Computer Programming

Strings

Robert Varga

Technical University of Cluj-Napoca
Computer Science Department

Course 8

Contents

String definition

- strings are represented in the C language as an array of characters, ended with a special character called the null-character
- the null-character has ASCII code 0 and the equivalent char representation `'\\0'`
- not to be confused with the NULL pointer
- strings have special syntax for initialization, printing and other operations
- other option would be to represent strings as the pair: pointer to char and length
  - used in other languages
  - has the advantage of quick access to length at the cost small storage space

Memory representation

- since strings are arrays, they are stored as a contiguous block of chars in the memory
- the string can contain multiple null-characters, but its length is given by the position of the first such character
- special care must be taken to have space for the null-character
- when a string is allocated dynamically the block size must be at least 1 larger than the number of useful characters

```
char s[] = "abc";
```

	0	1	2	3
s:	'a'	'b'	'c'	0

String declaration - 1st variant

```
char s[dim];
```

- as an array of given length
  - space is allocated for `dim` characters
  - this includes the null-character
  - it is recommended to allocate slightly more space than the required characters in case the string is changed
  - `s` is in fact a constant pointer, it cannot be changed
  - recommended in most cases

String declaration - 2nd variant

```
char s[] = "init";
```

- as an array whose length is deduced from initialization
  - the required space is deduced from the string literal provided during initialization
  - the space includes the null-character
  - cannot add additional characters to the string
  - `s` is in fact a constant pointer, it cannot be changed
  - recommended only if no characters will be added to it

String declaration - 3rd variant

```
char* s = "init";
```

- as a pointer to a string literal
  - string literals are stored in a read-only part of the memory
  - the literal contains all characters including the null-character
  - `s` is in fact a pointer to constant characters
  - its content cannot be changed
    - s[0] = 'x' produces run-time error
  - recommended if we only want to read from the string

String initialization

- a string can be initialized using the normal syntax from arrays
  - in this case we must specify all the individual characters and the null-character explicitly
  - a character literal is given using single quotation marks like: `'a'`
  - example: `char s[] = {'a', 'b', 'c', '\0'};`
- there is also a special initializer syntax for strings
  - in this case we must specify all the characters as a string literal
  - no need to write out the null-character, it is added on automatically
  - string literals are given using double quotation marks
  - example: `char s[] = "abc";`

String assignment

- the way we assign a string to another string differs based on its type
- for strings declared as arrays (variants 1 and 2)
  - we must use the dedicated copy function from the `string.h` library
  - or we must write out the loop ourselves

```
char s[20];
strcpy(s, "abc");
```

  - the required space for the string must be allocated
- for strings declared as pointers (variant 3)
  - we can assign a new pointer
  - the old memory address in s is lost

```
char* s;
s = "ab";
```

Length vs. size

- the length of a string is given by the number of useful characters it contains
  - it can be obtained with the `strlen` function
  - equivalently, it is equal to the position of the first null-character
  - 0-indexing is employed like in all arrays
  - the operation requires a search for the null-character, and this must be done every time the `strlen` is called
- the size of a string is the number of bytes it occupies in the memory
  - it can be obtained with the `sizeof` operator/function
  - may be different from length
  - for strings represented by an array
    - `sizeof` returns the size of the array, including the null-character
  - for string represented by a pointer
    - `sizeof` will return the size of the pointer
    - typically 4 or 8 bytes, depending on the OS

Program 8.1 - Length vs. size

```
1  #include <stdio.h>
2  #include <string.h>
3
4  void f(char s[]){
5      printf("sz %d\n", sizeof(s));
6  }
7  int main(){
8      char s[] = "a";
9      f(s);
10     printf("sz %d\n", sizeof(s));
11     printf("len %d\n", strlen(s));
12
13     char s2[20] = "a";
14     printf("sz %d\n", sizeof(s2));
15     s2[1] = 'b';
16     printf("sz %d\n", sizeof(s2));
17     printf("len %d\n", strlen(s2));
18     return 0;
19 }
```

Program 8.1 - Length vs. size

```
1  #include <stdio.h>
2  #include <string.h>
3
4  void f(char s[]){
5      printf("sz %d\n", sizeof(s));
6  }
7  int main(){
8      char s[] = "a";
9      f(s);
10     printf("sz %d\n", sizeof(s));
11     printf("len %d\n", strlen(s));
12
13     char s2[20] = "a";
14     printf("sz %d\n", sizeof(s2));
15     s2[1] = 'b';
16     printf("sz %d\n", sizeof(s2));
17     printf("len %d\n", strlen(s2));
18     return 0;
19 }
```

- size of a pointer 4 or 8
- 2 bytes, including '\0'
- length = 1 useful character
- the size of the array = 20
- still the size of the array = 20
- length = 2 useful characters

String/number conversions

- to convert a string representing a number to a fundamental type
  - we can use a specialized function like `atoi`, `atof`, `atoll` for types int, float/double, long long respectively
  - we can use the sscanf function to scan in any format from a given string

```
char s[] = "123.5";
double x;
sscanf(s, "%lf", &x);    x = atof(s);
```

- to convert a number to a string
  - we can use the sprintf function
  - this saves the print output in a given string instead of printing it on the screen

```
char s[20];
double x = 123.5;
sprintf(s, "%f", x);
```

Program 8.2 - String parsing

```
1  #include <stdio.h>
2  #include <string.h>
3
4  int main() {
5    char* s = "15/6 7/128 1/1";
6    while (*s) {
7      int x, y, n;
8      sscanf(s, " %d/%d%n",
9        &x, &y, &n);
10     printf("%d/%d %d\n",
11       x, y, n);
12     s += n;
13   }
14   return 0;
15 }
```

- read x, y and store the number of characters read in n using %n
- advance the pointer by n positions = change the starting position
- we lose access to the original string
- alternative: use `strtok` but this needs write access to the string

Contents

Functions from ctype.h - character manipulation

isdigit	returns true if the input character is a digit
isalpha	returns true if the input character is a letter
isupper	returns true if the input character is an upper-case letter
islower	returns true if the input character is an lower-case letter
toupper	returns the corresponding upper-case letter for a lower-case letter
tolower	returns the corresponding lower-case letter for an upper-case letter
isspace	returns true if the input character is a whitespace (space, tab, new-line)

Functions from string.h - string manipulation

strlen	returns the length of the input string
strcpy	copies the second string to the address from the first parameter
strncpy	same as previous, but only n characters and a null-character
strcat	concatenate two strings
strdup	returns a dynamically allocated duplicate of the input string
strcmp	compares two strings based on lexicographical order
strchr	search for a character in a string
strstr	search for a string in a string
strtok	separate the string in tokens based on delimiters

strlen function

```
size_t strlen(const char* s);
```

- determines the length of a string
- requires a search for the first null-character
- input parameters
  - const char* s - pointer to constant characters, input string which cannot be changed
- return value
  - length of the string
  - size_t is equivalent to unsigned long
- possible errors:
  - string is not correctly null-terminated
  - may access invalid memory zones in this case

strcpy function

```
char* strcpy(char* dst, const char* src);
```

- copies the string from `src` to `dst`
- includes the null-character
- input parameters
  - `char* dst` - pointer for destination string (location)
  - `const char* src` - pointer for source string
- return value
  - copy of the destination pointer in case of success
  - sets `dst[0] = 0` in case of an error, if `dst` is different from NULL
- possible errors:
  - `dst` is not a valid pointer
  - `dst` is not sufficiently large to hold `src`
  - two memory blocks overlap

strcmp function

```
int strcmp(const char* lhs, const char* rhs);
```

- determines the lexicographical order between the two strings
- input parameters:
  - `const char* lhs` - pointer indicating left string
  - `const char* rhs` - pointer indicating right string
- return value:
  - negative number in case `lhs < rhs`
  - zero in case `lsh = rhs`
  - positive number in case `lhs > rhs`
- possible errors:
  - `lhs` or `rhs` not a valid pointer

strtok function

```
char* strtok(char* s, const char* delim);
```

- separates the string in tokens based on the delimiters
- input parameters:
  - `char* s` - pointer indicating the string
  - `const char* delim` - string containing the delimiters
- return value:
  - pointer to the next token
  - NULL pointer if no next token exists
- strtok changes the input string and puts the null-character in every position where one of the characters from the delimiters is found

Program 8.3 - String tokenization

```
1  #include <stdio.h>
2  #include <string.h>
3
4  int main(){
5    char s[] =
6    "sir de cuvinte, separate"
7    "..,.prin.diferite caractere";
8    char* delim = " ,.";
9    char* tok = strtok(s, delim);
10   while(tok != NULL){
11     puts(tok);
12     tok = strtok(NULL, delim);
13   }
14   return 0;
15 }
```

- every character from delim defines the end of a token
- first, call strtok with the string s
- subsequent calls use NULL instead of s
- strtok changes the string s

Functions from string.h - memory management

memcpy	copy the content from the second address to the first address
memmove	same as previous, overlap possible
memcmp	compare the byte arrays from the two locations
memchr	search for a byte in the byte array
memset	set all bytes to a given value

memmove function

`void* memmove(void* dest, const void* src, size_t count`

- copy `count` bytes from `dest` to `src`
- input parameters
  - `dest` = void pointer to the destination
  - `src` = void pointer to the source
  - `count` = number of bytes to copy
- return value
  - returns a copy of the dest pointer in case of success
  - sets dest[0] to 0, if it is different from NULL in case of an error
- the source and destination memory zones may overlap

Sending strings to functions

- behavior is identical to sending arrays
- strings are sent to functions by value
  - during function call, a copy of the address of the first character is sent to the function as a pointer to `char`
  - changing values using the array [] syntax or the dereference operator * results in change outside the function
  - changing the pointer itself does not affect data outside the function
  - `sizeof` returns the size of a pointer

Dynamic allocation of strings

- respects rules discussed for arrays
- an additional byte should be allocated for the terminating null-character
- strings should be dynamically allocated if
  - they have a large size (more than 1 million characters)
  - they are created and returned from a function
  - their size changes during program execution
  - their size is unknown at compile-time
- small strings can be allocated on the stack
  - C99 standard allows the declaration of variable sized arrays, and hence strings, like `char s[n];`

Array of strings

- arrays of strings appear naturally in applications (list of words)
- there are two possible representations:
- using a bidimensional static array like `char s[dim1][dim2];`
  - allocated on the stack
  - only for small size
  - all strings have a fixed allocated size = `dim2`
  - changing strings can be done with `strcpy`
- using a 2nd order pointer like `char** s;`
  - allocated on the heap, dynamically
  - for larger size
  - each string occupies only the required space
  - we can change pointers

Program 8.4 - Sorting words - part 1

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  int main(){
6    char** words;
7    char s[100];
8    int n;
9    puts("How many words?");
10   scanf("%d", &n);
11   words = calloc(n, sizeof(char*));
12   for(int i=0; i<n; i++){
13     scanf("%s", s);
14     words[i] = calloc(strlen(s)+1, sizeof(char));
15     strcpy(words[i], s);
16   }
```

Program 8.4 - Sorting words - part 2

```
17    for(int i=0; i<n; i++){
18      for(int j=i+1; j<n; j++){
19        if (strcmp(words[i], words[j])>0){
20          char* tmp = words[i];
21          words[i] = words[j];
22          words[j] = tmp;
23        }
24      }
25    }
26    puts("words in sorted order:");
27    for(int i=0; i<n; i++)
28      puts(words[i]);
29    return 0;
30  }
```

Study problem - Substring count

- you are given a text `t` (long string)
- and a smaller string `s`
- find the number of times s appears in t as a substring
- overlaps between appearances are allowed
- example, s = "aba" appears three times in t = "xabacababa"
- let n denote the length of t, and m denote the length of s

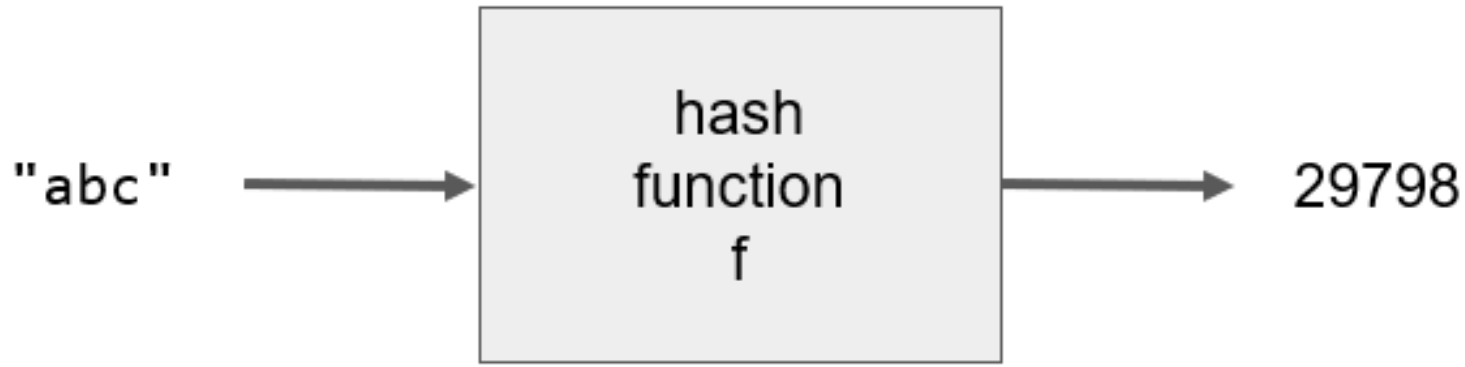
Naive approach:

- check every of the n-m+1 possible starting positions
- check if the substring from t is equal to s
- in the worst case: (n-m+1)*m checks

Study problem - Substring count

Hash functions for strings

- represent string by a number = hashing
- there are many possible strings, even for a small length
- inevitably some string will be mapped to the same value = collision
- a hash function receives as input a string and returns its hash value, a number which can be used to identify it (almost)



Study problem - Substring count

Rolling hash function

- define the rolling hash function of the string s

$$h(s) = s[0]p^{n-1} + s[1]p^{n-2} + ... + s[n-2]p + s[n-1]$$

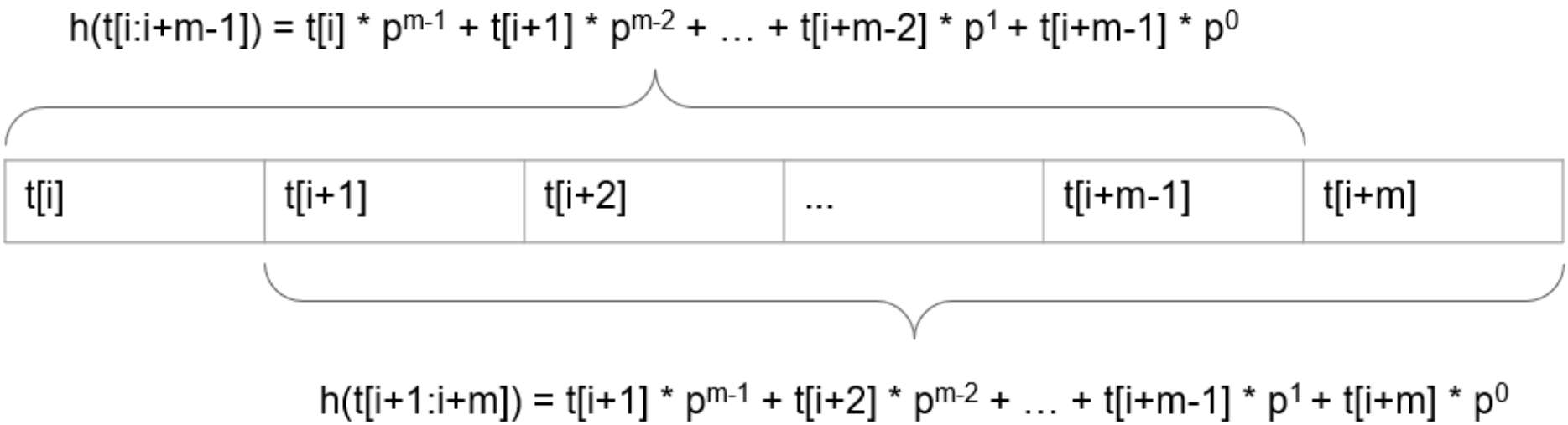
- where p is a prime number
- n is the length of s
- it is easy to update the hash value of a string after adding a character to it
- denote by $t[i : i + m - 1]$ the substring of length n of t starting from position i

Study problem - Substring count

Sliding the window

- how can we update the hash value when moving from one substring to the next one?
- we have

$$h(t[i+1:i+m]) = h(t[i:i+m-1]) \cdot p - t[i] \cdot p^m + t[i+m]$$



Study problem - Substring count

Karp-Rabin algorithm

- let hs be the rolling hash of s
- let ht be the rolling hash of the first m characters of t
- set $nr = 0$
- for every position i from 0 to n-m (inclusive)
  - if $hs = ht$
    - compare the subtring starting from i from t with s
    - if they are equal increment nr
  - slide the window one position in t and update ht

Program 8.5 - Substring count

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

const long long p = 17;
long long hash(char* s, int m){
  long long res = 0;
  for(int i=0; s[i] && i<m; i++)
    res = res * p + s[i];
  return res;
}

int main(){
  char t[] =
    "Abra abracadabra\n"
    "I wanna reach out and grab ya\n"
    "Abracadabra\n"
    "Abracadabra";
  char s[] = "bra";

  int n = strlen(t);
  int m = strlen(s);
  long long ht = hash(t, m);
  long long hs = hash(s, m);
  int nr = 0;
  long long pm = 1;
  for(int i=0; i<m; i++)
    pm *= p;
  for(int i=0; i<=n-m; i++){
    if (ht == hs){
      int match = 1;
      for(int j=0; j<m && match; j++)
        if (t[i+j]!=s[j])
          match = 0;
      nr += match;
    }
    ht = ht * p - pm * t[i] + t[i+m];
  }
  printf("%d\n", nr);
  return 0;
}
```