# 32/64位OS各数据类型的大小

**区别1：指针**

因为32位OS只需4个字节就能寻址整个内存空间，而64系统需要8字节。

**区别2：long 和 unsigned long**

| **类型** | **32位编译器** | **64位编译器** |
| --- | --- | --- |
| char | 1 | 1 |
| char\* | 4 | 8 |
|  |  |  |
| short int | 2 | 2 |
| int | 4 | 4 |
| unsigned int | 4 | 4 |
| float | 4 | 4 |
| double | 8 | 8 |
| long | 4 | 8 |
| long long | 8 | 8 |
| unsigned long | 4 | 8 |

# 字、字节 和 字长

**字**

字是word 长度与架构有关，如mips包括32个二进制位

**字节**

byte 包括8个二进制位

**字长**

xx位机的xx位是指字长。这个字和word不一样，指这种CPU一次能运算的数据长度，32位机就是一次运算32个二进制位，64位机就是一次运算64个二进制位。

# 原码、反码、补码

## 3.1 原码

**正数的原码** 是其二进制本身；

**负数X的原码**：符号位为1，数值部分取 X绝对值(若x=-1，则数值部分则取1) 的二进制。

## 3.2 反码

**正数的反码** 是其本身

**负数的反码** 是在其原码的基础上, 符号位不变，其余各个位取反，

例如：

[+1] =[00000001]原 = [00000001]反

[-1] =[10000001]原 = [11111110]反

## 3.3 补码

**正数的补码** 就是其本身

**负数的补码** 是在其原码的基础上, 符号位不变, 其余各位取反, 最后+1. (即在反码的基础上+1)

[+1] = [00000001]原 = [00000001]反 = [00000001]补

[-1] = [10000001]原 = [11111110]反 = [11111111]补

# 为何要使用原码, 反码和补码

对于计算机, 加减乘数已经是最基础的运算, 要设计的尽量简单. 计算机辨别"符号位"显然会让计算机的基础电路设计变得十分复杂! 于是人们想出了将符号位也参与运算的方法.

下面来看 计算十进制的表达式: 1-1=0

**首先来看原码:**

1 - 1 = 1 + (-1) = [00000001]原 + [10000001]原= [10000010]原 = -2

如果用原码表示, 让符号位也参与计算, 显然对于减法来说, 结果是不正确的.这也就是为何计算机内部不使用原码表示一个数.

**为了解决原码做减法的问题, 出现了反码:**

1 - 1 = 1 + (-1) = [0000 0001]原 + [1000 0001]原 = [0000 0001]反+ [1111 1110]反= [1111 1111]反= [1000 0000]原= -0

用反码计算减法, 结果的真值部分是正确的. 而唯一的问题其实就出现在"0"这个特殊的数值上. 虽然人们理解上+0和-0是一样的, 但是0带符号是没有任何意义的. 而且会有[0000 0000]原和[1000 0000]原两个编码表示0.

**于是补码的出现, 解决了0的符号以及两个编码的问题:**  
1-1 = 1 + (-1) = [0000 0001]原 + [1000 0001]原 = [0000 0001]补+ [1111 1111]补= [0000 0000]补=[0000 0000]原 = 0

这样0用[0000 0000]表示, 而以前出现问题的-0则不存在了.而且可以用[1000 0000]表示-128

# 计算机中 有符号数 和 无符号数 的表示

计算机不能区分有符号数和无符号数，一个数字你用 有符号数 和 无符号数 解释得到的结果是不一样的：

有符号数中，最高位作为符号位，0为正数，1位负数。

无符号数中，所有的位都用于直接表示该值的大小。例如：

0xFF 有符号: -1; 无符号: 255

0x02 有符号: 2; 无符号: 2

# 若将负数赋给了无符号数会发生什么？

正如上面所说，计算机不能区分有符号数和无符号数，一个数字你用 有符号数 和 无符号数 解释得到的结果是不一样的，比如 0xFF，如果你把它看作是有符号数，那他就等于 -1，但如果你把它看作是无符号数，那他就等于 255

# 含有 无符号数 的循环为什么有可能死循环？

例如如下代码：因为无符号数取值总是非负，所以（i >= 0）恒成立，造成死循环

unsigned int i = n;

for(; i >= 0; --i){

printf("%i\n", a[i]);

}

# 有符号数 和 无符号数 之间的运算可能带来哪些问题？

**规则：** 当 无符号数 和 有符号数 一起运算时，有符号数 也会被自动隐式转换为无符号数。

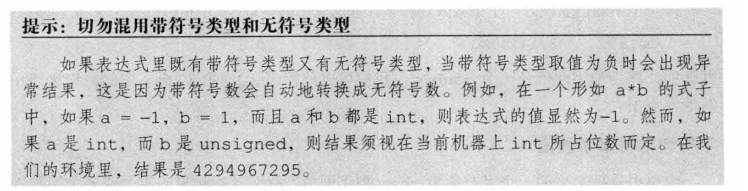
## 8.1 有符号数 和 无符号数 的比较

1. **int** fun()
2. {
3. unsigned **int** a = 10;
4. **int** b = -100;
5. **int** c;
6. (a+b > 0) ? (c=1) : (c=0);
7. **return** c;
8. }

以上答案是1，是不是觉得很奇怪呢？这就是有关的一个代码小陷阱了，当无符号数和有符号数一起运算时，有符号数也会被自动隐式转换为无符号数，因此这个时候对于 a+b 中 b 的值被当做一个很大的整数进行计算，所以得到的结果是大于0的；

## 8.2 有符号数 和 无符号数 的加减法

## 8.3 有符号数 和 无符号数 的乘法



# 十进制、八进制、十六进制分别怎么表示？

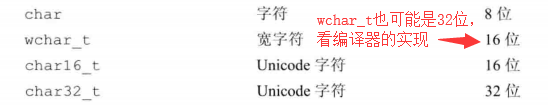
以0开头的代码八进制数，以0x或0X开头的代表十六进制数，例如：

十进制：   20        //decimal

八进制：   024      //octal

十六进制：0x14或者0X14  //hexadecimal

# C++有哪些char类型？



**注意**：wchar\_t没有指定到底几个字节，根据编译器操作系统而有不同定义：2个字节或 4个字节。

# 有了char之后为什么还需要其它char类型呢？

**① wchar\_t 存在的原因：**

char是八位字符类型，最多能包含256中字符，许多的外文字符集所包含的字符数目超过256个（如中文），char型不能表示。

wchar\_t  bob = L'p';

**② char16\_t和char32\_t的产生原因**：

我们知道，wchar\_t没有指定到底几个字节（可能是2个也可能4个），所以对于需要精准控制类型大小的程序来说，wchar\_t已经不能满足需求，因此新增了类型char16\_t 和 char32\_t 。

# 变量的 声明 和 定义

* **定义**：**用于为变量分配存储空间**，还可为变量指定初始值。程序中，变量有且仅有一个定义（只能定义一次）。
* **声明**：用于向程序表明变量的类型和名字，可多次声明，如果要声明，而不是定义一个变量，应该这么做：

① 在变量名前面加上 extern关键字；

② 不要 显示初始化 该变量。

* 定义也是声明，extern声明不是定义

**注意：**

变量在使用前就要被定义或者声明。

在一个程序中，变量只能定义一次，却可以声明多次。

定义分配存储空间，而声明不会。

* extern告诉编译器变量在其他地方定义了。

例如：

extern int i; //声明，不是定义  
int i; //声明，也是定义，未初始化

* 如果声明有初始化式，就被当作定义，即使前面加了extern。 **只有当extern声明位于函数外部时，才可以被初始化。**

例如：

extern int i; // 声明，不是定义

extern double pi = 3.141592654; // 定义

* 函数的声明和定义区别比较简单，带有{ }的就是定义，否则就是声明。

例如：

extern double max(double d1,double d2); //声明

* 除非有extern关键字，否则都是变量的定义。

例如：

extern int i; //声明  
int i; //定义

# 如何区分定义和初始化？

**定义** ：用于为变量分配存储空间，用于存放对应类型的数据，变量名就是对相应的内存单元的命名，还可为变量指定初始值。程序中，变量有且仅有一个定义。

**初始化** ：初始化是给对象赋予初值的过程，初始化由构造函数执行。所谓的default构造函数是一个可被调用而不带任何实际参数者，这样的构造函数要不没有参数，要不就是每个参数都有缺省值。

# 什么时候为变量分配空间？

变量被定义的时候。

# 初始化 和 赋值 的区别

**初始化**：

在创建变量时赋予其一个初始值。

**赋值**：

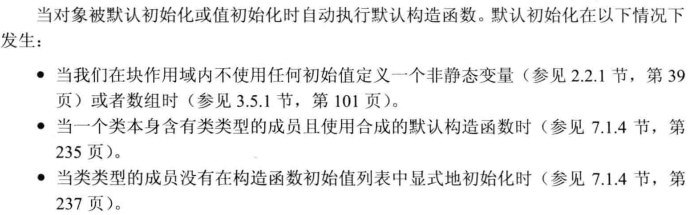
把对象当前的值擦掉，然后以一个新值来替代。

**注意**：这两个概念虽然很相似，所以很容易混淆。

# C++中的几种初始化方式

**注意**：没列表初始化用的是花括号，即：{ }，直接初始化用的是小括号，即：( )

## 默认初始化



若定义变量的时候没有指定初值，则变量将被默认初始化，此时变量被赋予了“默认值”，默认值到底是什么由变量类型决定，同时变量定义的位置也会对此产生影响。

* **内置类型**（如int，double，bool等）

定义于所有的函数体之外(全局变量) 的变量被初始化为0；

定义在函数体内部的内置类型变量(局部变量) 将不被初始化，这样的内置类型变量的值是未定义的。（但局部static变量若没有显示的初始值，它将执行值初始化，内置类型的局部static变量将被初始化为0。）

1. #include <iostream>
3. **using** **namespace** std;
5. **int** global;
7. **int** main()
8. {
9. **int** local;
10. cout<<"global:"<<global<<endl;
11. cout<<"local :"<<local<<endl;
13. **return** 0;
14. }

在上面的代码中，global变量将被默认初始化为0，而local变量 的值将是未定义的。

* **类类型**（如string或其他自定义类型）

根据默认构造函数(参数为空的构造函数)提供合适的初始化。如果该类没有默认构造函数，则会引发错误。因此，建议为每个类都定义一个默认构造函数（=default）。

## 直接初始化 和 拷贝初始化（C++ primer P76）

如果用等号(=)初始化一个变量，实际上执行的是**拷贝初始化**，编译器把等号右侧的初始值拷贝到新建的对象中去。

如果在新创建的变量右侧使用括号将初始值括住（不用等号），则执行的是**直接初始化（direct initialization）**，例如：

1. int a = 0; // 有等于号，是拷贝初始化
2. string s5 = “hiya”;  // 有等于号，是拷贝初始化
3. string s6 (“hiya”); //没有等于号，有小括号，是直接初始化
4. string s7(10,’c’); //直接初始化，内容是 cccccccccc

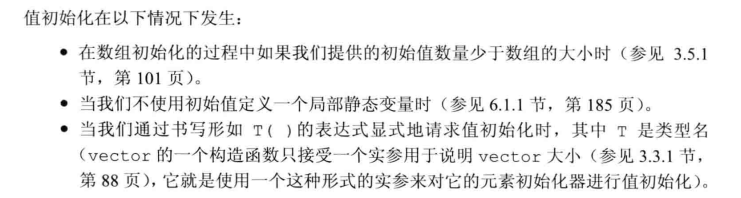
看下面一个语句：

string s8 = string(10, ’c’); // 拷贝初始化

其实它相当于下面两条语句：

1. string tmp= string(10, 'c'); // 拷贝初始化
2. string s8 = tmp;

## 值初始化



值初始化是定义对象时，要求初始化，但没有给出初始值的行为

* **局部静态变量**（C++ primer P88）

和其它局部变量不一样的是（默认初始化），局部static变量若没有显示的初始值，它将执行值初始化，内置类型的局部static变量将被初始化为0。

* **只提供vector对象容纳元素数量而不提供初始值的时候**（C++ primer P88）

在数组初始化的过程中，如果提供的初始值数量少于数组的大小，剩下的元素会进行值初始化；

* **当我们通过书写形如T()的表达式显示地请求值初始化时；**

string \*ps = new string();  // 值初始化为空string

int \* pi1 = new int;    // 默认初始化，\*pi1的值是未定义的

int \* pi2 = new int();  // 值初始化为0

## ****列表初始化****

用花括号给变量初始化，如：

1. **int** a{12};
2. string s{"123"};
3. vector<**int**> vec{1,2,3}

**注意：**当用列表初始化来初始化内置类型的时候，如果存在初始值丢失的风险，则编译器将报错，例如：

1. **long** **double** ld = 3.1415926;
2. **int** a{ld}, b = {ld};  // 错误：这是列表初始化，存在信息丢失的风险，编译器将报错
3. **int** c(ld); d = ld;    // 正确：c是直接初始化，d是拷贝初始化，执行了类型转换，但丢失了精度

# 什么是 分离式编译？

分离编译模式源于C语言，在C++中继续沿用。

分离编译模式是指：一个程序（项目）由若干个源文件共同实现，而每个源文件单独编译生成目标文件，最后将所有目标文件( .o文件)链接起来形成单一的可执行文件( .out文件)的过程。

# 为什么要 分离式编译？(有什么好处？)

在实际开发大型项目的时候，不可能把所有的源程序都放在一个头文件中，而是分别由不同的程序员开发不同的模块，再将这些模块汇总成为最终的可执行程序。

这里就涉及到不同的模块（源文件）定义的函数和变量之间的相互调用问题。C/C++语言所采用的方法是：

只要给出函数原型（或外部变量声明），就可以在本源文件中使用该函数（或变量）。

每个源文件都是独立的编译单元，在当前源文件中使用但未在此定义的变量或者函数，就假设在其他的源文件中定义好了。

每个源文件生成独立的目标文件（obj文件），然后通过连接（Linking）将目标文件组成最终的可执行文件。

# C++ 为什么要将 声明 和 定义 区分开来？

为了支持 **分离式编译**。

声明让变量为程序所知，因此一个文件想使用其他文件中定义的变量，就必须声明它，这样就避免了多次定义同一变量。

声明和定义的区别看起来不重要，但是却很关键，如果要在多个文件中使用同一个变量，就必须将 声明 和 定义 分离。此时，一个变量的定义 只能且必须 出在一个文件中，而其它用到该变量的文件必须对其进行声明，但却决不能重复定义。

# 如何在多个源文件中共享全局变量？

将 声明 和 定义 分离。此时，一个变量的定义 只能且必须 出在一个文件中，而其它用到该变量的文件必须对其进行声明，但却决不能重复定义。

# 静态类型语言 和 动态类型语言的主要区别是？

**静态类型语言**：

静态类型的 类型检查 发生在 编译阶段，如果在编译时知道变量的类型，则语言是静态类型的。C/C++、C#、JAVA都是静态类型语言的典型代表。

因为类型检查发生在编译阶段，所以我们在使用一个变量之前必须声明其类型。

**动态类型语言**：

在运行期间才去做数据类型检查的语言。在用动态语言编程时，不用给变量指定数据类型，该语言会在你第一次赋值给变量时，在内部将数据类型记录下来。Ruby、Python等都属于动态语言

**主要区别**：

它们之间的区别主要是 类型检查发生在哪个阶段，如果是发生在编译阶段则是静态类型语言，如果发生在运行时则是动态类型语言。

# 静态类型语言 和 动态类型语言各自的优缺点是？

* **静态类型语言**

主要优点在于其结构非常规范，便于调试，方便类型安全；

缺点是为此需要写更多的类型相关代码，导致不便于阅读、不清晰明了。

* **动态类型语言**

优点在于方便阅读，不需要写非常多的类型相关的代码；

缺点自然就是不方便调试，命名不规范时会造成读不懂，不利于理解等。

# 强类型语言 和 弱类型语言

* **强类型语言**，

一个变量不经过强制转换，它永远是这个数据类型，不允许隐式的类型转换。举个例子：如果你定义了一个double类型变量a,不经过强制类型转换那么程序int b = a无法通过编译。典型代表是Java。

* **弱类型语言**：

它与强类型语言定义相反,允许编译器进行隐式的类型转换，典型代表C/C++。

# 如何区分强/弱类型语言 和 静/动态类型语言 这两个概念？

强/弱类型语言： 是否容忍 隐式类型转换。

静/动态类型语言：类型检查发生在什么阶段。

# 在局部作用域中使用 同名的全局变量

## 作用域操作符：“::”

## 作用一：表示定义某个域的函数或类型;

Test::Test()引用Test类的Test()构造函数;

例:

class Test { Test();};Test::Test() {}

## 作用二：表示调用全局的函数或类型;

::value引用全局变量;

例:

1. #include <iostream>
2. // Program for illustration purposes only: It is bad style for a function
3. // to use a global variable and also define a local variable with the same name
4. **int** reused = 42; // reused has global scope
5. **int** main()
6. {
7. **int** unique = 0; // unique has block scope
8. // output #1: uses global reused; prints 42 0
9. std::cout << reused << " " << unique << std::endl;
10. **int** reused = 0; // new, local object named reused hides global reused
11. // output #2: uses local reused; prints 0 0
12. std::cout << reused << " " << unique << std::endl;
13. // output #3: explicitly requests the global reused; prints 42 0
14. std::cout << ::reused << " " << unique << std::endl;
15. **return** 0;
16. }

**注意：**虽然可以定义一个和全局变量同名的全局变量，但是最好不要这么做，这样很容易产生歧义。

# 当定义一个引用的时候，发生了什么？

当定义引用的时候，程序把引用和它的初值 **绑定** 在一起，而不是拷贝，而且将一直绑定在一起，所以以后不能将该引用绑定到另外一个对象上面。

# 使用引用的时候需要注意什么？

1) 引用必须初始化

**int** &refVal3;  // 错误：引用类型必须初始化

2) 引用只能绑定到对象上，不能将 字面值、计算结果上面：

**int** &refVal4 = 10;  // 错误：引用类型的初始值必须是一个对象

3）引用的类型 必须 和绑定的对象 严格匹配，例如：

1. **double** dval = 3.14;
2. **int** &refVal5 = dval;  // 错误：此处引用类型的初始值必须也是 int类型的。

4）引用在定义之后不能将该引用绑定到另外一个对象上面

# 引用 和 指针有什么不同？

1) 引用必须初始化

2) 指针可以修改

# 既然有了指针，为什么还需要引用呢？

上网搜了下，大家说是为了运算符重载，留着以后看了运算符重载再说。

# 26. 如何理解：int\*p; int \*& r = p; ？

面对一个比较复杂的指针或引用的声明语句，从右往左 阅读有助于弄清它的真实含义，离变量最近的符号对变量的类型有这最直接的影响，因此它相当于 (int \*) & r ， 也就是说r是一个引用，他引用的类型是 int\*，因此r必须初始化。

# 为什么引用一定要初始化？

由于引用一旦创建就不可更改，所以**引用必须初始化**（否则定义一个默认值且不可修改的变量没有任何意义）。

# 一个引用占多少内存？

引用和const修饰的变量很相似。实际上，C++编译器在编译的过程中使用**常量指针**作为引用的内部实现，因此引用所占用的空间大小与指针相同。从我们使用的角度看，引用会让我们误会它只是一个别名，没有自己的存储空间。这是C++为了使用性而做出的细节隐藏。

# const 变量初始化的问题

1) 用const变量 初始化 一个 非const变量

2) 用 非const变量 初始化 一个const变量

以上两种情况都没问题，如：

1. **int** i = 42;
2. **const** **int** ci = i;  // ok: the value in i is copied into ci
3. **int** j = ci;  // ok: the value in ci is copied into j

# 为什么const变量必须初始化？

由于const一旦创建就不可更改，所以**const对象必须初始化**（否则定义一个默认值且不可修改的变量没有任何意义）。

# const变量的值在什么时候可以确定？

当以编译时初始化的方式定义一个const对象时，如：

**extern** **const** **int** bufsize = 1024;

编译器将在编译过程中把用到该变量的地方都替换成对应的值，也就是说，编译器会找到代码中所有用到bufsize 的地方，然后用512替换。

# const全局变量的默认链接属性是什么？

在默认情况下，全局变量的链接属性为：**外部的**，但 const全局变量 比较特殊，默认const全局变量的链接属性为：**内部的**，也就是说，在 C++ 看来，全局 const 定义就像使用了 static 说明符一样。

# 如何在头文件中定义一个const全局变量？

如果想 **在头文件中** 定义一个const全局变量，需要这么做：

1) 在头文件以extern**声明**该const变量；

2) 在实现文件中 加上extern**定义**该变量；

2) 注意，无论是头文件还是实现文件，都要加上 extern

例如：

**头文件如下：**

1. // QVNDefine.h
2. #ifndef QVNDefine\_h
3. #define QVNDefine\_h
4. **extern** **const** **int** a;
5. #endif /\* QVNDefine\_h \*/

**实现文件如下：**

1. // QVNDefine.cpp
2. #include "QVNDefine.h"
3. **extern** **const** **int** a = 1;

**之所以不能再头文件中定义全局const变量，是因为如果多个实现文件包含该头文件，则会发生重复定义问题，**

# 将const变量放在头文件中时需要注意什么？

只能声明（加extern），不能定义，定义应该放在实现文件中。

# const全局变量和局部变量的存放位置

const全局变量：存放在数据段中的只读数据段(ro data)中。

const局部变量：存放在栈中。

# const 和 #define

35.1 区别：

**① 处理的时间（对应的处理器）不一样：**

**const常量:**  由 **编译器** 处理，它会对const常量进行类型检查和作用域检查。

**define宏定义:** 由 **预处理器** 处理，直接进行文本替换，不会进行各种检查。

**② 存储方式不一样**

**const常量：** 存放在数据段（初始化数据段）的只读数据段(ro data)中，只有一份拷贝

**define宏定义:** 直接进行文本替换。

**③ 效率不一样**

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。

**④ 能否调试**

define定义的常量不能被调试，const常量可以。

**⑤ 类型、安全检查**

define宏没有类型，不做类型检查，只做简单的展开；

const常量有类型，在编译阶段会执行类型检查。

# const对象在什么时候初始化？

1. **const** **int** i = get\_size();  // ok: 运行时初始化
2. **const** **int** j = 42;  // ok: 编译时初始化

# const引用

1） 若被引用的对象是 const类型，则引用的类型 必须也是 const引用；

2） 若被引用的对象是 非const类型，引用的类型 可以是 const引用。

例子如下：

1. **const** **int** ci = 1024;
2. **int** i = 2048;
3. **const** **int** &r1 = ci; // ok: both reference and underlying object are const
4. r1 = 42;   // error: r1 is a reference to const
5. **int** &r2 = ci;   // error: non const reference to a const object
6. **const** **int** &r3 = i;  // ok:

**解读：**

1） 假设 “**int** &r2 = ci;” 是合法的，则能通过 r2 去修改 ci 的值，而ci 是一个const类型，这显然是不对的。

2）需要注意的是，常量引用 仅仅对该应用可以参与的操作做出了限定，而对于所引用的对象本身未作限定，因此所引用的对象可以是一个非常量，只是不能通过常量引用去修改它而已。

# 什么类型的对象可以用来初始化const引用？

只要能从一种类型转换到另一种类型，那么就能用来初始化 const引用，例如：

1. **double** dval = 3.14159;
2. // 注意！以下3行仅对 const引用 才是合法的！
3. **const** **int** &ir = 1024;
4. **const** **int** &ir2 = dval;
5. **const** **double** &dr = dval + 1.0;

上面同样的初始化对于 非const引用 是不合法的，将导致编译错误。原因有些微妙，解释如下：

引用在内部存放的是一个对象的地址，它是该对象的别名。对于不可寻址的值，如文字常量，以及不同类型的对象，编译器为了实现引用，必须生成一个临时对象，引用时间上指向该对象，但用户不能访问它。例如:

1. **double** dval = 23;
2. **const** **int** &ri = dval;

编译器将其转换为:

1. **int** tmp = dval; // double -> int
2. **const** **int** &ri = tmp;

同理：上面代码，编译器内部转化为：

1. **double** dval = 3.14159;
3. // 对于 const int &ir = 1024;
4. // 不可寻址，文字常量
5. **int** tmp1 = 1024;
6. **const** **int** &ir = tmp1;
8. // 对于：const int &ir2 = dval;
9. //  不同类型
10. **int** tmp2 = dval;//double -> int
11. **const** **int** &ir2 = tmp2;
13. //对于 const double &dr = dval + 1.0;
14. // 另一种情况，不可寻址
15. **double** tmp3 = dval + 1.0;
16. **const** **double** &dr = tmp3;

# 指向常量的指针

其实和指向常量的引用类似：

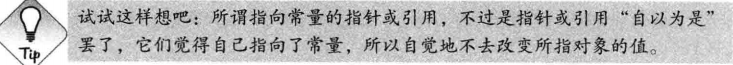
1） 若被指向的对象是 const类型，则指针的类型 必须也是 const引用；

2） 若被指向的对象是 非const类型，指针的类型 可以是 const引用。

例如：

1. **const** **double** pi = 3.14; // pi is const; its value may not be changed
2. **double** \*ptr = π  // error: ptr 是一个普通指针，而π是常量
3. **const** **double** \*cptr = π  // ok: cptr 是常量double类型
4. \*cptr = 42;  // error:  cptr是常量指针
5. **double** dval = 3.14;  // dval is a double; its value can be changed
6. cptr = &dval; // ok: 被指向的变量可以不是常量

# 如何理解指向常量的指针和引用？



# 常量指针(const指针)

1) 和引用不一样，指针本身也是对象；

2）所以可以把 **指针本身** 声明为const；

3）把指针本身声明为const意味着必须将其初始化(因为后面不允许修改指针的值了)

常量指针的声明：

1. **int** errNumb = 0;
2. **int** \***const** curErr = &errNumb; // curErr是常量指针，它将一直指向errNumb.

# 什么是顶层const、底层const？

**顶层const(top-level const )** ：指针本身是const（作用于对象本身）

**底层const(low-level const )** ：指针指向的对象是const

其实可以这么理解：指针所指向的对象 肯定是在 **更“底层”**，所以底层const指得就是指针所指向对象是const类型是。

# 分别声明一个顶层const 和 底层const：

**int** value = 1024;

顶层const

**int** \* **const**  topPtr = &value;

底层const

**const**  **int** \* topPtr = &value;

# 顶层const 和 底层const的拷贝操作

**顶层const**

不受影响。

**底层const**

拷入、拷出对象都必须有相同的底层const资格，如：

1. **int** i = 0;
2. **int** \***const** p1 = &i; // we can't change the value of p1; const is top-level
3. **const** **int** ci = 42;
4. **const** **int** \*p2 = &ci; // we can change p2; const is low-level
5. **const** **int** \***const** p3 = p2; // right-most const is top-level, left-most is not
6. **const** **int** &r = ci; // const in reference types is always low-level
8. **int** \*p = p3;  // error: p3 包含底层const含义，而 没有
9. p2 = p3;  // ok: p2和 p3都是底层const
10. p2 = &i;  // ok:  int\* 可以转换为 const int\*
11. **int** &r = ci; // error: can't bind an ordinary int& to a const int object
12. **const** **int** &r2 = i; // ok: const int& 可以绑定到一个 普通的int上

想象一下，如果拷入变量不是底层const，而拷出变量是底层const，如果拷贝合法的话，岂不是可以通过拷入变量修改指针所指向的const变量了吗？这显然是不对的。

# 什么是常量表达式？

**常量表达式（const expression）**有如下特点：

1) 值不会改变；

2) 在编译过程就能就能得到计算结果。

# 哪些属于常量表达式？

1) 字面值；

2) 用 常量表达式(如：字面值) 初始化的 const对象

# 如何判断是否常量表达式？

一个对象(表达式) 是不是 常量表达式由以下两个**共同决定**：

① 它的数据类型(是不是const)

② 它的初始值(是不是常量表达式，如字面值)

例如：

1. //是， max\_files为const int类型，且初始值为常量表达式(字面值)
2. **const** **int** max\_files = 20;
3. // 是，limit 是const int类型，且初始值max\_files 和、1 都是 常量表达式(或字面值)
4. **const** **int** limit = max\_files + 1;
5. **int** staff\_size = 27;  // 否，虽然处初始值是常量表达式，但 staff\_size的类型 不是const，因为它的值可以被改变
6. **const** **int** sz = get\_size();  // 否，sz 是const int，但初始值不是常量表达式。

# 什么是constexpr变量？

C++ 11 标准规定，允许将变量声明为 **constexpr类型** 以便**由编译器来验证**变量的值是否是一个常量表达式

**一般而言，如果你认定变量是一个常量表达式，那就把它声明成 constexpr 类型。constexpr 变量在定义时必须初始化.**

# 为什么需要constexpr变量？

常量表达式机制是为了：

提供了更多的通用的值不发生变化的表达式；

允许用户自定义的类型成为常量表达式；

提供了一种保证在编译期完成初始化的方法（可以在编译时期执行某些函数调用）；

# 指针 和 constexpr

在constexpr声明中 如果定义了一个指针，限定符constexpr 仅对指针有效，与指针所指向的对象无关：

1. **const** **int** \*p = nullptr;  // p 是一个 指向常量的指针
2. constexpr **int** \*q = nullptr;  // q 是一个 指针常量

# const 和 constexpr ？

修饰变量时没有必要同时使用const和constexpr 因为constexpr包含了 const的含义，下面两行代码的一起完全相同：

1. // 下面两行代码的意思完全相同
2. constexpr **const** **int** N = 5;
3. constexpr **int** N = 5;

# 什么时候同时使用const 和 constexpr ？

1) 有一些情况const和constexpr在修饰不同的东西，比如

1. **static** constexpr **int** N = 3;
2. **int** main()
3. {
4. constexpr **const** **int** \*NP = &N;
5. **return** 0;
6. }

在这里constexpr和const都必须要有：

① constexpr表示NP指针本身是常量表达式，而const表示指向的值是一个常量。

② 去掉const之后无法编译，因为不能用正常指针指向常量。

2) 在修饰成员函数的时候

在C++11中，对成员函数而言constexpr同样包含const的含义。然而在C++14中可能已经改变了。如

constexpr void f();

以后可能必须写成

constexpr void f() const;

虽然目前写成const仍然有效，但最好使用constexpr来防止以后修改大量代码的可能性。

# typedef 的语法规则

**TODO:**

# typedef double \*p中，p代表什么？

p是 double\* 的别名，考虑如下代码：

**double** \* a, b;

其实我们是想讲a和b都定义成 double\* 类型的，但是这行代码只将a定义成了double\*类型，b是double类型，为了达到原来的目的，我们可以这么做：

1. **typedef** **double** \*pDouble
2. pDouble a, b;

# 用typedef写 一个 含有5个元素的 int数组 的别名？

注意 [5] 要写在别名后面

* 1. **typedef** **int** arrs[5];
  2. **typedef** arrs \* p\_arr5;
  3. **typedef** p\_arr5 arrp10[10];
  4. arr5 togs;     // togs是具有5个元素的int数组
  5. p\_arr5 p2;     // p2是一个指针，指向具有元素的数组
  6. arrp10  ap;    // ap是具有十个元素的指针数组，每个指针指向具有5个元素的int数组

# 用typedef为一个struct定义别名

* 1. **typedef** **struct**{
  2. **int** a;
  3. **char** b;
  4. **double** c;
  5. } simple;

# 用typedef为 void \*test(int a,int b);取一个函数指针的别名

**格式为： typedef** 返回类型(\*别名)(形参列表)

* 1. #include <stdio.h>
  2. **typedef** **void**\*(\*Fun)(**int**,**int**);
  3. **void** \*test(**int** a,**int** b)
  4. {
  5. printf("%d,%d\n",a,b);
  6. //do something
  7. **return** NULL;
  8. }
  9. **int** main(**void**)
  10. {
  11. Fun myfun = test;//这里的Fun已经是一种类型名了
  12. myfun(1,1);
  13. **return** 0;
  14. }

# 用枚举定义一个bool变量

**typedef** **enum** { FALSE, TRUE } Boolean;

# 有哪些方法定义类型别名？

1) 用typedef

待补充例子

2) 用using

待补充例子

# 用using定义一个数别名 和 函数指针别名

定义**函数**别名（其实就是把名字移到了左边）

* 1. **using** F = **int**(**int**\*, **int**); // F is a function type, not a pointer

定义**函数指针**别名（其实就是把名字移到了左边，但（\*）不能省略！）

* 1. **using** PF = **int**(\*)(**int**\*, **int**); // 不要漏了“(\*) ”
  2. **using** PF = **int**\* (**int**\*, **int**); // 函数别名，该函数返回的是 int\*
  3. **using** PF = **int** (**int**\*, **int**); // 函数别名，该函数返回的是 int

# 使用auto需要注意的点

1) 一条声明语句只能有一个 基本数据类型

1. auto i = 0, \*p = &i;  // 正确: i是整形，p是整形指针
2. auto sz = 0, pi = 3.14;  // 错误: sz、pi的数据类型不一样

2) auto会忽略顶层const，但保留底层const

1. **const** **int** ci = i, &cr = ci;
2. auto b = ci;  // b 是一个整数( 因为ci的顶层const被忽略了)
3. auto c = cr;  // c 是一个整形 (cr是ci 的别名，因此cr是顶层const)
4. auto e = &ci;  // e 是一个const int\*(底层const得到了保留)

3) 若要 推断出的auto类型 是一个 顶层const，则需要明确指出

**const** auto f = ci; // 推断出来的是 int; 但加了const后f是顶层const

# 关于 decltype

decltype 可用来选择并返回操作数的数据类型，在此过程中，**编译器**分析表达式并得到它的类型，却不实际计算表达式的值。

**1) 常规用法：**

1. **int** ci = 0;
2. decltype(ci) x = 0; // x 是 int型

**2) 作用 于 函数**

注意对比下面两个语句，它们一个是 func()且带了\*；一个是func，有关函数指针的内容在第六章的笔记中：

decltype(func()) sum = x; // sum的类型将是 func 的返回值类型

decltype(func) \* sum1; // sum1的类型将是 func 的函数指针

**3) 作用于带有顶层const 的引用和指针**

1. **const** **int** ci = 0, &cj = ci;
2. decltype(ci) x = 0; // x 是顶层const
3. decltype(cj) y = x; // y 是 const int& ，是带有顶层cons的引用

**4) 作用于 解引用**

作用于解引用的时候，得到的是 引用的类型：

1. **int** i = 42, \*p = &i;
2. decltype(\*p) c; // 错误: \*p 是引用，因此必须初始化！

**5) 作用于引用**

和解引用一样，得到的是 引用的类型：

1. **int** i = 42, &r = i;
2. decltype(r + 0) b; // 正确: 因为r引用的是 int，因此b是一个int

**6) 作用于带括号的对象**

这个特性比较特殊，

decltype( (variable) ) **（注意是双括号）**得到的永远是引用；

decltype( variable ) 只有在variable本身是一个引用的时候才是引用

1. **int** i = 42;
2. decltype((i)) d; // 错误: d 是 int& ，因此必须初始化
3. decltype((i)) e; // 正确: e 是 int&
4. decltype(i) f;  // 正确: f 是 int

# 修改头文件后，包含它的相关源文件需要重新编译吗？

需要。头文件一旦改变，相关的源文件需要重新编译以获取更新过的声明

# C/C++ 通过什么来保证头文件多次被包含时还可以正常工作？

**1) 预处理器**

通过 **预处理器 来**保证头文件多次被包含还能正常工作，它在编译前运行一段程序，可以部分的修改我们的代码，当预处理器看到 #include标记 时就会用指定的头文件内容替代#include。

**2) 头文件保护符**

#ifndef、#define、#endif

如：

1. #ifndef SALES\_DATA\_H
2. #define SALES\_DATA\_H
3. #include <string>
4. **struct** Sales\_data {
5. std::string bookNo;
6. unsigned units\_sold = 0;
7. **double** revenue = 0.0;
8. };
9. #endif

原理：

第一次包含 **struct** Sales\_data 的时候，#ifndef SALES\_DATA\_H 检查为真，预处理器执行后面的代码直到 #endif ；

后面如果再次遇到 #ifndef SALES\_DATA\_H 检查则为假，则编译器将忽略 #ifndef 到 #endif 之间的内容。