4.7 — Structs

BY ALEX ON JUNE 20TH, 2007 | LAST MODIFIED BY ALEX ON OCTOBER 25TH, 2017

There are many instances in programming where we need more than one variable in order to represent an object. For example, to represent yourself, you might want to store your name, your birthday, your height, your weight, or any other number of characteristics about yourself. You could do so like this:

```
std::string myName;
int myBirthYear;
int myBirthMonth;
int myBirthDay;
int myHeightInches;
int myWeightPounds;
```

However, you now have 6 independent variables that are not grouped in any way. If you wanted to pass information about yourself to a function, you'd have to pass each variable individually. Furthermore, if you wanted to store information about someone else, you'd have to declare 6 more variables for each additional person! As you can see, this can quickly get out of control.

Fortunately, C++ allows us to create our own user-defined aggregate data types. An **aggregate data type** is a data type that groups multiple individual variables together. One of the simplest aggregate data types is the struct. A **struct** (short for structure) allows us to group variables of mixed data types together into a single unit.

Declaring and defining structs

Because structs are user-defined, we first have to tell the compiler what our struct looks like before we can begin using it. To do this, we declare our struct using the *struct* keyword. Here is an example of a struct declaration:

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

This tells the compiler that we are defining a struct named Employee. The Employee struct contains 3 variables inside of it: a short named id, an int named age, and a double named wage. These variables that are part of the struct are called **members** (or fields). Keep in mind that Employee is just a declaration -- even though we are telling the compiler that the struct will have member variables, no memory is allocated at this time. By convention, struct names start with a capital letter to distinguish them from variable names.

Warning: One of the easiest mistakes to make in C++ is to forget the semicolon at the end of a struct declaration. This will cause a compiler error on the *next* line of code. Modern compilers like Visual Studio 2010 will give you an indication that you may have forgotten a semicolon, but older or less sophisticated compilers may not, which can make the actual error hard to find.

In order to use the Employee struct, we simply declare a variable of type Employee:

```
Employee joe; // struct Employee is capitalized, variable joe is not
```

This defines a variable of type Employee named joe. As with normal variables, defining a struct variable allocates memory for that variable.

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 struct members

When we define a variable such as `Employee joe`, joe refers to the entire struct (which contains the member variables). In order to access the individual members, we use the **member selection operator** (which is a period). Here is an example of using the member selection operator to initialize each member variable:

```
Employee joe; // create an Employee struct for Joe
```

```
joe.id = 14; // assign a value to member id within struct joe
joe.age = 32; // assign a value to member age within struct joe
joe.wage = 24.15; // assign a value to member wage within struct joe

Employee frank; // create an Employee struct for Frank
frank.id = 15; // assign a value to member id within struct frank
frank.age = 28; // assign a value to member age within struct frank
frank.wage = 18.27; // assign a value to member wage within struct frank
```

As with normal variables, struct member variables are not initialized, and will typically contain junk. We must initialize them manually.

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 across multiple variables of the same struct type.

Struct member variables act just like normal variables, so it is possible to do normal operations on them:

```
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 += 2.50;

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

Initializing structs

Initializing structs by assigning values member by member is a little cumbersome, so C++ supports a faster way to initialize structs using an **initializer list**. This allows you to initialize some or all the members of a struct at declaration time.

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

Employee joe = { 1, 32, 60000.0 }; // joe.id = 1, joe.age = 32, joe.wage = 60000.0
Employee frank = { 2, 28 }; // frank.id = 2, frank.age = 28, frank.wage = 0.0 (default initialization)
```

In C++11, we can also use uniform initialization:

```
Employee joe { 1, 32, 60000.0 }; // joe.id = 1, joe.age = 32, joe.wage = 60000.0
Employee frank { 2, 28 }; // frank.id = 2, frank.age = 28, frank.wage = 0.0 (default initialization)
```

If the initializer list does not contain an initializer for some elements, those elements are initialized to a default value (that generally corresponds to the zero state for that type). In the above example, we see that frank.wage gets default initialized to 0.0 because we did not specify an explicit initialization value for it.

C++11/14: Non-static member initialization

Starting with C++11, it's possible to give non-static (normal) struct members a default value:

```
struct Rectangle
{
    double length = 1.0;
    double width = 1.0;
};

int main()
```

```
{
    Rectangle x; // length = 1.0, width = 1.0

    x.length = 2.0; // you can assign other values like normal

    return 0;
}
```

Unfortunately, in C++11, the non-static member initialization syntax is incompatible with the initializer list and uniform initialization syntax. For example, in C++11, the following program won't compile:

```
struct Rectangle
{
    double length = 1.0; // non-static member initialization
    double width = 1.0;
};

int main()
{
    Rectangle x{ 2.0, 2.0 }; // uniform initialization

    return 0;
}
```

Consequently, in C++11, you'll have to decide whether you want to use non-static member initialization or uniform initialization. Uniform initialization is more flexible, so we recommend sticking with that one.

However, in C++14, this restriction was lifted and both can be used. If both are provided, the initializer list/uniform initialization syntax takes precedence. In the above example, Rectangle x would be initialized with length and width 2.0. In C++14, using both should be preferred, as it allows you to declare a struct with or without initialization parameters and ensure the members are initialized.

We talk about what static members are in chapter 8. For now, don't worry about them.

Assigning to structs

Prior to C++11, if we wanted to assign values to the members of structs, we had to do so individually:

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

Employee joe;
joe.id = 1;
joe.age = 32;
joe.wage = 60000.0;
```

This is a pain, particularly for structs with many members. In C++11, you can now assign values to structs members using an initializer list:

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

Employee joe;
joe = { 1, 32, 60000.0 }; // C++11 only
```

Structs and functions

A big advantage of using structs over individual variables is that we can pass the entire struct to a function that needs to work with the members:

```cpp
#include <iostream>

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

void printInformation(Employee employee)
{
    std::cout << "ID:   " << employee.id << "\n";
    std::cout << "Age:  " << employee.age << "\n";
    std::cout << "Wage: " << employee.wage << "\n";
}

int main()
{
    Employee joe = { 14, 32, 24.15 };
    Employee frank = { 15, 28, 18.27 };

    // Print Joe's information
    printInformation(joe);

    std::cout << "\n";

    // Print Frank's information
    printInformation(frank);

    return 0;
}
```

In the above example, we pass an entire Employee struct to printInformation(). This prevents us from having to pass each variable individually. Furthermore, if we ever decide to add new members to our Employee struct, we will not have to change the function declaration or function call!

The above program outputs:

```
ID:   14
Age:  32
Wage: 24.15

ID:   15
Age:  28
Wage: 18.27
```

A function can also return a struct, which is one of the few ways to have a function return multiple variables.

```cpp
#include <iostream>

struct Point3d
{
    double x;
    double y;
    double z;
};

Point3d getZeroPoint()
{
    Point3d temp = { 0.0, 0.0, 0.0 };
    return temp;
}

int main()
{
    Point3d zero = getZeroPoint();

    if (zero.x == 0.0 && zero.y == 0.0 && zero.z == 0.0)
        std::cout << "The point is zero\n";
    else
        std::cout << "The point is not zero\n";

    return 0;
}
```

This prints:

```
The point is zero
```

Nested structs

Structs can contain other structs. For example:

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

struct Company
{
    Employee CEO; // Employee is a struct within the Company struct
    int numberOfEmployees;
};

Company myCompany;
```

In this case, if we wanted to know what the CEO's salary was, we simply use the member selection operator twice: `myCompany.CEO.wage;`

This selects the CEO member from myCompany, and then selects the wage member from within CEO.

You can use nested initializer lists for nested structs:

```
struct Employee
{
    short id;
    int age;
    float wage;
};

struct Company
{
    Employee CEO; // Employee is a struct within the Company struct
    int numberOfEmployees;
};

Company myCompany = {{ 1, 42, 60000.0f }, 5 };
```

Struct size and data structure alignment

Typically, the size of a struct is the sum of the size of all its members, but not always!

Consider the Employee struct. On many platforms, a short is 2 bytes, an int is 4 bytes, and a double is 8 bytes, so we'd expect Employee to be 2 + 4 + 8 = 14 bytes. To find out the exact size of Employee, we can use the sizeof operator:

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

int main()
{
    std::cout << "The size of Employee is " << sizeof(Employee) << "\n";

    return 0;
}
```

On the author’s machine, this prints:

```
The size of Employee is 16
```

It turns out, we can only say that the size of a struct will be *at least* as large as the size of all the variables it contains. But it could be larger! For performance reasons, the compiler will sometimes add gaps into structures (this is called **padding**).

In the Employee struct above, the compiler is invisibly adding 2 bytes of padding after member id, making the size of the structure 16 bytes instead of 14. The reason it does this is beyond the scope of this tutorial, but readers who want to learn more can read about **data structure alignment** on Wikipedia. This is optional reading and not required to understand structures or C++!

Accessing structs across multiple files

Because struct declarations do not take any memory, if you want to share a struct declaration across multiple files (so you can instantiate variables of that struct type in multiple files), put the struct declaration in a header file, and #include that header file anywhere you need it.

Struct variables are subject to the same rules as normal variables. Consequently, to make a struct variable accessible across multiple files, you can use the extern keyword to do so.

Final notes on structs

Structs are *very* important in C++, as understanding structs is the first major step towards object-oriented programming! Later on in these tutorials, you’ll learn about another aggregate data type called a class, which is built on top of structs. Understanding structs well will help make the transition to classes that much easier.

The structs introduced in this lesson are sometimes called **plain old data structs** (or POD structs) since the members are all data (variable) members. In the future (when we discuss classes) we’ll talk about other kinds of members.

Quiz

1) You are running a website, and you are trying to keep track of how much money you make per day from advertising. Declare an advertising struct that keeps track of how many ads you’ve shown to readers, what percentage of ads were clicked on by users, and how much you earned on average from each ad that was clicked. Read in values for each of these fields from the user. Pass the advertising struct to a function that prints each of the values, and then calculates how much you made for that day (multiply all 3 fields together).

2) Create a struct to hold a fraction. The struct should have an integer numerator and an integer denominator member. Declare 2 fraction variables and read them in from the user. Write a function called multiply that takes both fractions, multiplies them together, and prints the result out as a decimal number. You do not need to reduce the fraction to its lowest terms.

Quiz Answers

1) **Hide Solution**

```
#include <iostream>

// First we need to define our Advertising struct
struct Advertising
{
    int adsShown;
    double clickThroughRatePercentage;
    double averageEarningsPerClick;
};

```

```cpp
void printAdvertising(Advertising ad)
{
    std::cout << "Number of ads shown: " << ad.adsShown << '\n';
    std::cout << "Click through rate: " << ad.clickThroughRatePercentage << '\n';
    std::cout << "Average earnings per click: $" << ad.averageEarningsPerClick << '\n';

    // The following line is split up to reduce the length
    // We need to divide ad.clickThroughRatePercentage by 100 because it's a percent of 100, not a mult
iplier
    std::cout << "Total Earnings: $" <<
        (ad.adsShown * ad.clickThroughRatePercentage / 100 * ad.averageEarningsPerClick) << '\n';
}

int main()
{
    // Declare an Advertising struct variable
    Advertising ad;

    std::cout << "How many ads were shown today? ";
    std::cin >> ad.adsShown;
    std::cout << "What percentage of users clicked on the ads? ";
    std::cin >> ad.clickThroughRatePercentage;
    std::cout << "What was the average earnings per click? ";
    std::cin >> ad.averageEarningsPerClick;

    printAdvertising(ad);
    return 0;
}
```

2) **Hide Solution**

```cpp
#include <iostream>

struct Fraction
{
    int numerator;
    int denominator;
};

void multiply(Fraction f1, Fraction f2)
{
    // Don't forget the static cast, otherwise the compiler will do integer division!
    std::cout << static_cast<float>(f1.numerator * f2.numerator) /
        (f1.denominator* f2.denominator);
}

int main()
{
    // Allocate our first fraction
    Fraction f1;
    std::cout << "Input the first numerator: ";
    std::cin >> f1.numerator;
    std::cout << "Input the first denominator: ";
    std::cin >> f1.denominator;

    // Allocate our second fraction
    Fraction f2;
    std::cout << "Input the second numerator: ";
    std::cin >> f2.numerator;
    std::cout << "Input the second denominator: ";
    std::cin >> f2.denominator;

    multiply(f1, f2);

    return 0;
}
```