13.10 — Passing and returning structs

learncpp.com/cpp-tutorial/passing-and-returning-structs/

Consider an employee represented by 3 loose variables:

```
int main()
{
    int id { 1 };
    int age { 24 };
    double wage { 52400.0 };

    return 0;
}
```

If we want to pass this employee to a function, we have to pass three variables:

```
#include <iostream>

void printEmployee(int id, int age, double wage)
{
    std::cout << "ID:   " << id << '\n';
    std::cout << "Age:  " << age << '\n';
    std::cout << "Wage: " << wage << '\n';
}

int main()
{
    int id { 1 };
    int age { 24 };
    double wage { 52400.0 };

    printEmployee(id, age, wage);

    return 0;
}
```

While passing 3 individual employee variables isn't that bad, consider a function where we need to pass 10 or 12 employee variables. Passing each variable independently would be time consuming and error prone. Additionally, if we ever add a new attribute to our employee (e.g. name), we now have to modify all the functions declarations, definitions, and function calls to accept the new parameter and argument!

Passing structs (by reference)

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. Structs are generally passed by reference (typically by const reference) to avoid making copies.

```
#include <iostream>

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

void printEmployee(const Employee& employee) // note pass by reference here
{
    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
    printEmployee(joe);

    std::cout << '\n';

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

    return 0;
}
```

In the above example, we pass an entire `Employee` to `printEmployee()` (twice, once for `joe` and once for `frank`).

The above program outputs:

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

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

Because we are passing the entire struct object (rather than individual members), we only need one parameter no matter how many members the struct object has. And, in the future, 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 new member will automatically be included.

Related content

We talk about when to pass structs by value vs reference in lesson 12.6 -- Pass by const lvalue reference.

Passing temporary structs

In the prior example, we created Employee variable `joe` prior to passing it to the `printEmployee()` function. This allows us to give the Employee variable a name, which can be useful for documentation purposes. But it also requires two statements (one to create `joe`, one to use `joe`).

In cases where we only use a variable once, having to give the variable a name and separate the creation and use of that variable can increase complexity. In such cases, it may be preferable to use a temporary object instead. A temporary object is not a variable, so it does not have an identifier.

Here's the same example as above, but we've replaced variables `joe` and `frank` with temporary objects:

```
#include <iostream>

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

void printEmployee(const Employee& employee) // note pass by reference here
{
    std::cout << "ID:   " << employee.id << '\n';
    std::cout << "Age:  " << employee.age << '\n';
    std::cout << "Wage: " << employee.wage << '\n';
}

int main()
{
    // Print Joe's information
    printEmployee(Employee { 14, 32, 24.15 }); // construct a temporary Employee to
pass to function (type explicitly specified) (preferred)

    std::cout << '\n';

    // Print Frank's information
    printEmployee({ 15, 28, 18.27 }); // construct a temporary Employee to pass to
function (type deduced from parameter)

    return 0;
}
```

We can create a temporary `Employee` in two ways. In the first call, we use the syntax `Employee { 14, 32, 24.15 }`. This tells the compiler to create an `Employee` object and initialize it with the provided initializers. This is the preferred syntax because it makes clear what kind of temporary object we are creating, and there is no way for the compiler to misinterpret our intentions.

In the second call, we use the syntax `{ 15, 28, 18.27 }`. The compiler is smart enough to understand that the provided arguments must be converted to an `Employee` so that the function call will succeed. Note that this form is considered an implicit conversion, so it will not work in cases where only explicit conversions are acceptable.

Related content

We talk more about class type temporary objects and conversions in lesson 14.13 -- Temporary class objects.

A few more things about temporary objects: they are created and initialized at the point of definition, and is destroyed at the end of the full expression in which it is created. And because creation of the temporary object is an rvalue expression, it can only be used in places where rvalues are accepted. When a temporary object is used as a function argument, it will only bind to parameters that accept rvalues. This includes pass by value and pass by const reference, and excludes pass by non-const reference and pass by address.

Returning structs

Consider the case where we have a function that needs to return a point in 3-dimensional Cartesian space. Such a point has 3 attributes: an x-coordinate, a y-coordinate, and a z-coordinate. But functions can only return one value. So how do we return all 3 coordinates back the user?

One common way is to return a struct:

```
#include <iostream>

struct Point3d
{
    double x { 0.0 };
    double y { 0.0 };
    double z { 0.0 };
};

Point3d getZeroPoint()
{
    // We can create a variable and return the variable (we'll improve this below)
    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
```

Structs are usually returned by value, so as not to return a dangling reference.

In the `getZeroPoint()` function above, we create a new named object (`temp`) just so we could return it:

```
Point3d getZeroPoint()
{
    // We can create a variable and return the variable (we'll improve this below)
    Point3d temp { 0.0, 0.0, 0.0 };
    return temp;
}
```

The name of the object (`temp`) doesn't really provide any documentation value here.

We can make our function slightly better by returning a temporary (unnamed/anonymous) object instead:

```
Point3d getZeroPoint()
{
    return Point3d { 0.0, 0.0, 0.0 }; // return an unnamed Point3d
}
```

In this case, a temporary `Point3d` is constructed, copied back to the caller, and then destroyed at the end of the expression. Note how much cleaner this is (one line vs two, and no need to understand whether `temp` is used more than once).

Related content

We discuss anonymous objects in more detail in lesson 14.13 -- Temporary class objects.

Deducing the return type

In the case where the function has an explicit return type (e.g. `Point3d`), we can even omit the type in the return statement:

```
Point3d getZeroPoint()
{
    // We already specified the type at the function declaration
    // so we don't need to do so here again
    return { 0.0, 0.0, 0.0 }; // return an unnamed Point3d
}
```

This is considered to be an implicit conversion.

Also note that since in this case we're returning all zero values, we can use empty braces to return a value-initialized Point3d:

```
Point3d getZeroPoint()
{
    // We can use empty curly braces to value-initialize all members
    return {};
}
```

Structs are an important building block

While structs are useful in and of themselves, classes (which are the heart of C++ and object oriented programming) build directly on top of the concepts we've introduced here. Having a good understanding of structs (especially data members, member selection, and default member initialization) will make your transition to classes that much easier.

Quiz time

Question #1

You are running a website, and you are trying to calculate your advertising revenue. Write a program that allows you to enter 3 pieces of data:

- How many ads were watched.
- What percentage of users clicked on an ad.
- The average earnings per clicked ad.

Store those 3 values in a struct. Pass that struct to another function that prints each of the values. The print function should also print how much you made for that day (multiply the 3 fields together).

Show Hint

Show Solution

Question #2

Create a struct to hold a fraction. The struct should have an integer numerator and an integer denominator member.

Write a function to read in a Fraction from the user, and use it to read-in two fraction objects. Write another function to multiply two Fractions together and return the result as a Fraction (you don't need to reduce the fraction). Write another function that prints a fraction.

Your program's output should match the following:

```
Enter a value for the numerator: 1
Enter a value for the denominator: 2

Enter a value for the numerator: 3
Enter a value for the denominator: 4

Your fractions multiplied together: 3/8
```

When multiplying two fractions together, the resulting numerator is the product of the two numerators, and the resulting denominator is the product of the two denominators.

Show Solution

Question #3

In the solution to the prior quiz question, why does `getFraction()` return by value instead of by reference?

Show Solution