13.1 — Introduction to program-defined (user-defined) types

learncpp.com/cpp-tutorial/introduction-to-program-defined-user-defined-types/

Because fundamental types are defined as part of the core C++ language, they are available for immediate use. For example, if we want to define a variable with a type of `int` or `double`, we can just do so:

```
int x; // define variable of fundamental type 'int'
double d; // define variable of fundamental type 'double'
```

This is also true for the compound types that are simple extensions of fundamental types (including functions, pointers, references, and arrays):

```
void fcn(int) {}; // define a function of type void()(int)
int* ptr; // define variable of compound type 'pointer to int'
int& ref { x }; // define variable of compound type 'reference to int' (initialized
with x)
int arr[5]; // define an array of 5 integers of type int[5] (we'll cover this in a
future chapter)
```

This works because the C++ language already knows what the type names (and symbols) for these types mean -- we do not need to provide or import any definitions.

However, consider the case of a type alias (introduced in lesson 10.7 -- Typedefs and type aliases), which allows us to define a new name for an existing type. Because a type alias introduces a new identifier into the program, a type alias must be defined before it can be used:

```
#include <iostream>

using length = int; // define a type alias with identifier 'length'

int main()
{
    length x { 5 }; // we can use 'length' here since we defined it above
    std::cout << x << '\n';

    return 0;
}
```

If we were to omit the definition of `length`, the compiler wouldn't know what a `length` is, and would complain when we try to define a variable using that type. The definition for `length` doesn't create an object -- it just tells the compiler what a `length` is so it can be used later.

What are user-defined / program-defined types?

Back in the introduction to the previous chapter (12.1 -- Introduction to compound data types), we introduced the challenge of wanting to store a fraction, which has a numerator and denominator that are conceptually linked together. In that lesson, we discussed some of the challenges with using two separate integers to store a fraction's numerator and denominator independently.

If C++ had a built-in fraction type, that would have been perfect -- but it doesn't. And there are hundreds of other potentially useful types that C++ doesn't include because it's just not possible to anticipate everything that someone might need (let alone implement and test those things).

Instead, C++ solves such problems in a different way: by allowing us to create entirely new, custom types for use in our programs! Such types are often called **user-defined types** (though we think the term **program-defined types** is better -- we'll discuss the difference later in this lesson).

C++ has two different categories of compound types that can be used to create program-defined types:

- Enumerated types (including unscoped and scoped enumerations)
- Class types (including structs, classes, and unions).

Defining program-defined types

Just like type aliases, program-defined types must also be defined before they can be used. The definition for a program-defined type is called a **type definition**.

Although we haven't covered what a struct is yet, here's an example showing the definition of custom Fraction type and an instantiation of an object using that type:

```
// Define a program-defined type named Fraction so the compiler understands what a
Fraction is
// (we'll explain what a struct is and how to use them later in this chapter)
// This only defines what a Fraction type looks like, it doesn't create one
struct Fraction
{
	int numerator {};
	int denominator {};
};

// Now we can make use of our Fraction type
int main()
{
	Fraction f { 3, 4 }; // this actually instantiates a Fraction object named f

	return 0;
}
```

In this example, we're using the `struct` keyword to define a new program-defined type named `Fraction` (in the global scope, so it can be used anywhere in the rest of the file). This doesn't allocate any memory -- it just tells the compiler what a `Fraction` looks like, so we can allocate objects of a `Fraction` type later. Then, inside `main()`, we instantiate (and initialize) a variable of type `Fraction` named `f`.

Program-defined type definitions must end in a semicolon. Failure to include the semicolon at the end of a type definition is a common programmer error, and one that can be hard to debug because the compiler may error on the line *after* the type definition.

Warning

Don't forget to end your type definitions with a semicolon.

We'll show more examples of defining and using program-defined types in the next lesson (13.2 -- Unscoped enumerations), and we cover structs starting in lesson 13.7 -- Introduction to structs, members, and member selection.

Naming program-defined types

By convention, program-defined types are named starting with a capital letter and don't use a suffix (e.g. `Fraction`, not `fraction`, `fraction_t`, or `Fraction_t`).

Best practice

Name your program-defined types starting with a capital letter and do not use a suffix.

New programmers sometimes find variable definitions such as the following confusing because of the similarity between the type name and variable name:

```
Fraction fraction {}; // Instantiates a variable named fraction of type Fraction
```

This is no different than any other variable definition: the type (`Fraction`) comes first (and because Fraction is capitalized, we know it's a program-defined type), then the variable name (`fraction`), and then an optional initializer. Because C++ is case-sensitive, there is no naming conflict here!

Using program-defined types throughout a multi-file program

Every code file that uses a program-defined type needs to see the full type definition before it is used. A forward declaration is not sufficient. This is required so that the compiler knows how much memory to allocate for objects of that type.

To propagate type definitions into the code files that need them, program-defined types are typically defined in header files, and then #included into any code file that requires that type definition. These header files are typically given the same name as the program-defined type (e.g. a program-defined type named Fraction would be defined in Fraction.h)

Best practice

A program-defined type used in only one code file should be defined in that code file as close to the first point of use as possible.

A program-defined type used in multiple code files should be defined in a header file with the same name as the program-defined type and then #included into each code file as needed.

Here's an example of what our Fraction type would look like if we moved it to a header file (named Fraction.h) so that it could be included into multiple code files:

Fraction.h:

```
#ifndef FRACTION_H
#define FRACTION_H

// Define a new type named Fraction
// This only defines what a Fraction looks like, it doesn't create one
// Note that this is a full definition, not a forward declaration
struct Fraction
{
        int numerator {};
        int denominator {};
};

#endif
```

Fraction.cpp:

```
#include "Fraction.h" // include our Fraction definition in this code file

// Now we can make use of our Fraction type
int main()
{
	Fraction f{ 3, 4 }; // this actually creates a Fraction object named f

	return 0;
}
```

Type definitions are partially exempt from the one-definition rule (ODR)

In lesson 2.7 -- Forward declarations and definitions, we discussed how the one-definition rule requires that each function and global variable only have one definition per program. To use a given function or global variable in a file that does not contain the definition, we need a forward declaration (which we typically propagate via a header file). This works because declarations are enough to satisfy the compiler when it comes to functions and non-constexpr variables, and the linker can then connect everything up.

However, using forward declarations in a similar manner doesn't work for types, because the compiler typically needs to see the full definition to use a given type. We must be able to propagate the full type definition to each code file that needs it.

To allow for this, types are partially exempt from the one-definition rule: a given type is allowed to be defined in multiple code files.

You've already exercised this capability (likely without realizing it): if your program has two code files that both `#include <iostream>`, you're importing all of the input/output type definitions into both files.

There are two caveats that are worth knowing about. First, you can still only have one type definition per code file (this usually isn't a problem since header guards will prevent this). Second, all of the type definitions for a given type must be identical, otherwise undefined behavior will result.

Nomenclature: user-defined types vs program-defined types

The term "user-defined type" sometimes comes up in casual conversation, as well as being mentioned (but not defined) in the C++ language standard. In casual conversation, the term tends to mean "a type defined within your own programs" (such as the Fraction type example above).

The C++ language standard uses the term "user-defined type" in a non-conventional manner. In the language standard, a "user-defined type" is any class type or enumerated type that is defined by you, the standard library, or the implementation (e.g. types defined by the compiler to support language extensions). Perhaps counter-intuitively, this means `std::string` (a class type defined in the standard library) is considered to be a user-defined type!

To provide additional differentiation, the C++20 language standard helpfully defines the term "program-defined type" to mean class types and enumerated types that are not defined as part of the standard library, implementation, or core language. In other words, "program-defined types" only include class types and enum types that are defined by us (or a third-party library).

Consequently, when talking only about class types and enum types that we're defining for use in our own programs, we'll prefer the term "program-defined", as it has a more precise definition.

Type	Meaning	Examples
Fundamental	A type built into the core C++ language	int, std::nullptr_t
Compound	A type built from fundamental types	int&, double*, std::string, Fraction
User-defined	A class type or enumerated type (Includes those defined in the standard library or implementation) (In casual use, typically used to mean program-defined types)	std::string, Fraction
Program-defined	A class type or enumerated type (Excludes those defined in standard library or implementation)	Fraction