01.03 字符串转换成整数

题目描述

输入一个由数字组成的字符串，把它转换成整数并输出。例如：输入字符串"123"，输出整数123。

给定函数原型`int StrToInt(const char *str)`，实现字符串转换成整数的功能，不能使用库函数atoi。

分析与解法

本题考查的实际上就是字符串转换成整数的问题，或者说是要你自行实现atoi函数。那如何实现把表示整数的字符串正确地转换成整数呢？以"123"作为例子：

- 当我们扫描到字符串的第一个字符'1'时，由于我们知道这是第一位，所以得到数字1。
- 当扫描到第二个数字'2'时，而之前我们知道前面有一个1，所以便在后面加上一个数字2，那前面的1相当于10，因此得到数字：1*10+2=12。
- 继续扫描到字符'3'，'3'的前面已经有了12，由于前面的12相当于120，加上后面扫描到的3，最终得到的数是：12*10+3=123。

因此，此题的基本思路便是：从左至右扫描字符串，把之前得到的数字乘以10，再加上当前字符表示的数字。

思路有了，你可能不假思索，写下如下代码：

```
int StrToInt(const char *str)
{
        int n = 0;
        while (*str != 0)
        {
                int c = *str - '0';
                n = n * 10 + c;
                ++str;
        }
        return n;
}
```

显然，上述代码忽略了以下细节：

1. 空指针输入：输入的是指针，在访问空指针时程序会崩溃，因此在使用指针之前需要先判断指针是否为空。
2. 正负符号：整数不仅包含数字，还有可能是以'+'或'-'开头表示正负整数，因此如果第一个字符是'-'号，则要把得到的整数转换成负整数。
3. 非法字符：输入的字符串中可能含有不是数字的字符。因此，每当碰到这些非法的字符，程序应停止转换。
4. 整型溢出：输入的数字是以字符串的形式输入，因此输入一个很长的字符串将可能导致溢出。

上述其它问题比较好处理，但溢出问题比较麻烦，所以咱们来重点看下溢出问题。

一般说来，当发生溢出时，取最大或最小的int值。即大于正整数能表示的范围时返回MAX_INT：2147483647；小于负整数能表示的范围时返回MIN_INT：-2147483648。

我们先设置一些变量：

- sign用来处理数字的正负，当为正时sign > 0，当为负时sign < 0
- n存放最终转换后的结果
- c表示当前数字

而后，你可能会编写如下代码段处理溢出问题：

```
//当发生正溢出时，返回INT_MAX
if ((sign == '+') && (c > MAX_INT - n * 10))
{

        n = MAX_INT;
        break;
}
//发生负溢出时，返回INT_MIN
else if ((sign == '-') && (c - 1 > MAX_INT - n * 10))
{
        n = MIN_INT;
        break;
}
```

但当上述代码转换" 10522545459"会出错，因为正常的话理应得到MAX_INT：2147483647，但程序运行结果将会是：1932610867。

为什么呢？因为当给定字符串" 10522545459"时，而MAX_INT是2147483647，即MAX_INT(2147483647) < n*10(1052254545*10)，所以当扫描到最后一个字符‘9’的时候，执行上面的这行代码：

```
c > MAX_INT - n * 10
```

已无意义，因为此时(MAX_INT - n * 10)已经小于0，程序已经出错。

针对这种由于输入了一个很大的数字转换之后会超过能够表示的最大的整数而导致的溢出情况，我们有两种处理方式可以选择：

- 一个取巧的方式是把转换后返回的值n定义成long long，即long long n；
- 另外一种则是只比较n和MAX_INT / 10的大小，即：
- 若n > MAX_INT / 10，那么说明最后一步转换时，n*10必定大于MAX_INT，所以在得知n > MAX_INT / 10时，当即返回MAX_INT。
- 若n == MAX_INT / 10时，那么比较最后一个数字c跟MAX_INT % 10的大小，即如果n == MAX_INT / 10且c > MAX_INT % 10，则照样返回MAX_INT。

对于上面第一种方式，虽然我们把n定义了长整型，但最后返回时系统会自动转换成整型。咱们下面主要来看第二种处理方式。

对于上面第二种方式，先举两个例子说明下：

- 如果我们要转换的字符串是"2147483697"，那么当我扫描到字符'9'时，判断出214748369 > MAX_INT / 10 = 2147483647 / 10 = 214748364（C语言里，整数相除自动取整，不留小数），则返回MAX_INT；
- 如果我们要转换的字符串是"2147483648"，那么判断最后一个字符'8'所代表的数字8与MAX_INT % 10 = 7的大小，前者大，依然返回MAX_INT。

一直以来，我们努力的目的归根结底是为了更好的处理溢出，但上述第二种处理方式考虑到直接计算n * 10 + c可能会大于MAX_INT导致溢出，那么便两边同时除以10，只比较n和MAX_INT / 10的大小，从而巧妙的规避了计算n*10这一乘法步骤，转换成计算除法MAX_INT/10代替，不能不说此法颇妙。

如此我们可以写出正确的处理溢出的代码：

```
c = *str - '0';
if (sign > 0 && (n > MAX_INT / 10 || (n == MAX_INT / 10 && c > MAX_INT % 10)))
{
    n = MAX_INT;
    break;
}
else if (sign < 0 && (n > (unsigned)MIN_INT / 10 || (n == (unsigned)MIN_INT / 10
&& c > (unsigned)MIN_INT % 10)))
{
    n = MIN_INT;
    break;
}
```

从而，字符串转换成整数，完整的参考代码为：

```
int StrToInt(const char* str)
{
        static const int MAX_INT = (int)((unsigned)~0 >> 1);
        static const int MIN_INT = -(int)((unsigned)~0 >> 1) - 1;
        unsigned int n = 0;

        //判断是否输入为空
        if (str == 0)
        {
                return 0;
        }

        //处理空格
        while (isspace(*str))
                ++str;

        //处理正负
        int sign = 1;
        if (*str == '+' || *str == '-')
        {
                if (*str == '-')
                        sign = -1;
```

```
                ++str;
        }

        //确定是数字后才执行循环
        while (isdigit(*str))
        {
                //处理溢出
                int c = *str - '0';
                if (sign > 0 && (n > MAX_INT / 10 || (n == MAX_INT / 10 && c >
MAX_INT % 10)))
                {
                        n = MAX_INT;
                        break;
                }
                else if (sign < 0 && (n >(unsigned)MIN_INT / 10 || (n ==
(unsigned)MIN_INT / 10 && c > (unsigned)MIN_INT % 10)))
                {
                        n = MIN_INT;
                        break;
                }

                //把之前得到的数字乘以10，再加上当前字符表示的数字。
                n = n * 10 + c;
                ++str;
        }
        return sign > 0 ? n : -n;
}
```

举一反三

1. 实现string到double的转换

分析：此题虽然类似于atoi函数，但毕竟double为64位，而且支持小数，因而边界条件更加严格，写代码时需要更加注意。