Learning to Program with Haiku

Lesson 4

Written by DarkWyrm

It would be incredibly hard to write anything useful if there weren't ways for our programs to make decisions or to speed up tedious processes. In this lesson we are going to examine one way to run code based on a condition and a way to repeat a set of instructions as many times as we want.

Conditional Execution: Running Certain Code Sometimes

Even in programming, nothing is ever certain. One way a program can handle special conditions is if-then logic, as in **if** I see a one-eyed, one-horned flying purple people eater coming my way, **then** I'm going the *other* way. Quickly.

A computer can do the same thing, such as **if** some variable equals 10, **then** set some other variable to false. If we were to write this in code, this is what it would look like:

```
if (someVariable == 10)
{
	someOtherVariable = false;
}
```

The condition for this statement is placed inside the parentheses. Notice that instead of using just one equals sign for checking if the two variables are equal, we have two. In C and C++ a single equals sign is used for assigning a value to a variable and two equals signs are for comparing two values. If you forget, the compiler will warn you. Your program will still compile, but you will be warned about this possible programming error. Following the condition in parentheses, we have a block of instructions inside a pair of curly braces that will be executed if the condition is true. If it is false, the block of instructions is skipped and program execution continues on afterward. We can also specify what to do if the condition is false with an `else` block.

```
// If the dragon breathes fire
if (dragonBreathesFire)
{
	// Stop and break out the marshmallows. Yum!
	SetSpeedMPH(0);
	EquipMarshmallows(true);
}
else
{
	// The dragon isn't of the fire-breathing variety, so get ready to fight.
	DrawSword();
	RaiseShield();
	myVelocityMPH += 4;
}
```

`If` statements can be used in many fancy ways. They can be chained together or even **nested,** that is, put inside one another, and if you want to Work Smarter, Not Harder, (a.k.a. be a lazy programmer), you can skip the curly braces if you have only a single instruction to execute. Observe:

```
// This is the shortcut way of using an if statement - there is only one
// instruction to possibly run.
if (someVariable == 10)
	someOtherVariable = false;
```

```
if (PoliceVisible())
{
	myStress++;

	// Here we have a couple of if statements placed inside each other.
	// Every time there is another level of nesting, we increase the
	// indenting one level, as well.
	if (GetSpeed() > GetSpeedLimit())
		if (PoliceLightsFlashing())
			SetSpeedMPH(0);
		else
			SetSpeedMPH(GetSpeedLimit());
	else
		myStress--;
}
```

Here we check to see if there are police visible. If so, then our stress (naturally) goes up a little. Then we check to see if our speed is over the limit. If it isn't, our stress goes back down to where it was before. If, for some reason, it is over the limit, we check to see if the police car's lights are flashing and either slow down or stop, as appropriate.

Increasing the indenting with each nesting level is a coding style choice with a purpose – it makes understanding the logic of your if-then statements and helps you to quickly ascertain which instructions belong to which `if` statement. There are two schools of thought on how to indent: a series of spaces or using the Tab key. For the purposes of these lessons, we'll use the Tab key – it does the same job and requires less typing. Again, work smarter, not harder.

To add one final trick to our bag, we can also use Boolean logical operators to modify or chain together conditions with AND, OR, or NOT and set precedence with parentheses. For those unfamiliar, **Boolean logic** is just a way of chaining together conditions, such as IF some-condition AND some-other-condition.

Boolean Value	Operator	Example	Means
AND	&&	`if (a == 1 && b == 2)`	if `a` is 1 and `b` is 2
OR	`||`	`if (a == 1 || b == 2)`	if `a` is 1 or `b` is 2
NOT	!	`if ( !(a == 1 || b == 2) )`	if the opposite of `a` equaling 1 and `b` equaling 2

Note that if you have more than one condition in an `if` statement, it may be a good idea to enclose each one in parentheses to make your code a little clearer unless the conditions are pretty short.

```
if ( (a == 1 && b == 2) || c )
	DoSomething();
```

This example translates as "Call `DoSomething()` if either `a` is one and `b` is 2 or the value of `c` is nonzero." Integers can be used for logic, such as in this example. Zero is treated as false and everything else is treated as true, so in this example the condition `c` will be true if `c` is something other than zero.

Now while all of this might seem like a lot to handle at once, what we have learned working with `if` statements is the basis for other ways of controlling program execution C++. In practice, it isn't very complicated.

Loops

A **loop** is a feature of C and C++ where the program repeatedly executes a set of instructions while a certain condition is true. C++ has several kinds of loops, but for today, we will look at just one: the `for` loop. Let's look at one and then examine the details.

```
#include <stdio.h>

int main(void)
{
	int number = 0;

	// This is our for loop
	for (int i = 1; i < 10; i++)
	{
		number += i;
		printf("At step %d, the number is now %d\n",i,number);
	}
}
```

When we run this program, this is what it prints:

```
At step 1, the number is now 1
At step 2, the number is now 3
At step 3, the number is now 6
At step 4, the number is now 10
At step 5, the number is now 15
At step 6, the number is now 21
At step 7, the number is now 28
At step 8, the number is now 36
At step 9, the number is now 45
```

Our `for` loop has two parts: the control section, which is inside the parentheses, and the block of repeated instructions. The control section for the `for` loop has three parts: the initialization, the loop condition, and the step value. Each part is separated by a semicolon.

The initialization sets a variable to the starting value for the loop. We can declare it in this part of the `for` loop or use one that has already been declared. It is common practice to declare our index variable in this part of the loop, and many times programmers will use the lowercase i (as in short for index) for the index variable in many simple loops.

The loop condition is an expression that must be true for the loop to repeat itself. In this case, the loop will repeat while `i` is less than 10. The step value is an expression that changes the index variable. Most of the time, like in this example, we just add one to the index, but it's possible to use any math operation we want – we could change this to `i += 2` if we wanted to increment `i` by twos. Like many other aspects of C and C++, there is a lot of flexibility in what is allowed when constructing a `for` loop, but we'll keep things simple for now.

Applying Concepts

We're going to try to take the concepts we've learned in this lesson and put them together for something more than a little useful. When you take a loan out on a car, it's nice to know what the payments will be, so let's put together a function which calculates the payment. The equation that is used for this is the following:

$$A = \frac{P * \frac{r}{12}}{1 - (1 + \frac{r}{12})^{-n}}$$

A = the payment amount
P = the initial principal
r = the loan's interest rate
n = the number of months

With as complicated as it looks, this is definitely a job for a dedicated function. Let's translate this mathematical mess into something a little easier to wrap our minds around.

```
#include <math.h>

float Payment(float principal, float rate, int months)
{
	float top, bottom;

	// This line calculates the top half of the right side
	top = principal * (rate / 12.0);

	// This line calculates the bottom half of the right side
	bottom = 1 - pow(1 + (rate / 12.0),-months);

	return (top / bottom);
}
```

Translating an equation into a function is a matter of breaking it down into more manageable pieces. We've done this by calculating the value of the two halves of the division separately and placing them into two different variables. This will make our code easier to both read and debug.

Note that we used 12.0 and not 12 to force the compiler to treat the 12's as floating point numbers and not integers. Whenever we do math with integers, the compiler will drop any fractional results, so, for example, 10 / 4 would equal 2, but 10.0 / 4.0 would return 2.5. We want to avoid rounding errors in this case, so we use 12.0 and not 12.

We could have done the whole thing in one go, but it would've been harder to read and a real head-scratcher to debug. It would look like this:

```
return (principal * rate / 12.0) / ( 1 - pow(1 + (rate / 12.0),-months) );
```

Yuck. While some expert coders might prefer this because it's more compact, having readable, maintainable code is *far* more important. Now that we have the function to calculate the payment, let's put it to good use by making it calculate the monthly payments for car loans that last one to five years.

```c
#include <stdio.h>
#include <math.h>

float Payment(float principal, float rate, int months)
{
	float top, bottom;

	top = principal * (rate / 12.0);
	bottom = 1 - pow(1 + (rate / 12.0),-months);

	return (top / bottom);
}


int main(void)
{
	float principal, rate;
	int months;

	principal = 10000.0;
	rate = .05;

	for (int months = 12; months <= 60; months += 12)
	{
		// Because whitespace doesn't matter to the compiler, it makes
		// sense to break up long lines of code into multiple smaller ones
		// to make your code more readable.
		printf("The monthly payment for a %d month, $%f car loan "
			"at %f%% is $%f\n", months, principal, rate * 100,
			Payment(principal,rate,months));
	}
}
```

It works! When it runs, it shows us that having a 1 year loan is really expensive, but a 4 or 5 year loan is much more manageable.

Going Further

Try playing with the numbers and see what happens, such as if the 60 in the loop condition is changed to 72 or if the principal is $20000 instead of $10000.

Bug Hunt

Hunt #1

Code

```
#include <stdio.h>

int main(void)
{
      // Print the times tables for the 2's family that has odd-numbered factors

      for (int i = 1; i < 13; i++)
            printf("2 x %d = %d\n",i,2 * i);
}
```

Errors

It's supposed to print only 2 times the odd numbers (1,3,5,etc.) but it prints all of them.

Hunt #2

Code

```
#include <stdio.h>

int main(void)
{
      float pi = 3.141592;
      printf("pi equals %d\n",pi);
      printf("2 * pi equals %d\n",pi * 2.0);
      return 0;
}
```

Errors

```
foo.cpp: In function ‘int main()’:
foo.cpp:6: warning: format ‘%d’ expects type ‘int’, but argument 2 has type
‘double’
foo.cpp:7: warning: format ‘%d’ expects type ‘int’, but argument 2 has type
‘double’
```

And when it runs, it prints this:

```
pi equals 0
2 * pi equals 0
```

Lesson 3 Bug Hunt Answers

1. The variable `c` was not declared. Add a line `int c = 5;` after the one for b and this will compile (and work) properly.
2. The include for math.h has been forgotten. Add `#include <math.h>` to the top of the example.

Lesson 3 Project Review

Our job was to use the equation *Interest = Principal * rate * time* to calculate and print the simple interest incurred on a principal of $20000 at a rate of 5% per month for 24 months and we had to create a function to do the actual interest calculations. Here's one way to do it:

```
#include <stdio.h>

float SimpleInterest(float principal, float rate, int months)
{
	return principal * rate * months;
}

int main(void)
{
	// Our $20000 principal
	float p = 20000.00;

	// 5% interest is .05
	float r = .05;

	int m = 24;

	float interest = SimpleInterest(p,r,m);
	printf("The interest on %f at %f%% interest for %d months is $%f\n",
			p,r * 100, m, interest);

}
```