COLLAGE OF COMPUTING AND INFORMATICS

DEPARTMENT OF INFORMATION TECHNOLOGY

INDIVIDUAL ASSIGNMENT OF FUNDAMENTALS OF PROGRAMING II

NO	NAME	ID
1	***YOSSIEF ENYEW WONDIE***	**3734/14**

Instructor :Mr . Brhanu G.

Submitted date:24/03/2016

INTRODUCTION

In this assignment is tray to explain about array ,pointer,function and Memory Allocation.

Array is a linear data structure that is a collection of similar data types. Arrays are stored in contiguous memory locations. It is a static data structure with a fixed size. It combines data of similar types.also contain about accessing of array. In any point of the program in which the array is visible we can access individually anyone of its elements for reading or modifying it as if it was a normal variable

Also tray to explain about one , two and multi dimensional array.Compare and contrast one-dimensional arrays, two-dimensional arrays, and multidimensional arrays, how To declare a one,two and multi-dimensional array in the second portion of assignment.

Also tray to explain about dynamic arrays differ from static arrays, the benefits of dynamic memory allocation, and analyze the trade-offs involved in using dynamic arrays. In third portion of assignment.

Explain the concept of pointers in C++ programming and their role in memory management .in forth portion of assignment.

In the last part contain about functions . pass-by-value and pass-by-reference parameter passing mechanisms in programming

TABLE OF CONTENT

1. Discuss the fundamental concepts and applications of arrays in C++ programming. Provide examples of real-world scenarios where arrays are commonly used and explain how they contribute to efficient data manipulation?

1.1 *Fundamental concepts of arrays in C++*

Array is a linear data structure that is a collection of similar data types. Arrays are stored in contiguous memory locations. It is a static data structure with a fixed size. It combines data of similar types.

. An array must be defined just like any other object. The definition includes the array

name and the type and number of array elements.

Syntax: type name[count]; // Array name

• An array is a series of elements of the same type placed in a contiguous memory locations.

• An individual element of an array is identified by its own unique index (or subscript)

• Instead of declaring individual variables to record all the student's mark, such as int mark0, int mark1, ...,

up to int mark7, you can declare one array variable for all given data that are in the same type like:-

float mark[7];

• This declaration will cause the compiler to allocate space for 7 consecutive float variables in memory.

1.2 Initializing Arrays

Initialization List Arrays can be initialized when you define them. A list containing the values for the individual array elements is used to initialize the array:

Example: int num[3] = { 30, 50, 80 };

A value of 30 is assigned to num[0], 50 to num[1], and 80 to num[2]. If you initialize an array when you define it, you do not need to state its length.

Example: int num[] = { 30, 50, 80 };

In this case, the length of the array is equal to the number of initial values. If the array length is explicitly stated in the definition and is larger than the number of initial values, any remaining array elements are set to zero. If, in contrast, the number of initial values exceeds the array length, the surplus values are ignored.Locally defined arrays are created on the stack at program runtime. You should there fore be aware of the following issues when defining arrays:

■ Arrays that occupy a large amount of memory (e.g., more than one kbyte) should be defined as global or static.

■ Unless they are initialized, the elements of a local array will not necessarily have a definite value. Values are normally assigned by means of a loop.You cannot assign a vector to another vector. However, you can overload the assignment operator within a class designed to represent arrays.

int day [] = { 1, 2, 7, 4, 12,9 };
The compiler will count the number of initialization items which is 6 and set the size of the array day to 6 (i.e.: day[6])
You can use the initialization form only when defining the array. You cannot use it later, and cannot assign one array to another once. I.e.

```
int arr [] = {16, 2, 77, 40, 12071};
int ar [4];
ar[]={1,2,3,4};//not allowed
arr=ar;//not allowed
```

* Note: when initializing an array, we can provide fewer values than the array elements. E.g. int a [10] = {10, 2, 3}; in this case the compiler sets the remaining elements to zero.

1.3 Accessing and processing array elements

In any point of the program in which the array is visible we can access individually anyone of its elements for reading or modifying it as if it was a normal variable. To access individual elements, index or subscript is used. The format is the following:

name [index]

In c++ the first element has an index of 0 and the last element has an index, which is one less the size of the array (i.e. arraysize-1).

int day [] = { 1, 2, 7, 4, 12,9 };

Thus, from the above declaration, day[0] is the first element and day[4] is the last element.

day[3]=4

Notice that the third element of day is specified day[2] , since first is day[0] , second day[1] , and therefore, third is day[2] . By this same reason, its last element is day [4]. Since if we wrote day [5], we would be acceding to the sixth element of day and therefore exceeding the size of the array. This might give you either error or unexpected value depending on the compiler.

At this point it is important to be able to clearly distinguish between the

two uses the square brackets [ ] have for arrays.

o One is to set the size of arrays during declaration

o The other is to specify indices for a specific array element when

accessing the elements of the array

We must take care of not confusing these two possible uses of brackets [ ]

with arrays:

Eg: int day[5]; // declaration of a new Array (begins with a type name)

day[2] = 75; // access to an element of the Array.

Other valid operations with arrays in accessing and assigning:

int a=1;

day [0] = a;

day[a] = 5;

b = day [a+2];

day [day[a]] = day [2] + 5;

day [day[a]] = day[2] + 5;

array inter user in the kibord using for loop

```
#include <iostream>

using namespace std;

int main()

{

const int MAXCNT = 10; // Constant

float arr[MAXCNT], x; // Array, temp. variable

int i, cnt; // Index, quantity

cout << "Enter up to 10 numbers \n"<< "(Quit with a letter):" << endl;

for( i = 0; i < MAXCNT && cin >> x; ++i)

arr[i] = x;

cnt = i;

cout << "The given numbers:\n" << endl;

for( i = 0; i < cnt; ++i)

cout << setw(10) << arr[i]

cout << endl;

return 0;

}
```

1.4 applications of arrays in C++ programming

Below are some applications of arrays.

Storing and accessing data: Arrays are used to store and retrieve data in a specific order. For example, an array can be used to store the

scores of a group of students, or the temperatures recorded by a weather station.

Sorting: Arrays can be used to sort data in ascending or descending order. Sorting algorithms such as bubble sort, merge sort, and quick sort rely heavily on arrays.

Searching: Arrays can be searched for specific elements using algorithms such as linear search and binary search.

Matrices: Arrays are used to represent matrices in mathematical computations such as matrix multiplication, linear algebra, and image processing.

Stacks and queues: Arrays are used as the underlying data structure for implementing stacks and queues, which are commonly used in algorithms and data structures.

Graphs: Arrays can be used to represent graphs in computer science. Each element in the array represents a node in the graph, and the relationships between the nodes are represented by the values stored in the array.

Dynamic programming: Dynamic programming algorithms often use arrays to store intermediate results of sub problems in order to solve a larger problem.

1.5 Real-Time Applications of Array:

Below are some real-time applications of arrays.

Signal Processing: Arrays are used in signal processing to represent a set of samples that are collected over time. This can be used in applications such as speech recognition, image processing, and radar systems.

Multimedia Applications: Arrays are used in multimedia applications such as video and audio processing, where they are used to store the pixel or audio samples. For example, an array can be used to store the RGB values of an image.

Data Mining: Arrays are used in data mining applications to represent large datasets. This allows for efficient data access and processing, which is important in real-time applications.

Robotics: Arrays are used in robotics to represent the position and orientation of objects in 3D space. This can be used in applications such as motion planning and object recognition.

Real-time Monitoring and Control Systems: Arrays are used in real-time monitoring and control systems to store sensor data and control signals. This allows for real-time processing and decision-making, which is important in applications such as industrial automation and aerospace systems.

Financial Analysis: Arrays are used in financial analysis to store historical stock prices and other financial data. This allows for efficient data access and analysis, which is important in real-time trading systems.

Scientific Computing: Arrays are used in scientific computing to represent numerical data, such as measurements from experiments and simulations. This allows for efficient data processing and visualization, which is important in real-time scientific analysis and experimentation.

2. Compare and contrast one-dimensional arrays, two-dimensional arrays, and multidimensional arrays, highlighting their respective advantages and use cases. Illustrate your answer with code snippets and practical examples in C++ programming

2.1 Multi-Dimensional Arrays

A multi-dimensional array is an array of arrays.

To declare a multi-dimensional array, define the variable type, specify the name of the array followed by square brackets which specify how many elements the main array has, followed by another set of square brackets which indicates how many elements the sub-arrays have:

```
string letters[2][4];
```

As with ordinary arrays, you can insert values with an array literal - a comma-separated list inside curly braces. In a multi-dimensional array, each element in an array literal is another array literal.

```
string letters[2][4] = {
 { "A", "B", "C", "D" },
 { "E", "F", "G", "H" }
};
```

Arrays can have any number of dimensions. The more dimensions an array has, the more complex the code becomes. The following array has three dimensions:

```
string letters[2][2][2] = {
 {
  { "A", "B" },
  { "C", "D" }
 },
 {
```

```
  { "E", "F" },
  { "G", "H" }
 }
};
```

EXAMPLE

Accessing of E is letter[0][0][1]

2.2 Two Dimensional (2D) Array

- In a first (left) subscript as being the row, and

two dimensional array, it is convenient to think of the second (right) subscript as being the column.

- To access the elements of a two-dimensional array, simply use two subscripts:

- array[row][col]

2.2.1 Initializing a 2D Array

1.use nested braces, with each set of number representing a row:

```
array[3][5] ={{ 1, 2, 3, 4, 5 }, // row 0
{6, 7, 8, 9, 10 }, // row 1
{11, 12, 13, 14, 15 } // row 2};
```

2. We can also initializes this array as:

int array[3][5]={1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};

2.2.2 Omitting the Size of two 2D Array

• Two-dimensional arrays with initializer lists can omit (only) the

leftmost length specification:

int array[][5] ={{ 1, 2, 3, 4, 5 },{ 6, 7, 8, 9, 10},

{ 11, 12, 13,14, 15 }};

• The compiler can do the math to figure out what the array length is.

• However, the following is not allowed:

int array[ ][ ] = { 1, 2, 3, 4, 5, 6, 7, 8 };

• Because the inner parenthesis are ignored, the compiler can not tell whether you intend to declare a 1×8, 2×4, 4×2, or 8×1 array in this case.

• If a one-dimensional array is initialized, the size can be omitted as it can be found from the number of initializing elements:

int x[] = { 1, 2, 3, 4}; This initialization creates an array of

four elements

2.2.3 Accessing Elements in a 2D Array

• Accessing all of the elements of a two-dimensional array requires two loops: one for the row, and one for the column.

• Since two-dimensional arrays are typically accessed row by row, the row index is typically used as the outer loop.

```
for (int row=0;row<index1;row++) //loop for rows

{

for( into col=0; col<index2,col++) //columns

{

cout<< array[row][col]<<“ “;

}

cout<<endl;// to make space between matrixes

}
```

Multi-dimensional arrays are useful for representing three-dimensional spaces, image data with multiple color channels, and other complex data structures that require more than two dimensions.

In summary, one-dimensional arrays are simple arrays with a single dimension. Two-dimensional arrays are arrays with two dimensions, commonly used for representing grids and matrices. Multi-dimensional arrays have more than two dimensions and are used for representing complex data structures in higher-dimensional spaces.

3. Explain how dynamic arrays differ from static arrays, discuss the benefits of dynamic memory allocation, and analyze the trade-offs involved in using dynamic arrays?

Memory allocation is a fundamental aspect of programming that involves allocating and managing memory in a program. When it comes to dynamic arrays, there are important trade-offs to consider. Let's explore memory allocation and the trade-offs involved in using dynamic arrays in more detail.

3.1 Memory Allocation:

in programming, memory allocation refers to the process of reserving a portion of computer memory for the execution of programs and processes . There are two basic types of memory allocation:

3.1.1 Static Memory Allocation:

This is done at compile time, where the memory for variables, structures, and classes is allocated by the operating system. The size and location of the memory are determined before the program runs.

3.1.2 Dynamic Memory Allocation:

This is done at run time, where memory is allocated and deallocated as needed using functions like malloc(), calloc(), realloc(), or the new operator. With dynamic memory allocation, you can designate a block of memory of the appropriate size while the program is running

3.1.3 Trade-offs with Dynamic Arrays:

Dynamic arrays offer flexibility and scalability compared to fixed-size arrays. However, they also come with certain trade-offs:

3.2 Memory Overhead:

Dynamic arrays require additional memory to store information about the size, capacity, and address of the dynamically allocated block. This overhead can be significant when dealing with small arrays

3.3 .Time Complexity:

Dynamic arrays offer constant-time access to elements, as they can be indexed directly. However, resizing a dynamic array requires allocating new memory, copying existing elements, and deallocating the old memory block. This process takes O(n) time, where n is the number of elements in the array. Resizing frequently can impact performance

3.4 .Fragmentation:

Dynamic arrays can suffer from fragmentation, where memory becomes divided into small, non-contiguous blocks over time. Fragmentation affects memory utilization and can lead to inefficient memory usage

3.5 .Error Handling:

Dynamic memory allocation introduces the possibility of memory leaks or dangling pointers if not managed properly. It is essential to free dynamically allocated memory when it is no longer needed to avoid memory leaks

Overall, dynamic arrays provide flexibility and scalability but come with the trade-offs of memory overhead, time complexity for resizing, fragmentation, and the need for careful memory management. Considering these trade-offs, it is important to carefully analyze the requirements of your program and choose memory allocation strategies accordingly.

4. Explain the concept of pointers in C++ programming and discuss their role in memory management . Explore how pointers are used to store memory addresses and facilitate efficient data manipulation in C++ programming languages. Provide examples of pointer operations and discuss their benefits and potential risks.

4.1 A pointer

is a variable that stores the memory address as its value.

A pointer variable points to a data type (like int or string) of the same type, and is created with the * operator.
In C++, pointers are variables that store the memory address of another variable. They are used to manipulate the data in the computer's memory, which can reduce code and improve performance.

Pointers are particularly useful for implementing data structures like lists, trees, and graphs.

To create a pointer, you use the * operator in the variable declaration. For example, if you have an integer variable myAge, you can create a pointer to it like this:

int* ptr = &myAge;

. In this example, ptr is a pointer variable that points to the memory address of myAge.

4.2 Dereferencing

a pointer means accessing the value of the variable it points to. You can use the * operator to dereference a pointer. For example, if you have a pointer ptr, you can access the value it points to using *ptr.

Pointers in C ++ can be classified into different types based on what they are pointing to. Some common types of pointers include integer pointers, array pointers, structure pointers, function pointers, double pointers, null pointers, and void pointers.

There are several important concepts to understand when working with pointers in C++, such as pointer arithmetic, pointer to pointer, and dynamic memory allocation.

When using pointers, it is important to handle them with care to avoid damaging data stored in other memory addresses.

Overall, pointers in C++ are a powerful feature that allows for more efficient memory manipulation and implementation of complex data structures. They can be a bit challenging to understand at first, but with practice and understanding of the underlying concepts, you can effectively utilize pointers in your C++ programs.

int* p; // Declare a pointer to an integer

int num = 5;

p = # // Assign the address of num to the pointer p

*p = 10; // Update the value of num using the pointer

In this example, p stores the address of the num variable. By dereferencing the pointer using the * symbol, we can access and manipulate the value stored at that memory location.

The statement *p = 10; changes the value of num to 10.

4.3 role of pointers in memory management in C++

Pointers play a crucial role in memory management in C++. They allow for efficient memory usage and manipulation, enabling you to optimize your code and create dynamic data structures.

Here are some key aspects to understand about the role of pointers in memory management:

Allocating and Deallocating Memory: Pointers enable dynamic memory allocation and deallocation, allowing you to allocate memory for variables or data structures at runtime. The C library functions malloc() and calloc() are commonly used for dynamic memory allocation, while free() is used to deallocate memory.

For example, you can allocate memory for an integer variable using malloc(sizeof(int)) and deallocate it using free(p). It's important to manage allocated memory properly to avoid memory leaks and undefined behavior.

Pointers and Arrays: Pointers and arrays are closely related in C++. Arrays are implemented as pointers to their first elements, and pointer arithmetic allows for efficient traversal and manipulation of array elements. When an array is passed to a function, it is actually passed as a pointer to its first element. Understanding the relationship between pointers and arrays is essential for efficient array manipulation.

Pointers and Strings: In C++, strings are typically represented as arrays of characters, and pointers are often used to manipulate strings. String literals in C++ are also pointers to the first character of the string. Pointers can be used to perform various operations on strings, such as concatenation and comparison.

Pointer Arithmetic: Pointer arithmetic involves performing mathematical operations on pointers, such as addition, subtraction, and incrementing/decrementing pointers. Pointer arithmetic is particularly useful when iterating through arrays or modifying data structures.

Dynamic Memory Allocation: Pointers enable dynamic memory allocation, which is advantageous when working with unknown or variable amounts of data. Functions like malloc() and calloc() allow you to allocate memory at runtime, and managing dynamically allocated memory properly is crucial to prevent memory leaks.

By understanding how to use and manage pointers effectively, you can optimize memory usage, create efficient data structures, and write robust and performant C++ code. It's important to practice good memory management practices and ensure that allocated memory is deallocated when it is no longer needed.

4.4 how pointers are used in C++ programming

Pointers are a fundamental concept in C++ programming. They provide a way to access and manipulate memory directly, which can be very useful in many programming scenarios.

In C++, a pointer is a variable that holds a memory address. It allows you to refer to a specific location in memory and access or modify the value stored at that location.

To declare a pointer variable in C++, you use the asterisk (*) symbol before the variable name. For example, int* p; declares a pointer variable named \"p\" that can hold the memory address of an integer.

To initialize a pointer to point to a specific variable or memory location, you use the ampersand (&) operator to get the address of that variable. For example, int x = 42; int* p = &x; initializes the pointer \"p\" to point to the memory address of integer variable \"x\".

Once you have a pointer that points to a specific memory location, you can access or modify the value stored at that location by dereferencing the pointer. To dereference a pointer, you use the asterisk (*) symbol in front of the pointer variable itself. For example, *p accesses the value stored at the memory location pointed by \"p\".

Pointers can also be used for dynamic memory allocation in C++. This allows you to allocate memory at runtime, rather than at compile time. You use the new keyword to dynamically allocate memory, and it returns a pointer to the allocated memory. You should free the allocated memory using the delete keyword to avoid memory leaks.

Pointers can also be passed to functions as arguments, allowing the function to modify the value of the original variable passed in. This is known as \"passing by reference\" and is a powerful feature of C++.

Overall, understanding how pointers work in C++ is essential for efficient memory management and accessing data stored in memory locations. With practice and experience, you will become comfortable using pointers in your C++ programs.

Pointer operations in C allow us to manipulate memory addresses and perform various arithmetic operations. Here are some examples of pointer operations along with their benefits and potential risks:

Incrementing Pointer: By incrementing a pointer, we can make it point to the next memory location. This is useful when traversing an array or linked list. The benefit of incrementing a pointer is that it provides a convenient way to access consecutive elements in memory. However, a potential risk is incrementing the

pointer beyond the bounds of the allocated memory, leading to undefined behavior.

Decrementing Pointer: Similar to incrementing, decrementing a pointer allows us to move it to the previous memory location. This is useful when traversing an array or linked list in reverse order. The benefit is that it provides a straightforward way to access elements in reverse order. However, like incrementing, decrementing beyond the allocated memory can lead to undefined behavior.

Pointer Addition: Adding a value to a pointer allows us to move it by a certain number of elements. This is helpful when accessing specific elements in an array or when implementing algorithms that require pointer manipulation. The benefit is that it provides a flexible way to navigate through memory. However, incorrect handling of pointer arithmetic can lead to accessing invalid memory locations or introducing bugs in the code.

Pointer Subtraction: Subtracting a value from a pointer allows us to move it backward by a certain number of elements. This can be useful when working with arrays or when calculating the distance between two pointers. The benefit is that it provides a way to determine the relative position of two memory addresses. However, incorrect usage of pointer subtraction can result in accessing invalid memory locations or incorrect calculations.

Overall, pointer operations in C provide a powerful tool for memory manipulation and efficient data access. However, they require careful handling to avoid potential risks such as accessing invalid memory locations, corrupting memory, or introducing bugs in the code. It is important to thoroughly test and validate code that involves pointer operations to ensure correct and safe behavior.

5.Compare and contrast pass-by-value and pass-by-reference parameter

passing mechanisms in programming. Discuss the role of pointers in implementing pass-by-reference and explain how they enable functions to modify variables in the calling context. Provide code examples to support your explanation.

5.1 Modules in C++ are called functions

- Function - a subprogram that can act on data and return a value
- Every C++ program has at least one function, main(), where program execution begins
- A C++ program might contain more than one function.
- Functions may interact using function call

A function is a block of code which only runs when it is called.

Functions are used to perform certain actions, and they are important for reusing code: Define the code once, and use it many times.

5.2 function pass

You can pass data, known as parameters, into a function.

Ther is two types of functions calls this are

pass by value and

pass by reference

5.2.1 pass by value

Value of the function argument passed to the formal parameter of the function

– Copy of data passed to function

– Changes to copy do not change original

• Call by Reference

– Address of the function argument passed to the

formal parameter of the function

In pass by value, a copy of the parameter's value is made in memory . This approach is suitable for multi-threaded applications and distributed applications, as it prevents objects from being modified by other threads and saves overhead in network synchronization.

```
/* Incorrect function to switch two values */
void swap(int a, int b)
{
int hold;
hold = a;
a = b;
b = hold;
return;
}
```

5.2.2 pass by reference

If a formal parameter is a reference parameter

– It receives the address of the corresponding actual parameter

– A reference parameter stores the address of the corresponding

actual parameter

• During program execution to manipulate the data

– The address stored in the reference parameter directs it to the

memory space of the corresponding actual parameter

– A reference parameter receives the address of the actual

parameter

• Reference parameters can:

– Pass one or more values from a function

– Change the value of the actual parameter

Reference parameters are useful in three situations:

– Returning more than one value

– Changing the actual parameter

– When passing the address would save memory space and time

```
/* Correct function to switch two values */
void swap2(int& a, int& b)
{
int hold;
hold = a;
a = b;
b = hold;
return;
}
```

Pass by reference, does not create a new copy of the variable, saving the overhead of copying. This method is more efficient, especially when passing large objects such as structs or classes.

The main difference between pass-by-value and pass-by-reference is the way the actual parameters are passed to a function. In pass by value, a copy of the function is made, while in pass by reference, the actual parameters are passed directly to the function .

CONCLUSION

In summary, one-dimensional arrays are simple arrays with a single dimension. Two-dimensional arrays are arrays with two dimensions, commonly used for representing grids and matrices. Multi-dimensional arrays have more than two dimensions and are used for representing complex data structures in higher-dimensional spaces.

Overall, pointer operations in C provide a powerful tool for memory manipulation and efficient data access. However, they require careful handling to avoid potential risks such as accessing invalid memory locations, corrupting memory, or introducing bugs in the code. It is important to thoroughly test and validate code that involves pointer operations to ensure correct and safe behavior.

overall, dynamic arrays provide flexibility and scalability but come with the trade-offs of memory overhead, time complexity for resizing, fragmentation, and the need for careful memory management. Considering these trade-offs, it is important to carefully analyze the requirements of your program and choose memory allocation strategies accordingly.

The main difference between pass-by-value and pass-by-reference is the way the actual parameters are passed to a function. In pass by value, a copy of the function is made, while in pass by reference, the actual parameters are passed directly to the function.

SOURCE :

Google

w3school

Different ppt