

Typedef | Coursera

 coursera.org/learn/programming-fundamentals/supplement/Xkk9A/typedef

Typedef

Typedef

Options 1-4	Creating new tags (1-3) and types (2-4)	Instantiating the variable <code>myRect</code>
1. Define a tag (<code>rect_t</code>) only. Tag can only be used with the word struct as a prefix.	<pre>struct rect_t { int left; int bottom; int right; int top; };</pre>	<pre>int main() { struct rect_t myRect; myRect.left = 1; ... }</pre> <p>type struct rect_t</p>
2. Define a tag (<code>rect_tag</code>) and then define its type alias (<code>rect_t</code>). Struct declaration and typedef can occur in either order. Tag can be used on its own with struct prefix.	<pre>struct rect_tag { int left; int bottom; int right; int top; }; typedef struct rect_tag rect_t;</pre>	<pre>int main() { rect_t myRect; myRect.left = 1; ... }</pre> <p>type rect_t</p>
3. Abbreviation from 2. Declaration & definition occur in the same statement.	<pre>typedef struct rect_tag { int left; int bottom; int right; int top; } rect_t;</pre>	<pre>int main() { rect_t myRect; myRect.left = 1; ... }</pre> <p>type rect_t</p>
4. Type definition with no tag declaration. Downside: struct cannot refer to itself.	<pre>typedef struct { int left; int bottom; int right; int top; } rect_t;</pre>	<pre>int main() { rect_t myRect; myRect.left = 1; ... }</pre> <p>type rect_t</p>

Many consider Option 1 in the figure above to be somewhat unwieldy, because the type of the variable includes the word `struct` in it. For example, suppose you wanted a function called `shrinkRect` that takes a rectangle as its input and returns a smaller rectangle as its output. Using the syntax of Option 1, the function would have the signature **`struct rect_t shrinkRect(struct rect_t shrinkThisRectangle)`**. Depending on how often you need to write out the type of the structure, this syntax can become cumbersome and make your code appear cluttered.

The solution to needing to type out “`struct rect_t`” every time you want to declare, pass, or use your new struct is to create a new data type that is explicitly of type `struct`. We do this using the keyword `typedef`. The exact syntax is shown in Option 2 above. The first lines declare the `rect_tag` struct in the same way as before. However, after this struct definition, the last line (`typedef struct rect_tag rect_t;`) is the declaration of the type “`rect_t`” which is defined as having the type “`struct rect_tag`”. Using “`_tag`” makes code easier to read and encourages the use of the type over the tag. Options 3 and 4 also “`typedef`” a new type, however, they both combine the typedef into a single statement with the structure declaration.

Although typedefs can simplify the use of structs, that is far from their only use. Any time that you are writing code in a specific context, typedefs can help you make your code more readable, by naming a type according to its meaning and use. For example, suppose you

are writing a program that deals with colors.

In the context of programming color characteristics, you might want to define a new data type for the colors in an RGB value. For example, you could create a new data type called `rgb_t` (which represents one of the red, green, or blue components of the color), that is of type `unsigned int` (because we know the values should be positive integers) and then declare variables `red`, `green`, and `blue` of type `rgb_t`. An example of this is shown on the left side of the figure below.

The diagram illustrates a change in the definition of the `rgb_t` typedef. On the left, the original code defines `rgb_t` as `unsigned int`. On the right, an arrow labeled "change definition of rgb_t" points to the modified code where `rgb_t` is defined as `unsigned char`. The rest of the code, including the function definitions and the `main` function, remains unchanged.

```
typedef unsigned int rgb_t;

rgb_t getRedForPixel(int x, int y) {...}
rgb_t getGreenForPixel(int x, int y) {...}
rgb_t getBlueForPixel(int x, int y) {...}

int main(void) {
    rgb_t red, green, blue;

    red = getRedValue();
    green = getGreenValue();
    ...
}
```

change definition of rgb_t

```
typedef unsigned char rgb_t;
// nothing else changes!
```

Typedefs provide a helpful abstraction for programmers. Instead of having to write “unsigned int” throughout her code, or frankly even think about the range of acceptable values in RGB representations, the programmer simply uses the custom type `rgb_t` and gives it no further thought.

Typedefs have another nice property of limiting the definition of a particular type to a single place in the code base. Suppose a programmer wished to conserve the space dedicated to variables and therefore wished to use an unsigned char instead of an unsigned int (after all, the values from 0 to 255 all fit within the 8-bits of an unsigned char). Without a typedef, this change would require a tedious and error-prone search of many (but by no means all—it may be used for variables unrelated to colors) instances of unsigned int throughout the code, changing these types to unsigned char. With a typedef, the programmer simply changes the single line of code in which `rgb_t` was defined (see the right side of the figure). No other code changes are required.

Heads up about typedef: The use of typedefs is somewhat controversial in some programming circles. In the context of structs, there are those who believe that it is important not to abstract the struct away from a type. They believe that programmers should always know when a particular variable is a struct and when it is not. Similarly, they believe that programmers should always be aware of the actual types of the data they use lest they fall prey to typing errors that could have been otherwise avoided. Use typedefs when the abstraction simplifies rather than obfuscates your code.