Arrays

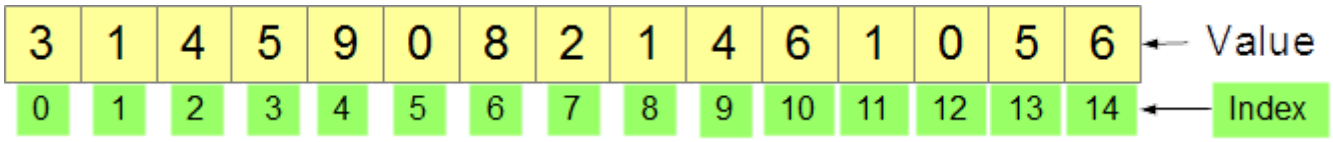
Learning Outcomes

After reading this section, you will be able to:

- Design data collections using arrays to manage information efficiently
- Introduce character strings as terminated collections of byte information

Introduction

Programs can process extremely large amounts of data much faster than well-established manual techniques. Whether this processing is efficient or not depends in large part on how that data is organized. For example, large collections of data can be organized in structures if each variable shares the same type with all other variables and the variables are stored contiguously in memory. Not only can structured data be processed efficiently but the programming of tasks performed on structured data can be simplified considerably. Instead of coding a separate instruction for each variable, we code the instruction that is common to all variables and apply that instruction in an iteration across the data structure.



The simplest data structure in the C language is a list of variables of the same type. We call such a list an array and the variables in that array its elements. We refer to any element by its index.

This chapter introduces the syntax for defining arrays, initializing them and accessing their elements directly. The chapter also demonstrates how to construct a table of values using the concept of a parallel array. This chapter concludes by introducing character strings as arrays with a special terminator.

Definition

An array is a data structure consisting of an ordered set of elements of common type that are stored contiguously in memory. Contiguous storage is storage without any gaps. An array definition takes the form

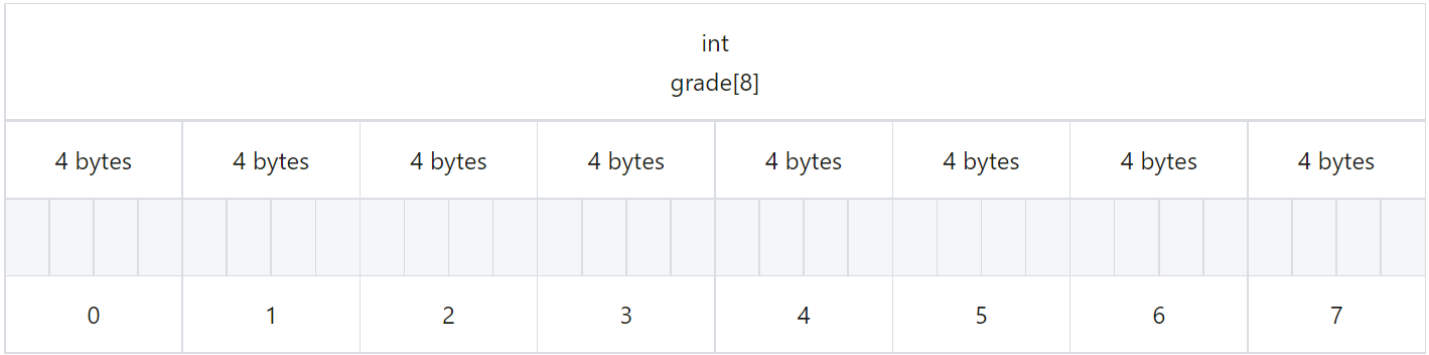
```
type identifier [ size ];
```

`type` is the type of each element, `identifier` is the array's name, the brackets `[ ]` identify the data structure as an array and `size` specifies the number of elements in the array.

For example, to define an array of 8 grades, we write:

```
int grade[8];
```

This statement allocates contiguous storage in RAM for an array named grade that consists of 8 int elements.



We can specify the size of the array using **`#define`** or **`const int`**:

```
#define NGRADES 8  // or const int NGRADES = 8;

int grade[NGRADES];

// ... record the grades

printf("Your last grade was %d\n", grade[NGRADES - 1]);
printf("Your second last grade was %d\n", grade[NGRADES - 2]);
```

This coding style facilitates modifiability. If we change the size, we need to do so in only one place.

Elements

Each element has a unique index and holds a single value. Index numbering starts at 0 and extends to one less than the number of elements in the array. To refer to a specific element, we write the array name followed by bracket notation around the element's index.

```
identifier[index]
```

For example, to access the first element of grade, we write:

```
grade[0]
```

To display all elements of grade, we iterate:

```
for (int i = 0; i < NGRADES; i++)
{
    printf("%d" , grade[i]);
}
```

Check Array Bounds

C compilers do not introduce code that checks whether an element's index is within the bounds of its array. It is our responsibility as programmers to ensure that our code does not include index values that point to elements outside the memory allocated for an array.

Initialization

We can initialize an array when we define it in the same way that we initialize variables. We suffix the declaration with an assignment operator followed by the set of initial values. We enclose the values in the set within a pair of braces and separate them with commas. Initialization takes the form:

```
type identifier[ size ] = { value, ... , value };
```

For example, to initialize grade, we write:

```
int grade[NGRADES] = {10,9,10,8,7,9,8,10};
```

Array Identifier	int grade							
Value	10	9	10	8	7	9	8	10
Element Index	0	1	2	3	4	5	6	7

If our initialization fills all elements in the array, C compilers infer the size of the array from the initialization set and we do not need to specify the size between the brackets. We may simply write:

```
int grade[] = {10,9,10,8,7,9,8,10};
```

If we specify fewer initial values than the size of the array, C compilers fill the uninitialized elements with zero values:

```
int grade[NGRADES] = {0};
```

This will initializes all 8 elements of `grade` to zero.

Specifying a size that is less than the number of initial values generates a syntax error.

Parallel Arrays

A convenient way to store tabular information is through two parallel arrays. One arrays holds the key, while the other hold values. The arrays are parallel because the elements at the same index hold data that are related to the same entity.

In the following example, `sku[i]` holds the stock keeping unit (sku) for a product, while `price[i]` holds its unit price.

```
// Parallel Arrays
// parallel.c

#include <stdio.h>

int main(void)
{
    int i;
    int sku[]      = { 2156, 4633, 3122, 5611};
    double price[] = { 2.34, 7.89, 6.56, 9.32};
    const int n    = 4;

    printf("  SKU Price\n");
    for (i = 0; i < n; i++)
    {
        printf("%5d $%.2lf\n", sku[i], price[i]);
    }

    return 0;
}
```

Output of the above program:

```
  SKU Price
 2156 $2.34
 4633 $7.89
 3122 $6.56
 5611 $9.32
```

The `sku[]` array holds the key data, while the `price[]` array holds the value data. Note how the elements of parallel arrays with the same index make up the fields of a single record of information.

Parallel arrays are simple to process. For example, once we find the index of the element that matches the specified sku, we also have the index of the unit price for that element.

Character Strings

The topic of character strings is covered in depth in the chapter entitled Character Strings. The following section introduces this topic at a high level.

Introduction

A ***string*** is a `char` array with a special property: a **terminator element** follows the last meaningful character in the string. We refer to this terminator as the ***null terminator*** and identify it by the escape sequence `'\0'`.

char																
																\0

The null terminator has the value 0 on any host platform (in its collating sequence). All of its bits are 0's. The null terminator occupies the first position in the ASCII and EBCDIC collating sequences.

The value of the index identifying the null terminator element is the number of meaningful characters in the string.

char **name**																
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	**17**
M	y		n	a	m	e		i	s		A	r	n	o	l	d	***\0***

The number of memory locations occupied by a string is one more than the number of meaningful characters in the string.

Syntax

We need to allocate memory for one additional byte to provide room for the null terminator:

```
const int NCHAR = 17;
char name[NCHAR + 1] = "My Name is Arnold";
```

We use the `"%s"` conversion specifier and the address of the start of the character string to send its contents to standard output:

```
printf("%s", name);
```

Formatting and handling syntax is covered later.