1. C语言数组

开发中数组是一种常见的数据结构，当然我们知道数组相当于一种容器，但是它不仅仅只能存数值和字符，同时它还可以存放函数的入口地址，以及结构体的数据。

typedef struct \_value

{

int val\_1;

int val\_2;

}VALUE;

typedef struct \_table

{

char index;

char data[100];

int (\*UniSetFunc)(VALUE\*);

}TABLE;

int add(VALUE \*val )

{

int temp = 0;

temp = val->val\_1 + val->val\_2;

return temp;

}

TABLE page[10]

{

{“ First section ”, “abcdefghijklmn”, add},

{“Second section”, “opqrstuvwxyz”, NULL}

};

int main()

{

VALUE AddValue;

AddValue.val\_1 = 2;

AddValue.val\_2 = 4;

int result = 0;

result = page[0]-> UniSetFunc(&AddValue);

printf(“The Result of add is %d\n”, result);

return 0;

}

此时数组就转换为类似于Python语言中的字典的结构，便于后续的开发利用以及追加升级和维护。

代码分析：首先我们知道函数的名字可以做为函数的入口地址（类似于数组的名代表数组的地址一样），所以在TABLE结构体中我们定义了一个成员函数int (\*UniSetFunc)(VALUE\*); 此处UniSetFunc作为函数的入口参数，VALUE\*代表函数的形参类型；TABLE类型的数组 page[10]即包含了结构体的数据以及函数的入口地址，可以通过调用page[0]-> UniSetFunc(&AddValue)来间接地调用add函数并实现AddValue中AddValue.val\_1和AddValue.val\_1两个数求和的运算。

内存命中率问题：

为了可以提高代码运行效率，要才充分的利用内存空间，应连续的访问内存区域。

我们知道数组在内存中存放的位置是一整块的区域，并且是连续存放的，对于定义的数组array[2][2]来说，假设array[0][0]的地址为0x04030,则array[0][1]，array[1][0]，array[1][1] 的地址分别为0x04031, 0x04032, 0x04033；提高内存命中率的即是应该尽可能的连续的访问内存空间区域，而非跳跃式的访问；接下来让我们来看一个矩阵相乘的问题。

for (int i = 0; i < 2; ++i)

{

for (int j = 0; j < 2; ++j)

{

for (int k = 0; k < 3; ++k)

{

matrix[i][j] += matrix1[i][k] \* matrix2[k][j];

}

}

}

以上代码是常用的将矩阵matrix1与matrix2相乘然后赋值给matrix的方法，即用matrix1矩阵得到行向量乘以矩阵matrix2的列向量，然后赋值给matrix，这样由于矩阵在内存存储的结构，我们可以清楚的知道访问matrix2的时候并非采用连续的访问方式，故内存的命中率较低。接下来我们看一种高内存命中率的方法。

for (int i = 0; i < 2; ++i)

{

for (int k = 0; k < 3; ++k)

{

for (int j = 0; j < 2; ++j)

{

matrix[i][j] += matrix1[i][k] \* matrix2[k][j];

}

}

}

可以看出代码仅仅将第二个for循环与第三个for循环交换了位置，而其他的部分没有任何变化，然而内存的命中率却大大的提高了，我们采用将matrix1与matrix2矩阵内部各原素依次相乘然后再累加的方式，来进行矩阵相乘的目的，这样在访问matrix1与matrix2矩阵时没有发生任何内存未命中的问题，从而提高了内存命中的概率。

volatile，const以及static之间的关系：

const关键字为常量关键字，它作用的量为常量，不允许程序去改变该常量的值，如const int value = 12；此常量value值不允许程序将其改变，在开发的过程const关键字会经常用到，为了防止程序意外的改变某一固定的常量，我们应及时的给其加上const关键字；另外const关键字作用于常量时必须直接给常量初始化，因为在整个程序运行大的过程中不允许对其改变，故必须立即初始化，例如：const int value = 12 是正确的，而const int value; value = 12;这样的语法是错误的! 接下来我们来研究一个稍微难一点的问题，即常量指针与指针常量。先看一段代码：

#define SWITCH 1

int main()

{

int val\_1 = 5;

int val\_2 = 10;

const int \*p1 = &val\_1;

int const \*p2 = &val\_1;

int \*const p3 = &val\_1;

#ifdef SWITCH // This is a switch

\*p1 = 20;

\*p2 = 21;

\*p3 = 22;

#endif

#ifndef SWITCH

p1 = &val\_2;

p2 = &val\_2;

p3 = &val\_2;

#endif

printf("%d\n", \*p1);

printf("%d\n", \*p2);

printf("%d\n", \*p3);

return 0;

}

在cygwin编译器下执行，我们可以看到这样的错误：



从图中我们可以清楚的看到，指针p1与p2仅能读取val\_1中的值为指针常量，即不能改变它所指的变量的内容，所以\*p1 = 20; \*p2 = 21;两条命令是错误的！(#ifdef SWITCH … #endif 为条件编译即为宏开关)。然后我们将#define SWITCH 1 语句给注释掉，此时将运行第二块代码，得到结果如下：



从错误中可以看出p3为常量指针，它只能指向一个固定的地址，而不能改变它所指的方向，故p3 = &val\_2;的操作是错误的，因此正确的代码如下：

int main()

{

int val\_1 = 5;

int val\_2 = 10;

const int \*p1 = &val\_1;

int const \*p2 = &val\_1;

int \*const p3 = &val\_1;

printf("Frist\n");

printf("%d\n", \*p1);

printf("%d\n", \*p2);

printf("%d\n", \*p3);

p1 = &val\_2;

p2 = &val\_2;

\*p3 = 22;

printf("Second\n");

printf("%d\n", \*p1);

printf("%d\n", \*p2);

printf("%d\n", \*p3);

return 0;

}

运行的结果为：



最后终结：常量指针（const int \*p或int const \*p）表示指针p不能改变它所指向地址里面所存的值，而可以改变它所指向的地址；指针常量（int \*const p）表示指针p不能改变它所指向的地址，即指针不能改变它所指向的位置，但是可以改变它所指的位置中的内容。若想要指针既不能改变所指向的位置，又不能改变该处的内容，那么可以这样定义：

const int \* const p = &a;或int const \*const p = &a;　在定义函数的时候,若该入口参数在程序执行的过程中不希望被改变,则一定要将该形参用const来修饰,一来这样可以防止该段程序将其改变,二来对于形参而言,一个无论是否是const修饰的实参都可以将其传入const形的形参,而一个const形的实参是无法传入非const形的形参中，所以为了使编译不出错在定义函数的时候，一定要将不希望被改变的量用const关键字来修饰。

Static关键字为静态关键字，它的作用是将作用的变量存入内存中，而非存入寄存器中（即将变量存入堆中而非栈中），并且该作用的变量仅保存最近一次获取的值。接下来我们来看一段代码。

void countfun ()

{

static int count = 0;

++count;

printf(“This is %d number, enter into this function !\n”, count );

}

int main()

{

for (int i = 0; i < 5; ++i)

{

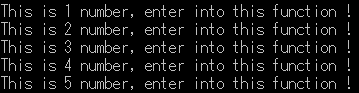
countfun();

}

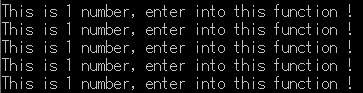
return 0;

}

这段代码的运行结果如下：



而若将除去static关键字，则运行的结果如下：



由此我们可以清楚的看出，static作用的变量count只会存入当前的结果，因此循环调用countfun( )函数的时候并没有从新将count变量置为0，而是保存了前一次的值。

Static关键字在项目中的应用是很广泛的，它不仅仅有上述所介绍的特点，同时若想要定义的全局变量在整个工程中仅在当前.C文件中有效时，也应该将这个全局变量用static来修饰，这样在其他的文件中是无法访问这个变量，从而降低了模块间的耦合度，提高了模块的内聚性，防止其他文件将其改变，从而更加的安全。

volatile关键字在嵌入式领域中是十分重要的一个关键字，尤其是在与硬件相关或多线程的编程中更为重要。volatile关键字修饰的变量说明它是可以随时发生改变的，我们不希望编译器去优化某些代码的时候，需要将这个变量用volatile关键字来修饰，从而程序每次访问该变量的时候是直接从内存中提取出来，而不是从临时的寄存器中将该变量的副本给提取出来利用！例如当我们想要实现某个中断处理时，其用来做判断条件的标记位则应该用volatile来修饰，这样当这个中断在别的地方被触发的时候就可以被实时的检测到，不至于由于优化而忽略中断。接下来我们看一段代码：

int main（）

{

volatile int i = 10;

int a = i;

printf(“i = %d\n”, a);

\_\_asm

{

mov dword ptr[ebp-4], 0x10

}

int b = i;

printf(“i = %d\n”, b);

return 0;

}

此程序输出结果为i = 10;i = 16; 若将volatile关键字去掉，则结果为i = 10;i = 10;即不加关键字会将汇编代码忽略掉，所以为了防止代码优化以及可以及时检测到外部程序对该变量的改变，我们必须将该变量加上volatile关键字。我们知道volatile关键字表征该量是易变的，const关键字代表该量是常量不能改变，那么volatile与const是否可以一起修饰同一个量呢，答案是肯定的，例如在硬件编程中ROM所存储的数据是不允许用户改变的，即指向该数据的指针必须为常量指针（const int \*p = &ram\_data），然而开发商却可以将其意外的改变，为了防止ROM的内容被意外的改变时，而用户程序没有及时的发现，必须将该量用volatile修饰，所以应这样定义该指针（volatile const int \*p = &rom\_data）。

**位运算**

在数字解码与编码的过程中，位运算的操作是司空见惯的事，同时位运算在提高程序的性能方面也独占鳌头，因此位运算操作是必需要深入了解的问题。

在乘法以及除法的操作中我可以使用未运行来提高代码的质量，例如：a = a \* 16;这种操作完全可以替换为:a = a << 4;我们知道左移一位相当于将原数乘以2,左移N位则相当于乘以2^N,前提是在没有发生溢出的情况下；故上例即相当于将数a左移4位，对于某些乘以非2的整数幂情况，如 a = a \* 9;则可以改写为a = (a << 3) + a; 同理右移相当于除以2的整数幂，当然以上所有情况都是在没有发生数据溢出的情况下，因此位运算操作要格外的小心，否则极有可能发生出错的情况。

在数据类型转换的过程中也需要做位运算操作，例如我们想将一个unsigned short类型的数据存入unsigned char类型的数组中，就需要进行位运算，首先分析知道unsigned short占用16个字节，unsigned char占用8个字节，想要将大字节的数据存入小字节，必须要对大字节进行分割，即将高8位与低8为分离开来分别存放，来看实现代码：

unsigned char \* DrawCompo\_Pj\_BT\_Change(unsigned short \*subarray)

{

unsigned char temp[500];

(void)\_sys\_memset(&temp, 0x00, sizeof(temp) );

unsigned short i = 0;

while (subarray[i] != 0x0000)

{

if( (subarray[i] & 0xff00) == 0x0000)

{

temp[i++] = (unsigned char)(subarray[i] & 0x00ff);

}

else

{

temp[i] = (unsigned char)( (subarray[i] & 0xff00) >> 8);

temp[i++] = (unsigned char)(subarray[i] & 0x00ff);

}

}

temp[i] = '\0';

return temp;

}

temp[i] = (unsigned char)( (subarray[i] & 0xff00) >> 8);即取subarray[i]数据的高8位，temp[i++] = (unsigned char)(subarray[i] & 0x00ff);取低8位。这样就可以实现将高字节的数据完整的存入到低字节中。

位运算还可以用来判断变量的符号,我们知道对于一个有符号的变量,其最高位为其符号位,故检查改变的最高位即可知道该变量为正还是为负。看一段代码：

int main()

{

short test\_data = -12;

if (test\_data & 0xF000)

{

printf("This number is negative ");

}

else

{

printf("This number is positive ");

}

return 0;

}

对于想要交换两个数的值的时候，通常我们的做法如下：

void swap（int &data1, int &data2）

{

int temp = 0;

temp = data1;

data1 = data2;

data2 = temp;

}

这样的代码比较简单易懂，然而美中不足的是它会产生一个临时变量temp,接下来我们用位运算来重写这个程序；

void swap（int &data1, int &data2）

{

data1 = data1 ^ data2;

data2 = data1 ^ data2;

data1 = data1 ^ data2;

}

从上面的代码我们可以看出少了一个临时变量，同时也加快了代码的运行效率。

尾递归:

递归调用给我们带来了很多方便,也简化了代码,使程序看起来更加的简洁和明了,但递归调用也通常伴随着一个潜在的危险:出栈，接下来我们来看一个常见的递归。

int factorial(int n)

{

if (n < 1)

{

return 1;

}

else

{

return factorial(n-1)\*n;

}

}

通常在求一个数的阶乘的时候我们习惯于采用上述方法，然而分析来看当输入的n值较大时，factorial(n-1)\*n所计算的值会不断的压入堆栈，生成很多的临时变量，等待下一个的值的确定才得以计算，然而在内存中堆栈的大小是固定的，当输入的n值很大时，极有可能产生堆栈溢出！因此，有一个好的方法解决这种问题即尾递归调用。接下来我们来看这种强大的算法。

int factorial(int n, int m)

{

if (n < 2)

{

return m;

}

else

{

factorial(n-1, n\*m);

}

}

从代码中可以看出，通过引入一个新的参数m来存放每次递归所产生的值，这样就避免了每次递归都要进行压栈操作，也就不会产生堆栈溢出的现象，而且普通的递归每次递归只能等待下一次递归得到的结果后才能继续运算，而尾递归每次执行都会进行运算，一次循环执行完毕即可得到结果，其时间复杂度为O(n);