**1. 空指针、NULL指针、零指针**

**1.1什么是空指针常量**

0、0L、”、3 – 3、0 \* 17 （它们都是“integer constant expression”）以及 (void\*)0 （我觉得(void\*)0应该算是一个空指针吧，更恰当一点）等都是空指针常量（注意 (char\*) 0 不叫空指针常量，只是一个空指针值）。至于**系统选取哪种形式作为空指针常量使用，则是实现相关的**。一般的 **C 系统选择 (void\*)0 或者 0** 的居多（也有个别的选择 0L）；至于**C++** 系统，由于存在严格的类型转化的要求，**void\* 不能象 C 中那样自由转换为其它指针类型**，所以**通常选 0 作为空指针常量**（C++标准推荐），而不选择 (void\*)0。

**1.2 什么是空指针**

如果 p 是一个指针变量，则 p = 0; p = 0L; p = ”; p = 3 – 3; p = 0 \* 17; 中的任何一种赋值操作之后（对于 C 来说还可以是 p = (void\*)0;）， p 都成为一个空指针，由系统保证**空指针不指向任何实际的对象或者函数。反过来说，任何对象或者函数的地址都不可能是空指针**。（比如这里的**(void\*)0就是一个空指针**。把它理解为null pointer还是null pointer constant会有微秒的不同，当然也不是紧要了）。其实空指针只是一种编程概念，就如一个容器可能有空和非空两种基本状态。

**1.3 NULL 指针**

NULL 是一个标准规定的宏定义，用来表示空指针常量。因此，除了上面的各种赋值方式之外，还可以用 p = NULL; 来使 p 成为一个空指针。

（很多系统中的实现：#define NULL (void\*)0，与这里的“a null pointer constant”并不是完全一致的）

C++标准库定义的NULL指针



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | // Define   NULL   pointer   value  #ifndef   NULL  #   ifdef   \_\_cplusplus  #     define   NULL      0  #   else  #     define   NULL      ((void   \*)0)  #   endif  #endif //   NULL |

NULL是一个宏，在C++里面被直接被定义成了整数立即数类型的0，而在没有\_\_cplusplus定义的前提下，就被定义成一个值是0的void   \*类型指针常量。

**1.4 零指针**

零值指针，是值为0的指针，可以是任何一种指针类型，可以是通用变体类型void\*，也可以是char\*，int\*等等。

在C++里面，任何一个概念都要以一种语言内存公认的形式表现出来，例如std::vector会提供一个empty()子函数来返回容器是否为空，然而对于一个基本数值类型（或者说只是一个类似整数类型的类型）我们不可能将其抽象成一个类（当然除了auto\_ptr等只能指针）来提供其详细的状态说明，所以我们需要一个特殊值来做为这种状态的表现。  
C++标准规定，当一个指针类型的数值是0时，认为这个指针是空的。（我们在其他的标准下或许可以使用其他的特殊值来定义我们需要的NULL实现，可以是1，可以是2，是随实现要求而定的，但是在标准C++下面我们用0来实现NULL指针）

**1.5 空指针向了内存的什么地方（空指针的内部实现）？**

标准并没有对空指针指向内存中的什么地方这一个问题作出规定，也就是说用哪个具体的地址值（0x0 地址还是某一特定地址）表示空指针取决于系统的实现。我们常见的**空指针一般指向 0 地址，即空指针的内部用全 0 来表示**（zero null pointer，零空指针）；也有一些系统用一些特殊的地址值或者特殊的方式表示空指针（nonzero null pointer，非零空指针），具体请参见C FAQ。

在实际编程中不需要了解在我们的系统上空指针到底是一个 zero null pointer 还是 nonzero null pointer，我们只需要了解一个指针是否是空指针就可以了——编译器会自动实现其中的转换，为我们屏蔽其中的实现细节。注意：不要把空指针的内部表示等同于整数 0 的对象表示——如上所述，有时它们是不同的。

**1.6 对空指针实现的保护政策**

既然我们选择了0作为空的概念，在非法访问空的时候我们需要保护以及报错。因此，编译器和系统提供了很好的政策。

我们程序中的**指针其实是WINDOWS内存段偏移后的地址**，而**不是实际的物理地址**，所以不同的程序中的零值指针指向的同一个0地址，其实在内存中都不是物理内存的开端的0，而是**分段的内存的开端**，这里我们需要简单介绍一下WINDOWS下的内存分配和管理制度：

WINDOWS下，执行文件（PE文件）在被调用后，**系统会分配给它一个额定大小的内存段用于映射这个程序的所有内容**（就是磁盘上的内容）并且**为这个段进行新的偏移计算**，也就是说我们的程序中访问的所有NEAR指针都是在我们“自家”的段里面的，当我们要访问FAR指针的时候，我们其实是跳出了“自家的院子”到了他人的地方，我们需要一个**段偏移地址**来完成新的偏移（人家家里的偏移）所以我们的指针可能是OE02:0045就是告诉系统我们要访问0E02个内存段的0045好偏移，然后WINDOWS会自动给我们找到0E02段的开始偏移，然后为我们计算真实的物理地址。

所以程序A中的零值指针和程序B中的零值指针指向的地方可能是完全不同的。

**保护政策：**

我们的程序在使用的是系统给定的一个段，程序中的**零值指针指向这个段的开端**，为了保证NULL概念，系统为我们**这个段的开头64K内存做了苛刻的规定**，根据虚拟内存访问权限控制，我们程序中（低访问权限）访问要求**高访问权限的这64K内存**被视作是不容许的，所以会必然引发**Access Volitation**错误，而这高权限的64K内存是一块**保留内存**（即**不能被程序动态内存分配器分配，不能被访问，也不能被使用**），就是简单的保留，不作任何使用。

我们在直接定义一个指针后并不知道这个指针指向何处（而不是有些程序员认为的如同JAVA等语言会自动零值初始化），所以我们一旦非法地直接访问这些未知地内容时，极其有可能会触碰到程序所不能触碰地内存（这时类似64K限制地保护政策又会起效，就如同你不仅随意闯入了陌生人的家（野指针），而且拿着刀子要问他要钱（访问），警察（WINDOWS内存访问保护政策）当然请你去警察局（报错）谈谈），所以**养成良好的指针初始化（赋值为NULL）以及使用FREE（或者时DELETE）之后立即再初始化为空是十分必要的！**

**1.7 为什么通过空指针读写的时候就会出现异常？**

NULL指针分配的分区：其范围是从 0x00000000到0x0000FFFF。这段空间是空闲的，对于空闲的空间而言，没有相应的物理存储器与之相对应，所以对这段空间来说，任何读写操作都是会引起异常的。空指针是程序无论在何时都没有物理存储器与之对应的地址。为了保障“无论何时”这个条件，需要人为划分一个空指针的区域，固有上面NULL指针分区。

**1.8 是否可以定义自己的 NULL 的实现？**

NULL 是标准库中的一个reserved identifier （保留标识符）。所以，如果包含了相应的标准头文件而引入了 NULL 的话，则再在程序中重新定义 NULL 为不同的内容是非法的，其行为是未定义的。也就是说，如果是符合标准的程序，其 NULL 的值只能是 0，不可能是除 0 之外的其它值，比如 1、2、3 等。

**1.9 malloc 函数在分配内存失败时返回 0 还是 NULL？**

malloc 函数是标准 C 规定的库函数。在标准中明确规定了在其内存分配失败时返回的是一个 “null pointer”（空指针）。对于空指针值，一般的文档（比如 man）中倾向于用 NULL 表示，而没有直接说成 0。但是我们应该清楚：对于指针类型来说，返回 NULL 和 返回 0 是完全等价的，因为 NULL 和 0 都表示 “null pointer”（空指针）。（一般系统中手册中都返回NULL）

C++里面的NEW再内存失败是会抛出一个BAD\_ALLOC异常。

**2. 野指针**

“野指针”不是NULL指针，是指向“垃圾”内存的指针。

**2.1 “野指针”的成因主要有两种**

1）指针变量没有被初始化。任何指针变量刚被创建时不会自动成为NULL指针，它的缺省值是随机的，它会乱指一气。所以，指针变量在创建的同时应当被初始化，要么将指针设置为NULL，要么让它指向合法的内存。例如：



|  |  |
| --- | --- |
| 1  2 | char \*p = NULL;  char \*str = (char \*) malloc(100); |

2）指针p被free或者delete之后，没有置为NULL，让人误以为p是个合法的指针。

free和delete只是把指针所指的内存给释放掉，但并没有把指针本身干掉。free以后其地址仍然不变（非NULL），只是该地址对应的内存是垃圾，p成了“野指针”。如果此时不把p设置为NULL，会让人误以为p是个合法的指针。如果程序比较长，我们有时记不住p所指的内存是否已经被释放，在继续使用p之前，通常会用语句if (p != NULL)进行防错处理。很遗憾，此时if语句起不到防错作用，因为即便p不是NULL指针，它也不指向合法的内存块。



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | char \*p = (char \*) malloc(100);  strcpy(p, “hello”);  free(p);   // p 所指的内存被释放，但是p所指的地址仍然不变       …  if(p != NULL)      // 没有起到防错作用  {      strcpy(p, “world”);      // 出错  } |

3）指针操作超越了变量的作用范围。这种情况让人防不胜防，示例程序如下：



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | class A  {  public:       void Func(void){ cout << “Func of class A” << endl; }  };  void Test(void)  {      A \*p;     {        A a;        p = &a; // 注意 a 的生命期 ，只在这个程序块中（花括号里面的两行），而不是整个test函数     }       p->Func();  // p是“野指针”  } |

函数Test在执行语句p->Func()时，对象a已经消失，而p是指向a的，所以p就成了“野指针”。