6.13 — Typedefs and type aliases

BY ALEX ON JUNE 19TH, 2007 | LAST MODIFIED BY NASCARDRIVER ON JANUARY 9TH, 2020

Typedefs allow the programmer to create an alias for a data type, and use the aliased name instead of the actual type name. Typedef literally stands for, "type definition".

To declare a typedef, simply use the `typedef` keyword, followed by the type to alias, followed by the alias name:

```cpp
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 the identifier represents a type, not a variable or function, and also helps prevent naming collisions with other identifiers.

Note that 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 semantically, it is valid C++:

```cpp
int main()
{
    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;

    return 0;
}
```

`typedefs` and type aliases follow the same scoping rules as variables. The `typedefs` `miles_t` and `speed_t` are only usable in the `main()` function. If they were placed in another function, `main()` wouldn't be able to access them. If they were placed outside of `main()`, in the global scope, all functions would be able to access them.

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

```cpp
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, particularly function pointers (which we will cover in future lesson **7.8 -- Function Pointers**):

Type aliases

To help address these issues, an improved syntax for `typedefs` has been introduced that mimics the way variables are declared. This syntax is called a **type alias**.

Given the following typedef:

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

This can be declared as the following type alias:

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

The two are functionally equivalent.

Note that although the type alias syntax uses the "using" keyword, this is an overloaded meaning, and does not have anything to do with the `using statements` related to namespaces.

This type alias syntax is cleaner for more advanced typedefing cases, and should be preferred.

Using type aliases for legibility

One use for type aliases 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.

```
using testScore_t = int;
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 type aliases for easier code maintenance

Type aliases 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 type alias, all you have to do is change `using studentID_t = short;` to `using studentID_t = long;`. However, caution is still necessary when changing the type of a type alias 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, or other issues that the old type did not.

Using type aliases for platform independent coding

Another advantage of type aliases is that they can be used to hide platform specific details. On some platforms, an `int` is 2 bytes, and on others, it is 4 bytes. 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 type aliases 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 type aliases 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 aliased type resolves to a type of the right size, type aliases of this kind are typically used in conjunction with preprocessor directives:

```
#ifdef INT_2_BYTES
using int8_t = char;
using int16_t = int;
using int32_t = long;
#else
using int8_t = char;
using int16_t = short;
using int32_t = int;
#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 type aliases. On machines where integers are 4 bytes, leaving INT_2_BYTES undefined will cause the bottom set of type aliases 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.

This is exactly how the fixed width integers (like int8_t) that were introduced in C++11 (covered in lesson **4.6 -- Fixed-width integers and size_t**) are defined!

This is also where the issue with int8_t being treated as a char comes from -- int8_t is a type alias of char, and thus is just an alias for a char rather than being a unique type. As a result:

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

int main()
{
    std::int8_t i(97); // int8_t is actually a type alias 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 type aliases 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 type alias:

```
using pairlist_t = std::vector<std::pair<std::string, int> >; // make pairlist_t an alias for t

```

```cpp
pairlist_t pairlist; // instantiate a pairlist_t variable

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 type aliases allow you to take complex types and give them a simple name, which makes those types easier to work with and understand.

Best practice

Favor type aliases over typedefs, and use them liberally to document the meaning of your types.

Quiz time

Question #1

Given the following function prototype:

```cpp
int printData();
```

Convert the int return value to a type alias named error_t. Include both the type alias statement and the updated function prototype.

Show Solution

6.14 -- The auto keyword

Index

6.12 -- Using statements

C++ TUTORIAL | PRINT THIS POST

77 comments to 6.13 — Typedefs and type aliases

« Older Comments 1 2

Patrick
January 16, 2020 at 6:12 am · Reply

Hey, thanks for the very thorough tutorials.

I tried to define a type alias like this

```
using globalconstdouble_t = static constexpr double;
```

This results in "error: expected type-specifier before 'static'".

Does anybody know why this does not work?

Can keywords like "static" or "constexpr" even be used in a type alias definition?

nascardriver
January 16, 2020 at 6:48 am · Reply

> Can keywords like "static" or "constexpr" even be used in a type alias definition?
No, they can't. That would makes declarations very confusing.

« Older Comments 1 2