目录

[const 2](#_Toc99186285)

[const和constexpr 3](#_Toc99186286)

[const和define 3](#_Toc99186287)

[typedef和define 4](#_Toc99186288)

[decltype 4](#_Toc99186289)

[=default和=delete 4](#_Toc99186290)

[inline 5](#_Toc99186291)

[inline和define 5](#_Toc99186292)

[static 5](#_Toc99186293)

[struct和class 7](#_Toc99186294)

[friend 7](#_Toc99186295)

[explicit 7](#_Toc99186296)

[bind 7](#_Toc99186297)

[final和override 7](#_Toc99186298)

[new/delete 8](#_Toc99186299)

[new有几种类型 8](#_Toc99186300)

[new/delete和malloc/free 9](#_Toc99186301)

[volatile 9](#_Toc99186302)

[extern 10](#_Toc99186303)

## const

1、修饰常量时：

表明常量本身不可改变

2、修饰指针时：

主要看const在\*的前后，在前则指针指向的内容为常量（底层const），在后则指针本身为常量（顶层const）；

3、const修饰类对象时：

const修饰类对象时，其对象中的任何成员都不能被修改。const修饰的对象，该对象的任何非const成员函数都不能调用该对象，因为任何非const成员函数都会有修改成员变量的可能。

4、const修饰成员变量：

const修饰的成员变量不能被修改，同时只能在初始化列表中被初始化，因为常量只能被初始化，不能被赋值；赋值是使用新值覆盖旧值，构造函数是先为其开辟空间然后为其赋值，不是初始化；而初始化列表开辟空间和初始化是同时完成的，直接给与一个值，所以const成员变量一定要在初始化列表中完成。

5、const修饰类的成员函数

const成员函数表示该成员函数不能修改类对象中的任何非const成员变量。一般const写在函数的后面，形如：void func() const;如果某个成员函数不会修改成员变量，那么最好将其声明为const，因为const成员函数不会对数据进行修改，如果修改，编译器将会报错；

对于const类对象，只能调用类中的const成员函数，所以const修饰成员函数的作用主要就是限制对const对象的使用。

6、const在函数声明中的使用：

在函数声明中，const可以修饰函数的返回值，也可以修饰具体某一个形参；修饰形参时，用相应的变量初始化const常量，在函数体内，按照const所修饰的部分进行常量化；修饰函数返回值时，一般情况下，const修饰返回值多用于操作符的重载。通常不建议用const修饰函数的引用类型返回值；

## const和constexpr

1、修饰对象来说：

* const修饰一个对象表示它是常量。这暗示对象一经初始化就不会再变动了，并且允许编译器使用这个特点优化程序。这也防止程序员修改了本不应该修改的对象。
* constexpr是修饰一个常量表达式。但注意constexpr不是修饰常量表达式的唯一途径。

2、修饰函数时：

* const只能用于非静态成员的函数而不是所有函数。它保证成员函数不修改任何非静态数据。
* constexpr可以用于含参和无参函数。constexpr函数适用于常量表达式，只有在下面的情况下编译器才会接受constexpr函数：

（1）函数体必须足够简单，除了typedef和静态元素，只允许有return语句。如构造函数只能有初始化列表，typedef和静态元素(实际上在C++14标准中已经允许定义语句存在于constexpr函数体内了)

（2）参数和返回值必须是字面值类型

## const和define

1、处理阶段不同：

* define宏定义是编译时概念，在预处理阶段展开
* const常量是运行时概念，表示只读

2、类型和安全检查不同

* define宏定义本质是字符替换，没有数据类型，没有类型安全检查
* const常量有明确的类型，在编译阶段会进行类型检查；

3、存储方式不同

* define宏定义直接替换，没有内存分配，存储在程序的代码段中；
* const修饰全局常量时，存放在常量区中，修饰局部常量时，通常不会为其分配空间，只是将它们保存在符号表内，使他们成为一个编译期间的一个常量，在编译期间任何出现该变量的地方都会被替换成常量表达式的值，没有读取内存的操作，效率很高；

4、作用域不同

* define定义的宏是全局的
* const常量作用域取决于位置

5、能否取消

* 宏定义可以通过#undef撤销
* const在作用域内永久有效

## typedef和define

* typedef是用来定义一种类型的新别名的，它不同于宏（#define），不是简单的字符串替换。它的新名字具有一定的封装性，所以新命名的标识符具有更易定义变量的功能，它是语言编译过程的一部分，有数据类型检查，但它并不实际分配内存空间。
* #define只是简单的字符串替换（原地扩展），它本身并不在编译过程中进行，而是在预处理阶段就已经完成了。因此，它不会做正确性检查，不管含义是否正确它照样会带入，只有在编译时才会发现可能的错误并报错

## decltype

作用：选择并返回操作数的数据类型。编译器分析表达式的类型，却不计算表达式的值

细节：如果decltype使用的是一个不加括号的变量，则得到的结果是该变量的类型；如果给变量加了一层或多层括号，编译器会把它当成是一个表达式

切记：decltype((variable))的结果永远是引用，而decltype(variable)结果只有当variable本身是一个引用时才是引用

## =default和=delete

* =default用于生成默认构造/析构函数等
* =delete禁止编译器生成默认的相关函数

## inline

使用：内联函数用inline修饰，关键字在函数声明和定义的时候都要加上

理解：内联函数是直接复制“镶嵌”到主函数中去的，就是将内联函数的代码直接放在内联函数的位置上，这与一般函数不同，主函数在调用一般函数的时候，是指令跳转到被调用函数的入口地址，执行完被调用函数后，指令再跳转到主函数上继续执行后面的代码；而由于内联函数是将函数的代码直接放在了函数的位置上，所以没有指令跳转，指令按顺序执行。一般函数的代码只有一份，放在内存中的某个位置上，当程序调用它是，指令就跳转过来；当下一次程序调用它是，指令又跳转过来；而内联函数是程序中调用几次内联函数，内联函数的代码就会复制几份放在对应的位置上

好处：避免了指令的来回跳转，加快程序执行速度

坏处：代码被多次复制，增加了代码量，占用更多的内存空间

## inline和define

* define：定义预编译时处理的宏，只是简单的字符串替换，没有类型检查。
* inline：

1、inline关键字用来定义一个类的内联函数，引入它的主要原因是用它替代C中表达式形式的宏定义，编译阶段完成。

2、内联函数要做类型安全检查，inline是指嵌入代码，在调用函数的地方不是跳转，而是把代码直接写到那里去，对于短小的函数来说，inline函数可以得到一定效率的提升，和c时代的宏函数相比，inline函数更加安全可靠，不过以增加空间的消耗为代价。

## static

1、静态成员变量（面向对象）

* 静态成员变量为类所有对象共有，而普通成员变量，每个类的对象都有自己的一份拷贝，只进行一次内存分配，处于全局数据区，不属于特定的类的对象，不占用对象的内存
* static成员变量的内存空间既不是在声明类时分配，也不是在创建对象时分配，而是在初始化时分配，静态成员变量必须初始化，且只能在类体外进行
* static成员变量和普通static变量一样，编译时在静态数据区分配内存，到程序结束时才释放，不随对象销毁而释放
* 静态成员变量与一般变量初始化不同，初始化可以不加static，但必须有数据类型，格式为＜数据类型＞＜类名＞::＜静态数据成员名＞=＜值＞
* 类的静态成员变量访问形式：

＜类对象名＞.＜静态数据成员名＞

＜类类型名＞::＜静态数据成员名＞ 即不需要对象就可访问

* 静态成员变量与普通成员一样遵从public/protected/private访问规则
* sizeof运算符不会计算静态成员变量

2、静态成员函数（面向对象）

* 静态成员之间可以互相访问，即静态成员函数可以访问静态成员变量和静态成员函数，不能访问非静态成员，而非静态成员函数可以任意访问静态成员
* 静态成员函数属于类本身，不像普通成员函数一样为类的对象服务，因此不存在this指针，从开减少了开销，速度更快
* 调用静态成员函数方式：

成员访问符：（.）和（->），即通过类的对象或者指向对象的指针调用

直接通过类调用：＜类名＞::＜静态成员函数名＞（＜参数表＞）

3、静态全部变量（面向过程）

* 内存分配在全局数据区
* 未经初始化的静态全部变量会被默认初始化为0
* 静态全部变量在申明它的整个文件都是可见的，在文件之外是不可见的

4、静态局部变量（面向过程）

* 静态局部变量在全局数据区分配内存
* 静态局部变量在程序执行到该对象的声明处时被首次初始化，即以后的函数调用不再进行初始化，若没有显式初始化，被程序自动初始化为0
* 静态局部变量始终驻留在全局数据区，直到程序运行结束。但其作用域为局部作用域，当定义它的函数或语句块结束时，其作用域随之结束

5、静态函数（面向过程）

* 静态函数不能被其他文件所用
* 其他文件可以定义相同名字的函数，不会发生冲突

## struct和class

* struct成员的默认访问权限是public
* class成员的默认访问权限是private

## friend

友元申明符，其他类或者函数作为该类的友元，即可访问该类的非共有成员，当把一个成员函数声明为友元时，必须显式指出该成员函数属于哪个类

注意：

1、友元关系不能被继承。

2、友元关系是单向的，不具有交换性。若类B是类A的友元，类A不一定是类B的友元，要看在类中是否有相应的声明。

3、友元关系不具有传递性。若类B是类A的友元，类C是B的友元，类C不一定是类A的友元，同样要看类中是否有相应的申明

## explicit

用于抑制构造函数的隐式转换，即调用过程中不允许产生隐式类型转换，必须直接初始化，只对一个实参的构造函数有效，只需要在类内声明时使用，类外部不应重复

## bind

定义于头文件functional中，可以将bind函数看作一个通用的函数适配器，它接受一个可调用对象，生成一个新的可调用对象来“适应“原对象的参数列表

形式：auto newCallable = bind (callable, arg\_list);

newCallable本身是一个可调用对象，arg\_list是一个逗号分割的参数列表，对应给callable的参数。当我们调用newCallable时，newCallable会调用callable，并传递给它arg\_list的参数

arg\_list中的参数可能包含形如\_n的名字，其中n为整数，表示占位符，比如\_1表示为newCallable的第一个参数。使用命名空间为using std::placeholders::\_1

## final和override

override：用于派生类对父类中的虚函数进行重写

final：当不希望某个类被继承，或不希望虚函数被重写，可以在类名或虚函数后添加final

## new/delete

new/new[]和delete/delete[]是C++用来实现动态内存管理的操作符

* new/new[] 操作符是用来申请空间的；
* delete/delete[]操作数是用来释放动态申请出来的空间；

底层实现：

* new在底层调用operator new全局函数来申请空间，而operator new 实际是通过malloc来申请空间的
* delete在底层通过operator delete全局函数来释放空间，而operator delete实际是通过free来释放空间的；

实现过程：

* new

1、调用operator new函数申请空间

2、在申请的空间上执行构造函数，完成对象的构造

* delete

1、在空间上执行析构函数，完成对象中资源的清理工作

2、调用operator delete函数释放对象的空间

* new[N]

1、调用operator new[]函数，在operator new[]中实际调用operator new函数完成N个对象空间的申请

2、在申请的空间上执行N次构造函数

* delete[N]

1、在释放的对象空间上执行N次析构函数，完成N个对象中资源的清理

2、调用operator delete[]释放空间，实际在operator delete[]中调用operator delete来释放空间

## new有几种类型

plain new：普通的new操作符

nothrow new：在空间分配失败的情况下不抛出异常，而是返回NULL

placement new：允许在一块已经分配成功的内存上重新构造对象，不用担心内存分配失败

注意：

1、palcement new的主要用途就是反复使用一块较大的动态分配的内存来构造不同类型的对象或者他们的数组

2、placement new构造起来的对象数组，要显式的调用他们的析构函数来销毁（析构函数并不释放对象的内存），千万不要使用delete，这是因为placement new构造起来的对象或数组大小并不一定等于原来分配的内存大小，使用delete会造成内存泄漏或者之后释放内存时出现运行时错误

## new/delete和malloc/free

相同点：都可用于申请动态内存和释放内存

不同点：

1、new/delete是C++关键字，需要编译器支持。malloc/free是库函数，需要头文件支持；

2、使用new操作符申请内存分配时无须指定内存块的大小，编译器会根据类型信息自行计算。而malloc则需要显式地指出所需内存的尺寸。

3、new操作符内存分配成功时，返回的是对象类型的指针，类型严格与对象匹配，无须进行类型转换，故new是符合类型安全性的操作符。而malloc内存分配成功则是返回void \* ，需要通过强制类型转换将void\*指针转换成我们需要的类型。

4、new内存分配失败时，会抛出bac\_alloc异常。malloc分配内存失败时返回NULL。

5、new会先调用operator new函数，申请足够的内存（通常底层使用malloc实现）。然后调用类型的构造函数，初始化成员变量，最后返回自定义类型指针。delete先调用析构函数，然后调用operator delete函数释放内存（通常底层使用free实现）。malloc/free是库函数，只能动态的申请和释放内存，无法强制要求其做自定义类型对象构造和析构工作。

## volatile

定义为volatile的变量意味着告诉编译器这个变量可能会被意想不到地改变，即在程序运行过程中一直会变，希望这个值被正确的处理，每次从内存中去读这个值，而不是因编译器优化从缓存的地方读取，比如读取缓存在寄存器中的数值，从而保证volatile变量被正确的读取。该关键字的作用是防止优化编译器把变量从内存装入CPU寄存器中。

单线程环境中，一个函数体内部，如果在两次读取变量的值之间的语句没有对变量的值进行修改，那么编译器就会设法对可执行代码进行优化。由于访问寄存器的速度要快过RAM（从RAM中读取变量的值到寄存器），以后只要变量的值没有改变，就一直从寄存器中读取变量的值，而不对RAM进行访问。

而在多线程环境中，虽然在一个函数体内部，在两次读取变量之间没有对变量的值进行修改，但是该变量仍然有可能被其他的程序（如中断程序、另外的线程等）所修改。如果这时还是从寄存器而不是从RAM中读取，就会出现被修改了的变量值不能被正确读取的问题。

## mutable

mutable的中文意思是“可变的，易变的”，跟constant（既C++中的const）是反义词。在C++中，mutable也是为了突破const的限制而设置的。被mutable修饰的变量，将永远处于可变的状态，即使在一个const函数中。我们知道，如果类的成员函数不会改变对象的状态，那么这个成员函数一般会声明成const的。但是，有些时候，我们需要在const函数里面修改一些跟类状态无关的数据成员，那么这个函数就应该被mutable来修饰，并且放在函数后后面关键字位置。

## extern

* C语言中，extern用在变量和函数声明前，用于说明“此变量/函数是在别处定义的，要在此处引用”
* C++语言中，使用链接指示指出任意非C++函数所用的语言，比如在C++中调用C的库函数，就需要声明。这是给链接器用的，告诉链接器在链接的时候使用C函数规范来链接。主要原因是C++和C程序在编译完成后目标代码中命名规则不同，用此来解决名字匹配的问题。

**为什么要用**：调用文件中函数和变量，只需用include含进来即可，用extern可以加速程序的编译过程，节省时间

**怎么用**：

extern的作用域取决于其声明的位置，不能出现在类定义或者函数定义的内部

单个的：extern “C” size\_t strlen(const char \*);

复合的：extern “C” {

int strcmp(const char \*, const char \*);

char \*strcat (char \*, const char \*);

}