



第 1 章 字符串处理

字符串几乎在所有程序设计语言中都占据了重要地位。在 C 语言中，C 语言库函数提供了一系列 API 处理各种字符串问题。而在 C++ 中，可以使用类 `std::string`、`std::istringstream`、`std::ostringstream`，以及若干 STL 算法解决各种字符串问题。



1.1 字符串基本操作

1.1.1 字符串复制

C 语言中可以使用 `strcpy` 函数实现字符串复制，`strcpy` 函数声明如下：

```
char * strcpy ( char * destination, const char * source );
```

`strcpy` 会将 `source` 指向的字符串内容连同末尾的 `null` 字符一起复制到 `destination` 指向的内存空间。为了避免溢出，`destination` 指向的字符串空间必须能容纳 `source` 中所有字符(包括 `null` 字符)，并且 `source` 和 `destination` 在内存上不能存在重叠现象。函数将返回 `destination` 指针。

例如，代码如下：

```
#include <stdio.h>
#include <string.h>

int main ()
{
    char str1[]="Test string";
    char str2[40];
    char str3[40];
    strcpy (str2,str1);
    strcpy (str3,"copy successful");
    printf ("str1: %s\nstr2: %s\nstr3: %s\n",str1,str2,str3);
    return 0;
}
```

输出：

```
str1: Test string
str2: Test string
str3: copy successful
```

一些公司在面试或笔试时喜欢直接使用 `strcpy` 的内部实现作为题目，以下是一个较为高效的实现过程，代码如下：

```
char *strcpy(char *dest, const char *src)
{
    char *save = dest;
    while(*dest++ = *src++);
    return save;
}
```

除了 `strcpy` 外，C 语言库函数中还提供了 `strncpy` 函数用于复制指定数目的字符到目标区域。`strncpy` 函数声明如下：

```
char * strncpy ( char * destination, const char * source, size_t num );
```

`strncpy` 从 `source` 指向的空间中复制前 `num` 个字符到 `destination` 指向的空间中。如果还没有复制完 `num` 个字符时，`source` 已经达到了字符串尾部，那么函数将补充复制若干个 `null` 字符到 `destination` 指向的空间中，直到复制 `num` 个字符为止。

如果这是第一次碰到 `size_t` 类型，请别慌张，`size_t` 是无符号整型，等价于 `unsigned int`。后续很多函数都会涉及此类型。

`strncpy` 例子，代码如下：

```
#include <stdio.h>
#include <string.h>

int main ()
{
    char str1[] = "To be or not to be";
    char str2[40];
    char str3[40];

    /* copy to sized buffer (overflow safe): */
    strncpy ( str2, str1, sizeof(str2) );

    /* partial copy (only 5 chars): */
    strncpy ( str3, str2, 5 );
    str3[5] = '\0';    /* null character manually added */

    puts (str1);
    puts (str2);
    puts (str3);

    return 0;
}
```

输出：

```
To be or not to be
To be or not to be
To be
```

需要指出的是，在使用 `strncpy` 函数时，通常需要自行添加 `null` 终结符到 `destination` 中。

在 C++ 语言中可以直接使用 `std::string` 类处理各种字符串问题。有三种方法可以对字符串复制：`std::string` 的构造函数，操作符重载函数 `operator=`，以及成员函数 `assign`。

第一种方法：

```
char str1[] = "To be or not to be";
std::string str2 = str1;
std::string str3;
str3 = str1;
```

第二种方法：

```
char str1[] = "To be or not to be";
std::string str2;
str2.assign(str1, str1+strlen(str1));
```

如果只想复制若干个字符，还可以有以下第三种方法：

```
char str1[] = "To be or not to be";
std::string str2;
str2.assign(str1, str1+2);
```

`assign` 是 `std::string` 的类成员函数，三个重载函数声明如下：

```
string& assign (const char* s);
string& assign (const char* s, size_t n);
template <class InputIterator>
string& assign (InputIterator first, InputIterator last);
```

1.1.2 字符串连接

C 语言中可以使用 `strcat` 函数实现字符串的连接，也就是将一个字符串复制到另一个字符串的尾部，`strcat` 函数声明如下：

```
char * strcat ( char * destination, const char * source );
```

`strcpy` 会将 `source` 指向的字符串内容连同末尾的 `null` 字符一起复制到 `destination` 末端的内存空间中。在连接开始时，`source` 的第一个字符将会覆盖 `destination` 末端的 `null` 字符，在连接结束时，`source` 最后的 `null` 字符将会复制到 `destination` 的最后。

`destination` 指向的字符串空间必须能容纳 `source`，以及 `destination` 中所有字符数之和（包括 `null` 字符），并且 `source` 和 `destination` 在内存上不能存在重叠现象。函数将返回 `destination` 指针。

例如，代码如下：

```
#include <stdio.h>
#include <string.h>

int main ()
{
    char str[80];
    strcpy (str,"these ");
    strcat (str,"strings ");
    strcat (str,"are ");
    strcat (str,"concatenated.");
    puts (str);
    return 0;
}
```

输出：

```
these strings are concatenated.
```

一些公司在面试或笔试时也喜欢直接使用 `strcat` 的内部实现作为题目，以下是一个较为高效的实现过程，代码如下：

```
char *strcat(char *dest, const char *src)
{
    char *save = dest;
    while (*dest)
        dest++;
    while (*dest++ = *src++);
    return save;
}
```

在 C++ 中，可以直接使用 `std::string` 类成员函数 `append` 或者直接调用操作符重载的 “+=” 符号函数用于字符串的连接。

例如，使用 `append` 成员函数，代码如下：

```
#include <iostream>
#include <string>

int main ()
{
    std::string str;
    std::string str2="Writing ";
    std::string str3="print 10 and then 5 more";

    // used in the same order as described above:
    str.append(str2);                // "Writing "
    str.append(str3,6,3);            // "10 "
    str.append("dots are cool",5);   // "dots "
    str.append("here: ");            // "here: "
```

```

str.append(10u, '.');           // "....."
str.append(str3.begin()+8, str3.end()); // " and then 5 more"
str.append<int>(5, 0x2E);       // "....."

std::cout << str << '\n';
return 0;
}

```

输出：

```
Writing 10 dots here: ... and then 5 more...
```

或者使用 `operator +=` 成员函数，代码如下：

```

#include <iostream>
#include <string>

int main ()
{
    std::string name ("John");
    std::string family ("Smith");
    name += " K. ";           // c-string
    name += family;           // string
    name += '\n';             // character

    std::cout << name;
    return 0;
}

```

输出：

```
John K. Smith
```

当然，还可以使用全局的 `operator +` 函数来处理 `std::string` 的连接，代码如下：

```

#include <iostream>
#include <string>

int main ()
{
    std::string firstlevel ("com");
    std::string secondlevel ("google");
    std::string scheme ("http://");
    std::string hostname;
    std::string url;

    hostname = "www." + secondlevel + '.' + firstlevel;
    url = scheme + hostname;
    std::cout << url << '\n';
}

```

```
    return 0;
}
```

输出:

```
http://www.google.com
```

1.1.3 反转字符串

反转字符串也是一些程序和算法需要的功能。在 Windows 下使用 C 时, 可以使用 `strrev` 反转字符串, `strrev` 函数声明如下:

```
char * strrev( char *str );
```

例如, 代码如下:

```
#include<stdio.h>
#include<string.h>
int main()
{
    char a[] = "my name";
    strrev(a);
    puts(a);
    return 0;
}
```

输出:

```
eman ym
```

需要说明的是, Linux 系统库中并没有包含这个函数, 所以必须自己定义反转字符串函数。`strrev` 的一个简单的实现过程如下, 代码如下:

```
#include <string.h>
char *strrev(char *str)
{
    int i = strlen(str) - 1, j = 0;
    char ch;
    while (i > j)
    {
        ch = str[i];
        str[i] = str[j];
        str[j] = ch;
        i--;
        j++;
    }
    return str;
}
```

利用指针和异或运算，可以高效实现字符串的反转，代码如下：

```
void strrev(char *p)
{
    char *q = p;
    while(q && *q) ++q;
    for(--q; p < q; ++p, --q)
        *p = *p ^ *q,
        *q = *p ^ *q,
        *p = *p ^ *q;
}
```

在 C++ 中，则可以直接使用 STL 算法库中的 `std::reverse` 函数对字符串或者 `std::string` 对象进行反转。例如，代码如下：

```
#include <iostream>
#include <string> //for std::string
#include <algorithm> //for std::reverse
#include <cstring> //for strlen
using namespace std;

int main()
{
    std::string s = "Hello world!";
    std::reverse(s.begin(), s.end());
    std::cout << s << endl;

    char str[] = "get ready!";
    std::reverse(str, str + strlen(str));
    cout << str << endl;
    return 0;
}
```

输出：

```
!dlrow olleH
!ydaer teg
```

从上例中可以看出，`std::reverse` 不仅可以对 `std::string` 对象进行反转，也可以对普通字符串进行反转。`std::reverse` 是 STL 算法函数之一，所以必须引入头文件 `<algorithm>`。

请大家检查如下字符串反转代码，程序运行时会崩溃，代码如下：

```
#include <iostream>
#include <algorithm> //for std::reverse
#include <cstring> //for strlen
using namespace std;

int main()
```

```
{
    char *str = "get ready!";
    std::reverse(str, str+strlen(str));
    cout << str << endl;
    return 0;
}
```

原因是 `str` 是一个字符指针，它指向的是一个字符串常量，而常量是无法修改的。

1.1.4 大小写转换

在 C 语言的库函数中，并没有提供对字符串大小写转换的函数。但库函数提供了 `toupper` 和 `tolower` 函数用于将单个字符的大小写转换。`toupper` 将字符转换成大写，`tolower` 将字符转换成小写。函数声明如下：

```
int toupper ( int c );
int tolower ( int c );
```

如果传入的参数 `c` 没有对应的大小写字符，那么返回值就是传入的 `c` 值。另外，这两个库函数声明在 `<ctype.h>` 中而不是在 `<string.h>` 中。

例如，代码如下：

8

```
#include <stdio.h>
#include <ctype.h>
int main ()
{
    int i=0;
    char str[]="Test String.\n";
    char c;
    while (str[i])
    {
        c=str[i];
        putchar (tolower(c));
        i++;
    }
    return 0;
}
```

输出：

```
test string.
```

可以利用字符大小写转换的函数，实现字符串的大小写转换，一个基本的小写转换方法如下，代码如下：

```
char* ToLower(char* str)
{
    int i;
```



```

    if(!str) return NULL;
    for(i = 0; str[i]; ++i){
        str[i] = tolower(str[i]);
    }
    return str;
}

```

利用指针进行大写转换的方法如下，代码如下：

```

char* ToUpper(char *str)
{
    char* ostr = str;
    if(!str) return NULL;
    for ( ; *str; ++str) *str = toupper(*str);
    return ostr;
}

```

在 C++中，可以直接使用 STL 算法库中的 `std::transform` 函数对字符串或者 `std::string` 对象进行大小写转换。代码如下：

```

#include <iostream>
#include <algorithm> //for std::transform
#include <cstring> //for strlen
#include <string> //for std::string

using namespace std;
int main()
{
    std::string str = "Hello World";
    std::transform(str.begin(), str.end(), str.begin(), ::toupper);
    cout<<str<<endl;

    char str2[] = "My Name";
    std::transform(str2, str2+strlen(str2), str2, ::tolower);
    cout<<str2<<endl;
}

```

输出：

```

HELLO WORLD
my name

```

这里 `std::transform` 函数的声明如下：

```

template <class InputIterator, class OutputIterator, class UnaryOperation>
OutputIterator transform (InputIterator first1, InputIterator last1,
                        OutputIterator result, UnaryOperation op);

```

其中，第一参数 `first1` 和第二参数 `last1` 比较容易理解，它们代表输入的起始和终止迭代

器位置。而第三个参数是进行变换后数据要输出的迭代器位置。如果想把这个变换函数应用到容器自身，那么只要将 `result` 的值设置成和 `first1` 一样就可以。最后一个参数是一元变换函数。

如果这是你第一次接触迭代器，那么不用慌张，可以把它想象成指针。后面章节中还会出现各种迭代器的用法。

1.1.5 字符串与数的转换

首先，学习 C 语言中字符串与数字的互换。在 C 语言中，`sprintf` 函数可以将任何一个或多个数字格式化成指定格式的字符串。`sprintf` 函数声明如下：

```
int sprintf ( char * str, const char * format, ... );
```

和 `printf` 函数相比，它多了第一个参数 `str`。这是因为 `printf` 函数只需要将格式化的内容输出到 `console` 窗口，也就是屏幕。而 `sprintf` 需要将格式化的内容输出到缓冲区，第一个参数 `str` 就是这个缓冲区内内存。因为是输出到缓冲区，所以需要自行确保 `str` 的内存空间足够容纳格式化的内容，否则存在内存溢出风险。最后需要指出的是，一个 `null` 字符会被自动添加到格式化内容的最后位置。

例如，代码如下：

```
#include <stdio.h>

int main ()
{
    char buffer [64];
    int n, a=6, b=8;
    n=sprintf(buffer, "%d plus %d is %d", a, b, a+b);
    printf("[%s] is a string %d chars long\n",buffer,n);
    return 0;
}
```

输出：

```
[6 plus 8 is 14] is a string 14 chars long
```

和 `printf` 一样，如果想以指定精度输出浮点数，参照如下代码：

```
double a = 3.1415926;
sprintf(buffer, "%.2f", a);
```

输出：

```
3.14
```

在 C 语言中，将字符串变成一个整数，可以使用 `atoi` 函数，需要引入头文件 `<stdlib.h>`。`atoi` 函数声明如下：

```
int atoi (const char * str);
```

例如，代码如下：

```
#include <stdio.h>      /* printf, fgets */
#include <stdlib.h>      /* atoi */

int main ()
{
    int i;
    char buffer[256];
    printf ("Enter a number: ");
    fgets (buffer, 256, stdin);
    i = atoi (buffer);
    printf ("The value entered is %d.",i);
    return 0;
}
```

输出：

```
Enter a number: 64
The value entered is 64.
```

将字符串变成一个浮点数，可以使用 `atof` 函数。`atof` 函数声明如下：

```
double atof (const char* str);
```

例如，代码如下：

```
#include <stdio.h>      /* printf, fgets */
#include <stdlib.h>      /* atof */
#include <math.h>        /* sin */

int main ()
{
    double n,m;
    double pi=3.1415926535;
    char buffer[256];
    printf ("Enter degrees: ");
    fgets (buffer,256,stdin);
    n = atof (buffer);
    m = sin (n*pi/180);
    printf ("The sine of %f degrees is %f\n" , n, m);
    return 0;
}
```

输出：

```
Enter degrees: 45
The sine of 45.000000 degrees is 0.707101
```

相对于 `atoi` 和 `atof`，C 语言还提供了 `sscanf` 函数用于从格式化的字符串中提取多个类型的数据。从名字上可以看出 `sscanf` 与 `scanf` 的作用类似，区别在于 `scanf` 是从用户键盘输入的

字符串中提取数据，而 `sscanf` 是从一个字符串缓冲中提取数据。`sscanf` 函数声明如下：

```
int sscanf ( const char * s, const char * format, ...);
```

例如，代码如下：

```
#include <stdio.h>

int main ()
{
    char str[]="I am 18 years old";
    char subStr[20];
    int n;

    sscanf (str,"%s %s %d",subStr,&n);
    printf ("%s -> %d\n",subStr, n);

    return 0;
}
```

输出：

```
I -> 18
```

12

在 C++ 中，新手通常认为 `std::string` 类应该提供字符串与数字转换的成员函数，但实际上 `std::string` 类并没有提供任何字符串与数字转换的方法，这个任务交给了 `stringstream` 流。

想把字符串转换成数字，使用 `istringstream` 类，需要引入头文件 `<sstream>`。例如，代码如下：

```
#include <iostream>
#include <sstream> //std::istringstream
#include <string> //std::string

using namespace std;
int main ()
{
    char str[]="3.5 + 2.5 = 6";
    std::istringstream iss(str);
    double a, b, c;
    std::string s;

    iss>>a>>s>>b>>s>>c;
    cout<<"a = "<<a<<" , b = "<<b<<" , c = "<<c<<endl;
    return 0;
}
```

输出：

```
a = 3.5, b = 2.5, c = 6
```

想把数字转换成字符串，使用 `ostringstream` 类，同样需要引入头文件 `<sstream>`。例如，代码如下：

```
#include <iostream>
#include <sstream> //std::ostringstream
#include <string>   //std::string

using namespace std;
int main ()
{
    std::ostringstream oss;
    double a=3.5, b=2.5, c;
    c = a+b;
    std::string s;

    oss<<a<<" + "<<b<<" = "<<c;
    s = oss.str();
    cout<<s<<endl;
    return 0;
}
```

输出：

3.5 + 2.5 = 6

13

如果编译器完全支持 C++11 标准，那么还可以使用新 STL 标准的库函数 `std::stoi`, `std::stof` 和 `std::stod` 从 `std::string` 对象获得 `int`, `float` 和 `double` 值。三个函数声明如下：

```
int stoi (const string& str, size_t* idx = 0, int base = 10);
float stof (const string& str, size_t* idx = 0);
double stod (const string& str, size_t* idx = 0);
```

例如，代码如下：

```
#include <iostream> // std::cout
#include <string>    // std::string, std::stoi, std::stod

int main()
{
    std::string str = "1997, my";
    std::string dataStr("3.1415 6.2830");
    size_t sz;
    int year = std::stoi(str);
    double pi = std::stod(dataStr, &sz);
    double pi2 = std::stod(dataStr.substr(sz));
    std::cout << year << " " << pi << " " << pi2<<std::endl;
    return 0;
}
```

输出:

```
1997 3.1415 6.283
```

1.1.6 字符串查找

在 C 语言的库函数中, `strstr` 函数用于从一个字符串中查找一个子串, 函数声明如下:

```
char * strstr ( const char * str1, const char * str2 );
```

函数会返回 `str2` 在 `str1` 中第一次出现的位置。如果 `str1` 中没有 `str2` 出现, 则返回 `null` 指针。例如, 代码如下:

```
#include <stdio.h>
#include <string.h>

int main ()
{
    char str[] = "It is apple tree.";
    char * pch;
    pch = strstr (str, "apple");
    strncpy (pch, "lemon", 5);
    puts (str);
    return 0;
}
```

输出:

```
It is lemon tree.
```

如果只想从字符串中查找某个字符出现的位置, 则可以使用 `strchr` 函数, 声明如下:

```
char * strchr ( const char * str, int c);
```

例如, 代码如下:

```
#include <stdio.h>
#include <string.h>

int main ()
{
    char str[] = "It's a long long story.";
    char * pch;
    printf ("Looking for the 'l' character in \"%s\" ...\\n", str);
    pch=strchr(str,'l');
    while (pch!=NULL)
    {
        printf ("found at %d\\n", pch-str+1);
        pch=strchr(pch+1,'l');
    }
}
```

```

    }
    return 0;
}

```

输出：

```

Looking for the 'l' character in "It's a long long story."...
found at 8
found at 13

```

如果想从字符串中查找任意一个指定的字符出现的位置，则可以使用 `strpbrk` 函数。读者可以自行搜索该函数的用法，这里不再赘述。

在 C++ 中，`std::string` 类成员函数 `find` 可以用来查找指定的字符串或字符，四个 `find` 重载函数声明如下：

```

size_t find (const string& str, size_t pos = 0) const;
size_t find (const char* s, size_t pos = 0) const;
size_t find (const char* s, size_t pos, size_t n) const;
size_t find (char c, size_t pos = 0) const;

```

例如，代码如下：

```

#include <iostream>          // std::cout
#include <string>             // std::string

int main ()
{
    std::string str ("Gather as many roses as you possibly can.");
    std::size_t found = str.find("as");
    while (found!=std::string::npos)
    {
        std::cout << "found at: " << found << "\n";
        found = str.find("as", found+1);
    }
    return 0;
}

```

输出：

```

found at: 7
found at: 21

```

需要指出的是，在 `std::string` 对象中如果没有找到指定的字符或字符串，其返回的 `size_t` 的值是 `std::string::npos`。使用如下的代码，就能明白 `std::string::npos` 的具体值。

```
std::cout<<std::hex<<std::string::npos<<std::endl;
```

答案就是：

```
ffffff
```

也就是`(unsigned int)-1`。

如果想从`std::string` 中查找指定的字符串中任意一个字符出现的位置, 则可以使用成员函数 `find_first_of` 或者 `find_last_of` 函数。读者可以自行搜索该函数的用法。

1.1.7 删除字符

C 语言的库函数中并没有提供删除字符串中指定字符的函数。以下是一个较为高效的字符删除算法, 代码如下:

```
char* remchr(char *str, char c)
{
    char *src, *dst;
    for (src = dst = str; *src != '\0'; src++) {
        *dst = *src;
        if (*dst != c) dst++;
    }
    *dst = '\0';
    return str;
}
```

16 算法的巧妙之处在于 `dst` 偏移量始终不会超过 `src` 指针的偏移量, 所以避免了额外开辟内存空间用于存储新的字符串。利用该方法举例, 代码如下:

```
#include <stdio.h>
char* remchr(char *str, char c)
{
    char *src, *dst;
    for (src = dst = str; *src != '\0'; src++) {
        *dst = *src;
        if (*dst != c) dst++;
    }
    *dst = '\0';
    return str;
}

int main ()
{
    char str[]="test string.\n";
    puts(remchr(remchr(str, 't'), 's'));
    return 0;
}
```

输出:

```
e ring.
```

在 C++ 中, `std::string` 类没有提供去除指定字符的成员函数, 这的确令人沮丧。但是, STL

的算法库提供了一个 `std::remove` 函数，它可以删除两个迭代器区间内的指定值。例如，代码如下：

```
#include <iostream>      // std::cout
#include <string>         // std::string
#include <algorithm>      // std::remove

int main ()
{
    std::string str ("Gather as many roses as you can.");
    std::remove(str.begin(), str.end(), 'a');
    std::cout<<str;
    return 0;
}
```

输出：

```
Gther s mny roses s you cn. can.
```

以上输出结果中，最后五个字符(含一个空格) `can.` 是怎么回事？事实上，`std::remove` 只负责从区间内删除指定的值，但它不负责调整（缩小）容器的容量。此时，`std::remove` 的返回值可以提供一些帮助。它的返回值是一个迭代器，指向新区间的末端位置。因此最终的方法就是结合 `std::remove` 和 `std::string` 类成员函数 `erase`，实现指定字符的删除功能。标准写法如下：

```
str.erase(std::remove(str.begin(), str.end(), 'a'), str.end());
```

例如，代码如下：

```
#include <iostream>      // std::cout
#include <string>         // std::string
#include <algorithm>

int main ()
{
    std::string str ("Gather as many roses as you can.");
    str.erase(std::remove(str.begin(), str.end(), 'a'), str.end());
    std::cout<<str;
    return 0;
}
```

输出：

```
Gther s mny roses s you cn.
```

1.1.8 字符串字典比较

在 C 语言的库函数中，提供了 `strcmp` 函数用于比较两个字符串的字典顺序。函数声明如下：

```
int strcmp ( const char * str1, const char * str2 );
```

strcmp 返回值见表 1-1。

表 1-1 strcmp 返回值

返回值	含义
<0	str1 在字典中排在 str2 的前面
=0	str1 和 str2 完全相同
>0	str1 在字典中排在 str2 的后面

例如，代码如下：

```
#include <stdio.h>
#include <string.h>

int main ()
{
    char key[] = "volleyball";
    char buffer[80];
    do {
        printf ("Guess my favorite sport? ");
        fflush (stdout);
        scanf ("%79s",buffer);
    } while (strcmp (key,buffer) != 0);
    puts ("Correct answer!");
    return 0;
}
```

输出：

```
Guess my favorite sport? football
Guess my favorite sport? basketball
Guess my favorite sport? volleyball
Correct answer!
```

C 语言库函数中还提供了 strcmp 函数用于比较两个字符串前面若干字节的字典顺序。函数声明如下：

```
int strcmp ( const char * str1, const char * str2, size_t num );
```

可以看到，strcmp 比 strcmp 多了最后一个参数，它表示要比较的字节长度。strcmp 用法如下，代码如下：

```
#include <stdio.h>
#include <string.h>

int main ()
{
    char str[][10] = { "Green", "Apple", "Great", "Ten" };
    int n;
    puts ("Looking for string starting with \"Gr\"...");
```

```

    for (n=0 ; n<4 ; n++)
    {
        if (strncmp (str[n],"Gr",2) == 0)
        {
            printf ("found %s\n",str[n]);
        }
    }
    return 0;
}

```

输出:

```

Looking for string starting with "Gr"...
found Green
found Great

```

需要指出的是, `strcmp` 和 `strncmp` 都是对大小写敏感的字符串比较函数, 这也意味 "apple" 和 "Apple" 在字典中不是相同的字符串。如果调用 `strcmp("apple", "Apple")`, 它的返回值将是一个正值, 这时因为在 ASCII 码表中, 字符 'a' 大于 'A'。

那么如何不区分大小写, 进行实际意义的字典序比较呢? C 语言库函数中, 提供了库函数解决这个问题。需要指出的是, 在 Windows、Linux 和 Mac 下, 他们使用了不同的库函数名。在 Windows 下, 这个函数叫作 `stricmp`, 但在 Linux 或者 Mac 下, 它叫作 `strcasecmp`。所以, 如果你是一个跨平台的 C/C++ 程序员, 通常需要预先通过宏定义的方式统一这个函数的调用, 方法如下:

```

#ifdef LINUX || defined _MACX
#define _stricmp strcasecmp
#define stricmp strcasecmp
#endif

```

关于 `strcmp` 函数, 最后还要强调两点:

- (1) `strcmp` 的返回值为 0 才表示两个字符串是完全一样的;
- (2) 为了检测 `str1` 在字典中排在 `str2` 的前面, 不要写成 `strcmp(str1, str2) == -1`, 而应该更稳妥地写成 `strcmp(str1, str2) < 0`。

在 C++ 中, `std::string` 类的字典序比较功能非常强大, 它提供了操作符重载的成员函数 `<`, `<=`, `==`, `!=`, `>`, `>=` 进行字典序比较。举例如下, 代码如下:

```

#include <iostream>
#include <string>

int main ()
{
    std::string a = "alpha";
    std::string b = "beta";

    if (a==b) std::cout << "a and b are equal\n";
    if (a!=b) std::cout << "a and b are not equal\n";
}

```

```

    if (a < b) std::cout << "a is less than b\n";
    if (a > b) std::cout << "a is greater than b\n";
    if (a <= b) std::cout << "a is less than or equal to b\n";
    if (a >= b) std::cout << "a is greater than or equal to b\n";
    return 0;
}

```

输出:

```

a and b are not equal
a is less than b
a is less than or equal to b

```

不过, `std::string` 类中并没有提供不区分大小写的字典序比较函数。所以, 在调用操作符重载的比较函数之前, 可以先调用 `std::transform` 进行小写转换:

```

#include <algorithm>
...
std::transform(a.begin(), a.end(), a.begin(), tolower);
std::transform(b.begin(), b.end(), b.begin(), tolower);

```



1.2 字符串处理常见问题

1.2.1 居民身份证号的表示

中国居民身份证号的存储是程序设计中经常碰到的问题。当今, 中国居民的身份证号码有 15 位和 18 位两种。1985 年我国实行居民身份证制度, 当时签发的身份证号码是 15 位, 而 1999 年签发的身份证由于年份的扩展和末尾加了校验码, 就成了 18 位。我们知道一个 `int` 或者 `long` 类型最多只能表达 20 多亿的数字, 也就是 10 位整数, 这远远不够表达一个身份证号。幸好还有一个 `long long` 类型, 可以通过如下代码知道它的最大值, 代码如下:

```

#include <iostream>
#include <limits>
int main ()
{
    std::cout << std::numeric_limits<long long>::max();
    return 0;
}

```

输出:

```
9223372036854775807
```

这个最大值一共是 19 位, 已经足够容纳中国居民身份证号的 18 位。似乎用 `long long` 问题就能解决。不过可惜的是, 有些人的身份证号的最后一位是字母 X 而不是数字。这是因为, 18 位身份证号的最后一位作为尾号的校验码。它是由号码编制单位按统一的公式计算的, 如