Learning to Program with Haiku

Lesson 3

Written by DarkWyrm

So far we've been learning about programming basics, such as how to write a function and how to start looking for bugs in our code. This time we'll be learning about different types of data and ways to store and pass it around.

There is only so much that can be done with direct manipulation like `return 1 + 1;` which is why we have **variables** in programming. A variable is simply a storage container for information, and just like any real world container, they come in different shapes, sizes, and uses.

Creating a variable is merely a matter of declaring its existence, as seen below.

```c
#include <stdio.h>

int main(void)
{
	// Declaring one integer variable named a
	int a;

	// Declaring two at once: b and c
	int b, c;

	a = 1;
	b = 2;
	c = 3;

	// print out the values of our variables
	printf("a is %d, b is %d, and c is %d.\n",a,b,c);

	return a + b + c;
}
```

Variables can be declared one at a time, such as our variable `a`, or several at once, like `b` and `c`. In addition to declaring variables for our use in functions, many times we'll get them for free because many functions require input data to do their work. These variables are called **parameters** or **arguments**.

Function parameters are declared both in a function's declaration and definition. They are listed in a comma-separated list. Functions which do not take any arguments use the word `void` to say so.

```c
// Declaration of a function with two arguments.
int SomeFunction(int someNumber, int anotherNumber);

// Definition of a function with two arguments.
int MultiplyNumbers(int value, int secondValue)
{
	return value * secondValue;
}

// A function which needs no arguments. Must be a really peaceful one.
int main(void)
{
	return MultiplyNumbers(2,3);
}
```

When we call a function with parameters, such as `MultiplyNumbers()` in the above example, we don't list the type beside each one. The compiler keeps track of the types and warns us when we make mistakes.

Speaking of types, there are more types of data than just integers in C++. Each type takes up a different amount of space in memory, measured in bytes. This is important to remember because the number of bytes a variable occupies has a direct impact on how much information it can hold. This difference is evident in the range of values a `char` can hold as opposed to a `short`. Type sizes vary from platform to platform, but here is a pretty good list for Haiku on a 32-bit processor.

Type	**Size (bytes)**	**Range**	**Description**
char	1	-128 to 127	Letters – each char variable only holds 1 letter
unsigned char	1	0 to 255	Letters – each char variable only holds 1 letter
short	2	–32,768 to 32,767	Whole numbers
unsigned short	2	0 to 65, 535	Whole numbers
int	4	–2,147,483,648 to 2,147,483,647	Whole numbers
unsigned int	4	0 to 4,294,967,295	Whole numbers
long	4	–2,147,483,648 to 2,147,483,647	Whole numbers
long long	8	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	Whole numbers
unsigned long long	8	0 to 18,446,744,073,709,551,615	Whole numbers
float	4	3.4E +/- 38 (7 digits)	Numbers with a fractional part (floating point)
double	8	1.7E +/- 308 (15 digits)	Numbers with a fractional part (floating point)
long double	12		Numbers with a fractional part (floating point)
bool	1	True or false	True or false
wchar_t	2 or 4	As either `short` or `int`	“Wide” characters, which can support international characters.. As with `char`, each variable only holds 1 letter.

Using printf()

Now do you remember the weirdness that we saw a little bit ago with `printf()`? Let's take another look at it.

```c
#include <stdio.h>

int main(void)
{
	int a;
	int b, c;

	a = 1;
	b = 2;
	c = 3;

	printf("a is %d, b is %d, and c is %d.\n",a,b,c);

	return a + b + c;
}
```

`printf` is one of a few functions which take a variable number of arguments. Its first parameter is always a string. Depending on the number of placeholders in the string, though, it may or may not have additional parameters following the string. For example, in our above example, the string has three `%d` placeholders, one each for `a`, `b`, and `c`. Three placeholders, three "extra" parameters. Here are some other possible placeholders for `printf` that we'll use later on. Note that this is not an exhaustive list to `printf` – there are many more options, but these will suffice for now.

Placeholder	**Type**	**Sample output**
%c	Character	a
%d, %i	Signed decimal integer	234
%e, %E	Scientific notation using e/E	1.7e+5, 1.7E+5
%f	Floating point number	3.14
%g	Double precision number	3.14
%o	Integer in octal notation	711
%u	Unsigned integer	255
%x, %X	Integer in hexadecimal notation	0xff, 0xFF
%%	Percent sign	%

Operators

Operators give us ways of working with variables and numbers without calling functions. `+`, `-`, and `*` are all examples of operators, but C++ has many more than just these. Here are the arithmetic operators that we'll need for now.

Operator	**Operation**	**Description**
a + b	addition	adds b to a
a - b	subtraction	subtracts b from a
a * b	multiplication	multiplies a by b
a / b	division	divides a by b

Operator	Operation	Description
a % b	modulo	the remainder of a / b
a = b	assignment	sets a to the value of b
++a	pre-increment	adds 1 to a before the rest of the expression is evaluated
a++	post-increment	adds 1 to a after the rest of the expression is evaluated
--a	pre-decrement	subtracts 1 from a before the rest of the expression is evaluated
a--	post-decrement	subtracts 1 from a after the rest of the expression is evaluated
a += b	assign with addition	Short for a = a + b
a -= b	assign with subtraction	Short for a = a - b
a *= b	assign with multiplication	Short for a = a * b
a /= b	assign with division	Short for a = a / b
a %= b	assign with modulo	Short for a = a % b

The -- and ++ operators need a little more explanation than is possible in the table. Let's take a look at some code to explain it best.

```
#include <stdio.h>

int main(void)
{
	int a = 1;
	int b = 2;

	// The result here will be 3 because we add 1 to a
	// after a + b is calculated
	printf("a++ + b = %d\n",a++ + b);

	// Because we added 1 to a, this prints a 4.
	printf("a + b = %d\n",a + b);

	// This is 5 because the compiler will add 1 to a before calculating
	// a + b
	printf("++a + b = %d\n",++a + b);

	return 0;
}
```

Whew! We covered a lot of stuff in this lesson, but using all of it let's us do all sorts of fancy stuff. Let's put it to use.

```c
#include <stdio.h>

// math.h gives us access to a lot of mathematical functions. We're
// including it here so we can access sqrt(), which calculates
// square roots.
#include <math.h>

double hypotenuse(int a, int b)
{
	return sqrt((a*a) + (b*b));
}

int main(void)
{
	int a = 3;
	int b = 4;

	printf("For the triangle with legs %d and %d, the hypotenuse will be %g\n",
		a,b,hypotenuse(a,b));

	return 0;
}
```

`hypotenuse()` returns a `double` because we want some sort of precision beyond whole numbers. It is also the return type for `sqrt()`.

Bug Hunt

Hunt #1

Code

```c
int sum(int first, int second, int third)
{
	return first + second + third;
}

int main(void)
{
	int a = 3;
	int b = 4;

	printf("The sum is %d\n", sum(a,b,c));

	return 0;
}
```

Errors

```
foo.cpp: In function ‘int main()’:
foo.cpp:14: error: ‘c’ was not declared in this scope
```

Hunt #2

Code

```
#include <stdio.h>

double distance(int x1, int y1, int x2, int y2)
{
	int deltax = x2 - x1;
	int deltay = y2 - y1;

	return sqrt((deltax * deltax) + (deltay * deltay));
}

int main(void)
{
	int x1,y1,x2,y2;

	x1 = 3;
	y1 = 3;

	x2 = 8;
	y2 = 3;

	printf("The distance between (%d,%d) and (%d,%d) is %g\n", x1,y1, x2,y2,
			distance(x1,y1,x2,y2));

	return 0;
}
```

Errors

```
foo.cpp: In function ‘double distance(int, int, int, int)’:
foo.cpp:8: error: ‘sqrt’ was not declared in this scope
```

Project

Using the equation *Interest* = *Principal* * *rate* * *time*, calculate and print the simple interest incurred on a principal of $20000 at a rate of 5% per month for 24 months. Use a function to do the actual interest calculations.