Struct

```
struct structName
{
    Body of struct
};
```

A **struct** is a collection of data. Variables of any type are allowed in one struct. This makes for a more compact way to represent data. For instance, if you want to keep record of various customer's first names, last names, and customer IDs, you would have to create a variable for each and every customer's first name, last name, and customer ID. This would make for many variables with no set order. Luckily, structs will reduce the number of your total variables. Here is an example of a **struct** followed by an explanation:

```
struct CustomerData
{
    string firstName = "";
    string lastName = "";
    int customerID = 0;   //Default values optional
};
```

You normally will see structs on the top of your program above function declarations (and the main function). All structs start with using the keyword **struct**. After the keyword is the name of your struct. Here, the name is **CustomerData**. The name of your struct is usually uppercase. In between the curly brackets is the body of the struct. This is where your variables (and functions) will go. Lastly, a struct is a statement. Therefore, there needs to be a **semicolon** after the closing curly bracket. Your program will error during compilation if you omit the **semicolon**.

By default, a struct is **public**. This means that the data inside may be used anywhere in any function of the program. All you need to access the data is a struct 'variable' (known as an **instance** of the struct), which is shown below.

In order to use a struct, you declare an instance the same way you would a **char** or an **int**. Here is an example of declaring struct variables using the struct above:

```
int main()
{
    CustomerData Person1, Person2, Person3;
}
```

CustomerData is the type and **Person1**, **Person2**, and **Person3** are the instances. The names of the struct instances are usually uppercase. Now you have three unique instances that all will house unique values in the struct.

In order to access the variables in the struct, you use the **dot operator** just like you do when dealing with input and output files.

The **dot operator** lets you specify which specific variable of the struct you want to access. If you want **Person1's** first name, then you would do **Person1.firstName**. You are able to initialize and display the information using the same model. If you wanted to initialize **Person1's** data, you can do so as such:

```
Person1.firstName = "Aaron";
Person1.lastName = "Malone";
Person1.customerID = 919;
```

If you wanted to display **Person1's** information, you can do so as such:

```
cout << Person1.firstName;
cout << Person1.lastName;
cout << Person1.customerID;
```

Remember, the values for **firstName**, **lastName**, and **customerID** will be unique to every instance of the struct. **Person1.firstName** will be different than **Person2.firstName**. If you would like to create a function to set all of the different variables, you can make a global function to do so. This would eliminate having to initialize every person like how **Person1** was initialized, thus giving you smaller code. This option also works for displaying information. See **Customer.cpp** after this section for an example and explanation.

Since a struct is a type, you are also able to have a struct type within another struct. That is, if you have a **struct Birthday** and a **struct Person**, you are able to put type **Birthday** into the **Person** struct just as you would a type **int** or **string**. In order to access the **Birthday** struct from within the **Person** struct, you will need to use the **dot operator** twice. An example program with an explanation is **TwoStructs.cpp** after this section.

Structs are a precursor to classes. **Structs are used only when all variables are public.** You will see similarities between the two. However, classes are the main aspect of Object-Oriented Programming and are more flexible in what you can do with them.

Please take note that a **private** section/functions are allowed in a **struct** but is not standard convention. This trait is used in classes.

If you wish to dynamically allocate a **struct**, you would by using the **new operator** as discussed in the Pointers section (read Pointers for more information). If you know what dynamic allocation is, you know that you are dealing with pointers. Once a struct has been dynamically allocated, in order to get to the data, you will have to **dereference** the pointer and use the **dot operator**. This can be combined into what is called the **arrow operator ->**. For example, **(*pointerToStruct).structData** can be more cleanly written as **pointerToStruct->structData**.