Hello C...TYI!

Domhnall O'Hanlon

February 28, 2017

Part 4: Arrays & Strings

Up until now we have typically worked with only one piece of information - for example integers, floats, chars etc. In many cases - such as our grade calculator - all the information was of the same type, yet we still created seperate variables to store individual grades. Wouldn't it be much simpler if we could collect similar information like this in the same place?

Collections

An array is just another word of a collection. Examples of arrays can be found in mathematics, astronomy, and even biology. What sort of things do people typically collect? What sort of collections can you think of?


In the case of all of these collections, each consists of the same **type** of thing - stamps, coins etc. In programming that same is true. A collection of data, or an **array** must contain items that all have the same data type. For example, an array of ints could contain people's ages, an array of floats might contain bank balances, or an array of chars could store a student's letter grades.

Initialising

What sort of information do you need to know in order to be able to create, or initialise, a loop?

Like any variable, it will need a name. You also need to know what sort of information you're working with, so you have to know variable type. Finally you need to know how many items should go in the array.

Typically the syntax for an array is as follows:

```
int numbers[5] = {1,2,3,5,8};

char letters[3] = {'A', 'B', 'C'};

float myArray[10];
```

Accessing Elements

It is important to note that the first item of an array has an index of 0 (lives at index 0?) as this is often the source of off-by-one errors. For example,to print the letter **A** from the array `letters[3]` above, you select the 0th item from the array with the following piece of code:

```
printf("%c \n ", letters[0]);
```

Combining with Loops

You can quickly scan items to or prints items from an array by using an incrementer to both keep track of the iterations of your loop and the location (index) in the array that you want to access.

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

int main()
{
    int values[5];
    int counter;

    /*read 5 integers into the "values" array*/
    printf("Please enter 5 values\n");
        for(counter = 0; counter < 5; counter++){
                scanf("%d", &values[counter]);
            }
```

```
    /*print out the values from the array*/
    printf("\n The values you entered were: \n");
        for(counter = 0; counter < 5; counter++){
            printf("%d\n", values[counter]);
        }


    return 0;
}
```

Working with unknows

Let's say that you want your user to enter all their test results - the only problem is you don't know in advance how many exams they've taken. Take a look at the code below and see if you can understand what's going on.

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

int main()
{
    int inputs;
    printf("How many exams have you taken?\n");
    scanf("%d", &inputs);

    int results[inputs] = {};
    int index;

    printf("Please enter your grades in any order:\n");
        for (index = 0; index < inputs; inputs++)
        {
            scanf("%d", &results[index]);
        }

    for (index = 0; index < (sizeof(results)/sizeof(results[0])); index++){
        printf("Result %d is : %d\n", index+1, results[index]);
    }

    return 0;
}
```

Programming Challenges Using Arrays

Strings

In other lanugages, such as Java, there are dedicated *String* data types, but in C the convention is to use an array of chars, which does essentially the exact same thing. A C string is any array of chars followed by the null character, `\0`. The null character is also known as the string terminator (see figure 1). When you go to run your code the C compiler needs to know where every string ends, or terminates. To do this, the compiler parses your program and everywhere it finds closing quotation marks it insterts a character known as the **string terminator**. The string terminator is simply `\0` and takes up one additional byte of memory. This means that if you create a String variable to store your name, and if your name is 8 characters long, the name variable will actually take up 9 bytes of memory. This section outlines a number of different ways that strings can be created in C, as well as looking at how to manipulate stings with functions from the `string.h` library.

Array of Chars

The long way of creating a string is one character at a time so, if you really want to, you *could* do the following:

```
char str1[20] = {'H','e','l','l','o',' ','W','o','r','l','d', '!'};
```

Note that when working with chars that each element of the array has to be inside single quotes. A more concise way to acheive exactly the same thing is by enclosing the entire string in double quotes like so:

```
char str2[20] = "Hello World!";
```

Try creating a program that includes `str1` and `str2` and prints the both to the console.

string.h

Since the original C language doesn't contain functions for working with strings, much of this functionality is provided by the `string.h` library.

Copying Strings

To overwrite an existing string use the `strcpy()` function. This function takes two **arguments** or parameters. Remember, the `strcpy()` function can be used to overwrite *any* string, so the first thing you need to tell is is what string to replace (overwrite), then you need to tell it what to replace it with i.e. the string you wish to duplicate.

What do you expect the output of the following piece of code will be?

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

int main()
{
    /* code */
  char str1[20] = {'H','e','l','l','o',' ','W','o','r','l','d', '!'};

  char str2[20] = "Goodbye World!";

  printf("String 1 is: %s\n", str1);
  printf("String 2 is: %s\n", str2);

  /* Overwrite string 1 with the contents of string 2 */
  strcpy(str1, str2);

  /* display them again */
  printf("String 1 is: %s\n", str1);
  printf("String 2 is: %s\n", str2);



    return 0;
}
```

Measuring Strings

To find the length of a string simply use `strlen()` function. This function only accepts one argument, the string you want to measure the length of, and it returns an integer value with the length of the string.

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

int main() {
    char greeting[50] = "Hello World";
    int buffer_size;

    printf("%s\n", greeting);
    buffer_size = strlen(greeting);
    printf("The greeting is %d characters in length", buffer_size);
```

```
    return 0;
}
```

Write a program that asks the user for two strings and then tells them which string is longer.

Joining Strings

In many areas, particularly when working with databases, you frequently have to put two (or more) strings of text together. This joining operation is known as **concatenation**. You can concatenate two strings by using the `strcat()` function. Write a program that ask the user for their first name and their second name and concatenates these two strings, with a space in between.

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

int main() {

    char name[100], lastName[100];

    printf("What is your first name?\n");
    scanf("%s", name);

    printf("What is your second name?\n");
    scanf("%s", lastName);

    //add a space after firstName
    strcat(name, " ");

    //concatenate
    strcat(name, lastName);

    printf("Hello %s \n", name);
    return 0;
}
```

Comparing Strings

You can check if two strings are identical or not by using the string compare function `strcmp()`. As you can probably imageine, this function requires two arguments - the two strings you want to compare. If, for example, you want to compare *str1* and *str2* there are three possile outcomes. They're either identical, or string 1 might be smaller than string 2, or string 1 might be bigger than string

2. The `strcmp(str1, str2)` function returns 0 if both strings are identical, a negative number if *str1* is less than *str2* or a positive number if *str1* is greater than *str2*.

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

int main() {
    char s1[32] = "apples";
    char s2[32] = "oranges";
    char s3[32] = "apples";
    int result;

    result = strcmp(s1, s2);
    printf("Comparing %s and %s returned %d\n", s1, s2, result);

    result = strcmp(s1, s3);
    printf("Comparing %s and %s returned %d\n", s1, s3, result);

    if(result > 0){
        printf("%s is longer than %s\n", s1, s3);
    }
    else if(result < 0){
        printf("%s is longer than %s\n", s3, s1);
    }
    else{
        printf("%s and %s are identical\n", s1, s3);
    }

    return 0;
}
```

Coding challenge

Create a simple password app. Your code should should include a user-configurable access key. You should also make sure that your user can not enter a password longer than 20 characters.

```c
char key[20] = "YourPassword";
```

and some way to check if the users password is the same as your stored key.

More About scanf()

In the case of both printf() and scanf() the f stands for *formatted*. When you use `scanf()` the function reads in characters from the console until it encounters a **space** or a **new line**. This means that for things like sentences, or even just someone's full name, the `scanf()` function as we have been using it so far will not work.

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

int main() {

    char fullName[100];

    printf("\nWhat is your full name? \n");
    scanf("%100[^\n]s", name);
    printf("You typed \n%s\n", fullName);

    return 0;
}
```

In the example above, we use `%s` to specify that we are reading in a string, and then send it to the `fullName` buffer, just like with any other string we've seen so far. The two new things added here are actually between the `%` and `s`. The first value, `100`, is simply the buffer size. Since the fullName string can only contain 100 characters we limit the number of characters that the user can enter to 100 to avoid buffer overflow. If you were using a larger or smaller buffer you'd adjust this number accordingly. The second value `[^\n]` tells scanf to keep reading until it encounters a new line, i.e., until someone hits the enter key.

Summary

Having read this chapter and completed the programming exercises you shold now be able to: