1.const含义

常类型是指使用类型修饰符**const**说明的类型，常类型的变量或对象的值是不能被更新的。

2.const作用

- 可以定义常量

```
const int a=100;
```

- 类型检查
    - const常量与#define宏定义常量的区别：

> ~~**const常量具有类型，编译器可以进行安全检查；#define宏定义没有数据类型，只是简单的字符串替换，不能进行安全检查。**~~感谢两位大佬指出这里问题，见：issue

    - const定义的变量只有类型为整数或枚举，且以常量表达式初始化时才能作为常量表达式。
    - 其他情况下它只是一个 const 限定的变量，不要将与常量混淆。

- 防止修改，起保护作用，增加程序健壮性

```
void f(const int i){
    i++; //error!
}
```

- 可以节省空间，避免不必要的内存分配
    - const定义常量从汇编的角度来看，只是给出了对应的内存地址，而不是像#define一样给出的是立即数。
    - const定义的常量在程序运行过程中只有一份拷贝，而#define定义的常量在内存中有若干个拷贝。

3.const对象默认为文件局部变量

注意：非const变量默认为extern。要使const变量能够在其他文件中访问，必须在文件中显式地指定它为extern。

> 未被const修饰的变量在不同文件的访问

```
// file1.cpp
int ext
// file2.cpp
#include<iostream>

extern int ext;
int main(){
```

```
    std::cout<<(ext+10)<<std::endl;
}
```

> const常量在不同文件的访问

```
//extern_file1.cpp
extern const int ext=12;
//extern_file2.cpp
#include<iostream>
extern const int ext;
int main(){
    std::cout<<ext<<std::endl;
}
```

> 小结：
> 可以发现未被const修饰的变量不需要extern显式声明！而const常量需要显式声明extern，并且需要做初始化！因为常量在定义后就不能被修改，所以定义时必须初始化。

4.定义常量

```
const int b = 10;
b = 0; // error: assignment of read-only variable ‘b’
const string s = "helloworld";
const int i,j=0 // error: uninitialized const ‘i’
```

上述有两个错误：

- b 为常量，不可更改！
- i 为常量，必须进行初始化！(因为常量在定义后就不能被修改，所以定义时必须初始化。)

5.指针与const

与指针相关的const有四种：

```
const char * a; //指向const对象的指针或者说指向常量的指针。
char const * a; //同上
char * const a; //指向类型对象的const指针。或者说常指针、const指针。
const char * const a; //指向const对象的const指针。
```

> **小结：**
> 如果*const*位于*的左侧，则const就是用来修饰指针所指向的变量，即指针指向为常量；
> 如果const位于*的右侧，*const*就是修饰指针本身，即指针本身是常量。

具体使用如下：

（1） 指向常量的指针

```
const int *ptr;
*ptr = 10; //error
```

ptr是一个指向int类型const对象的指针，const定义的是int类型，也就是ptr所指向的对象类型，而不是ptr本身，所以ptr可以不用赋初始值。但是不能通过ptr去修改所指对象的值。

除此之外，也不能使用void*指针保存const对象的地址，必须使用const void*类型的指针保存const对象的地址。

```
const int p = 10;
const void * vp = &p;
void *vp = &p; //error
```

另外一个重点是：**允许把非const对象的地址赋给指向const对象的指针**。

将非const对象的地址赋给const对象的指针:

```
const int *ptr;
int val = 3;
ptr = &val; //ok
```

我们不能通过ptr指针来修改val的值，即使它指向的是非const对象!

我们不能使用指向const对象的指针修改基础对象，然而如果该指针指向了非const对象，可用其他方式修改其所指的对象。可以修改const指针所指向的值的，但是不能通过const对象指针来进行而已！如下修改:

```
int *ptr1 = &val;
*ptr1=4;
cout<<*ptr<<endl;
```

> 小结:
> 1.对于指向常量的指针，不能通过指针来修改对象的值。
> 2.不能使用void*指针保存const对象的地址，必须使用const void*类型的指针保存const对象的地址。
> 3.允许把非const对象的地址赋值给const对象的指针，如果要修改指针所指向的对象值，必须通过其他方式修改，不能直接通过当前指针直接修改。

（2） 常指针

const指针必须进行初始化，且const指针的值不能修改。

```

```

```
#include<iostream>
using namespace std;
int main(){

    int num=0;
    int * const ptr=&num; //const指针必须初始化！且const指针的值不能修改
    int * t = &num;
    *t = 1;
    cout<<*ptr<<endl;
}
```

上述修改ptr指针所指向的值，可以通过非const指针来修改。

最后，当把一个const常量的地址赋值给ptr时候，由于ptr指向的是一个变量，而不是const常量，所以会报错，出现：const int* -> int *错误！

```
#include<iostream>
using namespace std;
int main(){
    const int num=0;
    int * const ptr=&num; //error! const int* -> int*
    cout<<*ptr<<endl;
}
```

上述若改为 const int *ptr或者改为const int *const ptr，都可以正常！

（3）**指向常量的常指针**

理解完前两种情况，下面这个情况就比较好理解了：

```
const int p = 3;
const int * const ptr = &p;
```

ptr是一个const指针，然后指向了一个int 类型的const对象。

6.函数中使用const

> const修饰函数返回值

这个跟const修饰普通变量以及指针的含义基本相同：

（1）**const int**

```
const int func1();
```

这个本身无意义，因为参数返回本身就是赋值给其他的变量！

（2）**const int***

```
const int* func2();
```

指针指向的内容不变。

（3）int *const

```
int *const func2();
```

指针本身不可变。

> const修饰函数参数

（1）传递过来的参数及指针本身在函数内不可变，无意义！

```
void func(const int var); // 传递过来的参数不可变
void func(int *const var); // 指针本身不可变
```

表明参数在函数体内不能被修改，但此处没有任何意义，var本身就是形参，在函数内不会改变。包括传入的形参是指针也是一样。

输入参数采用“值传递”，由于函数将自动产生临时变量用于复制该参数，该输入参数本来就无需保护，所以不要加const 修饰。

（2）**参数指针所指内容为常量不可变**

```
void StringCopy(char *dst, const char *src);
```

其中src 是输入参数，dst 是输出参数。给src加上const修饰后，如果函数体内的语句试图改动src的内容，编译器将指出错误。这就是加了const的作用之一。

（3）**参数为引用，为了增加效率同时防止修改。**

```
void func(const A &a)
```

对于非内部数据类型的参数而言，象void func(A a) 这样声明的函数注定效率比较低。因为函数体内将产生A类型的临时对象用于复制参数a，而临时对象的构造、复制、析构过程都将消耗时间。

为了提高效率，可以将函数声明改为void func(A &a)，因为“引用传递”仅借用一下参数的别名而已，不需要产生临 时对象。

> 但是函数void func(A &a) 存在一个缺点：
>
> “引用传递”有可能改变参数a，这是我们不期望的。解决这个问题很容易，加const修饰即可，因此函数最终成为 void func(const A &a)。

以此类推，是否应将void func(int x) 改写为void func(const int &x)，以便提高效率？完全没有必要，因为内部数

据类型的参数不存在构造、析构的过程，而复制也非常快，“值传递”和“引用传递”的效率几乎相当。

> 小结：
> 1.对于非内部数据类型的输入参数，应该将“值传递”的方式改为“const 引用传递”，目的是提高效率。例如将void func(A a) 改为void func(const A &a)。
>
> 2.对于内部数据类型的输入参数，不要将“值传递”的方式改为“const 引用传递”。否则既达不到提高效率的目的，又降低了函数的可理解性。例如void func(int x) 不应该改为void func(const int &x)。

以上解决了两个面试问题：

- 如果函数需要传入一个指针，是否需要为该指针加上const，把const加在指针不同的位置有什么区别；
- 如果写的函数需要传入的参数是一个复杂类型的实例，传入值参数或者引用参数有什么区别，什么时候需要为传入的引用参数加上const。

7.类中使用const

在一个类中，任何不会修改数据成员的函数都应该声明为const类型。如果在编写const成员函数时，不慎修改 数据成员，或者调用了其它非const成员函数，编译器将指出错误，这无疑会提高程序的健壮性。

使用const关键字进行说明的成员函数，称为常成员函数。只有常成员函数才有资格操作常量或常对象，没有使用const关键字进行说明的成员函数不能用来操作常对象。

对于类中的const成员变量必须通过初始化列表进行初始化，如下所示：

```
class Apple{
private:
    int people[100];
public:
    Apple(int i);
    const int apple_number;
};

Apple::Apple(int i):apple_number(i)
{

}
```

const对象只能访问const成员函数,而非const对象可以访问任意的成员函数,包括const成员函数.

例如：

```
//apple.cpp
class Apple
{
private:
    int people[100];
public:
    Apple(int i);
    const int apple_number;
    void take(int num) const;
    int add(int num);
    int add(int num) const;
    int getCount() const;

};
//main.cpp
#include<iostream>
#include"apple.cpp"
using namespace std;

Apple::Apple(int i):apple_number(i)
{

}
int Apple::add(int num){
    take(num);
}
int Apple::add(int num) const{
    take(num);
}
void Apple::take(int num) const
{
    cout<<"take func "<<num<<endl;
}
int Apple::getCount() const
{
    take(1);
//    add(); //error
    return apple_number;
}
int main(){
    Apple a(2);
    cout<<a.getCount()<<endl;
    a.add(10);
    const Apple b(3);
    b.add(100);
    return 0;
}
```

> 编译： g++ -o main main.cpp apple.cpp

结果：

```
take func 1
2
take func 10
take func 100
```

上面getCount()方法中调用了一个add方法，而add方法并非const修饰，所以运行报错。也就是说const对象只能访问const成员函数。

而add方法又调用了const修饰的take方法，证明了非const对象可以访问任意的成员函数,包括const成员函数。

除此之外，我们也看到add的一个重载函数，也输出了两个结果，说明const对象默认调用const成员函数。

我们除了上述的初始化const常量用初始化列表方式外，也可以通过下面方法：

第一：将常量定义与static结合，也就是：

```
static const int apple_number
```

第二：在外面初始化：

```
const int Apple::apple_number=10;
```

当然，如果你使用c++11进行编译，直接可以在定义出初始化，可以直接写成：

```
static const int apple_number=10;
// 或者
const int apple_number=10;
```

这两种都在c++11中支持！

编译的时候加上`-std=c++11`即可！

这里提到了static，下面简单的说一下：

在C++中，static静态成员变量不能在类的内部初始化。在类的内部只是声明，定义必须在类定义体的外部，通常在类的实现文件中初始化。

在类中声明：

```
static int ap;
```

在类实现文件中使用：

```
int Apple::ap=666
```

对于此项，c++11不能进行声明并初始化，也就是上述使用方法。