Object-Oriented Programming  
Tutorial 01 - Creating Composite Data Types Using Structures

## Introduction

So far you have been using relatively basic data structures, using either variables representing a single value of a given data type, or arrays that store a group of sequential values of a **single** fundamental data type. This tutorial will introduce you to the creation of a wider range of data types using the C/C++ concept of a **structure**. Structures allow you to **group together related data**, as well as allowing you to **store different data types together** as a single unit. Understanding how structures work will help you understand the concept of **classes**, which will be introduced later in the unit.

## Representing Related Data

At the moment, if you wanted to store several different pieces of information about something then you would need to store them in separately created variables, e.g. if you wanted to store data on the physical characteristics of a person you would need to use:

float person\_height\_metres;

float person\_weight\_kilograms;

float person\_waist\_cms;

To store the same data for a number of different people would require three separate arrays:

float person\_height\_metres[MAX\_NUM\_PEOPLE];

float person\_weight\_kilograms[MAX\_NUM\_PEOPLE];

float person\_waist\_cms[MAX\_NUM\_PEOPLE];

One way that you might consider combining these into a single array might be to use a 2D array, e.g.:

float person\_info[MAX\_NUM\_PEOPLE][3];

person\_info[0][0] = 1.75f; // height

person\_info[0][1] = 75.5f; // weight

person\_info[0][2] = 76.2f; // waist

but one problem with this method is that you have to remember which array column element represents which piece of data. Another problem is that it cannot be used to store different data types together. If you needed to store different data types you would **have** to store them in **separate** arrays, e.g.:

**int person\_age[MAX\_NUM\_PEOPLE];**

float person\_height\_metres[MAX\_NUM\_PEOPLE];

All of these representations can be greatly improved by using **structures** instead.

## Structures

Structures allow you to group together variables to store a set of related data, of any combination of data types, **creating a new data type** in the process. This is an important point to understand as you will no longer be limited to using the fundamental data types, but can create **composite data types** that can represent almost anything.

### Creating Structures

Structures are simple to define, you simply encapsulate all of the data elements you require within the body of a **struct** declaration. The basic format of a structure declaration is:

struct structure\_name

{  
 data\_type\_1 m\_name\_1; // struct member variable  
 data\_type\_2 m\_name\_2;  
 ...  
 ...

data\_type\_n m\_name\_n;  
};

It is important to remember to end a structure declaration with a semicolon as indicated. Each **member** of a structure can be **any** data type, and all of the names within a structure must be unique within that structure. You can now use a structure to group all of the information required for a given collection of data in a single place, e.g. for the information about a person:

struct Person\_Info

{

int m\_age;

float m\_height\_metres;

float m\_weight\_kilograms;

float m\_waist\_cms;

}; // don't forget to end with semicolon

Notice that each **member** variable name has been preceded by 'm'. This is not a requirement but is a convention widely used to indicate a variable is a member of a structure (or a class). It's worth getting into the habit of using this convention as it will help prevent common mistakes when you start using classes and objects later in the unit.

The Person\_Infostructure is now a data type itself, and can therefore be used to create variables just like any other data type, e.g.

**Person\_Info** steve; // creates a variable called steve, of type **Person\_Info**

**Person\_Info** kylie; // creates a variable called kylie, of type **Person\_Info**

### Using Structures

Of course, now you need some method of accessing the data stored inside the members of the structure. This is done using the '.' (**dot**) operator. Here the values of the two Person\_Infostructures are set:

steve.m\_age = 51;

steve.m\_height\_metres = 1.80f;

steve.m\_weight\_kilograms = 80.25f;

steve.m\_waist\_cms = 71.5f;

kylie.m\_age = 43;

kylie.m\_height\_meters = 1.55f;

kylie.m\_weight\_kilograms = 46.75f;

kylie.m\_waist\_cms = 53.0f;

When accessing a member of a struct using the dot operator, the data type that is being used at that time is the same as the member data type, so steve.m\_age is an int, and kylie.m\_height\_metres is a float. Members accessed in this way are treated in **exactly the same way** as normal variables and you can use them in all the same ways, such as the assignment statements above, or for instance:

int half\_age = steve.m\_age / 2;

if(kylie.m\_weight\_kilograms > 75.0f)

{

go\_on\_diet();

}

### Initialising Structures

You can initialise structures at declaration time in a similar way to initialising arrays by using an **initialiser list**. Here is an example using the Person\_Infovariable bob:

Person\_Info bob = { 68, 170.0f, 92.2f, 80.7f }; // initialise age, height, weight and waist

If you initialise **fewer values than there are member variables** in the structure, then the **remaining variables are set to zero**, so if you initialised bob like this:

Person\_Info bob = { 68 }; // initialise age; height, weight and waist are **zeroed**

then the age member is set to **68**, but the m\_height\_metres, m\_weight\_kilograms and m\_waist\_cms variables are set to **zero**.

Remember **you can only use the initialiser list at declaration time**, in the same way as with arrays. If you try to do this with a variable after it is declared the program will not compile:

bob = { 68, 170.0f, 92.2f, 80.7f }; // DON'T DO THIS - it won't compile

### Assigning Structures

Structures behave like any other data type when it comes to assignment. If you have two Person\_Info structures, let's say bob and copy\_of\_bob, then you can use the assignment operator to copy **the entire contents** of one structure to the other:

Person\_Info bob = { 68, 170.0f, 92.2f, 80.7f }

Person\_Info copy\_of\_bob;

copy\_of\_bob = bob;

After the assignment every element of copy\_of\_bob will match bob.

**Using other Operators with Structures**

By default, most of the other operators (such as add and subtract) are **undefined** for new structures. If you think about it, what does it mean to add two Person\_Info variables together? Because the compiler can't possibly know what you, as the designer of the data structure, means by this addition (or whatever other operator) it leaves it down to you to define such things. This is possible using structures, but is more generally used in classes.

### Nesting Structures

As structures can contain any data type, it follows that a structure can **contain other structures**. This **nesting of structures** can be extremely useful when designing and creating the data representation for your programs, as it allows you to build up complex data types from simpler building blocks.

As a simple example, let's say a modelling agency wanted to store information about its models, storing a model ID number, age, height, weight and waist measurements. One way to do this would be to create a new structure like so:

struct Model

{

int m\_ID; // new member

int m\_age;

float m\_height\_metres;

float m\_weight\_kilograms;

float m\_waist\_cms;

};

This is a perfectly valid way to do this, but you might notice that this shares a great deal of similarity to the previously defined Person\_Infostructure. So instead of declaring the structure as shown above, a better way would be to incorporate the Person\_Infostructure, as demonstrated below:

struct Model

{

int m\_ID;

Person\_Info m\_info;

}

There does, of course, need to be a way to access the members of the inner Person\_Info structure m\_info. This is in fact extremely straightforward - simply use the dot operator again a **second** time, in this case after referencing info using a first dot operator:

Model client\_1;

client\_1.m\_ID = 236;

client\_1.m\_info.m\_age = 22;

You can also of course use the assignment operator to copy an entire structure into (or out of) this member in one go, e.g.:

Person\_Info bob = { 68, 170.0f, 92.2f, 80.7f };

Person\_Info copy\_of\_bob;

Model client\_2;

client\_2.m\_info = bob;

copy\_of\_bob = client\_2.m\_info;

When using an initialiser list to set the values of a structure that contains a structure you need to either pass in a variable of the same type of the inner structure, or use nested curly braces to correctly initialise the entire inner structure:

Person\_Info bob = { 68, 170.0f, 92.2f, 80.7f };

Model client\_3 = { 237, **bob** };

Model client\_4 = { 238, { 24, 165.5f, 55.0f, 63.9f } };

You can nest structures as many times as you like, structures within structures within structures, etc., and the way of referencing using the dot operator remains the same for each level in, so:

struct Modelling\_Session

{

int m\_locationID;

Model m\_assigned\_model;

};

would be referenced in the following way:

Modelling\_Session milan;

//... assign data to fill entire milan structure

int model\_ID = milan.m\_assigned\_model.m\_ID;

int model\_age = milan.m\_assigned\_model.m\_info.m\_age;

### Reusing Structures

Another useful feature is being able to **reuse structures** to build other data structures, either within the same program, or even possibly in a completely different program. You might at some point need to develop a medical application that needs to record similar information to the modelling program. Since you already have a Person\_Infostructure it might be possible to reuse it in this case:

struct Medical\_Info

{

bool m\_is\_alive;

Person\_Info m\_patient\_data; // reuse of Person\_Info

};

The real power of this feature will become more apparent when you start designing complex classes instead of just structures later in the unit.

### Arrays in Structures

Since you can use **any data type** as a member variable of a structure you can also include **arrays**, so you could use an array of chars to add a name member to the structure:

struct Person\_Info

{

char m\_name[50];

int m\_age;

float m\_height\_metres;

float m\_weight\_kilograms;

float m\_waist\_cms;

};

Again you would reference this using the dot operator, and then you can use that array member just like any other array:

Person\_Info bob = {"Bobby", 68, 170.0f, 92.2f, 80.7f }; // use char array initialise syntax

char first\_letter\_of\_name = bob.m\_name[0]; // address the first element of the array

Here is a good place to re-consider the difference between a **data type**, a **variable name**, and the **data contained within a variable**, as this is often a source of confusion for those new to programming. In the above code, Person\_Info is the data type. This means it is a blueprint for variables, not a variable in itself. Therefore using Person\_Info.m\_name[0] is meaningless, the use of the dot operator only makes sense with variable names created using the data type. A similar misunderstanding happens between variable names and data. In the above code, bob is a variable name of type Person\_Info, and **Bobby** is simply data contained within the name member of the data structure, so using Bobby.m\_name[0] would also be meaningless. Take the time to ensure that you are correctly using variables if you are having compilation problems, especially as the data structures begin to get more complex.

### Arrays of Structures

Just as you can have arrays inside structures, you can also have **arrays comprised of structure**s. New data types defined by structures can be treated in exactly the same way as any other type when it comes to declaring arrays. If you think back to the early part of this tutorial you might remember the problems with trying to group related data together using separate arrays. Now that you can use structures this becomes very easy to do.

As you may remember from the array tutorial you can use arrays to store a contiguous list of a single data type using a single variable name, and index into that array to access individual elements. You can do exactly the same using arrays of structures. Let's say you now wanted to store data for a lot of new models for your modelling agency. Instead of creating dozens of individual Model variables, you could define a Model **array** and **access its elements** as follows:

Model list\_of\_models[100];

list\_of\_models[0].m\_ID = 266;

list\_of\_models[0].m\_info.m\_age = 26;

...

... // etc

...

You simply index the required Model element from the array, and then use the dot operator as before to access the members of the structure.

You can also use the initialiser syntax exactly as before, though it does start getting complicated with more complex data types with structures and arrays that contain nested structures and arrays. It's a good idea to group the data using spaces and tabs to help visualise where the braces and commas should be, for instance:

Model top\_three\_models[3] = {

{ 1, { "Kate", 24, 1.5f, 47.0f, 52.3f } },

{ 2, { "Billie", 25, 1.6f, 48.0f, 53.3f } },

{ 3, { "Anna", 26, 1.7f, 49.0f, 54.3f } }

};

### Structures and Functions

Another place where structures can be of great use is for both **passing and returning data to and from functions**. Structures can be treated in the same way as other data types in that they can be used for both parameters and return values from functions. Using structures can often improve the readability of code where functions need to input or output a lot of data, so for instance a display information function defined as:

void display\_model\_info(int ID, int age, float height, float weight, float waist);

could be changed to use a structure instead:

void display\_model\_info(struct Model m);

In this case the structure is being passed by value, so all of the values of the structure are **copied** into a local copy of the variable inside the function. You need to be aware of the potential execution time cost of this operation, especially with large data structures. You might want to consider using pass by reference in this case:

void display\_model\_info(struct Model &m);

so the external structure is accessed directly from the function.

If you wanted to pass information back to the calling code you could use pass by reference to directly update the external structure, alternatively you can use the **return** type and keyword to do so:

Person\_Info return\_shortest(Person\_Info a, Person\_Info b)

{

if(a.m\_height\_metres > b.m\_height\_metres)

{

return b;

}

else

{

return a;

}

}

## Exercises

**In order to aid understanding of how the exercises work, use the debugger to step through all of the code you write for the exercises.**

**You are advised to write notes on all aspects of the tutorial and exercises in your logbooks. This can then be used to help with your assignments.**

#### Exercise 01

Create a new project as usual, then copy the following **basic code** into **main.cpp**, to use as the basis for the exercise:

#include <iostream>

using std::cin;

using std::cout;

using std::endl;

using std::string;

// DECLARE structures

struct Person\_Info

{

int m\_age;

float m\_height\_metres;

float m\_weight\_kilograms;

float m\_waist\_cms;

string m\_name;

};

// DECLARE functions

void wait\_for\_keypress(void);

int main()

{

Person\_Info dave; // creates a variable called dave, of type Person\_Info

wait\_for\_keypress();

}

// DEFINE functions

void wait\_for\_keypress(void)

{

cout << "Press Enter to continue" << endl;

cin.get();

}

1. Change the name of the variable dave to one **of your own choice**.
2. Set values **of your own choice** to **ALL** of the variable members. You will need to encapsulate the string in **“quotation marks”**.
3. Output the value of each member of the structure to the screen on a separate line.
4. Create a second Person\_Info variable with a different name, e.g. dave2, but this time declare its values using **initialiser syntax** (and different values). Output the value of each member of the new structure to the screen.
5. Create a third Person\_Info variable with another name using **initialiser syntax** (and different values), but only initialise the **first two** elements, m\_age and m\_height\_metres. Output each member of the new structure to the screen. Does it do what you expect? Use comments to explain what happens.
6. Initialise all the elements of a fourth Person\_Info variable, and output them to screen. Use the **assignment operator** to assign the first Person\_Info structure you declared to this new one, e.g. dave4 = dave. Output the fourth Person\_Info members to the screen again. Explain what has happened in comments.

#### Exercise 02

Once you have completed exercise 01 and it all works correctly, create a new project for exercise 02, then copy **all of the code** from the **main.cpp** of exercise 01 into the **main.cpp** of exercise 02. Check that it all still works before beginning the exercise.

1. Create a Model structure that contains an integer m\_ID, a char **array** called m\_location and a Person\_Info member called m\_info. Make sure the location array is big enough to contain any reasonable place name.
2. Create a Model variable with a name of your choosing, e.g. client\_1, and assign values to **all** of the members, including the m\_location array and **all** members in the inner Person\_Info structure.
3. Output all of the model member variable values to the screen. Check using the debugger that all values are being output.
4. Create a second Model, this time **initialising each individual member** at declaration. Output this new model's data.
5. Create a third Model, this time initialising the Person\_Info inner structure using one of the previously declared Person\_Info variables, e.g. dave,at declaration. Output this new model's data, checking that the values correctly match the variable used to initialise the inner structure.
6. Using the third model again, use the assignment operator to assign a **different** Person\_Info variable (e.g. dave2) to the m\_info member. Output the model data, and explain in comments what has happened.
7. Create an **array of five** Models. Use initialiser syntax to assign values to all elements of all of the models in the array, then use a loop to output all the values to the screen to check that everything has been correctly set.
8. Use assignment syntax to replace **one of the array of Model’s** elements to one of the previously created models. Output the array element to inspect if it has correctly changed.

#### Exercise 03

Once you have completed exercise 02 and it all works correctly, create a new project for exercise 03, then copy **all of the code** from the **main.cpp** of exercise 02 into the **main.cpp** of exercise 03. Check that it all still works before beginning the exercise.

1. Declare and define a function called display\_person\_info() that takes a Person\_Info structure as a parameter and has no return value.
2. In the body of the function output all of the Person\_Info data to the screen.
3. Test the function by replacing all of the code that displays Person\_Info data in main() by calling the function with the Person\_Info variable instead.
4. Create a second function called display\_model\_info() that takes a Model structure as a parameter and has no return value.
5. In the body of this function output all of the Model's data to the screen. Consider how you might use display\_person\_info()in this function to simplify the task.
6. Replace all the relevant code in your program with the new function, and test that it all still works.
7. Both these functions use pass by value to copy the argument passed in to a local variable. Explain in your logbook why this might not be the best way of writing the function, and how it could be improved. Implement the improvement using two **new** functions, and test that they work by replacing some or all of the old ones.

#### Exercise 04

1. Create a structure that can store the various different pieces of information required for a postal address.
2. Create a second structure that represents someone's first and last names.
3. Create a third structure that uses the first two structures to create a name and address data type.
4. Declare an array with the type of this third structure.
5. Write a function that inputs first and last names from the user that takes no arguments and returns a name structure.
6. Write a function that inputs address information from the user that takes no arguments and returns an address structure.
7. Use a loop to repeatedly ask for names and addresses, and use the return values from the functions to update each 'name and address' array element until the array is full.
8. Output the entire name and address array to screen.

#### Exercise 05

Create a new project, and copy the **basic code** from the start of exercise 01 listed above into **main.cpp**. Design a data structure - you can reuse the Person\_Info structure - which can store information about a track athlete, including at least their nationality, the distance they run, the fastest time they have run over that distance, and the number of times they have run this year. Add more if you wish. Carefully consider and justify what types these variables should be, and what structures they should be in. Implement and test your design.

There are several legitimate ways of designing this data structure. Try to create another data structure that stores the same information, but has some of the member variables arranged differently, i.e. they will be moved in or out of a nested structure. Discuss which solution you think is better, and why.