4.6 — Typedefs and type aliases

BY ALEX ON JUNE 19TH, 2007 | LAST MODIFIED BY ALEX ON FEBRUARY 27TH, 2018

Typedefs allow the programmer to create an alias for a data type, and use the aliased name instead of the actual type name. To declare a typedef, simply use the *typedef* keyword, followed by the type to alias, followed by the alias name:

```
typedef double distance_t; // define distance_t as an alias for type double

// The following two statements are equivalent:
double howFar;
distance_t howFar;
```

By convention, typedef names are declared using a "_t" suffix. This helps indicate that they are types, not variables, and also helps prevent naming collisions with similarly named variables.

A typedef does not define a new type. Rather, it is simply an alias (another name) for an existing type. A typedef can be used interchangeably anywhere a regular type can be used.

Even though the following does not make sense conceptually, it is valid C++:

```
typedef long miles_t;
typedef long speed_t;

miles_t distance = 5;
speed_t mhz = 3200;

// The following is valid, because distance and mhz are both actually type long
distance = mhz;
```

Typedefs are useful in a number of different situations.

Using typedefs for legibility

One use for typedefs is to help with documentation and legibility. Data type names such as char, int, long, double, and bool are good for describing what *type* a function returns, but more often we want to know what *purpose* a return value serves.

For example, consider the following function:

```
int GradeTest();
```

We can see that the return value is an integer, but what does the integer mean? A letter grade? The number of questions missed? The student's ID number? An error code? Who knows! Int does not tell us anything.

```
typedef int testScore_t;
testScore_t GradeTest();
```

However, using a return type of testScore_t makes it obvious that the function is returning a type that represents a test score.

Using typedefs for easier code maintenance

Typedefs also allow you to change the underlying type of an object without having to change lots of code. For example, if you were using a short to hold a student's ID number, but then later decided you needed a long instead, you'd have to comb through lots of code and replace short with long. It would probably be difficult to figure out which shorts were being used to hold ID numbers and which were being used for other purposes.

However, with a typedef, all you have to do is change `typedef short studentID_t` to `typedef long studentID_t`. However, precaution is necessary when changing the type of a typedef to a type in a different type family (e.g. an integer to a floating point value, or vice versa)! The new type may have comparison or integer/floating point division issues that the old type did not.

Platform independent coding

One big advantage of typedefs is that they can be used to hide platform specific details. On some platforms, an integer is 2 bytes, and on others, it is 4. Thus, using int to store more than 2 bytes of information can be potentially dangerous when writing platform independent code.

Because char, short, int, and long give no indication of their size, it is fairly common for cross-platform programs to use typedefs to define aliases that include the type’s size in bits. For example, int8_t would be an 8-bit signed integer, int16_t a 16-bit signed integer, and int32_t a 32-bit signed integer. Using typedef names in this manner helps prevent mistakes and makes it more clear about what kind of assumptions have been made about the size of the variable.

In order to make sure each typedef type resolves to a type of the right size, typedefs of this kind are typically used in conjunction with the preprocessor:

```
#ifdef INT_2_BYTES
typedef char int8_t;
typedef int int16_t;
typedef long int32_t;
#else
typedef char int8_t;
typedef short int16_t;
typedef int int32_t;
#endif
```

On machines where integers are only 2 bytes, INT_2_BYTES can be #defined, and the program will be compiled with the top set of typedefs. On machines where integers are 4 bytes, leaving INT_2_BYTES undefined will cause the bottom set of typedefs to be used. In this way, int8_t will resolve to a 1 byte integer, int16_t will resolve to a 2 bytes integer, and int32_t will resolve to a 4 byte integer using the combination of char, short, int, and long that is appropriate for the machine the program is being compiled on.

In C++11, this is actually how the fixed width integers (like int8_t) were defined! As a side-effect of the fact that int8_t is actually a typedef of char, the following code acts somewhat unexpectedly:

```
#include <cstdint> // for fixed-width integers
#include <iostream>

int main()
{
    std::int8_t i(97); // int8_t is actually a typedef for char
    std::cout << i;

    return 0;
}
```

This program prints:

```
a
```

not 97, because std::cout prints char as an ASCII character, not a number.

Using typedefs to make complex types simple

Although we have only dealt with simple data types so far, in advanced C++, you could see a variable and function declared like this:

```
std::vector<std::pair<std::string, int> > pairlist;

bool hasDuplicates(std::vector<std::pair<std::string, int> > pairlist)
{
    // some code here
}
```

Typing `std::vector<std::pair<std::string, int> >` everywhere you need to use that type can get cumbersome. It’s much easier to use a typedef:

```
typedef std::vector<std::pair<std::string, int> > pairlist_t; // make pairlist_t an alias for this crazy
 type

pairlist_t pairlist; // instantiate a pairlist_t

bool hasDuplicates(pairlist_t pairlist) // use pairlist_t in a function parameter
```

```
{
    // some code here
}
```

Much better! Now we only have to type “pairlist_t” instead of `std::vector<std::pair<std::string, int> >`.

Don’t worry if you don’t know what std::vector, std::pair, or all these crazy angle brackets are yet. The only thing you really need to understand here is that typedefs allow you to take complex types and give them a simple name, which makes those types easier to work with and understand.

Type aliases in C++11

Typedefs have a few issues. First, it’s easy to forget whether the type name or type definition come first. Which is correct?

```
typedef distance_t double; // incorrect
typedef double distance_t; // correct
```

I can never remember.

Second, the syntax for typedefs gets ugly with more complex types (as we’ll explore further in the section on function pointers).

To help address these issues, in C++11, a new, improved syntax for typedefs has been introduced that mimics the way variables are declared. This syntax is called a **type alias**. A **type alias** introduces a name that can be used as a synonym for a type.

Given the following typedef:

```
typedef double distance_t; // define distance_t as an alias for type double
```

In C++11, this can be declared as:

```
using distance_t = double; // define distance_t as an alias for type double
```

The two are functionally equivalent.

Note that although the C++11 syntax uses the “using” keyword, this is an overloaded meaning, and does not have anything to do with the using statements related to namespacing.

This new syntax is cleaner for more advanced typedefing cases, and should be preferred if your compiler is C++11 capable.

Rule: Favor type aliases over typedefs if your compiler is C++11 compatible.

Quiz time

1) Given the following function prototype:

```
int printData();
```

1a) Convert the int return value to a typedef named error_t using the typedef keyword. Include both the typedef statement and the updated function prototype.

Hide Solution

typedef int error_t;
error_t printData();

1b) Convert the int return value to a typedef named error_t using the using keyword (C++11). Include both the typedef statement and the updated function prototype.

Hide Solution

using error_t = int;
error_t printData();

4.7 -- Structs