**COSC 1437 Programming Fundamentals II**

L E S S O N S E T

**By KariAnn Harjo**

Structures and Abstract Data Types

11

|  |  |  |
| --- | --- | --- |
| PURPOSE | 1. | To introduce the concept of an abstract data type |
|  | 2. | To introduce the concept of a structure |
|  | 3. | To develop and manipulate an array of structures |
|  | 4. | To use structures as parameters |
|  | 5. | To use hierarchical (nested) structures |
| PROCEDURE | 1. | Students should read the Pre-lab Reading Assignment before coming to lab. |
|  | 2. | Students should complete the Pre-lab Writing Assignment before coming to lab. |
|  | 3. | In the lab, students should complete labs assigned to them by the instructor. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Contents | Pre-requisites | Approximate completion time | Page number | Check when done |
| Pre-lab Reading Assignment |  | 20 min. | 196 |  |
| Pre-lab Writing Assignment | Pre-lab reading | 10 min. | 205 |  |
| LESSON 11 A |  |  |  |  |
| Lab 11.1  Working with Basic Structures | Knowledge of previous | 15 min. | 205 |  |
|  | chapters |  |  |  |
| Lab 11.2  Initializing Structures | Basic understanding of | 15 min. | 206 |  |
|  | structures |  |  |  |
| Lab 11.3  Arrays of Structures | Basic understanding of | 20 min. | 208 |  |
|  | arrays and structures |  |  |  |
| LESSON 11 B |  |  |  |  |
| Lab 11.4  Nested Structures | Basic understanding of | 20 min. | 209 |  |
|  | functions and nested logic | |  |  |

195

# P R E - LA B R E A D I N G A S S I G N M E N T

So far we have learned of data types such as float, int, char, etc. In some applications the programmer needs to create their own data type. A user defined data type is often an **abstract data type (ADT)**. The programmer must decide which values are valid for the data type and which operations may be performed on the data type. It may even be necessary for the programmer to design new operations to be applied to the data. We will study this style of programming exten- sively when we introduce *object-oriented programming* in the lesson set from Chapter 13.

As an example, suppose you want to create a program to simulate a calen- dar. The program may contain the following ADTs: year, month, and day. Note that month could take on values January, February, . . . , December or even 1,2,

. . . ,12 depending on the wishes of the programmer. Likewise, the range of val- ues for day could be Monday, Tuesday, . . . , Sunday or even 1,2, . . . ,7. There is much more flexibility in the choice of allowable values for year. If the pro- grammer is thinking short term they may wish to restrict year to the range 1990–2010. Of course there are many other possibilities.

In this lab we study the **structure**. Like arrays, structures allow the pro- grammer to group data together. However, unlike an array, structures allow you to group together items of *different* data types. To see how this could be useful in practice, consider what a student must do to register for a college course. Typically, one obtains the current list of available courses and then selects the desired course or courses. The following is an example of a course you may choose:

CHEM 310 Physical Chemistry 4 Credits

Note that there are four items related to this course: the course discipline (CHEM), the course number (310), the course title (Physical Chemistry), and the number of credit hours (4). We could define variables as follows:

**Variable Definition Information Held**

string discipline 4-letter abbreviation for discipline int courseNumber Integer valued course number string courseTitle First 20 characters of course title short credits Number of credit hours

All of these variables are related because they can hold information about the same course. We can package these together by creating a structure. Here is the declaration:

struct course

{

string discipline; int courseNumber; string courseTitle;

short credits;

}; //note the semi-colon here

The **tag** is the name of the structure, course in this case. The tag is used like a data type name. Inside the braces we have the variable declarations that are the **members** of the structure. So the code above declares a structure named course which has four members: discipline, courseNumber, courseTitle, and credits.

The programmer needs to realize that the structure declaration does not define a variable. Rather it lets the compiler know what a course structure is composed of. That is, the declaration creates a new data type called course. We can now define variables of type course as follows:

course pChem; course colonialHist;

Both pChem and colonialHist will contain the four members previously listed. We could have also defined these two structure variables on a single line:

course pChem, colonialHist;

Both pChem and colonialHist are called **instances** of the course structure. In other words, they are both user defined variables that exist in computer memory. Each structure variable contains the four structure members.

## Access to Structure Members

Certainly the programmer will need to assign the members values and also keep track of what values the members have. C++ allows you to access structure members using the **dot operator**. Consider the following syntax:

colonialHist.credits = 3;

In this statement the integer 3 is assigned to the credits member of colonialHist. The dot operator is used to connect the member name to the structure variable it belongs to.

Now let us put all of these ideas together into a program. Sample Program 11.1 below uses the course structure just described. This interactive program allows a student to add requested courses and keeps track of the number of credit hours for which they have enrolled. The execution is controlled by a do-while loop.

*Sample Program 11.1:*

#include <iostream> #include <string>

#include <cctype> using namespace std;

// This program demonstrates the use of structures

// structure declaration struct course

{

string discipline; int courseNumber; string courseTitle; short credits;

};

*continues*

int main()

{

course nextClass; // next class is a course structure int numCredits = 0;

char addClass; do

{

cout << "Please enter course discipline area: "; cin >> nextClass.discipline;

cout << endl << "Pleae enter the course number: "; cin >> nextClass.courseNumber;

cout << endl << "Please enter the course title: "; cin.ignore(); // necessary for the next line getline(cin, nextClass.courseTitle);

// use getline because course title may have a blank space cout << "Please enter the number of credit hours: ";

cin >> nextClass.credits;

numCredits = numCredits + nextClass.credits;

// output the selected course and pertinent information

cout << "You have been registered for the following: " << endl; cout << nextClass.discipline << " " << nextClass.courseNumber

<< " " << nextClass.courseTitle

<< " " << nextClass.credits << "credits" << endl;

cout << “Would you like to add another class? (Y/N)" << endl; cin >> addClass;

} while(toupper(addClass) == 'Y');

cout << "The total number of credit hours registered for is: "

<< numCredits << endl;

return 0;

}

Make sure that you understand the logic of this program and, in particular, how structures are used. Notice the line at the end of the while loop that reads

while(toupper(addclass) == 'Y');

What do you think the purpose of toupper is?

As a second example, suppose we would like a simple program that com- putes the area and circumference of two circles input by the user. Although we can easily do this using previously developed techniques, let us see how this can be done using structures. We will also determine which circle’s center is further from the origin.

*Sample Program 11.2:*

#include <iostream>

#include <cmath> // necessary for pow function #include <iomanip>

using namespace std;

struct circle // declares the structure circle

{ // This structure has 6 members float centerX; // x coordinate of center float centerY; // y coordinate of center float radius;

float area;

float circumference;

float distance\_from\_origin;

};

const float PI = 3.14159;

int main()

{

circle circ1, circ2; // defines 2 circle structure variables

cout << "Please enter the radius of the first circle: "; cin >> circ1.radius;

cout << endl

<< "Please enter the x-coordinate of the center: "; cin >> circ1.centerX;

cout << endl

<< "Please enter the y-coordinate of the center: "; cin >> circ1.centerY;

circ1.area = PI \* pow(circ1.radius, 2.0); circ1.circumference = 2 \* PI \* circ1.radius; circ1.distance\_from\_origin = sqrt(pow(circ1.centerX,2.0)

+ pow(circ1.centerY,2.0)); cout << endl << endl;

cout << "Please enter the radius of the second circle: "; cin >> circ2.radius;

cout << endl

<< "Please enter the x-coordinate of the center: "; cin >> circ2.centerX;

cout << endl

<< "Please enter the y-coordinate of the center: "; cin >> circ2.centerY;

circ2.area = PI \* pow(circ2.radius, 2.0); circ2.circumference = 2 \* PI \* circ2.radius; circ2.distance\_from\_origin = sqrt(pow(circ2.centerX,2.0)

+ pow(circ2.centerY,2.0));

*continues*

cout << endl << endl;

// This next section determines which circle's center is

// closer to the origin

if (circ1.distance\_from\_origin > circ2.distance\_from\_origin)

{

cout << "The first circle is further from the origin"

<< endl << endl;

}

else if (circ1.distance\_from\_origin < circ2.distance\_from\_origin)

{

cout << "The first circle is closer to the origin"

<< endl << endl;

}

else

cout << "The two circles are equidistant from the origin"; cout << endl << endl;

cout << setprecision(2) << fixed << showpoint;

cout << "The area of the first circle is : "; cout << circ1.area << endl;

cout << "The circumference of the first circle is: "; cout << circ1.circumference << endl << endl;

cout << "The area of the second circle is : "; cout << circ2.area << endl;

cout << "The circumference of the second circle is: "; cout << circ2.circumference << endl << endl;

return 0;

}

## Arrays of Structures

In the previous sample program we were interested in two instances of the cir- cle structure. What if we need a much larger number, say 100, instances of this structure? Rather than define each one separately, we can use an **array of struc- tures**. An array of structures is defined just like any other array. For example sup- pose we already have the following structure declaration in our program:

struct circle

{

float centerX; // x coordinate of center float centerY; // y coordinate of center float radius;

float area;

float circumference;

float distance\_from\_origin; // distance of center from origin

};

Then the following statement defines an array, circn, which has 100 elements. Each of these elements is a circle structure variable:

circle circn[100];

Like the arrays encountered in previous lessons, you can access an array ele- ment using its subscript. So circn[0] is the first structure in the array, circn[1] is the second, and so on. The last structure in the array is circn[99]. To access a member of one of these array elements, we still use the dot operator. For instance, circn[9].circumference gives the circumference member of circn[9]. If we want to display the center and distance from the origin of the first 30 cir- cles we can use the following:

for (int count = 0; count < 30; count++)

{

cout << circn[count].centerX << endl; cout << circn[count].centerY << endl; cout << circn[count].distance\_from\_origin;

}

## Initializing Structures

When studying arrays you may have seen two-dimensional arrays which allow one to have “a collection of collections” of data. An array of structures allows one to do the same thing. However, we have already noted that structures permit you to group together items of different data type, whereas arrays do not. So an array of structures can sometimes be used when a two-dimensional array cannot.

We have already seen numerous examples of initializing variables and arrays at the time of their definition in the previous labs. Members of structures can also be initialized when they are defined. Suppose we have the following structure dec- laration in our program:

struct course

{

string discipline; int courseNumber; string courseTitle; short credits;

};

A structure variable colonialHist can be defined and initialized:

course colonialHist = {"HIST",302,"Colonial History",3};

The values in this list are assigned to course’s members in the order they appear. Thus, the string "HIST" is assigned to colonialHist.discipline, the integer 302 is assigned to colonialHist.courseNumber, the string "Colonial History" is assigned to colonialHist.courseTitle, and the short value 3 is assigned to colonialHist.credits. It is not necessary to initialize all the members of a struc- ture variable. For example, we could initialize just the first member:

course colonialHist = {"HIST"};

This statement leaves the last three members uninitialized. We could also initial- ize only the first two members:

course colonialHist = {"HIST",302};

There is one important rule, however, when initializing structure members. If one structure member is left uninitialized, then all structure members that follow it must be uninitialized. In other words, we cannot skip members of a structure during the initialization process.

It is also worth pointing out that you cannot initialize a structure member in the declaration of the structure. The following is an illegal declaration:

// illegal structure declaration struct course

{

string discipline = "HIST"; // illegal

int courseNumber = 302; // illegal string courseTitle = "Colonial History"; // illegal short credits = 3; // illegal

};

If we recall what a structure declaration does, it is clear why the above code is illegal. A structure declaration simply lets the compiler know what a structure is composed of. That is, the declaration creates a new data type (called course in this case). So the structure declaration does not define any variables. Hence there is nothing that can be initialized there.

## Hierarchical (Nested) Structures

Often it is useful to nest one structure inside of another structure. Consider the following:

*Sample Program 11.3:*

#include <iostream> #include <iomanip> #include <cmath> using namespace std;

**struct center\_struct**

**{**

**float x; // x coordinate of center float y; // y coordinate of center**

**};**

struct circle

{

float radius; float area;

float circumference;

**center\_struct coordinate;**

};

const float PI = 3.14159;

int main()

{

circle circ1, circ2; // defines 2 circle structure variables

cout << "Please enter the radius of the first circle: "; cin >> circ1.radius;

cout << endl

<< "Please enter the x-coordinate of the center: "; cin >> **circ1.coordinate.x;**

cout << endl

<< "Please enter the y-coordinate of the center: "; cin >> **circ1.coordinate.y;**

circ1.area = PI \* pow(circ1.radius, 2.0); circ1.circumference = 2 \* PI \* circ1.radius;

cout << endl << endl;

cout << "Please enter the radius of the second circle: "; cin >> circ2.radius;

cout << endl

<< "Please enter the x-coordinate of the center: "; cin >> **circ2.coordinate.x;**

cout << endl

<< "Please enter the y-coordinate of the center: "; cin >> **circ2.coordinate.y;**

circ2.area = PI \* pow(circ2.radius, 2.0); circ2.circumference = 2 \* PI \* circ2.radius;

cout << endl << endl;

cout << setprecision(2) << fixed << showpoint;

cout << "The area of the first circle is : "; cout << circ1.area << endl;

cout << "The circumference of the first circle is: "; cout << circ1.circumference << endl;

cout << "Circle 1 is centered at (" << **circ1.coordinate.x**

<< "," << **circ1.coordinate.y** << ")." << endl << endl;

cout << "The area of the second circle is : "; cout << circ2.area << endl;

cout << "The circumference of the second circle is: "; cout << circ2.circumference << endl ;

cout << "Circle 2 is centered at (" << **circ2.coordinate.x**

<< "," << **circ2.coordinate.y** << ")." << endl << endl;

return 0;

}

Note that the programs in this lesson so far have not been modularized. Everything is done within the main function. In practice, this is not good structured pro- gramming. It can lead to unreadable and overly repetitious code. To solve this problem, we need to be able to pass structures and structure members to func- tions. In this next section, you will see how to do this.

## Structures and Functions

Just as we can use other variables as function arguments, structure members may be used as function arguments. Consider the following structure declaration:

struct circle

{

float centerX; // x coordinate of center float centerY; // y coordinate of center float radius;

float area;

};

Suppose we also have the following function definition in the same program:

float computeArea(float r)

{

return PI \* r \* r; // PI must previously be defined as a

// constant float

}

Let firstCircle be a variable of the circle structure type. The following func- tion call passes firstCircle.radius into r. The return value is stored in firstCircle.area:

firstCircle.area = computeArea(firstCircle.radius);

It is also possible to pass an entire structure variable into a function rather than an individual member.

struct course

{

string discipline; int courseNumber; string courseTitle; short credits;

};

course pChem;

Suppose the following function definition uses a course structure variable as its parameter:

void displayInfo(course c)

{

cout << c.discipline << endl; cout << c.courseNumber << endl; cout << c.courseTitle << endl; cout << c.credits << endl;

}

Then the following call passes the pChem variable into c:

displayInfo(pChem);

There are many other topics relating to functions and structures such as return- ing a structure from a function and pointers to structures. Although we do not have time to develop these concepts in this lab, the text does contain detailed cover- age of these topics for the interested programmer.

# P R E - LA B W R I T I N G A S S I G N M E N T

Fill-in-the-Blank Questions

# LESSON 11 A

1. The name of a structure is called the tag .
2. An advantage of structures over arrays is that structures allow one to use items of different data types, whereas arrays do not.
3. One structure inside of another structure is an example of a nested structure .
4. The variables declared inside the structure declaration are called the

members of the structure.

1. When initializing structure members, if one structure member is left uninitialized, then all the structure members that follow must be

uninitialized .

1. A user defined data type is often an abstract data type .
2. Once an array of structures has been defined, you can access an array element using its index .
3. The dot operator allows the programmer to access structure members.
4. You may not initialize a structure member in the structure declaration .
5. Like variables, structure members may be used as function arguments.

## LAB 11.1 Working with Basic Structures

Bring in program rect\_struct.cpp from the Lab 11 folder. The code is shown below.

#include <iostream> #include <iomanip> using namespace std;

// This program uses a structure to hold data about a rectangle

// **KariAnn Harjo**

**// Fill in code to declare a structure named rectangle which has**

**// members length, width, area, and perimeter all of which are floats**

**struct rectangle {**

**float length;**

**float width;**

**float area;**

**float perimeter;**

**};**

int main()

{

**// Fill in code to define a rectangle variable named box  
rectangle box;**

cout << "Enter the length of a rectangle: ";

**// Fill in code to read in the length member of box  
cin >> box.length;**

cout << "Enter the width of a rectangle: ";

*continues*

**// Fill in code to read in the width member of box  
cin >> box.width;**

cout << endl << endl;

**// Fill in code to compute the area member of box  
box.area = box.length \* box.width;**

**// Fill in code to compute the perimeter member of box  
box.perimeter = 2 \* (box.length + box.width);**

cout << fixed << showpoint << setprecision(2);

**// Fill in code to output the area with an appropriate message  
cout << "The area of the rectangle is: " << box.area << endl;**

**// Fill in code to output the perimeter with an appropriate message  
cout << "The perimeter of the rectangle is: " << box.perimeter << endl;**

**if (box.length == box.width) {**

**cout << "The rectangle is a square." << endl;**

**} else {**

**cout << "The rectangle is not a square." << endl;**

**}**

return 0;

}

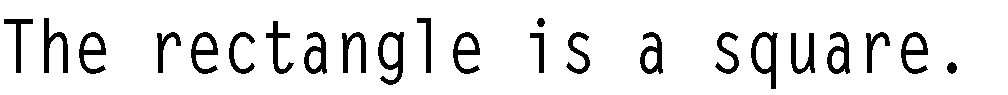
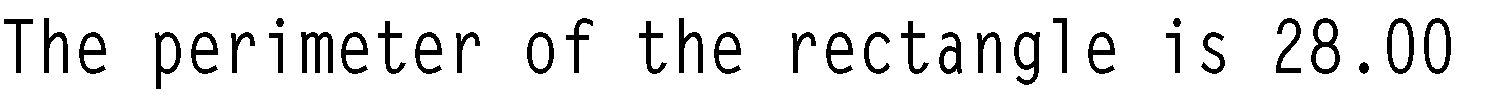
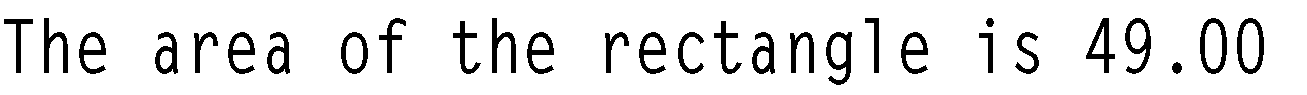
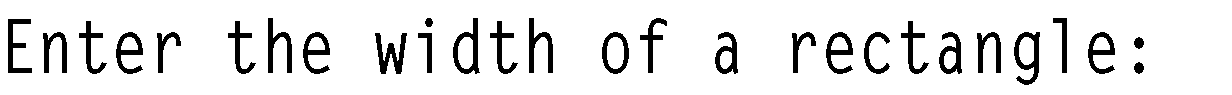
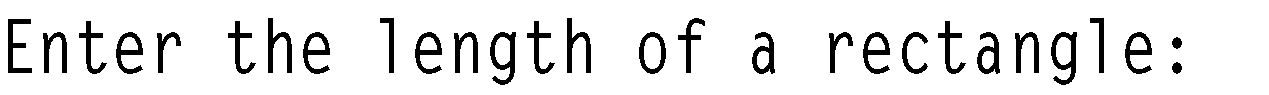
*Exercise 1:* Fill in the code as indicated by the comments in bold.

*Exercise 2:* Add code to the program above so that the modified program will determine whether or not the rectangle entered by the user is a square.

A screenshot of a computer program

Description automatically generated with medium confidence

*Sample Run:*



## LAB 11.2 Initializing Structures

Bring in program init\_struct.cpp from the Lab 11 folder. The code is shown below.

#include <iostream> #include <string> #include <iomanip>

using namespace std;

// This program demonstrates partially initialized structure variables

**// KariAnn Harjo**

struct taxPayer

{

string name;

long socialSecNum; float taxRate; float income; float taxes;

};

int main()

{

**// Fill in code to initialize a structure variable named citizen1 so that**

**// the first three members are initialized. Assume the name is Tim**

**// McGuiness, the social security number is 255871234, and the tax rate is .35**

**taxPayer citizen1, citizen2;**

**citizen1.name = "Tim McGuiness";**

**citizen1.socialSecNum = 255871234;**

**citizen1.taxRate = 0.35;**

**// Fill in code to initialize a structure variable named citizen2 so that**

**// the first three members are initialized. Assume the name is John Kane,**

**// the social security number is 278990582, and the tax rate is .29**

**citizen2.name = "John Kane";**

**citizen2.socialSecNum = 278990582;**

**citizen2.taxRate = 0.29;**

cout << fixed << showpoint << setprecision(2);

// calculate taxes due for citizen1

**// Fill in code to prompt the user to enter this year's income for the citizen1  
cout << "Enter this year's income for " << citizen1.name << ": ";**

**// Fill in code to read in this income to the appropriate structure member  
cin >> citizen1.income;**

**// Fill in code to determine this year's taxes for citizen1  
citizen1.taxes = citizen1.income \* citizen1.taxRate;**

cout << "Name: " << citizen1.name << endl;

cout << "Social Security Number: " << citizen1.socialSecNum << endl; cout << "Taxes due for this year: $" << citizen1.taxes << endl << endl;

// calculate taxes due for citizen2

**// Fill in code to prompt the user to enter this year's income for citizen2  
cout << "Enter this year's income for " << citizen2.name << ": ";**

**// Fill in code to read in this income to the appropriate structure member  
cin >> citizen2.income;**

**// Fill in code to determine this year's taxes for citizen2  
citizen2.taxes = citizen2.income \* citizen2.taxRate;**

cout << "Name: " << citizen2.name << endl;

cout << "Social Security Number: " << citizen2.socialSecNum << endl; cout << "Taxes due for this year: $" << citizen2.taxes << endl << endl;

return 0;

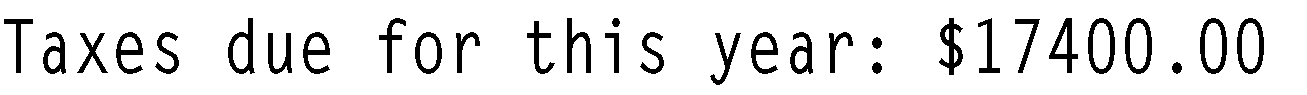
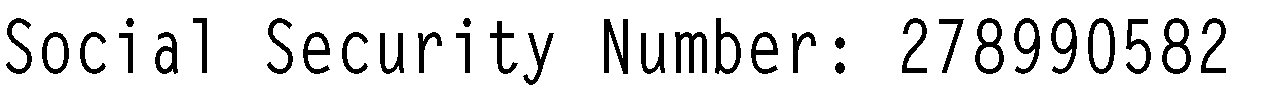
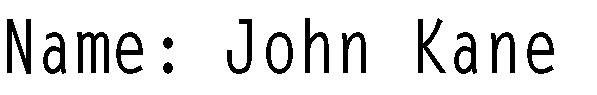
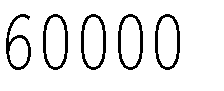
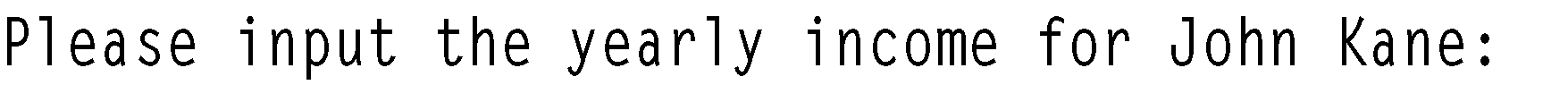
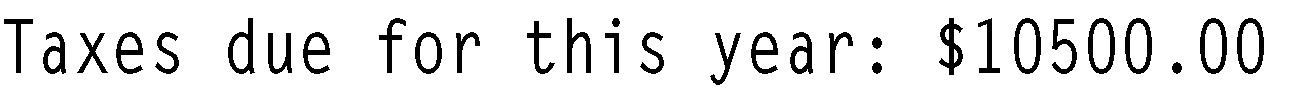
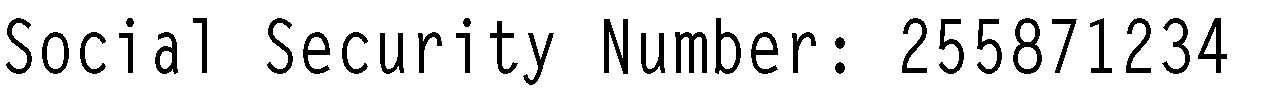
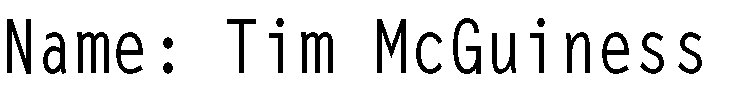
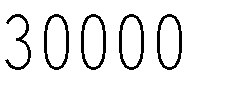
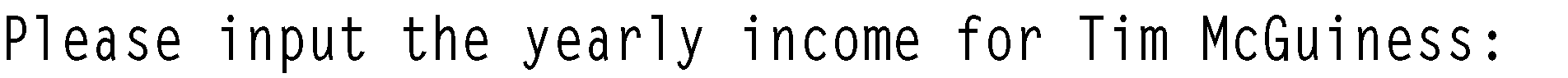
}

*Exercise 1:* Fill in the code as indicated by the comments in bold.

A picture containing text, screenshot, font

Description automatically generated

*Sample Run:*



## LAB 11.3 Arrays of Structures

Bring in program array\_struct.cpp from the Lab 11 folder. The code is shown below.

#include <iostream> #include <iomanip>

using namespace std;

// This program demonstrates how to use an array of structures

**// KariAnn Harjo**

**// Fill in code to declare a structure called taxPayer that has three**

**// members: taxRate, income, and taxes — each of type float  
  
struct taxPayer {**

**float taxRate;**

**float income;**

**float taxes;**

**};**

int main()

{

**// Fill in code to define an array named citizen which holds**

**// 5 taxPayers structures  
taxPayer citizen[5];**

cout << fixed << showpoint << setprecision(2);

cout << "Please enter the annual income and tax rate for 5 tax payers: "; cout << endl << endl << endl;

for(int count = 0;count < 5;count++)

{

cout << "Enter this year's income for tax payer " << (count + 1); cout << ": ";

**// Fill in code to read in the income to the appropriate place  
cin >> citizen[count].income;**

cout << "Enter the tax rate for tax payer # " << (count + 1); cout << ": ";

**// Fill in code to read in the tax rate to the appropriate place  
cin >> citizen[count].taxRate;**

**// Fill in code to compute the taxes for the citizen and store it**

**// in the appropriate place**

**citizen[count].taxes = citizen[count].income \* citizen[count].taxRate;**

cout << endl;

}

cout << "Taxes due for this year: " << endl << endl;

**// Fill in code for the first line of a loop that will output the**

**// tax information**

**for (int index = 0; index < 5; index++) {**

**cout << "Tax Payer # " << (index + 1) << ": $" << citizen[index].taxes << endl;**

**}**

return 0;

}

*Exercise 1:* Fill in the code as indicated by the comments in bold.

*Exercise 2:* In the previous code we have the following:

cout << "Tax Payer # " << (index+1) << ": " << "$ "

<< citizen[index].taxes << endl;

Why do you think we need (index+1) in the first line but index in the second?

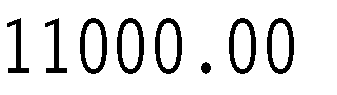
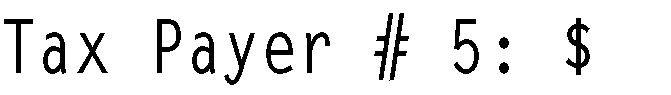
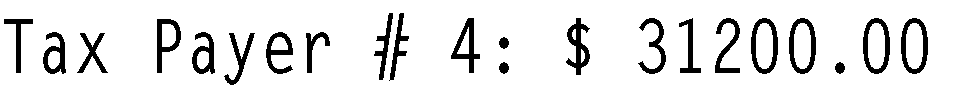
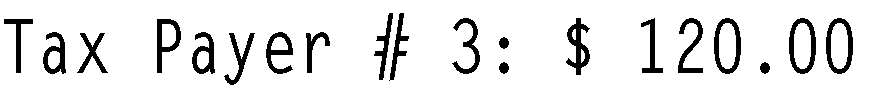
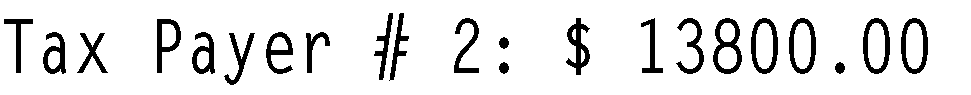
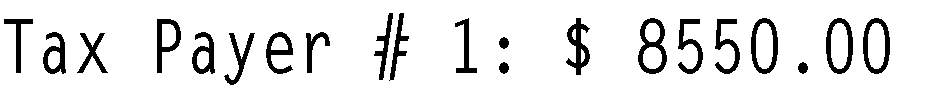
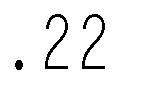
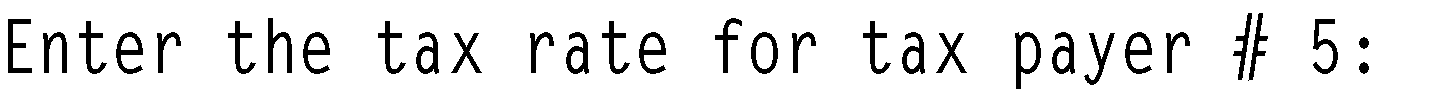
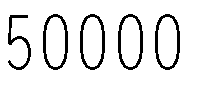
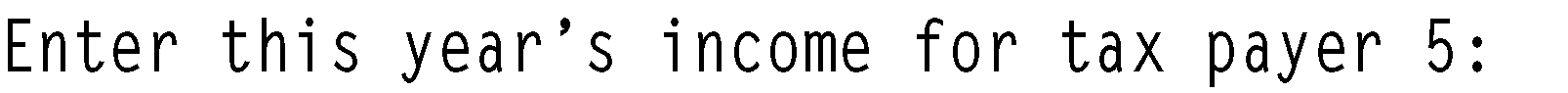
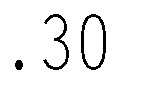
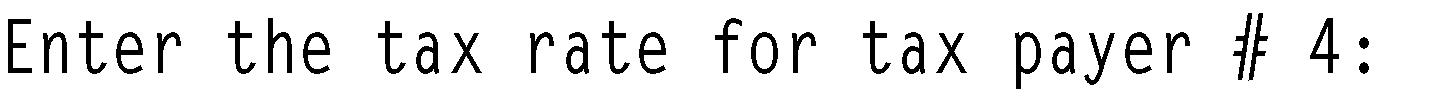
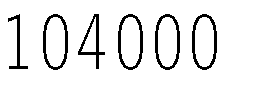
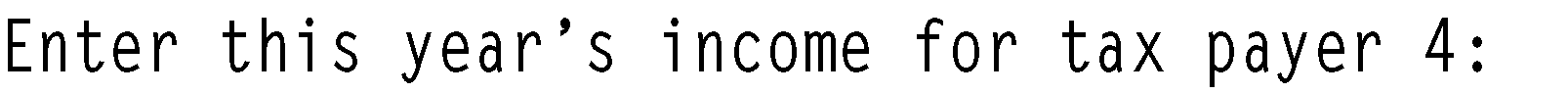
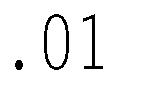
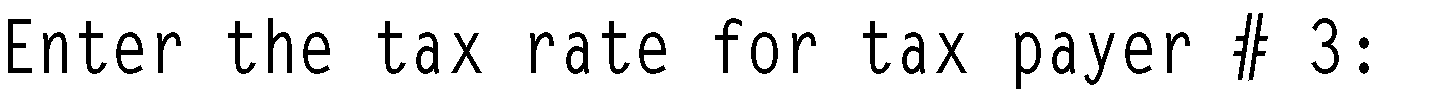
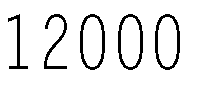
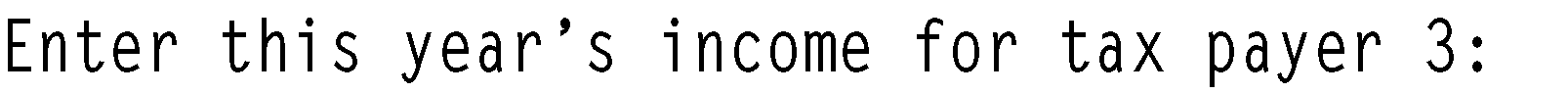
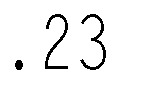
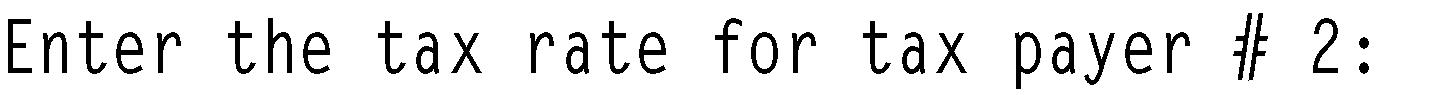
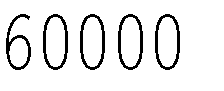
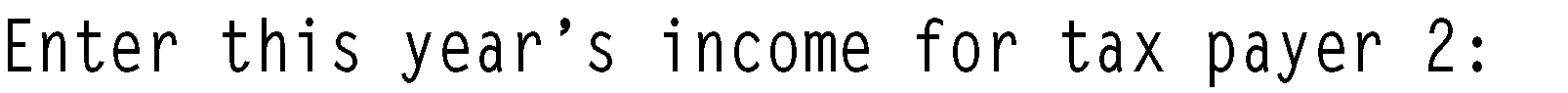
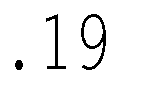
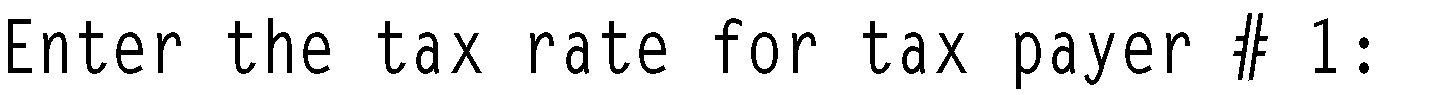
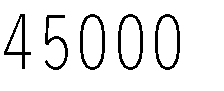
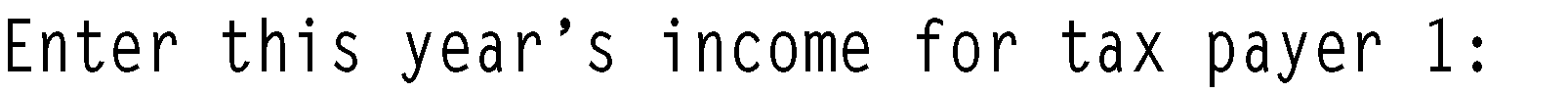
By adding 1 to index using (index + 1), we adjust the index value to be 1-based instead of 0-based. This way, the output will show tax payer numbers as 1, 2, 3, and so on, rather than 0, 1, 2, and so on.

On the other hand, in the second line, citizen[index].taxes, we do not need to add 1 because we are accessing the array citizen using the 0-based index as per the C++ array indexing convention.

A screenshot of a computer

Description automatically generated

*Sample Run:*



# LESSON 11 B

## LAB 11.4 Nested Structures

Bring in program nestedRect\_struct.cpp from the Lab 11 folder. This code is very similar to the rectangle program from Lab 11.1. However, this time you will complete the code using nested structures. The code is shown below.

*Exercise 1:* Fill in the code as indicated by the comments in bold.

*Exercise 2:* Modify the program above by adding a third structure named results which has two members area and perimeter**.** Adjust the rectan- gle structure so that both of its members are structure variables.

#include <iostream> #include <iomanip>

using namespace std;

*continues*

// This program uses a structure to hold data about a rectangle

// It calculates the area and perimeter of a box

**// KariAnn Harjo**

**// Fill in code to declare a structure named dimensions that**

**// contains 2 float members, length and width  
struct dimensions {**

**float length;**

**float width;**

**};**

**// Fill in code to declare a structure named rectangle that contains**

**// 3 members, area, perimeter, and sizes. area and perimeter should be**

**// floats, whereas sizes should be a dimensions structure variable  
  
struct results {**

**float area;**

**float perimeter;**

**};  
  
struct rectangle {**

**dimensions sizes;**

**results attributes;**

**};**

int main()

{

**// Fill in code to define a rectangle structure variable named box.  
rectangle box;**

cout << "Enter the length of a rectangle: ";

**// Fill in code to read in the length to the appropriate location  
cin >> box.sizes.length;**

cout << "Enter the width of a rectangle: ";

**// Fill in code to read in the width to the appropriate location  
cin >> box.sizes.width;**

cout << endl << endl;

**// Fill in code to compute the area and store it in the appropriate**

**// location  
box.attributes.area = box.sizes.length \* box.sizes.width;**

**// Fill in code to compute the perimeter and store it in the**

**// appropriate location  
box.attributes.perimeter = 2 \* (box.sizes.length + box.sizes.width);**

cout << fixed << showpoint << setprecision(2);

cout << "The area of the rectangle is " << box.attributes.area << endl; cout << "The perimeter of the rectangle is " << box.attributes.perimeter

<< endl;

return 0;

}

*Exercise 3:* Modify the program above by adding functions that compute the area and perimeter. The structure variables should be passed as arguments to the functions.

#include <iostream>

#include <iomanip>

using namespace std;

// This program uses a structure to hold data about a rectangle

// It calculates the area and perimeter of a box

**// KariAnn Harjo**

**// Fill in code to declare a structure named dimensions that**

**// contains 2 float members, length and width**

**struct dimensions {**

**float length;**

**float width;**

**};**

**// Fill in code to declare a structure named rectangle that contains**

**// 3 members, area, perimeter, and sizes. area and perimeter should be**

**// floats, whereas sizes should be a dimensions structure variable**

**struct results {**

**float area;**

**float perimeter;**

**};**

**struct rectangle {**

**dimensions sizes;**

**results attributes;**

**};**

**// Function to compute the area**

**void computeArea(rectangle& rect) {**

**rect.attributes.area = rect.sizes.length \* rect.sizes.width;**

**}**

**// Function to compute the perimeter**

**void computePerimeter(rectangle& rect) {**

**rect.attributes.perimeter = 2 \* (rect.sizes.length + rect.sizes.width);**

**}**

**// Fill in code to define a rectangle structure variable named box.**

int main() {

// Define a rectangle structure variable named box

**rectangle box;**

cout << "Enter the length of a rectangle: ";  
  
**// Fill in code to read in the length to the appropriate location**

**cin >> box.sizes.length;**

cout << "Enter the width of a rectangle: ";  
  
**// Fill in code to read in the width to the appropriate location**

**cin >> box.sizes.width;**

cout << endl << endl;

**// Fill in code to compute the area and store it in the appropriate**

**// location**

computeArea(box);

**// Fill in code to compute the perimeter and store it in the**

**// appropriate location**

**computePerimeter(box);**

cout << fixed << showpoint << setprecision(2);

cout << "The area of the rectangle is " << box.attributes.area << endl;

cout << "The perimeter of the rectangle is " << box.attributes.perimeter << endl;

return 0;

}

*Exercise 1:* Fill in the code as indicated by the comments in bold.

*Exercise 2:* Modify the program above by adding a third structure named results which has two members area and perimeter**.** Adjust the rectan- gle structure so that both of its members are structure variables.

*Exercise 3:* Modify the program above by adding functions that compute the area and perimeter. The structure variables should be passed as arguments to the functions.

A screenshot of a computer screen

Description automatically generated with low confidence

A screenshot of a computer program

Description automatically generated with medium confidence

*Sample Run:*

