10.7 — Typedefs and type aliases

Type aliases

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

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

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

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

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

```
#include <iostream>

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

    Distance 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` as an alias for type `double`.

Next, we define a variable named `milesToDestination` of type alias `Distance`. Because the compiler knows `Distance` is a type 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.

For advanced readers

Type aliases can also be templated. We cover this in lesson 13.14 -- Class template argument deduction (CTAD) and deduction guides.

Naming type aliases

Historically, there hasn't been a lot of consistency in how type aliases have been named. There are three common naming conventions (and you will run across all of them):

> Type aliases that end in a "_t" suffix (the "_t" is short for "type"). This convention is often used by the standard library for globally scoped type names (like `size_t` and `nullptr_t`).

This convention was inherited from C, and used to be the most popular when defining your own type aliases (and sometimes other types), but has fallen out of favor in modern C++. Note that POSIX reserves the "_t" suffix for globally scoped type names, so using this convention may cause type naming conflicts on POSIX systems.

> Type aliases that end in a "_type" suffix. This convention is used by some standard library types (like `std::string`) to name nested type aliases (e.g. `std::string::size_type`).

But many such nested type aliases do not use a suffix at all (e.g. `std::string::iterator`), so this usage is inconsistent at best.

> Type aliases that use no suffix.

In modern C++, the convention is to name type aliases (or any other type) that you define yourself starting with a capital letter, and using no suffix. The capital letter helps differentiate the names of types from the names of variables and functions (which start with a lower case letter), and prevents naming collisions between them.

When using this naming convention, it is common to see this usage:

```
void printDistance(Distance distance); // Distance is some defined type
```

In this case, `Distance` is the type, and `distance` is the parameter name. C++ is case-sensitive, so this is fine.

Best practice

Name your type aliases starting with a capital letter and do not use a suffix (unless you have a specific reason to do otherwise).

Author's note

Some future lessons in this tutorial series still use the "_t" or "_type" suffix. Please feel free to leave a comment on those lessons so we can make them consistent with best practices.

Type aliases are not distinct types

An alias does not actually define a new, distinct type (one that is considered separate from other types) -- it just introduces a new identifier for an existing type. A type 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 = long; // define Miles as an alias for type long
    using Speed = long; // define Speed as an alias for type long

    Miles distance { 5 }; // distance is actually just a long
    Speed 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` and `Speed` to have distinct meanings, both are just aliases for type `long`. This effectively means `Miles`, `Speed`, and `long` can all be used interchangeably. And indeed, when we assign a value of type `Speed` to a variable of type `Miles`, 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 13.6 -- Scoped enumerations (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 global scope and is usable to the end of the file. In the above example, `Miles` and `Speed` are only usable in the `main()` function.

If you need to use one or more type aliases 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_H
#define MYTYPES_H

    using Miles = long;
    using Speed = long;

#endif
```

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

Typedefs

A **typedef** (which is short for “type definition”) is an older way of creating an alias for a type. To create a typedef alias, we use the `typedef` keyword:

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

Typedefs are still in C++ for backwards compatibility reasons, but they have been largely replaced by type aliases in modern C++.

Typedefs have a few syntactical issues. First, it’s easy to forget whether the name of the typedef or the name of the type to alias comes first. Which is correct?

```
typedef Distance double; // incorrect (typedef name first)
typedef double Distance; // 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:

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

In the above typedef definition, the name of the new type (`FcnType`) is buried in the middle of the definition, whereas in the type alias, the name of the new type and the rest of the definition are separated by an equals sign.

Third, the name “typedef” suggests that a new type is being defined, but that’s not true. A typedef is just an alias.

Best practice

Prefer type aliases over typedefs.

Nomenclature

The C++ standard uses the term “typedef names” for the names of both typedefs and type aliases.

In conventional language, the term “typedef” is often used to mean “either a typedef or a type alias” since they effectively do the same thing.

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 primary uses for type aliases is 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 (as a compiler/preprocessor setting), 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, as long as `INT_2_BYTES` is #defined correctly, `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 integer types (such as `std::int16_t` and `std::uint32_t`) and the `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::int8_t x{ 97 }; // int8_t is usually a typedef for signed char
    std::cout << x << '\n';

    return 0;
}
```

This program prints:

```
a
```

Because `std::int8_t` is typically a typedef for `signed char`, variable `x` will likely be defined as a `signed char`. And char types print their values as ASCII characters rather than as integer values.

Using type aliases to make complex types easier to read

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:

```
#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:

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

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

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

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

     return 0;
}
```

Much better! Now we only have to type `VectPairSI` 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 simpler name, which makes your code easier to read and saves typing.

This is probably the best use for type aliases.

Using type aliases to document the meaning of a value

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` describe what *type* of value a function returns, but more often we want to know what the *meaning* of a return value is.

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 = int;
TestScore gradeTest();
```

The return type of `TestScore` 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 multiple functions passing or returning such a type, creating a type alias might be worthwhile.

Using type aliases for easier code maintenance

Type aliases also allow you to change the underlying type of an object without having to update lots of hardcoded types. 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 = short;` to `using StudentId = 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 a signed to unsigned value)! 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:

```
int printData();
```

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

Show Solution