# Programming Notes 12: Structs

As usual, more examples of the notes are in the example program.

1. Abstract data types

As of yet, we've only used the basic, or primitive data types that are built into C++, as well as a few more advanced types for topics such as file IO. However, I'm sure you've noticed by now that it becomes very complicated and lengthy to work with large quantities of data, even when using arrays. For example, if you need to pass a function an x, y, and z coordinate, you have to write out three parameters and specify three parameters when you call it. However, the most powerful feature of object-oriented languages such as C++ is the ability to define and use your own datatypes (although some other languages, such as C, provide much less powerful ways to do the same). In fact, in very high level languages such as Python, literally everything is one of these advanced datatypes, from basic integer numbers to complicated data structures.

2. Structures

These datatypes are called "abstract" data types, because you the programmer can define them yourself to your specifications. In C++, the most common way to implement an abstract datatype is through classes, which are the basic component of object-oriented programming, or OOP. However, classes are quite complicated, and we won't get into them for a couple more weeks. First, we're going to learn about structures, (more commonly referred to as simply "structs"), which are more or less extremely barebones classes. However, although I've probably just made abstract datatypes sound like some sort of magical solution to all your problems, you must realize that even the most complicated ADTs are, at their core, made up of the basic primitive types of C++. Of course, one ADT can talk about another ADT, and that type can talk about yet another ADT, but eventually, if you drill down to the basic data contained in each type, it will be primitive data.

So, what exactly is a structure in C++? Of course, it's a basic ADT, but what does it really do? Well, it's just like any other datatype, like an int, or a double, or even a bool, in that it simply stores data, but the important part is that it can hold multiple pieces of data, and what data it holds is completely up to you. To continue the previous example of passing three values describing a coordinate to a function, with structs, you can instead pass the function a structure that you've described—remember, these are just like the normal data types in most usage—which itself contains the three values.

3. Defining structs

The actual syntax of defining a structure is quite simple: to start, type the keyword "struct." Then, type what is called the type name of your structure. This is different than an identifier (the name of a variable), because what you're doing is not declaring a variable, but describing what a new data type will contain. Hence, the type name will be what your new type is called—for example if you created a struct with the type name "student," you can now create variables of *type* "student," with any name you wish. Then, add a pair of curly brackets, a semicolon after the closing one, and you're technically done! However, your new data type won't be very useful if it doesn't hold any data. To describe what data your new type will hold, simply add values within your brackets as if you were declaring

variables. These are called data members, because they "belong" to the ADT that is your new structure. Finally, remember that these aren't actually variables—they're simply describing what will be in your type, and will only be created later. This data can be of any primitive or previously described ADT.

For example, if we were to create a basic struct type called "student"…

```
struct student {
  string name;
  float gpa;
};
```

This declares what a "student" is, so now you can declare variables of type "student" later in your code. This is called creating instances of your ADT. For example…

```
int main() {
  student aStudent;
  //Creates a new variable of type "student," or in other words a new instance of "student."
  Because you described a "student" previously, this variable will hold a string and a float.

  student anotherStudent;
  //You can create as many of these as you want, just like you would with, for example, integers
}
```

4. Using structs

That's all well and good, but your special new datatype wouldn't be very useful if you couldn't access the data contained within them—right now, you just have a variable of your type, but no way to actually use them. Enter the dot operator ("."), which allows you to manipulate the data members of your structure. This is where the names of your data members come into play—they will be the same for all instances of your ADT, and to access them you simply type their identifier after the dot operator. This new statement will act exactly as if you simply had a variable of the member data type, meaning you can assign values to it, test it, read values from it, etc. Note that you cannot use the dot operator on the actual name of your type ("student" for example), because the type name simply describes what that type will contain, and does not actually hold data. You need an instance of that type to access data, because it will actually contain the described information. For example:

```
int main() {
  student aStudent;
  aStudent.name = "Steve";
  aStudent.gpa = 3.7;

  cout << "Student name: " << aStudent.name << endl;
}
```

Lastly, because your ADT works like any other datatype, you can pass it to functions, pass it by reference, and do (almost) everything else you would do with a normal primitive data type.

5. Pointers, dereferencing, and the arrow operator

Now that you know how to more or less use your new ADTs, you should know about using pointers to ADTs. As expected, they work in more or less exactly the same ways as pointers to primitive data types. You declare them in the same way as other pointers, and dereferencing them will give you the actual data that the pointer points to. However, you must realize that the data you get from dereferencing an ADT pointer is still of the abstract type, and acts like one. This means that if you deference an ADT pointer, you must then use the dot operator to access the data within. For example, if we have:

```
student myStudent;
student* stuPtr = &myStudent;
```

*stuPtr will give you data of "student" type. However, because of how the order of operations is structured in C++, if you were to write something like this,

```
cout << *stuPtr.name << endl;
```

you will get a complier error, because your program is trying to use the dot operator before the dereference operator, and using the dot operator on a pointer doesn't make sense. Hence, you must force it to dereference first with parenthesis:

```
cout << (*stuPtr).name << endl;
```

Which works as expected. However, this syntax is quite clunky and annoying to type, so there is another way to do this. This is through the arrow operator ("->"). The arrow operator does the exact same logical operations as dereferencing the value in parenthesis, then using the dot operator. However, it is much easier and cleaner to work with. For example:

```
cout << stuPtr->name << endl;
```

Works in exactly the same way as the previous example.

6. Dynamic Memory and ADTs

Finally, there's the last little topic of dynamic memory with ADTs. First, there is the simpler part—creating dynamic arrays of ADTs is exactly the same as creating dynamic arrays of any other type. You use the same "new" and "delete" syntax. However, because structures can contain any data type, including pointers, it follows that they can contain their own dynamically allocated memory. This memory is not any different than other dynamic memory we've worked with, except that its address is stored within an instance of your ADT. This means that to create the memory, you must assign the address to the correct data member within an instance of your ADT and eventually delete it as well. When we get into classes, this will become much cleaner, but we're not there yet.

For example, if you want to include two dynamically allocated c-style strings in your "student" type:

```
struct student {
    char* firstName;
```

```
    char* lastName;
    float gpa;
};

int main() {
    student s1;

    //Add the dynamic memory
    s1.firstName = new char[10];
    s1.lastName = new char[20];

    //Assign the name
    strcpy(s1.firstName, "Steve");
    strcpy(s1.lastName, "Irwin");

    //Output the name
    cout << "Student name: " << s1.firstName << " " << s1.lastName << endl;

    //Always remember to delete an dynamic memory
    delete[] s1.firstName;
    delete[] s1.lastName;
}
```