# Exercise 8: Arrays of Strings

S Milton Rajendram

19-25 March 2018 (Mon-Thurs)

## 1 Array of strings

1. Declare an array of pointers to store strings.

```
char* names[N];
```

2. Declare an array of pointers to store strings and initialize them (maybe, with the names of the 12 months). Terminate the array with NULL pointer.

## 2 Passing an array of strings

1. Define a function to count the number of strings in the pointer array.

```
int strings_length (char* names[])
{
  for (int i = 0; names[i] != NULL; i++)
    ;
  return i;
}
```

2. Print an array of strings.

```
void strings_print (char* names[], int n) {
  for (int i = 0; i < n; i++)
    printf ("%s\n", entries[i]);
}
```

## 3 Allocate memory for a string

A character pointer variable is not the same as a C string. It can store a character pointer. It does not have array, yet.

```
char *s;
```

Variable `t` is a character pointer. It points to a character array of 5 characters. `t` *can be* changed later on to point to some other variable.

```
char *t = "June";
```

Variable `u` is a character pointer. It points to a character array of 5 characters. `u` cannot be changed to point to any other variable.

```
char u[] = "June";
```

Variable `v` is a character pointer. It points to a character array of 10 characters. The first 5 characters are initialized.

```
char v[10] = "June";
```

The differences between the four declarations (and initializarions) are illustrated in Figure 1.

```
char *s;
char *t = "June";
char u[] = "June";
char v[10] = "June";
```

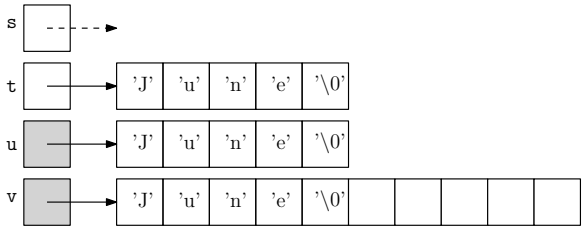


Figure 1: Difference between character pointer and character array

## 4 Allocate memory for a string

We want to clone a C-string. Function `string_clone(s)` takes a C-string as input and returns another C-string as the output. It allocates just enough memory for a new character array, using `malloc()` and copies `s` to the newly created character array, making it a C-string.

```
char *s = "In␣the␣beginning␣was␣the␣word.";
char *t = string_clone (s);
char* string_clone (char s[])
{
  char *t = (char*) malloc (strlen(s));
  strcpy (t, s);
  return t;
}
```

## 5 Sort an array of strings.

You have written `selection_sort()` to sort an array of numbers. Do necessary changes in `selection_sort()` to sort an array of $n$ lines in lexicographic order. The specification is

```
void selection_sort (char* names[], int n)
```

Test your sort function from `main()`. Use an array of strings, initialized along with declaration, in `main()`.

## 6 Read a sequence of strings (lines) from `stdin`

Write a function `strings_read(lines)` to read a sequence of lines from `stdin`. It stores the lines in an array of strings `char* lines[]`, and returns the count of lines as the result. Aftar you read each line from `stdin`, allocate memory using `string_clone()` and store it as a string in `char* lines[]`.

Test your function. Read the name list of your class from stdin. Sort it and print it.

## 7 Sort an array of strings based on the string length

The strings are sorted according to their length so that shorter lines come before longer ones in the result.

## 8 Search a string in an array of strings

We wish to insert a new string into an array of sorted strings. First, we need to find the right position where the new strings has to be inserted. Do the needed changes in `binary_partition()` to find the right position of a target string in a sorted array of strings.

## 9 Insert a target string in the right position in a sorted array of strings

Using `binary_partition()` find the "right" position of a target string in an array of sorted strings. Write a function `shift_right()` to shift the strings to the right of the target's position to make room for the target. Insert the target so that new array remains sorted.