Arrays, Strings, and Structures

Overview

We will begin by working with some of more interesting data structures. These collectively called *containers*.

Because they hold stuff

In this group will find

- Array
  - A group of data of the same type
- String
  - Array of characters
- struct
  - A group of data of the possibly different types
  - The struct permits us to define our own data types
    - Have them be treated like the intrinsic – built in types
- Enumeration
  - A group with only small number of constant values

Arrays

Introducing the Array

Array

- Group of consecutive memory locations
- Organized to hold collection of values
- Single data type

Values are not explicitly named

Accessed by position in the array

Such access called

- Indexing
- Subscripting
Indexes or subscripts represent memory addresses

<table>
<thead>
<tr>
<th>syntax</th>
<th>type identifier [dimension]</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>Specifies the type of elements stored in array</td>
</tr>
<tr>
<td>identifier</td>
<td>Names the array</td>
</tr>
<tr>
<td>dimension</td>
<td>Specifies the number of elements of the named type stored in the array</td>
</tr>
<tr>
<td></td>
<td>dimension always enclosed in [ ]</td>
</tr>
<tr>
<td></td>
<td>• Must be ≥1</td>
</tr>
<tr>
<td></td>
<td>• Computable at compile time</td>
</tr>
<tr>
<td></td>
<td>• Cannot be a variable</td>
</tr>
</tbody>
</table>

**Example**
Declaring – defining an array
```c
int myArray [10]
```

Specifies
- ✓ Space in memory to contain a block of 10 consecutive elements
- ✓ Elements are integers
- ✓ The name of the array is ‘myArray’

**Array elements**
Numbered beginning at 0
Index values in above array a[]
0 - 9
not
1 - 10

**Example**
```c
printf("%d \n", a[i]);
```

Will display the ith element of array a[]
Schematically an array `a[]` in memory looks like

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>

Since each element of `a[]` is an int
- On 32 bit machine
  - Each integer element requires 32 bits of memory
  - Array `a[]` requires 320 bits memory

If `a[10]` is placed into memory address 3000

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a[0]</code></td>
<td></td>
<td></td>
<td></td>
<td>3000 – 30FF</td>
</tr>
<tr>
<td><code>a[1]</code></td>
<td></td>
<td></td>
<td></td>
<td>3100 – 31FF</td>
</tr>
<tr>
<td><code>a[2]</code></td>
<td></td>
<td></td>
<td></td>
<td>3200 – 32FF</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>a[9]</code></td>
<td></td>
<td></td>
<td></td>
<td>3900 – 39FF</td>
</tr>
</tbody>
</table>

Thus
- When accessing `a[5]` actually accessing the 6th element
- Compiler computes – in hex
  - `3000 + 5 \times 32 = 3400`
  - `3000` - starting address
  - `5` - the 6th element
  - `32` - size of 1 element (integer)

Size of an array is determined
- When it is defined
- Cannot be changed later

**Accessing an Array**
- The declaration
  ```
  int aValue0 = myArray[3];
  ```
- Assigns to `aValue0`
  - Value stored in element of `myArray[]`
Indexed by 3
This is the 4th element
Conversely
    myArray[7] = aValue1;

Assigns to the element of myArray[]
Indexed by 7
The value of aValue1

Initializing an Array
The expression
    int myArray[3];

Declares an array of 3 elements
Does not set the elements to any particular value

Compiler allocates 12 bytes of storage
Assuming 32 bit integers

To initialize array
1. Manually assign values

2. Use for or while loop
   Assign value to each element
   Note:

These first two cases not actually an initialization
Rather they are an assignment

3. Specify values as part of declaration
   Specify initializers as comma separated list of values
   Enclosed in {  }

- 4 of 24 -
1. With manual initialization (assignment) we can write following lines of code
   myArray[0] = 4;
   myArray[1] = 5;
   myArray[2] = 6;

2. Using a loop construct we might write
   How would you write the code to do this

3. When initializers specified as part of declaration
   Not necessary to specify dimension

   Can write
   int myArray[] = { 4, 5, 6 };

   With such a declaration
   • If dimension specified
     Number of initializers
     Must not exceed the dimension

   • If number of initializers < dimension
     Remaining elements
     Are not initialized
     Are not guaranteed to be zero

Number of Elements in an Array
We can calculate the number of elements in an array
By using the sizeof operator
Introduced in an earlier lesson

Number of elements in an array is given by

```
sizeof(array) / sizeof(array[0])
```
Pointers and Arrays

Let’s now look at the relationship between pointers and arrays

Recall

[✓] Array is collection of values
[✓] Single data type
[✓] Represented as

If a [ ] is array of integers
Stored in memory as show

By definition in C

The value of a variable of type array
Is the address of the 0th element

For the above array the following are equivalent

&a [ 0 ]
a

Thus

If a [ ] has been loaded at memory address 3000
Then &a[0] = a = 3000

Such equivalence allows us to write

int a[10];  // declare an array of 10 ints
int *pa;  // declare a pointer to an int
pa = a;  // assign the address of a[ ] to the pointer

A bit more surprising although should not be is

a [ j ] is equivalent to

*(a + j)

We know

a + j is the address of the jth element beyond a (the first element)
We have also seen
\[ a[j] = a + j \cdot \text{sizeof (an element of } a) \]

Note:
We do not have to write
\&a + \text{sizeof an element of } a
Since a is the address of the 0th element
That is the address of the jth element beyond a

Example
Have several ways of addressing an array
\begin{verbatim}
int a[] = {1, 2, 3, 4, 5};
int *pa = a;
a[i]
*(pa + i)
*pa++
(*pa)++  // which increments the contents
\end{verbatim}

Let's look at following example

Example
arrayPtr1.c
#include <stdio.h>
int main(void)
{
    int a[] = {1,2,3,4,5};
    int *pa = a;  // pa is a pointer to int

    // print the address of the 0th element
    printf("The value of &a[0] is:  %x\n", &a[0]);

    // print the value of the name of the array
    printf("The value of a is:  %x\n", a);

    // print the value of the 4th element
    printf("The value of a[3] is:  %i\n", a[3]);

    // access the 4th element via pointer and print the value
    printf("The value of a[3] is:  %i\n", *(pa + 3));

    return 0;
}

Example

arrayPtr2.c
#include <stdio.h>
int main(void)
{
    int a[] = {1,2,3,4,5};
    int *pa = a;  // pa is a pointer to int
    int i;
    for (i = 0; i< 5; i++)
        printf("The value of a[i] is:  %i\n", a[i]);
    for (i = 0; i< 5; i++)
        printf("The value of *(pa + i) is:  %i\n", *(pa+i));
    // *p++ increments the pointer and is equivalent to *(pa++)
    for (i = 0; i< 5; i++)
        printf("The value of *pa++ is:  %i\n", *pa++);
    // (*pa)++ increments the contents of the array entry
    for (i = 0, pa = a+4; i< 5; i++)
        printf("The value of *(pa)++ is:  %i\n", *(pa)++);
    return 0;
}

C Style Strings
In C textual data managed using strings

A string is sequence of char integers
With one additional requirement
Last character must have a value of zero
In C we call this a C-style string
Programmers refer to strings as
Terminated by a null character

This null character is simply a char
With a value of zero

Array of char integers
That does not have a terminating zero character
Simply a character array
It is not a string
Example

The following declaration
Creates array of characters
This is not a string

```
char greeting[5] = {'H', 'e', 'l', 'l', 'o'};
```

However following declaration
Does create a string
```
char greeting[5 + 1] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

Observe in this example
The array size is expanded from 5 to 6
But it is coded as 5+1 rather than 6

Good Style

Distinguishing character array from C style string through the container’s dimension…
```
char myString[x + 1]
```
Traditional way of letting other programmers know
✓ This is an array of x char integers
✓ Plus a null char integer

We may use any of several equivalent initialization syntaxes
When we declare a string
Some are easier to read than others
```
char greeting[5 + 1] = {'H', 'e', 'l', 'l', 'o', '\0'};
```
```
char greeting[5 + 1] = "Hello";
```
```
char greeting[] = "Hello";
```
Each of these statements does the same thing

- In the first statement
  String is initialized using C's array initialization syntax
- In the second statement
  String initialization syntax is used
    By definition
    A phrase enclosed between double quotes
    Is a string
    The terminating null is there even if you can't see it
    Notice in the second statement that 5+1 is used for the array size
- In the third statement
  The array size is not specified
  In this case the compiler will
  Count the characters in the string "Hello"
  Add 1 for the terminating null
  To create an array of 6 char integers

Notice strings are arrays
Thus we can write
\[
\text{printf}("%c\n", \text{hello}[1]);
\]
…and we’ll display e on the screen

**Standard String Operations**
There are many standard string operations
In the libraries that come with your compiler

`strlen()`
One handy library function is `strlen()`

\[
\text{syntax}
\begin{align*}
\text{int } & \text{strlen(char* );}
\end{align*}
\]

This function returns the length of the string
\[
\text{printf}("%d\n", \text{strlen("hello")});
\]

Executing this line of code will display 5 on the screen
Not 6
Function `strlen()`
Only returns the number of data characters in the string
Does not count the terminating null

Be careful of this when working with strings
You may find yourself "off-by-1".

`strcat()`
The library function `strcat()`
Accepts
Two strings as parameters
Returns
String with second appended to first

```c
char* strcat(char* dest, char* src);
```

Example
`strcat0.c`

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

int main(void)
{
    // declare the destination string
    char s0[20];
    // set s0 as an empty string
    s0[0] = '\0';
    // build a string
    strcat(s0, "Bonjour ");
    strcat(s0, "Le Monde");
    // print the string: Bonjour Le Monde
    printf("%s \n", s0);

    return 0;
}
```
The library function is `strcmp()`

**Syntax**
```c
int strcmp(char* s0, char* s1);
```

Accepts two strings as input
Lexicographically compares them
Returns an int that is
- Less than 0 if string s0 is less than string s1
- Equal to 0 if string s0 is equal to string s1
- Greater than 0 if string s0 is greater than string s1

**Example**
`strcmp.c`
```c
#include <stdio.h>
#include <string.h>
int main(void)
{
    // declare some strings

    char s0[] = "Hello";
    char s1[] = "World";
    int result = 0;

    // compare the two strings: s0 < s1
    result = strcmp(s0, s1);
    // print the result
    printf("%d \n", result);

    // compare the two strings: s0 == s1
    result = strcmp(s0, s0);
    // print the result
    printf("%d \n", result);

    // compare the two strings: s1 > s0
```
Using Strings as Input

Earlier we used `scanf()` to obtain data from the keyboard.

Remember that if we have a series of numbers separated by spaces, `scanf()` will stop reading
- After first number
- At the first space

To read them all
- We would need to use `scanf()` again
- To get the next number and so on

Let’s consider a simple experiment

Using the following code fragment

```c
int x;
int y;
int z;

scanf("%d", &x);
scanf("%d", &y);
scanf("%d", &z);
```

Let’s enter 45 56 74 from the keyboard
- We will see that the variables take on following values
  - x has value 45
  - y has value 56
  - z has value 74

If change the code fragment the enter the same 3 numbers

```c
char a;
char b;
char c;
```
char d;
char e;
char f;

scanf("%c", &a);
scanf("%c", &b);
scanf("%c", &c);
scanf("%c", &d);
scanf("%c", &e);
scanf("%c", &f);

We will see that the variables take on following values…is this correct
a gets 4 as a character which is 52 as an integer
b gets 5 as a character which is 53 as an integer
c gets 5 as a character which is 53 as an integer
d gets 6 as a character which is 54 as an integer
e gets 7 as a character which is 55 as an integer
f gets 4 as a character which is 52 as an integer

There was no way to get all of the data from the keyboard at once

**Advanced Data Types - Structures**

Examine more sophisticated data types

Contemporary view is data types rather than data structures

**ADT**

An *abstract data type* specifies
- Data type
- Collection of operations
  - On that data type

**Operations**

Specify
Access to data
  - Retrieval
  - Modification
Internal implementation of
Data type
Operations
Hidden from the programmer

Structures
First new type
Structure

Structure
Collection of one or more variables
Perhaps of different types
Permits collection of related items
To be grouped under a single name

First look at simple grouping of data

Point in rectangular co-ordinate system has
• x co-ordinate
• y co-ordinate

Points can be grouped as structure
struct Point
{
    int x;
    int y;
};

Structure
Identified by keyword struct
Optional name
Structure tag
Names the structure
Above - Point
Collection of variables
Called data members
In C when referring to a struct
Must use \textit{struct} keyword

Declaring Instance of a Struct
To declare instance of a struct
Use following
\textit{Syntax}
\begin{itemize}
  \item First
    \textit{Declare} structure of intended struct
    \begin{verbatim}
    struct Point
    {
       int x;
       int y;
    }
    \end{verbatim}
    Important to note
    Declaration above \textit{does not} create instance of named struct
    No memory is allocated to hold such a struct
  
  \item Second
    \textit{Define} the instance of the struct
    \begin{verbatim}
    struct Point pt;
    \end{verbatim}
    Observe
    We had to use \textit{struct} keyword in the definition
\end{itemize}

Initialization
A struct is initialized
Like array
Can following the declaration with a list of initializers
\begin{verbatim}
    struct Point pt = {100, 200};
\end{verbatim}
1 - 1 correspondence
   Between
   Data Members and initializer values

Access
   Members of a struct referred to or accesses two different ways

Direct Access
   When working with instance of struct
   Access with construct of form

   Syntax
   structureName.member;

Given declaration
   struct Point pt;
   Which declares pt
   structure
   type Point

   Can access the x and y members by
   pt.x  // refers to the x variable
   pt.y  // refers to the y variable

Indirect Access
   If we have a pointer to a struct
   Members of a struct pointed to accessed with construct of form

   Syntax
   structurePtr->member;

Notation ‘->’ called arrow
   Composed of two key strokes
   Dash ‘-‘
   Greater than symbol ‘>’
Example

```c
struct Point myPoint = {100, 200};
struct Point* myPointPtr = &myPoint;

myPointPtr->x // refers to the x variable
myPointPtr->y // refers to the y variable

(*myPointPtr).x // refers to the x variable
(*myPointPtr).y // refers to the y variable
```

Nesting

Structures can be nested

Continuing example

Rectangle fully specified

By two diagonally opposite points

Thus can write

```c
struct Rectangle
{
    struct Point pt1;
    struct Point pt2;
};
```

Specifies a structure of type Rectangle

**Observe:**

- Each member line ends in a ;
- Struct itself ends in ;
- Must use keyword struct when specifying struct instances as members of a struct.

Accessing Nested Members

Direct Access

Directly accessing member of nested struct

Simple extension of what we’ve done so far
Given declaration:
struct Rectangle r1;
    r1.pt1.x
    Refers to x co-ordinate of pt1
    r1.pt1.y
    Refers to y co-ordinate of pt1

Example
Observe:
r1 initialized
    With bracketed list of initializers

Cannot
    Initialize with bracket list of points
    Rectangle r1 = {pt1, pt2};

Can assign values
    Point tempPoint = {3, 4};
    r1.pt1 = tempPoint;

Indirect Access
    Indirectly accessing member of nested struct
    Also simple extension of what we’ve done so far
Given declarations:
struct Rectangle r1;
struct Rectangle* rectPtr = &r1;

rectPtr ->(pt1.x);
    Refers to x co-ordinate of pt1
rectPtr ->(pt1.y);
    Refers to y co-ordinate of pt1

Operations
    The only legal operations on a struct
    • Copying
    • Assignment to
        As a unit
    • Taking address
        Using & operator
Structs
Cannot be compared
Operations such as following

    struct Point{...};
    Point pt1, pt2;

    if (pt1 == pt2)
    {
    
    }

are illegal

Enumerations
Enumerations are one of the last new types we will study
An enumeration is a type that represents
Set of integer values
Enumerations defined as follows

    syntax
    enum enumName(enumeration-definition-list);

    enumName
    Identifier name of the enum
    enumeration-definition-list
    Comma-separated list of names
    Each name corresponds to integer value
Put another way
Enumerations associate name
With set of integer values
Enumeration also called

    enum
Declaring Instance of an enum
To declare instance of an enum
Use following
Syntax
• First
  Declare structure of intended enum
  enum CarColor {RED, BLUE, GREEN, SAGE_GREEN};

  Important to note – like struct
  Declaration above does not create instance of named enum
  No memory is allocated to hold such an enum

• Second
  Define the instance of the enum
  enum CarColor myColors;

  Observe
  We had to use enum keyword in the definition

  Alternately could write in one step
  enum CarColor {RED, BLUE, GREEN, SAGE_GREEN} myColors;

Examples

  enum VarType {INT, CHAR, FLOAT, DOUBLE};
  enum Make {FORD, CHEVROLET, CHRYSLER};

A name in an enumeration
  Automatically assigned an integer value
  By default
    Value is assigned number corresponding to
    Position of the name within the enumeration

Like arrays the first position is 0
In the enumerations written above
  RED and FORD
    Have the value zero
  GREEN and Chevrolet have value of 2
Once a name is in an enumeration

We may use the name

In place of the actual number

For instance FORD or BLUE in struct show in following

Example

Given the following struct:

```c
struct Automobile
{
    Make name;
    BodyColor color;
};
```

We can assign the Make and BodyColor
To member elements of the struct

Using the enumeration:

```c
Car myCar;
myCar.name = CHEVROLET;
myCar.color = SAGE_GREEN;
```

and without using it.

```c
Car myCar;
myCar.name = 1;
myCar.color = 3;
```

Which form is more readable

Also can use enumeration

To utilize integer value to represent a string

Such as `SAGE_GREEN`

String “SAGE_GREEN”

Requires 11 bytes

Whereas storing it as an integer value

May require only 4 bytes
Another advantage of using enumerations is
  All members of an enumeration are the same size
  Eliminates overhead of managing strings of differing sizes
  For instance
    When the car is repainted

Further
  Using an enumeration
    Eliminates hard-coded 1's and 3's
    Throughout the program
  Hard-coding of fixed values
    Called magic numbers
    Is a maintenance and readability nightmare

Earlier we saw
  Elements of enumeration
    Assigned consecutive integers
      To elements of enumeration

  We can assign specific values to
    An enumeration or
      To certain members of an enumeration

  Values of the subsequent elements to the right
    Of the one you specifically assigned

  Will be based on your assigned value

For example consider following code fragment

    enum BodyColor {RED, BLUE=4, GREEN, SAGE_GREEN=2};

In this enumeration
  RED is 0,
  BLUE is 4,
  GREEN is 5, and
  SAGE_GREEN is 2
We find the language places no requirement on the assigned values that their values be unique:

```
enum BodyColor {RED, BLUE, GREEN=1, SAGE_GREEN=1};
```

In this enumeration
GREEN and SAGE_GREEN have the same value

A program using this enumeration
Would behave the same
Regardless of whether
GREEN or SAGE_GREEN used in the coding

We need to think carefully
When we duplicate values in an enumeration
We are in effect, saying
That you don't care which name is picked
As long as they have the same value.

**Summary and Review of Objectives**
Following this lesson you should

- Be able to work with several different C containers ...specifically
  - Create and manipulate *arrays*.
  - Create C stype *strings*
  - Use standard string functions
    - To find the lengths of strings
    - To copy strings and
    - To lexicographically compare strings
  - Create your own variable types by using a *struct*
  - Create objects of your structs either singly or in arrays
  - Create and utilize enumerations