Fundamentals of Programming 1

Strings of Characters

Arkadiusz Chrobot

Department of Information Systems

December 13, 2024

Outline

Single Character Operations

The `getchar()` Function

The C language offers a set of functions that we call *the standard library*. Among the functions are those that perform operations on single characters represented as ASCII codes and usually stored in variables of `char` type. The C language makes it also possible to use characters encoded in other codes, like UTF, but this possibility won't be further discussed in this lecture.

The `getchar()` function allows the user to enter a single character using a keyboard. It is declared in the `stdio.h` header file. The function takes no arguments and returns the ASCII code of the character, however the returned value is of the `int` type instead of the `char` type.

Single Character Operations

Functions From the `ctype.h` Header File

The `ctype.h` header file provides other functions that perform operations on single characters. All of them take take one argument, which is a character (the ASCII code) and some of them also return a character (the ASCII code). However, the type of the parameter as well as the type of the returned value is `int`. The table below describes some of the functions.

Function's Prototype	**Description**
`int tolower(int c)`	If the argument is an uppercase letter, the function returns a corresponding lowercase letter.
`int toupper(int c)`	If the argument is a lowercase letter, the function returns a corresponding uppercase letter.
`int isalnum(int c)`	The function returns a nonzero value, if the argument is a digit or a letter. Otherwise it returns zero.

Single Character Operations

Functions From the `ctype.h` Header File — Continued

Function's Prototype	Description
`int isalpha(int c)`	If the character is a letter the function returns a nonzero value. Otherwise it returns zero.
`int isdigit(int c)`	If the character is a digit the function returns a nonzero value. Otherwise it returns zero.
`int isspace(int c)`	If the character is a whitespace the function returns a nonzero value. Otherwise is returns zero.
`int islower(int c)`	If the character is a lowercase letter the function returns a nonzero value. Otherwise it returns zero.
`int isupper(int c)`	If the character is an uppercase letter the function returns a nonzero value. Otherwise is returns zero.

Single Character Operations

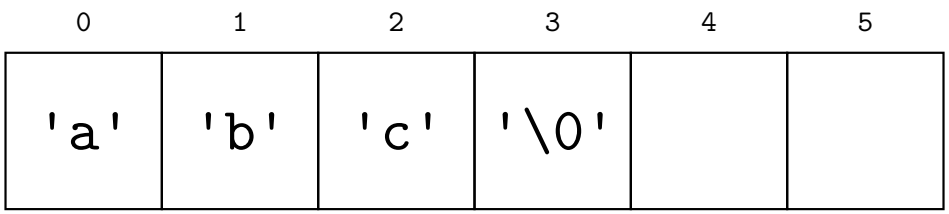
Implementation Example

Most of the functions introduced in the previous slides are easy to implement. Following is a function that takes a character as an argument and if it's a letter the function returns it after altering its case. To understand how it works it is enough to know that the ASCII codes of corresponding lowercase and uppercase letters differ by the value of the sixth bit ($2^5 = 32$). For the uppercase letters it is cleared (its value is zero) and for the lowercase letters it is set (its value is one).

```
char set_upper_or_lower(char input)
{
    if((input>='a'&&input<='z')||(input>='A'&&input<='Z'))
       input ^= 32;
    return input;
}
```

Data Types for Strings of Characters

Aside from individual characters, computers are able to process effectively *strings of characters* (*strings* for short). The C language has no special data type for strings. They are stored in arrays which elements are of `char` type. Those arrays are called `char` arrays. Not all elements of such arrays must be occupied by characters belonging to the string. Therefore each string is terminated by a special character that marks the end of the string. The value of the character's ASCII code is zero and in the C language notation it is represented as `'\0'`. The picture shows an array of six elements that stores the *abc* string.



String Initialization

The `char` array may be initialized in the place of its declaration, like other regular arrays. The initial characters have to separated by commas and embraced by curly braces. The last character has to be the `'\0'`. However, there is a more convenient way of initializing the `char` array. Is is easier to assign to the array a string of characters surrounded by quotation marks. The array has to have an additional element for the character marking the end of the string. It is better to not determine the number of elements in the array. The compiler will deduce it basing on the numbers of characters in the string. The string can also by assigned to the `char *` pointer. However, such a string should not be modified during the program run. Otherwise the program will be aborted.

String Initialization

Examples

```
int main(void)
{
        char first_string[] = {'E','x','a','m','p','l','e','\0'};
        char second_string[] = "Example";
        char *third_string = "Example";
        char fourth_string[10];

        second_string[1] = 'z';
        // third_string[1] = 'z'; // This is dangerous.

        return 0;
}
```

All arrays in the program are local variables, but in general they can also be declared global or passed by parameters to functions like any other arrays. The `forth_string` variable is not initialized, but it can store a string of 9 characters (without the character that marks the end of string). Also, a string can be assigned to the array in the place of its declaration.

Entering and Printing Strings

The strings embraced by quotation marks can be printed directly on the screen using the `printf()` and `puts()` functions. If the string is stored in an array, it can be printed by applying the `puts()` function or using the `printf()` function with the `"%s"` formating string. To allow the user to enter a string using a keyboard the `scanf()` function can be applied also with the `"%s"` formating string. However, in that case the `scanf()` function stops reading the input when it finds a whitespace character. Entering, for example, a full sentence may be accomplished with the use of the following formating string: `"%[^\n]s"`. The brackets allows the programmer to define which characters should be accepted by the `scanf()` function. The `^\n` notation means "any character except the new line character (associated with the Enter key in the keyboard)". The same effect may be achieved with the use of `fgets()` function. It takes three arguments: the name of the `char` array, the maximum number of characters the array can store and the `stdin` variable. The function returns `NULL` if it is unable to read the input. Otherwise it returns the pointer to the array.

Entering and Printing Strings

Example

```c
#include<stdio.h>

int main(void)
{
        char str[40];

         scanf("%s",str); // Reads the string until the first whitespace character.
         while(getchar()!='\n'); // Deletes all characters from input stream
                                 // including the \n character.
         scanf("%[^\n]s",str); //  Reads a string with whitespaces.
         while(getchar()!='\n'); // Deletes the \n character form the input stream.
         fgets(str,40,stdin); // Reads a string with whitespaces.

         return 0;
}
```

The `stdin` variable is the input stream, i.e. a pointer to a file. Please notice, that the name of a array is also a pointer, therefore it can be directly passed to the `scanf()` and `fgets()` functions as an argument. The address operator should not be applied in such a case.

Entering and Printing Strings

Example

```c
#include <stdio.h>


int main(void)
{
    char string[11];
    scanf("%10[^\n]s",string);
    printf("%s\n",string);
    return 0;
}
```

The example shows how to limit the number of characters read from the input with the use of the `scanf()` function. Without this limitation the function can try to store more characters in the array than it has elements. Such an error is called *an array overflow* and it has a potentially dangerous consequences.

String Operations

The `char` arrays and strings should not be compared with the use of relational operators, like `==`. If those operators are however applied to `char` arrays they will compare addresses of the variables, not their content. The same goes for the strings. It is also not possible to assign (except of the initialization) a string to the `char` array with the use of `=` operator.

The aforementioned operations (comparison and assigning) can only be performed by accessing each of the `char` arrays elements separately. To simplify them the authors of the C language have provided functions that perform such operations and are a part of the C language standard library. Those functions are available when the `string.h` or `strings.h` header files are included to the source code of a program. Next, some of the functions declared in the `string.h` header file are discussed.

String Operations

Example of Using	Description
`unsigned long int a = strlen(string);`	The function returns the number of characters in the argument, without the end of string character.
`strcpy(string_1, string_2);`	The function copies a string from its second argument to the array passed as its first argument. It returns the address of the first argument, but it is usually ignored.
`strncpy(string_1, string_2, number);`	The function does the same as the `strcpy()`, but copies at most the `number` of characters to the first argument. This is a protection from the `string_1` array overflow.

`if(strcmp(string_1,string_2)==0) {`
`…`
`} else {`
`…`
`}`
	The function compares two strings and returns a value less than zero if the `string_1` is less than the `string_2` or a value greater than zero if the `string_1` is greater than the `string_2` or zero, if they are equal. The exact way the strings are compared is discussed latter in the lecture.

String Operations

Example of Using	Description

`if(strncmp(string_1,string_2,number)==0) {`
`…`
`} else {`
`…`
`}`
	The function does the same as the **strcmp()**, but it compares at most the **number** of characters in both strings. This is protection from attempting to read more characters than the compared strings have.

`strcat(string_1,string_2);`	The function appends a string from the second argument to the string in the first argument. In other words it concatenates the strings. The function returns the address of its first argument, but it is usually ignored.
`strncat(string_1,string_2,number);`	The function does the same as the **strcat()** function, but it appends at most the **number** of characters of the second string to the first string.
`char *result = strstr(string,pattern);`	The function looks for the first occurrence of the **pattern** in the **string**. It returns the address of the first element of the **string** that stores the first character of the **pattern**.

String Operations

Example of Using	Description
`char *result = strtok(string,delimiters);`	The function splits the **`string`** into smaller parts according to the characters in the **`delimiters`** string. Using it is complicated, therefore an example showing how to apply it correctly is presented latter in the lecture.
`char *result = strchr(string,character);`	The function returns the address of the first element of the **`string`** that stores the **`character`**. The second parameter of the function is of **`int`** type.
`char *result = strrchr(string,character);`	The function returns the address of the last element of the **`string`** that stores the **`character`**.

String Operations

The `strcmp()` Function

The `strcmp()` function compares two strings stored in its arguments. It tests the pairs of characters of both strings, i.e. the first character of the first string with the first character of the second string and so on. If one of those characters has a greater ASCII code than the other then the string to which the character belongs is recognized as the greater one. In the converse case the function recognizes the second one as the greater. If both characters are the same the function proceeds to the next pair. The comparison can be finished in the following cases:

1. Both strings have the same number of characters and the characters are the same. The function returns zero because the strings are the same.
2. If the first string is a prefix of the second string, the functions recognizes the first as a less than the other. In the opposite case it is acknowledged as the greater one.
3. If the function finds a pair of different characters it returns a value as described before.

String Operations

Summary

In the table presented in previous slides are described some of the most frequently used functions in the C language for processing strings. If a function has a version that enables limiting the number of characters it processes then that version should be used, because it is more secure. For example the `strncpy()` should be used instead of the `strcpy()`. Any programmer may implement those functions on her or his own, given the fact, that elements of a `char` array may be accessed the same way as the elements of a regular array. Some implementations of those functions can be found in the book "The C Programming Language" by B. W. Kernighan and D. M. Ritchie. In this lecture implementations of some of those functions are also shown.

String Operations

The `strlen()` Function

The `strlen()` function is quite simple. It searches for the `'\0'` character in the `char` array and counts how many characters it has to pass before finding it. In the next slide is presented implementation of this function, but under a different name.

String Operations

The `strlen()` Implementation

```
unsigned int string_length(char *string)
{
    unsigned int i;
    for(i=0;string[i];i++)
        ;
    return i;
}
```

String Operations

The `strlen()` Implementation — a Comment

The code presented in the previous slide is only one of the many possibilities of implementing the `strlen()` function. Its implementation that uses the pointers arithmetic is presented in the already mentioned book by Kernighan and Ritchie. Please notice, that the header of the `for` loop is the most important part of the function.

String Operations

The `strncpy()` Function

The `strncpy()` function copies the content of its second argument to its first argument, but no more characters than it is given by the third argument. The maximal number of characters that can be stored in the first argument should be specified as the third argument of the function. It prevents storing some character outside the `char` array. Next are presented two implementations of the function with the names and the order of the first two parameters changed.

String Operations

The `strncpy()` Implementation — The First Version

```
void string_copy(char source[], char destination[], int length)
{
    int i = 0;
    while(length!=0 && source[i]!='\0') {
        destination[i]=source[i];
        i++;
        length--;
    }
    destination[i]='\0';
}
```

String Operations

Comment to the First Version of `strncpy()`

The string is copied in the `while` loop, where its characters are read from the `source` array and stored in the `destination` array. The loop terminates when the value of the `length` parameter becomes zero or when the end of string character is read from the `source` array. After the loop terminates the aforementioned character must be stored in the i-th element of the `destination` array.

String Operations

The `strncpy()` Implementation — a Version That Uses the Pointers Arithmetic

```
void string_copy(char *source, char *destination, int length)
{
    destination[length]='\0';
    while((length--)&&(*destination++=*source++))
        ;
}
```

String Operations

Comment to the Second Version of `strncpy()`

The second version of the `strncpy()` function is shorter than the first one thanks to the use of pointers arithmetics and some others features of the C language. The expressions `length--` and `*destination++=*source++` are not compared with zero, because their values are directly interpreted as true or false. The short-circuit evaluation of the `&&` operator means that the second expression is not evaluated when the first one is false. The elements of the arrays are accessed with the use of the pointer arithmetics. The post-increment operator does not increment the value of the element pointed by the pointer but the address stored in the pointer. It means that the pointer is ”advanced“ to the next element of the array. Before the loop is performed the end of string character is stored in the element of the destination array specified by the `length` variable. Thanks to that the copy of the string is terminated with the `\0` character even if the original has more than `length` elements.

String Operations

The `strstr()` Function

The **`strstr()`** function looks for the string passed as its second argument (the *pattern*) in the string passed as its first argument (the *text*). If it locates the pattern then it returns the pointer to the element of the text that stores the first character of the pattern. Otherwise it returns **`NULL`**. This operation is called *pattern matching*. There are many algorithms that performs this operation. The Boyer-Moore and KMP algorithms are effective for long patterns and texts. The naive algorithm that is introduced in the lecture is effective, according to prof. Steven S. Skiena, for patterns no longer than five characters. The algorithm looks for the occurrence of the pattern's first character in the text. If it locates it, then it checks if the rest of the characters also matches. If so then the pattern is located.

String Operations

Pattern Matching — an Implementation

```
int find_match(char string[], char pattern[])
{
    int i,j,
    pattern_length=strlen(pattern),
    string_length=strlen(string);

    for(i=0;i<=(string_length-pattern_length);i++) {
        j=0;
        while((j<pattern_length)&&(string[i+j]==pattern[j]))
            j++;
        if(j==pattern_length)
            return i;
    }
    return -1;
}
```

String Operations

Pattern Matching — a Comment

The `find_match()` function behaves a little differently than the `strstr()` function. It returns the index of the text string element that stores the first character of the pattern or `-1` if it is unable to locate the pattern. First, the function counts the lengths (the number of characters) of both strings. Next, in the `for` loop it checks elements of the text string to locate the first character of the pattern. If it finds this character then it performs the `while` loop as long as the subsequent characters in the text string match the next characters in the pattern or if the pattern ends. The latter is detected in the `if` statement. If the condition is satisfied then the pattern is located and the i-th element of the text string stores its first character. Therefore, the function returns the value of the `i` variable. If the condition is false then the function checks subsequent characters of the text string.

String Operations

Pattern Matching — a Comment

Please observe, that the `for` loop terminates when there is less characters to be checked in the text string than there is characters in the pattern. If the pattern is not yet located than there is no more chance of locating it and the function returns `-1`.

String Operations

The `strtok()` Function — Example of Usage

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

int main(void)
{
    char string[51] = {'\0'};
    scanf("%50[^\n]s",string);
    char *result = strtok(string," ");
    while(result!=NULL) {
        printf("%s\n",result);
        result=strtok(NULL," ");
    }
    return 0;
}
```

String Operations

Example of Using `strtok()` — a Comment

The program from the previous slide shows how to use the `strtok()` function to split a string into parts separated by spaces. The program prints all of those parts on the screen, each in a separate line. The function is invoked for the first time outside the loop. It takes the string to be split and a string containing a single space (the delimiter). If the function returns a value different than `NULL`, then the `while` loop is performed. Inside this loop the part of the string currently pointed by the `result` pointer is displayed on the screen with the help of `printf()` function. Then the pointer is used for storing results of subsequent calls of `strtok()`. The function is invoked in order to find the next part of the string, but this time it takes `NULL`[1] as its first argument. The operations of splitting and printing take place inside the `while` loop until the `strtok()` function returns `NULL` which means that there are no more characters in the string left.

[1] Zero can be used in place of this constant.

String Operations

The next two presented functions do not have their equivalents in the standard library of the C language. There are similar subroutines in the Pascal language. They perform such useful operations that it is worth creating their equivalents in the C language.

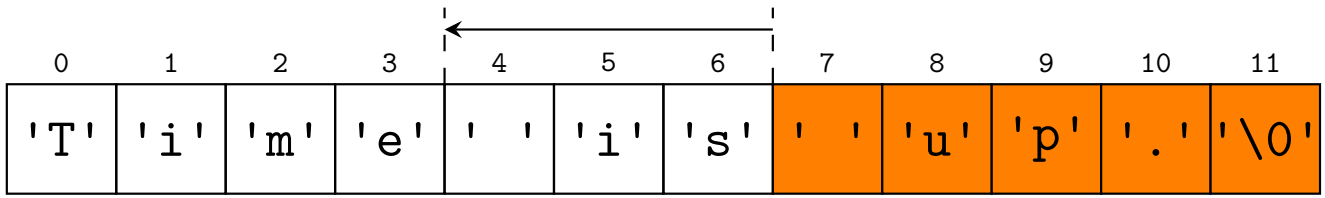
String Operations

Deleting Part of a String

The first function deletes part of a string containing a given number of characters, starting from a specified character. Removing a part of a string requires moving to the left all characters behind that part by as many places as there are characters in the part being deleted. To better understand this description please look at the visualization in the next slide. There, the " is" phrase is removed from the sentence "Time is up.". Please notice the space before the "is" word.

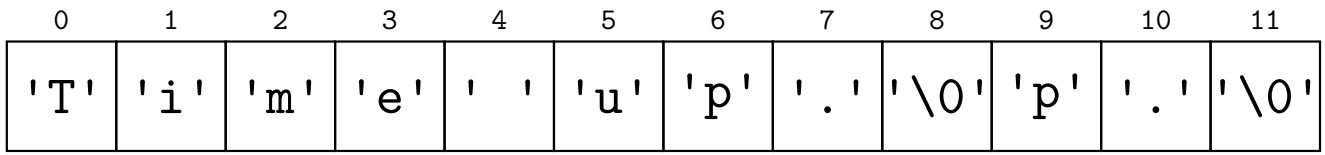
String Operations

Visualization — Deleting a Part of a String



String Operations

Visualization — Deleting a Part of a String



String Operations

Implementation — Deleting a Part of a String

```
int delete_from_string(char *string, unsigned int where, unsigned int how_many)
{
    if(where>=strlen(string))
        return -1;
    if(how_many>strlen(string)-where)
        return -2;

    int i;
    for(i=where;i<strlen(string);i++)
        string[i] = string[how_many+i];

    return 0;
}
```

String Operations

Deleting a Part of a String — a Comment

The string, whose part is to be removed, is passed to the function by its first parameter. The `where` parameter is the index of the element that stores the first character to be deleted. The number of characters to be removed is passed by the `how_many` parameter.
The `delete_from_string()` function first checks if the operation can be carried out. Should the value of the `where` parameter be greater than the number of the characters in the string, it will return `-1` and take no further actions. Similarly, if the `how_many` parameter value is greater than the number of characters possible to remove, it will return `-2` and terminate. If however the values of the parameters are correct, then the characters are moved accordingly in the `for` loop. Next, the function returns zero, signalling a success and exits.

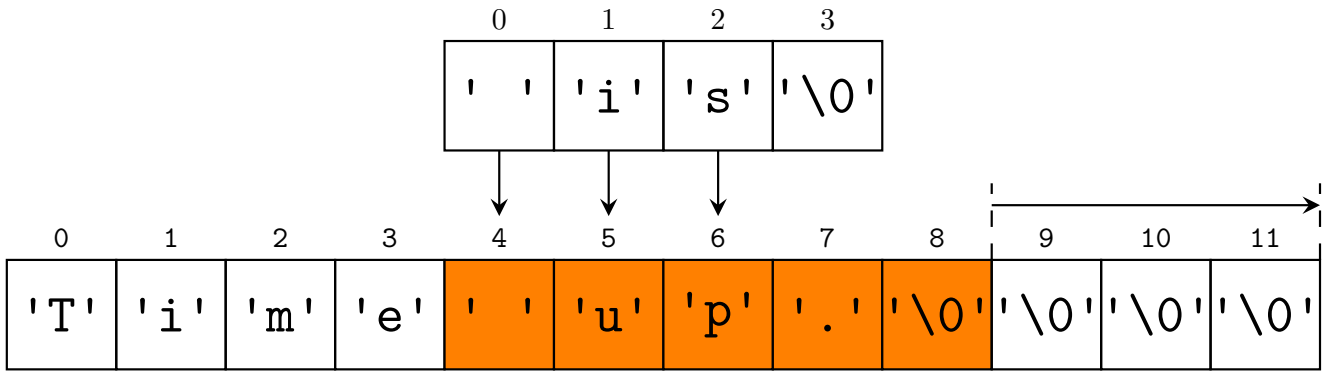
String Operations

Inserting Into a String

The next function inserts a string (let's call it a "pattern") into another string, starting from a given element. To carry out such an operation it is first necessary to move characters in the string to the right, by as many elements as the pattern has characters, starting from the last character in the string, and finishing with the character before which the pattern has to be inserted. Only then the pattern should be copied to the string. The operation is shown in the next slide, where the pattern " is" is inserted into the string "Time up.", starting from the fifth element.

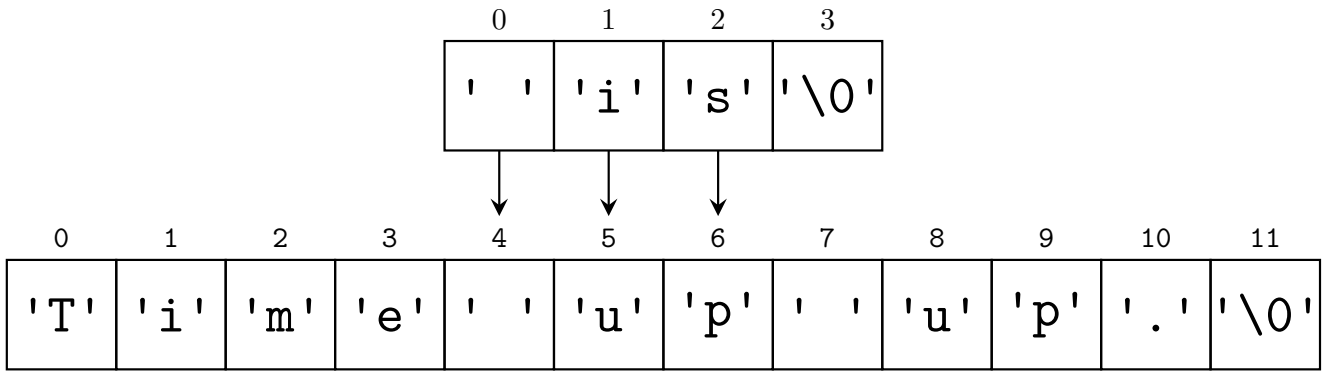
String Operations

Visualization — Inserting Into a String



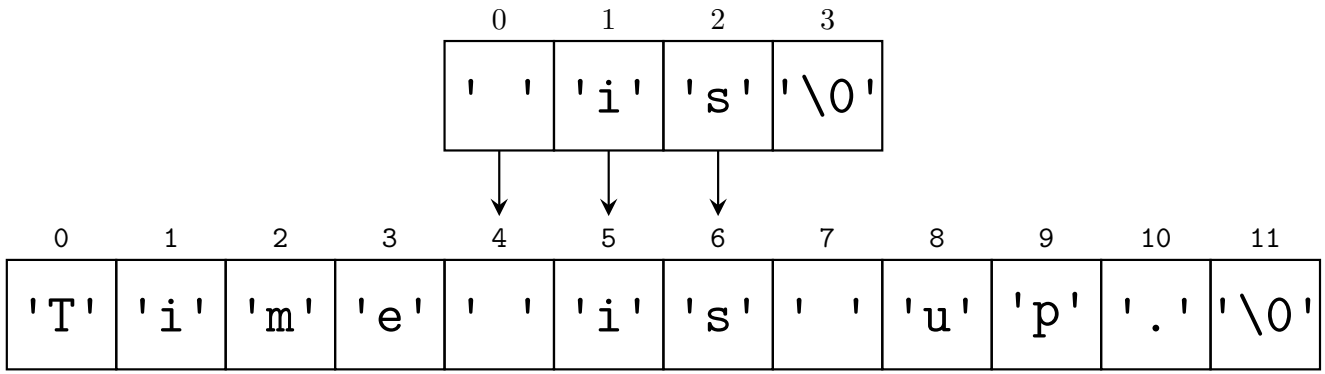
String Operations

Visualization — Inserting Into a String



String Operations

Visualization — Inserting Into a String



String Operations

Implementation — Inserting Into a String

```
int insert_into_string(char *string, const char *pattern, unsigned int where)
{
    if(strlen(pattern)+strlen(string)+1>NUMBER_OF_ELEMENTS)
        return -1;
    if(where>strlen(string))
        return -2;
    int i;
    for(i=strlen(string);i>=where;i--)
        string[i+strlen(pattern)]=string[i];
    for(i=0;i<strlen(pattern);i++)
        string[where+i]=pattern[i];
    return 0;
}
```

String Operations

Inserting Into String — a Comment

The `insert_into_string()` function inserts the string passed by the `pattern` parameter into the string passed by the `string` parameter, starting from the element whose index is passed by the `where` parameter. It first checks if the resulting string is fitting the `string` array. Should it have more characters than the array is capable of storing, the function returns `-1` and quits. Otherwise the function checks if the `where` parameter has a correct value, i.e. it specifies an element inside the string, not outside. Should the test have failed the function returns `-2` and also quits. If both tests are successfully passed then the characters are moved to the right in the first `for` loop by as many elements as there are characters in the pattern. In the second `for` loop the characters from the pattern are copied to the specified place in the string. Please observe the way the loop counters are initialized and applied. Please also notice the expressions used for indexing the strings. After the last loop finishes the function returns `0`, signalling success, and quits.

Conversions

The standard C library provides functions that convert a string passed to them as an argument into a number of a specified type. They are declared in the `stdlib.h` header file. The table in the next slide contains descriptions of several such functions.

Conversions

Example of Using	Description
`int number = atoi("45");`	The function converts a string into a number of the `int` type. The string has to *at least start* with digits or with the `+/-` character followed by digits. If the string cannot be converted, the function returns `0`.
`long int number = atol("45");`	The function works like `atoi()`, but returns a `long int` type number.
`long long int number = atoll("45");`	The function works like `atoi()`, but returns a `long long int` type number.
`double number = atof("45.5");`	The function works like `atoi()`, but returns a `double` type number. The string may also represent a number in the scientific notation.

Conversions

A number can be converted into a string with the use of the `sprintf()` and `snprintf()` functions, which are declared in the `stdio.h` header file. The functions behave similarly to the `printf()` function, but instead of printing they store the resulting string in a `char` array, passed to them as their first argument. The `snprintf()` function takes an additional, second argument which specifies the maximal number of characters that the `char` array may store. It is safer to use that function instead of the `sprintf()` function.

Security Considerations

The `snpritnf()` function may be used as replacement of `strncpy()` and `strncat()` functions, offering a better security for copy and concatenation operations. As well as limiting the number of characters to be copied, it also returns the total number of characters that it *would* copy, if the limit was not exceeded. This allows the program to detect whether the resulting string has been truncated. In some cases, it may matter. The `snprintf()` is applied securely if its formatting string is fixed and has been specified by the programmer.
Strings that a program gets directly from a user should never be processed with the use of `strcpy()` and `strcat()` and similar functions.

Thanks

Many thanks to Grzegorz Łukawski, PhD and Leszek Ciopiński, PhD for helping me to complete the Polish version of this slides.

Questions

?

The End

Thank You for Your attention!