目录

[一、 const关键字 1](#_Toc18741)

[1. 概述： 1](#_Toc18344)

[2. 比define预编译的优点： 2](#_Toc31545)

[3. 具体修饰对象 2](#_Toc18941)

[二、static关键字 5](#_Toc32439)

[1.面向过程具体修饰对象 5](#_Toc9512)

[2. 面向对象具体修饰对象 6](#_Toc11498)

[三、extern关键字 7](#_Toc2446)

[1.概述： 7](#_Toc10326)

[2.extern "C"  7](#_Toc15506)

[四、 volatile关键字 7](#_Toc16915)

[1. 编译器优化介绍 7](#_Toc28454)

[2. volatile关键字概述 8](#_Toc25241)

[3. 事例： 8](#_Toc15847)

[4. 应用场景 9](#_Toc23032)

[5. 几个问题 11](#_Toc20230)

[6. volatile关键字本质 11](#_Toc23182)

[7. 注意： 12](#_Toc25739)

[五、 宏 12](#_Toc15044)

[1.含参数的宏与函数的优缺点 12](#_Toc11710)

[六、 内联函数 12](#_Toc15700)

[1. 引入 12](#_Toc14513)

[2. 事例 13](#_Toc23850)

[3. 编程风格 13](#_Toc11586)

[4. 优缺点 14](#_Toc16605)

[5. 不宜使用场景 14](#_Toc1299)

[6. 注意 14](#_Toc23373)

# const关键字

## 1. 概述：

用来定义常量，如果一个变量被const修饰，那么它的值就不能再被改变

## 比define预编译的优点：

### .类型检查

预编译指令只是对值进行简单的替换，不能进行类型检查;

### .健壮性

可以保护被修饰的东西，防止意外修改，增强程序的健壮性;

### .效率

编译器通常不为普通const常量分配存储空间，而是将它们保存在符号表中，这使得它成为一个编译期间的常量，没有了存储与读内存的操作，使得它的效率也很高。

## 具体修饰对象

### .修饰局部变量

1. .一般

微信图片_20210616115429

两种写法是一样的，都是表示变量n的值不能被改变了，需要注意的是，用const修饰变量时，一定要给变脸初始化，否则之后就不能再进行赋值了。

1. .修饰常量静态字符串

2

如果没有const的修饰，我们可能会在后面有意无意的写str[4]=’x’这样的语句，这样会导致对只读内存区域的赋值，然后程序会立刻异常终止。有了const，这个错误就能在程序被编译的时候就立即检查出来，这就是const的好处。让逻辑错误在编译期被发现。

### .常量指针与指针常量

1. .常量指针

<1>.概述：

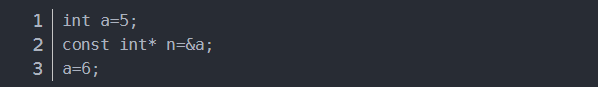
常量指针是指针指向的内容是常量；

<2>.两种定义方式：

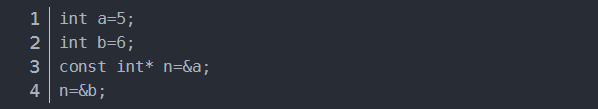
33

<3>.注意：

1. . 常量指针说的是不能通过这个指针改变变量的值，但是还是可以通过其他的引用来改变变量的值的。



1. .常量指针指向的值不能改变，但是这并不是意味着指针本身不能改变，常量指针可以指向其他的地址。



1. .指针常量

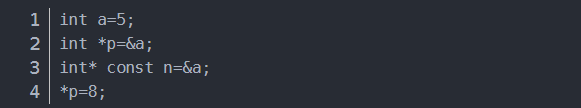
<1>.概述:

指针常量是指指针本身是个常量，不能在指向其他的地址

666

<2>.注意：

指针常量指向的地址不能改变，但是地址中保存的数值是可以改变的，可以通过其他指向改地址的指针来修改



1. .指向常量的常指针

<1>.概述：

是以上两种的结合，指针指向的位置不能改变并且也不能通过这个指针改变变量的值，但是依然可以通过其他的普通指针改变变量的值。

888

### .修饰函数的参数

根据常量指针与指针常量，const修饰函数的参数也是分为三种情况：

1. .防止修改指针指向的内容；

999

1. .防止修改指针指向的地址；

0000

1. .以上两种的结合。

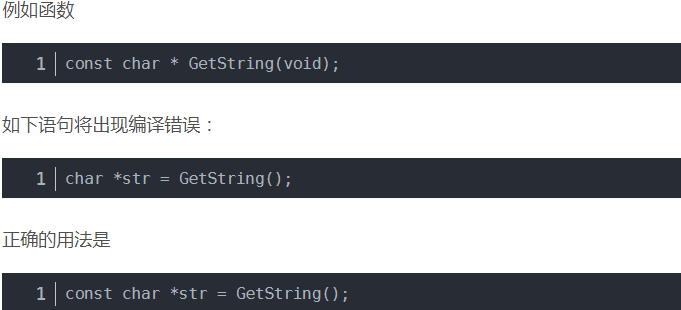
### .修饰函数的返回值

1. .概述：

如果给以“指针传递”方式的函数返回值加 const 修饰，那么函数返回值（即指针）的内容不能被修改，该返回值只能被赋给加const 修饰的同类型指针。

1. .注意：

该返回值只能被赋给加const 修饰的同类型指针；



### .修饰全局变量

1).概述：

全局变量的作用域是整个文件，我们应该尽量避免使用全局变量，因为一旦有一个函数改变了全局变量的值，它也会影响到其他引用这个变量的函数，导致除了bug后很难发现，如果一定要用全局变量，我们应该尽量的使用const修饰符进行修饰，这样防止不必要的人为修改，使用的方法与局部变量是相同的。

# 二、static关键字

## 1.面向过程具体修饰对象

### (1).局部变量

1).相对于普通局部变量差异

<1>.不同：

1. . 普通局部变量存储于进程栈空间，使用完毕会立即释放。static修饰的变量在全局数据区分配内存空间；
2. . 编译器一般不对普通局部变量进行初始化，也就是说它的值在初始时是不确定的，除非对其显式赋值。而static修饰的变量编译器自动对其初始化；

<2>.同：

1>.其作用域为局部作用域，当定义它的函数结束时，其作用域随之结束。

2).优点

静态局部变量的效果跟全局变量有一拼，但是位于函数体内部，就极有利于程序的模块化了。

### .全局变量

1. .概述：

全局变量定义在函数体外部，在全局数据区分配存储空间，且编译器会自动对其初始化。

1. .和普通全局变量异同：

普通全局变量： 对整个工程可见，其他文件可以使用extern外部声明后直接使用。也就是说其他文件不能再定义一个与其相同名字的变量了（否则编译器会认为它们是同一个变量）。

静态全局变量：仅对当前文件可见，其他文件不可访问，其他文件可以定义与其同名的变量，两者互不影响。

1. .优点:

在定义不需要与其他文件共享的全局变量时，加上static关键字能够有效地降低程序模块之间的耦合，避免不同文件同名变量的冲突，且不会误使用。

### (3).函数

1).静态函数定义：

函数的使用方式与全局变量类似，在函数的返回类型前加上static，就是静态函数。

2).静态函数特性:

<1>.静态函数只能在声明它的文件中可见，其他文件不能引用该函数;

<2>.不同的文件可以使用相同名字的静态函数，互不影响

3).和非静态函数差异：

非静态函数可以在另一个文件中直接引用，甚至不必使用extern声明,而静态函数外部文件不可用。

## 面向对象具体修饰对象

### .静态数据成员

1. .定义:

在类内数据成员的声明前加上static关键字，该数据成员就是类内的静态数据成员。

1. .特点:

<1>.静态数据成员存储在全局数据区，静态数据成员在定义时分配存储空间，所以不能在类声明中定义;

<2>.静态数据成员是类的成员，无论定义了多少个类的对象，静态数据成员的拷贝只有一个，且对该类的所有对象可见。也就是说任一对象都可以对静态数据成员进行操作。而对于非静态数据成员，每个对象都有自己的一份拷贝;

<3>.由于上面的原因，静态数据成员不属于任何对象，在没有类的实例时其作用域就可见，在没有任何对象时，就可以进行操作；

<4>.和普通数据成员一样，静态数据成员也遵从public, protected, private访问规则；

1. . 初始化格式：

<数据类型><类名>::<静态数据成员名>=<值>

1. .两种访问方式:

<1>.<类对象名>.<静态数据成员名>

<2>.<类类型名>::<静态数据成员名>

1. .相对于全局变量优点:

<1>.静态数据成员没有进入程序的全局名字空间，因此不存在与程序中其它全局名字冲突的可能性;

<2>.可以实现信息隐藏。静态数据成员可以是private成员，而全局变量不能。

### .静态成员函数

1. .概述：

与静态数据成员类似，静态成员函数属于整个类，而不是某一个对象。

1. .特性:

<1>.静态成员函数: 没有this指针，它无法访问属于类对象的非静态数据成员，也无法访问非静态成员函数，它只能调用其余的静态成员函数;

非静态成员函数: 可以任意地访问静态成员函数和静态数据成员

<2>.出现在类体外的函数定义不能指定关键字static.。

# 三、extern关键字

## 1.概述：

extern修饰全局变量或函数，以标示变量或者函数的定义在别的文件中，提示编译器遇到此变量和函数时在其他模块中寻找其定义。

 与之对应的关键字是static，static表明变量或函数是静态的，只能在本编译单元中使用。

## 2.extern "C"

### (1). 概述：

修饰的变量或函数是按C语言方式进行编译和链接的；

### (2). 作用：

因为C++支持多态性，函数可以重载，而C不支持。extern "C"策略，是为了使C++可以调用C语言编写的代码，使得C++和C之间的接口具有互通性。

# volatile关键字

## 编译器优化介绍

### .硬件优化

由于内存访问速度远不及CPU处理速度，为提高机器整体性能，在硬件上引入硬件高速缓存Cache，加速对内存的访问。另外在现代CPU中指令的执行并不一定严格按照顺序执行，没有相关性的指令可以乱序执行，以充分利用CPU的指令流水线，提高执行速度。

### .软件优化

1. .在编写代码时由程序员优化；
2. .编译器进行优化

<1>.将内存变量缓存到寄存器;

<2>.调整指令顺序充分利用CPU指令流水线,常见的是重新排序读写指令。

1. .编译器优化问题解决:

对常规内存进行优化的时候，这些优化是透明的，而且效率很好。由编译器优化或者硬件重新排序引起的问题的解决办法是在从硬件（或者其他处理器）的角度看必须以特定顺序执行的操作之间设置内存屏障（memory barrier），linux 提供了一个宏解决编译器的执行顺序问题。

void Barrier(void)

这个函数通知编译器插入一个内存屏障，但对硬件无效，编译后的代码会把当前CPU寄存器中的所有修改过的数值存入内存，需要这些数据的时候再重新从内存中读出。

1. . volatile关键字与优化关系:

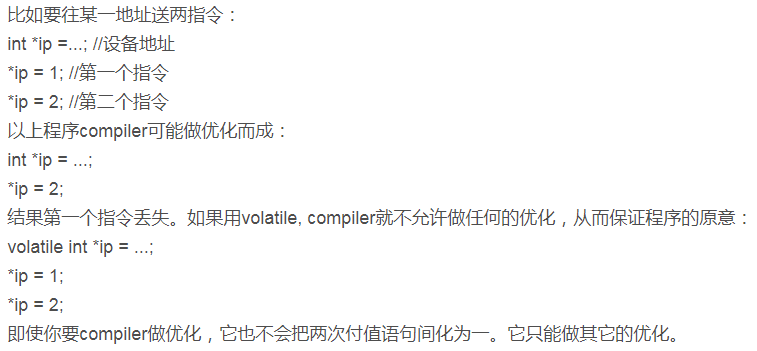
volatile总是与优化有关，编译器有一种技术叫做数据流分析，分析程序中的变量在哪里赋值、在哪里使用、在哪里失效，分析结果可以用于常量合并，常量传播等优化，进一步可以消除一些代码。但有时这些优化不是程序所需要的，这时可以用volatile关键字禁止做这些优化。

## volatile关键字概述

volatile的本意是“易变的” 因为访问寄存器要比访问内存单元快的多,所以编译器一般都会作减少存取内存的优化，但有可能会读脏数据。当要求使用volatile声明变量值的时候，系统总是重新从它所在的内存读取数据，即使它前面的指令刚刚从该处读取过数据。精确地说就是，遇到这个关键字声明的变量，编译器对访问该变量的代码就不再进行优化，从而可以提供对特殊地址的稳定访问；如果不使用valatile，则编译器将对所声明的语句进行优化。（简洁的说就是：volatile关键词影响编译器编译的结果，用volatile声明的变量表示该变量随时可能发生变化，与该变量有关的运算，不要进行编译优化，以免出错）。

## 事例：

1. .告诉compiler不能做任何优化

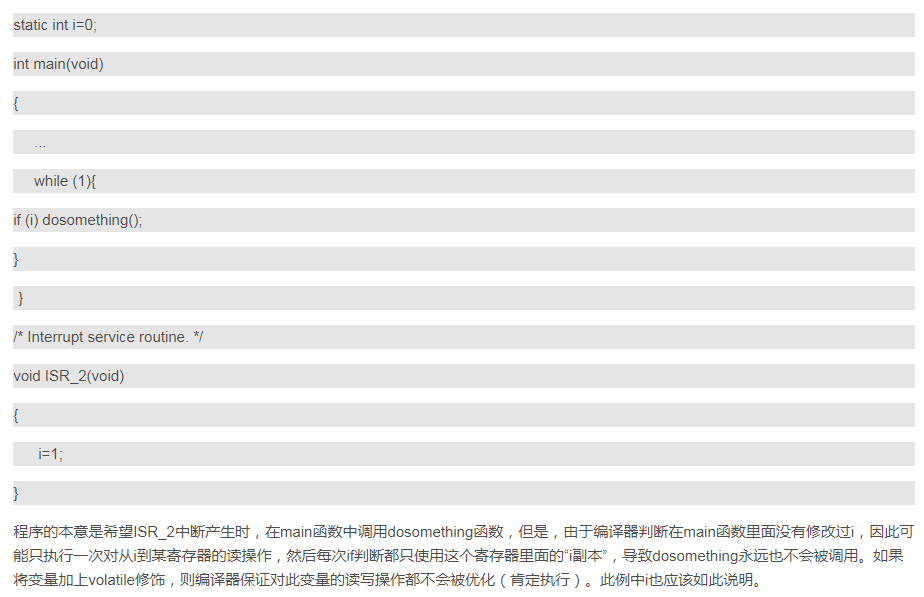


1. .用volatile定义的变量会在程序外被改变,每次都必须从内存中读取，而不能重复使用放在cache或寄存器中的备份。

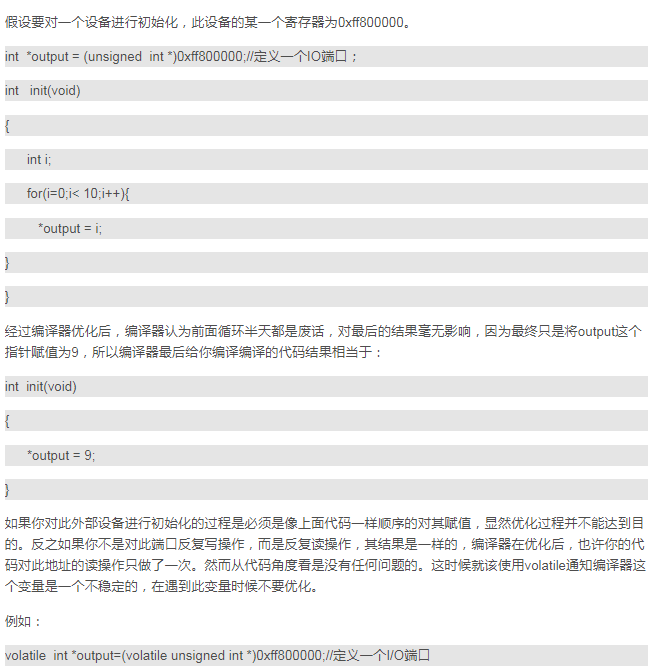


## 应用场景

1. .一个中断服务子程序中会访问到的非自动变量(Non-automatic variables);



1. .多任务环境下各任务间共享的标志应该加volatile;
2. .存储器映射的硬件寄存器通常也要加voliate，因为每次对它的读写都可能有不同意义。并行设备的硬件寄存器（如：状态寄存器）。



## 几个问题

1. .一个参数既可以是const还可以是volatile吗？

可以的，例如只读的状态寄存器。它是volatile因为它可能被意想不到地改变。它是const因为程序不应该试图去修改它。

1. .一个指针可以是volatile 吗？

可以，当一个中断服务子程序修该一个指向一个buffer的指针时。

## volatile关键字本质

1. .编译器的优化

1). 在本次线程内, 当读取一个变量时，为提高存取速度，编译器优化时有时会先把变量读取到一个寄存器中；以后，再取变量值时，就直接从寄存器中取值；当变量值在本线程里改变时，会同时把变量的新值copy到该寄存器中，以便保持一致;

2). 当变量在因别的线程等而改变了值，该寄存器的值不会相应改变，从而造成应用程序读取的值和实际的变量值不一致;

3). 当该寄存器在因别的线程等而改变了值，原变量的值不会改变，从而造成应用程序读取的值和实际的变量值不一致。

1. .volatile应该解释为“直接存取原始内存地址”比较合适，“易变的”这种解释简直有点误导人。

## 注意：

频繁地使用volatile很可能会增加代码尺寸和降低性能,因此要合理的使用volatile

# 宏

## 1.含参数的宏与函数的优缺点

|  |  |  |
| --- | --- | --- |
| 方面 | 带宏参 | 函数 |
| 处理时间 | 编译时 | 程序运行时 |
| 参数类型 | 没有参数类型问题 | 定义实参、形参类型 |
| 处理过程 | 不分配内存 | 分配内存 |
| 程序长度 | 变长 | 不变 |
| 运行速度 | 不占运行时间 | 调用和返回占用时间 |

# 内联函数

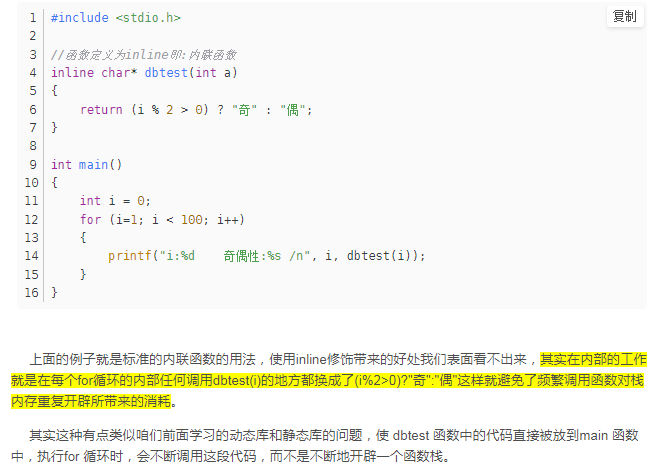
## 引入

在C语言中，如果一些函数被频繁调用，不断地有函数入栈，即函数栈，会造成栈空间或栈内存的大量消耗。

为了解决这个问题，特别的引入了inline修饰符，表示为内联函数。

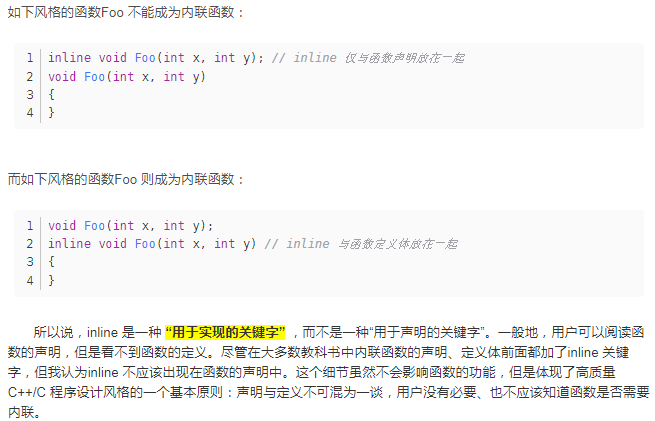
 栈空间就是指放置程式的局部数据也就是函数内数据的内存空间，在系统下，栈空间是有限的，假如频繁大量的使用就会造成因栈空间不足所造成的程式出错的问题，函数的死循环递归调用的最终结果就是导致栈内存空间枯竭。

## 事例



## 编程风格

1. .关键字inline 必须与函数定义体放在一起才能使函数成为内联，仅将inline 放在函数声明前面不起任何作用。



1. .inline的使用是有所限制的

 inline只适合涵数体内代码简单的函数数使用，不能包含复杂的结构控制语句例如while、switch，并且内联函数本身不能是直接递归函数(自己内部还调用自己的函数)。

## 优缺点

1. .内联是以代码膨胀（复制）为代价，仅仅省去了函数调用的开销，从而提高函数的执行效率。如果执行函数体内代码的时间，相比于函数调用的开销较大，那么效率的收  
   获会很少。
2. .每一处内联函数的调用都要复制代码，将使程序的总代码量增大，消耗更多的内存空间。

## 不宜使用场景

1. .如果函数体内的代码比较长，使用内联将导致内存消耗代价较高；
2. .如果函数体内出现循环，那么执行函数体内代码的时间要比函数调用的开销大。

## 注意

 因此,将内联函数放在头文件里实现是合适的,省却你为每个文件实现一次的麻烦.而所以声明跟定义要一致,其实是指,如果在每个文件里都实现一次该内联函数的话,那么,最好保证每个定义都是一样的,否则,将会引起未定义的行为,即是说,如果不是每个文件里的定义都一样,那么,编译器展开的是哪一个,那要看具体的编译器而定.所以,最好将内联函数定义放在头文件中。