第1章　C:穿越时空的迷雾

作者一开始简单介绍了 C 语言出现的背景，对于先有 C 语言还是先有 UNIX 系统，这就有点像争论先有蛋还是先有鸡。

编译器：

编译器设计者认为： 效率(几乎)就是一切。当然编译器还是关心有意义的错误，良好的文档和产品支持等其他的东西。对于编程器来说，效率是非常重要的。

编译器效率： 它可以分为运行效率和编译效率

运行效率　代码的运行速度　　起决定性作用

编译效率　产生可执行代码的速度

我们时常通过编译优化措施来延长编译的时间，但可以缩短运行的时间。还有一些既可以缩短编译时间，又可以缩短运行时间，比如：清除无用代码，忽略运行时检查等。

许多的 C 语言特性都是为了方便编译器设计者而建立的。

1. 增加类型
   增加类型的目的是为了帮助编译器设计者区分新型机器所拥有的不同数据类型，像 float，double 都是底层硬件支持之后才加入的。
2. 数组下标
   数组下标从 0 开始是因为编译器的设计者根据偏移量的概念来设计产生的。比如：int a[100] 在编译器里面解释 a a[0]都是一个地址 而 a[1]就是 a 地址偏移 int 所占字节数。
3. 数组名当做指针
   把数组当指针运算简化了很多东西，而且在传递数组时我们不需要把整个数组都复制，提高了效率，但是数组和指针并不是都相同，后面会有介绍。
4. auto 关键字
   现在很难看到 auto，那是因为编译器默认变量的内存分配方式就是 auto，但是这种设计是对创建符号表入口的编译器设计者有意义。
5. 不允许嵌套函数
   这样做也是简化编译器，后面有介绍允许时数据结构。
6. register 关键字
   我们现在也一般不用。对于什么变量存在寄存器中，程序员现在都可以不关心，但是在设计这个关键字的时候是为了简化编译器。

C 预处理器：3 个主要的功能

1. 字符串替换　　#define PI 3.14
2. 头文件包含　　采用.h 的头文件包含，但是命名可以随意
3. 通用代码模板的扩展　#define add(x) add_ex(x)

对于宏的使用，适当就可以，在 C++中就不要用宏了。

一个非比寻常的 **Bug**：

B=-3;和 **B= -3;**他们之间容易混淆，所以后面修改成了 **B -= 3;**

但是标准形式也有 **bug**，比如：**exsilon=.0001;** 标准形式 **exsilon.=00001;**

Value=!open 和 **value!=open** 含义不一样。所以用的时候要注意。

可移植性和不可移植性

不可移植的代码： 不同的编译器所采取的行为可能不相同，但他们都是正确的。
如：整型数右移 未确定的 参数求值顺序

坏代码： 在某些不正确的情况下，但标准并未规定应该怎么做
如：一个有符号数溢出

未定义行为： 编译器只在违反语法规则和约束条件时产生错误信息，如果不属于则属于未定义。如： 自己定义一个 **malloc()**函数

可移植代码：

1. 只使用已确定的特性

2. 不突破任何由编译器实现的限制

3. 不产生依赖由编译器定义或未确定的或未定义的特性的输出

如：在程序中输出 **INT_MAX**

保证程序的可移植性是非常重要的，在编码中始终要保证加上必要的类型转换，返回值。

K&R C 和 **ASCII C** 的不同

1. 原型

原型把形参的类型作为函数的声明的一部分。原型主要是为了对函数进行前向声明，编译器在编译时可以对函数调用中的实参和函数声明中的形参进行一致性检查。**K&R C** 中推迟到了链接时，或者干脆不做检查。

如：**char *strcpy(char *,const char *);**

最好写成 **char *strcpy(char *dest,const char *src);**可以给程序员提示。

2. 关键字

ASCII C 增加了 **enum** 关键字 解释了 **const volatile signed void** 不同的含义

除掉了 **entry** 关键字

3. 轻微改变了一些规则。

如： 相邻的字符串字面值会被自动连接在一起。

4. 现实中几乎碰不到的一些区别，这里不再举例

传参相容原理：

这里举个例子来说明问题：

```
#include <stdio.h>
int  foo(const char **p)
{
    return 0;
}
int main(int argc,char **argv)
{
    foo(argv);
    return 0;
}
```

在 GCC 中编译，得到如下信息。但是在 windows 竟然可以顺利编译通过。

```
test.c:11:9: warning: passing argument 1 of ‘foo’ from incompatible
pointer type [enabled by default]
test.c:4:6: note: expected ‘const char **’ but argument is of type ‘char
**
```

其实传递参数和赋值是一样的，那么 const char **和 char **能够赋值么，有人可能认为 const char *和 char *都能赋值，那么这两个肯定能赋值。这种想法是错的。赋值的条件是：两个操作数都是指向有限符或者无限定符的相容类型的指针，左边指针所指向的类型必须具有右边指针所指向类型的全部限定符。

例子： char * 和 const char *

```
char *cp; const char *cpp;
cpp                =            cp; //正确  反过来赋值不行
```

cpp 有 const 限定符　　cp 无限定符

char 和 char 相容且 cpp(左边)所指的类型具有 cp(右边)所指类型的全部限定符

char** 和 const char **

char**指针所指向的类型为 char * 而 const char** 则指向 const char*，所以它们不相容。

但是就这种赋值方式在 windows 的 C++编程器里面竟然合法。

容易混淆的 const

const 不能把变量变成常量。加上 const 只表示不能赋值，这个值只是只读，但是有方法可以修改这个值。

比如： const int limit = 10

而如果我们这样写：

```
const int * limitp = &limit;
int i = 27;
limitp = &i;
```

limitp 本身是可以修改的，可以指向不同的值。

一个 BUG：
关于有符号数和无符号数
程序：

```
#include <stdio.h>
int array[] = {1,2,3,4,5};
#define  TOTAL_EMEMENTS  (sizeof(array)/sizeof(array[0]))
int main()
{
    int d = -1,x=100;
    if ( d <= TOTAL_EMEMENTS -2)
        x = array[d+1];

    printf("%d\n",x);
}
```

看程序，好像没什么错误。但是运行之后就发现不对了。为什么呢？
因为 sizeof 返回的是无符号数，所以 TOTAL_EMEMENTS 返回的是无符号数，if 在判断有符号数和无符号数时。会把 signed int 转换成 unsigned int，这个程序-1 转换成了很大的数，导致 if 语句判断为假。
可以这样修改：if (d <= (int)TOTAL_EMEMENTS -2) 对TOTAL_EMEMENTS进行强制转换。在很多笔试题中也出现这种题目，比如2012年的360校园招聘题目：

```
int foo(int& nParam)
{
    nParam = 10;
    return -1;
}
int main()
{
    int nTom = 0;
    const unsigned int uJerry = 10;
    nTom = foo(nTom);

    while( nTom < uJerry)
    {
        nTom++;
    }
    if ( nTom > uJerry)
        nTom = 100;
    if (nTom < uJerry)
        nTom = 200;
    cout<<nTom<<endl;
    return 0;
}
```

对于无符号数的建议：

1. 尽量不是用无符号数
2. 只有在使用位段和二进制掩码时可以使用无符号数

关于**#pragma** 指示符

用于向编译器提示一些信息，比如特定函数扩展为内联等。不过这个指示符遭到了 **gcc** 编译器设计者的抵制，在 **gcc1.3.4** 中使用它会运行一个叫 **hack** 的游戏。顺便提下**#pragma** 名字的来历，蔡学镛蔡老师的微博有提到，其实 **pragma** 是 **pragmatic, pragmatism** 的词根，都表示行为相关的意思