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:

Hello C...TYI!

Domhnall O'Hanlon

March 7, 2017

Functions

As you've no doubt seen during your maths studies, a function is something that takes one or more arguments (inputs) and produces some return (output). In C you can of course create lots of different types of functions, using any of the data types that the language supports, but for the purpose learning about functions some mathematical examples are quite useful.

Creating Functions

So, what data type does a function have? Well, that depends on what type of data it returns. If you want to create a function that squares two numbers and returns the result then it will probably have an integer type. If you want to get the average or root of some numbers then it makes more sense to use a float. The process of creating a function is known as **declaring** a function. Typically, functions are declared just below any `#include` calls at the beginning of your code. This declaration tells the compiler not just what return type to expect but all the number and type of arguments the function accepts. This can also be refered to as the function **prototype**. The code associated with how the function actually performs is known as the **definition** of the function. Let's create a simple function that squares an integer input:

```
#include <stdio.h>

 int num, result;
 int square(int i);

 int main(){

     printf("Please enter a number to square\n");
     scanf("%d", &num);

    result = square(num);
```

```
     printf("%d squared is %d \n",num, result );
     return 0;
 }

  int square(int i){
    return i*i;
}
```

Maths

Create functions to calculate the area of a square, circle, cube and sphere.

Square

see the first example for squaring number.

Circle

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

#define PI 3.141593

float circle();

float result;

int main() {

    circle();
    printf("The area is %.2f units squared.\n\n", result);
}

float circle(){
    float rad;
    printf("What is the radius of the circle?\n");
    scanf("%f", &rad);
    result = PI*rad*rad;
    return result;
}
```

Cube & Sphere

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

#define PI 3.141593

float circle();
float cube();
float sphere();

float result;

int main() {

    cube();
    printf("The area is %.2f units squared.\n\n", result);

    sphere();
    printf("The area is %.2f units squared.\n\n", result);
}



float cube(){
    float side;
    printf("Enter the side length: \n");
    scanf("%f", &side);
    result = 6 * side * side;
    return result;
}


float sphere(){
    float rad;
    printf("What is the radius of the circle?\n");
    scanf("%f", &rad);
    result = 4*PI*rad*rad*rad;
    result = result/3;
    return result;
}
```

Summary

Having read this chapter, and completed all the programming exercises, you should now be comfortable with creating your own functions and calling them from anywhere in your main method.

Hello C...TYI!

Domhnall O'Hanlon

March 14, 2017

Pointers

The topic of pointers in C is one that many beginners find challenging, but once you get comfortable with them your applications will become more powerful and more efficient.

Welcome to the Hotel California

A pointer is a special type of variable (it's actually a constant!) whose value is the memory address of another variable. To help you visualise this, image a hotel with lots of floors, and many rooms on each floor. The rooms are analogous to locations in computer memory in that they don't ever change their pyhsical location or address. The occupants of the rooms however can change - so your hotel visitors are like programming variables, free to come and go as they please and easily replaced by other variables. ~~We can also extend this analogy a bit by considering twin rooms, double rooms, suites etc. These types of rooms might correspond to different types of data~~

A pointer variable is declared by prefacing its name with an asterisk. The address in memory is a numeric value so it's ok to use an interger as the type, but by starting your variable name with an asterisk (star) rather an a letter you are telling the compiler that this variable is actually a pointer rather than a conventional integer.

```
int main(){
    int *pointer;

    return 0;
}
```

You can still declare multiple varaible on one line, but each pointer has to start with `*`

```c
int main(){
    /*one pointer and one integer*/
    int *my_pointer, not_a_pointer;

    /*two pointers*/
    int *first_pointer, *second_pointer;
}
```

Convention

In many cases the accepted convention is to start a pointer variable with the letter p, for example:

```c
int main(){
    //create an integer variable
     int age;

     //create a pointer variable
     int *pAge;

     return 0;
}
```

Where are variables stored in memory?

To acces memory locations in C use a pointer thingy:

`%p`

We can echo a memory address to the console using something like the following snippet:

```c
    int i = 42;

    printf("%p ", i);

    return 0;
```

Some notes on Hex.

Creating Pointers:

Take for example: `int myInt = 42`, this can store any numeric value between x and y. It has a memory address and we can create `int * pMyInt = &myInt`

Dereference Pointer

If you want to retrieve the value that a pointer points to (as opposed to the address that it points to) then you use the unary operator, *, which is used to dereference the pointer.

```
int i = 42;
int *prt = &i;

int main(){
    printf("The address of int i is: %p \n", prt);
    printf("The value stored  at i is: %d \n", *prt);

    return 0;
}
```

Arrays and Pointers

create and array of vars loop through the array and print it's contents, and memory pointer.

int luckyNums[6] = [4,8,15,16,23,42];

```
printf(" Element \t Address \t Value")
for (i=0; i<6; i++){
   printf(" luckyNums[%d] \t %p \t %d", i, &luckyNums[i], luckyNums[i]);
}

//array names are just pointers to the first element of that array.
printf("\n luckyNums %p", luckyNums);
```

Strings and Pointers

Test it Out

```
int main(){
   j = 1;
   k = 2;
   ptr = &k;
   printf("\n");
   printf("j has the value %d and is stored at %p\n", j, (void *)&j);
   printf("k has the value %d and is stored at %p\n", k, (void *)&k);
   printf("ptr has the value %p and is stored at %p\n", ptr, (void *)&ptr);
   printf("The value of the integer pointed to by ptr is %d\n", *ptr);
```

```
    return 0;
}
```

Summary

Hello C...TYI!

Domhnall O'Hanlon

March 21, 2017

Why Structures?

We've seen lots of examples so far of where we would want to store multiple pieces of information. Up until now we've always used an array, however the main drawback of using an array is that it can only store one **type** of information, like an array of ints or an array of chars. The main advantage of using a struct is that it allows you to use different types of data.

Convention

The convention in C is to create your structs in a seperate header `.h` file, but for the time being we'll create them in our main.c files. In this lesson we'll create a basic car structure, make a few different cars and then modify them using a few functions we'll create ourselves.

Creating a struct

In something as complex as car, there are lots of different properties that we might want to assign. We can use strings to represent things like the make, model, features etc. numbers to represent properties such as horse power or top speed and arrays to capture the variety of engine sizes available, or perhaps types of fuel. The properties of a struct are contained between curly braces, and just like when you initialise any other data type, you must end your declaration with a semicolon.

```
struct car{
    char* make;
    char* model;
    int bhp, topSpeed;
}; /* don't forget the semicolon */
```

Using a Struct

Now that we've created a prototype for all cars in general, we can now go ahead and make specific cars by using our car struct.

```
    /* Add a new car by using our struct */
    struct car bv = {"Bugatti", "Veyron", 1000, 407};
```

The Dot Operator

The dot operator, `.`, is used to assess specific members (elements) of a struct. In the example of our car structure, it has properties, or elements corresponding to the make, model, horse power and, top speed. Each of these can be accessed using the syntax `name.element` like in the example below:

```
    printf("Make: \t %s\n", bv.make);
    printf("Model: \t %s\n", bv.model);
    printf("Horse Power: %d\n", bv.bhp);
    printf("Top Speed \t %d \n", bv.topSpeed);
```

Typedef

```
/* TYPEDEF: we don't have to be limited to just ints, chars, arrays etc - now that we can c
can create, or define, our own types using typedef */

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

typedef struct car{
    char* make;
    char* model;
    int bhp, topSpeed;
}car; /* don't forget the semicolon */

/*create a function which prints the top speed of any car */
void getTopSpeed(struct car anyCar){
    printf("The top speed of the %s %s is: %d km\\hr. \n",
        anyCar.make, anyCar.model, anyCar.topSpeed);
}

int main(int argc, char* argv[])
{
```

```
    /* Creating cars without having to use struct all the time */
    car bv = {"Bugatti", "Veyron", 1000, 407};
    car vg = {"Volkswagen", "Golf", 227,170};

    /* Use the function to get the top speed of the cars */
    getTopSpeed(bv);
    getTopSpeed(vg);

    return 0;
}
```

Example Code

A quick example of sending data to functions and making it globally accessible throughout your application.

Here is the complete code for this section:

```
/*
Now that we're using pointers the variables will update correctly.
 */

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

typedef struct person {
    char* name;
    int bankBalance;
} person;

void developApp();

int main(int argc, char* argv[])
{
    /* create a few people */
    person mz = {"Mark", 500};
    person es = {"Eduardo", 10000};

    developApp(&mz, 1000);

    /*print out mz.bankBalance here too */
    printf("In main() the balance is: %d ", mz.bankBalance);
    return 0;
}
```

```
void developApp(person *somebody, int sales)
{
    printf("%s just launched a new app\n", somebody->name);

    somebody->bankBalance += sales;

    printf("%s's new net worth is: %d \n", somebody->name, somebody->bankBalance);
```

Summary