8.6 — Typedefs and type aliases

ALEX SEPTEMBER 5, 2021

Type aliases

In C++, using is a keyword that creates an alias for an existing data type. To create such an alias, we use the `using` keyword, followed by a name for the alias, followed by an equals sign and an existing data type. For example:

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

By convention, type alias names use a "_t", "_type", or no, suffix. The suffixes both indicate that the name is an alias, and helps prevent naming collisions with other types of identifiers.

```
using distance_type = double; // Also ok, more about this in a later chapter
using distance = double; // Also ok, but could be confused for- and collide with variable names
```

Once defined, an alias can be used anywhere a type is needed. For example, we can create a variable with the alias name as the type:

```
distance_t milesToDestination{ 3.4 }; // defines a variable of type double
```

When the compiler encounters an alias name, it will substitute in the aliased type. For example:

```
#include <iostream>

int main()
{
    using distance_t = double; // define distance_t as an alias for type double

    distance_t milesToDestination{ 3.4 }; // defines a variable of type double

    std::cout << milesToDestination << '\n'; // prints a double value

    return 0;
}
```

This prints:

```
3.4
```

In the above program, we first define `distance_t` as an alias for type `double`.

Next, we define a variable named `milesToDestination` of type `distance_t`. Because the compiler knows `distance_t` is an alias, it will use the aliased type, which is `double`. Thus, variable `milesToDestination` is actually compiled to be a variable of type `double`, and it will behave as a `double` in all regards.

Finally, we print the value of `milesToDestination`, which prints as a `double` value.

Type aliases are not new types

An alias does not actually define a new type -- it just introduces a new identifier for an existing type. An alias is completely interchangeable with the aliased type.

This allows us to do things that are syntactically valid but semantically meaningless. For example:

```
int main()
{
    using miles_t = long; // define miles_t as an alias for type long
    using speed_t = long; // define speed_t as an alias for type long

    miles_t distance { 5 }; // distance is actually just a long
    speed_t mhz  { 3200 };  // mhz is actually just a long

    // The following is syntactically valid (but semantically
meaningless)
    distance = mhz;

    return 0;
}
```

Although conceptually we intend `miles_t` and `speed_t` to have distinct meanings, both are just aliases for type `long`. This effectively means `miles_t`, `speed_t`, and `long` can all be used interchangeably. And indeed, when we assign a value of type `speed_t` to a variable of type `miles_t`, the compiler only sees that we're assigning a value of type `long` to a variable of type `long`, and it will not complain.

Because the compiler does not prevent these kinds of semantic errors for type aliases, we say that aliases are not type safe. In spite of that, they are still useful.

Warning

Care must be taken not to mix values of aliases that are intended to be semantically distinct.

As an aside...

Some languages support the concept of a strong typedef (or strong type alias). A strong typedef actually creates a new type that has all the original properties of the original type, but the compiler will throw an error if you try to mix values of the aliased type and the strong typedef. As of C++20, C++ does not directly support strong typedefs (though enum classes, covered in lesson 9.3 -- Enum classes, are similar), but there are quite a few 3rd party C++ libraries that implement strong typedef-like behavior.

The scope of a type alias

Because scope is a property of an identifier, type alias identifiers follow the same scoping rules as variable identifiers: a type alias defined inside a block has block scope and is usable only within that block, whereas a type alias defined in the global namespace has file scope and is usable to the end of the file. In the above example, `miles_t` and `speed_t` are only usable in the `main()` function.

If you need to use one or more type alias across multiple files, they can be defined in a header file and #included into any code files that needs to use the definition:

mytypes.h:

```
#ifndef MYTYPES
#define MYTYPES

    using miles_t =
long;
    using speed_t =
long;

#endif

```

Type aliases #included this way will be imported into the global namespace and thus have global scope.

Typedef

typedef (which is short for "type definition") is a keyword with the same semantics as "using", but reversed syntax.

```
// The following aliases are
identical
typedef long miles_t;
using miles_t = long;
```

Typedefs are still in C++ for historical reasons, but their use is discouraged.

Typedefs have a few syntactical issues. First, it's easy to forget whether the *typedef name* or *aliased type name* come first. Which is correct?

```
typedef distance_t double; // incorrect (typedef name first)
typedef double distance_t; // correct (aliased type name
first)
```

It's easy to get backwards. Fortunately, in such cases, the compiler will complain.

Second, the syntax for typedefs can get ugly with more complex types. For example, here is a hard-to-read typedef, along with an equivalent (and slightly easier to read) type alias with "using":

```
typedef int (*fcn_t)(double, char); // fcn_t hard to
find
using fcn_t = int(*)(double, char); // fcn_t easier to
find
```

In the above typedef definition, the name of the new type (`fcn_t`) is buried in the middle of the definition, making the definition hard to read.

Third, the name "typedef" suggests that a new type is being defined, but that's not true. As we have seen above, an alias is interchangeable with the aliased type.

Best practice

When creating aliased types, prefer the type alias syntax over the typedef syntax.

When should we use type aliases?

Now that we've covered what type aliases are, let's talk about what they are useful for.

Using type aliases for platform independent coding

One of the uses for 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.

The fixed-width integers (such as `std::int_fast16_t` and `std::int_least32_t`) and `size_t` type (both covered in lesson 4.6 -- Fixed-width integers and size_t) are actually just type aliases to various fundamental types.

This is also why when you print an 8-bit fixed-width integer using `std::cout`, you're likely to get a character value. For example:

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

int main()
{
    std::int_least8_t x{ 97 }; // int_least8_t is actually a type alias for a char type
    std::cout << x;

    return 0;
}
```

This program prints:

```
a
```

Because `std::int_least8_t` is typically defined as a type alias for one of the char types, variable `x` will be defined as a char type. And char types print their values as ASCII characters rather than as integer values.

Using type aliases to make complex types simple

Although we have only dealt with simple data types so far, in advanced C++, types can be complicated and lengthy to manually enter on your keyboard. For example, you might see a function and variable defined like this:

```cpp
#include <string> // for std::string
#include <vector> // for std::vector
#include <utility> // for std::pair

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

int main()
{
     std::vector<std::pair<std::string, int> > pairlist;

     return 0;
}
```

Typing `std::vector<std::pair<std::string, int> >` everywhere you need to use that type is cumbersome, and it is easy to make a typing mistake. It's much easier to use a type alias:

```cpp
#include <string> // for std::string
#include <vector> // for std::vector
#include <utility> // for std::pair

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

bool hasDuplicates(pairlist_t pairlist) // use pairlist_t in a function parameter
{
    // some code here
    return false;
}

int main()
{
     pairlist_t pairlist; // instantiate a pairlist_t variable

     return 0;
}
```

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 your code easier to read and saves typing.

This is probably the best use for type aliases.

Using type aliases for legibility

Type aliases can also help with code documentation and comprehension.

With variables, we have the variable's identifier to help document the purpose of the variable. But consider the case of a function's return value. Data types 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, given 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! The return type of `int` does not tell us much. If we're lucky, documentation for the function exists somewhere that we can reference. If we're unlucky, we have to read the code and infer the purpose.

Now let's do an equivalent version using a type alias:

```
using testScore_t =
int;
testScore_t
gradeTest();
```

The return type of `testScore_t` makes it a little more obvious that the function is returning a type that represents a test score.

In our experience, creating a type alias just to document the return type of a single function isn't worth it (use a comment instead). But if you have already created a type alias for other reasons, this can be a nice additional benefit.

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 objects of type `short` were being used to hold ID numbers and which were being used for other purposes.

However, if you use type aliases, then changing types becomes as simple as updating the type alias (e.g. from `using studentID_t = short;` to `using studentID_t = long;`).

While this seems like a nice benefit, caution is necessary whenever a type is changed, as the behavior of the program may also change. This is especially true 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. If you change an existing type to some other type, your code should be thoroughly retested.

Downsides and conclusion

While type aliases offer some benefits, they also introduce yet another identifier into your code that needs to be understood. If this isn't offset by some benefit to readability or comprehension, then the type alias is doing more harm than good.

A poorly utilized type alias can take a familiar type (such as `std::string`) and hide it behind a custom name that needs to be looked up. In some cases (such as with smart pointers, which we'll cover in a future chapter), obscuring the type information can also be harmful to understanding how the type should be expected to work.

For this reason, type aliases should be used primarily in cases where there is a clear benefit to code readability or code maintenance. This is as much of an art as a science. Type aliases are most useful when they can be used in many places throughout your code, rather than in fewer places.

Best practice

Use type aliases judiciously, when they provide a clear benefit to code readability or code maintenance.

Quiz time

Question #1

Given the following function prototype:

```
1 | 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

Next lesson

8.7 Type deduction for objects using the auto keyword

Back to table of contents

Previous lesson

8.5 Explicit type conversion (casting) and static_cast

Notify me about replies: POST COMMENT

DP N N FOUT

Newest