Computer science notes assignment

All these databases are defined within the C++ compiler and that s why they are also called as primitive. We can define a length using into , a weight using float , a name using characters etc. But suppose I tell you “ please define a fruit for me in program” , then your mind starts wondering, as you can’t define a fruit Just with any one database as you did with length , weight etc. A Fruit itself is a composite entity having following attributes : color : can be described with a name I. E. Char [ ] taste : can be described with a name I. E. Chart] season : can be described with into I. E. For summer , 2 for winter price : can be described as float mans so on… This means that to describe a fruit we need to have a collection of data-types bundled together so that all the attributes of the fruit can be captured. This is true for any real world thing around you say student , mobile , plant etc. So when we bundle many primitive data types together to define a real world thing then it is known as derived data type or structured data type or User defined data types. In this chapter we would look onto two important structured data types , one is Array and the other one is Structure.

Sometimes we need to have variables in very large quantities , that too of same data yep I. E. Suppose we want 200 integer variables. In this situation will you declare 200 individual variables ? Absolutely not. This is because it will : a) wastage of time as well time taking task b) we won’t be able to manage these 200 variables in our program , it will be difficult to remember the names of variable every now and then during programming. So there exist special structured data type in C++ to tackle this situation. C++ allows a programmer to bundle together all these same type of 200 variable under a same tag name called as Arrays.

So we are observing that structuring data type means undying primitive data type in some or other way so that it solves some special programming situations. 93 4. 1 Arrays An array is a series of elements of the same type placed in contiguous memory locations that can be individually referenced by adding an index to a unique identifier. That means that, for example, we can store 5 values of type into in an array without having to declare 5 different variables, each one with a different identifier. Instead of that, using an array we can store 5 different values of the same type, into for example, with a unique identifier.

For example, an array to contain 5 integer values of yep into called marry could be represented like this: marry 2 3 4 where each blank panel represents an element of the array, that in this case are integer values of type into. These elements are numbered from O to 4 since in arrays the first index is always O, independently of its length. Like a regular variable, an array must be declared before it is used. A typical declaration for an array in C++ is: Syntax : array_name [elements]; where database is a valid type (like into, float... , name is a valid identifier and the elements field (which is always enclosed in square brackets [l), specifies how many f these elements the array has to contain. Therefore, in order to declare an array called marry as the one shown in the above diagram it is as simple as: into marry [5]; NOTE: The elements field within brackets [ ] which represents the number of elements the array is going to hold, must be a constant value, since arrays are blocks of non-dynamic memory whose size must be determined before execution.

In order to create arrays with a variable length dynamic memory is needed, which is explained later in these tutorials. Initializing arrays. When declaring a regular array of local scope (within a function, for example), if we o not specify otherwise, its elements will not be initialized to any value by default, so their content will be undetermined until we store some value in them. The elements of global and static arrays, on the other hand, are automatically initialized with their default values, which for all fundamental types this means they are filled with zeros.

In both cases, local and global, when we declare an array, we have the possibility to assign initial values to each one of its elements by enclosing the values in braces { }. For example: into Aryan [5] = { 16, 2, 77, 40, 12071 This declaration would have created an array like this: The amount of values between braces { } must not be larger than the number of elements that we declare for the array between square brackets [For example, in the example of array marry we have declared that it has 5 elements and in the list of initial values within braces { } we have specified 5 values, one for each element.

When an initialization of values is provided for an array, C++ allows the possibility of leaving the square brackets empty C]. In this case, the compiler will assume a size for the array that matches the number of values included between braces { }: NT Aryan [ ] = { 16, z 77, 40, 12071 After this declaration, array marry would be 5 nits long, since we have provided 5 initialization values. Accessing the values of an array. In any point of a program in which an array is visible, we can access the value of any of its

elements individually as if it was a normal variable, thus being able to both read and modify its value.

The format is as simple as: Syntax: array_name[index] Following the previous examples in which marry had 5 elements and each of those elements was of type into, the name which we can use to refer to each element is the allowing: For example, to store the value 75 in the third element of marry, we could write the following statement: marry[2] = 75; and, for example, to pass the value of the third element of marry to a variable called a, we could write: a = marry[2]; Therefore, the expression marry[2] is for all purposes like a variable of type into.

Notice that the third element of marry is specified marry[2], since the first one is marry[O], the second one is marry[1], and therefore, the third one is marry[2]. By this same reason, its last element is marry[4]. Therefore, if we write marry[5], we would e accessing the sixth element of marry and therefore exceeding the size of the array. In C++ it is syntactically correct to exceed the valid range of indices for an array. This can create problems, since accessing out-of-range elements do not cause compilation errors but can cause runtime errors.

The reason why this is allowed will be seen further ahead when we begin to use pointers. At this point it is important to be able to clearly distinguish between the two uses that brackets [ ] have related to arrays. They perform two different tasks: one is to specify the size of arrays when they are cleared; and the second one is to specify indices for concrete array elements. Do not confuse these two possible uses of brackets [ ] with arrays.

into marry[5]; Aryan[2] = 75; // declaration of a new array // access to an element of the array.

If you read carefully, you will see that a type specifies always precedes a variable or array declaration, while it never precedes an access. 95 Some other valid operations with arrays: marry[O] = a; marry[a] = 75; b = marry [a+2]; marry[marry[a]] program 4. 1 = Aryan[2] + 5; // adding all the elements of an array #include using namespace SST; NT Aryan O = {16, 2, 77, 40, 12071}; into n, into main 0 for (; nThe following program shows how to input values into arrays during rum time : program 4. 2 // Inputting value in an array during run-time #include main() into my_ear[5]; // name of array. Values at: “, for(into I I++) //program 4. 3 // program to store 10 integers and show them. //stores value at tit index. For(into I = O; I > my_ear[I coot #include main( ) into a[10] , vale -O; coot Max) Max = ear[I]; 97 cootCheck your progress : 2. 3. 4. Write a program to store 10 elements and increase its value by 5 and show the array. Write the program to divide each of the array element by 3 and show the array. Write a program to find the average of all elements of an array of size 20. Write a program to find second minimum value out of an array containing 10 elements. 4. 2 Strings : Array of characters Strings are in fact sequences of characters, we can represent them also as plain arrays of char elements terminated by a ‘ W character.

For example, the following array: char mystery [20]; is an array that can store up to 20 elements of type char. It can be represented as: mystery ‘ W 19 Therefore, in this array, in theory, we can store sequences of characters up to 20 characters long. But we can also store shorter sequences. For example, mystery could store at some point in a program either the

sequence “ Hello” or the sequence “ Merry Christmas”, since both are shorter than 20 characters.

Therefore, since the array of characters can store shorter sequences than its total length, a special character is seed to signal the end of the valid sequence: the null character, whose literal constant can be written as ‘ O’ (backslash, zero). Our array of 20 elements of type char, called mystery, can be represented storing the characters sequence “ Merry Christmas” as: ‘ h’ Notice how after the valid content a null character (A) has been included in order to indicate the end of the sequence. The panels in gray color represent char elements with undetermined values.

Initialization of null-terminated character sequences Because arrays of characters are ordinary arrays they follow all their same rules. For example, if we want to initialize an array of characters with some predetermined sequence of characters we can do it Just like any other array: char memory[]={H e ‘ l’, In this case we would have declared an array of 6 elements of type char initialized with the characters that form the word “ Hello” plus a null character ‘ W at the end. But arrays of char elements have an additional method to initialize their values: using string literals.

In the expressions we have used in some examples in previous chapters, constants that represent entire strings of characters have already showed p several times. These are specified enclosing the text to become a string literal between double quotes (“). For example: “ the result is: ” is a constant string literal that we have probably used already. Double quoted strings (“) are literal constants whose type is in fact a null-terminated array of

characters. So string literals enclosed between double quotes always have a null character automatically appended at the end.

Therefore we can initialize the array of char elements called memory with a null- terminated sequence of characters by either one of these two methods: char memory e hard memory [ ] = “ Hello”; In both cases the array of characters memory is declared with a size of 6 elements of type char: the 5 characters that compose the word “ Hello” plus a final null character which specifies the end of the sequence and that, in the second case, when using double quotes (“) it is appended automatically.

Please notice that we are talking about initializing an array of characters in the moment it is being declared, and not about assigning values to them once they have already been declared. In fact because this type of null-terminated arrays of characters are regular arrays we have he same restrictions that we have with any other array, so we are not able to copy blocks of data with an assignment operation.

Assuming mystery is a chart] variable, expressions within a source code like: mystery = “ Hello”; mystery[] = “ Hello”; would not be valid, like neither would be: mystery={H e ‘ l’, The reason for this may become more comprehensible once you know a bit more about pointers, since then it will be clarified that an array is in fact a constant pointer pointing to a block of memory. Using null-terminated sequences of characters Null-terminated sequences of characters are the natural way of treating strings in C+ , so they can be used as such in many procedures.

In fact, regular string literals have this type (chart]) and can also be used in most cases. For example, cine and coot support null-terminated sequences as valid containers for sequences of characters, so they can be used directly to extract strings of characters from cine or to insert them into coot. For example: //program 4. 6 //null-terminated sequences of characters #include char question[] = “ Please, enter your first name: “; char greeting[] = “ Hello, “; char yearned [80]; coot question; cine yearned; coot greeting yearned output : Please, enter your first name: John Hello, John!

As you can see, we have declared three arrays of char elements. The first two were initialized with string literal constants, while the third one was left initialized. In any case, we have to specify the size of the array: in the first two (question and greeting) the size was implicitly defined by the length of the literal constant they were initialized to. While for yearned we have explicitly specified that it has a size of 80 chars. //Program 4. 7 // program to count total number of vowels present in a string : into main() char string[35]; into count = O;