Data Structures

Lecture 3

Second Stage

First Course - 2024-2025

ALI MUWAFAQ SHABAN

Master of Computer Engineering

Data Structures

Memory Allocations in Data Structures

Memory allocation is the process of setting aside sections of memory in a program to be used to store variables, and instances of structures and classes.

There are two types of memory allocations possible in C++:

- Compile-time or Static allocation.
- Run-time or Dynamic allocation (using pointers).

Compile-time or Static allocation

.

Static memory allocation allocated by the compiler. Exact size and type of memory must be known at compile time.

int x, y;
float a[5];

When the first statement is encountered, the compiler will allocate two bytes to each variables x and y. The second statement results into the allocction of 20 bytes to the array a (5*4, where there are five elements and each element of float type tales four bytes). Note that as there is no bound checking in C++ for array boundaries, i.e., if you have declared an array of five elements, as above and by mistake you are intending to read more than five values in the array a, it will still work without error.

In Static Memory Allocation the memory for your data is allocated when the program starts. The size is fixed when the program is created. It applies to global variables, file scope variables, and variables qualified with static defined inside functions. This memory allocation is fixed and cannot be changed, i.e. increased or decreased after allocation. So, exact memory requirements must be known in advance.

Array

The simplest type of data structure is a linear array. This is also called one dimensional array. An array holds several values of the same kind. Accessing the elements is very fast. It may not be possible to add more values than defined at the start, without copying all values into a new array. In computer science, an array data structure or simply an array is a data structure consisting of a collection of elements (values or variables), each identified by at least one array index or key. An array is stored so that the position of each element can be computed from its index tuple by a mathematical formula. For example, an array of 10 integer variables, with indices 0 through 9, may be stored as 10 words at memory addresses 2000, 2004, 2008, 2036, so that the element with index i has the address 2000 + 4 × i.

C++ programming language provides a data structure called the array, which can store a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type. Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables. A specific element in an array is accessed by an

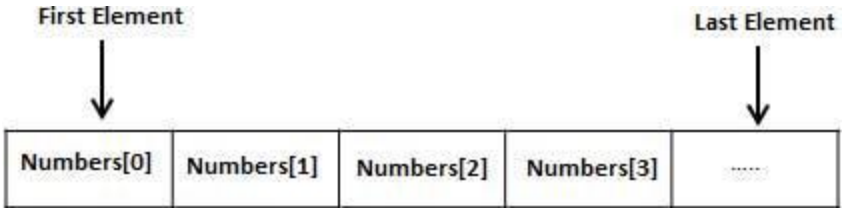
.

index.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.

The array may be categorized into:

- One dimensional array
- Two dimensional array



- Fundamental data structure
- Homogeneous collection of values (all the same type)
- Store values sequentially in memory
- Associate INDEX with each value
- Use array name and index to quickly access k_{th} for any k

Memory address	Index	Content
100	0	h
101	1	e
102	2	l
103	3	L
104	4	o

Representation of 1D array

An array is a data structure which can store multiple homogenous data elements under one name.

Suppose you want to store age of 50 students using a programming language. This can be done by taking 50 different variables for each student. But it is not easy to handle 50 different variables for this. An alternative to this is arrays where we need not declare 50 variables for 50 students. When a list of data item is specified under one name using a single subscript, then such a variable is called a one dimensional(1-D) array. An array is declared in following way:

datatype array-name[size]

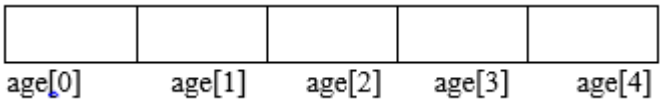
where type specifies type of element such as int, float, char etc .and size indicates maximum number of elements that can be stored inside the array. For

.

example, to represent age of 5 students(in C /C++ language), we can declare it as

int age [5]

An array is represented in memory in contiguous memory location as shown below



In above figure 0,1,2...4 are called index of an array. Index of an array start from 0 to (size-1).

Initialization of 1-D array

Elements of an array can be initialized after an array is declared. An array can be declared at Compile time or at Run time

Compile time initialization

int age[5] = {20,21,19,23,25}

Here first element of age is initialized to 20, second to 21 and so on.

Run time initialization

```
for( int i =0; i<5;i++){
age[i]= 20;
}
```

Here all elements of array age are initialized to 20

- A one-dimensional array is one in which only one subscript specification is needed to specify a particular element of the array.
- A one-dimensional array is a list of related variables. Such lists are common in programming.

Initializing One-Dimensional Array

ANSI C++ allows automatic array variables to be initialized in declaration by constant initializers as we have seen we can do for scalar variables. These initializing expressions must be constant value; expressions with identifiers or function calls may not be used in the initializers. The initializers are specified within braces and separated by commas.

```
int ex[5] = { 10, 5, 15, 20, 25} ;
char word[10] = { 'h', 'e', 'l', 'l', 'o' } ;
```

Example:-

.

// Program to enter the element of Array

```
#include <iostream>
using namespace std;
int main ()
{
    const int MAX_ARRAY = 5;
    int  a[MAX_ARRAY];
    int  index;
      // Ask users to enter values for array a[].
      for (index = 0; index < MAX_ARRAY; index++)
        {
              cout<<"Please input a number for the array element:";
              cin >>a[index];
        }
return 0;
}
```

// Program to print the element of Array

```
#include <iostream>
using namespace std;
int main() {
   int numbers[5];
   cout << "Enter 5 numbers: " << endl;
   //  store input from user to array
   for (int i = 0; i < 5; ++i)
      cin >> numbers[i];
   cout << "The numbers are: ";
   //  print array elements
   for (int n = 0; n < 5; ++n)
      cout << numbers[n] << "  ";
return 0;
}
```

// Program to delete an element from Array

```
#include <iostream>
using namespace std;
int main() {
        int arr[50], size, i, del, count=0;
        cout<<"Enter array size : ";
```

.

```
        cin>>size;
        cout<<"Enter array elements : ";
        for(i=0; i<size; i++)
                cin>>arr[i];
        cout<<"Enter element to be delete : ";
        cin>>del;
        for(i=0; i<size; i++)
        {
                if(arr[i]==del)
                {
                        for(int j=i; j<(size-1); j++)
                                arr[j]=arr[j+1];
                        count++;
                        break;
                }
        }
        if(count==0)
                cout<<"Element not found..!!";
        else
        {
                cout<<"Element deleted successfully..!!\n";
                cout<<"Now the new array is :\n";
                for(i=0; i<(size-1); i++)
                        cout<<arr[i]<<" ";
        }
return 0;
}
```

// Program to find largest element in array

```
#include <iostream>
using namespace std;
int main(){
   int i, n;
   float arr[100];
   cout << "Enter total number of elements: ";
   cin >> n;
   cout << endl;
   // Store number entered by the user
   for(i = 0; i < n; ++i)
   {
```

.

```
    cout << "Enter Number " << i + 1 << " : ";
    cin >> arr[i];
  }
  // Loop to store largest number to arr[0]
  for(i = 1;i < n; ++i)
    // Change < to > if you want to find the smallest element
    if(arr[0] < arr[i])
      arr[0] = arr[i];
  cout << "Largest element = " << arr[0];
return 0;
}
```

Representation of 2D array

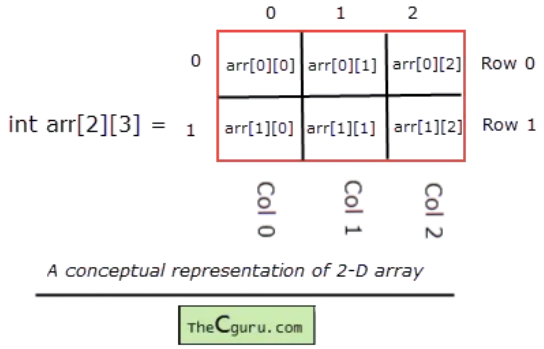
The syntax declaration of 2-D array is not much different from 1-D array. In 2-D array, to declare and access elements of a 2-D array we use 2 subscripts instead of 1.

datatype array_name[ROW][COL];

The total number of elements in a 2-D array is ROW*COL. Let's take an example.

int arr[2][3];

This array can store 2*3=6 elements. You can visualize this 2-D array as a matrix of 2 rows and 3 columns.



The individual elements of the above array can be accessed by using two subscript instead of one. The first subscript denotes row number and second denotes column number. The above image both rows and columns are indexed from 0. So the first element of this array is at arr[0][0] and the last element is at arr[1][2]. Here are how you can access all the other elements:

arr[0][0] - refers to the first element

.

arr[0][1] - refers to the second element
arr[0][2] - refers to the third element
arr[1][0] - refers to the fourth element
arr[1][1] - refers to the fifth element
arr[1][2] - refers to the sixth element

If you try to access an element beyond valid ROW and COL, C++ compiler will not display any kind of error message, instead, a garbage value will be printed. It is the responsibility of the programmer to handle the bounds.

arr[1][3] - a garbage value will be printed, because the last valid index of COL is 2
arr[2][3] - a garbage value will be printed, because the last valid index of ROW and COL is 1 and 2 respectively

Just like 1-D arrays, we can only also use constants and symbolic constants to specify the size of a 2-D array.

```
#define ROW 2
#define COL 3

int i = 4, j = 6;
int arr[ROW][COL]; // OK
int new_arr[i][j]; // ERROR
```

To process elements of a 2-D array, by use two nested loop. The outer for loop to loop through all the rows and inner for loop to loop through all the columns. The following program will clear everything.

```
#include<stdio.h>
#include <iostream>
using namespace std;
#define ROW 3
#define COL 4
int main(){
   int arr[ROW][COL], i, j;
   for(i = 0; i < ROW; i++)
      for(j = 0; j < COL; j++)
      {
         cout<<"Enter arr["<<i<<"]["<<j<<"]: ";
         cin>>arr[i][j];
      }
   cout<<"\nEntered 2-D array is: \n\n";
```

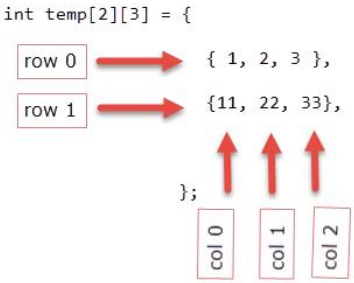
.

```
    for(i = 0; i < ROW; i++)
    {
        for(j = 0; j < COL; j++)
            cout<<arr[i][j]<<"\t";
        cout<<"\n";
    }
    return 0;
}
```

How it works: There is nothing new in this previous program that deserves any explanation. We are just using two nested for loops. The first nested for loop takes input from the user. And the second for loop prints the elements of a 2-D array like a matrix.

Initialization of 2-D array is similar to a 1-D array. For e.g:

```
int temp[2][3] = {
    { 1, 2, 3 }, // row 0
    {11, 22, 33} // row 1
};
```



In 2-D arrays, it is optional to specify the first dimension but the second dimension must always be present. This works only when you are declaring and initializing the array at the same time. For example:

```
int two_d[][3] = {
                    {13,23,34},
                    {15,27,35}
                  };
```

is same as

```
int two_d[2][3] = {
                    {13, 23, 34},
                    {15, 27, 35}
                  };
```

.

As discussed earlier you can visualize a 2-D array as a matrix. The following program demonstrates the addition of two matrices.

```
#include <iostream>
using namespace std;
int main(){
   int r, c, a[100][100], b[100][100], sum[100][100], i, j;
   cout << "Enter number of rows: ";
   cin >> r;
   cout << "Enter number of columns: ";
   cin >> c;
   cout << endl << "Enter elements of 1st matrix: " << endl;
   for(i = 0; i < r; ++i)
     for(j = 0; j < c; ++j)
     {
        cout << "Enter element a" << i + 1 << j + 1 << " : ";
        cin >> a[i][j];
     }
   cout << endl << "Enter elements of 2nd matrix: " << endl;
   for(i = 0; i < r; ++i)
     for(j = 0; j < c; ++j)
     {
        cout << "Enter element b" << i + 1 << j + 1 << " : ";
        cin >> b[i][j];
     }
   // Adding Two matrices
   for(i = 0; i < r; ++i)
      for(j = 0; j < c; ++j)
         sum[i][j] = a[i][j] + b[i][j];
   // Displaying the resultant sum matrix.
   cout << endl << "Sum of two matrix is: " << endl;
   for(i = 0; i < r; ++i)
      for(j = 0; j < c; ++j)
      {
         cout << sum[i][j] << "  ";
         if(j == c - 1)
            cout << endl;
      }
return 0;
}
```

Strings

It is data structure include set of elements of char

char st [10];

The most important function used with string:

strlen(st) return length of string

Ex: write program to read string and compute the numbers in it.

```
#include<iostream>
#include<string.h>
using namespace std;
int main(){
   char st[100],l;
   cout<<"Enter st: "<<endl;
   cin>>st;
   l=strlen(st);
   int c=0;
   for(int i=0;i<l;i++)
      if((st[i]>='0')&&(st[i]<='9'))
         c++;
   cout<<"The program find "<< c << " number(s) in string";
return 0;
}
```

Ex2: write program to print the middle character in string.

```
#include<iostream>
#include<string.h>
using namespace std;
int main(){
   int i,l;
   char st[30];
   cout<<"enter string: " ;
   cin>>st;
   l=strlen(st);
   i=l/2;
   cout<<st[i];
return 0;
}
```

.

Applications of array

Arrays are used to implement mathematical vectors and matrices, as well as other kinds of rectangular tables. Many databases, small and large, consist of (or include) one-dimensional arrays whose elements are records. Arrays are used to implement other data structures, such as heaps, hash tables, deques, queues, stacks, strings, and VLists. One or more large arrays are sometimes used to emulate in-program dynamic memory allocation, particularly memory pool allocation. Historically, this has sometimes been the only way to allocate "dynamic memory" portably. Arrays can be used to determine partial or complete control flow in programs, as a compact alternative to (otherwise repetitive) multiple IF statements. They are known in this context as control tables and are used in conjunction with a purpose built interpreter whose control flow is altered according to values contained in the array.

.

References:

- Frank Carrano, D.J. Henry: Data Abstraction and Solving with C++, 2012, 6th edition, Pearson Education, Inc.
- Mark Allen Weiss: Data Structures and Algorithm Analysis in C++, 2014, 4th edition, Pearson Education, Inc.