

A& operator /= (const A& );

A& operator %= (const A& );

A& operator &= (const A& );

A& operator |= (const A& );

A& operator ^= (const A& );

A& operator <<= (int i);

A& operator >>= (int i);

//内存运算符重载

void \*operator new(size\_t size);

void \*operator new(size\_t size, int i);

void \*operator new[](size\_t size);

void operator delete(void\*p);

void operator delete(void\*p, int i, int j);

void operator delete [](void\* p);

//特殊运算符重载

A& operator = (const A& );

char operator [] (int i);//返回值不能作为左值

const char\* operator () ();

T operator -> ();

//类型转换符

operator char\* () const;

operator int ();

operator const char () const;

operator short int () const;

operator long long () const;

friend inline ostream &operator << (ostream&, A&);//输出流

friend inline istream &operator >> (istream&, A&);//输入流

* 1. 友元

**1）语法**

声明： friend + 普通函数声明

实现位置：可以在类外或类中

实现代码：与普通函数相同

调用：友元函数可以直接被调用，不需要通过对象或指针；但要访问非static成员时，需要对象做参数。

**2）类Y的一个成员函数为类X的友元函数**

声明位置：声明在公有中 （本身为函数）

声明：friend + 成员函数的声明

调用：先定义Y的对象y---使用y调用自己的成员函数---自己的成员函数中使用了友元机制

**3）注意**

友元函数的声明要friend关键字，而定义不需要friend关键字。

成员函数有this指针，而友元函数没有this指针。

友元函数是不能被继承的，就像父亲的朋友未必是儿子的朋友。

* 1. 类模板

**类模板的格式为：**

template<class 形参名，class 形参名，…>

class 类名

{ ... };

如

template<class T>

class A

{

public:

T a;

T b;

T hy(T c, T &d);

};

类模板中有默认参数类型的情况

template<class T=int>

class A{};

类模板中有两种参数的情况

template<class T1,class T2>

class myclass{};

**类模板对象的创建：**

A<int> m;在类A后面跟上一个<>尖括号并在里面填上相应的类型，这样的话类A中凡是用到模板形参的地方都会被int 所代替。当类模板有两个模板形参时创建对象的方法为A<int, double> m;类型之间用逗号隔开。

**类模板对象的继承：**

继承时候要明确类型

1)模板类继承模板类

template<class T1>

class A{

public:

A(T1 t1):x(t1){};

T1 x;

};

template<class T2>

class B:public A<T2>//类模板类型会传递，T1，T2为一个类型

{

public:

B(T1 t1,T2 t2):A(t1),y(t2) {};

T2 y;

};

2)普通类继承模板类

template<class T1>

class A{

public:

A(T1 t1):x(t1){};

T1 x;

};

class B:public A<int>//这里A必须实例化

{

public:

B(int t1,int t2):A<int>(t1),y(t2) {};

int y;

};

3)模板类继承普通类

class A

{

public:

A(aa):a(aa){};

int a;

}

template<class T1>

class B:public A

{

public：

B(T1 t1,int aa):A(aa),b(t1){};

T1 b；

}

**类模板作为函数的参数：**

函数的形参可以是类模板对象或是类模板对象的引用，实参则是类模板实例化的模板类对象。

当一个函数拥有类模板参数的时候，这个函数也必须是函数模板。

如：

template<class T> //类模板

class A

{

public:

T x;

A(T xx):x(xx){};

}

template<class T>

void funcA(A &a) //类模板的函数

{

std::cout << a.x << std::endl;

}

**含有虚函数的模板类：**

联系虚函数的概念，没什么特别

template<class T1>

class A

{

public:

T1 x;

A(T1 t1):x(t1){};

virtual func(){};

};

**模板抽象类：**

含有纯虚函数的模板类

* 1. 继承

**1）final,override关键字**

final限定的类不能被继承

语法：struct B1 final {};

final限定的函数，不能被子类重写。

语法：void f() final {};

override关键字限定的函数，父类必须有这个函数。

语法：void g(double) override {};

1. **继承方式与访问权限**

**public继承**

基类成员 继承为 子类同访问权限成员，基类private成员除外，子类无权访问

**private继承**

基类成员 继承为 子类private成员，基类private成员除外，子类无权访问

**protected继承**

基类public成员 继承为 子类的protected成员，其它成员的访问权限不变，基类private成员除外，子类无权访问

1. **构造函数/析构函数执行顺序**

建立对象时，首先调用基类的构造函数，然后在调用下一个派生类的构造函数，依次类推；

析构对象时，其顺序正好与构造相反。

1. **同名变量与函数**

同名变量相互独立，同名函数重载。看变量是子类还是父类，函数执行就近原则。

* 1. 虚函数与纯虚函数

**1）虚函数的意义**

同样一个指针能操作很多子类，实现多态，适配器模式。

定义一个函数为虚函数是为了允许用基类的指针来调用子类的这个函数。

虚函数只能借助于指针或者引用来达到多态的效果。

析构函数可以是虚函数，这样可以通过父类指针正确释放子类的指针，防止内存泄漏。

**2）虚函数的相关知识**

基类和继承类的虚函数必须函数名，变量类型，返回值都一致。

友元不可以是虚函数，但另一个类的该函数可以是虚函数。【因为在一个类里声明友元时 由于友元不是自己的成员函数 自然在自己的类里不能把它声明为虚函数

但是在友元本身可以是虚函数。注意，友元是另外一个类的成员函数(在那个类里 它可以定义为虚函数)

这个类将它声明为自己的友元,只是让它可以存取自己的私有变量】

构造函数不能为虚函数

虚函数的本质是指针，所以虚函数占用四个字节（32位机）

**3）虚函数的使用方法**

1）在基类用virtual声明成员函数为虚函数。

这样就可以在派生类中重新定义此函数，为它赋予新的功能，并能方便地被调用。在类外定义虚函数时，不必再加virtual。

2）在派生类中重新定义此函数，要求函数名、函数类型、函数参数个数和类型全部与基类的虚函数相同，并根据派生类的需要重新定义函数体。

C++规定，当一个成员函数被声明为虚函数后，**其派生类中的同名函数都自动成为虚函数**。因此在派生类重新声明该虚函数时，可以加virtual，也可以不加，但习惯上一般在每一层声明该函数时都加virtual，使程序更加清晰。如果在派生类中没有对基类的虚函数重新定义，则派生类简单地继承其直接基类的虚函数。

3）定义一个指向基类对象的指针变量，并使它指向同一类族中需要调用该函数的对象。

4）通过该指针变量调用此虚函数，此时调用的就是指针变量指向的对象的同名函数。

通过虚函数与指向基类对象的指针变量的配合使用，就能方便地调用同一类族中不同类的同名函数，只要先用基类指针指向即可。如果指针不断地指向同一类族中不同类的对象，就能不断地调用这些对象中的同名函数。

**4）纯虚函数**

（方法：virtual ReturnType Function()= 0;）

含有纯虚拟函数的类称为抽象类，它不能生成对象。

定义纯虚函数是为了实现一个接口，起到一个规范的作用，规范继承这个类的程序员必须实现这个函数。

**5）抽象类**

抽象类不能生成对象，但可以实例化指针。

抽象类的子类如果**不实现所有父类的纯虚函数**，这个子类仍然为抽象类。

抽象类是一个指向虚函数列表的指针，**所以抽象类只占四个字节。**

标准模板库STL

(b)

**算法**

[1 位操作 14](#_Toc482813840)

[2 Model Training Based on Feature-Fusion 14](#_Toc482813841)

[2.1 Image Sampling and Tagging 14](#_Toc482813842)

[2.2 Feature-Fusion Based on HOG and LBP 14](#_Toc482813843)

[2.3 Classification Based on SVM 16](#_Toc482813844)

[3 Insulator Detection 17](#_Toc482813845)

[4 Image Preprocess 17](#_Toc482813846)

[5 Candidate Windows and Prediction 18](#_Toc482813847)

[6 Image Fusion Based on NMS 18](#_Toc482813848)

[7 Image Mergence 18](#_Toc482813849)

[8 Experimental Results 19](#_Toc482813850)

[9 Conclusion 20](#_Toc482813851)

[References 20](#_Toc482813852)

[[2]](#endnote-2)

1. 位操作
2. 获取数字的第i位：num & ( 1 << i )
3. 置位数字第i位： num | ( 1 << i )
4. 清零数字第i位： num & ~( 1 << i )
5. 生成32位（系统）1： ~0
6. > i 位为1， 其余为0：~0 << ( i + 1 )
7. <= i 都为1， 其余为0：( 1 << i ) – 1
8. 最高位至i位（含）清零：( 1 << i) -1 & num
9. 将i位（含）到0位（含）清零：

~ ( ( 1 << i + 1 ) -1 ) -& num

1. 0.1012  = 1 \* 1/2 + 0 \* 1/4 + 1 \* 1/8
2. C = C & (C - 1) 循环，看C中有多少1
3. 排序
   1. 快排

vector<int> a = { 0,5,1,2,5,8,2,4,10 };

template<typename iterator>

void quick\_sort(iterator left, iterator right) {

if (left < right) {

auto l = left, r = right;

auto mid\_val = \*left;

while (l < r) {

while (l < r && \*r >= mid\_val) r--;

if (l < r) \*(l++) = \*r;

while (l < r && \*l < mid\_val) l++;

if (l < r) \*(r--) = \*l;

}

\*l = mid\_val;

if(l > left)

quick\_sort(left, l - 1);

if(right > l)

quick\_sort(l + 1, right);

}

}

void main() {

quick\_sort(a.begin(), a.end() - 1);

int b = 0;

}

* 1. 归并排序

template<typename iterator, typename type>

void merge(iterator begin, iterator mid, iterator end) {

vector<type> num2;

copy(mid, end, back\_inserter(num2));

auto lr = mid, rr = num2.end(), r = end ;

while (lr > begin && rr > num2.begin()) {

\*(--r) = \*(lr-1) >= \*(rr-1) ? \*(--lr) : \*(--rr);

}

while (rr > num2.begin())

\*(--r) = \*(--rr);

}

template<typename iterator>

void merge\_sort(iterator begin, iterator end) {

if (begin >= end - 1) return;

auto mid = begin + (end - begin) / 2;

merge\_sort(begin, mid);

merge\_sort(mid, end);

merge<vector<int>::iterator, int>(begin, mid, end);

}

void main() {

vector<int> a = { 5,1,2,5,8,2,4 };

merge\_sort(a.begin(), a.end());

int v = 0;

}

* 1. Classification Based on SVM

There are many classification algorithms, such as SVM classifier, AdaBoost, k neighbor, etc. The SVM algorithm is used because the number of the training dataset is small and there are only insulator and background two kinds of samples. Besides, SVM algorithm is suitable for both the linear and nonlinear classification.

For linear separable samples, there has a hyperplane  to separate the samples completely.

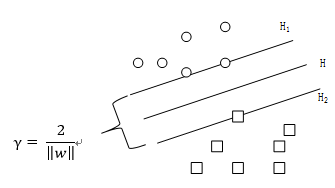


Fig. 5: SVM linear classification

Circles and squares represent two kinds of samples respectively in Fig. 5. By finding the parameters *w* and *b* which are calculated by formula (14), the hyperplane , the straight, and the straight can be found, which are parallel to each other.

 (7)

The optimal classification hyperplane can allow the fault samples. At this time the largest interval is equivalent to calculating the minimum of , where *C* is the penalty factor.

For nonlinear classification, SVM will find the optimal separating hyperplane by using nonlinear transformation which can map the input variables with low dimension into high-dimensional feature space. Solving parameters *w* and *b*, will involve calculating the , which needs to calculate the inner product of sample *xi* and *xj* in high dimension. Because of the difficulty in calculating the inner product, the kernel function is introduced whose expression is as follows

 (8)

There are mainly linear kernel, polynomial kernel, Gaussian kernel (RBF kernel), Laplace kernel, Sigmoid kernel and so on. This method uses the RBF kernel, whose function is as follows:

 (9)

In training process, 700 local sub insulator images are as the positive samples and 2100 background images are as the negative samples. The performance of classification is mainly related to the *C* and , so the SVM parameters are optimized by using the K cross-validation. The optimal parameters based on HOG, LBP and fusion feature are close to each other, therefore *C* and are set to 4.0 and 1.0 uniformly. The test dataset of classification includes 300 positive images and 900 negative images. The results shown in Table 1 indicate that the classification accuracy based on fusion feature is better than that using the single feature.

Table 1: The classification accuracy of Hog, LBP, HOG+LBP

|  |  |
| --- | --- |
| Feature | Accuracy |
| HOG | 75.3% |
| LBP | 76.9% |
| HOG + LBP | 83.5% |

1. Insulator Detection

For locating the insulator strings, the candidate sub-windows need to be extracted from the input image, and the training model is then used to judge whether the candidate windows contain insulators or not. If there contains insulator, the sub-windows will be saved. The redundant windows will then be removed by the window-fusing algorithm. The detection process is as shown in Fig. 6. The sliding window method is used to get the candidate sub-windows, which is a simple way. Before getting the candidate windows, the threshold segmentation and morphological operation are firstly used to preprocess the image, which can reduce the number of sub-windows that need to do feature extraction and prediction, which can save the detection time. Then, the model trained by SVM classifier is used to predict the candidate windows and the NMS algorithm is used to fuse the windows. Because of the local sub-images of insulators located firstly, the linear fitting is finally conducted to merge windows and the final location of the insulator strings can be marked.



Fig. 6: The detection phase

1. Image Preprocess

There have two main advantages of preprocessing. The one is to save detecting time. When the candidate windows are obtained and all the values of the sub windows are equal to zero, it is unnecessary to implement feature extraction and prediction. While the images have many elements similar to sky or thin wires, this method is more effective. However, when the backgrounds are all elements such as grassland or pylons, this method does not work. It is necessary to preprocess the images firstly, which is less time-consuming and has no effect on the detection speed. Preprocessing can help find the sub-images which contain the elements such as the sky or thin wires, so it is unnecessary to collect these negative images. Thus the other advantage is that it can reduce the number of negative samples, reduce the complexity of the training model, speed up the classification, and at the same time improve the classification accuracy even though the glass insulators are similar to the sky.

In the preprocessing stage, the main steps are as follows: apply the Otsu threshold segmentation, make the morphological close operation, and get the mask image after reverse operation, finally multiply the mask image by the original image to get the final image to be detected. The preprocessing results are shown in Fig. 7.

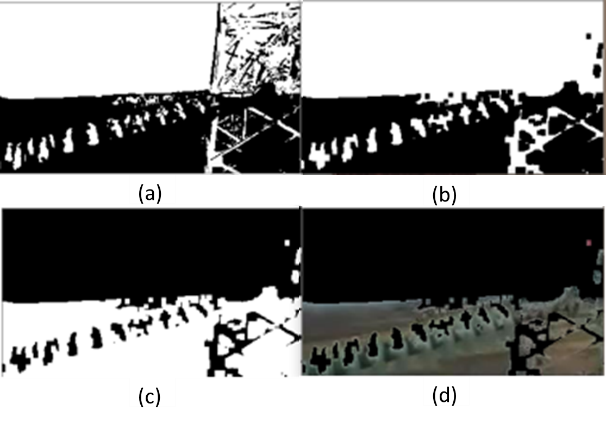


Fig. 7: Image preprocessing. (a) Otsu segmentation.

(b) Morphological close operation. (c) Mask image by reverse operation. (d) Image to be detected

The threshold segmentation results in Fig. 7(a) show that, it is difficult to separate the insulators completely from the complex backgrounds by segmentation technology, and difficult to extract the edge or shape characteristics such as the ellipse characteristics, let alone detect the insulator strings when backgrounds and insulators are similar to each other. In this paper, the threshold segmentation and morphological technology are only to reduce the number of candidate sub-windows to be detected, and the whole detection process of insulator strings do not excessively rely on the results of segmentation.

1. Candidate Windows and Prediction

There are many methods to get the candidate windows, such as the selective search algorithm and the sliding window method. Because the candidate windows will contain the whole multi-angle insulators with much background information by the selective search method, which has conflicts with our training model by using the local sub images of insulator strings. Therefore, the sliding window method is adopted which is also a simple way. In the experiment, sliding windows with five scales are used because of the uncertain scales of insulators.

After saving the candidate sub-windows, the function of parallel\_for\_ is then used, which can get the classification probability of each candidate windows in parallel. The sub-windows, whose prediction probability is greater than the threshold, are saved to a global vector and then used to fuse. Owing to the dual-core CPU, the speed of prediction is increased about 1.5 times.

1. Image Fusion Based on NMS

The non-maximum suppression (NMS) is an algorithm which can search the local maximum values and restrain the non-maximum elements. Based on the classification probabilities of the candidate windows, the sub-windows with higher probability can be retained and the rests with lower probability are removed, which can reduce the number of final candidate windows to reduce the computational complexity of the window merging.

Its main steps are as follows: sort and traverse the sub-windows according to the possibilities if they contain insulators; retain the windows with bigger probability and remove the ones with smaller probability, where the overlaps between them are greater than the threshold.

The results of NMS algorithm are shown in Fig. 8, where the number of candidate sub-windows is 25 before NMS, and changes to 11 after the window fusion.

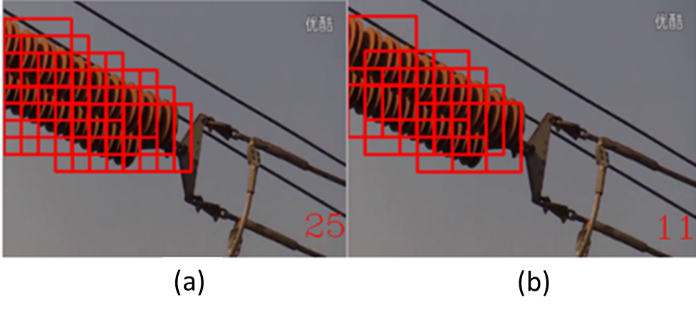


Fig. 8: (a) Before NMS. (b) After NMS

1. Image Mergence

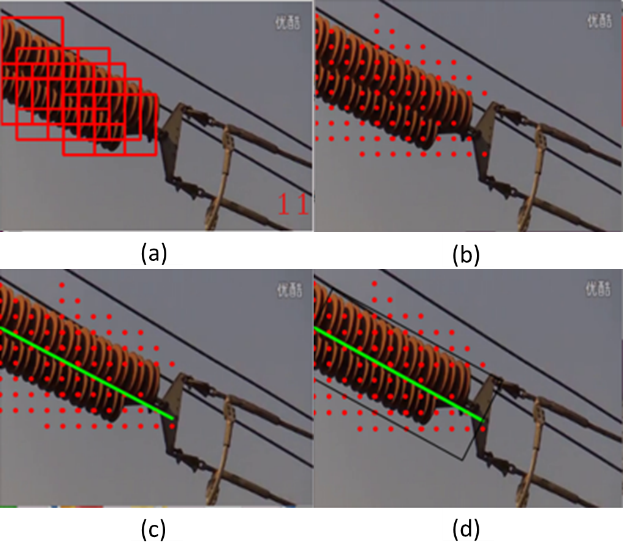


Fig. 9: Image mergence by linear fitting. (a) Final candidate windows after NMS. (b) The set of vertices. (c) The center line. (d)The location of the insulator

Finally, the final candidate sub-windows after NMS need to be merged to get the final location of insulator strings. The main steps are as follows:

1. Group the final candidate sub-windows. Adjacent windows are divided into the same group if having overlaps with each other. There may be many groups because the number of insulator strings is more than one in an image;
2. Storage the vertex coordinates of every sub-windows in the same group into a vector, while the number of windows in a group is greater than 1. The number of coordinate vectors is the same with groups’;
3. Get the center line of each group by using linear fitting based on least squares method;
4. Do statistics for the distances of all points to the center line within the same group, and calculate the average and maximum of the distances, and then combine them linearly to get the width of the insulators;
5. Mark the location boxes in the image according to the center line and the width of insulators.

The results of window mergence are shown in Fig. 9.

1. Experimental Results

To verify the effectiveness of the proposed method in this paper, we locate insulators from three aerial videos of high voltage transmission lines. From the Fig. 10, it can be observed that three types of insulators (including glass insulators) with different angles are located accurately. For the insulators that are hard to segment and extract characteristics, this method can still detect them.

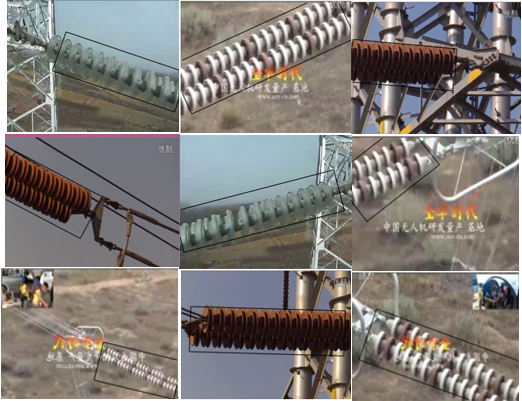


Fig. 10: The insulator detection

In the detection test, we choose 500 images from three videos, 400 of them contain insulators, others are absolute backgrounds. There are 430 insulators in total because of some images containing more than one insulator. For those insulators that are too far or too close in the image and their characteristics are barely distinguishable, they will be excluded as miss detections.

The final detection results are shown in Table 2. From the results, we can see that our fusion feature has better performance on detection accuracy, almost 90%, in comparison with the single feature based on HOG or LBP.

Table 2: The detection results of Hog, LBP, HOG+LBP

|  |  |  |  |
| --- | --- | --- | --- |
| Feature | HOG | LBP | HOG + LBP |
| Right detection | 366 | 352 | 383 |
| Right detection rate | 85.1% | 81.8% | 89.1% |
| Wrong detection | 34 | 13 | 9 |
| Wrong detection rate | 7.9% | 3.0% | 2.1% |
| Miss detection | 64 | 78 | 47 |
| Miss detection rate | 14.9% | 18.1% | 10.9% |

The performance of the localization mainly depends on the accuracy of the classification, as shown in Fig. 11 giving some miss and wrong recognition results. Miss recognition is mainly because the HOG and LBP features has not rotation invariance. Besides samples can not contain insulator images with every angle and scale, resulting in bad influences on recognition results. Wrong recognition is because some backgrounds and insulators are similar on the shape and texture which are difficult to distinguish. So further improvement in classification accuracy is needed.

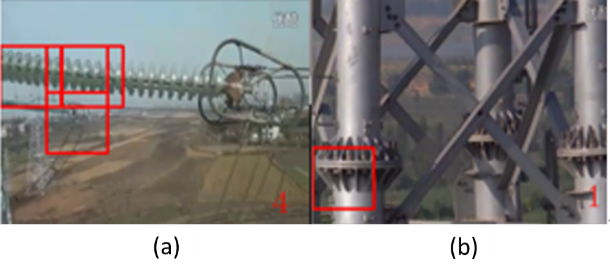


Fig. 11: (a) Miss recognition. (b) Wrong recognition

In addition, we test the running time of this method. For the images including some elements such as the sky, the detection time is about 800 ms because of the preprocessing stage. For the images with complex backgrounds, this method need to predict all candidate sub-windows, which is more time-consuming, about 2 s. The most time-consuming parts in the algorithm to predict a candidate window are HOG, LBP feature extraction, and the SVM classification. The LBP feature extraction consumes most among them and needs to be improved.

Table 3: The consuming time

|  |  |
| --- | --- |
| Detecting an image | |
| No preprocessing | 2 s |
| Preprocessing | 800 ms~2 s |
| Main parts while predicting a sub-window | |
| HOG | 5 ms |
| LBP | 10 ms |
| SVM | 5 ms |

1. Conclusion

In this paper, a detection method of insulator stings for aerial inspection has been developed based on feature-fusion. In the training stage, the local sub images of insulator strings are collected and tagged. The fusion feature is then combined by the HOG feature and LBP feature after PCA dimension reduction separately. The SVM classification algorithm is adopted to get a training model. At the detection phase, threshold segmentation and morphological operation are used to preprocess the images, which can speed up the detection process. The candidate sub windows are extracted by the sliding window method, which are then predicted by SVM in parallel. The NMS algorithm is then used to fuse the candidate windows. The final positions of the insulator strings are finally obtained by the linear fitting. The results of the experiment indicate that the proposed method can locate multi-angle insulators under the complex backgrounds, and that the detection accuracy can be guaranteed satisfactorily.

References

1. Z. Zhao, S Jin, Y. Liu. Aerial insulator image edge extraction method based on NSCT [J]. *Chinese Journal of Scientific Instrument*, 2012, 33(9): 2045-2052.
2. H.R. Jiang, L.J. Jin, S.J. Yan. Recognition and fault diagnosis of insulator string in aerial images [J]. *Mechanical & Electrical Engineering Magazine*, 2015, 32(2): 274-278.
3. L.Y Yu, B. Yao, W.H. Wu, H.L. Liu, W.Q. Tan. Insulator identification method based on multi-feature [J]. *Insulators and Surge Arresters*, 2016(3): 79-83.
4. J. Zhang, J. Han, Y. Zhao, *et al.* Insulator recognition and defects detection based on shape perceptual [J]. J*ournal of Image and Graphics*, 2014, 19(8): 1194-1201.
5. Z. Zhao, W. Le. Aerial insulator string image automatic location method [J]. *Chinese Journal of Scientific Instrument*, 2014, 35(3): 558-565.
6. Z. Zhao, N. Liu, L. Wang. Localization of multiple insulators by orientation angle detection and binary shape prior knowledge [J]. *IEEE Transactions on Dielectrics and Electrical Insulation*, 2015, 22(6): 3421-3428.
7. R. Chen, M. Wang, X. Chen. Human detection based on HOG-PCA and LBP characteristics [J]. *Information Technology*, 2015, 2: 101-105.
8. Qu Y.Y, Liu Q, Guo J M, *et al*. HOG and color based pedestrian detection [J]. *Journal of Wuhan University of Technology*, 2011, 33(4): 134-138.
9. Y.J. Zhai, D. Wang, Y. Wu, H.Y. Cheng. Two-stage recognition method of aerial insulator images based on skeleton extraction[J]*. Journal of North China Electric Power University*, 2005, 42(3): 105-110.
10. W. Yang, C. Li, S.L. Yang, H. Zhou. Study on aerial image positioning of wide glass insulator [J]. *Electric Test*, 2016, (15):23-26.
11. S. Liao, J. An. A robust insulator detection algorithm based on local features and spatial orders for aerial images [J]. *IEEE Geoscience and Remote Sensing Letters*, 2015, 12(5): 963-967.
12. Y. Liu, J. Yong, L. Liu, *et al*. The method of insulator recognition based on deep learning [C]. *International Conference on Applied Robotics for the Power Industry*. 2016: 1-5.
13. L. Y. Insulator location and recognition algorithm based on HOG characteristics and SVMD [J]. *Journal of Transportation Engineering and Information*, 2015, 13(4):53-60.

1. [↑](#endnote-ref-1)
2. [↑](#endnote-ref-2)