**Data Structure, Handout No.1: Arrays, array of structures, functions (methods to communicate with functions, overloading functions, function templates)**

**Assignments 1**

**Definitions:**

**Bit**: is the smallest location in memory which can hold a “0” or “1”

**Byte**: is a collection of 8 bits. One byte holds the binary value of a single character. The binary value or decimal value of each character is called the **ASCII** ( American Standard Code for Information Interchange) of that character.

Consider the following table to see the decimal and binary representation of each character in memory

|  |  |  |
| --- | --- | --- |
| character | ASCII in decimal or base 10 | Binary representation in 1 byte |
| A  B  ….  Z | 65  66  …..  90 | |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | | value | 128=27 | 64 | 32 | 16 | 8 | 4 | 2=21 | 1=20 | | A | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | | B | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | | Z | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | |
| space | 32 | |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | | value | 128=27 | 64 | 32 | 16 | 8 | 4 | 2=21 | 1=20 | | ‘ ‘ | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | |
| ‘0’  ‘1’  …  ‘9’ | 48  49  ….  57 | |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | | value | 128=27 | 64 | 32 | 16 | 8 | 4 | 2=21 | 1=20 | | ‘0’ | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | | ‘1’ | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | | ‘9’ | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | |
| ‘a’  ’b’  ……  ‘z’ | 97  98  ….  122 | |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | | value | 128=27 | 64 | 32 | 16 | 8 | 4 | 2=21 | 1=20 | | ‘a’ | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | | ‘b’ | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | | ‘z’ | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | |

There are a total of 255 characters. To see them all including their ASCII , a simple for loop does it

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Displaying ASCII table** | **‘0’..’9’** | **‘A..’Z’** | | **‘a’..’z’** | |
| #include <iostream>  using namespace std;  int main()  {  for (int i=1; i<255; ++i)  cout<<i<<'\t'<<char(i)<<endl;  system("pause");  return 0;  } | 48 0  49 1  50 2  51 3  52 4  53 5  54 6  55 7  56 8  57 9 | 65 A  66 B  67 C  68 D  69 E  70 F  71 G  72 H  73 I  74 J  75 K  76 L  77 M | 78 N  79 O  80 P  81 Q  82 R  83 S  84 T  85 U  86 V  87 W  88 X  89 Y  90 Z | 97 a  98 b  99 c  100 d  101 e  102 f  103 g  104 h  105 i  106 j  107 k  108 l  109 m | 110 n  111 o  112 p  113 q  114 r  115 s  116 t  117 u  118 v  119 w  120 x  121 y  122 z |

NOTE. cout<< int( ‘A’); displays the ASCII of ‘A’ which is 65. The cout<<char(65); displays the character whose ASCII is 65 which is A

**Data types:** The term *data type* refers to the basic kinds of data that variables may contain.

* **Primitive Data types**: Built-in or predefined data types. Consider the following table

|  |  |
| --- | --- |
| Data types (primitive) | Size in byte |
| char c; | cout<<sizeof ( char) ; returns 1 where the unit is byte. So each character is using 1 byte of memory. The ***sizeof*** is a predefined function which is a member of <iostram>  1 Kilo byte(Kb)=1000 byte = 103 bytes  1 Mega byte (Mb)= 1000 Kb = 1000.103= 103. 103 = 106 bytes  1 Giga byte (Gb)=1000 Mb=103.106 = 109 bytes  1 Terra byte (Tb) = 1000Gb= 103.109=1012 bytes  1 Peta byte (Pb)= 1000 Tb = 103.1012 = 1015 bytes  1 Zetta byte(Zb)=1000 Pb= 103.1015= 1018 bytes  1 Yotta byte (Yb) = 1000 ZB=103.1018 = 1021 bytes  1 Bronta byte (Bb) = 1000 Yb= 103.1021 = 1024 bytes |
| bool b; | cout<<sizeof( bool ); 1 byte = 8 bits |
| short s; | cout<<sizeof( short ); 2 bytes = 16 bits |
| int a; | cout<<sizeof( int); returns 4 bytes. Means each variable of int type will occupy 4 bytes=32 bits of memory. |
| float f; | cout<<sizeof( float); returns 4 bytes= 32 bits |
| double d; | cout<<sizeof( double ); returns 8 bytes =64 bits |
| long int x; | cout<<sizeof( long int ); 8 bytes =64 bits |

* **Abstract Data Type (ADT )** : An ADT specifies the data and operations of an objet implemented by C++ class construct

Example: class Stack{ …….. }; int main() { Stack s; …. } the class Stack is called ADT

* **Classes:** Classes define types or templates of objects. The type concepts derived from ADT
* **Arrays**: Arrays are a collection of consecutive components, all of the same type
* **One-Dimensional Arrays**

|  |  |
| --- | --- |
| Declaration | Meaning |
| int a[4]; | 4 consecutive locations in memory. The content of each location is unknown   |  |  |  |  | | --- | --- | --- | --- | | a[0] | a[1] | a[2] | a[3] | | ? | ? | ? | ? | |
| int a[4]={3,9,1,4}; | Declare and initialize all locations   |  |  |  |  | | --- | --- | --- | --- | | a[0] | a[1] | a[2] | a[3] | | 3 | 9 | 1 | 4 | |
| int a[4]={3,9}; | If you don’t provide enough data, the rest of locations are initialized according to the array type ( in this case all zeros )   |  |  |  |  | | --- | --- | --- | --- | | a[0] | a[1] | a[2] | a[3] | | 3 | 9 | 0 | 0 | |
| int a[4]={0}; | The content of the first location is ZERO, the rest of locations will be initialized according to the array type .   |  |  |  |  | | --- | --- | --- | --- | | a[0] | a[1] | a[2] | a[3] | | 0 | 0 | 0 | 0 |   This is the best way to initialize array of any size of type integer with all zeros: int a[100]={0}; |
| int a[ ]={3, 9, 1, 4}; | The size of the array is the same as the number of listed data which is 4   |  |  |  |  | | --- | --- | --- | --- | | a[0] | a[1] | a[2] | a[3] | | 3 | 9 | 0 | 0 | |

**Examples**

Given declaration : int a[5];

|  |  |
| --- | --- |
| **//read data in array a**  void readData( int x[], int n)  {  for( int i=0; i<n, ++I )  {  cout<<”Enter an integer data:”;  cin>>x[i];  }  }  //calling statement  readData( a, 5); | **//copy data from file “data.txt” into //array a**  Suppose the data are already data.txt  stored in file “data.txt” . We open the  file and copy data from the  file into array x  void copyData(string fname, int x[], int n)  {  //open the file  fstream f;  f.open(fnmae, ios::in);  //copy data from file into array  for( int i=0; i<n, ++I )  {  f>>x[i];  }  f.close(); //close the file  }  //calling statement  copyData(“data.txt”, a, 5); |
| **//display array a**  void displayArray( int c[], int n)  {  for( int i=0; i<n, ++I )  {  cout<<x[i]<<’\t’;  }  cout<<endl;  }  //calling statement  displayArray( a, 5) | **//compute the total of data in array a**  int computeSum(int x[], int n )  {  int sum=0;  for( int i=0; i<n, ++I )  {  sum += x[i];  }  return sun;  }  //calling statement  cout<<computeSum(a, 5); |
| **//compute and return the max and min //data**  void findMaxMin( int x[], int n, int& mx, int& mn)  {  int mx=mn=x[0];  for( int i=1; i<n, ++i )  {  if( x[i] > mx ) mx=x[i];  if ( x[i] < mn ) mn=x[i];  }  }  //Calling statement  int max, min;  findMaxMin( a, 5, max, min); | **//sort array a using member of //<algorithm library**  We will talk about sorting and introducing different methods of sorting in detail before the end of semester. At this time, we use the predefined ***sort***function which is a member of **<algorithm>** library  void sortArray ( int x[], int n)  {  //use the predefined sort function  **sort ( x, x+n );**  }  //calling statement  sortArray( a, 5) |

NOTE Arrays are always pass by reference

**Formatting output**

We use members of <iomanip> such as : left, right, setw, setfil, and setprecision to format the program’s output

Example

|  |  |
| --- | --- |
| #include <iostream>  #include <iomanip>  #include <string>  using namespace std;  void display(string name[], string  capital[], float population[])  {  cout <<fixed<<showpoint<< setprecision(2);  cout << setfill('.');  for (int i = 0; i < 5; ++i)  cout << left << setw(12) << name[i]  << setw(15) << capital[i]  << right<<setw(6)<<population[i]<< endl;    cout << endl;  } | int main()  {  string name[] = { "California", "Texas",  "Oragon", "Utah","Florida" };  string capital[] = { "Sacarmento", "Austin",  "Salem","Salt Lake City", "Tallahassee" };  float population[] = { 39.56, 29, 4.19, 3.16,  21.67 };  display(name, capital, population);  system("pause");  return 0;  }  /\*---------------------------------  California..Sacarmento......39.56  Texas.......Austin..........29.00  Oragon......Salem............4.19  Utah........Salt Lake City...3.16  Florida.....Tallahassee.....21.67  ------------------------------------\* |
| Lets explain the job of each member  a.cout <<fixed<<showpoint<< setprecision(2); | ***fixed*** **: show output in decimal form and not in a scientific form**  ***setprecision(2****)*: **only 2 digits after decimal point**  ***showpoint:* if there are not enough digits after decimal point use extra zeros** |
| b. cout << left << setw(12) << name[i]  << setw(15) << capital[i]  << right<<setw(6)<<population[i]<< endl; | ***Left*. Display the next output using left justifications**  ***setw(n)*: display the next item using n spaces**  ***right:* display next items using right justifications** |
| c. cout << setfill('.'); | **Fill in all spaces left blank by setw with ‘.’** |
|  | |

Lets look at the first line of the program output

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| For name[0]=”Californa”, capital[0]=”Sacramento”, and population[0]=39.56. The members will do this:  cout <<left <<setw(12) <<name[i]<<setw(15) <<capital[i]<< right<<setw(6)<<population[i]<< endl;   |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | Setw(12)  Left justified | | | | | | | | | | | | Setw(15)  Left justified | | | | | | | | | | | | | | | Right,setw(6),2 digits after | | | | | | | C | l | i | f | o | r | n | i | a | . | . | . | S | a | c | r | a | m | e | n | t | o | . | . | . | . | . | . | 3 | 9 | . | 5 | 6 | |

**Methods to communicate with functions**

* ***Pass-by-value***

|  |  |
| --- | --- |
| void sum( int a, int b)  { a 4 b 6  cout<< a+b ;  } b=y  a =x    //calling statement  int main( )  {  int x=4, y= 6; x 4 y 6  cout<<x<<” + “<<y<<” = “; sum( x, y);  } | sum(int a , int b )  a b a receives the x  value and b  receives the y  value  sum( x , y ); //calling state.  the output produced by **cout<<a+b;** in the function  NOTE in pass-by-value , x and a are two different locations. Same for y and b  Output: 4 + 6 = 10 |

* ***Pass-by-return***

|  |  |
| --- | --- |
| int sum( int a, int b) a=x=4 b=y=6  {  return a+b ;  }  Function returns a+b to calling statement  //calling statement  int x=4, y= 6;  cout<<x<<”+”<<y<<”=”<< sum( x, y); | Tracing:  (i)The calling statement passes the x and y values to function sum  (ii)In function sum a=4 and b=6. Function computes a+b and returns 8 to calling statement  (iii) the cout displays the value that functions returns to the calling statement namely 8  Output: 4+6=10 |

* ***Pass-by-reference***

|  |  |
| --- | --- |
| void sum( int a, int b, int& total )  {  total = a+b;  } a=4 b=6 t=total  int x=4, y=6, t;  //calling statement  sum( x, y, t );  cout<<x<<”+”<<y<<”=”<<t; | x and a are pass-by-value, **different locations**  y and b are pass-by-value, **different locations**  t and total are pass-by reference, **same location**  a b total    x=4 y=6 t=10  Output: 4 + 6 = 10  X and y have same values, t gets the new value |

* ***Pass-by-default-values***

|  |  |  |
| --- | --- | --- |
| void print( int a=0, int b=0, int c=0)  {  cout<<a+b+c<<endl;  }  If a,b,c do not receive any values from the calling statement, the compiler will use their initial value | //calling statements  print( );  print(4);  print(4,5);  print(4,5,6); | //output  **a=0,b=0,c=0**. **Output: 0**  a=4,**b=0,c-0,** **Output: 4**  a=4,b=5, **c=0**, **Output: 9**  a=4,b=5,c=6. **Output:16** |

**Example**. Trace the following and show their final output

|  |  |
| --- | --- |
| void f( int& a, int b)  { a++; --b; }  void g( int a, int & b)  { a \*= 2; b+= 3; }  int main( )  {  int x =3, y=6;  f( x, y);//  g(x, y);  cout<<x<<” “<<y;  } | In **Pass-by-reference** , variables a, and x share the same location, any changes in value of a will pass to x immediately. In **pass-by-value** variables b and y are two different locations, any changes in value of b is not going to effect y.  f( int& a , int b)  a b  a b the value of a changes  and will effect x, but  value of b changes in  x=~~3~~ 4 y=6 function and will not  effect y  f( x , y );  After this call x=4 and y=6. Now trace the next call  g( int a, int& b) a and x are two  a b different locations  but, b and y are same  locations  x=4 y=~~6~~ 9  g( x, y );  cout<<x<<” “<<y; output: 4 9 |

* **Overloading functions.** Group of functions may have the same name in one program. That group is called ***overloaded function***. The compiler chooses the correct function, by checking the following conditions:
* Number of arguments in calling statement are **the same** number of varaibles in the function’s heading
* Type of arguments in calling statement are  **the same** as the type of variables in the function’s heading
* Type of the outputs of the function matches with the type of information the calling statement is waiting for

**Example.**

|  |  |
| --- | --- |
| **Overloaded function print** | **Calling statements** |
| **void print( )** // #1  {  cout<<”Hello\n”;  }  //-------------------------------------------------------  **void print( int a )** //#2  {  cout<< a<<endl;  }  //-----------------------------------------------------  **void print( int a, int b)** //#3  {  cout<< a+ b <<endl;  }  //------------------------------------------------------  **void print( string a, string b)** //#4  {  cout<<a+b<<endl;  }  //------------------------------------------------------  **string print( )**  //#5  {  return “Bye\n”;  }  //------------------------------------------------------  **int print( )** //#6  {  retrun 2020;  } | //calling statements  print();  -no arguments ( nothing in ( ) )  Candidates are #1, #5, and #6  -calling statement does not accept any  return value, therefore the only  function matches this calling statement  is function #1  Output: Hello  //Calling statement  string message=print();  The matching function is the one not  asking for any arguments (#1,#5,#6), and  return a string type value which is #5  Output: Bye  //calling statement  print( 2,3);  The candidate must receive two int data  and not to return any value, which is #3  Output: 5  //calling statement  print(“good”,”bye”);  The candidate must receive two string  Data type and not return a value, #4 is  the matching candidate  Output: goodbye |

iii. **Template functions:** if a group of overloaded functions have the same number of arguments, we can convert them to only ONE function template

|  |  |
| --- | --- |
| **Overloaded functions** | **Equivalent function template** |
| **Example 1.**  void print( int a, int b)  {  cout<<a+b<<endl;  }  //--------------------------------------------------  void print(string a, string b)  {  cout<<a+b<<endl;  }  //------------------------------------------------  int main()  {  print( 4,5);  print(“Good”,”bye”);  return 0;  }  The equivalent code looks like this:  **template <class T >**  **void print( T a, T b)**  **{**  **cout<<a+b<<endl;**  **}**  //------------------------------------------------  int main()  {  print( 4,5); // T becomes int  print(“Good”,”bye”); //T becomes string  return 0;  }  You can also type the template function as  **template <class T> void print(T a, T b )**  **{**  **cout<< a + b <<endl;**  **}** | The overloaded function name is : “print” , and both ask for same number of data. Also, the body of both functions does the same thing to both data. Therefore, this overloaded functions are candidate to change to a single template function.  To convert these two function to ONE function, everything remains the same except the type of arguments must change to a variable for example T  void print( T a, T b)  {  cout<<a+b<<endl;  }  This function acts like the first one when **T=int** and becomes the second function when **T=string**.  Now, we need to add one more line to make it a function template  **template <class T >**  **void print( T a, T b)**  **{**  **cout<<a+b<<endl;**  **}**  The main( ) remains the same and produce exactly the same output  The prototype could be typed in one of the following forms  **template <class T >**  **void print( T a, T b);**  **or**  **template <class T > void print( T a, T b);** |
| **Example 2.**  void display(string name, int age)  {  cout<<name<<” “<<age<< “\n”;  }  //--------------------------------------------------  void display(int age, string name )  {  cout<< age<<” “<<name<< “\n”;  }  //---------------------------------------------------  int main ()  {  display(“John”, 25);  display(20, “Bob”);  }  Output:  John 25  20 Bob | Both functions accept the same number of arguments, in the first function their types are **string** and **int**, but in the second function their types are **int** and **string**. Two different type of data, so we write a function template in which there are two argument types **T1** and **T2**.  NOTE. Must repeat the word ***class*** for each data type  template <class T1, class T2>  void display(T1 a, T2 b)  {  cout << a << " " << b << "\n";  }  int main()  {  display("John",25);//T1 is string,T2 is int  display(20, "Bob");//T1 is int,T2 is string  system("pause");  return 0;  } |

**More Examples:**

|  |  |
| --- | --- |
| **//display integer array of size n**  void show( int x[], **int n**)  {  for( int i=0; i<n, ++I )  {  cout<<x[i]<<’\t’;  } int n *same for both*  cout<<endl;  }  **//display string array of size n**  void show( string x[], **int n**)  {  for( int i=0; i<n, ++I )  {  cout<<x[i]<<’\t’;  }  cout<<endl;  }  main( ) is the same | In both functions the second argument is int n, so we don’t need to use a variable type. The first item in 1st function is array of int and in the 2nd function is array of string, so we use variable type T only for their first argument.  This is the function template to replace both functions  template <class T>  void show(T x[], int n)  {  for (int i = 0; i<n; ++i)  {  cout << x[i] << '\t';  }  cout << endl;  }  int main()  {  int age[4] = { 20,15,30,17 };  //display array age  cout << "This is array age:";  show(age, 4);  string names[2]={"Mary", "John"};  //display array names  cout<<”This is array names:”;  show(names, 2);  system("pause");  return 0;  }  Output:  This is array age: 20 15 30 17  This is array names: Mary John |

* **Array of structure** (array of records)

Given the following two arrays consist of 4 student’s age and their names

int age[4]={20,15,30,17};

|  |  |  |  |
| --- | --- | --- | --- |
| age[0] | age[1] | age[2] | age[3] |
| 20 | 15 | 30 | 17 |

string names[4]={“Mary”, “Bob”, “Tom”, “John”};

|  |  |  |  |
| --- | --- | --- | --- |
| name[0] | name[1] | name[2] | name[3] |
| Mary | Bob | Tom | John |

Now suppose we want to merge them into one new array in which at each location the name and age of students are stored. Let’s name the array student[4]

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| student[0] | | student[1] | | student[2] | | student[3] | |
| Mary | 20 | Bob | 15 | Tom | 30 | John | 17 |

This is called an array of ***structures*** ( or records, each record consist of name and age of a student).

**Declaration:** Since each record consist of two info: student’s name and student’s age, we declare the content of each location as:

**struct** person

{

string Name; declare the items in each location

int Age;

};

Now, for 4 students we need to declare an arrays of size 4, where each location consists of one of the above records. Hence we declare array student[4] as ***person*** type

**person student[4]; //** this declaration reserves 4 consecutive locations to store the name and age of 4 students. The following is the picture of the array **student[4]**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| student[0] | | student[1] | | student[2] | | student[3] | |
|  |  |  |  |  |  |  |  |

To initialize each location, we use the dot operator “.”

|  |  |
| --- | --- |
| student[0] | |
| Mary | 20 |

student[0].Name=”Mary”;

student[0].Age=20;

……….

|  |  |
| --- | --- |
| student[3] | |
| John | 17 |

student[3].Name=”John”;

student[3].Age=17;

or we can declare and initialize array student all at once:

person student[4]={ {“Mary”, 20} , {“Bob”,15}, {“Tom”, 30}, {“John”, 17} };

**Example.** The following are examples of passing array of structure to a functions. The technique is exactly the same as one-dimensional array

|  |  |
| --- | --- |
| //display array student  void display( **person** x[], int n)  {  cout<<”NAME\tAGE\n”;  cout<<””=============\n”;  for( int i=0; i<n; ++i)  {  cout<<x[i].Name <<’\t’<<x[i].Age<<endl;    }  }  //calling statement  display( student, 4); | //compute their average age  int computeAgeAve(person x[], int n)  {  int totalAge=0;  for( int i=0; i<n; ++i)  {  totalAge += x[i].Age;  }  return totalAge/n;  }  //calling statement  ageAve=computeAgeAve ( student, 4); |
| //who is the oldest student?  string findOldest( person x[], int n)  {  //suppose the 1st student is the oldest  int mxAge=x[0].Age, oldestIndex=0;  //compare other student’s age with mxAge. If  //someone is older, change the maxAge value  for( int i= 1 ; i<n ; ++i)  {  if ( x[i].Age > mxAge )  {  mxAge = x[i].Age;  //remember the oldest person’s index  oldestIndex = i;  }  //return the name of the oldes student  return x[oldestIndx].Name;  }  //calling statement  string oldestName=findOldest( student, 4);  cout<<”The oldest student name is :”  << oldestName <<endl; | //How many student’s age are above age average  //*ageAve*  int aboveAve( person x[], int n, int ageAverage)  {  int aboveCount=0;  for ( int i=0; i<n; ++i)  {  if (x[i].Age > ageAverage) aboveCount++;  }  return aboveCount;  }  //calling statement  cout<<”No. of students age above age average=”;  cout<<aboveCount(student, 4, ageAve)<<endl; |
| //copy records from a file into array student void copyData( string fname, person x[], int n)  {  //suppose records are in data.txt fstream f;  //text file “data.txt” f.open( fname, ios:: in) //open to read  for (int i=0; i< n; ++i)  {  //read one record at a time from the file  // and copy it into array x  Note. Each line is called a **record** and component is  called a **field**  f>>x[i].Name>>x[i].Age;  }  Calling statement: // close file  f.close( );  copyData(“data.txt”, student, 4 ); } | |

Please follow this template for all programming assignments

**Data Structures Names …………………………………………row………….**

**Assignment No. 1 ………………………………………..**

* (10 points) Consider the following records stored in “data.txt” text file and declarations.

|  |  |
| --- | --- |
| “data.txt” text file. Each record consists of president’s name, president’s years served, and wife’s name  Joe\_Biden 2021-present Jill  Jimmy\_Carter 1977-1981 Rosalynn  Ronald\_Reagon 1981-1989 Nancy  George\_Bush 1989-1993 Barbara  Bill\_Clinton 1993-2001 Hillary  George\_W\_Bush 2001-2009 Laura | declarations |
| struct president  {  string presName;  string yearsServed  string wifeName;    };  person presidents[6 ]; |

Write a program to do the following

* Function **copyData(“data.txt”, presidents);** to copy records from file “data.txt” into array presidents
* Function **displayAll(presidents);** to display a 3 columns

Table (use <iomanip> members) with heading:

Name, Years Served, and wife’s Name

Format your output so that all names and years

served line up left justified. The wife’s name must

end up at the same column by using right justification function

* Given the following arrays:

**int a[6] = { 44,55,22,88,33, 11};**

**string months[12]={“Jan”, ”Feb”, ……, ”Dec”};**

**char vowels[5]={ ‘E’,’U’,’A’,’I’,’O’};**

* Use the template function **displayArray** to display the original arrays

Calling statements: **displayArray(a,6); displayArray(months, 12); displayArray(vowels,5)**

* Callthe function template **sortArray** to sort the original arrays. Use the predefined sort function from the <algorithm> library

Calling statements: **sortArray(a,6); sortArray(months,12); sorstArray(vowels,5);**

* Call **displayArray** again to display the sorted form of each array

**Sample I/O:**

Original array a: 44 55 22 88 33 11

Sorted array a : 11 22 33 44 55 88

Original array months: Jan Feb Mar …….. Dec

Sorted array months : Apr ………..

Original array vowels: E U ……. O

Sorted array vowels : A E……. U

* Part II. Trace the following by hand and show their final output. **Just type your answers**

|  |  |
| --- | --- |
| void print ()  { cout<<” one\n” ;}  void print (int a )  { cout<<” two\n” ; }  void print ( int a, int b)  { cout<<” three\n”; }  Calling statements:  print( 4); output……two……………………………………  print(); output………one………………………………..  print(3, 4); output………three……………………………. | void f(int& x, int& y )  {x += 3; y\*= 2; }  void g(int x, int& y )  { x++; y++; }  Calling statements:  int a=4, b=8;  f( a, b );  cout<<a<<” “<<b<<endl;  …………7 16…………………….  g(a,b );  cout<<a<<” “<<b<<endl;  …………7 17……………. |
| void f( int&a, int& b, int& c)  { c = b; b = a; a = c; }  Calling statement:  int x=4, y=8, z =2;  f(x, y, z );  f( y, z, x);  cout<<x<<”\t”<<y<<”\t”<<z;  ……8  8  4………………………….. | void display( int a=0, int b=0, int c=0)  { cout<<”a=”<<a<<” b=”<<b<<” c=”<<c;}    Calling statements:  display( );  output: a=0 b=0 c=0  display(1, 2 );  output: a=1 b=2 c=0  display(3,4,5);  output: a=3 b=4 c=5 |
| void f( int a, int& b)  { a++; --b; }  void g( int& a, int& b )  { a +=4; b \*=2 ; }  Calling statements  int x=3, y=7;  f( x, y);  cout<<x<<”\t”<<y;  ………3  6……………………..  g(x, y);  cout<<x<<”\t”<<y;  …………7  12…………………... | int h( int a, int b, int c)  { a +=b;  b \*= c;  return a+b+c;  }  void g(int& a, int& b, int c)  { a +=c;  b += a;  c++;  }  Calling statements:  int x= 4, y=3, z= 2;  g(x,y,z);  cout<<x<<” “<<y<<” “<<z;  ………6 9 2…………………  cout<<h(y, z, x);  ………………25………….. |