13.7 — Introduction to structs, members, and member selection

There are many instances in programming where we need more than one variable in order to represent something of interest. As we discussed in the introduction to the previous chapter (12.1 -- Introduction to compound data types), a fraction has a numerator and denominator that are linked together into a single mathematical object.

Alternatively, lets say we want to write a program where we need to store information about the employees in a company. We might be interested in keeping track of attributes such as the employee's name, title, age, employee id, manager id, wage, birthday, hire date, etc…

If we were to use independent variables to track all of this information, that might look something like this:

```
std::string name;
std::string title;
int age;
int id;
int managerId;
double wage;
int birthdayYear;
int birthdayMonth;
int birthdayDay;
int hireYear;
int hireMonth;
int hireDay;
```

However, there are a number of problems with this approach. First, it's not immediately clear whether these variables are actually related or not (you'd have to read comments, or see how they are used in context). Second, there are now 12 variables to manage. If we wanted to pass this employee to a function, we'd have to pass 12 arguments (and in the correct order), which would make a mess of our function prototypes and function calls. And since a function can only return a single value, how would a function even return an employee?

And if we wanted more than one employee, we'd need to define 12 more variables for each additional employee (each of which would require a unique name)! This clearly doesn't scale at all. What we really need is some way to organize all of these related pieces of data together, to make them easier to manage.

Fortunately, C++ comes with two compound types designed to solve such challenges: structs (which we'll introduce now) and classes (which we'll explore soon). A **struct** (short for **structure**) is a program-defined data type (13.1 -- Introduction to program-defined (user-

defined) types) that allows us to bundle multiple variables together into a single type. As you'll see shortly, this makes management of related sets of variables much simpler!

Defining structs

Because structs are a program-defined type, we first have to tell the compiler what our struct type looks like before we can begin using it. Here is an example of a struct definition for a simplified employee:

```
struct Employee
{
    int id {};
    int age {};
    double wage {};
};
```

The `struct` keyword is used to tell the compiler that we're defining a struct, which we've named `Employee` (since program-defined types are typically given names starting with a capital letter).

Then, inside a pair of curly braces, we define the variables that each Employee object will contain. In this example, each `Employee` we create will have 3 variables: an `int id`, an `int age`, and a `double wage`. The variables that are part of the struct are called **data members** (or **member variables**).

Tip

In everyday language, a *member* is a individual who belongs to a group. For example, you might be a member of the basketball team, and your sister might be a member of the choir.

In C++, a **member** is a variable, function, or type that belongs to a struct (or class). All members must be declared within the struct (or class) definition.

We'll use the term *member* a lot in future lessons, so make sure you remember what it means.

Just like we use an empty set of curly braces to value initialize (1.4 -- Variable assignment and initialization) normal variables, the empty curly braces after each member variable ensures that the member variables inside our `Employee` are value initialized when an `Employee` is created. We'll talk more about this when we cover default member initialization in a few lessons (13.9 -- Default member initialization).

Finally, we end the type definition with a semicolon.

As a reminder, `Employee` is just a type definition -- no objects are actually created at this point.

Defining struct objects

In order to use the `Employee` type, we simply define a variable of type `Employee`:

```
Employee joe {}; // Employee is the type, joe is the variable name
```

This defines a variable of type `Employee` named `joe`. When the code is executed, an Employee object is instantiated that contains the 3 data members. The empty braces ensures our object is value-initialized.

Just like any other type, it is possible to define multiple variables of the same struct type:

```
Employee joe {}; // create an Employee struct for Joe
Employee frank {}; // create an Employee struct for Frank
```

Accessing members

Consider the following example:

```
struct Employee
{
    int id {};
    int age {};
    double wage {};
};

int main()
{
    Employee joe {};

    return 0;
}
```

In the above example, the name `joe` refers to the entire struct object (which contains the member variables). To access a specific member variable, we use the **member selection operator** (`operator.`) in between the struct variable name and the member name. For example, to access Joe's age member, we'd use `joe.age`.

Struct member variables work just like normal variables, so it is possible to do normal operations on them, including assignment, arithmetic, comparison, etc…

```
#include <iostream>

struct Employee
{
    int id {};
    int age {};
    double wage {};
};

int main()
{
    Employee joe {};

    joe.age = 32;  // use member selection operator (.) to select the age member of
variable joe

    std::cout << joe.age << '\n'; // print joe's age

    return 0;
}
```

This prints:

```
32
```

One of the biggest advantages of structs is that we only need to create one new name per struct variable (the member names are fixed as part of the struct type definition). In the following example, we instantiate two `Employee` objects: `joe` and `frank`.

```
#include <iostream>

struct Employee
{
    int id {};
    int age {};
    double wage {};
};

int main()
{
    Employee joe {};
    joe.id = 14;
    joe.age = 32;
    joe.wage = 60000.0;

    Employee frank {};
    frank.id = 15;
    frank.age = 28;
    frank.wage = 45000.0;

    int totalAge { joe.age + frank.age };

    if (joe.wage > frank.wage)
        std::cout << "Joe makes more than Frank\n";
    else if (joe.wage < frank.wage)
        std::cout << "Joe makes less than Frank\n";
    else
        std::cout << "Joe and Frank make the same amount\n";

    // Frank got a promotion
    frank.wage += 5000.0;

    // Today is Joe's birthday
    ++joe.age; // use pre-increment to increment Joe's age by 1

    return 0;
}
```

In the above example, it is very easy to tell which member variables belong to Joe and which belong to Frank. This provides a much higher level of organization than individual variables would. Furthermore, because Joe’s and Frank’s members have the same names, this provides consistency when you have multiple variables of the same struct type.

We’ll continue our exploration of structs in the next lesson, including a look at how to initialize them.