Topic 6 –Arrays

SIMPLE VS COMPLEX TYPES

- Data types are critically important in that they determine the properties of the variables and values being processed.
- All of the data types we have dealt with so far have been built into the language.
    - For example, `char`, `int`, and `float` are all part of C.

    - These built-in types are called *simple* or *primitive* data types.

    - In general these cover the majority of different types of data that we need to store and process in our programs.

    - However, sometimes they are not enough and we need to create our own data types.
    - These new types are no longer built-in and they are called *complex* data types.

    - Complex data types are created by joining together the simple data types that are built into the language.
        - In this case, *complex* means made up of multiple parts, rather than difficult.

- In this topic we will be start looking at one of the major complex data types: the array.

INTRODUCING ARRAYS

What are Arrays?

- So far we have made programs that contain a small number of variables.
- However, if we have lots of data to store, this soon becomes impractical.

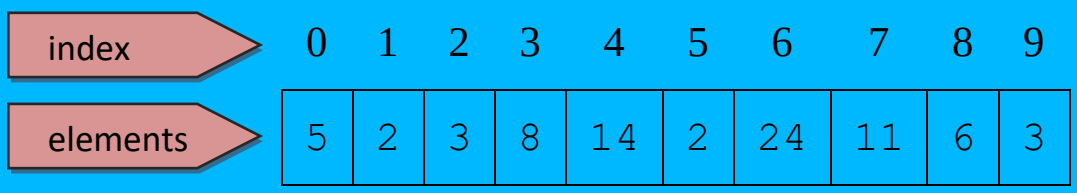
- An array is a complex data type that groups together a number of different values *of the same type* into a single structure.

  - The advantage of using an array is it becomes easy to create algorithms to solve problems that otherwise would be very difficult and time-consuming to deal with.

  - An array is defined as a contiguous block of memory holding *n* items, called *elements*, of a given type.

  - Each of these elements is assigned a number to identify it.
  - This value is called an *index*.
  - The index value starts at 0.

  - For example, the following is an array storing 10 integers:



Array Advantages

- So instead of having lots of separate variables, each storing a value, we can have a single variable storing many pieces of data.
- There are a number of advantages to this.

- First, unlike ordinary variables, the elements in an array do not need to be declared individually.
  - This means we can create as many pieces of data as we like simply by specifying **the number** of elements.

  - For example, to create an array with holding 10,000 integers, we just have to specify this *size*.

  - As indicated before, index values given to array elements always begin at 0.

  - Therefore, the first element in this array of 10,000 elements would be **0** and the last element would be **9999**.

- The second big advantage is that processing all of the elements in the array is (almost) as easy as processing a single individual value.
  - This makes working with a large amounts of data extremely efficient when using an array.

We will now look at examples of both of these properties.

Declaring Arrays

- Despite being very powerful, arrays are quite easy to use.
- Like variables they must be declared before you can use them and you need to specify the type of the array, its size and its name.
- The line of code below creates a new array called `myNums` containing 50 integers:

```
int myNums[50];
```

- Note: the number 50 here is the array's *size*, which is the number of elements the array contains.
- When declaring an array, the number in the square brackets represents the *size* of the array.
- However, it is important to note that, after declaration, the number in the square brackets refers to something slightly different.
- Specifically, after the array has been declared, the number in the square brackets becomes an *index*.

Accessing Array Elements

- Once created you can then access specific elements within the array by their *index*.
  - Remember that array elements are numbered from 0 to one less than the size of the array:

```
/* Put values in the first and last elements
of the myNums array */
myNums[0] = 0;
myNums[49] = 0;
```

  - By indexing a specific element in the array, you can treat this just like a variable of that particular type.
    - Note that element with index `49` is actually the 50th and last element in the array.
    - This is because array numbering begins at `0`.
  - In other words `myNums` is an array and has certain special properties but `myNums[0]` behaves just like a normal `int`.

- Also note that the value inside the square brackets can be *anything that evaluates as an integer*.
- This includes literal values, arithmetic expressions or even integer return values from function calls.
  - For example:

```
const int SIZE=50;
…

int myNums[SIZE];

myNums[49] = 5;
myNums[SIZE-1] = 5;
```

Assuming the declaration of the constant, the last two lines are equivalent.

Why are Arrays Useful?

- The best way to understand why arrays are so useful is to look at an example of how they work.

- Assume we have a class made up of just three students and we want to write a program to calculate the average grade for the class.
- Part of the code to solve this problem might look like:

```
/* Number of students. */
const int NUMBER_OF_MARKS = 3;

/* One variable required for each mark. */
int mark1, mark2, mark3;

/* Variables to store the totals and average.
*/
int total = 0;
float avg;

scanf("%d%*c", &mark1);
scanf("%d%*c", &mark2);
scanf("%d%*c", &mark3);


/* Add up marks */
total = mark1 + mark2 + mark3;

/* Calculate average but note that total must
   be cast to a float to ensure floating
   point division.
*/
avg = ((float) total)/ NUMBER_OF_MARKS;

printf("Average = %f\n", avg);
```

- A pretty simple program, right?

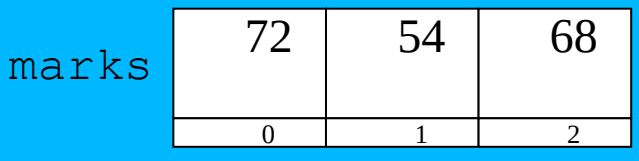
- But it has an obvious problem...
  - If the number of students starts to get larger, the amount of code we have to write increases *proportionally*.
  - This might be alright if there is only five or perhaps even 10 students.
  - But if the class has 200 students then there will be a lot of pointless typing needed!

- In other words... the solution *does not scale*.

- However, by using arrays we can write a program that will scale *very* easily.
- Instead of creating separate variables to store each mark we can create a single array with the appropriate number of elements.

- Using three separate variables this is stored in memory as:

- By creating an array all variables are stored under one name but can be accessed separately via their index:



- Here is the same program fragment implemented using an array:

```
/* The number of students - now easily changed */
const int NUMBER_OF_MARKS = 3;

int total = 0;
float avg;

/* A counter variable for accessing array elements.
*/
int index;

/* Define a new array to store all of the values. */
int marks[NUMBER_OF_MARKS];

/* for loops are very commonly used to process
arrays*/
for(index = 0; index < NUMBER_OF_MARKS; index++)
{
    /* Read in mark and add it to the running total.
    */
   scanf("%d%*c", &marks[index]);
   total = total + marks[index];
}

avg = ((float) total) /NUMBER_OF_MARKS);
printf("Average = %f\n", avg);
```

- Things to note:
  - We use a *for* loop to iterate through the array because we know the size of the array.
  - The loop's counter variable `index` is used to access each element in the array.
  - Because of this `index` starts at 0 and goes up to one less than `NUMBER_OF_MARKS` which is the size of the array.
  - By using the size of the array as a constant throughout the code, we can easily change the number of students by changing the value of `NUMBER_OF_MARKS`.
  - Note that `marks[index]` behaves just like an ordinary `int`.

BASIC ARRAY OPERATION

- Working with arrays is generally very simple and involves the use of a for loop:
  - The principle is to apply the logic for processing a single array element to the entire array by looping through it.
  - For each iteration, a different array element will be processed based upon the counter variable of the for loop.

```
#include <stdio.h>

const int SIZE = 5;

int main()
{

     int array[SIZE];
     int i;

      for(i = 0; i < SIZE; i++)
      {
         printf("Enter element %d: ", i);
         scanf("%d%*c", &array[i]);
      }

      printf("Array contents are:\n");
     for(i = 0; i < SIZE; i++)
      {
         printf(" Element %d is %d\n", i, array[i]);
      }

     return(0);
  }
```

ADVANCED ARRAY USAGE

Array Declarations

- We've already seen how empty arrays can be declared:

```
float scores[35];
```

- However, you can also declare arrays and initialise them to a set of values, just like with ordinary variables:

```
int shutter[] = {30, 60, 125, 250, 500};
float aperture[] = {2.8, 4.0, 5.6, 8.0,
                    11.0};
```

- Obviously this is only practical for small arrays.

- Note that specifying the array size when initialising an array is *optional*.

- However, if you do specify the size of the array but do not provide enough initial elements to fill it, the remaining elements will be automatically initialised to 0.

- This can be used as a shortcut for initialising a large array to 0.
- For example:

```
int scores[100] = {0};
```

Static vs Dynamic Arrays

- The C language allows for both static and dynamic arrays.
- Static arrays are those where the size of the array is defined by the programmer when they write the program.
    - The examples from this topic are static arrays.

- Dynamically allocated arrays are those where the program calculates the size an array needs to be and creates the array of this size while the program is running.
- This means that the programmer does not need to fix the maximum size of the array.

- In this unit we will only cover static arrays, as dynamic arrays are more difficult to work with in C.
    - If you go on and study further programming units, it is likely you will work with dynamic data structures.

- However, in this course this limitation means you will need to sometimes estimate or fix the maximum amount of data to be stored in an array, and then ensure that this limit is not exceeded.

Bounds Checking

- Creating an array gives you an area of memory to store as many pieces of data as you specify.
- Some languages keep track to make sure you stay within this area — called *bounds checking*.
  - **However, C does not do this.**

- This can lead to some subtle errors where your program attempts to access array elements that don't exist.
  - These errors will likely lead to your program behaving strangely, or very probably crashing.

- This most commonly happens when you forget that array numbering begins at 0.

- Examples for an array declared as:

```
const int SIZE = 20;
int nums[SIZE];


nums[20] = 5;
```

or:

```
for(i = 0; i <= SIZE; i++)
{
     printf("Enter element %d: ", i);
     scanf("%d%*c", &nums[i]);
}
```

Both of these errors occur because the program attempts to access an element one passed the end of the array.

Array Size

- Partly because of this, keeping track of the size of an array is very important.
- This is achieved in two ways.

- Firstly, the size of an array issue should always be defined as a constant; for example, at the top of the program.
    - This constant should then always be used whenever you need to reference the size of the array.
    - For example, when processing the array with a `for` loop.
- This also makes it easy to change the size of the array later on.

- Secondly, when passing an array into a function as a parameter, always pass its size also as the next parameter.
    - This means the size of the array is always available.
    - It also keeps the function independent of where the array size is defined.

    - We will now look at some examples of passing array data as parameters.

Passing Arrays as Parameters

- Because arrays occupy much more memory than single variables, these are always automatically passed by reference without using an &.
- Passing by reference avoids the need to duplicate often huge amounts of data as would happen if passing by value.

- Therefore, when you pass an array into a function, any values you put into the array remain there when you return.
- However, it is not as easy to create an array in a function and get this new array back.

- However, if the array is made up of a simple data type (e.g., `int`) then passing of individual array elements will not be automatically by reference.

Here is some code ent showing the passing of a whole array:

```
void MyFunc (int param[], int size)
{
     param[0] = 20; /*change 1st element to
                        something*/
     return;
}

int main()
{
   int marks[5] = {0};

   printf("Before call marks[0] is %d", marks[0]);
   /* value should be 0 - the default value */

   /* pass single array element as parameter */
   MyFunc(marks, 5);

   printf("\nAfter call marks[0] is %d", marks[0]);
   return(0);
}
```

However, here is an example passing a single array element:

```
void MyFunc (int param)
{
     param = 20; /*change 1st element to
                        something*/
     return;
}

int main()
{
    int marks[5] = {0};

    printf("Before call marks[0] is %d", marks[0]);
    /* value should be 0 - the default value */

    /* pass entire array as parameter */
    MyFunc(marks[0]);

    printf("\nAfter call marks[0] is %d", marks[0]);
    return(0);
}
```

- In this case, the single integer element is passed by value and the change inside the called function only applies to that parameter.
- Therefore, the assignment inside `MyFunc()` has no effect on the contents of the original array.

- This reinforces the fact that individual array elements are treated in exactly the same way as ordinary variables of that type.

- Note that, as far as `MyFunc()` knows, it is receiving only an ordinary integer.
- The fact that this integer comes from an array makes no difference.

Operations on Arrays

- Primitive, built-in data types like `int` have a set of built-in operators that can be used on them.
- For example, if we have two integer variables `a` and `b` then the following statements are valid:

```
sum = a + b;
...

a = b;

...

if(a == b)
  ...
```

- However, in general operators like sum and assignment do not work on complex data types like arrays.
- For example it may not be possible to simply add two arrays together, e.g., if they are different sizes.

- Similar limitations apply to assignment and testing for equality.

- So, in general, most built-in operators *cannot* be used at all on complex data types.

- If you want to add the contents of one array to another, copy the values from one array to another or test two arrays for equality then you have to write the code to do this yourself.

- So be careful not to use built-in operators on complex data types when writing your programs.

COMMON ARRAY ALGORITHMS

Finding the Maximum and Minimum

- A common task that arises is to search through an array and find either the maximum or minimum (or both) values contained within it.

Here is a simple algorithm to implement this:

```
max = array[0]
for i = 1 to array.size
        if array[i] > max
                max = array[i]

print "The largest element in the array is ", max
```

This algorithm can be easily adapted to find the minimum as well.

Here is some code to implement this algorithm:

```
#include <stdio.h>

const int SIZE = 8;

int main()
{
    int array[SIZE] = {-20, 19, 1, 5, -1, 27, 19,
5};
    int max;
    int i;

    /* initialize the current maximum */
    max = array[0];

    /* scan the array */
    for(i = 1; i < SIZE; i++)
    {
      if(array[i] > max)
          max = array[i];
    }

    printf("The maximum of this array is: %d\n",
                                            max);
}
```

Linear Search

- *Linear search* is just a fancy way of saying "start at the beginning and work your way through."

- If you are looking for a specific element in an unsorted array then a linear search is the only way you have of finding that element.

- By iterating through the array until you find what you are looking for you will have to check, at most, the entire array.
- If you are lucky though you may find the target on the first try.
- However, on average you will have to search half of the array.

- This means the time involved in completing the algorithm is proportional to the amount of data being processed.
    - In computer science terminology this means the order of complexity of the algorithm is O(n) (i.e., proportional to the number of elements).

Here is a code fragment to implement a simple linear search:

```
for(i = 0; i < SIZE; i++)
{
   if(array[i] == target)
   {
     found = 1;
     break;
   }
}

if(found)
   printf("Target found at index %d\n", i);
else
   printf("Target was not found.\n");
```

ARRAY SUMMARY

- Arrays are a complex data type that hold many elements of a specific type bundled together with a single variable name.

- However, each element is assigned a number and can be specifically referenced.
- Elements are numbered from 0 up until one less than the size of the array.

- Arrays are extremely useful for processing large amounts of data that would otherwise be very difficult to manage.

- Because *for* loops are specifically designed for iterating a fixed number of times these are very commonly used for processing an entire array.

When defining and processing an array it is very important to be aware of the array's size and ensure that you only access elements that actually exist.

- Because arrays are complex data types most built-in operators like + and = do not work on them.
  - Instead you have to write your own code to process the arrays.

THE WORD ON STRINGS

Introduction

- You may have noticed that so far in this unit we have carefully avoided one particular form of data that is actually *very* common.

- While computer programs deal easily with numbers, because a lot of the data they process relates to humans, they also have to deal with our languages.
  - Strings are the data type for doing this.

- A *string* is simply a sequence or "string" of characters of a given length.
  - Specifically, a string can be zero or more characters in length.

- Although very common, different programming languages handle this quite differently.

- In C strings are usually implemented (more or less) simply as an array of `char`'s.

STORING STRINGS IN C

- Since a string in C is stored as a character array, the length of this array can be defined either dynamically or statically.
    - Again, in this unit, we are only going to work with statically allocated strings for the sake of simplicity.

- This means that you need to decide the maximum number of characters the string will store when writing the program.
-
- However, the actual string you store does not need to use up all of these characters.

- **So the *length* of the string may be less than the actual *size* of the array it is stored in.**

- Because of this, it is necessary to mark where the actual end of the string is within the array.
- C does this using a special character called a *null*.

- The null character is essentially binary 0, has no printable representation and can be represented using the symbol `'\0'`
    - This is similar to new-line characters `'\n'` etc.

- C functions that process strings therefore look for the null character and use this to determine where the end of the string is.
    - We therefore say that strings in C are *null terminated*.

- There are a couple of additional issues in relation to null terminated strings that need to be taken into account.

- Firstly, while functions processing strings can look for the null character to mark the end of the string, unless explicitly given this information, these functions do not know the size of the array in which the string is stored.
    - They will often keep processing until they find the null character.
    - **It is therefore important to make sure that you stay within the bounds of the array.**

- Secondly, when creating a `char` array in order to store a string, you need to allow for storing the null character.

- For example:

```
const int SIZE = 10;
char name[SIZE] = {'\0'};
```

  can only store a name that is a maximum of 9 characters long (NOT 10) because one character needs to be left to store the null.

  Also, if the name being stored is the full 9 characters long, remember the last character in the name will be stored with index 8 (because numbering begins at 0).

0	1	2	3	4	5	6	7	8	9
C	h	r	i	s	t	i	a	n	\0

Finding the Size and Length of Strings

- As just explained, the *length* of a string and the *size* of the array which stores it, although related, are not the same.
  - The *size* refers to the overall capacity of the array, including the null character.
  - The *length* of the string refers to the meaningful characters stored within the array that make up the string, **not** including the null character.

The size of the array containing the string will be whatever it is declared as:

```
const int SIZE = 10;
char name[SIZE] = {'\0'};

printf("Capacity of array is: %d\n", SIZE);
```

Initialise the character array to all null bytes.

This statement will display the number 10.

- However, this only works in the function where the array is declared.
- Inside other functions you should always pass the length of the array as a separate value parameter (as discussed above).

- To calculate the length of the string in the array, you use the `strlen()` function.
  - Note that this **does not** include the null character.
  - We will look at an example of this function shortly.

- Note, initialising the char array to all null characters as in the examples here ensures that your string is always properly terminated (... unless you mess it up!)

Reading Strings

- There are many ways of reading in strings from the keyboard.
- Unfortunately these are all awkward and/or unreliable.
- This is the easiest, safest way.

```
char line[MAXSIZE] = {'\0'};

printf("Enter string: ");
fgets(line, MAXSIZE, stdin);
```

- This code reads in a string and stores it in the variable line.
    - The second parameter gives the size of the array, which prevents too much data being read.
        - Note: if the input string is the same length as this or greater, one character less than this will actually be read and the last character will be made a null.
    - The third parameter specifies that the data is being read from the *standard input*, normally the keyboard.

- While the safest way of reading a string, this technique has one disadvantage.

- When the user presses Enter to indicate the end of the line, this new-line character is also stored in the string variable and must be manually removed.
- This can be achieved with the following code:

```
line[strlen(line) - 1] = '\0';
```

- This stores a null character in the index of the last character in the string, normally the new-line.

Before:

0	1	2	3	4	5	6	7	8	9
T	e	s	t	\n	\0				

After:

0	1	2	3	4	5	6	7	8	9
T	e	s	t	\0	\0				

Here is a program that demonstrates all these concepts.

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

const int MAXSIZE = 10;

int main()
{
    char line[MAXSIZE] = {'\0'};

    printf("Enter string: ");
    fgets(line, MAXSIZE, stdin);

    printf("String entered was :%s:\n", line);

    printf("String length is %d.\n", strlen(line));
    printf("Array defined size is %d.\n", MAXSIZE);

    printf("Removing the new line character...\n\n");
    line[strlen(line) - 1] = '\0';

    printf("String entered was :%s:\n", line);
    printf("String length is %d.\n", strlen(line));

    return(0);
}
```

Copying Strings

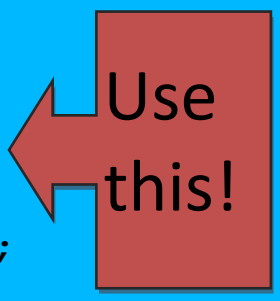
- C provides a range of functions for working with strings.
- We will focus on comparing, copying and concatenating (joining) strings.

- The contents of one string can be copied to a second character array variable using the `strcpy()` function.
- This function takes to parameters and copies of value in the second parameter to the string location specified in the first:

```
strcpy(dst, src);
```

- HOWEVER: you *must* make sure that there is sufficient room in the destination string to store the source string, including the null terminator.
- If there is not enough room then BAD THINGS WILL HAPPEN™

```
if(arraysize >= (strlen(src) + 1))
   strcpy(dst, src);
else
   printf("Not enough room to copy!\n");
```



This code checks that the size of the destination string is big enough to take the contents of the source string plus the null byte.

- Without this, it is possible to overflow the destination string.
- At best, this results in an unreliable program.
- At worst, it can create very serious security problems.

Concatenating Strings

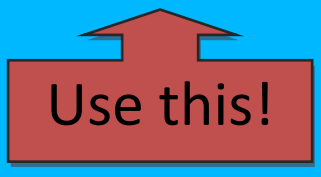
- Concatenating two strings means joining them together.
- This is also very common operation and is done in C using the `strcat()` function.

```
strcat(dst, src);
```

- As with the `strcpy()` function, data is copied from the second parameter to the first.
- However, in this function it is added on to the end of whatever is already there, replacing the existing null byte and null terminating the result where possible.

- HOWEVER: as with the `strcpy()` function, you must make sure that there is sufficient room in the destination string to store the source string and null byte, or otherwise...

```
if((strlen(str1) + strlen(str2) + 1) <=
                                    arraysize)
  strcat(str1, str2);
else
  printf("Strings are too large to join.\n");
```



This code checks that the length of the two strings joined together (including the null byte) is not bigger than the size of the destination string.

Comparing Strings

- Comparing whether two strings are the same is a very common task.
- C uses the `strcmp()` function to do this.

- The function returns a number based on a numerical comparison of the two strings.
- For the purposes of what we are doing in this unit, this means that the return value will be 0 if the two strings are equal or non-zero if they are not.

```
if(strcmp(str1, str2) == 0)
      printf("Strings are the same.\n");
```

The following program demonstrates all of these concepts together.

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

const int SIZE = 50;

int main()
{
  char str1[SIZE] = {'\0'};
  char str2[SIZE] = {'\0'};

  printf("Enter string: ");
  fgets(str1, SIZE, stdin);
  str1[strlen(str1) - 1] = '\0';

  printf("Making a copy of the string...\n");
```

```
    if(SIZE >= (strlen(str1) + 1))
        strcpy(str2, str1);
    else
        printf("Not enough room to copy
         strings!\n\n");


    if(strcmp(str1, str2) == 0)
        printf("Strings are the same.\n\n");
    else
        printf("Strings are not the same - this
                shouldn't happen!\n\n");

    printf("Enter another string: ");
    fgets(str2, SIZE, stdin);
    str2[strlen(str2) - 1] = '\0';

    if(strcmp(str1, str2) == 0)
        printf("Strings are the same.\n\n");
    else
        printf("Strings are not the same.\n\n");

    /*  Note it is very important to check there
    is enough space to do this */
    if((strlen(str1) + strlen(str2) + 1) <=
                                SIZE)
    {
        printf("Joining strings...\n");
        strcat(str1, str2);
        printf("New string is \"%s\"\n", str1);
    }
    else
        printf("Strings are too large to
         join.\n");

    return(0);
}
```

COMMAND LINE ARGUMENTS

- In addition to reading in data while the program is running, programs can also be pre-supplied with data via the command line.
- This means the program can immediately begin doing its job without waiting for input from the user, and can be a very flexible way of interacting with a program.

- A simple example of how to use command line arguments in C is given below:

```
#include <stdio.h>

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

   if(argc == 1)
   {
       printf("No command line arguments
given!\n");
       return(1);
   }

   printf("Arguments are:\n");

   for(i=0; i < argc; i++)
       printf("%d. %s\n", i, argv[i]);

   return(0);
}
```

Examples:

```
cmdline
No command line arguments given!

cmdline abc
Arguments are:
0. cmdline.exe
1. abc

cmdline abc def ghi
Arguments are:
0. cmdline.exe
1. abc
2. def
3. ghi
```

Note, in the examples above, the full path to the program has been omitted which will appear if you run it under Windows.

- Command line arguments are implemented in C using two different parts:
  - `argc`: is the argument counter (how many command line arguments were supplied)
  - `argv`: is an array of strings, each element of which is a separate command line argument.

- Note that the details of how this work are not important for this unit, however you are still expected to be able to use this feature.