Arrays

Problem
Suppose we want to store daily expenses for a three day trip. How would we go about it?

```c
int expenseDay1 = 74;
int expenseDay2 = 80;
int expenseDay3 = 63;
...
prientf("Expense on day 1: %d\n", expenseDay1);
prientf("Expense on day 2: %d\n", expenseDay2);
prientf("Expense on day 3: %d\n", expenseDay3);
```

Disadvantages
- cumbersome, not easy to add extra days
- repetitive, a lot of replicated code
- becomes unfeasible if dealing with a lot of values

Arrays

Solution
We use an array to store a collection of related values.

Definition
A C array is a data structure consisting of an indexed collection of elements with the same type.

Example: an array to store daily expenses:

```c
int dailyExpenses[3] = {74, 80, 63};
```

Closer Look:
- the type of the elements in the array, `int`
- the name of the array, `dailyExpenses`
- the size of the array, `[3]`
- the (optional) initialiser list, = `{74, 80, 63}`

Arrays

Compound Types

Arrays form compound types built from the type of their elements, e.g., `dailyExpenses` has type `array of int`.

Other ways to define arrays:

```c
int dailyExpenses[3];
int dailyExpenses[] = {74, 80, 63};
int dailyExpenses[3] = {74};
```

Each definition creates a `int` array with 3 elements. What are the differences?

Array Size

C arrays are fixed at compile time, they cannot be resized (at run-time once created).

Arrays

Array elements are accessed using an integer index:

```c
int dailyExpenses[3];
dailyExpenses[0] = 74; // set values
dailyExpenses[1] = 80; // of individual
dailyExpenses[2] = 63; // elements
```

Array Indices

Note that arrays are indexed from 0 to `size - 1`. Attempting to access an invalid array index is a run-time error!

Array Size

C does not store information about the size of arrays, so it is the responsibility of the programmer to manage this information.
This code fragment prints the values in the array using a loop:

```c
#define DAYS 3
...  
int i;  
int dailyExpenses[DAYS] = {74, 80, 63};  
for (i = 0; i < DAYS; i = i + 1) {  
  printf("Expense on day %d: %d\n", i + 1, dailyExpenses[i]);  
}
```

We define the constant `DAYS` to eliminate the 'magic numbers' from our code and to help keep track of the array size.

Note the `for` loop counter ranges from 0 to `DAYS − 1`.

---

Any expression evaluating to an integer can be used as index:

```c
int numbers[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};  
int i;  
for (i = 0; 2 * i < 10; i = i + 1) {  
  // print all odd numbers between 1 and 10  
  printf("The number %d is odd.\n", numbers[2*i]);  
}
```

---

Array Size

In this course we will only work with statically allocated arrays, this means that array sizes must be known at compile time, i.e., they must be constants!

---

Copying Arrays

Suppose we have the following:

```c
int arr1[5] = {1, 2, 3, 4, 5};  
int arr2[5];  
```

Can we copy `arr1` into `arr2`?

```c
arr2 = arr1;  
```

**No!** Arrays cannot be copied using the assignment operator!

---

**Solution:** We must copy the elements one by one:

```c
i = 0;  
while (i < 5) {  
  arr2[i] = arr1[i];  
  i = i + 1;  
}
```

**Warning:** We must make sure the array we copy the elements into is big enough!
2D Arrays

Two-dimensional arrays are a simple extension to regular, one-dimensional arrays.

Examples: Cartesian coordinates, tables, spreadsheets, etc.

C supports multi-dimensional arrays, but 2D suffices for our purposes.

Definition

A 2D array is a doubly-indexed structure, literally an array of arrays.

Consider:

```c
int matrix[3][3] = {{1, 2, 3},
                    {4, 5, 6},
                    {7, 8, 9}};

printf("%d\n", matrix[1][1]);
```

Andrew Taylor

COMP1911 Computing IA