CMPT 125

Introduction to Computing Science and Programming II

September 20, 2021

Strings

Char

How can we implement strings?
A natural idea: an array of chars

char - represents one symbol (letter / digit / punctuation mark)

```
char c1 = ‘a’, c2 = ‘B’, c3 = ‘;’, c4 = ‘6’;
printf("c1 = %c",  c1);
```

char also represents a number (1 byte).
Allows arithmetic on chars

```
char ch = ‘a’;
ch = ch+3; // sets ch = ‘d’
```

Strings

```
#include <stdio.h>
#include <string.h> // includes functions related to strings

int main() {

        char* password = “ABBBAC”;
        char* guess = “ABC”;

        if (password == guess) // WRONG – compares pointers
                … do something…

        if (strcmp(password, guess) == 0)
                printf(“CORRECT”);
        else
                printf(“WRONG”);
}
```

Strings

```
#include <stdio.h>
#include <string.h> // includes functions related to strings

int main() {

        char* password = “ABBBAC”;
        char* guess = “ABC”;

        if (strcmp(password, guess) == 0)
                printf(“CORRECT”);
        else
                printf(“WRONG”);
}
```

Strings

```
#include <stdio.h>
#include <string.h> // includes functions related to strings

int main() {

        char* password = “ABBBAC”;
        char* guess = “ABC”;

        if (strcmp(password, guess) == 0)
                printf(“CORRECT”);
        else
                printf(“WRONG”);
}
```

***__Question__**: How does strcmp() know the length of the strings?*

Strings

***<u>Question</u>**: how does strcmp() know the length of the strings?*

***<u>Answer</u>**: A string is an array of chars terminating with ‘\0’.*

‘\0’ is the char with value 0.

***<u>Comment</u>**: The length of the array can be longer than strlen().*

Example:

```
    char* word1 = “Hello”;
    char word2[6] = {‘H’, ‘e’, ‘l’, ‘l’, ‘o’, ‘\0’};


    printf (“%s \n”, word1); // prints Hello
    printf (“%s \n”, word2); // prints Hello
```

Strings

Example:

```
            char* word1 = “Hello”;
            char word2[6] = {‘H’, ‘e’, ‘l’, ‘l’, ‘o’, ‘\0’};
```

A comment:

word1 - initializing with “Hello”, creates an array of const chars
immutable strings - cannot be changed

This allows the compiler to perform optimizations on the code

word2 - can be modified, e.g.

```
            word2[3] = ‘p’; word2[4] = ‘\0’;
            printf (“%s\n”, word2); // prints Help
```

Strings

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

int main() {

    char* password = “ABBBAC”;
    char* guess = “ABC”

    if (strcmp(password,guess) == 0)
            printf(“CORRECT”);
    else
            printf(“WRONG”);
}
```

returns 0 if the strings are equal

returns i>0 if first > last

returns i<0 if first < last

Implement strcmp

```
int strcmp(const char *s1, const char *s2);
```

- If the strings are equal
  returns 0
- Otherwise,
  returns s1[j] – s2[j] for the minimal j where they differ

String.h - two useful functions

int strlen(const char s[])

- Returns the length of the string
- Counts until null terminator
- What happens if there is no ‘\0’ in the string?

char* strcpy(char* dest, const char* src)

- Copies the string src into dest
- Returns the pointer to dest
- What are our requirements about the parameters?
  The length of dest must be sufficient to copy src

- Implement the two functions

String.h – strlen() and strcpy()

```
char str1[]=“Hello";
char str2[40];

strcpy(str2,str1);

printf("%s\n", str2); // prints Hello
printf("%d\n", strlen(str2)); // prints 5

str2[4] = ‘\0’;

printf("%s\n", str2); // prints Hell
printf("%d\n", strlen(str2)); // prints 4
```

String.h – strcat()

```
char* strcat(char* dest, const char* src)
```

- Appends src to the end of dest
- What are our requirements about the parameters?

```
char str[80];
str[0] = ‘\0’;

strcpy (str,"these ");
strcat (str,"strings ");
strcat (str,"are ");
strcat (str,"concatenated.");

printf ("%s", str); // prints “these strings are concatenated.”
```

String.h - strcat

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

const char* colors[] = {"Red", “Blue", "Green"}; // array of char*
const char* widths[] = {"Thin", "Medium", "Thick", "Bold" };
...

char penText[20]; // array not initialized
...
int penColor = 2, penThickness = 2;

strcpy(penText, colors[penColor]);
strcat(penText, widths[penThickness]);

printf("My pen is %s\n", penText); // prints “My pen is GreenThick”
```

Reading user input

Reading user input

- So far we interacted with the user using printf()
- We can also read user’s input using the function scanf()
- The parameter to scanf() is a reference (address) to a variable

```
char name[];
int age;

printf(“Enter your name: ”);
scanf(“%s”, name); //&name[0]

printf(“Enter your age: ”);
scanf(“%d”, &age);

printf(“%s is %d years old\n”, name, age);
```

For scanf:
Why are we using &age?
Why name without &?

Initializing Arrays

Initializing arrays

- int arr1[5] = {31, 12, 5, -89, 3}; // initializing the 5 values
- int arr2[] = {31, 12, 5, -89, 3}; // same as above
- int arr3[10] = {31, 12, 5, -89, 3}; // the length of the array is 10,
  // but only the first 5 values have been initialized
- int* arr_ptr1 = arr1; // arr_ptr1 points to the beginning of arr1
- int* arr_ptr2 = {31, 12, 5, -89, 3};

Initializing arrays

- int arr1[5] = {31, 12, 5, -89, 3}; // initializing the 5 values
- int arr2[] = {31, 12, 5, -89, 3}; // same as above
- int arr3[10] = {31, 12, 5, -89, 3}; // the length of the array is 10,
  // but only the first 5 values have been initialized
- int* arr_ptr1 = arr1; // arr_ptr1 points to the beginning of arr1
- ~~int* arr_ptr2 = {31, 12, 5, -89, 3};~~

  // NO!!! The effect is arr_ptr2 = 31

Initializing Strings

Initializing strings

- char str1[5] = {'w', 'o', 'r', 'd', '\0'};   // initializing the 5 values
- char str2[] = {'w', 'o', 'r', 'd', '\0'};   // same as above
- char str3[10] = {'w', 'o', 'r', 'd', '\0'}; // the length of the array is 10,
  // but only the first 5 chars have been initialized
  // str3 can store strings of length <= 9
- char* str_ptr1 = str1; // arr_ptr1 points to the beginning of str1
- char* str4 = "word"; // creates an *immutable* string
- char* str5 = {'w', 'o', 'r', 'd', '\0'};

str4[3] = ‘k’ will crash

Initializing strings

- char str1[5] = {'w', 'o', 'r', 'd', '\0'};   // initializing the 5 values
- char str2[] = {'w', 'o', 'r', 'd', '\0'};   // same as above
- char str3[10] = {'w', 'o', 'r', 'd', '\0'}; // the length of the array is 10,
  // but only the first 5 chars have been initialized
  // str3 can store strings of length <= 9
- char* str_ptr1 = str1; // arr_ptr1 points to the beginning of str1
- char* str4 = "word"; // creates an *immutable* string
- ~~char* str5 = {'w', 'o', 'r', 'd', '\0'};~~

  // NO!!! The effect is str5 = ‘w‘

A comment on immutable strings

Consider the following code

```
char* str1 = "word";
char* str2 = "word";
if (str1 == str2)
    printf("same pointer");
else
   printf("differen
```

Are they pointing to the same location in memory?

Multidimensional arrays

Multidimensional array

- So far we've only seen 1D arrays.
- But often we need 2D / 3D arrays
- Examples:
  - **Picture**: each entry in the array is color of a pixel. (~ .bmp format)
  - **Temperature** in each point in the room.

Multidimensional array

```
int width = 640, height = 480;
int image[height][width]
```

Accessing a 2D array:

```
int i,j;
...

image[i][j] = 128;
```

OR

```
(*(image+i))[j] = 128; // each element of image is of type int[width]
                        // the size of each element is sizeof(int)*width
```

See multid_array.c

Multidimensional array

Q: Is the type int** equivalent to int[][] ?

A: int** is

- Array of pointers
- Array of (1-d) arrays, possibly of different lengths
- Pointer to a pointer

Multidimensional array

Example: Write a function that gets a 2d array, and checks if there are two equal rows in the array.

stdbool.h

stdbool.h

Write a function that gets an array of ints and a number and checks if it is contained in the array

```
#include <stdbool.h>

bool contains(const int* array, int len, int elt) {
        bool found = false;
        int i = 0;
        while (i < len && !found)    {
                if (array[i] == elt)
                        found = true;
                i++;
        }
        return found;
}
```

found == false

Enum/Typedef/Struct

Enum

User defined data types. Mainly used to assign names to integers.

Examples:

```
        enum suit {Hearts, Spades, Clubs, Diamonds};
                // default values are assigned starting from 0
                // i.e., Hearts = 0, Spades = 1, Clubs = 2, Diamonds = 3;


        enum emphasis {Bold = 1, Italic = 2, Underline = 4};
                // can define integer values of the names
```

Usage:

```
        enum siut card = Spades; // variable of type enum suit
```

The name of the type is
enum suit

Typedef

Typedef is used to give a name to a data type.

Examples:

1. typedef int whole_number;
2. enum months {January, February,...};
   typedef enum months month;
3. typedef enum boolean_values {false, true} bool; // all in one line

- Usage:

```
whole_number amount = 23;
bool flag = true;
month my_month = January;
```

Typedef

- Typedef is used to give a name to a data type.
- Typically we define new types outside of all functions.
- This allows the types to be used everywhere in the program
- More examples in types.c

Struct

- So far we have considered only simple types of variables (int, float, char, pointers).
- What if we want a more complicated data type?
- Example:

  A student has:

  First name
  Last name
  ID
  List of grades in homeworks

- We want an array of students.
- We can have array of first names, array of last names, array of IDs...

Hard to keep track of all the information in different arrays.

Struct

```
struct student_info {
        char* first_name;
        char* last_name;
        int ID;
        int grades[5];
};

struct student_info var_student;

typedef struct student_info student;

student list_of_students[180];

list_of_students[10].first_name = "Jack";
...
```

The type is called
struct student_info

Same as struct student_info

Struct

Could also write:

```
typedef struct student_info {
          char* first_name;
          char* last_name;
          int ID;
          int grades[5];
}  student;

student list_of_students[180];

list_of_students[10].first_name = "Jack";
...
```

Struct

Using pointers with structs:

```
student clark;
clark.first_name = "Clark";



student* student_ptr = &clark;
(*student_ptr).last_name = "Kent"; // accessing a field in struct
student_ptr->id = 123; // accessing a field in pointer to struct
```

A bit more on syntax:

Return values and conditions

Return values and conditions

- All command in C return a value. (Exception: void functions)
- Examples:

```
printf("%d\n" , 3 < 5); // prints 1
printf("%d\n" , 3 == 5); // prints 0
```

if statements:

```
if (cond)
    do_something();
else
    do_something_else();
```

Equivalent to:
if (cond != 0)

while statements:

```
while (cond)
    do_something();
```

Equivalent to:
while (cond != 0)

Multiple conditions

AND of conditions:

```
    if (cond1 && cond2)
        do_something();
    else
        do_something_else();
```

&& - and

- checks first if cond1 is satisfied

  (i.e., cond1 !=0)

- runs cond2 **only** if cond1 is satisfied

OR of conditions:

```
    while (cond1 || cond2)
        do_something();
```

|| - or

- checks first if cond1 is satisfied

  (i.e., cond1 !=0)

- runs cond2 **only** if cond1 is not satisfied

Recall the example: while (i < len && !found)

Global variables
vs
Static variables

Global variables

So far we have seen only variables defined inside functions.
The scope of the variables is limited only to the function.

It is possible to define a **global variable** that is visible everywhere.

```
#include <stdio.h>

int counter = 0; // init the global variable to zero

int main() {
        printf("global counter %d\n", counter); // prints 0
        counter++;
        printf("global counter %d\n", counter); // prints 1
        int counter = 0; // local variable
        printf(“local counter %d\n", counter); // prints 0
        return 0;
}
```

Static variables

It is also possible to define a **static variable** that will "remember“ its value in different calls of the function.

```
#include <stdio.h>

void test_static_count() {
        static int count = 0; // initialized only once!
        // do something...
        count++;
        printf("count = %d\n", count);
}

int main(void) {
        test_static_count(); // prints “count = 1”
        test_static_count(); // prints “count = 2”
        test_static_count(); // prints “count = 3”
        …
```

Macros

Using macros: #define

#define creates a constant that can be use globally.

Macros are not variables. Cannot be changed by the program.

```
#include <stdio.h>

#define COURSE_NAME "CMPT125"
#define PI 3.1415925

int main() {
    printf("%f\n", PI); // prints 3.1415925
    printf("%s\n", COURSE_NAME); // prints CMPT125
    printf("PI\n"); // prints PI
    …
```

Using macros: #define

#define macros are simply textual substitutions.

Preprocessor replaces the occurrences of the macros before compiling the code.

```
#include <stdio.h>

#define MY_FRAC 70/14
#define SQR(a) a*a

int main() {
        float x = MY_FRAC; // replaced by float x = 70/14;
        int y = SQR(5); // replaced by int y = 5*5;
        int z = SQR((5+2)); // replaced by int z = (5+2)*(5+2);
        int w = SQR(5+2); // replaced by int z = 5+2*5+2;
        …
```

Type casting in C

Type casting

- C allows conversions between different types of variable.
- Done when one data type can be changed to a different data type.
- Example:
  - int to float
  - short to int
  - int to long
- We can also type cast the result to make it of a particular data type.
- Need to be careful. Information may be lost!
- Examples:
  - float to int
  - int to char
  - char* to int*

Questions?

Comments?