特殊工具与技术

C++ Primer第五版

第19章

控制内存分配


```
//new表达式
string *sp = new string("a value");   //分配并初始化一个string对象
string *arr = new string[10];         //分配10个默认初始化的string对象
```

new表达式实际执行三个步骤：

- new表达式调用一个名为operator new（或者operator new[]）的标准库函数，分配一块足够大的、原始的、未命名的内存空间。
- 编译器运行相应的构造函数以构造这些对象，并为其传入初始值。
- 对象被分配了空间并构造完成，返回一个指向该对象的指针。

```
delete sp;          //销毁*sp,然后释放sp指向的内存空间
delete[] arr;       //销毁数组中的元素，然后释放对应的内存空间
```

delete实际执行两个步骤：

- 对sp所指向的对象或者arr所指的数组中的元素指向对应的析构函数。
- 编译器调用名为operator delete（或者operator delete[]）的标准库函数释放内存空间。

如果希望控制内存分配的过程，则他们需要定义自己的operator new函数和operator delete函数。

operator new和operator delete接口

```
#include <iostream>

struct MyClass {
        MyClass() {std::cout <<"MyClass constructed\n";}
        ~MyClass() {std::cout <<"MyClass destroyed\n";}
};

int main () {
        MyClass * pt = new (std::nothrow) MyClass;
        delete pt;
}
```

可以通过判断pt是否为空，判断是否有异常

operator new函数和operator delete函数，可以定义为全局作用域，也可以定义为成员函数。

void *operator new(size_t,void*); //不允许重载

重载

```
#include <iostream>     // std::cout

struct MyClass {
      MyClass() { std::cout << "MyClass constructed\n"; }
      ~MyClass() { std::cout << "MyClass destroyed\n"; }
      void* operator new(size_t t) {
            puts("override version:normal...");
            void* m = malloc(t);
            if (m) return m;
            else throw std::bad_alloc();
      }

      void* operator new(size_t t, std::nothrow_t obj) noexcept {
            puts("override version:nothrow");
            //return operator new(t);
            void* m = malloc(t);
            return m;
      }

      void operator delete(void* p) noexcept {
            puts("deleting normal...");
            free(p);
      }

      void operator delete(void* p, std::nothrow_t obj) noexcept {
            puts("deleting nothrow...");
            free(p);
      }
};

int main() {
      MyClass* pt = new (std::nothrow) MyClass;
      delete pt;

      std::cout << std::endl;

      MyClass* pt2 = new MyClass;
      delete pt2;
}
```

```
override version:nothrow
MyClass constructed
MyClass destroyed
deleting normal...

override version:normal...
MyClass constructed
MyClass destroyed
deleting normal...
```

定位new表达式

```
#include <iostream>
class Foo
{
public:
	Foo(int val = 0){_val = val;}

	void writeFoo()
	{
		std::cout << "_val:" << _val  << " address:" << this;
		std::cout << std::endl;
	}
	~Foo() = default;

private:
	int _val;
};

int main(int argc, char* argv[])
{
	//创建char数组，大小为3个Foo
	char* buf = new char[sizeof(Foo) * 3];
	//实例化Foo对象，并将其放置到buf中第1个Foo“位置”处
	Foo* pb = new (buf) Foo(0);
	//实例化Foo对象，并将其放置到buf中第3个Foo“位置”处
	Foo* pb1= new (buf + sizeof(Foo) * 2) Foo(1);
	//实例化Foo对象，并将其放置到buf中第2个Foo“位置”处
	Foo* pb2= new (buf + sizeof(Foo)) Foo(2);

	pb->~Foo();
	//delete pb;

	pb->writeFoo();
	pb1->writeFoo();
	pb2->writeFoo();
}
```

当只传入一个指针类型的实参时，定义new表达式构造对象但是不分配内存

调用析构函数会销毁对象，但是不会释放内存

```
_val:0 address:014DEE90
_val:1 address:014DEE98
_val:2 address:014DEE94
```

运行时类型识别（run-time type identification,RTTI）

typeid运算符，用于返回表达式的类型

dynamic_cast运算符，用于将基类的指针或引用安全地转换成派生类的指针或引用

在可能的情况下，最好定义虚函数而非直接接管类型管理的重任

在条件部分执行dynamic_cast操作可以确保类型转换和结果检查在同一条表达式中完成

```
struct Base {
	virtual ~Base() {};
};

struct Derived : public Base { };

int main()
{
	Base* bp;
	bp = new Derived;  // bp actually points to a Derived object
	//bp = new Base;  // bp points to a Base object
	if (Derived * dp = dynamic_cast<Derived*>(bp)){
		puts(" use the Derived object to which dp points");
	}
	else {  // bp points at a Base object
		puts("  use the Base object to which bp points");
	}
}
```

改写成使用引用类型：

```
void cast_to_ref(const Base& b)
{
	try {
		const Derived& d = dynamic_cast<const Derived&>(b);
		// use the Derived object to which b referred
	}
		catch (bad_cast) {
		cout << "called f with an object that is not a Derived" << endl;
	}
}

int main()
{
	Base* bp;
	//bp = new Derived;  // bp actually points to a Derived object
	bp = new Base;  // bp points to a Base object
	cast_to_ref(*bp);
}
```

使用typeid运算符


```
Derived* dp = new Derived;
Base* bp = dp; //两个指针都指向Derived对象
//在运行时比较两个对象的类型
if (typeid(*bp) == typeid(*dp))
        cout << "指向的是同一种类型" << endl;
//检查运行时类型是否是某种给定的类型
if (typeid(*bp) == typeid(Derived))
        cout << "bp实际指向Derived对象" << endl;

//下面的检查永远是失败的：bp的类型是指向Base的指针
if (typeid(bp) == typeid(Derived)) {

}
```

指向的是同一种类型
bp实际指向Derived对象

当typeid作用于指针时（而非对象），返回的结果是该指针的静态编译时类型

使用RTTI

```
class Base {
        friend bool operator==(const Base&, const Base&);
public:
        //Base的接口成员
protected:
        virtual bool equal(const Base&) const;
        //Base的数据成员和其他用于实现的成员
};
class Derived :public Base {
public:
        //Derived的其他成员
protected:
        bool equal(const Base&) const;
        //Derived的数据成员和其他用于实现的成员
};

bool operator==(const Base& lhs, const Base& rhs)
{
        //如果typeid不同，返回false；否则虚调用equal
        return typeid(lhs) == typeid(rhs) && lhs.equal(rhs);
}

bool Derived::equal(const Base& rhs) const
{
        //我们清楚这两个类型是相等的，所以转换过程不会抛出异常
        auto r = dynamic_cast<const Derived&>(rhs);
        //指向比较两个Derived对象的操作并返回结果
}
```

虚函数的基类版本和派生类版本必须具有相同的形参类型。如果定义一个虚函数equal，则该函数的形参必须是基类的引用。此时，equal函数将只能使用基类的成员，而不能比较派生类独有的成员

typeinfo类：创建对象的唯一途径是使用typeid运算符

type_info的操作	
t1 == t2	如果 type_info 对象 t1 和 t2 表示同一种类型，返回 true；否则返回 false
t1 != t2	如果 type_info 对象 t1 和 t2 表示不同的类型，返回 true；否则返回 false
t.name()	返回一个 C 风格字符串，表示类型名字的可打印形式。类型名字的生成方式因系统而异
t1.before(t2)	返回一个 bool 值，表示 t1 是否位于 t2 之前。before 所采用的顺序关系是依赖于编译器的

```
struct Base {
   virtual ~Base() { }
};
struct Derived : Base { };

int main()
{
      int arr[10];
      Derived d;
      Base *p = &d;

      cout << typeid(42).name() << ", "<<std::endl
                << typeid(arr).name() << ", "<<std::endl
                << typeid(std::string).name() << ", "<<std::endl
                << typeid(p).name() << ", "<<std::endl
                << typeid(*p).name() << endl;

      return 0;
}
```

```
int,
int [10],
class std::basic_string<char,struct std::char_traits<char>,class std::allocator<char> >,
struct Base *,
struct Derived
```

枚举类型：将一组整形常量组织在一起

不限定作用域：enum

限定作用域：enum class（或struct）

```
enum class open_modes { input, output, append };
enum color {red, yellow, green}; //不限定作用域的枚举类型
//未命名的、不限定作用域的枚举类型
enum { floatPrec = 6, doublePrec = 10, double_doublePrec = 10 };
```

限定作用域的枚举类型，在枚举类型的作用域外是不可以访问的

```
enum color { red, yellow, green };
enum stoplight { red, yellow, green };          //错误：重复定义了枚举成员
enum class peppers { red,yellow,green };        //正确：枚举成员被隐藏了
color eyes = green;             //正确：枚举类型的枚举成员位于有效的作用域中
peppers p = green;              //错误：pepers的枚举成员不在有效的作用域中
                                //color::green在有效的作用域中，但是类型错误
color hair = color::red;        //正确
peppers p2 = peppers::red; //正确：显式的使用pappers的red
```

默认情况下，枚举值从0开始，依次加1

```
enum class intTypes {
      charTyp = 8, shortTyp = 16, intTyp = 16,
      longTyp = 32, long_longTyp = 64
};
```

和类一样，枚举也定义新的类型

```
open_modes om = 2;        //错误：2不属于类型open_modes
om = open_modes::input; //正确：input是open_modes的一个枚举成员

int i = color::red;  //正确：不限定作用域的枚举类型的枚举成语隐式地转换成int
int j = peppers::red; //错误：限定作用域的枚举类型不会隐式转换
```

指定enum的大小

```
enum intValues : unsigned long long {
      charType = 255, shortTyp 65535, intTyp = 65535,
      longTyp = 4294967295UL,
      long_longTyp = 18446744073709551615ULL
};
//不限定作用域的枚举类型intValues的前置声明
enum intValues : unsigned long long; //不限定作用域的，必须指定成员类型
enum class open_modes;  //限定作用域的枚举类型可以使用默认成员类型int
enum class _Values;
enum class _Values;          //可以重复声明
enum _ Values;               //错误：intValues已经被声明成限定作用域的enum
enum _ Values :long;         //错误：intValues已被声明为int
```

默认情况下，的enum成员类型是int

对于不限定作用域的enum成员，没有默认类型，我们只知道潜在类型足够大

形参匹配与枚举类型

```
//不限定作用域的枚举类型，潜在类型因机器而异
enum Tokens { INLINE = 128, VIRTUAL = 129 };
void ff(Tokens);
void ff(int);
int main(){
        Tokens curTok = INLINE;
        ff(128);        //精确匹配ff(int)
        ff(INLINE);     //精确匹配ff(Tokens)
        ff(curTok);     //精确匹配ff(Tokens)
        return 0;
}
```

可以将一个不限定作用域的枚举类型的对象或枚举成员转给整形形参

```
void newf(unsigned char);
void newf(int);
unsigned char uc = VIRTUAL;
newf(VIRTUAL);   //调用newf(int)
newf(uc);        //调用newf(unsigned char)
```

不管Tokens的潜在类型到底是什么，
它的对象和枚举成员都提升成int

类成员指针

可以指向类的非静态成员的指针

```
class Screen {
public:
      typedef std::string::size_type pos;
      char get_cursor() const { return contents[cursor]; }
      char get() const;
      char get(pos ht, pos wd) const;
private:
      std::string contents;
      pos cursor;
      pos height, width;
};
```

数据成员的指针

```
//pdata可以指向一个常量（非常量）Screen对象的string成员
const string Screen::*pdata;
//一个指向Screen类的const string成员的指针
pdata = &Screen::contents; //初始化
//auto pdata = &Screen::contenets;
```

初始化后，该指针并没有指向任何对象

使用数据成员指针

```
Screen myScreen, *pScreen = &myScreen;
//.*解引用pdata以获得myScreen对象的contents成员
auto s = myScreen.*pdata;
//->*解引用pdata以获得pScreen多指向对象的contents成员
s = pScreen->*pdata;
```

- 首先解引用成员指针以得到所需的成员
- 然后向成员访问运算符一样，通过对象（.*）或指针(->*)获得成员。

数据成员一般是私有的，通常需要定义一个函数，返回指向该成员的指针

```
class Screen {
public:
      //data是一个静态成员，返回一个成员指针
      static const std::string Screen::*data()
      { return &Screen::contents; }
      //其他成员与之前的版本一致
};
//data()返回一个指向Screen类的contents成员的指针
const string Screen::*pdata = Screen::data();
//获得myScreen对象的contents成员
auto s = myScreen.*pdata;
```

成员函数指针

```
//pmf是一个指针，它可以指向Screen的某个常量成员函数
auto pmf = &Screen::get_cursor;

char (Screen::*pmf2)(Screen::pos,Screen::pos) const;
pmf2 = &Screen::get;//&不能省
```

和普通函数指针不同，不存在自动转换规则

使用成员函数指针

```
Screen myScreen, *pScreen = &myScreen;
//通过pScreen所指的对象调用pmf所指的函数
char c1 = (pScreen->*pmf)();
//通过myScreen对象将实参0,0传给含有两个型参的get函数
char c2 = (myScreen.*pmf2)(0,0);
```

使用成员指针的类型别名

```
//指向Screen成员函数的指针，接受两个pos实参，返回一个char
using Action = char(Screen::*)(Screen::pos,Screen::pos) const;
Action get = &Screen::get; //get指向Screen的get成员

//接受一个Screen的引用，和一个指向Screen成员函数的指针
Screen& action(Screen&, Action = &Screen::get);

Screen myScreen;
//等价调用
action(myScreen);          //使用默认实参
action(myScreen,get);      //使用我们之前定义的变量get
action(myScreen,&Screen::get); //显式传入地址
```

成员指针函数表

```
class Screen {
public:
   //其他接口和实现成员与之前一致
   using Action = Screen&(Screen::*)();
   //光标移动函数
   Screen& home() { cursor = 0; return *this; }
   Screen& forward() { ++cursor; return *this; }
   Screen& back() { --cursor; return *this; }
   Screen& up() { cursor += height; return *this; }
   Screen& down() {cursor -= height; return *this; }

   enum Directions { HOME, FORWARD, BACK, UP, DOWN };
   Screen& move(Directions);
private:
   static Action Menu[];     // 函数表
};

Screen& Screen::move(Directions cm)
{
   // run the element indexed by cm on this object
   return (this->*Menu[cm])(); // Menu[cm] points to a member function
}

Screen::Action Screen::Menu[] = { &Screen::home,
                          &Screen::forward,
                          &Screen::back,
                          &Screen::up,
                          &Screen::down,
                        };
```

```
Screen myScreen;
myScreen.move(Screen::HOME);  // invokes myScreen.home
myScreen.move(Screen::DOWN);  // invokes myScreen.down
```

将成员函数用作可调用对象

```
auto fp = &string::empty; //fp指向string的empty函数
//错误，必须使用.*或->*调用成员指针
find_if(svec.begin(),svec.end(),fp);

//检查对当前元素的断言是否真
if(fp(*it)) //错误：要想通过成员指针调用函数，必须使用->*运算符
```

成员指针不是可调用对象

使用function生成一个可调用对象

```
vector<string*> pvec;
function<bool (const string&)> fcn = &string::empty;
find_if(pvec.begin(),pvec.end(),fcn);

//指向成员函数的对象将被传给隐式的this形参
//假设it是find_if内部的迭代器，则*it是给定范围的一个对象
if(fcn(*it))
//本质上function将函数调用转换成了如下形式：
if((*it).*p()) //假设p是fcn内部的一个指向成员函数的指针
```

使用mem_fn生成一个可调用对象

和function不同，mem_fn可以根据成员指针的类型推断可调用对象的类型

```
//mem_fn(&string::empty)生成一个可调用对象
find_if(svec.begin().svc.end(),mem_fn(&string::empty));
//mem_fn生成的可调用对象含有一对重载的函数调用运算符
auto f = mem_fn(&string::empty); //f接受一个string或者一个string*
f(*svec.begin());  //正确：传入一个string对象，f使用.*调用empty
f(&svec[0]);       //正确：传入一个string的指针，f使用->*调用emtpy
```

使用bind生成一个可调用对象

```
//选择范围中的每个string，并将其bind到empty的第一个隐式实参上
auto it = find_if(svec.begin(),svec.end(),bind(&string::empty,_1));
auto f = bind(&string::empty,_1);
f(*svec.begin());  //正确：实参是一个string,f使用.*调用empty
f(&svec.[0]);      //正确：实参是一个string的指针，f使用->*调用empty
```

嵌套类

一个类可以定义在另一个类的内部

```
class TextQuery::QueryResult {
friend std::ostream& print(std::ostream&, const QueryResult&);
public:
    //无须定义QueryResult::line_no
    //嵌套类可以直接使用外层类的成员
    QueryResult(std::string,
                std::shared_ptr<std::set<line_no>>,
                std::shared_ptr<st::vector<std::string>>);
    //...
};
```

嵌套类是一个独立的类，与外层类基本没什么关系

定义嵌套类的成员

```
TextQuery::QueryResult::QueryResult(string s,
                                    shared_ptr<set<line_no>> p,
                                    shared_ptr<vector<string>> f):
                                    sought(s),lines(p),file(f) { }

//假设QueryResult有一个静态成员,则该成员的定义形式如下：
int TextQuery::QueryResult::static_mem = 1024;
```

嵌套类作用域中的名字查找

```
//返回类型必须指明QueryResult是一个嵌套类
TextQuery::QueryResult
TextQuery::query(const string &sought) const
{
    //如果没有找到sought，则返回set的指针
    static shared_ptr<set<line_no>> nodata(new set<line_no>);
    //使用find而非下标以避免像wm中添加单词
    auto loc = wm.find(sought);
    if(loc == wm.end())
        return QueryResult(sought,nodata,file);
    else
        return QueryResult(sought,loc->second,file);
}
```

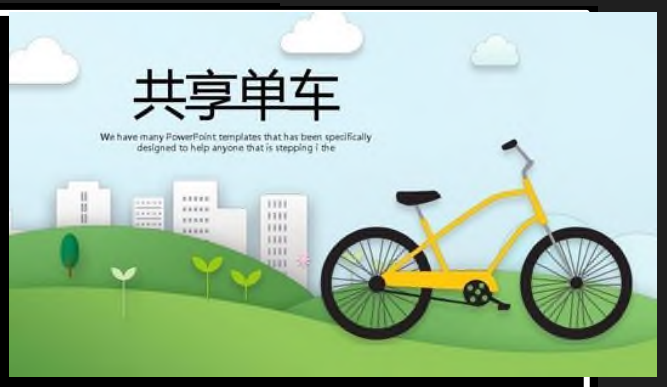
union:一种节省空间的类

联合可以有多个数据成员，但在任意时刻只有一个数据成员可以有值

```
union Token {
        //默认情况下成员是公有的
        char cval;
        int ival;
        double dval;
};

Token first_token = {'a'};    //初始化cval成员
Token last_token;             //未初始化的Token对象
Token *pt = new Token;        //指向一个未初始化的Token对象的指针

last_token.cavl = 'z';
pt->ival = 42;
```



union不能含有虚函数

未命名的union：编译器会自动地为其创建一个未命名的对象

```
union {
        char cval;
        int ival;
        double dval;
}; //定义一个未命名的对象，我们可以直接访问它的成员

//在匿名union的定义所在的作用域，union的成员都可以直接访问
cval = 'c'; //为未命名的对象赋值
ival = 42; //该对象状态改变
```

匿名union不能包含受保护的成员或私有成员，也不能定义成员函数

含有类类型成员的union


```
class Token {
public:
    // copy control needed because our class has a union with a string member
    Token(): tok(INT), ival(0) { }
    Token(const Token &t): tok(t.tok) { copyUnion(t); }
    Token &operator=(const Token&);
    // if the  union holds a string, we must destroy it;
    ~Token() { if (tok == STR) sval.~string(); }
    // assignment operators to set the differing members of the union
    Token &operator=(const std::string&);
    Token &operator=(char);
    Token &operator=(int);
    Token &operator=(double);
private:
    enum {INT, CHAR, DBL, STR} tok; // 判别式：追踪union的状态
    union {                 // anonymous union
        char        cval;
        int         ival;
        double      dval;
        std::string sval;
    }; // each Token object has an unnamed member of this unnamed union type

    // 检查判别式，然后酌情拷贝union成员
    void copyUnion(const Token&);
};
```

我们通常把含有类类型的union内嵌在另一个类当中。

管理判别式并销毁string

```
Token &Token::operator=(int i){
    if (tok == STR) sval.~string();     // if we have a string, free it
    ival = i;                           // assign to the appropriate member
    tok = INT;                          // update the discriminant
    return *this;
}
// char,double的版本基本和上面一致...
Token &Token::operator=(const std::string &s){
    if (tok == STR)             // if we already hold a string, just do an assignment
        sval = s;
    else
        new(&sval) std::string(s);  // otherwise construct a string
    tok = STR;                      // update the discriminant
    return *this;
}
```

管理需要拷贝控制的联合成员


```
void Token::copyUnion(const Token &t)
{
	switch (t.tok) {
		case Token::INT: ival = t.ival; break;
		case Token::CHAR: cval = t.cval; break;
		case Token::DBL: dval = t.dval; break;
		// to copy a string, construct it using placement new;
		case Token::STR: new(&sval) std::string(t.sval); break;
	}
}

Token &Token::operator=(const Token &t)
{
	// if this object holds a string and t doesn't, we have to free the old string
	if (tok == STR && t.tok != STR) sval.~string();
	if (tok == STR && t.tok == STR)
		sval = t.sval;  // no need to construct a new string
	else
		copyUnion(t);   // will construct a string if t.tok is STR
	tok = t.tok;
	return *this;
}
```

局部类

类可以定义在某个函数的内部

```
int a, val;
void foo(int val)
{
    static int si;
    enum Loc { a = 1024, b };
    //Bar是foo的局部类
    struct Bar {
        Loc locVal; //正确：使用一个局部类型名
        int barVar;

        void fooBar(Loc l = a) //正确：默认实参是Loc::a
        {
            barVal = val;       //错误：val是foo的局部变量
            barVal = ::val;     //正确：使用一个全局对象
            barVal = si;        //正确：使用一个静态局部对象
            locVal = b;         //正确：使用一个枚举成员
        }
    };
    //...
}
```

局部类的所有成员都必须完整定义在类的内部。

局部类不能使用函数作用域中的变量

可以在局部类的内部再嵌套一个类

```
void foo()
{
    class Bar {
    public:
        //...
        class Nested; //声明Nested类
    };
    //定义Nested类
    class Bar::Nested {
        //...
    };
}
```

局部类内的嵌套类也是一个局部类

固有的不可移植的特性

因机器而异的特性，移植需要重新编译，例如：算术类型

类可以将其（非静态）数据成员定义为位域（bit-field）

```
typedef unsigned int Bit;
class File {
    Bit mode: 2;              // mode has 2 bits
    Bit modified: 1;          // modified has 1 bit
    Bit prot_owner: 3;        // prot_owner has 3 bits
    Bit prot_group: 3;        // prot_group has 3 bits
    Bit prot_world: 3;        // prot_world has 3 bits
public:
        enum modes { READ = 01, WRITE = 02, EXECUTE = 03 };
        File &open(modes);
        void close();
        void write();
        bool isRead() const;
        void setWrite();
        void execute();
        bool isExecute() const;
};

void File::write(){   modified = 1;  // . . .}
void File::close(){if (modified) // . . . save contents; }

inline bool File::isRead() const { return mode & READ; }
inline void File::setWrite() { mode |= WRITE; }

File &File::open(File::modes m){
        mode |= READ;    // set the READ bit by default
        // other processing
        if (m & WRITE) // if opening READ and WRITE
                // processing to open the file in read/write mode
                cout << "myFile.mode WRITE is set" << endl;
        return *this;
}

int main(){
        File myFile;

        myFile.open(File::READ);
        if (myFile.isRead())
                cout << "reading" << endl;

        return 0;
}
```

位域(bit-field)在内存中的布局与机器是相关的

位域(bit-field)必须是整型或枚举类型

取地址符不能用于位域，因此指针无法指向类的位域

volatile限定符:告诉编译器不应对这样的对象进行优化 @阿西拜-南昌

```
volatile int display_register;          //该int值可能发生改变
volatile Task *curr_task;               //curr_task指向一个volatile对象
volatile int iax[max_size];             //iax的每个元素都是volatile
volatile Screen bitmapBuf;              //bitmapBufd的每个成员都是volatile

volatile int v;       //v是一个volatile int
int *volatile vip;    //vip是一个volatile指针，它指向int
volatile int * ivp;   //vip是一个指针，指向volatile int
//vivp是一个volatile指针，它指向一个volatile int
volatile int *volatile vivp;

int *ip = &v;         //错误：必须使用指向volatile的指针
ivp = &v;             //正确：ivp是一个指向volatile的指针
vivp = &v;            //正确：vivp是一个指向volatile的volatile指针
```

如果一个变量被volatile修饰，编译器将不会把它保存到寄存器中，而是每一次都去访问内存中实际保存该变量的位置上

合成的拷贝对volatile无效

```
class Foo {
public:
    Foo(const volatile Foo&); //从一个volatile对象进行拷贝
    //将一个volatile对象赋值给一个非volatile对象
    Foo& operator=(volatile const Foo&);
    //将一个volatile对象赋值给一个volatile对象
    Foo& operator=(volatile const Foo&) volatile;
    //Foo类的剩余部分...
}
```

链接指示指出任意非C++函数所用的语言


```
//可能出现在C++头文件<cstring>中的链接指示
//单语句链接指示
extern "C" size_t strlen(const char *);
//复合语句链接指示
extern "C" {
      int strcmp(const char*, const char*);
      char *strcat(char*,const char*);
}
```

链接指示与头文件

```
//复合语句链接指示
extern "C" {
#include <string.h> //操作C风格字符串的C函数
//头文件中的所有普通函数声明都被认为是由链接指示的语言编写的
//链接指示可以嵌套
}
```

指向extern “C”函数的指针

```
//pf指向一个C函数，该函数接受一个int返回void
exern "C" void (*pf)(int);

void (*pf1)(int);              //指向一个C++函数
extern "C" void (*pf2)(int); //指向一个C函数
pf1 = pf2;                     //错误：pf1和pf2的类型不同
```

编写函数所用的语言是函数类型的一部分

链接指示对整个声明都有效

```
//f1是一个C函数，它的形参是一个指向C函数的指针
extern "C" void f1(void(*)(int));

//FC是一个指向C函数的指针
extern "C" typedef void FC(int);
//f2是一个C++函数，该函数的形参是执行C函数的指针
void f2(FC*);
```

导出C++函数到其他语言：通过链接指示对函数进行定义

```
//calc函数可以被C程序调用
extern "C" double calc(double dparm) {/*...*/}
//编译器将为该函数生成适合于指定语言的代码
```

重载函数与链接指示

```
//错误：两个extern"C"函数的名字相同
extern "C" void print(const char*);
extern "C" void print(int);

class SamllInt { /*...*/ };
class BigNum { /*...*/ };
//C函数可以在C或C++程序中调用
//C++函数重载了该函数，可以在C++程序中调用
extern "C" double calc(double);
extern SmallInt calc(const SmallInt&);
extern BigNum calc(const BigNum&);
```

https://ke.qq.com/course/3854997?flowToken=1039424