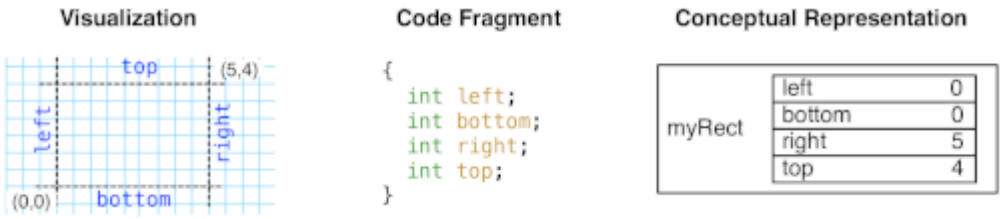
Structs

coursera.org/learn/programming-fundamentals/supplement/9LmqH/structs

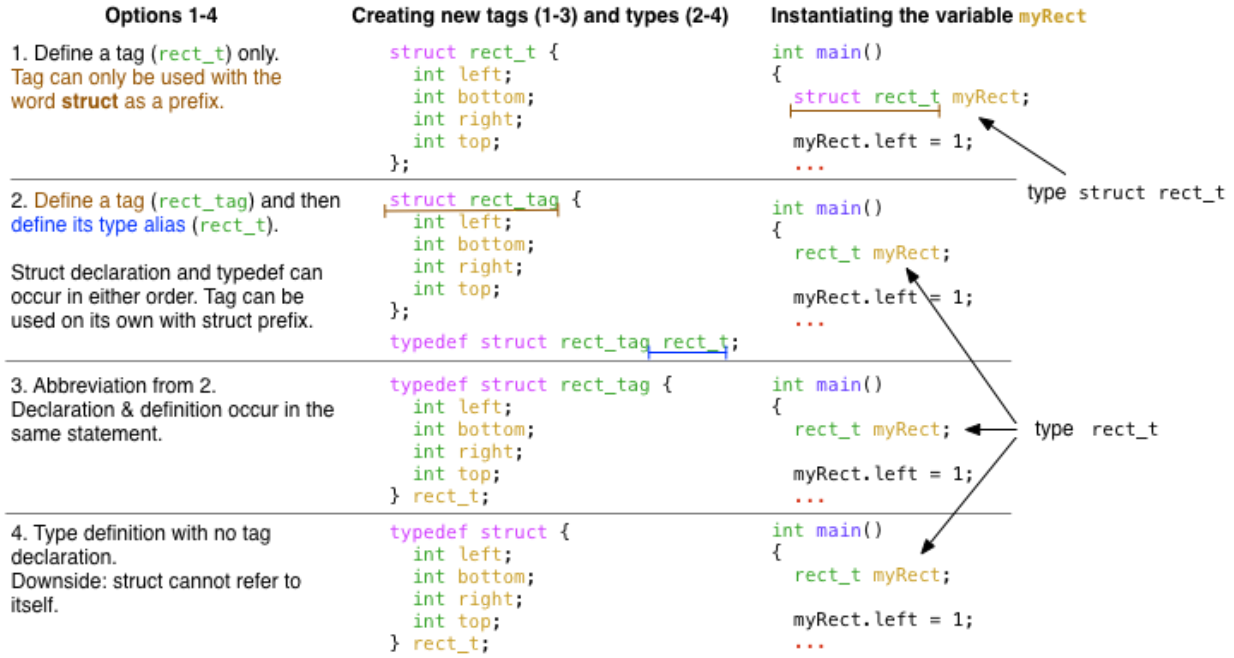
You may be starting to notice that the definitions of many data types are essentially a set of agreed upon conventions. One of the great things about rich programming languages like C is that they give a programmer the power to create new data types and associated conventions. Some conventions, like the IEEE floating point standard, are agreed upon across multiple programming languages, compilers, machine languages, and the architecture of the processors they run on. This requires the coordination of hundreds of companies and tens of thousands of engineers. Other conventions can be more local, existing only in a particular code base, or a collection of files that all use a common library. This may require the coordination of multiple people (who are usually working together already) or may only affect a single person who simply wishes to produce clean, modifiable, and debuggable programs.

Suppose you are designing a program that regularly draws and computes various properties of rectangles. It would be very convenient to have a data type that captures the basic properties of a rectangle. In C, this is accomplished via the keyword **struct.**



A *struct* allows a programmer to bundle multiple variables into a single entity. For example, if we wish to define a rectangle via its 4 coordinates on an X-Y plane as shown in the left-most figure above (Visualization), these four coordinates can be bundled into a single, conglomerate data structure, whose internal structure will look like the code in center figure above (Code Fragment). Structs are represented conceptually with a single box in which all the component fields reside, each with their own box. The right-most figure above (Conceptual Representation) shows a variable called myRect with its 4 fields.

Syntactically, there are multiple ways to declare, define, and use structs. The figure below shows four different syntactic options that all create the same conceptual struct. Regardless of which syntactic option you choose, the drawing of your conceptual representation will be the same. It is not important for you to be “fluent” in all four options. You may choose a single approach and stick with it. However, it is important for you to know about all four options because others contributing to the same code base as you may have a different style, and internet searches will also result in many versions of the effectively the same code. You need to be aware of these differences so that you can correctly understand and extend code whose syntax differs from your preferred style.



Struct declarations do not go inside functions; they live in the global scope of the program, next to function declarations and definitions. All of them use the keyword **struct**. Option 1 in the figure above begins with the keyword struct, followed by the tag of our choosing. In this case, we use the tag rect_t. Ending the tag in "_t" is a convention that makes it easier to recognize the name as identifying a type throughout your code. A name such as rect would be acceptable, just a little less reader-friendly. Everything inside the braces belongs to the definition of the newly defined struct named rect_t. The semi-colon indicates the completion of the definition.

The far right column in the above figure shows how to instantiate a variable for each syntactic option. For Option 1, the type of the variable is struct rect_t, and the name of the variable is myRect. Once you declare the variable, you can access the fields of the struct using a dot (period): myRect.top gives you access to the field top of type int inside the myRect variable. Note: when you instantiate a variable of type struct rect_t, you choose a top level name (myRect) only. The names of the fields are determined in the definition of the structure and cannot be customized on a per-instance basis.

The following video shows an example of a declaration of a struct for a rectangle, as well as its initialization and use. Note that the assignment statements follow the same basic rules we have seen so far: you find the box named by the left side, evaluate the right side to a value, and store that value into the box named by the left side. The dot operator just gives you a different way to name a box—you can name a "box inside a box."

A key part of good programming is using good abstractions. Structs are another form of abstraction. Once we have a rectangle struct, other pieces of code can operate on rectangles without looking at the implementation. We could write many functions to manipulate rectangles, and those functions could be the only pieces of code that know the internal details of rectangles—the rest of the code could just call those functions.

However, part of using good abstractions is using them correctly. In the case of structs, remember that their primary purpose is to group together data that belongs logically together. In this example, we use a struct for a rectangle—something that logically makes sense as a combination of other pieces of data. In the first figure above we illustrate the connection between the conceptual idea (the visualization on the left), and the declaration in the middle. We can think about operations on rectangles and understand what they are conceptually, without looking at the implementation details.

While it may seem silly to say: do not just group data together into structs without a logical purpose. Sometimes novice programmers are tempted to just put a bunch of things in a struct because they get used in the same parts of a program (to pass one parameter around instead of a few). However, if you cannot articulate why those data make sense together, they do not belong in a struct together.