Lecture 4: Arrays

Introduction

Some situations require collecting and analyzing large data sets. Creating individual variables for each data entry diminishes readability and increases complexity and falsity. Using an *array*, a collection of objects of the same type, is ideal for tackling such situations.

A Possible Int Array Declaration Of Size 9

r-values:									
l-values:	0x10	0x14	0x18	0x1C	0x20	0x24	0x28	0x2C	0x30
indices:	0	1	2	3	4	5	6	7	8

Specifically, an array is a sequential block of memory with a size equal to the product of its data type length and the number of its elements such that each element's address is the data type length times one less than its position plus the address of the first element.

Array Declaration

The array declaration syntax is the *subscript operator* [also known as the *indexer operator*] (`[]`) with a size between its symbols appended to the end of the identifier of the variable declaration as illustrated below

data-type identifier`[`*size*`];`

where *size* is either an integer type literal or constant variable representing a positive number; an error will be produced if *size* is not positive.

Example:
The following code segment declares several arrays

```
int ages[16]; //an integer array of size 16 named ages
const int n = 22;
double grades[22]; //a double array of size 22 named grade
string names[45]; //a string array of size 45 named names
int a, b[12], c[76]; //an integer variable and two array, respectively
```

Array Assignment

Only a single element of an array can be accessed at a time. The access syntax is

array-name`[`*index*`]`

where *index* is a nonnegative integer between 0 and $n-1$, inclusive, where n is the size of the array. More specifically, an element's index is one less than its position in the array. Using an index outside the mentioned range will cause an error.

Once accessed, elements behave as variables; hence, their content can be accessed when on the right of the assignment operator and content can be assigned to them when they are on the left of the assignment operator as illustrated below

array-name`[`*index*`] = `*argument*`;`

Example:
Using the previous array declaration, the following example provides instances of array access and assignment.

```
ages[0] = 15; //assigns 15 to the first element of ages
for(int i = 1;i < 16;i += 1) {ages[i] = ages[i-1] + 2;}
//loop assigns each element of ages 2 more than its previous
grades[21] = 88; //assigns 88 to the last element of grades
```

Array Initialization

An array can be initialized in one of two ways. One way partially initializes the array while the other fully initializes it. The partial initialization syntax is

data-type identifier`[`*size*`] = {`*argument-list*`};`

where *size* is the array size. The length of the argument list can be at most the array size; otherwise, an error will be produced.

The full initialization syntax is

[`const`]? *data-type identifier*`[] = {`*argument-list*`};`

where the length of the argument list is the array size. Although both versions can be made constant, it is only sensible to make the full version constant since all content is provided.

Finally, character arrays can be initialized using string literals. Such an array is called a *null-terminated character array* (or a *c-string*) since it terminates with the null character, `\0`. A character array initialized in a standard way is a null-terminated character array if its argument list contains the null character. A null-terminated character array is the only array that can display its content using its name but only up to its null character (first null character if it contains more). All other arrays will display the address of its first element if its name is used. Last, using a string literal may cause an error when a partial initialization is used because its null character is counted.

Example:
Some examples of array initializations are as follows.

```
int count[100] = {1,2,3,4,5,6,7,8,9,10}; //partial array initialization
double scores[] = {0.0,0.0,0.0,0.0,0.0}; //full array initialization
char alpha[26] = "abc"; //partial character array initialization
const char unknown[] = "N/A"; //full character array initialization
const cpy[] = {'N','/','A','\0'}; //equivalent to array unknown
```

Array Parameters

Function can also accept array parameters that can only accept array arguments. Its syntax is

[`const`]? *data-type identifier*`[]`

This parameter behaves like a reference parameter in which changes to the parameter are reflected in the caller's argument. To prohibit modifications to the argument, make the parameter constant. It is possible to produce an out-of-bound error when using array parameters since the size of the argument is unknown. Therefore, a function has an integer parameter for each array parameter that holds the size of the array parameter.

Example:
The code segment below provides an example of a function that has an array parameter

```
double sum(const double v[])
{
  return (v[0] + v[1] + v[2] + v[4]);
}

int main()
{
  double a[8] = {1.2,33.2,21.5,7.1,9.0};
  cout << sum(a) << '\n';
  return 0;
}
```

An error will be produced if the argument's size is less than 4.

Multidimensional Arrays

In some scenarios, it will be necessary to represent a grid of values; in other words, it needs an array of arrays known as a *multidimensional array*. To declare an array, the subscript operator with a size needs to be appended to the declaration; hence, to declare a multidimensional array, append additional subscript operators with sizes to the end of the array declaration as below

$$\textit{data-type identifier}\texttt{[}\textit{size}\texttt{]}\left[\texttt{[}\textit{size}\texttt{]}\right]^{+}$$

Anyway, only a single element of a multidimensional array can be accessed and assigned at once. An index for each subscript operator is provided to access an element. Likewise, a multidimensional array can be initialized and used as a parameter; however, all dimensions except for the first must be bounded; otherwise, an error will occur.

Example:
The code segment below provides multidimensional array examples

```
//only accept 3d arrays with leading sizes 7 and 6
int& value(int a[][7][6],int n)
{
  //accesses an element of a
  return a[n][0][0];
}

int main()
{
  int gd[5][16]; //array size 5 of arrays size 16
  double point[8][16][5]; //array of dimensions 8, 16, 5
  int p[][4] = {{1,5,7,8},{7,9,1,5}} //array initialization
  return 0;
}
```

A one-dimensional array can always be used instead of a multidimensional array. The equivalent one-dimensional array must have a size equal to the product of the sizes of the multidimensional array. Next, the formula to transform a multidimensional index tuple to a one-dimensional index is

$$f(i_1, \ldots, i_n) = \sum_{i=1}^{n-1} \prod_{j=i+1}^{n} size_j + i_n$$

where $size_i$ is the size of the ith dimension and each i is a valid index of its dimension.

Example:
A two dimensional array with sizes m and n, respectively, transformation formula to a one-dimensional array will be

$$f(i_1, i_2) = n \times i_1 + i_2$$