The strchr() Function

The strchr() function finds the first occurrence of a specified character in a string. The prototype is

```
char *strchr(char *str, int ch);
```

The function strchr() searches str from left to right until the character ch is found or the terminating null character is found. If ch is found, a pointer to it is returned. If not, NULL is returned.

When strchr() finds the character, it returns a pointer to that character. Knowing that str is a pointer to the first character in the string, you can obtain the position of the found character by subtracting str from the pointer value returned by strchr(). Following Listing illustrates this. Remember that the first character in a string is at position 0. Like many of C's string functions, strchr() is case-sensitive. For example, it would report that the character F isn't found in the string raffle.

Using strchr() to search a string for a single character.

```
/* Searching for a single character with strchr(). */

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

 main()
 {
     char *loc, buf[80];
     int ch;

    /* Input the string and the character. */

    printf("Enter the string to be searched: ");
    gets(buf);
    printf("Enter the character to search for: ");
    ch = getchar();

    /* Perform the search. */

    loc = strchr(buf, ch);

    if ( loc == NULL )
        printf("The character %c was not found.", ch);
    else
        printf("The character %c was found at position %d.\n",
                ch, loc-buf);
    return(0); }
```

```
Enter the string to be searched: How now Brown Cow?
Enter the character to search for: C
The character C was found at position 14.
```

The strcspn() Function

The library function strcspn() searches one string for the first occurrence of any of the characters in a second string. Its prototype is

```
size_t strcspn(char *str1, char *str2);
```

The function strcspn() starts searching at the first character of str1, looking for any of the individual characters contained in str2. This is important to remember. The function doesn't look for the string str2, but only the characters it contains. If the function finds a match, it returns the offset from the beginning of str1, where the matching character is located. If it finds no match, strcspn() returns the value of strlen(str1). This indicates that the first match was the null character terminating the string

Searching for a set of characters with strcspn().

```
/* Searching with strcspn(). */

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

 main()
 {
     char  buf1[80], buf2[80];
     size_t loc;

    /* Input the strings. */

    printf("Enter the string to be searched: ");
    gets(buf1);
    printf("Enter the string containing target characters: ");
    gets(buf2);

    /* Perform the search. */

    loc = strcspn(buf1, buf2);

    if ( loc ==  strlen(buf1) )
        printf("No match was found.");
    else
        printf("The first match was found at position %d.\n", loc);
    return(0);
}
```

```
Enter the string to be searched: How now Brown Cow?
Enter the string containing target characters: Cat
The first match was found at position 14.
```

The strpbrk() Function

The library function strpbrk() is similar to strcspn(), searching one string for the first occurrence of any character contained in another string. It differs in that it doesn't include the terminating null characters in the search. The function prototype is

```
char *strpbrk(char *str1, char *str2);
```

The function strpbrk() returns a pointer to the first character in str1 that matches any of the characters in str2. If it doesn't find a match, the function returns NULL. As previously explained for the function strchr(), you can obtain the offset of the first match in str1 by subtracting the pointer str1 from the pointer returned by strpbrk() (if it isn't NULL, of course).

The strstr() Function

The final, and perhaps most useful, C string-searching function is strstr(). This function searches for the first occurrence of one string within another, and it searches for the entire string, not for individual characters within the string. Its prototype is

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

The function strstr() returns a pointer to the first occurrence of str2 within str1. If it finds no match, the function returns NULL. If the length of str2 is 0, the function returns str1. When strstr() finds a match, you can obtain the offset of str2 within str1 by pointer subtraction, as explained earlier for strchr(). The matching procedure that strstr() uses is case-sensitive.

Using strstr() to search for one string within another.

```
  /* Searching with strstr(). */

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

  main()
  {
      char *loc, buf1[80], buf2[80];

     /* Input the strings. */

     printf("Enter the string to be searched: ");
     gets(buf1);
     printf("Enter the target string: ");
     gets(buf2);

     /* Perform the search. */

     loc = strstr(buf1, buf2);

     if ( loc ==  NULL )
         printf("No match was found.\n");
     else
         printf("%s was found at position %d.\n", buf2, loc-buf1);
     return(0);}
Enter the string to be searched: How now brown cow?
Enter the target string: cow
Cow was found at position 14.
```

The strrev() Function

The function strrev() reverses the order of all the characters in a string (Not ANSI Standard). Its prototype is

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

The order of all characters in str is reversed, with the terminating null character remaining at the end.

String-to-Number Conversions

Sometimes you will need to convert the string representation of a number to an actual numeric variable. For example, the string "123" can be converted to a type int variable with the value 123. Three functions can be used to convert a string to a number. They are explained in the following sections; their prototypes are in STDLIB.H.

The atoi() Function

The library function atoi() converts a string to an integer. The prototype is

```
int atoi(char *ptr);
```

The function atoi() converts the string pointed to by ptr to an integer. Besides digits, the string can contain leading white space and a + or -- sign. Conversion starts at the beginning of the string and proceeds until an unconvertible character (for example, a letter or punctuation mark) is encountered. The resulting integer is returned to the calling program. If it finds no convertible characters, atoi() returns 0. Table lists some examples.

String-to-number conversions with atoi().

String	Value Returned by atoi()
"157"	157
"-1.6"	-1
"+50x"	50
"twelve"	0
"x506"	0

The atol() Function

The library function atol() works exactly like atoi(), except that it returns a type long. The function prototype is

```
long atol(char *ptr);
```

The atof() Function

The function atof() converts a string to a type double. The prototype is

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

The argument str points to the string to be converted. This string can contain leading white space and a + or -- character. The number can contain the digits 0 through 9, the decimal point, and the exponent indicator E or e. If there are no convertible characters, atof() returns 0. Table 17.3 lists some examples of using atof().

String-to-number conversions with atof().

String	Value Returned by atof()
"12"	12.000000
"-0.123"	-0.123000
"123E+3"	123000.000000
"123.1e-5"	0.001231

Character Test Functions

The header file CTYPE.H contains the prototypes for a number of functions that test characters, returning TRUE or FALSE depending on whether the character meets a certain condition. For example, is it a letter or is it a numeral? The is*xxxx*() functions are actually macros, defined in CTYPE.H.

The is*xxxx*() macros all have the same prototype:

```
int isxxxx(int ch);
```

In the preceding line, ch is the character being tested. The return value is TRUE (nonzero) if the condition is met or FALSE (zero) if it isn't. Table lists the complete set of is*xxxx*() macros.

The isxxxx() macros.

Macro	Action
isalnum()	Returns TRUE if ch is a letter or a digit.
isalpha()	Returns TRUE if ch is a letter.
isascii()	Returns TRUE if ch is a standard ASCII character (between 0 and 127).
iscntrl()	Returns TRUE if ch is a control character.
isdigit()	Returns TRUE if ch is a digit.
isgraph()	Returns TRUE if ch is a printing character (other than a space).
islower()	Returns TRUE if ch is a lowercase letter.
isprint()	Returns TRUE if ch is a printing character (including a space).
ispunct()	Returns TRUE if ch is a punctuation character.
isspace()	Returns TRUE if ch is a whitespace character (space, tab, vertical tab, line feed, form feed, or carriage return).
isupper()	Returns TRUE if ch is an uppercase letter.
isxdigit()	Returns TRUE if ch is a hexadecimal digit (0 through 9, a through f, A through F).

You can do many interesting things with the character-test macros. One example is the function get_int(), shown in Listing. This function inputs an integer from stdin and returns it as a type int variable. The function skips over leading white space and returns 0 if the first nonspace character isn't a numeric character.

Using the isxxxx() macros to implement a function that inputs an integer.

```
/* Using character test macros to create an integer */
/* input function. */
 #include <stdio.h>
```

```
  #include <ctype.h>

 int get_int(void);

 main()
{
    int x;
    x =  get_int();
    printf("You entered %d.\n", x);
}

int get_int(void)
{
    int ch, i, sign = 1;

    while ( isspace(ch = getchar()) );

    if (ch != `-' && ch != `+' && !isdigit(ch) && ch != EOF)
    {
        ungetc(ch, stdin);
        return 0;
    }

    /* If the first character is a minus sign, set */
    /* sign accordingly. */

    if (ch == `-')
        sign = -1;

    /* If the first character was a plus or minus sign, */
    /* get the next character. */

    if (ch == `+' || ch == `-')
        ch = getchar();

    /* Read characters until a nondigit is input. Assign */
    /* values, multiplied by proper power of 10, to i. */

    for (i = 0; isdigit(ch); ch = getchar() )
        i = 10 * i + (ch - `0');

    /* Make result negative if sign is negative. */

    i *= sign;

    /* If EOF was not encountered, a nondigit character */
    /* must have been read in, so unget it. */
    if (ch != EOF)
        ungetc(ch, stdin);

    /* Return the input value. */

    return i;
}
```

```
-100
You entered -100.
abc3.145
You entered 0.
9 9 9
You entered 9.
2.5
You entered 2.
```

Mathematical Functions

The C standard library contains a variety of functions that perform mathematical operations. Prototypes for the mathematical functions are in the header file MATH.H. The math functions all return a type double. For the trigonometric functions, angles are expressed in radians. Remember, one radian equals 57.296 degrees, and a full circle (360 degrees) contains 2p radians.

Trigonometric Functions

Function	Prototype	Description
acos()	double acos(double x)	Returns the arccosine of its argument. The argument must be in the range -1 <= x <= 1, and the return value is in the range 0 <= acos <= p.
asin()	double asin(double x)	Returns the arcsine of its argument. The argument must be in the range -1 <= x <= 1, and the return value is in the range -p/2 <= asin <= p/2.
atan()	double atan(double x)	Returns the arctangent of its argument. The return value is in the range -p/2 <= atan <= p/2.
atan2()	double atan2(double x, double y)	Returns the arctangent of x/y. The value returned is in the range -p <= atan2 <= p.
cos()	double cos(double x)	Returns the cosine of its argument.
sin()	double sin(double x)	Returns the sine of its argument.
tan()	double tan(double x)	Returns the tangent of its argument.

Exponential and Logarithmic Functions

Function	**Prototype**	**Description**
exp()	double exp(double x)	Returns the natural exponent of its argument, that is, e^x where e equals 2.7182818284590452354.
log()	double log(double x)	Returns the natural logarithm of its argument. The argument must be greater than 0.
log10()	double log10	Returns the base-10 logarithm of its argument. The argument must be greater than 0.

Hyperbolic Functions

Function	Prototype	Description
cosh()	double cosh(double x)	Returns the hyperbolic cosine of its argument.
sinh()	double sinh(double x)	Returns the hyperbolic sine of its argument.
tanh()	double tanh(double x)	Returns the hyperbolic tangent of its argument.

Other Mathematical Functions

Function	Prototype	Description
sqrt()	double sqrt(double x)	Returns the square root of its argument. The argument must be zero or greater.
ceil()	double ceil(double x)	Returns the smallest integer not less than its argument. For example, ceil(4.5) returns 5.0, and ceil(-4.5) returns -4.0. Although ceil() returns an integer value, it is returned as a type double.
abs()	int abs(int x)	Returns the absolute
labs()	long labs(long x)	value of their arguments.
floor()	double floor(double x)	Returns the largest integer not greater than its argument. For example, floor(4.5) returns 4.0, and floor(-4.5) returns -5.0.
modf()	double modf(double x, double *y)	Splits x into integral and fractional parts, each with the same sign as x. The fractional part is returned by the function, and the integral part is assigned to *y.
pow()	double pow(double x, double y)	Returns xy. An error occurs if x == 0 and y <= 0, or if x < 0 and y is not an integer.

CHAPTER 4 ESSENTIAL DATA STRUCTRURES

In every algorithm, there is a need to store data. Ranging from storing a single value in a single variable, to more complex data structures. In programming contests, there are several aspect of data structures to consider when selecting the proper way to represent the data for a problem. This chapter will give you some guidelines and list some basic data structures to start with.[2]

Will it work?

If the data structures won't work, it's not helpful at all. Ask yourself what questions the algorithm will need to be able to ask the data structure, and make sure the data structure can handle it. If not, then either more data must be added to the structure, or you need to find a different representation.

Can I code it?

If you don't know or can't remember how to code a given data structure, pick a different one. Make sure that you have a good idea how each of the operations will affect the structure of the data.

Another consideration here is memory. Will the data structure fit in the available memory? If not, compact it or pick a new one. Otherwise, it is already clear from the beginning that it won't work.

Can I code it in time? or has my programming language support it yet?

As this is a timed contest, you have three to five programs to write in, say, five hours. If it'll take you an hour and a half to code just the data structure for the first problem, then you're almost certainly looking at the wrong structure.

Another very important consideration is, whether your programming language has actually provide you the required data structure. For C++ programmers, STL has a wide options of a very good built-in data structure, what you need to do is to master them, rather than trying to built your own data structure.

In contest time, this will be one of the winning strategy. Consider this scenario. All teams are given a set of problems. Some of them required the usage of 'stack'. The best programmer from team A directly implement the best known stack implementation, he need 10 minutes to do so. Surprisingly for them, other teams type in only two lines:

"#include <stack>" and "std::stack<int> s;", can you see who have 10 minutes advantage now?

Can I debug it?

It is easy to forget this particular aspect of data structure selection. Remember that a program is useless unless it works. Don't forget that debugging time is a large portion of the contest time, so include its consideration in calculating coding time.

What makes a data structure easy to debug? That is basically determined by the following two properties.

1. State is easy to examine. The smaller, more compact the representation, in general, the easier it is to examine. Also, statically allocated arrays are **much** easier to examine than linked lists or even dynamically allocated arrays.

2. State can be displayed easily. For the more complex data structures, the easiest way to examine them is to write a small routine to output the data. Unfortunately, given time constraints, you'll probably want to limit yourself to text output. This means that structures like trees and graphs are going to be difficult to examine.

Is it fast?

The usage of more sophisticated data structure will reduce the amount of overall algorithm complexity. It will be better if you know some advanced data structures.

Things to Avoid: Dynamic Memory

In general, you should avoid dynamic memory, because:

1. It is too easy to make mistakes using dynamic memory. Overwriting past allocated memory, not freeing memory, and not allocating memory are only some of the mistakes that are introduced when dynamic memory is used. In addition, the failure modes for these errors are such that it's hard to tell where the error occurred, as it's likely to be at a (potentially much later) memory operation.

2. It is too hard to examine the data structure's contents. The interactive development environments available don't handle dynamic memory well, especially for C. Consider parallel arrays as an alternative to dynamic memory. One way to do a linked list, where instead of keeping a next point, you keep a second array, which has the index of the next element. Sometimes you may have to dynamically allocate these, but as it should only be done once, it's much easier to get right than allocating and freeing the memory for each insert and delete.

All of this notwithstanding, sometimes dynamic memory is the way to go, especially for large data structures where the size of the structure is not known until you have the input.

Things to Avoid: Coolness Factor

Try not to fall into the 'coolness' trap. You may have just seen the neatest data structure, but remember:

1. Cool ideas that don't work aren't.
2. Cool ideas that'll take forever to code aren't, either

It's much more important that your data structure and program work than how impressive your data structure is.

Basic Data Structures

There are five basic data structures: arrays, linked lists, stacks, queues, and deque (pronounced deck, a double ended queue). It will not be discussed in this section, go for their respective section.

Arrays

Array is the most useful data structures, in fact, this data structure will almost always used in all contest problems.Lets look at the good side first: if index is known, searching an element in an array is very fast, O(1), this good for looping/iteration. Array can also be used to implement other sophisticated data structures such as stacks, queues, hash tables.However, being the easiest data structure doesn't mean that array is efficient. In some cases array can be very inefficient. Moreover, in standard array, the size is fixed. If you don't know the input size beforehand, it may be wiser to use vector (a resizable array). Array also suffer a very slow insertion in ordered array, another slow searching in unordered array, and unavoidable slow deletion because you have to shift all elements.

Search	O(n/2) comparisons	O(n) comparisons
Insertion	No comparison, O(1)	No comparisons, O(1)
Deletion	O(n/2) comparisons,O(n/2) moves	O(n) comparisons more than O(n/2) moves

We commonly use one-dimensional array for standard use and two-dimensional array to represent matrices, board, or anything that two-dimensional. Three-dimensional is rarely used to model 3D situations.

Row major, cache-friendly, use this method to access all items in array sequentially.

```
for (i=0; i<numRow; i++)     // ROW FIRST = EFFECTIVE
   for (j=0; j<numCol; j++)
      array[i][j] = 0;
```

Column major, NOT cache-friendly, DO NOT use this method or you'll get very poor performance. Why? you will learn this in Computer Architecture course. For now, just take note that computer access array data in row major.

```
for (j=0; j<numCol; j++)     // COLUMN FIRST = INEFFECTIVE
   for (i=0; i<numRow; i++)
      array[i][j] = 0;
```

Vector Trick - A resizable array

Static array has a drawback when the size of input is not known beforehand. If that is the case, you may want to consider vector, a resizable array.

There are other data structures which offer much more efficient resizing capability such as Linked List, etc.

TIPS: UTILIZING C++ VECTOR STL

C++ STL has vector template for you.

```
#include <stdio.h>
#include <vector>
#include <algorithm>

using namespace std;

void main() {
  // just do this, write vector<the type you want,
  // in this case, integer> and the vector name
  vector<int> v;

  // try inserting 7 different integers, not ordered
  v.push_back(3); v.push_back(1); v.push_back(2);
  v.push_back(7); v.push_back(6); v.push_back(5);
  v.push_back(4);
  // to access the element, you need an iterator...
  vector<int>::iterator i;

  printf("Unsorted version\n");
  // start with 'begin', end with 'end', advance with i++
  for (i = v.begin(); i!= v.end(); i++)
    printf("%d ",*i); // iterator's pointer hold the value
  printf("\n");

  sort(v.begin(),v.end()); // default sort, ascending

  printf("Sorted version\n");
  for (i = v.begin(); i!= v.end(); i++)
    printf("%d ",*i); // iterator's pointer hold the value
  printf("\n");
}
```

Linked List

Motivation for using linked list: Array is static and even though it has O(1) access time if index is known, Array must shift its elements if an item is going to be inserted or deleted and this is absolutely inefficient. Array cannot be resized if it is full (see resizable array - vector for the trick but slow resizable array).

Linked list can have a very fast insertion and deletion. The physical location of data in linked list can be anywhere in the memory but each node must know which part in memory is the next item after them.

Linked list can be any big as you wish as long there is sufficient memory. The side effect of Linked list is there will be wasted memory when a node is "deleted" (only flagged as deleted). This wasted memory will only be freed up when garbage collector doing its action (depends on compiler / Operating System used).

Linked list is a data structure that is commonly used because of it's dynamic feature. Linked list is not used for fun, it's very complicated and have a tendency to create run time memory access error). Some programming languages such as Java and C++ actually support Linked list implementation through API (Application Programming Interface) and STL (Standard Template Library).

Linked list is composed of a data (and sometimes pointer to the data) and a pointer to next item. In Linked list, you can only find an item through complete search from head until it found the item or until tail (not found). This is the bad side for Linked list, especially for a very long list. And for insertion in Ordered Linked List, we have to search for appropriate place using Complete Search method, and this is slow too. (There are some tricks to improve searching in Linked List, such as remembering references to specific nodes, etc).

Variations of Linked List

With tail pointer
Instead of standard head pointer, we use another pointer to keep track the last item. This is useful for queue-like structures since in Queue, we enter Queue from rear (tail) and delete item from the front (head).

With dummy node (sentinels)
This variation is to simplify our code (I prefer this way), It can simplify empty list code and inserting to the front of the list code.

Doubly Linked List
Efficient if we need to traverse the list in both direction (forward and backward). Each node now have 2 pointers, one point to the next item, the other one point to the previous item. We need dummy head & dummy tail for this type of linked list.

TIPS: UTILIZING C++ LIST STL

A demo on the usage of STL list. The underlying data structure is a doubly link list.

```
#include <stdio.h>

// this is where list implementation resides
#include <list>

// use this to avoid specifying "std::" everywhere
using namespace std;

// just do this, write list<the type you want,
// in this case, integer> and the list name
list<int> l;
list<int>::iterator i;

void print() {
  for (i = l.begin(); i != l.end(); i++)
    printf("%d ",*i); // remember... use pointer!!!
  printf("\n");
}

void main() {
  // try inserting 8 different integers, has duplicates
  l.push_back(3); l.push_back(1); l.push_back(2);
  l.push_back(7); l.push_back(6); l.push_back(5);
  l.push_back(4); l.push_back(7);
  print();
  l.sort(); // sort the list, wow sorting linked list...
  print();

  l.remove(3); // remove element '3' from the list
  print();

  l.unique(); // remove duplicates in SORTED list!!!
  print();

  i = l.begin(); // set iterator to head of the list
  i++; // 2nd node of the list
  l.insert(i,1,10); // insert 1 copy of '10' here
  print();
}
```

Stack

A data structures which only allow insertion (push) and deletion (pop) from the top only. This behavior is called Last In First Out (LIFO), similar to normal stack in the real world.

Important stack operations

1. Push (C++ STL: push())
   Adds new item at the top of the stack.
2. Pop (C++ STL: pop())
   Retrieves and removes the top of a stack.
3. Peek (C++ STL: top())
   Retrieves the top of a stack without deleting it.
4. IsEmpty (C++ STL: empty())
   Determines whether a stack is empty.

Some stack applications

1. To model "real stack" in computer world: Recursion, Procedure Calling, etc.
2. Checking palindrome (although checking palindrome using Queue & Stack is 'stupid').
3. To read an input from keyboard in text editing with backspace key.
4. To reverse input data, (another stupid idea to use stack for reversing data).
5. Checking balanced parentheses.
6. Postfix calculation.
7. Converting mathematical expressions. Prefix, Infix, or Postfix.

Some stack implementations

1. Linked List with head pointer only (Best)
2. Array
3. Resizeable Array

TIPS: UTILIZING C++ STACK STL

```
Stack is not difficult to implement.Stack STL's implementation is
very efficient, even though it will be slightly slower than your
custom made stack.
```

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

using namespace std;

void main() {
  // just do this, write stack<the type you want,
  // in this case, integer> and the stack name
  stack<int> s;

  // try inserting 7 different integers, not ordered
  s.push(3); s.push(1); s.push(2);
  s.push(7); s.push(6); s.push(5);
  s.push(4);

  // the item that is inserted first will come out last
  // Last In First Out (LIFO) order...
  while (!s.empty()) {
    printf("%d ",s.top());
    s.pop();
  }
  printf("\n");}
```

Queue

A data structures which only allow insertion from the back (rear), and only allow deletion from the head (front). This behavior is called First In First Out (FIFO), similar to normal queue in the real world.

Important queue operations:

1. Enqueue (C++ STL: push())
   Adds new item at the back (rear) of a queue.
2. Dequeue (C++ STL: pop())

Retrieves and removes the front of a queue at the back (rear) of a queue.
3. Peek (C++ STL: top())
Retrieves the front of a queue without deleting it.
4. IsEmpty (C++ STL: empty())
Determines whether a queue is empty.

TIPS: UTILIZING C++ QUEUE STL

Standard queue is also not difficult to implement. Again, why trouble yourself, just use C++ queue STL.

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

// use this to avoid specifying "std::" everywhere
using namespace std;

void main() {
  // just do this, write queue<the type you want,
  // in this case, integer> and the queue name
  queue<int> q;

  // try inserting 7 different integers, not ordered
  q.push(3); q.push(1); q.push(2);
  q.push(7); q.push(6); q.push(5);
  q.push(4);

  // the item that is inserted first will come out first
  // First In First Out (FIFO) order...
  while (!q.empty()) {
    // notice that this is not "top()" !!!
    printf("%d ",q.front());
    q.pop();
  }
 printf("\n");}
```

CHAPTER 5 INPUT/OUTPUT TECHNIQUES

In all programming contest, or more specifically, in all useful program, you need to read in input and process it. However, the input data can be as nasty as possible, and this can be very troublesome to parse.[2]

If you spent too much time in coding how to parse the input efficiently and you are using C/C++ as your programming language, then this tip is for you. Let's see an example:

How to read the following input:

```
1 2 2 3 1 2 3 1 2
```

The fastest way to do it is:

```
#include <stdio.h>
int N;
void main() {
while(scanf("%d",&N==1)){
    process N…  }
}
```

There are N lines, each lines always start with character '0' followed by '.', then unknown number of digits x, finally the line always terminated by three dots "...".

```
N
0.xxxx...
```

The fastest way to do it is:

```
#include <stdio.h>
char digits[100];



void main() {
scanf("%d",&N);
for (i=0; i<N; i++) {
    scanf("0.%[0-9]...",&digits); // surprised?
    printf("the digits are 0.%s\n",digits);
  }
}
```

This is the trick that many C/C++ programmers doesn't aware of. Why we said C++ while scanf/printf is a standard C I/O routines? This is because many C++ programmers "forcing" themselves to use cin/cout all the time, without realizing that scanf/printf can still be used inside all C++ programs.

Mastery of programming language I/O routines will help you a lot in programming contests.

Advanced use of printf() and scanf()

Those who have forgotten the advanced use of printf() and scanf(), recall the following examples:

```
scanf("%[ABCDEFGHIJKLMNOPQRSTUVWXYZ]",&line); //line is a string
```

This scanf() function takes only uppercase letters as input to line and any other characters other than A..Z terminates the string. Similarly the following scanf() will behave like gets():

```
scanf("%[^\n]",line); //line is a string
```

Learn the default terminating characters for scanf(). Try to read all the advanced features of scanf() and printf(). This will help you in the long run.

Using new line with scanf()

If the content of a file (input.txt) is

```
abc
def
```

And the following program is executed to take input from the file:

```
char input[100],ch;
void main(void)
{
    freopen("input.txt","rb",stdin);
    scanf("%s",&input);
    scanf("%c",&ch);
}
```