C++中的const可用于修饰变量、函数，且在不同的地方有着不同的含义，现总结如下。

const的语义

C++中的const的目的是通过编译器来保证对象的常量性，强制编译器将所有可能违背const对象的常量性的操作都视为error。

对象的常量性可以分为两种：物理常量性（即每个bit都不可改变）和逻辑常量性（即对象的表现保持不变）。C++中采用的是物理常量性，例如下面的例子：

```
struct A {
    int *ptr;
};
int k = 5, r = 6;
const A a = {&k};
a.ptr = &r; // !error
*a.ptr = 7; // no error
```

a是const对象，则对a的任何成员进行赋值都会被视为error，但如果不改动ptr，而是改动ptr指向的对象，编译器就不会报错。这实际上违背了逻辑常量性，因为A的表现已经改变了！

逻辑常量性的另一个特点是，const对象中可以有某些用户不可见的域，改变它们不会违背逻辑常量性。Effective C++中的例子是：

```
class CTextBlock {
public:
    ...
    std::size_t length() const;
private:
    char *pText;
    std::size_t textLength;            // last calculated length of textb
lock
    bool lengthIsValid;                // whether length is currently val
id
};
```

CTextBlock对象每次调用length方法后，都会将当前的长度缓存到textLength成员中，而lengthIsValid对象则表示缓存的有效性。这个场景中textLength和lengthIsValid如果改变了，其实是不违背CTextBlock对象的逻辑常量性的，但

因为改变了对象中的某些bit，就会被编译器阻止。C++中为了解决此问题，增加了mutable关键字。
本部分总结：C++中const的语义是保证物理常量性，但通过mutable关键字可以支持一部分的逻辑常量性。

const修饰变量

如上节所述，用const修饰变量的语义是要求编译器去阻止所有对该变量的赋值行为。因此，必须在const变量初始化时就提供给它初值：

```
const int i;
i = 5; // !error
const int j = 10; // ok
```

这个初值可以是编译时即确定的值，也可以是运行期才确定的值。如果给整数类型的const变量一个编译时初值，那么可以用这个变量作为声明数组时的长度：

```

const int COMPILE_CONST = 10;
const int RunTimeConst = cin.get();
int a1[COMPLIE_CONST]; // ok in C++ and error in C
int a2[RunTimeConst]; // !error in C++
```

因为C++编译器可以将数组长度中出现的编译时常量直接替换为其字面值，相当于自动的宏替换。（gcc验证发现，只有数组长度那里直接做了替换，而其它用COMPILE_CONST赋值的地方并没有进行替换。）
文件域的const变量默认是文件内可见的，如果需要在b.cpp中使用a.cpp中的const变量M，需要在M的初始化处增加extern：

```
//a.cpp
extern const int M = 20;

//b.cpp
extern const int M;
```

一般认为将变量的定义放在.h文件中会导致所有include该.h文件的.cpp文件都有此变量的定义，在链接时会造成冲突。但将const变量的定义放在.h文件中是可以的，编译器会将这个变量放入每个.cpp文件的匿名namespace中，因而属

于是不同变量，不会造成链接冲突。（注意：但如果头文件中的const量的初始值依赖于某个函数，而每次调用此函数的返回值不固定的话，会导致不同的编译单元中看到的该const量的值不相等。猜测：此时将该const量作为某个类的static成员可能会解决此问题。）

const修饰指针与引用

const修饰引用时，其意义与修饰变量相同。但const在修饰指针时，规则就有些复杂了。

简单的说，可以将指针变量的类型按变量名左边最近的‘*’分成两部分，右边的部分表示指针变量自己的性质，而左边的部分则表示它指向元素的性质：

```
1 const int *p1; // p1 is a non-const pointer and points to a const int
2 int * const p2; // p2 is a const pointer and points to a non-const int
3 const int * const p3; // p3 is a const pointer and points to a const it
4 const int *pa1[10]; // pa1 is an array and contains 10 non-const pointer
point to a const int
5 int * const pa2[10]; // pa2 is an array and contains 10 const pointer poi
nt to a non-const int
6 const int (* p4)[10]; // p4 is a non-const pointer and points to an array
contains 10 const int
7 const int (*pf)(); // pf is a non-const pointer and points to a function
which has no arguments and returns a const int
8 ...
```

const指针的解读规则差不多就是这些了……

指针自身为const表示不可对该指针进行赋值，而指向物为const则表示不可对其指向进行赋值。因此可以将引用看成是一个自身为const的指针，而const引用则是const Type * const指针。

指向为const的指针是不可以赋值给指向为非const的指针，const引用也不可以赋值给非const引用，但反过来就没有问题了，这也是为了保证const语义不被破坏。

可以用const_cast来去掉某个指针或引用的const性质，或者用static_cast来为某个非const指针或引用加上const性质：

```
1 int i;
2 const int *cp = &i;
3 int *p = const_cast<int *>(cp);
4 const int *cp2 = static_cast<const int *>(p); // here the static_cast is
optional
```

C++类中的this指针就是一个自身为const的指针，而类的const方法中的this指针则是自身和指向都为const的指针。

类中的const成员变量

类中的const成员变量可分为两种：非static常量和static常量。

非static常量：

类中的非static常量必须在构造函数的初始化列表中进行初始化，因为类中的非static成员是在进入构造函数的函数体之前就要构造完成的，而const常量在构造时就必须初始化，构造后的赋值会被编译器阻止。

```
class B {
public:
    B(): name("aaa") {
        name = "bbb"; // !error
    }
private:
    const std::string name;
};
```

static常量：

static常量是在类中直接声明的，但要在类外进行唯一的定义和初始值，常用的方法是在对应的.cpp中包含类的static常量的定义：

```
// a.h
class A {
    ...
    static const std::string name;
};

// a.cpp
const std::string A::name("aaa");
```

一个特例是，如果static常量的类型是内置的整数类型，如char、int、size_t等，那么可以在类中直接给出初始值，且不需要在类外再进行定义了。编译器会将这种static常量直接替换为相应的初始值，相当于宏替换。但如果在代码中我们像正常变量那样使用这个static常量，如取它的地址，而不是像宏一样只使用它的值，那么我们还是需要在类外给它提供一个定义，但不需要初始值了（因为在声明处已经有了）。

```
// a.h
```

```
class A {
    ...
    static const int SIZE = 50;
};

// a.cpp
const int A::SIZE = 50; // if use SIZE as a variable, not a macro
```

const修饰函数

C++中可以用const去修饰一个类的非static成员函数，其语义是保证该函数所对应的对象本身的const性。在const成员函数中，所有可能违背this指针const性（const成员函数中的this指针是一个双const指针）的操作都会被阻止，如对其它成员变量的赋值以及调用它们的非const方法、调用对象本身的非const方法。但对一个声明为mutable的成员变量所做的任何操作都不会被阻止。这里保证了一定的逻辑常量性。

另外，const修饰函数时还会参与到函数的重载中，即通过const对象、const指针或引用调用方法时，优先调用const方法。

```
class A {
public:
    int &operator[](int i) {
        ++cachedReadCount;
        return data[i];
    }
    const int &operator[](int i) const {
        ++size; // !error
        --size; // !error
         ++cachedReadCount; // ok
         return data[i];
     }
private:
    int size;
    mutable cachedReadCount;
    std::vector<int> data;
};

A &a = ...;
const A &ca = ...;
```

```
int i = a[0]; // call operator[]
int j = ca[0]; // call const operator[]
a[0] = 2; // ok
ca[0] = 2; // !error
```

这个例子中，如果两个版本的operator[]有着基本相同的代码，可以考虑在其中一个函数中去调用另一个函数来实现代码的重用（参考Effective C++）。这里我们只能用非const版本去调用const版本。

```
int &A::operator[](int i) {
    return const_cast<int &>(static_cast<const A &>(*this).operator[](i));
}
```

其中为了避免调用自身导致死循环，首先要将*this转型为const A &，可以使用static_cast来完成。而在获取到const operator[]的返回值后，还要手动去掉它的const，可以使用const_cast来完成。一般来说const_cast是不推荐使用的，但这里我们明确知道我们处理的对象其实是非const的，那么这里使用const_cast就是安全的。

constexpr

constexpr是C++11中新增的关键字，其语义是“常量表达式”，也就是在编译期可求值的表达式。最基础的常量表达式就是字面值或全局变量/函数的地址或sizeof等关键字返回的结果，而其它常量表达式都是由基础表达式通过各种确定的运算得到的。constexpr值可用于enum、switch、数组长度等场合。constexpr所修饰的变量一定是编译期可求值的，所修饰的函数在其所有参数都是constexpr时，一定会返回constexpr。

```
constexpr int Inc(int i) {
    return i + 1;
}

constexpr int a = Inc(1); // ok
constexpr int b = Inc(cin.get()); // !error
constexpr int c = a * 2 + 1; // ok
```

constexpr还能用于修饰类的构造函数，即保证如果提供给该构造函数的参数都是constexpr，那么产生的对象中的所有成员都会是constexpr，该对象也就是

constexpr对象了，可用于各种只能使用constexpr的场合。注意，constexpr构造函数必须有一个空的函数体，即所有成员变量的初始化都放到初始化列表中。

```
structA {
 constexprA(intxx, intyy): x(xx), y(yy) {}
 intx, y;
};
constexprA a(1, 2);
enum{SIZE_X = a.x, SIZE_Y = a.y};
```

constexpr的好处：

1. 是一种很强的约束，更好地保证程序的正确语义不被破坏。
2. 编译器可以在编译期对constexpr的代码进行非常大的优化，比如将用到的constexpr表达式都直接替换成最终结果等。
3. 相比宏来说，没有额外的开销，但更安全可靠。